OPERATING SYSTEM SUPPORT FOR DATA BASE MANAGEMENT

by

Michael Stonebraker

Memorandum No. UCB/ERL M80/47

4 November 1980

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

OPERATING SYSTEM SUPPORT FOR DATA BASE MANAGEMENT

by

Michael Stonebraker
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
UNIVERSITY OF CALIFORNIA
BERKELEY, CA.

ABSTRACT

This paper examines several operating system services with a view toward their applicability to support of data base management functions. These services include buffer management, file systems, scheduling, interprocess communication and consistency control.

I INTRODUCTION

In this paper we examine several popular operating system services and indicate whether they are appropriate for support of data base management (DBMS) functions. Often we will see that the wrong service is provided or that severe performance problems exist. When possible, we offer some suggestions concerning possible improvements. In the next several sections we treat the services provided by buffer pool management, the file system, scheduling, interprocess communication and consistency control. We then conclude with a discussion of the merits of including all files in a paged virtual memory.

II BUFFER POOL MANAGEMENT

Many modern operating systems (e.g. UNIX [RITC75]) provide a main memory cache for the file system. In brief, UNIX provides a buffer pool whose size is set when the operating system is compiled. Then, all file I/O is handled through this cache. A file read returns data directly from a block in the cache, if possible, else it causes a block to be "pushed" to disk and replaced by the desired block. A file write simply moves data into the cache; at some later time the buffer manager writes the block to the disk. The UNIX buffer manager uses the popular LRU [MATT70] replacement strategy. Lastly, when UNIX detects sequential access to a file, it prefetches blocks before they are requested.

Conceptually, this service is desirable because blocks for which there is so-called "locality of reference" [MATT70, SHAW74] will remain in the cache over repeated reads and writes. However, the following problems arise in using this service for data base management.

2.1 Performance

The overhead to fetch a block from the buffer pool manager usually includes that of a system call and a core-to-core move. For UNIX on a PDP-11/70 the cost to fetch 512 bytes exceeds 5000 instructions. To fetch 1 byte from the buffer pool requires about 1800 instructions. It appears that these numbers are somewhat higher for UNIX than other contemporary operating systems. Moreover, they can be cut somewhat for VAX 11/780 hardware [KASH80]. This trend toward lower overhead access will hopefully continue.

However, many DBMS's including INGRES [STON80] and SYSTEM R [BLAS79a] choose to put a DBMS managed buffer pool in user space to reduce overhead. Hence, each of these systems has gone to the trouble of constructing their own buffer pool manager to enhance performance.

In order for an Operating System (OS) provided buffer pool manager to be attractive, the access overhead must be cut to a few hundred instructions. The trend toward providing the file system as a part of shared virtual memory (e.g. PILOT [REDE80]) may provide a solution to this problem. This topic is examined in Section VII.

2.2 LRU Management

Although the folklore indicates that LRU is a generally good tactic for block management, it appears to perform only marginally in a data base environment. Data base access is a combination of:

1) sequential access to blocks which will not be re-referenced

2) sequential access to blocks which will be cyclically re-referenced

3) random access to blocks which will not be referenced again

4) random access to blocks for which there is a non-zero probability of re-reference

Although LRU works well for case 4), it is a bad strategy for the other situations. Since a DBMS knows which blocks are in each category, it can use a composite strategy. For case 4) it should use LRU while for 1) and 3) it should use "toss immediately". For blocks in class 3) the reference pattern is 1,2,3,...,n,1,2,3,... Clearly LRU is the worst possible replacement algorithm for this situation. Unless all n pages can be kept in the cache, the strategy should be to "toss immediately". Initial studies [KAPL80] suggest that the miss ratio can be cut 10-15 percent by a DBMS specific algorithm.

In order for a OS to provide buffer management, some means must be found to allow it to accept "advice" from an application program (e.g. a DBMS) concerning the replacement strategy. It appears to be an interesting problem to design a clean buffer management interface with this feature.

2.3 Prefetch

Although UNIX correctly prefetches pages when sequential access is detected, there are important cases where it fails.

Except in rare cases INGRES knows at (or very shortly after) the beginning of its examination of a block EXACTLY which block it will access next. Unfortunately, this block is not necessarily the next one in logical file order. Hence, there is no way for an OS to implement the correct prefetch strategy.

2.4 Crash Recovery

An important DBMS service is to provide recovery from hard and soft crashes. The desired effect is for a unit of work (a transaction) which may be quite large and span multiple files either to be completely done or look like it had never started.

The way many DBMSs provide this service is to maintain an "intentions list". When the intentions list is complete a "commit flag" is set. The last step of a transaction is to process the intentions list making the actual updates. The DBMS makes the last operation idempotent (i.e. it generates the same final outcome no matter how many times the intentions list is processed) by careful programming. The general procedure is described in [GRAY78, LAMP76]. An alternate process is to do updates directly as they are found and maintain a log of "before images" so that backout is possible.

During recovery from a crash the commit flag is examined. If set, the DBMS recovery utility processes the intentions list to correctly install the changes made by updates in progress at the time of the crash. If the flag is not set, the utility removes the intentions list thereby backing out the transaction. The impact of crash recovery on the buffer pool manager is the following.

The page on which the commit flag exists must be forced to disk AFTER all pages in the intentions list. Moreover, the transaction is not reliably committed until the commit flag is forced out to the disk, and no response can be given to the person submitting the transaction until this time.

The service required from an OS buffer manager is a "selected force out" which would push the intentions list and the commit flag to disk in the proper order. Such a service is not present in any buffer manager known to the author.

2.5 Summary

Although it is possible to provide an OS buffer manager with the required features, none exists currently to this author's knowledge. It would be an interesting exercise to design such a facility with prefetch advice, block management advice and selected force out. This exercise is of interest both in a paged virtual memory context and in an ordinary file system.

III THE FILE SYSTEM

The file system provided by UNIX supports objects (files) which are character arrays of dynamically varying size. On top of this abstraction, a DBMS can provide whatever higher level objects it wishes.

This is one of two popular approaches to file systems; the second is to provide a record management system inside the OS (e.g RMS-11 for DEC machines or Enscribe for Tandem machines). In this approach structured files are provided (with or without variable length records). Moreover, efficient access is often supported for fetching records corresponding to a user supplied value (or key) for a designated field or fields. Multilevel directories, hashing and secondary indexes are used to provide this service.

The point to be made in this section is that the second service, which is what the DBMS wants, is not always efficient when constructed on top of a character array object. The following subsections explain some considerations.

3.1 Physical Contiguity

The character array object can usually be expanded one block at a time. Often the result is blocks of a given file scattered over a disk volume. Hence, the next logical block in a file is not necessarily physically close to the previous one. Since a DBMS does considerable sequential access, the result is considerable arm movement.

The desired service is for blocks to be stored physically contiguous and a whole collection to be read when sequential access is desired. This naturally leads a DBMS to prefer a so-called extent based file system (e.g. VSAM [KEEH74]) to one which scatters blocks. Of course, such files must grow an extent at a time rather than a block at a time.

3,2 Tree Structured File Systems

UNIX implements two services by means of data structures which are trees. The blocks in a given file are kept track of in a tree (of indirect blocks) pointed to by a file control block (i-node). Second, the files in a given mounted file system have a user visible hierarchical structure composed of directories, subdirectories, etc. This is implemented by a second tree. A DBMS such as INGRES then adds a third tree structure to support keyed access via a multilevel directory structure (e.g. ISAM [IBM66], B-trees [BAYE70, KNUT78], VSAM [LACE76], etc.).

Clearly, one tree with all three kinds of information is more efficient than three separately managed trees. It is suspected that the extra overhead is substantial.

3.3 Summary

It is clear that a character array is not a useful object to a DBMS. Rather it is the abstraction presumably desired by language processors, editors, etc. Instead of providing records management on top of character arrays, it is possible to do the converse; the only issue is one of efficiency. Moreover, editors can probably use records management structures as efficiently as the ones which they create themselves [BIRS77]. It is suspected that OS designers should change their thinking toward providing DBMS facilities as lower level objects and character arrays as higher level ones. This philosophy is already present in [EPST80]..

IV SCHEDULING AND PROCESS MANAGEMENT

Often, the simplist way to organize a multi-user data base system is to have one process per user; i.e. each concurrent user runs in a separate process. Hopefully, all users share the same copy of the code segment of the data base system and perhaps one or more data segments. In particular, a DBMS buffer pool and lock table should be handled as a shared segment. The above structure is followed by System R and in part by INGRES. Since UNIX has no shared data segments, INGRES must put the lock table inside the operating system and provide buffering private to each user.

The alternative organization is to allocate one run time data base process which acts as a server. All concurrent users send messages to this server with work requests. The one run time server schedules requests through its own mechanisms and may support its own multi-tasking system. This organization is followed by IMS [IBM74] and by Enscribe [TAND79].

Although Lauer [LAUE79] points out that the two methods are equally viable conceptually, the design of most modern operating systems strongly favors the first approach. For example, UNIX contains a message system (pipes) which is incompatible with the notion of a server process. Hence, it forces the first alternative. There are at least two problems with the process-per-user alternative.

4.1 Performance

Every time a run time data base process issues an I/O request that cannot be satisfied by data in the buffer pool, a task switch is inevitable. The DBMS suspends waiting for required data to appear and another process is run. It is possible to make task switches very efficient, and some operating systems can perform a task switch in a few hundred instructions. However, many operating systems have "large" processes, i.e. ones with a great deal of state information (e.g. accounting) and a sophisticated scheduler. This tends to cause task switches costing a thousand instructions or more. This is a high price to pay for a buffer pool "miss".

4.2 Critical Sections

Blasgen [BLAS79] has pointed out that some DBMS processes have critical sections. If the buffer pool is a shared data segment, then portions of the buffer pool manager are necessarily critical sections. System R handles critical sections by setting and releasing locks which basically simulate semaphores. A problem occurs if the operating system scheduler deschedules a data base process while it is holding such a lock. All other run time data base processes immediately queue up behind the locked resource. Although the probability of this occuring is low, the resulting convoy [BLAS79] has a devastiating effect on performance.

As a result of these two problems with the process-per-user model one might expect the server model to be especially attractive. The following section explores this point of view.

4.3 The Server Model

A server model becomes viable if the operating system provides a message facility where n processes can originate messages to a single destination process. However, such a server must do its own scheduling and multi-tasking. This involves a painful duplication of operating system facilities. In order to avoid such duplication, one must resort to the following tactics.

One can avoid multi-tasking and a scheduler by a first-come-first-served server with no internal parallelism. A work request would be read from the message system and executed to completion before the next one was started. This situation makes little sense if there is more than one physical disk. Each work request will tend to have one disk read outstanding at any instant. Hence, at most one disk will be active with a non-multi-tasking server. Even with a single disk, there is the issue that a long work request will be processed to completion while shorter requests must wait. The penalty on average response time may be considerable [SHAW74].

To achieve internal parallelism yet avoid multi-tasking, one could have user processes send work requests to one of perhaps several common servers. However, such servers would have to share a lock table and are only slightly different from the shared-code process per user model. Alternately, one could have a collection of servers, each of which would send low level requests to a group of disk processes which actually do the I/O and handle locking. A disk process would process requests in first-in-first-out order. Although this organization appears potentially desirable, it still may have the response time penalty mentioned above. Moreover, it results in one message per I/O request. In reality one has traded a task switch per I/O for a message per I/O; and the latter may be more expensive than the former.

4.4 Summary

There appears no way out of the scheduling dilema; both the server model and the individual process model appear unattractive. The common solution for high performance DBMSs is multi-tasking in user space, thus duplicating operating system features.

One ultimate solution might be for an operating system to create a special scheduling class for the DBMS and other "favored" users. Processes in this class would never be forcibly descheduled but might voluntarily relinquish the CPU at appropriate intervals. This would solve the convoy problem mentioned in Section 4.2. Moreover, such special processes might also be provided with a fast path through the task switch/scheduler loop to pass control to one of their sibling processes. Hence, a DBMS process could pass control to another DBMS process at low overhead.

V INTERPROCESS COMMUNICATION

It has been pointed out that an operating system message system often makes a server organization impossible. The only other point to be made concerns performance.

5.1 Performance

Although the author has never been offered a good explanation concerning why messages should be so expensive, the fact remains that in most operating systems the cost for a round trip message is several thousand instructions. For example, in PDP-11/70 UNIX the number is about 5000. As a result care must be exercised in a DBMS to avoid overuse of a facility that is not cheap. Hence, otherwise viable DBMS organizations must sometimes be rejected because of excessive message overhead.

5.2 Summary

The problem in Section 4 and 5 is at least in part the overhead in some operating systems of task switches and messages. Either operating system designers must make these facilities cheaper or provide special "fast path" functions for DBMS consumers. If this does not happen, DBMS designers will presumably continue the present practice; implementing their own multi-tasking, scheduling and message systems entirely in user space. The result is a "mini" operating system in addition to a DBMS.

VI CONSISTENCY CONTROL

The services provided by an operating system in this area include the ability to lock objects for shared or exclusive access and support for crash recovery. Although most operating systems provide locking for files, there are a lesser number which support finer granularity locks, such as ones on pages or records. Such smaller locks are deemed essential in some data base environments.

Moreover, many operating systems provide some cleanup after crashes. If they do not provide support for data base transactions such as discussed in Section 2.4, then a DBMS must provide transaction crash recovery on top of whatever is provided.

It is sometimes proposed that both concurrency control and crash recovery for transactions be provided entirely inside the operating system (e.g. [LAMP76]). At least conceptually, they should be at least as efficient as if provided in user space. The only problem with this approach is buffer management. If a DBMS provides buffer management in addition to whatever is done by the operating system, then transaction management by the operating system is impacted as discussed in the following sections.

6.1 Commit Point

When a data base transaction commits, a user space buffer manager must ensure that all apppropriate blocks are flushed and a commit delivered to the operating system. Hence, the buffer manager cannot be immune from knowledge of transactions, and operating system functions are duplicated.

6.2 Halloween Problem

Consider the following employee data:

Empname	Salary	Manager
Smith	10000	Brown
Jones	9000	none
Brown	11000	Jones

and the update which gives a 20 percent pay cut to all employees who earn more than their managers. Presumably, Brown is the only employee who receives a decrease, although there are alternative semantic definitions.

Suppose the DBMS updates the data set as if finds "overpaid" employees, depending on the operating system to provide backout or recover-forward on crashes. If so, Brown might be updated before Smith was examined, and as a result, Smith might also get the pay cut. It is clearly undesirable to have the outcome of an update depend on the order of execution.

If the operating system maintains the buffer pool and an intentions list for crash recovery, it can avoid this problem [STON76]. However, if there is a buffer pool manager in user space, it must maintain its own intentions list in order to properly process this update. Again operating system facilities are being duplicated.

6.3 Summary

It is certainly possible to have buffering, concurrency control and crash recovery all provided by the operating system. However, to be successful the performance problems mentioned in Section 2 must be overcome. It is also reasonable to have all three services provided by the DBMS in user space. However, if buffering remains in user space and consistency control does not, then a lot of code duplication appears inevitable. Presumably, this will cause performance problems in addition to increased human effort.

VII PAGED VIRTUAL MEMORY

It is often claimed that the appropriate operating system tactic for data base management support is to bind files into a user's paged virtual address space. We briefly discuss the problems inherent in this approach.

7.1 Files are Large

Any virtual memory scheme must handle files which are large objects. Popular paging hardware creates an overhead of 4 bytes per 4096 byte page. Consequently, a 100 Mbyte file will have an overhead of 100K bytes for the page table. As a result, one should not necessarily assume that the page table is main memory resident. Therefore, one has the possibility that an I/O operation will induce two page faults; one for the page containing the page table for the data in question and one on the data itself. To avoid the second fault one must "wire down" a large page table in main memory.

Conventional file systems include the information contained in the page table in a file control block. Especially in extent based file systems, a very compact representation of the information is possible. A run of 1000 consecutive blocks can be represented as a starting block and a length field. However, a page table for this information would store each of the 1000 addresses even though they differ by one from their predecessor. Consequently, a file control block is usually made main memory resident at the time the file is opened. As a result, the second I/O need never be paid.

The alternative is to bind "chunks" of a file into ones address space. Not only does this provide a multi-user DBMS with a substantial bookkeeping problem concerning whether needed data is currently addressable but also may require a number of bind-unbind pairs in a transaction. Since the overhead of a "bind" is likely to be comparable to that of a file open, this may slow performance substantially.

An open question concerns whether novel paging organizations can assist with the problems of this section.

7.2 Buffering

All of the problems discussed in Section 2 concerning buffering (e.g. prefetch, non LRU management and selected force out) exist in a paged virtual memory context. How they can be cleanly handled in this context is an open question.

VIII CONCLUSIONS

The bottom line appears to be that operating system services in many existing systems are either too slow or inappropriate. Existing DBMSs usually provide their own and make little or no use of the ones provided by the operating system. Hopefully, future operating system designers will become more sensitive to DBMS needs.

A DBMS would prefer a small efficient operating system with only desired services provided. The closest thing currently available are so called "real time" operating systems which provide minimal facilities efficiently. On the other hand, most general purpose operating systems provide all things to all people at much higher overhead. Hopefully, future operating systems will be able to provide both sets of services in one environment.

ACKNOWLEDGEMENT

This research was sponsored by the U.S. Air Force Office of Scientific Research Grant 78-3596, the U.S. Army Research Office Grant DAAG29-76-G-0245, the Naval Electronics Systems Command Contract N00039-78-G-0013 and the National Science Foundation Grant MCS75-03839-A01.

REFERENCES

[BAYE70] Bayer, R., "Organization and Maintenance of Large Ordered Indices," Proc. 1970 ACM-SIGFIDET Workshop on Data Description and Access, Houston, Texas, November 1970.

[BIRS77] Birss, E., hewlett-Packard Corp., General Systems Division, private communication.

[BLAS79] Blasgen, M. et. al., "The Convoy Phenomenon," Operating Systems Review, April 1979.

[BLAS79a] Blasgen, M. et. al., "System R: An Architectural Update," IBM Research, San Jose, Ca., Report RJ 2581, July 1979.

[EPST80] Epstein, R. and hawthorn,, P., "Design Decisions for the Intelligent Database Machine," Proc. 1980 National Computer Conference, Anaheim Ca., May 1980.

[GRAY78] Gray, J., "Notes on Operating Systems," IBM Research, San Jose, Ca., Report RJ 3120, October 1978.

[IBM66] IBM Corp., "OS ISAM Logic," IBM, White Plains, N.Y., GY28-6618, June 1966.

[IBM74] IBM Corp., "IMS-VS General Information manual," IBM, White Plains, N.Y., GH20-1260, April 1974.

[KAPL80] Kaplan, J., "Buffer Management Policies in a Data base System," M.S. Thesis, University of California, Berkeley, Ca., 1980.

[KASH80] Kashtan, D., "UNIX and VMS: Some Performance Comparisons," SRI International, Menlo Park, Ca., unpublished working paper.

[KEEH74] Keehn, D. and Lacy, J., "VSAM Data Set Design parameters," IBM Systems Journal, September 1974.

[KNUT78] Knuth, D., "The Art of Computer Programming, Vol 3: Sorting and Searching" Addison Wesley, Reading, Mass., 1978.

[LAMP76] Lampson, B. and Sturgis, H., "Crash Recovery in a Distributed System," Xerox PARC, Palo Alto, Ca., 1976 (working paper).

[LAUE79] Lauer, H. and Needham, R., "On the Duality of Operating System Structures," Operating Systems Review, April 1979.

[MATT70] Mattson, R. et. al., "Evaluation Techniques for Storage Hierarchies," IBM Systems Journal, June 1970.

[REDE80] Redell, D., et. al., "Pilot: An Operating System for a Personal Computer," CACM, February, 1980.

[RITC75] Ritchie, D. and Thompson, K., "The UNIX Time-sharing System," CACM, June 1975.

[SHAW74] Shaw, A., "The Logical Design of Operating Systems," Prentice-Hall, Englewood Cliffs, N.J., 1974.

[STON76] Stonebraker, M. et. al., "The Design and Implementation of INGRES," ACM-TODS, September 1976.

[STON80] Stonebraker, M., "Retrospection on a Data Base System," ACM-TODS, June, 1980.

[TAND79] Tandem Computers, "Enscribe Reference Manual," Tandem, Cupertino Ca., August 1979.