THE BELL SYSTEM TECHNICAL JOURNAL
Vol. 57, No. 6, July-August 1978
Printed in U. S. A.

UNIX Time-Sharing System:

The Programmer's Workbench

By T. A. DOLOTTA, R. C. HAIGHT, and J. R. MASHEY
(Manuscript received December 5, 1977)

Many, if not most, UNIX systems are dedicated to specific projects and serve small, cohesive groups of (usually technically oriented) users. The Programmer's Workbench UNIX system (PWB/UNIX for short) is a facility based on the UNIX system that serves as a large, general-purpose, "utility" computing service. It provides a convenient working environment and a uniform set of programming tools to a very diverse group of users. The PWB/UNIX system has several interesting characteristics:*

(i) Many of its facilities were built in close cooperation between developers and users.
(ii) It has proven itself to be sufficiently reliable so that its users, who develop production software, have abandoned punched cards, private backup tapes, etc.
(iii) It offers a large number of simple, understandable program-development tools that can be combined in a variety of ways; users "package" these tools to create their own specialized environments.
(iv) Most importantly, the above were achieved without compromising the basic elegance, simplicity, generality, and ease of use of the UNIX system.

The result has been an environment that helps large numbers of users to get their work done, that improves their productivity, that adapts quickly to their individual needs, and that provides reliable service at a relatively low cost. This paper discusses some of the problems we encountered in building the PWB/UNIX system, how we solved them, how our system is used, and some of the lessons we learned in the process.

* UNIX is a trademark of Bell Laboratories.

I. INTRODUCTION

The Programmer's Workbench UNIX* system (hereafter called PWB/UNIX for brevity) is a specialized computing facility dedicated to supporting large software-development projects. It is a production system that has been used for several years in the Business Information Systems Programs (BISP) area of Bell Laboratories and that supports there a user community of about 1,100 people. It was developed mainly as an attempt to improve the quality, reliability, flexibility, and consistency of the programming environment. The concepts behind the PWB/UNIX system emphasize several ideas:

(*i*) Program development and execution of the resulting programs are two radically different functions. Much can be gained by assigning each function to a computer best suited to it. Thus, as much of the development as possible should be done on a computer dedicated to that task, i.e., one that acts as a "development facility" and provides a superior programming environment. Production running of the developed products very often occurs on another computer, called a "target" system. For some projects, a single system may successfully fill both roles, but this is rare, because most current operating systems were designed primarily for *running* programs, with little thought having been given to the requirements of the program-development process; we did the exact opposite of this in the PWB/UNIX system.

(*ii*) Although there may be several target systems (possibly supplied by different vendors), the development facility should present a single, uniform interface to its users. Current targets for the PWB/UNIX system include IBM System/370 and UNIVAC 1100-series computers; in some sense, the PWB/UNIX system is also a target, because it is built and maintained with its own tools.

(*iii*) A development facility can be implemented on computers of moderate size, even when the target machines consist of very large systems.

Although PWB/UNIX is a special-purpose system (in the same sense that a "front-end" computer is a special-purpose system), it is specialized for use by human beings. As shown in Fig. 1, it provides the interface between program developers and their target

* UNIX is a trademark of Bell Laboratories.

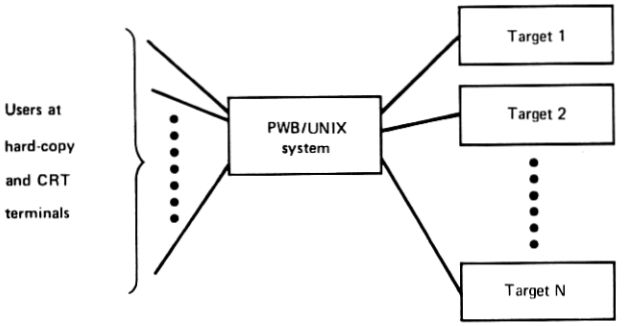

Fig. 1—PWB/UNIX™ interface with its users.

computer(s). Unlike a typical "front-end," the PWB/UNIX system supplies a separate, visible, uniform environment for program-development work.

II. CURRENT STATUS

The PWB/UNIX installation at BISP currently consists of a network of DEC PDP-11/45s and /70s running a modified version of the UNIX system.* By most measures, it is the largest known UNIX installation in the world. Table I gives a "snapshot" of it as of October 1977.

The systems are connected to each other so that each can be backed up by another, and so that files can be transmitted efficiently among systems. They are also connected by communications lines to the following target systems: two IBM 370/168s, two UNIVAC 1100-series systems, and one XDS Sigma 5. Of the card images processed by these targets, 90 to 95 percent are received from PWB/UNIX systems. Average figures for prime-shift connect time

Table I—PWB/UNIX™ hardware at BISP (10/77)

System name	CPU type	Memory (K-bytes)	Disk (M-bytes)	Dial-up ports	Login names
A	/45	256	160	15	153
B	/70	768	480	48	260
D	/70	512	320	48	361
E	/45	256	160	20	114
F	/70	768	320	48	262
G	/70	512	160	48	133
H	/70	512	320	48	139
Totals	—	3,328	1,920	275	1,422

* In order to avoid ambiguity, we use in this paper the expression "Research UNIX system" to refer to the UNIX system itself (Refs. 1 and 2).

(9 a.m. to 5 p.m., Monday through Friday) and total connect time per day are 1,342 and 1,794 hours, respectively. Because some login names are duplicated across systems, the figure of 1,422 is a bit misleading. The figure of 1,100 *distinct* login names is a better indicator of the size of the user community.

This installation offers fairly inexpensive time-sharing service to large numbers of users. An "average" PWB/UNIX user consumes 25 hours of prime-shift connect time per month, and uses 0.5 megabytes of active disk storage. Heavy use is made of the available resources. Typically, 90 percent of available disk space is in use, and between 75 and 80 percent of possible prime-time connect hours are consumed; during periods of heavy load, CPU usage occasionally exceeds 95 percent.

The PWB/UNIX system has been adopted outside of BISP, primarily for computer-center, program-development, and text-processing services. In addition to the original PWB/UNIX installation, there are currently about ten other installations within Bell Laboratories and six installations in other parts of the Bell System. A number of these installations utilize more than one CPU; thus, within the Bell System, there are over thirty PDP-11s running the PWB/UNIX system and there are plans for several more in the near future.* There is also a growing number of PWB/UNIX installations outside of the Bell System.

III. HISTORY

The concept underlying the PWB/UNIX system was suggested in mid-1973 and the first PDP-11/45 was installed late that year. This machine was used at first for our own education and experimentation, while early versions of various facilities were constructed. At first, ours was an experimental project that faced considerable indifference from a user community heavily oriented to large computer systems, working under difficult schedules, and a bit wary of what then seemed like a radical idea. However, as word about the system spread, demand for service grew rapidly, almost always outrunning the supply. Users consistently underestimated their requirements for service, because they kept discovering unexpected applications for PWB/UNIX facilities. In four years, the original PWB/UNIX installation has grown from a single PDP-11 serving 16

* The number of PDP-11's in the Bell System that operate under the PWB/UNIX system has doubled, on the average, every 11 months during the past 4½ years.

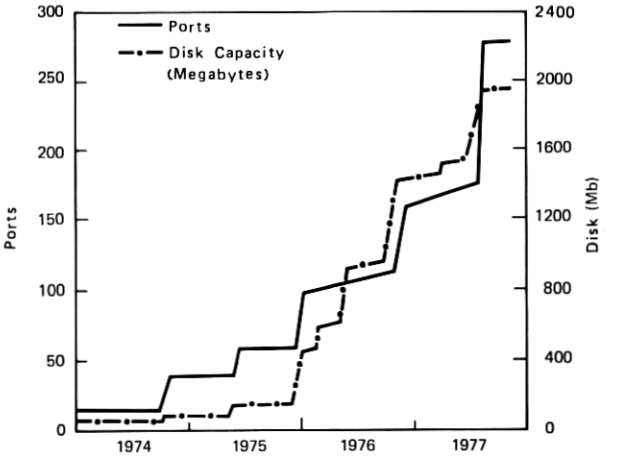

Fig. 2—Growth of PWB/UNIX™ at BISP—number of ports and disk capacity.

users to a network of seven PDP-11s serving 1,100 users. Figure 2 shows two other aspects of the growth of that installation; see Refs. 3 and 4 for "snapshots" of that installation earlier in its lifetime.

IV. MOTIVATION FOR THE PWB/UNIX APPROACH

The approach of using small computers to build a development facility for use with much larger targets has both good and bad points. At the outset, the following were thought to be potential problem areas:

(*i*) Cost of additional hardware.
(*ii*) Inconvenience of splitting data and functions among machines.
(*iii*) Use of incompatible character sets, i.e., ASCII and EBCDIC.
(*iv*) Limited size and speed of minicomputers, as compared to the speed and size of the target systems.
(*v*) Degradation of reliability caused by the increased complexity of the composite system.

Of these, only the last has required any significant, continuing effort; the main problem has been in maintaining reliable communications with the targets in the face of continually changing configurations of the targets, of the PWB/UNIX systems, and of the communications links themselves.

The approach embodied in the PWB/UNIX system offers significant advantages in the presence of certain conditions, all of which existed at the original PWB/UNIX installation, thus giving us a strong motivation for adopting this approach. We discuss these conditions below.

4.1 Gain by effective specialization

The computer requirements of software *developers* often diverge quite sharply from those of the *users* of that software. This observation seems especially applicable to software-development organizations such as BISP, i.e., organizations that develop large, data-base-oriented systems. Primary needs of developers include:

(*i*) Interactive computing services that are convenient, inexpensive, and continually available during normal working hours (where often the meaning of the expression "normal working hours" is "22 hours per day, 7 days per week").
(*ii*) A file structure designed for convenient interactive use; in particular, one that never requires the user to explicitly allocate or compact disk storage, or even to be aware of these activities.
(*iii*) Good, uniform tools for the manipulation of documents, source programs, and other forms of text. In our opinion, all the tasks that make up the program-development process and that are carried out by computers are nothing more than (sometimes very arcane) forms of text processing and text manipulation.
(*iv*) A command language simple enough for everyone to use, but one that offers enough programming capability to help automate the operational procedures used to track and control project development.
(*v*) Adaptability to frequent and unpredictable changes in location, structure, and personnel of user organizations.

On the other hand, users of the end products may have any or all of the following needs:

(*i*) Hardware of the appropriate size and speed to run the end products, possibly under stringent real-time or deadline constraints.
(*ii*) File structures and access methods that can be optimized to handle large amounts of data.
(*iii*) Transaction-oriented teleprocessing facilities.

(*iv*) The use of a specific type of computer and operating system, to meet any one of a number of possible (often externally imposed) requirements.

Few systems meet all the requirements of both developers and users. As a result, it is possible to make significant gains by providing two separate kinds of facilities and optimizing each to match one of two distinct sets of requirements.

4.2 Availability of better software

Time-sharing systems that run on large computers often retain significant vestiges of batch processing. Separation of support functions onto an appropriate minicomputer may offer an easy transition to more up-to-date software. Much of the stimulus for PWB/UNIX arose from the desire to make effective use of the UNIX system, whose facilities are extremely well matched to the developers' needs discussed above.

4.3 Installations with target systems from different vendors

It is desirable to have a uniform, target-independent set of tools to ease training and to permit the transfer of personnel between projects. File structures, command languages, and communications protocols differ widely among targets. Thus, it is expensive, if not impossible, to build a single set of effective and efficient tools that can be used on all targets. Effort is better expended in building a single good development facility.

4.4 Changing environments

Changes to hardware and software occur and cause problems even in single-vendor installations. Such changes may be disastrous if they affect both development and production environments at the same time. The problem is at least partially solved by using a separate development system. As an example, in the last few years, every BISP target system has undergone several major reconfigurations in both hardware and software, and the geographic work locations of most users have changed, in some cases more than once. The availability of the PWB/UNIX system often has been able to minimize the impact of these changes on the users.

4.5 Effective testing of terminal-oriented systems

It is difficult enough to test small batch programs; effective testing of large, interactive, data-base management applications is far more difficult. It is especially difficult to perform load testing when the same computer is both generating the load and running the program being tested. It is simpler and more realistic to perform such testing with the aid of a separate computer.

V. DESIGN APPROACH

In early 1974, much thought was given to what should be the overall design approach for the PWB/UNIX system. One proposal consisted of first designing it as a completely integrated facility, then implementing it, and finally obtaining users for it. A much different, less traditional approach was actually adopted; its elements were:

(*i*) Follow the UNIX system's philosophy of building small, independent tools rather than large, interrelated ones. Follow the UNIX system's approach of minimizing the number of different file formats.

(*ii*) Get users on the system quickly, work with them closely, and let their needs and problems drive the design.

(*iii*) Build software quickly, and expect to throw much of it away, or to have to adapt it to the users' real needs, as these needs become clear. In general, emphasize the ability to adapt to change, rather than try to build perfect products that are meant to last forever.

(*iv*) Make changes to the UNIX system only after much deliberation, and only when major gains can be made. Avoid changing the UNIX system's interfaces, and isolate any such changes as much as possible. Stay close to the Research UNIX system, in order to take advantage of continuing improvements.

This approach may appear chaotic, but, in practice, it has worked better than designing supposedly perfect systems that turn out to be obsolete or unusable by the time they are implemented. Unlike many other systems, the UNIX system both permits and encourages this approach.

VI. DIFFERENCES BETWEEN RESEARCH UNIX AND PWB/UNIX

The usage and operation of the PWB/UNIX system differ somewhat

from those of most UNIX systems within Bell Laboratories. Many of the changes and additions described below derive from these crucial differences.

A good many UNIX (as opposed to PWB/UNIX) systems are run as "friendly-user" systems, and are each used by a fairly small number of people who often work closely together. A large fraction of these users have read/write permissions for most (or all) of the files on the system, have permission to add commands to the public directories, are capable of "re-booting" the operating system, and even know how to repair damaged file systems.

The PWB/UNIX system, on the other hand, is most often found in a computer-center environment. Larger numbers of users are served, and they often represent different organizations. It is undesirable for everyone to have general read/write permissions. Although groups of users may wish to have sets of commands and files whose use they share, too many people must be served to permit everyone to add commands to public directories. Few users write C programs, and even fewer are interested in file-system internals. Machines are run by operators who are not expert system programmers. Many users have to to deal with large quantities of existing source code for target computers. Many must integrate their use of the PWB/UNIX system into existing procedures and working methods.

Notwithstanding all the above problems, we continually made every attempt to retain the "friendly-user" environment wherever possible, while extending service to a large group of users characterized by a very wide spectrum of needs, work habits, and usage patterns. By and large, we succeeded in this endeavor.

VII. NEW FACILITIES

A number of major facilities had to be made available in the PWB/UNIX system to make it truly useful in the BISP environment. Initial versions of many of these additional components were written and in use during early 1974. This section describes the current form of these additions (most of which have been heavily revised with the passage of time).

7.1 Remote job entry

The PWB/UNIX Remote Job Entry (RJE) subsystem handles the problems of transmitting jobs to target systems and returning

output to the appropriate users; RJE per se consists of several components, and its use is supported by various other commands.

The **send** command is used to generate job streams for target systems; it is a form of macro-processor, providing facilities for file inclusion, keyword substitution, prompting, and character translation (e.g., ASCII to EBCDIC). It also includes a generalized interface to other UNIX commands, so that all or parts of job streams can be generated dynamically by such commands; **send** offers the users a uniform job-submission mechanism that is almost entirely target-independent.

A transmission subsystem exists to handle communications with each target. "Daemon" programs arrange for queuing jobs, submitting these jobs to the proper target, and routing output back to the user. Device drivers are included in the operating system to control the physical communications links. Some of the code in this subsystem is target-specific, but this subsystem is not visible to end users.

Several commands are used to provide status reporting. Users may inquire about the status of jobs on the target systems, and can elect to be notified in various ways (i.e., on-line or in absentia) of the occurrence of major events during the processing of their jobs.

A user may route the target's output to a remote printer or may elect to have part or all of it returned to the originating PWB/UNIX system. On return, output may be processed automatically by a user-written procedure, or may be placed in a file; it may be examined with the standard UNIX editor, or it can be scanned with a read-only editor (the "big file scanner") that can peruse larger files; RJE hides from the user the distinction between PWB/UNIX files, which are basically character-oriented, and the files of the target system, which are typically record-oriented (e.g., card images and print lines). See Ref. 5 for examples of the use of RJE.

7.2 Source code control system

The PWB/UNIX Source Code Control System (SCCS) consists of a small set of commands that can be used to give unusually powerful control over changes to *modules* of text (i.e., files of source code, documentation, data, or any other text). It records every change made to a module, can recreate a module as it existed at any point in time, controls and manages any number of

concurrently existing versions of a module, and offers various audit and administrative features.[6]

7.3 Text processing and document preparation

One of the distinguishing characteristics of the Research UNIX system is that, while it is a general-purpose time-sharing system, it also provides very good text-processing and document-preparation tools.[7] A major addition in this area provided by the PWB/UNIX system is PWB/MM, a package of formatting "macros" that make the power of the UNIX text formatters available to a wider audience; PWB/MM has, by now, become the de facto Bell Laboratories standard text-processing macro package; it is used by hundreds of clerical and technical employees. It is an easily observable fact that, regardless of the initial reasons that attract users to the PWB/UNIX system, most of them end up using it extensively for text processing. See Ref. 8 for a further discussion of this topic.

7.4 Test drivers

The PWB/UNIX system is often used as a simulator of interactive terminals to execute various kinds of tests of IBM and UNIVAC data-base management and data communications systems, and of applications implemented on these systems; it contains two test drivers that can generate repeatable tests for very complex systems; these drivers are used both to measure performance under well-controlled load and to help verify the initial and continuing correct operation of this software while it is being built and maintained. One driver simulates a *TELETYPE*® CDT cluster controller of up to four terminals, and is used to test programs running on UNIVAC 1100-series computers. The other (LEAP) simulates one or more IBM 3270 cluster controllers, each controlling up to 32 terminals. During a test, the actions of each simulated terminal are directed by a *scenario,* which is a specification of what *scripts* should be executed by that terminal. A script consists of a set of actions that a human operator might perform to accomplish some specific, functional task (e.g., update of a data-base record). A script can be invoked one or more times by one or more scenarios. High-level programming languages exist for both scripts and scenarios; these languages allow one to specify the actions of the simulated terminal-operator pairs, as well as a large

variety of test-data recording, error-detection, and error-correction actions. See Ref. 9 for more details on LEAP.

VIII. MODIFICATIONS TO THE UNIX SYSTEM

Changes that we made to the UNIX operating system and commands were made very carefully, and only after a great deal of thoughtful deliberation. Interface changes were especially avoided. Some changes were made to allow the effective use of the UNIX system in a computer-center environment. In addition, a number of changes were required to extend the effective use of the UNIX system to larger hardware configurations, to larger numbers of simultaneous users, and to larger organizations sharing the machines.

8.1 Reliability

The UNIX system has generally been very reliable. However, some problems surfaced on PWB/UNIX before showing up on other UNIX systems simply because PWB/UNIX systems supported a larger and heavier time-sharing load than most other installations based on UNIX. The continual need for more service required these systems to be run near the limits of their resources much of the time, causing, in the beginning, problems seldom seen on other UNIX systems. Many such problems arose from the lack of detection of, or reasonable remedial action for, exhaustion of resources. As a result, we made a number of minor changes to various parts of the operating system to assure such detection of resource exhaustion, especially to avoid crashes and to minimize peculiar behavior caused by exceeding the sizes of certain tables.

The first major set of reliability improvements concerned the handling of disk files. It is a fact of life that time-sharing systems are continually short of disk space; PWB/UNIX is especially prone to rapid surges in disk usage, due to the speed at which the RJE subsystem can transfer data and use disk space. Experience showed that reliable operation requires RJE to be able to suspend operations temporarily, rather than throwing away good output. The **ustat** system call was added to allow programs to discover the amount of free space remaining in a file system. Such programs could issue appropriate warnings or suspend operation, rather than attempt to write a file that would consume all

the free disk space and be, itself, truncated in the process, causing loss of precious data; the **ustat** system call is also used by the PWB/UNIX text editor to offer a warning message instead of silently truncating a file when writing it into a file system that is nearly full. In general, the relative importance of files depends on their cost in terms of human effort needed to (re)generate them. We consider information typed by people to be more valuable than that generated mechanically.

A number of operational procedures were instituted to improve file-system reliability. The main use of the PWB/UNIX system is to store and organize files, rather than to perform computations. Therefore, every weekday morning, each user file is copied to a backup disk, which is saved for a week. A weekly tape backup copy is kept for two months; bimonthly tape copies are kept "forever"—we still have the tapes from January 1974. The disk backup copies permit fast recovery from disk failure or other (rare) disasters, and also offer very fast recovery when individual user files are lost; almost always, such files are lost not because of system malfunctions, but because people inevitably make mistakes and delete files that they really wish to retain. The long-term tape backup copies, on the other hand, offer users the chance to delete files that they might want back at some time in the future, without requiring them to make "personal" copies.

A second area of improvement was motivated by the need for reliable execution of long-running procedures on machines that operate near the limits of their resources. Any UNIX system has some bound on the maximum number of processes permitted at any one time. If all processes are used, it is impossible to successfully issue the **fork** system call to create a new process. When this happens, it is difficult for useful work to get done, because most commands execute as separate processes. Such transient conditions (often lasting only a few seconds) do cause occasional, random failures that can be extremely irritating to the users (and, potentially, destroy their trust in the system). To remedy this situation, the shell was changed so that it attempts several **fork** calls, separated from one another by increasing lengths of time. Although this is not a general solution, it did have the practical effect of decreasing the probability of failure to the point that user complaints ceased. A similar remedy was applied to the command-execution failures due to the near-simultaneous attempts by several processes to execute the same pure-text program.

These efforts have yielded production systems that users are willing to trust. Although a file is occasionally lost or scrambled by the system, such an event is rare enough to be a topic for discussion, rather than a typical occurrence. Most users trust their files to the system and have thrown away their decks of cards. This is illustrated by the relative numbers of keypunches (30) and terminals (550) in BISP. Users have also come to trust the fact that their machines stay up and work. On the average, each machine is down once a week during prime shift, averaging 48 minutes of lost time, for total prime-shift availability of about 98 percent. These figures include the occasional loss of a machine for several hours at a time, i.e., for hardware problems. However, the net availability to most users has been closer to 99 percent, because most of the machines are paired and operational procedures exist so that they can be used to back each other up. This eliminates the intolerable loss of time caused by denying to an entire organization access to the PWB/UNIX system for as much as a morning or an afternoon. Such availability of service is especially critical for organizations whose daily working procedures have become intertwined with PWB/UNIX facilities, as well as for clerical users, who may have literally nothing to do if they cannot obtain access to the system.

Thus, users have come to trust the systems to run reliably and to crash very seldom. Prime-shift down-time may occur in several ways. A machine may be taken down voluntarily for a short period of time, typically to fix or rearrange hardware, or for some systems programming function. If the period is short and users are given reasonable notice, this kind of down-time does not bother users very much. Some down-time is caused by hardware problems. Fortunately, these seldom cause outright crashes; rather, they cause noticeable failures in communications activities, or produce masses of console error messages about disk failures. A system can "lock-up" because it runs out of processes, out of disk space, or out of some other resource. An alert operator can fix some problems immediately, but occasionally must take the system down and reinitialize it. The causes and effects of the "resource-exhaustion" problems are fairly well-known and generally thought to offer little reason for consternation. Finally, there is the possibility of an outright system crash caused by software bugs. As of mid-1977, the last such crash on a production machine occurred in November 1975.

8.2 Operations

At most sites, UNIX systems have traditionally been operated and administered by highly trained technical personnel; initially, our site was operated in the same way. Growth in PWB/UNIX service eventually goaded us into getting clerical help. However, the insight that we gained from initially doing the job ourselves was invaluable; it enabled us to perceive the need for, and to provide, operational procedures and software that made it possible to manage a large, production-oriented, computer-center-like service. For instance, a major operational task is "patching up" the file system after a hardware failure. In the worst cases, this work is still done by system programmers, but cases where system recovery is fairly straightforward are now handled by trained clerks. Our first attempt at writing an operator's manual dates from that time.

In the area of system administration, resource allocation and usage accounting have become more formal as the number of systems has grown. Software was developed to move entire sections of a file system (and the corresponding groups of users) from volume to volume, or from one PWB/UNIX system to another without interfering with linked files or access history. A major task in this area has been the formalization and the speeding-up of the file-system backup procedures.

By mid-1975, it was clear that we would soon run out of unique "user-ID" numbers. We resisted user pressure to re-use numbers among PWB/UNIX systems. Our original reason was to preserve our ability to back up each PWB/UNIX system with another one; in other words, the users and files from any system that is down for an extended period should be able to be moved to another, properly configured system. This was difficult enough to do without the complication of duplicated user-IDs. Such backup has indeed been carried out several times. However, the two main advantages of retaining unique user-IDs were:

(*i*) Protecting our ability to move users *permanently* from one system to another for organizational or load-balancing purposes.

(*ii*) Allowing us to develop reasonable means for communicating among several systems without compromising file security.

We return to the subject of user-IDs in Section 8.4 below.

8.3 Performance improvements

A number of changes were made to increase the ability of the PWB/UNIX system to run on larger configurations and support more simultaneous users. Although demand for service almost always outran our ability to supply it, minor tuning was eschewed in favor of finding ways to gain large payoffs with relatively low investment.

For a system such as PWB/UNIX, it is much more important to optimize the use of moving-head disks than to optimize the use of the CPU. We installed a new disk driver* that made efficient use of the RP04 (IBM 3330-style) disk drives in multi-drive configurations. The seek algorithm was rewritten to use one (sorted) list of outstanding disk-access requests per disk drive, rather than just one list for the entire system; heuristic analysis was done to determine what I/O request lead-time yields minimal rotational delay and maximal throughput under heavy load. The effect of these changes and of changes in the organization of the disk free-space lists (which are now optimized by hardware type and load expectation) have nearly tripled the effective multi-drive transfer rate. Current PWB/UNIX systems have approached the theoretical maximum disk throughput. On a heavily loaded system, three moving-head drives have the transfer capacity of a single fixed-head disk of equivalent size. The C program listing for the disk driver is only four pages long; this made it possible to experiment with it and to tune it with relative ease.

Minor changes were made in process scheduling to avoid "hot spots" and to keep response time reasonable, even on heavily loaded systems. Similarly, the scheduler and the terminal driver were also modified to help maintain a reasonable rate of output to terminals on heavily loaded systems. We have consciously chosen to give up a small amount of performance under a light load in order to gain performance under a heavy load.

Several performance changes were made in the shell. First, a change of just a few lines of code permitted the shell to use buffered "reads," eliminating about 30 percent of the CPU time used by the shell. Second, a way was found to reduce the average number of processes created in a day, also by approximately 30 percent; this is a significant saving, because the creation of a process and the activation of the corresponding program typically

* Written by L. A. Wehr.

require about 0.1 second of CPU time and also incur overhead for I/O. To accomplish this, shell accounting data were analyzed to investigate the usage of commands. Each PWB/UNIX system typically had about 30,000 command executions per day. Of these, 30 percent resulted from the execution of just a few commands, namely, the commands used to implement flow-of-control constructs in shell procedures. The overhead for invoking them typically outweighed their actual execution time. They were absorbed (without significant changes) into the shell. This reduced somewhat the CPU overhead by eliminating many **fork** calls. Much more importantly, it reduced disk I/O in several ways: swapping due to **fork**s was reduced, as was searching for commands; it also reduced the response time perceived by users executing shell procedures—the improvement was enough to make the use of these procedures much more practical. These changes allowed us to provide service to many more users without degrading the response time of our systems to an unreasonable degree.

The most important decision that we made in this entire area of reliability and performance was our conscious choice to keep our system in step with the Research UNIX system; its developers have been most helpful: they quickly repaired serious bugs, gave good advice where our needs diverged from theirs, and "bought back" the best of our changes.

8.4 User environment

During 1975, a few changes that altered the user environment were made to the operating system, the shell, and a few other commands. The main result of these changes was to more than double the size of the user population to which we could provide service without doing major harm to the convenience of the UNIX system. In particular, several problems had to be overcome to maintain the ease of sharing data and commands. This aspect of the UNIX system is popular with its users, is especially crucial for groups of users working on common projects, and distinguishes the UNIX system from many other time-sharing systems, which impose complete user-from-user isolation under the pretense of providing privacy, security, and protection.

Initially, the UNIX system had a limit of 256 distinct user-IDs;[1] this was adequate for most UNIX installations, but totally inadequate for a user population the size of ours. Various solutions were studied, and most were rejected. Duplicating user-IDs across

machines was rejected for operational reasons, as noted in Section 8.2 above. A second option considered was that of decreasing the available number of the so-called "group-IDs," or removing them entirely, and using the bits thus freed to increase the number of distinct user-IDs. Although attractive in many ways, this solution required a change in the interpretation of information stored with every single disk file (and every backup copy thereof), changes to large numbers of commands, and a fundamental departure from the Research UNIX system during a time when thought was being given to possible changes to that system's protection mechanisms. For these reasons, this solution was deemed unwise.

Our solution was a modest one that depended heavily on the characteristics of the PWB/UNIX user community, which, as mentioned above, consists mostly of groups of cooperating users, rather than of individual users working in isolation from one another. Typical behavior and opinions in these groups were:

(*i*) Users in such a group cared very little about how much protection they had from each other, as long as their files were protected from damage by users outside their group.

(*ii*) A common password was often used by members of a group, even when they owned distinct user-IDs. This was often done so that a needed file could be accessed without delay when its owner was unavailable.

(*iii*) Most users were willing to have only one or two *user-IDs* per group, but wanted to retain their own *login names* and *login directories*. We also favored such a distinction, because experience showed that the use of a single login name by more than a few users almost always produced cluttered directory structures containing useless files.

(*iv*) Users wanted to retain the convenience of inter-user communication through commands (e.g., **write** and **mail**) that automatically identified the sending person.

The Research UNIX **login** command maps a login name into a user-ID, which thereafter identifies that user. Because the mapping from login name to user-ID is many-to-one in PWB/UNIX, a given user-ID may represent many login names. It was observed that the **login** command knew the login name, but did not record it in a way that permitted consistent retrieval. The login name was added to the data recorded for each process and the **udata**

system call was added to set or retrieve this value; the **login** command was modified to record the login name and a small number of other commands (such as **write** and **mail**) were changed to obtain the login name via the **udata** system call. Finally, to improve the security of files, a few commands were changed to create files with read/write permission for their owners, but read-only for everyone else. The net effect of these changes was to greatly enlarge the size of the user community that could be served, without destroying the convenience of the UNIX system and without requiring widespread and fundamental changes.

The second problem was that of sharing commands. When a command is invoked, the shell first searches for it in the current directory, then in directory **/bin**, and finally in directory **/usr/bin**. Thus, any user may have private commands in one of his or her private directories, while **/bin** is a repository for the most frequently used public commands, and **/usr/bin** usually contains less frequently used public commands. On many systems, almost anyone can install commands in **/usr/bin**. Although this is practical for a system with twenty or so users, it is unworkable for systems with 200 or more, especially when a number of unrelated organizations share a machine. Our users wanted to create their own commands, invoked in the same way as public commands. Users in large projects often wanted several sets of such commands: project, department, group, and individual.

The solution in this case was to change the shell (and a few other commands, such as **nohup** and **time**) to search a *user-specified* list of directories, instead of the existing fixed list. In order to preserve the consistency of command searching across different programs, it was desirable to place a user-specified list where it could be accessed by any program that needed it. This was accomplished through a mechanism similar to that used for solving the previous problem. The **login** command was changed to record the name of the user's login directory in the per-process data area. Each user could record a list of directories to be searched in a file named **.path** in his or her login directory, and the shell and other commands were changed to read this file. Although a few users wished to be able to change this list more dynamically than is possible by editing the **.path** file, most users were satisfied with this facility, and, as a matter of observed fact, altered that file infrequently. In many projects, the project administrator creates an appropriate **.path** file and then makes

links to it for everyone else, thus ensuring consistency of command names within the project.

These changes were implemented in mid-1975. Their effect was an upsurge in the number of project-specific commands, written to improve project communication, to manage project data bases, to automate procedures that would otherwise have to be performed manually, and, generally, to customize the user environment provided by the PWB/UNIX system to the needs of each project. The result was a perceived increase in user productivity, because our users (who are, by and large, designers and builders of software) began spending significantly less time on housekeeping tasks, and correspondingly more time on their end products; see Ref. 5 for comments on this entire process by some early PWB/UNIX users.

8.5 Extending the use of the shell

A number of extensions were made to the shell to improve its ability to support shell programming, while leaving its user interface as unchanged as possible. These changes were made only after a great deal of trepidation, because they clearly violated the UNIX system's principle of minimizing the complexity of "central" programs, and because they represented a departure from the Research UNIX shell; these departures consisted of minor changes in syntax, but major changes in intended usage.

During 1974 and early 1975, the PWB/UNIX shell was the same as the Research UNIX shell, and its usage pattern was similar, i.e., it was mainly used to interpret commands typed at a terminal and occasionally used to interpret (fairly simple) files of commands. A good explanation of the original shell philosophy and usage may be found in Ref. 10. At that time, shell programming abilities were limited to simple handling of a sequence of arguments, and flow of control was directed by **if**, **goto**, and **exit**—separate commands whose use gave a Fortran-like appearance to shell procedures. During this period, we started experimenting with the use of the shell. We noted that anything that could be written as a shell procedure could always be written in C, but the reverse was often not true. Although C programs almost always executed faster, users preferred to write shell procedures, if at all possible, for a number of reasons:

(*i*) Shell programming has a "shallow" learning curve, because

anyone who uses the UNIX system must learn something about the shell and a few other commands; thus little additional effort is needed to write simple shell procedures.

(*ii*) Shell programming can do the job quickly and at a low cost in terms of human effort.

(*iii*) Shell programming avoids waste of effort in premature optimization of code. Shell procedures are occasionally recoded as C programs, but only after they are shown to be worth the effort of being so recoded. Many shell procedures are executed no more frequently than a few times a day; it would be very difficult to justify the effort to rewrite them in C.

(*iv*) Shell procedures are small and easy to maintain, especially because there are no object programs or libraries to manage.

Experience with shell programming led us to believe that some very modest additions would yield large gains in the kinds of procedures that could be written with the shell. Thus, in mid-1975, we made a number of changes to the shell, as well as to other commands that are used primarily in shell procedures. The shell was changed to provide 26 character-string variables and a command that sets the value of such a variable to an already existing string, or to a line read from the standard input. The **if** command was extended to allow a "nestable" **if-then-else-endif** form, and the **expr** command was created to provide evaluation of character-string and arithmetic expressions. These changes, in conjunction with those described in Section 8.4 above, resulted in a dramatic increase in the use of shell programming. For example, procedures that lessened the users' need for detailed knowledge of the target system's job control language were written for submitting RJE jobs,* groups of commands were written to manage numerous small data bases, and many manual procedures were automated. A more detailed discussion of shell usage patterns (as of June 1976) may be found in Ref. 11.

Further changes have been made since that time, mainly to complete the set of control structures (by adding the **switch** and **while** commands), and also to improve performance, as explained in Section 8.3 above.

---

* Target-system users who interact with these targets via the PWB/UNIX RJE subsystem make about 20 percent fewer errors in their job control statements than those who interact directly with the targets.

Although the shell became larger, the resulting extensive use of shell programming made it unnecessary for us to build large amounts of centrally-supported software. Thus, these changes to the shell actually reduced the total amount of software that we had to build and maintain, while allowing each user project to customize its own work environment to best match its needs and terminology. A new version of the shell has been written recently;[12] it includes most of our additions, in one form or another.

IX. WHAT WE HAVE LEARNED

Several UNIX systems have served for many years as development facilities in support of minicomputers and microprocessors. The existence of the PWB/UNIX system shows that the UNIX system can also perform this function quite effectively for target machines that are among the largest of the currently available computers. The importance of this observation lies in the fact that the PWB/UNIX system can be used to provide a uniform interface and development facility for almost any programming project, regardless of its intended target, or the size of that target.

Our experience also proves that the UNIX system is readily adaptable to the computer-center environment, permitting its benefits to be offered to a very wide user population. Although some changes and additions to the UNIX system were required, they were accomplished without tampering with its basic fabric, and without significantly degrading its convenience and usability.

Finally, why is the PWB/UNIX system so successful? Certainly, most of the credit goes to the UNIX system itself. In addition, success came partly from what we added to the UNIX system, partly because we provided generally good service, and, perhaps most importantly, because, during the entire design and development process, we forcefully nurtured a close, continuing dialogue between ourselves and our users.

Another reason for the success of the PWB/UNIX system is that it adapts very easily to the individual needs of each user group. Without delving into the "Tower of Babel" effect, it appears that each programming group has strong functional (and perhaps social) needs to radically customize its work environment. This urge to specialize has often been carried out at great cost on other systems. On PWB/UNIX systems, users within such a group share files, build specialized **send** "scripts" for compiling, loading,

and testing their programs on target computers, and write commands (shared within the group) that incorporate nomenclature specific to the group's project. Our changes to the shell and to the user environment, along with the basic facilities of the UNIX system, combined to permit the writing of these new commands as shell procedures. The development of such shell procedures requires at least an order of magnitude less effort than the writing of equivalent C programs. The result is that today, on many PWB/UNIX systems, four out of five commands that are executed originate in such user-written shell procedures.

Speaking as developers of the PWB/UNIX system, we believe that our system fosters real improvement in our users' productivity. The contributing factors have all been touched upon above; the most important of these are:

(*i*) A single, uniform, consistent programming environment.
(*ii*) Good, basic tools that can be combined in a variety of ways to serve special needs.
(*iii*) Protection of data to free the users from time-consuming housekeeping chores.
(*iv*) Very high system availability and reliability.

Taken together, these characteristics instill confidence in our users and make them want to use our system.

One effect that we did not fully forsee was that our changes to the UNIX system (some made under considerable pressure from our users) would lead to an explosion of project-specific software and an expanded demand for PWB/UNIX service. However, we did keep to our original goals:

(*i*) To keep up-to-date with the Research UNIX system.
(*ii*) To change as little of the Research UNIX system as possible.
(*iii*) To make certain that our changes did not compromise the inherent simplicity, generality, flexibility, and efficiency of the UNIX system.
(*iv*) To provide to our users tools, rather than products.

X. ACKNOWLEDGMENTS

The basic concept of the PWB/UNIX system was first suggested by E. L. Ivie.[3] Many of our colleagues have contributed to the design, implementation, and continuing improvement of that system. Thanks must also go to several members of the Bell

Laboratories Computing Science Research Center for creating the UNIX system itself, as well as for their advice and support.

REFERENCES

1. D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," Commun. Assn. Comp. Mach., *17* (July 1974), pp. 365-375.
2. D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," B.S.T.J., this issue, pp. 1905-1929.
3. E. L. Ivie, "The Programmer's Workbench—A Machine for Software Development," Commun. Assn. Comp. Mach., *20* (October 1977), pp. 746-753.
4. T. A. Dolotta and J. R. Mashey, "An Introduction to the Programmer's Workbench," Proc. 2nd Int. Conf. on Software Engineering (October 13-15, 1976), pp. 164-168.
5. M. H. Bianchi and J. L. Wood, "A User's Viewpoint on the Programmer's Workbench," Proc. 2nd Int. Conf. on Software Engineering (October 13-15, 1976), pp. 193-199.
6. M. J. Rochkind, "The Source Code Control System," IEEE Trans. on Software Engineering, *SE-1* (December 1975), pp. 364-370.
7. B. W. Kernighan, M. E. Lesk, and J. F. Ossanna, "UNIX Time-Sharing System: Document Preparation," B.S.T.J., this issue, pp. 2115-2135.
8. J. R. Mashey and D. W. Smith, "Documentation Tools and Techniques," Proc. 2nd Int. Conf. on Software Engineering (October 13-15, 1976), pp. 177-181.
9. T. A. Dolotta, J. S. Licwinko, R. E. Menninger, and W. D. Roome, "The LEAP Load and Test Driver," Proc. 2nd Int. Conf. on Software Engineering (October 13-15, 1976), pp. 182-186.
10. K. Thompson, "The UNIX Command Language," in *Structured Programming—Infotech State of the Art Report,* Nicholson House, Maidenhead, Berkshire, England: Infotech International Ltd. (March 1975), pp. 375-384.
11. J. R. Mashey, "Using a Command Language as a High-Level Programming Language," Proc. 2nd Int. Conf. on Software Engineering (October 13-15, 1976), pp. 169-176.
12. S. R. Bourne, "UNIX Time-Sharing System: The UNIX Shell," B.S.T.J., this issue, pp. 1971-1990.