Locking

Jim Gray
University of California, Berkeley, California

This paper considers some pragmatic issues involved in providing locks for a computer operating system.[1] Particular problems touched are:

(a) locking primitives,

(b) enforcing lockout,

(c) scheduling conflicting requests for a lock,

(d) error handling for a process controlling a lock,

(e) a discipline for preventing deadly embraces,

(f) minimizing the time key system tables are locked (this interacts with page faults and interrupts).

To set the stage we propose that the universe of the operating system consists of objects of various types (e.g. processes, files, teletypes).[2] Objects are accessed via capabilities. A capability for an object is an access path which selectively allows certain operations to be performed on the object. For example, a capability for a file may or may not allow a process to read, write, execute, lengthen, or destroy the file. The actions allowed by a capability are called the options provided by the capability.[3]

A process is an object. It consists of a virtual-processor state, a list of capabilities (C-list), and some historical information. The domain of a process is the set of all objects accessible either directly or indirectly through its C-list.

If processes intend to share objects and no sharing process intends to modify the object in such a way that the object is momentarily in a bad state, then capabilities provide an excellent mechanism for sharing the object. One simply places the appropriate capability for the object in the C-list of each sharing process. Examples of such objects are read/execute-only files, clocks, and mail boxes. If, on the other hand, two processes may overlap[4] one another in modifying an object, then it is necessary to

---

[1] This paper benefitted from discussions with Butler Lampson.

[2] The papers by Dennis and Van Horn [1] are prerequisite to much of what follows.

[3] This extension of capabilities to carry options is not mentioned by Lampson [4] but is implemented in the system described in [5].

[4] Two processes are said to overlap if their concurrent execution results in a change in an object different from the change caused by either process A preceding process B or process A executing after process B.

provide some mechanism which allows each process to know when an object is stable and when it is unstable due to modification by some other process.

Dijkstra and Habermann [2, 3] present semaphores as a solution to this problem. A semaphore is a flag. An attempt to set an already set semaphore causes the setting process to block until the flag clears. It is sufficient that operations on semaphores cannot overlap.

Semaphores are used roughly as follows. Each process is given a capability for the object to be shared, for a semaphore associated with the object, and for operations which set and clear the semaphore. The processes agree to the following rules:

(rule 1) Each process will set the semaphore before touching the object. If the semaphore is already set this will cause the process to block until the semaphore is clear.

(rule 2) Each process agrees not to leave the semaphore set indefinitely. The process will clear the semaphore as soon as the process finishes touching the object.

It appears that semaphores allow a community of processes to communicate in arbitrarily complex ways. (Can this be formalized and proved?)[5] The remainder of this section argues that a more elaborate locking facility is needed. It proposes that a lock scheduler be associated with each lockable object. Its task is to lock the object, resolving conflicting requests and handling error conditions. Locking an object yields a capability (with the desired options) and clearing the lock returns the capability to the scheduler. Benefits of this scheme are:

(1) the ability to handle elaborate forms of concurrent access to objects,

(2) more reliable lockout,

(3) error handling, and

(4) prevention of deadly embraces.

The Need for a Scheduler

Observe that setting a lock is a request for a resource. Looked at in this light one expects to see some form of scheduler in the locking mechanism. Often the scheduling

[5] Both Dijkstra and Habermann are quite concerned with proving that their algorithms work. They indicate that this is a first step towards proving that operating systems work. However, except for the resource allocation problem (proving optimality), the definition of working is extremely complicated. No formal proof seems feasible in this context. The best one can hope for is that the style and architecture presented by these authors, and incarnate in the T.H.E. operating system, are amenable to convincing arguments that certain facilities are well behaved.

task is non-trivial. For example, one may be willing to allow other processes certain types of concurrent access to the object while the lock is set.

The classic example is an accounting file. Processes reading the file may share it concurrently. However, a process requesting write access to the file blocks until all processes currently reading have released the file. No new access requests to the file may be honored until the writing process has completed.

This simple (common) scheduling task is implemented by placing a program (scheduler) between the process and the file. Each process is given the capability to call the scheduler. Setting the lock amounts to calling the scheduler with a parameter specifying the desired class of access. The scheduler returns a capability for the object (with the appropriate options) when the lock is set. Clearing the lock returns the capability to the scheduler.

The need for a scheduler is even more apparent when one allows for priorities among processes or requests (the above simply allows concurrency). In this case a request for locking the printer might be "I would like to print 10000 lines in the next hour."

The scheduler may return control to the process if the lock is busy thus allowing it to compute on some other problem. It can notify the process by sending it an event or interrupt when the object becomes available.

There is one detail here: one cannot allow the capability returned to the process by the set operation to be copied from one C-list to another. Otherwise there will be little hope of the clear operation scavenging all reincarnations of the capability. This problem is settled by turning off the option which allows the capability to be moved. The process may still pass the capability to other processes by passing the entire C-list.[6]

Advantages Over Semaphores: Paranoia

Semaphores are for cooperating, completely debugged processes. All processes continually have the capability for the shared object in the discussion above. They voluntarily test and set the lock. Since possession of the capability is prima facie proof (to the system) that the process may access the object in the specified ways, the system cannot enforce rules 1 and 2 above. Nor can it be expected to detect violations of these rules. If something else is intended, the concept of capabilities must be extended. Lock schedulers are such an extension.

---

[6] In discussion it was suggested by several people (Denning, Miller) that one must be able to withdraw capabilities from a process. The similarity to Wilke's scheme of tickets was mentioned. This is easily implemented if the capability cannot reproduce itself. If, however, the capability can be copied from one C-list to another it is difficult (expensive) to track down all its reincarnations. The idea of ripping up the ticket breaks down here because one can't rip up all xeroxes of it. If one is willing to rip up all tickets for the object, one need only destroy the object itself. This is equivalent to cancelling the show. However, this may adversely affect other (non-offending) processes.

The lock scheduler enforces rule 1 above since the shared object enters the domain of the sharing process only after a specific request to the scheduler. Further the scheduler re-possesses the object when the process unlocks it. A bonus in this arrangement is that it minimizes the time that the shared object is in the domain of a potentially undebugged process. (We assume the scheduler is reliable.)

There seems to be little hope of enforcing rule 2 above unless the processes can agree on a time lock. If this be the case, the scheduler can initiate error processing if some process holds the lock beyond some real-time interval.

Error Handling

If a process which controls a lock attempts to destroy itself (log-out) or if the system is about to abort the process because of some error condition, some aid must be given to other processes waiting for that lock. Toward that end the names of all locks controlled by a process are saved in its state. This information is updated by the schedulers and is accessible to the system. In case of emergency the appropriate lock schedulers are notified that a process controlling that lock has been aborted. They can take diagnostic action as required (for example return to locking processes with the error message "JOHN DOE ABORTED LEAVING LOCK SET"). This contributes to the complexity of schedulers and argues further for their incorporation in the system.

The scheduler also maintains a list of processes controlling the lock on an object so it can answer the question "The lock has been set for 10 minutes, who forgot to clear it?"

Interlocks

Suppose there are two locks, L1 and L2. If process P1 has L1 and P2 has L2 and each wants the lock the other has as well, then we have a deadly embrace. These daisy chains of locks can get arbitrarily complex and there seems to be no economical way of detecting them.

Throughout this paper we have spoken of locking objects as though this were the designated goal of locking. We have adopted this vocabulary because it was convenient. It is generally the case, however, that one locks structures rather than objects. These structures are not necessarily made up of objects. They might be list structures within a file or the set of all teletypes. It may happen that a structure is an object but if this does happen it should be viewed as an accident. Further, locks are typically hierarchical. One may want to lock a teletype, all teletypes, all slow peripherals, ...

In order to solve the daisy chain problem we impose the following restrictions on structures which are to be locked:

(1) Any pair of lockable structures must be disjoint or else one must contain the other properly.

(2) Whenever a process has shared objects there must be a lock for the entire structure.

Restrictions (1) and (2) impose a partial order on the locks and they require that there be a master lock. That is, they induce an upper rooted tree. This tree is defined by the programmer. If the programmer is writing a high level language, he needs only speci specify which structures and substructures he wants locked and the translator can enforce the hierarchy.

The scheduler functions as follows:

(1) Process P1 calls lock scheduler S for lock L1 .

(2) S checks P1's priority and places it in the queue for L1 if P1 wasn't successful in setting L1 .

(3) In any case if L1 has subordinate locks, S returns to P1 with the remark that P1 is in the queue.

(4) When P1's turn comes ~~Before returning to P1, however,~~ S requests all subordinate locks. (It need only request direct subordinates.)

(5) When all subordinate locks alert S that they are set for it, S alerts P1 . In particular if L1 has no subordinates, S alerts P1 immediately.

Unlocking is done similarly. In this scheme, the scheduler does all locking and unlocking for the process.

Summary

So far we have been considering a general facility for locking. We have argued that a scheduler which maintains certain diagnostic information (and is either a separate process or an execute-only file shared by processes) is needed to relieve the subsystems writer from the complex and error prone task of setting, clearing, and scheduling locks. This also allows for a certain amount of paranoia among the subsystems writers. Subsystems simply call the scheduler (passing it some parameters) and the scheduler does the rest.

One might argue that one should write separate schedulers for each new application. However, the similarities between such schedulers are greater than their differences. In particular the error handling mechanisms will be similar and be quite complex. The main variation will be in the scheduler proper. It appears that this can be made a "pluggable" module in a general lock scheduler matrix provided by the system. The system will provide garden variety schedulers and users may write more elaborate schedulers at their leisure.

Hardware Locks

The discussion above has centered on locks in a rather abstract setting. The overhead of setting or clearing such a lock might be a few hundred instructions. Clearly a more efficient and limited locking facility is needed to do simple things. In particular note that there must be locks on the queues and flags used by the locking mechanism and schedulers.

The hardware is presumed to have two instructions:

LOCK i

CLEAR i

which set and clear the i-th lock respectively. Locks may be implemented as pseudo-memory banks and the hardware that resolves bank conflicts allocates these pseudo-banks to processors. A processor executing LOCK 100 pauses until bank 100 is free and then it continues. It monopolizes this pseudo-bank until it executes a CLEAR 100 .

Now consider the interaction between these locks, interrupts, and page faults. Suppose the system is setting a lock for a low priority process and an interrupt comes (external, or quantum runout). Then the lock will stay set until the process is swapped in again. If this is a critical lock, the interrupting process may not be able to function because it needs this lock to run. This will seriously degrade system response to interrupts.

So when a processor executes a LOCK , interrupts must be locked out. To put a ceiling on the interrupt response time the interrupt lock must have a time limit. If the processor does not CLEAR the lock before N (N~25) microseconds, the locking process is trapped, the lock cleared, and interrupts re-armed.

This guarantees that if a process wants to manipulate a table, it is given N microseconds to do it.[7] During that time no other processor (process) will touch that table since the processor will first try to get the lock and thus will pause.

Interrupts are locked out, but suppose the process has a virtual memory (not physical). In particular suppose that the tables and system code are not locked into fast memory. (If they are this is not a problem but lots of other problems suddenly appear.) Then the process can still be interrupted by instruction or data fetches which touch segments (pages) not in fast memory. In general it takes more than N microseconds to bring these pages in, so the timer may run out.

One approach might be:

[7] Note: a processor can only set one lock since setting the second lock might take more than N microseconds.

(1) to go off to the working set computer and tell it to keep all pages touched until further notice

(2) then to touch all necessary pages (including instruction pages)

(3) then to set the lock on the data

(4) change it

(5) release the lock

(6) and set the working set computer loose.

The overhead in doing this (informing the working set computer) is considerable. Also the data may change while pages are being touched but before the lock is set. So the computation of which pages are needed may be out of date. (Suppose one is locking the master object table. While touching needed objects a garbage collection strikes and moves some objects. Then the touched objects will not necessarily be on the touched pages.)

Similar problems lie in locking single pages of the table. Also this leads to an expensive proliferation of such locks.

A simpler and more workable scheme is to have the systems programmer worry about page faults due to instruction fetches. Data fetch page faults are handled as follows:

(a) The SET instruction saves the instruction counter of the processor in a word in the processor state. It inhibits interrupts and sets a 25 microsecond timer.

(b) If a page fault occurs, the processor clears the lock, and the instruction counter of the process is reset to the SET instruction.

In programming one uses this instruction as follows:

(1) set the lock

(2) touch all needed cells

(3) make all changes

(4) clear the lock.

It is clear (one can prove) that since interrupts are inhibited and since a page fault in step (2) causes the process to restart at step (1), then when one reaches step (3) all touched pages are in core. Further, the lock was set before the pages were touched so they are the pages needed for step (3).[8]

[8] It is possible to have the compiler (systems programming language) generate these extra fetches and insure that the code lies within a single page, and that it executes within N microseconds.

This scheme is inelegant but workable. It has been adopted for the BCC Model-1 computer. It does not appear that changes in technology will make these problems go away. As memory becomes cheaper, system tables become larger. In fact the tables grow faster than the memories. It is doubtful that the practice of locking large segments into the top level of a memory hierarchy will be able to keep pace with this growth.

References

1. J. B. Dennis and E. C. Van Horn, Programming semantics for multiprogrammed computations. Comm. of the ACM, Vol. 9, No. 3 (March 1966), pp 143-155.

2. E. W. Dijkstra, Cooperating sequential processes. In Programming Languages, F. Genuys, Ed., Academic Press, New York 1968.

3. A. N. Habermann, On the Harmonious Co-Operation of Abstract Machines. Ph.D. Dissertation, Technical University of Eindoven, the Netherlands, 1967.

4. B. W. Lampson, Dynamic protection structures. AFIPS Conference Proceedings, Vol. 35, Fall 1969, pp 27-38.

5. B. W. Lampson, An Overview of CAL-TSS. Computer Center, University of California, Berkeley,