VIRTUAL MEMORY TRANSACTION MANAGEMENT

by

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

ABSTRACT

In this paper we examine the consequences of an operating system providing transaction management in an environment where files are bound into a user's address space. The discussion focuses on inherent limitations in providing concurrency control and crash recovery services in this environment and on hardware extensions needed to overcome these deficiencies.

I INTRODUCTION

It is widely suggested that transaction management be performed as part of the function of an operating system. This approach provides clients (such as editors and data base managers) access to concurrency control and crash recovery services without special user-level code. Operating systems (OS) have started to appear with this orientation, e.g. [MITC82], and new systems are being proposed which incorporate object managers with this outlook, e.g. [SPEC83].

It is also widely suggested that an operating system map files into a user's virtual address space [REDD81, ORGA72]. Then, a client program can simply read a file by referencing virtual memory locations. If the page holding the desired data is not resident in main memory, a page fault will cause it to be brought in. Writes to a file would be done by moving data values to virtual memory locations. At some later time, modified pages would be written out to disk by the page manager. In this fashion, the page manager would subsume all buffering decisions, and clients (including data managers) would be spared the necessity of providing custom routines.

This paper explores the implications to an OS transaction manager of binding files into virtual memory. In Section II we indicate the problems which can arise in the three major mechanisms for providing concurrency control. Then, in Section III we review the two main themes for providing crash recovery, namely deferred update and write ahead logs (WAL) and discuss the problems which they cause in a virtual memory environment. Next, section IV contains a hardware proposal to overcome most of the indicated deficiencies. Lastly, Section V indicates our conclusions.

II TRANSACTION MANAGEMENT

A transaction manager provides two services, concurrency control and crash recovery. A client has three available commands:

begin xact
end xact
abort xact

The 'begin xact' and 'end xact' statements bracket a collection of read and write operations that must be

This research was sponsored by the U.S. Air Force Office of Scientific Research Grant 78-3596, the National Science Foundation Grant MCS-8211528 and the Naval Electronics Systems Command Contract N00039-78-G-0013.

serialized [BERN80] with respect to all other concurrent file activity. A client can also abort a transaction with the 'abort xact' command. This will cause all data base modifications by this transaction to be undone.

The second service is crash recovery. In the event of a system failure, a client is guaranteed that his transactions will either be performed to completion or aborted.

Concurrency control can be provided by many different algorithms, In [BERN81] a large collection of possibilities are surveyed. In this paper we assume that concurrency control is provided by either dynamic locking [GRAY78, MENA78, ASTR76], by serial validation [BHAR80, KUNG81, CERI82], or by timestamp ordering [BERN77, BERN80]. The reader is referred to [BERN81] for a discussion of these and other approaches.

2.1 Locking

There are several problems with dynamic locking in a virtual memory environment.

L1) The unit of locking cannot be smaller than a page

An operating system cannot enforce a finer granularity than one page as a lockable object. A client can read (update) any data on a page bound into his address space with read (write) permission. Hence, two clients cannot update the same page with any guarantee of consistency.

Traiger [TRAI82] suggests that page locking is unacceptable to some data managers, and therefore concludes that locking cannot be done by a virtual memory OS. On the other hand, there are several data base systems which lock at this granularity, eg. [BROW81, RTI83], and no study has definitively proved that page locking imposes an unacceptable performance penalty. Hence, we assume that page level granularity is acceptable.

L2) Only pages which are locked can be addressible.

Consider two clients, each holding a write lock to single page in the same file. If the file is in a shared data segment and both locked pages happen to be resident in main memory, then each client can read (and with hardware permission) write the page locked by the other without an interrupt transferring control to the OS. This breach of concurrency control occurs because an OS which binds a whole file into a user's address space implicitly gives a read or write lock to that client for the whole file. If page level locking is desired, then addressibility must be provided only when locking is accomplished.

This will guarantee that the cost of setting page level locks is VERY high. In current OSs several thousand instructions are required to bind a page into an address space. This compares unfavorably with current lock management components of data base systems which require a few hundred instructions to set a lock [GRAY81]. This economy results from running the lock manager in user space and directly accessing a lock table in a shared segment.

L3) Implicit OS locking may not be reasonable when accessing B+-trees [COME79].

If a data manager is using a B+-tree access method, then a hierarchical directory of pages with key values will reside in a file. Pages containing data records will be intermixed with these directory pages. Clearly, data pages must be locked on access and such locks must be held to the end of a transaction [ESWA76]. However, locking must also be provided for directory pages. Efficient schemes (e.g. [BAYE77]) do not hold locks on directory pages throughout a transaction. Hence, accesses to index pages can obey a less restrictive protocol than accesses to data pages. An OS page manager cannot distinguish between the two cases and must treat all pages equally thereby reducing parallelism.

2.2 Serial Validation

The following points apply to serial validation [KUNG81].

S1) Validated objects cannot be smaller than one page.

The page manager cannot distinguish read access to objects smaller than one page. Moreover, it cannot distinguish write access to objects smaller than one page unless it differences the old and new versions of the page. This is the same as point L1) above.

S2) All addressible pages must automatically be in a client's read set.

A page manager cannot tell what addressible pages a client has touched. Therefore, it must assume that the client has touched all of them.

The read set of a transaction should be kept as small as possible in order to maximize its chances of successfully being validated. Hence, the OS should add pages to a client's address space one at a time. Binding individual pages, as discussed above, will be VERY expensive.

S3 An Interrupt Must be Generated on the First Write to a Page

In current validation schemes the transaction manager must maintain a list of objects which have been read (Or) and a list of objects which have been written (Ow). Private copies are usually made for any modified objects which are then installed into the data base at the time the transaction is validated.

Hence, the OS must get an interrupt on the first write to a page so it can allocate a private copy of the referenced page. Special hardware is needed to detect this event.

S4) Serial validation does not work well for B+-tree index pages

A semantically consistent data structure will be produced by applying the serial validation algorithm to the index pages of a B+-tree. However, throughput can be increased by certifying transactions even if they read an index page which was written by a recently committed transaction. Unfortunately, an OS cannot know when to apply this less restrictive algorithm because it cannot distinguish index pages from data pages.

2.3 Timestamp Ordering

The following considerations apply to timestamp schemes.

T1) Granularity of objects cannot be smaller than a single page

Since there is no way for the OS to know what part of a page was accessed by a client, it cannot run a timestamp algorithm for any smaller granularity objects.

T2) Every page access must trigger timestamp processing

Consider a page which a client wishes to read. When it is made addressible, the client's timestamp is compared to the timestamp of the transaction which last wrote data on the page. If the client's timestamp is later, the read is allowed; otherwise, the client is aborted and restarted. Unfortunately, all subsequent accesses to the same page by the client must also be checked because another user could have committed an update to the page in the meantime. Hence, timestamp processing must be built into the page mapping hardware because it will be disasterously expensive to perform in software.

T3) Timestamp processing does not work well for B+-tree index pages.

The basic timestamp algorithm [BERN81] will ensure a consistent B+-tree index if applied for all reads and writes in the data structure. However, more parallelism can be obtained by a special case algorithm that allows readers of index pages to proceed even if there is a later write to that page. Of course, the normal algorithms must be run for data level pages. Since the OS cannot distinguish data and index pages, it must run the regular algorithm for all pages.

2.4 Summary

The following conclusions can be drawn from the above comments.

C1) Without hardware support specific to a given concurrency control scheme, all options appear to have substantial performance problems. In Section IV we will present a modest hardware proposal which appears to perform competitively against an application software lock manager.

C2) Concurrency control for B+-tree index pages should have a special case algorithm applied. Appropriate system calls must be provided to communicate the collection of pages for which a less restrictive algorithm can be run.

III CRASH RECOVERY IN A VIRTUAL MEMORY ENVIRONMENT

Transaction management systems use different techniques to provide crash recovery. We indicate the salient characteristics of the two main approaches, deferred update and write ahead logging (WAL).

1) deferred update

A file update is not installed directly in the file. Rather, it is stored in an "intentions list" [LAMP76, STON76]. When 'end xact' is reached, a commit flag is stored at the end of the intentions list. All pages of the intentions list must now be forced out to disk, and the page with the commit flag must go out to disk last.

The intentions list is now processed sequentially and all updates are installed. The installation procedure is carefully programmed so that it can be performed again from the beginning in the event of a system failure. After a system failure the OS is first restored. Then, a special recovery utility inspects all intentions lists. If the commit flag is present for a given list, the utility reinstalls the updates in the file; otherwise, the intentions list is discarded. A "system log" of the intentions lists of all committed transactions is often maintained to ease recovery from media failures.

The intentions list must be visible to the client program. In this way the client can see proposed changes or make multiple changes to the same data object. A clever technique to support this function is used in System R [ASTR76].

2) Write Ahead Log

Alternately, an update can be directly processed. First, a log record containing the old values which the update replaces must be created. This log record must be forced out to disk prior to the time that the file data is written to disk. The name Write Ahead Log (WAL) is derived from this requirement. The last step of a transaction is to install the commit flag in the log and force it out to disk. At recovery time the log is inspected and the updates of all uncommited transactions must be undone by a utility program.

All current recovery techniques are variants on these techniques, and we assume that a virtual memory transaction manager will choose one of these two schemes. The following considerations apply in this environment.

R1) All page modifications must be logged.

Since the OS has no way of distinguishing data pages from index pages, both must be logged in the same manner. As discussed in [TRAI82], the deferred update approach does not require that index pages be logged. Moreover, structural changes on a page (such as allocating an overflow page and updating a page pointer to reference this new page) may be recreatable at the time of recovery. Hence, they do no need to be logged by an application level recovery system. Again the OS cannot distinguish this kind of modification from an ordinary update and may make unnecessary log entries.

R2) Least Recently Used (LRU) cannot be used as a page replacement policy

With either approach to crash recovery, there are restrictions concerning the order in which pages must be written out to disk. A page manager cannot simply implement the popular LRU replacement policy.

IV HARDWARE SUPPORT FOR LOCKING

The following hardware expedites page level locking in a virtual memory transaction manager. It is similar to the hardware support which would be required by serial validation or timestamp approaches. We assume that memory mapping hardware is supported; however, its exact composition is of limited interest. Lastly, we assume that direct update is selected as the crash recovery technique.

The following bits are associated with each page presumably as part of the hardware which maps logical pages to physical pages.

don't care bit
 0 = deactivate hardware checking
 1 = activate hardware checking
write-lock bit
 0 = page does not have a write lock
 1 = page has a write lock
count field
 value = number of readers who have referenced the page
access bit
 0 = page has not been read by the current process
 1 = page has been read by the current process
update bit
 0 = page has not been modified by the current process
 1 = page has been modified by the current process

At each page reference for which the don't care bit is set to one, the hardware must perform the following algorithm:

```
IF reference is a update THEN
    IF write-lock = 0 and count = 0 THEN
        update = 1
        write-lock = 1
        generate an interrupt to log the current page
    ELSEIF write-lock = 1 and update = 1 THEN
        do nothing
    ELSE
        generate an interrupt noting the page requested
        so the process can be blocked

ELSEIF reference is a read THEN
    IF (write-lock = 0 and access = 0) or
    (write-lock = 1 and update = 1 and access = 0) THEN
        access = 1
        count = count + 1
    ELSEIF access = 1 and write-lock = 0 THEN
        do nothing
    ELSE
        generate an interrupt noting the page requested
        so the process can be blocked
```

The lock manager must get control when an interrupt occurs. It maintains the following information for each process:

process-id
vector of access bits
vector of update bits

At a task switch, the lock manager must save the vectors of access and update bits from the memory

management hardware for the halted process. Its last step is to load the memory management hardware with the vectors for the new process.

When a transaction is committed, the lock manager first forces all modified pages to disk and into the log so that recovery from media failure is possible and then sets all write-lock bits to zero. Moreover, it decrements the count for each page with an access bit of one. Then, it can destroy the two bit vectors for the committed process.

When a transaction begins, the lock manager must assign two new bit vectors initialized to all zeros. Notice, that a new transaction can have addressibility over a large collection of data base objects when it begins. It can also bind large objects into its address space as necessary.

Deadlock detection can be done by any of the currently popular schemes. Whenever a failed lock request generates an interrupt, an entry must be inserted in the "waits for" graph and the graph inspected for cycles. This latter step can either be done periodically or on every access [AGRA83].

Write ahead logging can be performed efficiently by taking a snapshot of each page before it is updated. When a modified page is written to disk, it is differenced against the snapshot, and only the old and new values of changed bits are written to disk. This scheme will result in a more compact log than would result if the entire old and new pages were written to disk. In addition, the log is approximately the same size as a record oriented log managed by application software.

It should be noted that 16 concurrent readers can be supported in this scheme with 8 bits per page, 4 for the count and 1 each for the other fields. Because the page table already holds minimally a presence bit and a physical page address (typically 16 bits on current hardware), the page table must be expanded from two bytes to three bytes per entry. As a result, there is a 50 percent space overhead.

The two bit vectors suggested above contain the read and write sets for each transaction. Since serial validation algorithms use this information, these vectors may be useful for other concurrency control schemes. However, the count and write lock bits are particular to locking schemes.

Some systems implement virtual memory by a hashing scheme (e.g. the IBM System 38 [IBM78]). The above algorithm can be incorporated easily into this architecture. The only complication is that the vectors of access and modify bits are somewhat more difficult to save and restore.

Standard private data segments can be supported using this hardware scheme. The lock manager need never save or restore the update or access bits if there is only a single user.

However, read-only code segments present a problem because they require shared read access while denying write access. This feature can be supported by a "read-dirty-data" bit. With the inclusion of this extra bit, a read only segment is bound into a user’s address space with both the write-lock and read-dirty-data bits set for each page. An added advantage of this extra bit is that hardware support for level 1 consistency [GRAY78] is automatically provided by setting the read-dirty-data bit for all pages that a transaction might read.

V CONCLUSIONS

We have proposed a hardware scheme that overcomes some of the performance problems associated with OS virtual memory transaction management.

Additional OS software must save and restore the access and update bit vectors at each task switch and perform deadlock detection. Reasonably simple hardware can do the remainder of the work. This scheme should perform comparably to current software-only DBMS algorithms that run in user space.

Although page level logging is required, the log can be easily compressed to a size nearly equal to those produced by current DBMS logging schemes. The overhead to perform this compression may make an OS scheme use more CPU time than current techniques.

There are three remaining drawbacks. First, page level locking may reduce parallelism in high volume applications. The performance consequences of page level granularity on very large data bases are presently unknown. Traiger [TRAI82] also notes that page level locks are very inefficient for B+-tree index pages. However, this inefficiency will only occur if a transaction changes the key value of a record, inserts

a new record into a data base that causes a B+-Tree index to split or deletes a record which causes a B+-Tree underflow. In a well designed data base these events may be relatively infrequent, and the performance consequences of page level locking may not be severe.

The second drawback is the absence of a "fairness" provision. A transaction that wishes to write a heavily accessed page may wait indefinitely. As long as new readers appear to keep the count above zero, it will remain blocked. A software transaction manager can easily implement a scheme that will deny read locks to any transaction if there are any waiting writers. It would require another hardware bit to implement this tactic in conjunction with the algorithm in Section IV.

The final drawback concerns presetting of write locks. A data base update is typically made by first reading a page and then writing it. Hence, a read lock would be obtained first followed by a write lock. On pages that are sure to be subsequently written, it is more efficient to obtain a write lock on the first lock request and avoid the necessity of a subsequent upgrade which could increase the chance of deadlock. Directly setting write locks would have to be done by a user immediately rewriting accessed data to set the write-lock bit. This mild inconvenience may not be tolerated by all clients.

REFERENCES

[AGRA83] Agrawal, R., et. al., "Deadlock Detection is Cheap," SIGMOD Record, January, 1983.

[ASTR76] Astrahan, M., et. al., "System R: A Relational Approach to Data," TODS, June 1976.

[BAYE77] Bayer, R. and Schkolnick, M., "Concurrency of Operations on B-Trees," IBM Research, San Jose, Ca., RJ1791, May 1976., ""

[BERN77] Bernstein, P. A. et. al., "The Concurrency Control Mechanism of SDD-1: A System for Distributed Data Bases," Computer Corp of America, Cambridge, Mass., December 1977.

[BERN80] Bernstein, P. et. al., "Concurrency Control in a System for Distributed Data Bases (SDD-1)," TODS, March 1980, pp 18-51.

[BERN81] Bernstein, P. and Goodman, N., "Concurrency Control in Distributed Data Base Systems," Computing Surveys, June 1981, pp185-222.

[BHAR80] Bhargawa, B., "An Optimistic Concurrency Control Algorithm and Its Performance Evaluation Against Locking Algorithms," University Of Pittsburgh, Pittsburgh, Pa., June 1980.

[BROW81] Brown, M. et. al., "The Cedar Database Management System," Proc. 1981 ACM-SIGMOD Conference on Management of Data, Ann Arbor, Mich., June 1981.

[CERI82] Ceri, S. and Owicki, S., "On the Use of Optimistic Methods for Concurrency Control in Distributed Data Bases," Proc. 6th Berkeley Workshop on Distributed Data Bases and Computer Networks, Pacific Grove, Ca., February 1982.

[COME79] Comer, D., "The Ubiquitous B-Tree," Computing Surveys, Vol. 11, No. 2, June 1979.

[ESWA76] Eswaren, K., et. al., "On the Notions of Consistency and Predicate Locks in a Data Base System," CACM, October, 1976.

[GRAY78] Gray, J., "Notes on Data Base Operating Systems," in Operating Systems: An Advanced Course, Springer-Verlag, 1978, pp393-481.

[GRAY81] Gray, J., "Experience With the System R Lock Manager," unpublished collection of notes.

[IBM78] IBM Corp., "IBM System/38 Technical Developments", IBM, White Plains, N.Y., IBSN 0-933186-03-7, 1978.

[KUNG81] Kung, H. and Robinson, J., "On Optimistic Methods for Concurrency Control," TODS, June 1981, pp 213-226.

[LAMP76] Lampson, B. and Sturgis, H., "Crash Recovery in a Distributed System," Xerox Palo Alto Research Center, 1976.

[MENA78] Menasce, D. A. et. al., "A Locking Protocol for Resource Coordination in Distributed Systems," TODS, June 1980, pp 103-138.

[MITC82] Mitchell, J. and Dion, J., "A Comparison of Two Network-Based File Servers," CACM, April 1982.

[ORGA72] Organick, E., "The Multics System, An Examination of Its Structure," MIT Press, Cambridge, Mass.

[REDD81] Redell, D. et. al., "Pilot: An Operating System for a Personal Computer," CACM, February 1981.

[RTI83] Relational Technology, Inc., "INGRES Reference Manual," 1982

[SPEC83] Spector, A. and Schwartz, P., "Transactions: A Construct for Reliable Distributed Computing," Operating Systems Review, Vol 17, No 2, April 1983.

[STON76] Stonebraker, M. et. al., "The Design and Implementation of INGRES," TODS 2, 3, September 1976.

[TRAI82] Traiger, I., "Virtual Memory Management for Data Base Systems," Operating Systems Review, Vol 16, No 4, October 1982.