THE BELL SYSTEM TECHNICAL JOURNAL
Vol. 61, No. 9, November 1982
Printed in U.S.A.

Database Systems:

Making *UNIX** Operating Systems Safe for Databases

By P. J. WEINBERGER

(Manuscript received September 4, 1981)

The UNIX operating system was written for document preparation and software development. Its general utility has led to its use for many other things, including database management. There have been complaints that the system kernel is unsuitable for running applications that use databases: the file system is accused of being too slow and unrobust; the process structure is accused of fatal inefficiencies; and there is neither enough sharing nor any way of controlling what there is. Modest changes to the kernel refute the attack. The file system is robust and can be made much faster; in 32-bit paged systems the process structure matches the needs of transaction processing systems, and a few additional system calls provide all that is needed for sharing and its control. This paper discusses the problems and shows how to solve them.*

I. INTRODUCTION

The interaction between operating systems and database systems deserves more attention than it has received in the literature.[1,5] On the one hand, a database system depends on services provided by the operating system, and the difficulty of writing a database system depends on how many of the facilities it needs are already provided by the operating system. On the other hand, research in databases has provided models for concurrency and atomicity that could be usefully applied within operating systems, for instance, in simplifying the implementation of file systems.

I shall discuss what might be done to the *UNIX* operating system[3,7]

* *UNIX* is a trademark of Bell Laboratories.

to make it more suitable as a host for database systems. *UNIX* systems, in all their varieties, are time-sharing systems that were initially written to make it easy to develop software and to do document processing. As these systems have grown and changed over the years, various decisions have resulted in a system not particularly attuned to the needs of database systems. My thesis is that by modest changes to the operating system, and with some discipline in the design of database systems, the *UNIX* operating system can be made into a system well suited to run database systems.

An alternative[5] is to develop a minimal operating system, and have the database system do the work customarily done by the operating system. This might lead to systems with better performance than can be achieved by adapting the present system, but it might not. The approach of this paper may do well enough by supporting transaction processing systems running a few transactions per second, and it would have the advantage that the resulting system is also convenient for developing software.

I shall present some specific proposals in support of my thesis. The proposals have been chosen to be simple to explain and easy to implement. I do not have enough space for a detailed discussion of each proposal. Nor is there enough space even to describe many alternatives, all with their own advantages and disadvantages.

II. SUMMARY

It is interesting that the worst complaints about the *UNIX* system are not about things like concurrency control—where there is much published work, and where the system is weak—but about the simpler matters of performance and of robustness (not even safety*) against crashes. (Data communications play an important role in applications, but a discussion would double the size of this paper.) The places where the system seems weakest are:

(*i*) Disk performance
(*ii*) File and file system robustness against crashes
(*iii*) Concurrency control
(*iv*) Fast processing of small transactions.

These topics are discussed in more detail later, but here is a summary of the problems. Database systems exist to handle large amounts of data. If the data just trickles in from the disks, it takes too long to

* The file system is safe if stopping the cpu at any time does no more damage than losing free space on the disk.

handle requests made to the database. The disparity in speed between disks and cpus is remarkable. (I shall use performance data based on equipment of about the power of VAX/780s and associated peripherals.) If the heads of a disk are already on the desired cylinder, rotational latency is about 8 milliseconds. Medium-speed hardware executes an instruction every microsecond, say. If one scales time by a factor of a million so that the machine executes an instruction every second, then following a disk request, somewhere between 3 and 14 hours will pass before the data begins to arrive, at the rate of about a byte per second. This enormous discrepancy in speed is the source of much evil in file system organization (inflexible designs in the name of efficiency), and the source of some inadequate performance. For instance, recent *UNIX* systems typically complete about one transfer (of 512 bytes) every two disk revolutions, which provides about 3 percent of the raw disk bandwidth.

Crashes present a particular peril to database systems, not only because there is a lot of data in a database system, but because of the peculiar guarantees about the data that the database system makes to the owner of the data. A regular time-sharing user understands that he or she is responsible for the organization of data inside the files. The user of a database system has had the details of the organization of the data hidden and can only get at the data through the database system. In exchange for taking control of the user's data, the database system guarantees that it will be kept consistent. (The data in the database will be as if the user's programs ran one at a time.) When a crash occurs, operating systems have ways of discovering and repairing the damage to their data structures, and of notifying users when damage cannot be repaired. Interpolating a database system between operating system and user confounds this model. Restoring acceptable risk requires lowering the chance of losing data in a crash. There are two obvious areas in which to lower the risk, one is in the robustness of the operating system's data structures used to find files and data in a file, and the other is in ameliorating any bad effect of writes interrupted by crashes. In *UNIX* systems the data structures are inodes and directories and the writes are controlled by the write-behind buffer caching algorithm. Of course a database system has to be able to maintain transaction logs and audit trails for recovery itself.

Much of the research in database systems relevant to this paper has been in concurrency control, particularly locking. Concurrent users of a database need something to keep them from tripping over each other's changes. Obviously the control represents something shared among users. Sharing now resides entirely within operating system tables where it is carried on invisibly to users, and within the file system where it is available to users. File system access is relatively

slow, certainly too slow for locks of small granularity, so that some other locking mechanism has to be provided.

Many databases are used to support "transaction processing." An example is making an airline reservation on a single flight. Interaction with the database consists of a small set of simple queries and updates. Efficient processing of this work load means that the start-up cost of each transaction has to be low. In the PDP11 *UNIX* system, simple implementations used a separate process for each transaction, and applications of any complexity needed several processes to get around, for example, the small address space.[6] Each process had to be spawned, which involved a lot of copying of process data in memory, and had to gain access to the database, which involved a lot of file opening and directory searching. The result was that time (hundreds of milliseconds) went into overhead and transactions ran slowly.

III. PROBLEMS

The issues sketched above have to be addressed in a database system, but no system would be used for applications if it is too slow or loses files, so these two concerns get more detailed attention in the following sections.

3.1 Disk bandwidth

The I/O architectures of computers differ. The hardware most *UNIX* systems run on interrupts the cpu after each disk command (read, write, or seek) is completed. After the interrupt, system software has to see if the command was executed correctly and then issue the next command. An alternative hardware organization allows the I/O hardware to read sequences of commands out of memory, optionally interrupting the cpu as each transfer of data is completed, but continuing the I/O. The discussion here is limited to the first architecture.

The cpu has one or more disk controllers, and each controller controls one or more disks. At most, one disk on each controller can be sending or receiving data, but all the disks on a controller can be moving their heads simultaneously. If several disks are transferring data at once, it is possible for the memory bus to become saturated, but that problem will be postponed until many others have been resolved. For definiteness I shall assume that the disks are formatted into 512-byte sectors. A reasonable disk has 32 sectors per track and 10 or 20 tracks per cylinder, although improvements in technology keep increasing these numbers.

The data rate the system sees from a single disk, assuming the controller is not busy with other disks, is a function of how far the heads have to move between requests, how much is transferred at each request, and how the data to be transferred are scattered about the

disk. The three factors are not independent. The observed rate of one transfer every two revolutions is only weakly dependent on how much is read at each request. Thus, all other things being equal, a l ·ger fraction of the disk bandwidth can be obtained by transferring more each time. Older *UNIX* systems transfer 512 bytes at a time (hence the 3 percent figure given above). It would appear that the apparent rate could be nearly doubled by transferring 1024 bytes at a time, and experience at the University of California at Berkeley bears this out. There is a point of diminishing returns. If the heads are on the correct cylinder, the average delay until the h ads have moved across a set of blocks is half a rotation plus the time to read the blocks. Thus, the average time to read 3 percent of a track is 0.53 revolution, that to read 25 percent of a track is 0.75 revolution, and that to read a whole track is 1.5 revolutions. Even if seek times were zero, with the heads starting at a random sector on the track, reading a track at a time only gives 2/3 of the raw disk bandwidth. Given that the minimum seek time to an adjacent cylinder is 5 or 6 milliseconds and that the machine takes an interrupt at the end of each transfer, even the greedy strategy of reading a whole track at a time gives less than half the raw disk bandwidth. This silly strategy does provide an upper bound for the bandwidth obtainable if all files are laid out contiguously and read a track at a time, a scheme that corresponds in no wise to the organization of the file system.

3.2 The UNIX file system

Understanding the source of some of the problems, and the opportunities for solving them, requires a description of the file system. A file system is the abstraction of a disk provided by the operating system, although the space used by a file system can be only part of a disk, or several disks. To the user, a file system consists of a tree of directories, each directory containing the names of other directories and names of files. To the operating system, a file system has three parts, a super block at the beginning containing descriptive information, a contiguous collection of blocks containing *inodes*, and the bulk of the file system containing all the file and directory blocks, and some other blocks described below.

A file is an indefinitely extensible vector of bytes with operations read, write, and seek. A directory is a file, except that users cannot write in one, and the operating system interprets the contents as an array of (inode, file-name) pairs. These pairs provide the mapping from file names to inode numbers. The inode number says which inode in the array of inodes in the file system describes the file. The inode for a file contains information about the file's type and accessibility, its length, and where on the disk the data are. The latter is specified

by a short array of block addresses. The first ten entries in the array are the addresses of the first ten data blocks of the file. The next is the address of an indirect block, which is filled with addresses of data blocks. The next is the address of a doubly indirect block, which is filled with addresses of indirect blocks, and the last is the address of a triply indirect block, which is filled with addresses of doubly indirect blocks. Hence, a request out of the blue for a byte near the end of a very large file (one containing a triply indirect block) may require four disk accesses: for the triply indirect block, a doubly indirect, a singly indirect block, and some data (see Table I).

The addresses of the free blocks in the file system are kept in some of the free blocks, which are chained together in a list. The operating system maintains the list of free blocks approximately as a stack.

3.3 The buffer cache

Data transfers are between the file system and a buffer cache maintained by all versions of the *UNIX* system. When a block is to be read, the system checks to see if it is in the cache. If it is, the system doesn't have to read it from the disk. When a block is to be written, it is put in the cache and on the disk driver's queue, but control is returned to the writer before the block goes to the disk. Under a time-sharing load the cache is effective: on the system on which I am writing this paper there have been—since it was last restarted—10,283,000 requests for file system blocks, of which 8,466,000 (82 percent) were found in the cache. (For the record, there were 507,000 blocks written.)

The operating system provides some transparent asynchronous I/O. If two blocks in a row are read from a file, thereafter, while the file is being read sequentially, the operating system reads ahead one block. To the extent that files are scanned sequentially, this decreases the latency of reads. Of the 1,817,000 blocks read from the disk on my machine, 1,413,000 were read-ahead blocks, while 404,000 were read because they weren't in the cache when needed. Reading ahead is effective in databases too.[4]

The strategy for replacing buffers in the cache gets some attention below.

Table I—Disk accesses

	Maximum Size of File	
Block Size (bytes)	With One Indirect Block	With One Doubly Indirect Block
512	64 kbytes	8 Mbytes
1024	256 kbytes	64 Mbytes
2048	1 Mbyte	512 Mbytes
4096	4 Mbytes	4096 Mbytes

IV. SOLUTIONS

4.1 Disk performance

Although databases add difficulties, most of the problems with disk performance already occur for time-sharing loads. Part of the trouble comes from the blocks of a file being scattered randomly across the disk. Therefore, successive reads (from a file or from different files) can be expected to require a fair amount of motion of the disk heads. Most disk drivers sort their queues by cylinder to reduce the average seek time. It is then possible for an arbitrarily long time to elapse between the time when a block to be written is placed on the queue and when it goes to the disk, which time lapse is unsatisfactory for database systems.

Average seek time could be further reduced by better matching of locality on the disk with locality of reference. A reasonable hypothesis is that several blocks will be accessed in each open file, so that if the blocks of a file are near each other on the disk, it will be faster for a program to read them. The conclusion is not so obvious if there are several programs simultaneously using different files on one disk. Regardless, it is hard to see how things would be made worse by keeping the blocks of a file close together; and if the requests of users for data are not exactly interleaved, things will be better than if the blocks are scattered at random. Time spent seeking is wasted; anything that lessens it increases disk throughput.

Not only is there a long time between data transfers from the disk, but the amount of data transferred is small. The larger the size of a block to be transferred, the larger is the fraction of the disk bandwidth being used. Unfortunately, it takes time to transfer data, and memory to put it in; reading data that will never be used consumes resources with no return. There is a trade-off between block size and the fraction of the disk bandwidth used productively. Very small and very large block sizes waste resources; the happy medium depends on characteristics of the load.

The advantages of using blocks larger than 1024 bytes may seem speculative; one disadvantage occurs to everyone: the amount of space occupied by a given set of files is likely to increase as the block size increases because occupied disk space rounds up to a multiple of the block size, so more space is wasted on the disk. Indeed if indirect blocks are ignored, then one-byte blocks waste no space, and the amount of wasted space is an increasing function of the block size. However, the magnitude of the effect varies widely when you measure how much larger specific file systems would be if the block size were increased. Looking at the file systems available to me (at least half full and bigger than 80 megabytes), I produced one in which the size of the file system would be multiplied by 2.4 if the block size were raised

from 512 to 4096, and one in which the increase would only be 0.2 percent. The difference are explicable using two measures, the fraction of files that are directories, and when this is large, the fraction of directories that are nearly empty. In the 2.4 case, 62 percent of all files were directories, 97 percent of the directories were no larger than 128 bytes, and 40 percent of all files were empty directories (which are 32 bytes long to hold the pointers for the directory tree). Cases like this appear to be the result of using the file system directory tree as the access method of a database and files as records in the database. The file system at the other end holds large amounts of data. The average file size in it is 250,000 bytes. The typical increase caused by going from 512 byte blocks to 4096 byte blocks is about 40 percent. This is about one year of disk improvement.

Clever people will have, and have had, no trouble thinking of ways of ameliorating the effect of breakage at the end of files, and coping with short directories. It is not likely that either will be a problem to a database system.

The advantage to using bigger blocks is that each transfer gets more data. To the extent that file access is sequential, the bigger the block the faster the program. Sequential access is common. Programs that use temporary files tend to write them and read them sequentially, or, like the editor, are doing software paging on the contents. Sequential scanning is also popular in databases because it is faster, even in a plain *UNIX* system environment, than random access to data. The drawback to big blocks, other than wasting disk space, is that it takes longer to read them. Reading a big block for a few bytes of data wastes more time than reading a small block. The most common instance of this, based on the data reported above, is reading a directory that is nearly empty. Another case would be random retrieval of single records from a large file. There doesn't seem to be any method for choosing the optimal block size without unrealistically precise knowledge, but my choice is 4096 bytes.

Another advantage of large blocks is that it is easier to allocate them so that the blocks in a file are close together. Instead of the stack-and-list discipline now used to manage free blocks, consider the possibility of using a bit map, with one bit per block. In this case, it is harder to find a free block if there are few, but finding free inodes when there are few is even harder, and is done adequately now. The advantage is that it is easy to see if there is a free block near the rest of the file. Before describing how that might be done, notice the effect of large blocks on a bit map. Only one-eighth the number of bits are needed for 4096-byte blocks as are needed for 512-byte blocks. The entire bit map for a 120,000,000-byte file system fits in the otherwise unused part of the super block. An advantage to putting it there is that a flag bit

could be added to the super block indicating whether the version of the bit mask on the disk is current. This bit would be set as part of an orderly shutdown. After a crash, when the bit was not set, the bit map would be automatically reconstructed as one step in bringing up the machine. In any case, the 32 to 1 space reduction of bit maps over disk addresses means that the total space taken up by a bit map resident in memory is small for 4096-byte blocks.

The stack-and-list discipline for free blocks makes it hard to discover if there are any free blocks near a given place on the disk. A block added to the end of a file should be allocated so that there is only a small delay between reading the previous block in the file and reading the new block. This is particularly easy with a bit map. The exact algorithm depends on how the I/O hardware is organized and how the disk driver organizes its request queue. If there is a delay between finishing one read and starting another, then blocks adjacent on the disk cannot be read as fast as blocks spaced a little farther apart on a track. Likewise, if the disk controller sorts its queue by cylinder (thereby avoiding giving blocks near the middle of the disk better service), then allocating a block with a slightly smaller cylinder number than the previous block in the file is clearly much worse than allocating it with a slightly larger cylinder number.

Here is a specific example. The disk contains 800 cylinders, each of which has 10 tracks, each of which contains 32 512-byte sectors. The disk controller transfers a contiguous set of sectors at each request, and then generates an interrupt. Empirical results show that if sector 0 is read on one request, then the next request cannot read sectors 1, 2, or 3 on the same rotation. (Part of the seemingly wasted time is spent in the controller, which buffers whole sectors to do error checking and correction, and the rest is spent in the system, handling the interrupt and sending the next request to the controller.) Each 4096-byte block uses eight contiguous sectors, so there are four blocks on each track. After reading a block, the next block that can be read without losing a revolution is two farther around the disk. If the last block of a file is the nth block on a track, then the following algorithm decides where the next block should go. First look for a free block in the same cylinder whose number on its track is $n+2$ modulo 4. Failing this, try for $n+3$. Now the choice is between staying on this cylinder and losing a rotation or moving to the next and losing a rotation. There are some interesting questions here, but a rule of thumb is to try to treat the disk as a drum, and stay on the same cylinder as long as possible.

An alternative would be to allocate the blocks in the order 0, 3, 2, 1 as long as possible. For either method, the result should be that sequential reads of a file could get at least one-third of the raw disk

bandwidth, assuming that the requests are not interspersed with requests for other files, and that the program asks for large reads without doing much computation in between.

The discussion above would have a different character if the hardware permitted chaining disk requests together so that the channel remained active as long as there was work for it. The fact that main frames permit this style of I/O may account for the advantages they appear to have over minicomputers for commercial data processing, even when the cpu speeds are the same.

I modified a copy of the kernel for the *UNIX* system to support ordinary file systems with 1024-byte blocks, and the bit-mapped file systems with 4096-byte blocks described above. File systems that would grow too much can be left as ordinary file systems, and other file systems would get the advantages of the new way of doing things. Early measurements bear out the predictions made above. Two programs writing into files at top speed have a total data rate of about 120 kilobytes per second, and groups of programs reading at top speed have about 250 kilobytes per second. If the machine was not busy when a file was written, a single program can read it back at about 300 kilobytes per second. (The disks I use rotate at 3000 rpm, so the raw disk bandwidth is 814 kilobytes per second.) While reading, the cpu (VAX/750) is about 60 percent busy, about two-thirds of this is managing the I/O, and the other one-third is copying the data from the buffer cache to the user. Making the /tmp file system (conventionally used for temporary files) into a 4096-byte file system had good effects. C compilations typically took one-third less system time and one-third less elapsed time than before. The decreased system time is the result of having to do only one-fourth the number of I/O operations, thereby having to make only one-fourth the number of decisions as with a conventional file system.

4.2 Robustness against crashes

From the point of view of the operating system, the file system is consistent when every block is accounted for exactly once (in the free list, as a data block or as a species of indirect block), when the blocks of the files can be found from their inodes, when the number of places a file occurs in the directory tree matches the count in the inode, and so forth. Even in this relatively trivial case it is not easy to write down the exact set of consistency conditions. For instance, it is legal and useful for a file to be in no directories. The rule is that the space for a file is not freed while some process has it open. However, if the system crashes, then when it comes back up, the file is in no directory and no process has it open, so the file system is inconsistent.

Fortunately it is easy to ensure that the file system is safe—a weaker

condition than consistent—whenever there is a crash. Even the weaker guarantee requires some cooperation from the hardware; sector writes must be atomic. Head crashes and sufficiently nasty power glitches can violate this assumption, but well-designed hardware will cope with the ordinary run of power failures. A rough definition is that safety of the file system means that the file system can be restored to a consistent state without deleting files or entries in directories that a user thinks are there. A detailed examination shows that if for each system call, the blocks that are changed are written in the right order, then the file system is always safe. The ordering can be built into the code that implements the system call. All sufficiently new versions of the kernel for the *UNIX* system use this strategy, and it is now unusual to lose a file in a crash.

With the file system thus protected against a class of common failures, it is reasonable to ask about the effect of the loss of a disk block or sector. Damage that cannot be detected when the block is read may contaminate the entire file system or database. Because of the error correction used on disks, such blocks are likely to be the result of the operating system or user software writing bad data. (But not exclusively, for the controller could lie about where the heads are. There is an enormous variety of possible failures. The effects of most of them can barely be ameliorated.) The damage from detectably bad blocks depends on what the block contains. The system will not use the data in these blocks. Bad data blocks and indirect blocks make parts of files inaccessible. Bad directory blocks make whole files inaccessible. The contents of the files can be recovered through their inodes, but their names have been lost. A bad super block can easily be fixed, because the critical information in it is static. The worst case is the loss of an inode block, for although the set of data blocks and indirect blocks accessible through the lost inodes can be determined (if the free blocks are known), there is no way of deciding the ordering of the blocks, nor which is associated with each lost inode. However the names of the lost files are accessible through the directory tree.

Thus, the detected failure of a single sector will result in the contents of one or more files being inaccessible, and possibly the names of some files too. In a database system, loss or damage to a single file can be undone by standard procedures using the transaction log and a checkpoint of the database. The same holds for the directory trees associated with a database. To recover from the loss of an inode block, the database needs to be able to identify files from contents, or it has to be prepared to recover a number of missing files. The latter alternative is covered by standard database recovery mechanisms. Doing the former would doubtless be much faster, but requires the database system to make sure it can identify files by the contents of individual blocks.

In summary, modern versions of the file system are robust against crashes. The places where the file system of the *UNIX* operating system is least robust are exactly the places where database systems have to have their own recovery mechanisms anyway.

To implement these mechanisms, particularly logging, database systems need additional operating system services. At certain times during a transaction the database system has to be sure that all the blocks so far affected by the transaction are on the disk. In particular, before a transaction can commit it has to have written its log to (what passes for) stable storage, namely the disk. Frequently there is a partial ordering that should be satisfied on the blocks going to the disk as part of committing. Both these needs can be satisfied by a slight extension to the system to provide a "wait until all my disk I/O is finished" system call. The changes to the system to provide this service are neither hard nor extensive.

It might appear that additional help is needed to maintain the log. A typical log is a tape, or a large circular buffer. Each transaction writes its log records at the end of the log. The operating system could either provide a "write at end of file", or some efficient form of locking such as that described below. Since there are several sectors in each file system block, the database system has to make sure it can tell if the whole log record was written when it has to recover from a crash that occurs during writing the log.

4.3 Concurrency control

Concurrency control is more like science than any other area of databases[1]. The only question that is relevant here is how to provide the services needed within the kernel. The ideal solution should be efficient: setting and releasing locks should not require thousands of instructions, nor should the process have to give up the cpu unless there is a locking conflict. A solution must be flexible: there is no reason every database system should be compelled to use exactly the same locking algorithms.

The essence of concurrency control is sharing of information and control. The operating system provides sharing between independent processes only through the file system. Hence a natural way to implement locking is through a device driver. In the first place, the interface between device drivers and the rest of the system is sharply defined and simple. Therefore, adding one or more locking 'I/O devices' is no more of an operating system modification than, say, adding a new kind of printer. Second, the overhead in locking above the implementation of the lock algorithms is just that of a system call, a few hundred instruction executions. In particular, the process does not give up the cpu as a result of such a system call, since it invokes no external I/O.

In one scenario there would be a device named */dev/db-lock* associated with the database system. A process using a database would first open */dev/db-lock*. It would then obtain and release locks using the *ioctl* system call. If the call returned normally all would be fine, otherwise an error code would indicate that the process had had its transaction blighted because of deadlock. The transaction would then have to invoke some database subroutines to back itself out. This is consistent with the philosophy that an error causes a process to do something to itself, and allows the database system to try to redo transactions without bothering the user.

The locking driver has advantages over using a separate process to manage locks. A separate process would have to be separately scheduled, opening up the possibility of subtle queueing problems in addition to the extra overhead of process switching. The lock driver is notified automatically when a process dies, because a process's open files are closed as part of process termination. A lock process would have to have a special arrangement to learn about the death of a process holding locks. Finally, a locking driver generalizes smoothly to distributed systems (although things have to be added) without any change to user programs.

4.4 Transaction processing

A process of the *UNIX* system executes a program in two steps. The process first clones itself, using the *fork* system call, and then one of the siblings overlays itself with the new program, using the *exec* system call. In some versions a *fork* requires copying all of the process's writable memory. (The executable text of a process may be unwritable.) In all versions the *exec* requires finding a copy of the program to be executed.

The process structure of a database application has to be designed carefully to avoid unacceptable inefficiencies. Avoiding them may not be possible on machines with 16 bits of addressing.[6] The biggest inefficiency is associated with *execs*, one parameter of which is the name of the file to be executed. The operating system has to find the named file and read it, either from the file system or from a copy in the swap/paging area. Thus, making *exec* efficient requires that it not be hard to find the named file, and that there be a conveniently located copy of the program. Both of these requirements are easy to achieve. Not surprisingly the solutions mimic the functionality of the daily initialization of commercial transaction processing systems.

The key to efficiency is caching. When the system starts up each morning, one process executes—or otherwise gets into swap/paging space—all of the programs that will be executed. The process has no other function than to force the operating system to cache copies of

frequently used programs. It also opens important directories and files, so that their inodes are cached in memory. The result is a savings in the number of disk accesses required to open commonly used files and to execute programs.

A paging version of the system can save time during the *forks* too. The memory pages of the two copies of the program can be marked as shared. A page needs to be copied only if one of the sharers writes on it. (The hardware has to support this.) Any implementation has to balance the cost of page faults with the cost of copying more pages than necessary. The traditional method takes no page faults and copies too much.

Little copying is particularly effective if the application is organized so that all the most common interactive jobs are combined into one process. One instance of the process is started when the day begins. When it receives a message specifying work to be done, it forks a copy of itself, and the copy handles the work without having to issue an *exec*. The copy needs to open no files, since it inherits them at its birth. Instead it *fdups** the file descriptors, which requires no I/O. Since much of its inherited data will be irrelevant to the job it is doing, most of the memory pages will not be copied. The machine on which this is being written (VAX/750) will handle 10,000 simple transactions an hour, where a simple transaction is a fork followed by eight random reads and four random writes of 1024 bytes each. Copying an extra 100,000 bytes of program data at each fork cuts the rate in half.

Requests for uncommon jobs require an *exec* after the fork, and requests that imply off-hours processing can be queued by having the program execute standard *UNIX* system commands.

4.5 Access methods and the buffer pool

Conventional wisdom is that a database system needs to manage its own buffer pool, and that this pool should be in shared memory. Although there are many reasons for having a buffer pool, it is not clear that the database system needs to manage it, and even if it does, it is not clear that the buffer pool has to be in the database system's memory. The most common claim denigrating the operating system's ability to manage the pool is that the database system knows better which pages should be prefetched. That is, the database system's notion of sequence in a file does not agree with the order of the bytes, which is the operating system's notion.

In organizations similar to ISAM there are two places where the operating system's idea of sequence is incorrect. In the tree that leads

* A new version of *dup* which creates a new file table entry in the kernel.

to the block in which the record should be, there is no idea of sequence at all, since the tree is branching. Then as data blocks overflow, they are chained to overflow blocks, so the correct view of sequence is to follow the overflow chain. My view is that such file organizations are obsolete. A careful implementation of file organizations having the flavor of B-trees has all the advantages of any bushy tree organization, together with a guaranteed access time.

A similar situation holds with hashing. Any version of hashing into bins can produce overflow chains. The alternative is extensible hashing. Extensible hashing can guarantee that at most two file system accesses are required to retrieve a record. For files that are growing, other forms of hashing cannot balance wasted space and retrieval cost nearly as well as extensible hashing.

In either case, if there are a lot of overflow chains it is unlikely that the file will be laid out properly on the disk, so that there will be excess head motion, slowing the flow of data into the machine.

A more important consideration is the handling of the buffer cache. Clearly blocks that contain structural data or directory data are more likely to be reused than blocks that are read while doing a sequential scan of a file. This generalization is true both for databases and for operating systems. Blocks of the first sort can be handled by replacing the least recently used ones, while blocks of the second sort need not be kept at all. The cache algorithm of the *UNIX* system recognizes this. (The assertion to the contrary in Ref. 5 is no longer true.) Some more recent versions of the *UNIX* system distinguish among several kinds of blocks.

Thus, the only situation where the system's cache may not be handled in the same manner as the database system is where the system does not recognize index or dictionary blocks as deserving to be cached, and where the database system might tend to cache them. Discovering if this affects performance requires specification of the file structures and algorithms used by the database system.

Another argument against the database system keeping a cache in its memory is the possibility of self-defeating interactions with paging. See Ref. 2 and further references therein.

V. CONCLUSION

By now the reader should be persuaded that a little effort inside the operating system would make the *UNIX* system much more satisfactory for database systems. Larger blocks and nearly contiguous file allocations give better I/O performance. New system calls for synchronizing processes with the completion of I/O and for locking are the tools with which the standard concurrency control, transaction logging, and auditing algorithms can be implemented. Adding a simple analog

of *dup* makes efficient the natural match of system processes and the flow of control needed for transaction processing. Combining modern file organizations and the services of the system buffer cache should decrease latency. None of these changes will make the system less satisfactory for what it is used for now.

REFERENCES

1. J.N. Gray, "Notes on Data Base Operating Systems," in *Operating Systems, an Advanced Course*, New York: Springer Verlag, 1978.
2. Tomás Lang, C. Wood, and E.B. Fernández, "Database Buffer Paging in Virtual Storage Systems," Trans. Database Systems, *3,* No. 4 (December 1977), pp. 339–51.
3. D.M. Ritchie and K. Thompson, "The *UNIX* Time-Sharing System," B.S.T.J., *57,* No. 6, Part 2 (July–August 1978), pp. 1905–30.
4. A.J. Smith, "Sequentiality and Prefetching in Database Systems," Trans. Database Systems, *3,* No. 3 (September 1978) pp. 223–47.
5. M. Stonebraker, "Operating System Support for Database Management," Commun. ACM, *24,* No. 7 (July 1981), pp. 412–18.
6. M. Stonebraker, "Retrospective on a Database System," Trans. Database Systems, *5,* No. 2 (June 1980), pp. 225–40.
7. K. Thompson, "*UNIX* Implementation," B.S.T.J., *57,* No. 6, Part 2 (July–August 1978), pp. 1931–46.