PROCESS SYNCHRONIZATION WITHOUT LONG-TERM INTERLOCK

William B. Easton
Applied Logic Corporation
Princeton, New Jersey

Abstract: A technique is presented for replacing long-term interlocking of shared data by the possible repetition of unprivileged code in case a version number (associated with the shared data) has been changed by another process. Four principles of operating system architecture (which have desirable effects on the intrinsic reliability of a system) are presented; implementation of a system adhering to these principles requires that long-term lockout be avoided.

Introduction

An important feature of modern multiprogramming systems is the ability to allow independent, concurrently executing processes to work on the same data base. If a process needs to modify a data base which is being shared by other processes, then some means of synchronization must be provided to prevent chaos when two processes attempt to overlap each other in accessing the data. The usual means for providing such synchronization is to lock out all other processes while one process is modifying the data. We propose, instead, that (with careful system design) access can be allowed to all processes at all times, and that a process can determine whether the data have been modified by another process and take corrective action.

The proposed technique, in brief, is to provide each shared object with a version number, to remember the version number prior to making a decision about modifying the object, and then, when the actual modification takes place, to compare the version number with the remembered one, to change the version number, and to perform the modification, all "in a single instant of time." This technique is somewhat inelegant, in that a process may be forced to repeat work it has already done. However, the system's scheduling mechanism can be greatly simplified, since it need not be concerned with such matters as the release of shared resources. Furthermore, the description of the state of the entire multiprogramming system at any point in time is made simpler, since no process is ever interrupted while it is part way through modifying critical data. The additional cost entailed by occasional repetition is readily made small enough in an actual implementation.

An Example

Suppose that a process desires to create a new file of a certain name in a given directory, and that the newly-created file must be the only file with that name in the directory. We will examine the interlock requirements for this guarantee.

Let a directory be a file whose records (entries) each contain the external name of a file and further information about the file (such as pointers to the data, time of creation, and so on). We assume that a directory may, in principle, be arbitrarily large, so that operations on it may involve an undetermined number of physical data transfers to and from a mass-storage device. In order to create a new file, a process must search the directory to determine that there is not already a file with the same name; it must then insert, in an unused entry position, the desired name and whatever further information is required (the file is empty, its creation time is right now, and so on). Of course, some or all of these operations will be done by privileged system code.

The usual implementation of file creation, using interlocks to provide one-process-at-a-time access to the directory, is shown in Figure 1.

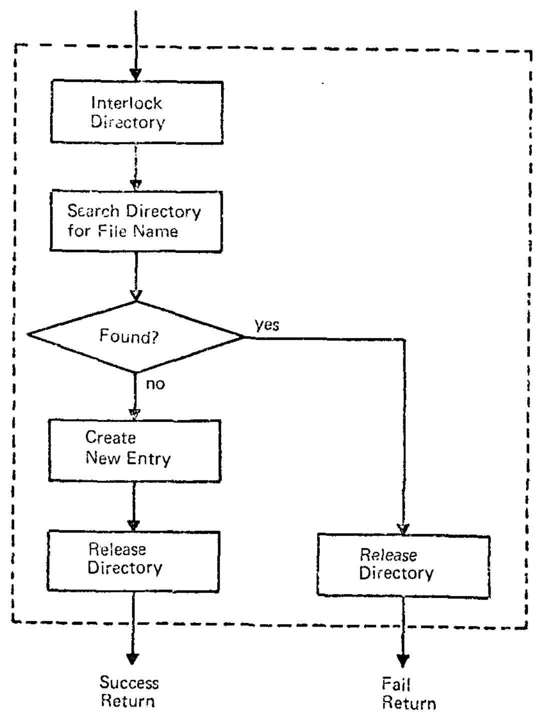

Fig. 1. File creation with directory interlock.

During the entire operation, the directory is inaccessible to other processes. This privileged status is shown by a broken line around the flow chart; the operations within the broken line are done "in a single instant of time," as far as processes accessing the directory are concerned. The directory-searching operation may require the transfer of data from mass storage; the central processor will, in general, turn its attention to other processes while waiting for the completion of such transfers. Since these other processes may also wish to use the directory in question, these processes may become blocked and action must then be taken to awaken them when the directory is available. Furthermore, the system's scheduler must provide that the process currently using the directory retain enough priority to quickly release the directory; otherwise, access to the directory by other processes will be delayed.

The implementation of file creation using the version-check technique is illustrated in Figure 2. Prior to searching the directory, the process fetches and remembers the directory's version number. After the search operation is complete, the remembered version number is compared with the current one; if they differ, the directory has been modified by another process and the search operation must be restarted. (Note that the remembered version number need not itself be a protected object, since the unprivileged program can gain no additional power by falsifying it.)

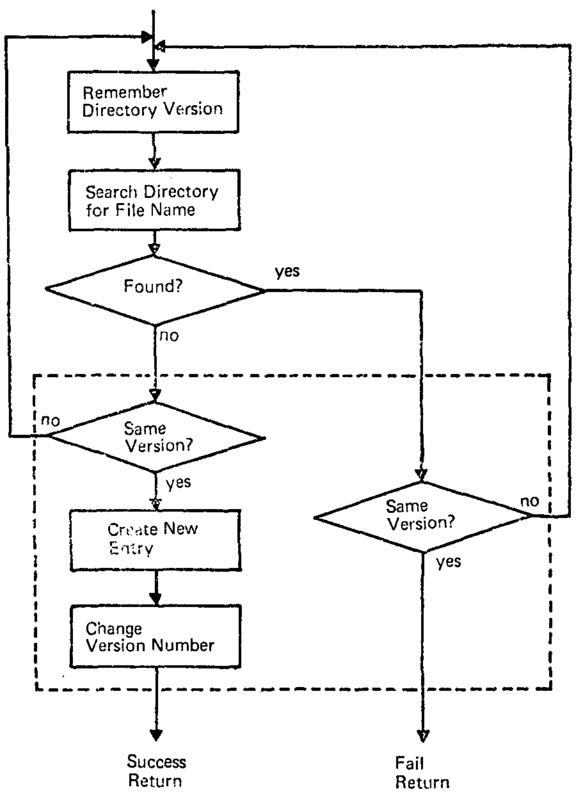

Fig. 2. File creation using the version-check technique.

Again, operations which must be done "in one instant of time" are enclosed in broken lines. The directory search operation has been removed from the critical section; instead, the entire search operation is to be restarted if the directory is modified by another process during the search.

It is still the case that several operations must be done "all at once." There is, however, an essential difference in the required interlock. If the location of a free record position is known, the creation of the new entry requires references to a small, fixed number of pages. If these pages are brought into core before the version check and if they are guaranteed to remain in core during the creation of the entry, then the updating of the entry can proceed at central-processor speeds without waiting for mass-storage operations. Thus, the system may prevent the central processor from being taken away from the process for the small number of microseconds required to complete the entry creation; in turn, it becomes possible to implement the required interlock by looping on a test-and-set instruction without involving the scheduler.

The technique just described does provide the required synchronization between processes since (in this example) the directory is guaranteed not to have been modified between the start of the search and a successful version check. Furthermore, this synchronization is provided in such a way that no process is denied access to the directory because another process is using it.

There are, however, two important costs associated with the version-check technique. First, processes must make decisions on the basis of data structures which may be changing in time; some means must be provided to avoid erratic behavior on the part of a process because it happens to look at such a data structure at a bad time. Second, there is the potential of much "useless" repetition of unprivileged code because several processes are "fighting for" a particular data structure.

Preliminaries

A process is an object which consists of a virtual-processor state, a description of an address space, and some historical information. [2, 5, 9, 11, 12] Multiprogramming is a technique for sharing a (normally) smaller number of physical processors among a larger number of processes.

We will assume that the address space of a process consists of pages of fixed length which may or may not be shared with other processes. These pages may be organized into segments or files; we will use the terms "segment" and "file" interchangeably to refer either to named collections of data normally kept on mass storage devices or to data which are directly accessible by a process in execution. A page is assumed to have an existence as a collection of data which can be described by some unique name known to the system; this existence is independent of particular copies of the page in core storage or on a mass-storage device. Thus, for example, the same page may, at different times, exist only in core storage, only on a mass storage device, or in both of these places.

A page is said to contain (system) critical data if it contains data required for the proper functioning of the multiprogramming mechanism or the system's protection facilities. Examples of critical data are the contents of the processes' state vectors or the contents of a directory. A process is said to have privileged status if it can modify critical data directly; a process is said to have unprivileged status if it does not have this power. By a privileged procedure, we will mean a procedure being executed by a process with privileged status.

Note that the distinction between privileged and unprivileged status is not necessarily the same as the distinction between an all-powerful supervisory state and a limited problem state--a process executing in a hardware state that prevents it from directly executing I/O commands may nevertheless have privileged status if it has critical data in the writable part of its address space. Note, too, that we have not set the boundary between privileged and unprivileged status as the only protection boundary in the system; good system design practice will certainly include "firewalls" separating programs with different requirements operating in both statuses. The same process may have either status at different points in time.

The way a procedure operating in unprivileged status can cause a change in critical data is to enter privileged status (by some protected means provided by the system) with a request for a desired modification; a procedure operating in privileged status will then check that the specific modification is permitted and then do the operation before returning the process to unprivileged status.

A Philosophy of System Architecture

A common theme in many papers on the design of multiprogramming systems is the ability to limit the privileged operations a process can perform to the minimum required for correct functioning of the process. We believe that this sort of limitation is a necessary condition for a system to be capable of continuous operation for periods of weeks or months in spite of occasional hardware malfunctions and evolutionary modification of software. Indeed, we would push such limitations to the extreme. Specifically, we would insist that processes be allowed to execute in privileged status only if they observe the following requirements:

(1) No operations of a decision-making or strategy-determining nature will be done in privileged status.

(2) No process will ever be removed from a physical processor while it has privileged status.

(3) No process will return to unprivileged status while any critical data which it has modified are in an inconsistent state.

(4) No unprivileged process may prevent access by any other process (which would otherwise be entitled to such access) to any critical data.

Adherence to these principles can provide a number of advantages in the intrinsic reliability of the system, but imposes a severe discipline on the system architecture.

The requirement that no strategy-determining operations take place in privileged status is an expression of the desire to provide a process with as little power as possible. A strategy decision which requires the examination of critical data but not its modification can be made without privileged status, and, therefore, should be. Implementation of the decision may then be done by a privileged procedure which will check that the requested action is permitted. The decision process will be restarted if the version check indicates that its decision has been invalidated by another process' modification of the critical data.

An example of a decision which can take place in unprivileged status is the decision to assign a block of physical core to a page. A procedure without privilege can decide that a particular page is to be assigned to a particular block of core; such a decision can be made by examining the current contents of core, the status of processes wishing to access the page, and the status of mass-storage devices. To implement the decision, a request is made to a privileged procedure. That procedure checks that the page exists, that there is not already a copy in core, and that the requested location is available for assignment.

The decision to assign physical core to a page may involve complex algorithms which take into account such factors as the sizes of the working sets of various processes and various parameters relating to the use of core by processes. Furthermore, such algorithms are likely to be changed as part of the process of system evolution. The actual operation of assignment, on the other hand, requires only a small number of machine instructions to check the status of the page and the physical core location and to perform the actual assignment. Presumably, one would prefer not to allow a program written by any user of the system to directly cause the assignment of real core to pages; such a restriction on a user's program can easily be provided within the above framework. Note, however, that the consequences of such an assignment by a user's program (if it were permitted) would affect only the efficiency of the system and not the integrity of critical data.

The requirement that no process ever be removed from a physical processor while it has privileged status precludes awaiting the completion of physical I/O operations within a privileged procedure. Furthermore, it requires all interlocks within the system to be <u>short-term</u> interlocks (which can be implemented by looping on a test-and-set instruction) rather than <u>long-term</u> interlocks (which require that a process be unscheduled when the interlock is already set and scheduled again when the interlock is cleared).

Finally, the requirements that no process ever return to unprivileged status while part way through modification of critical data or while preventing access to critical data by other processes allows the system's scheduling and resource-allocation mechanisms to not be concerned with the need for a particular process to complete action on critical data.

Implementation of the Philosophy

The implementation of the philosophy described in the preceding section implies a number of restrictions on the system architecture. We describe below a means of implementation which adheres to this philosophy.

Whenever a change to critical data is to be made, a process will enter privileged status from unprivileged status. In privileged status, prior to any modification of critical data, the process will check

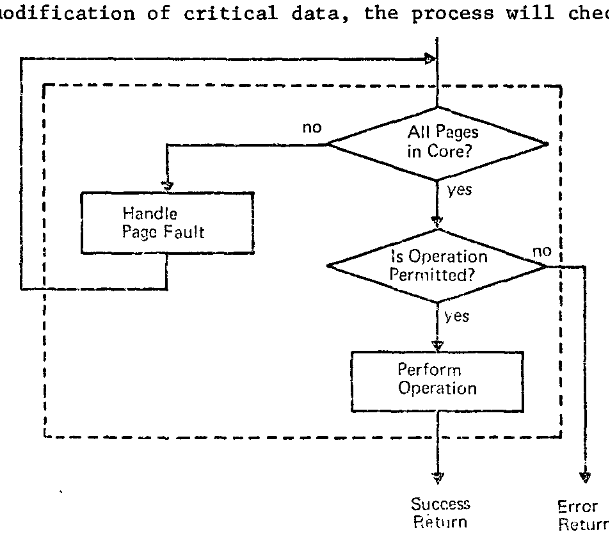

Fig. 3. Flow chart of a privileged procedure.

that the operation is permitted and that all pages required are in core and accessible. If either of these conditions is not satisfied, the process will return to unprivileged status with an error indication in the first case and after handling a page fault in the second case. If the conditions are satisfied, the required changes will be made and control returned to unprivileged status. This is illustrated in Figure 3 (on the preceding page).

Since all actions take place at central-processor speeds, it is reasonable to allow privileged operations implemented in this way to run to completion. The status of a process at any time when it can be interrupted is always known: Either the privileged operation has not yet been started, or it has been completed.

The remaining requirements are satisfied if long-term interlocks are prohibited and short-term interlocks are permitted only for the duration of a privileged procedure. To allow for the possibility that critical data may have been changed between the time a decision was made and entry to a privileged procedure, we attach to each shared object a version number (which may be a number maintained by the system and incremented on request or a set of unique bits provided by a system timer [8, 13]). Figure 4 illustrates the unprivileged decision to modify system data and its relationship to the privileged procedure doing the modification, with checking of version numbers.

When the version check has been passed, it is guaranteed that no modification to the critical data has occurred since the beginning of the current process' decision to modify it; it is further guaranteed that the version number will be changed before another process can pass the version check.

In describing the implementation of our four principles, we have ignored their relationship to the system's protection mechanism. If decisions on protection require searching of hierarchies of files to determine if a particular action is allowed, then it is clearly not possible to satisfy our criteria. Several authors [5, 7, 10, 14] have suggested an implementation of protection using protected names called capabilities or access keys retained in a privileged data structure associated with a process; using such a technique, checks for protection can also be implemented in a few machine instructions.

Finally, we believe that all of the privileged procedures which are really necessary for a general-purpose multiprogramming system can be implemented in two or three thousand machine instructions. Such an amount of code--if it is entirely made up of small, simple procedures--can be completely tested in a reasonable time and can be expected to remain relatively static since the mechanisms which change with time in a system are its complex strategy algorithms and not its basic data structures.

A Possible Hardware Implementation

The flow chart in Figure 4 uses a software interlock (with a test-and-set instruction) to provide one-at-a-time execution of a privileged procedure. Lampson has suggested the implementation of a special "protect" instruction [9, 11] which would guarantee that a small sequence of instructions is executed without interruption and would cause operation to be restarted at the protect instruction in case of a page fault. A possible implementation in hardware of the entry to a privileged procedure is to include as part

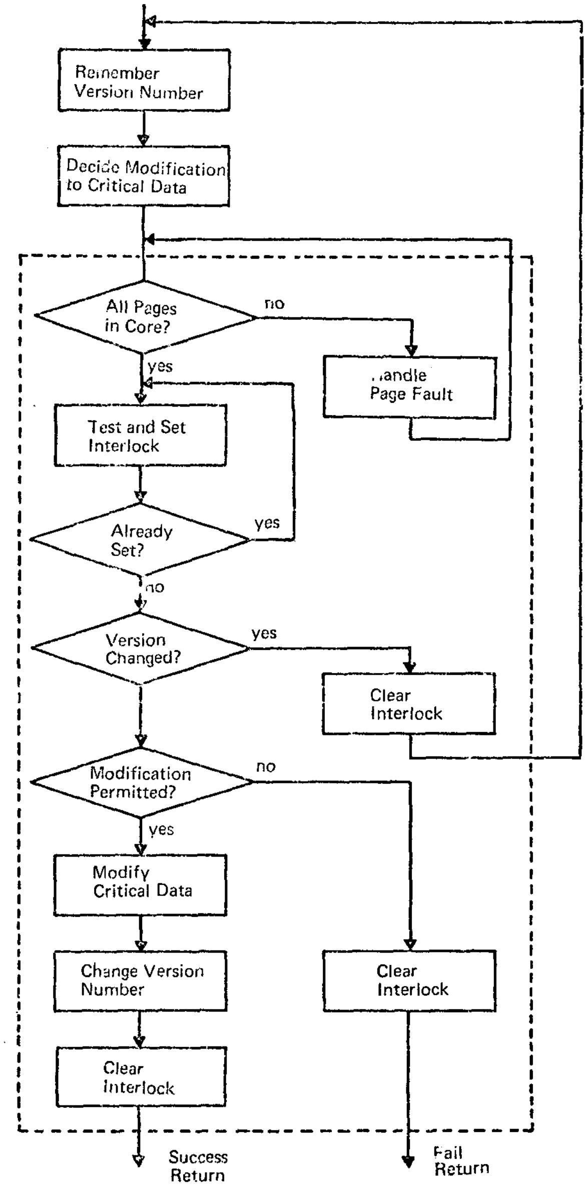

Fig. 4. Relationship between a decision to modify critical data and the actual modification.

of the access rights to a segment the right to enter privileged code in that segment; transfer to the segment at any location containing a special "entry" instruction would cause execution to continue in privileged status, with the entry instruction acting like Lampson's protect instruction. [The idea of using a special instruction to mark an allowed entry point in an execute-only segment was suggested to the author by Alan Kotok.]

Accessing a Changing Data Structure

The technique that we have described requires that a process access a data structure for the purpose of making a decision with no assurance that the data structure will not be changed at an arbitrary time. The version check avoids the possibility that a resulting incorrect decision will be implemented. There is still the possibility that the process will behave erratically because of data fetched at an inopportune moment. For example, the process might fetch a number from the data structure which it believes to be an address in some segment but which has been changed by another process to be a portion of some alphabetic text; the use of the number thus obtained could, in turn, cause the process to destroy its own data or to loop forever.

The version-check technique provides a solution to this difficulty. The process need only check that the data structure has not been changed whenever it fetches data which, if incorrect, could cause erratic behavior. The data structure has been changed if the version number has been changed. In addition, the data structure may have been changed by another processor executing in privileged status which has not yet updated the version number.

The implementation of the version check is shown in Figure 5. Prior to checking the version number, the process guarantees that at some time after the data were fetched, no processor was in the midst of modifying it; if the version check is then passed, it is guaranteed that the data structure had not been modified at the time the data were fetched.

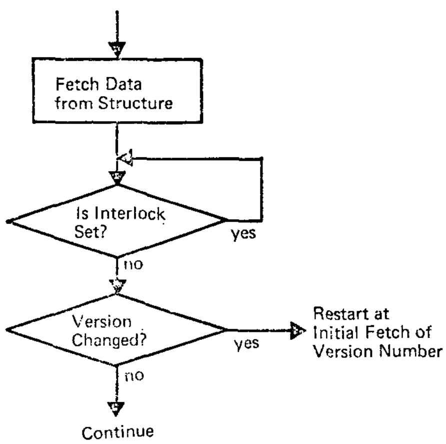

Fig. 5. Checking the version number when accessing a possibly changing data structure.

If there is available a segmentation mechanism which allows the interlock and the version number to be accessed directly, the two decisions in Figure 5 would normally require one or two instructions each. We have allowed the process to loop while waiting for the interlock to be cleared; we would expect that, in a typical system, the expected waiting time would be of the same order as the overhead of changing processes.

We note in passing that the check of the version number in our original example (see Figure 2) prior to a fail return was of the same nature as the version check after fetching data during the directory search; the version check could thus be removed from privileged status in Figure 2 if the process waits until the interlock is cleared prior to the version check.

We have shown the updating of version numbers following the modification of critical data. Alternatively, the version number could be updated prior to the modification. It would then be necessary to test that the interlock is not set after fetching the version number and to fetch the version number again if the interlock is set. In this case, the status of the interlock need not be tested when checking the version number after fetching of data.

The Overhead Cost of the Version-Check Technique

The technique of checking version numbers to determine if a decision process should be restarted costs some system overhead in that code is repeated "uselessly." One can imagine a process cycling forever, always unlucky in performing a desired privileged operation. We claim that this is unlikely, provided that the paging algorithm normally retains in core several of a process' most recently referenced pages.

Suppose that we are dealing with a system with one central processor. Upon entry to a privileged procedure, there may be a number of page faults. The code repeated on reentry to the procedure will presumably be small compared to the overhead of changing processes while pages are brought in. The real potential for additional overhead is in the repetition of the decision process. With an implementation of a working-set paging algorithm, [3] we would expect that if the version number is found to be incorrect, then the process will have recently been unscheduled (so that another process could change the version) and will have its required pages in core. Thus, if the system's minimum quantum is large enough to allow the decision to be made in less than a quantum, the process will not again be unscheduled until after the privileged procedure has been executed.

We cannot make as strong a statement for a system with more than one processor. With a small number of processors, there is still some chance of interference, but we would still expect it to be small unless processors are spending a very large fraction of their time deciding to modify one particular object. With a large number of processors, one would probably require a "protect" instruction with some sort of hardware-implemented scheduling.

REFERENCES

1. Corbató, F. J., and Vyssotsky, V. A. Introduction and overview of the MULTICS system. Proc. AFIPS 1965 Fall Joint Comput. Conf., Vol. 27, Part 1. Spartan Books, New York, pp. 185-197.

2. Daley, R. C., and Dennis, J. B. Virtual memory, processes, and sharing in MULTICS. Comm. ACM 11, 5 (May 1968) 306-312.

3. Denning, P. J. The working set model for program behavior. Comm. ACM 11, 5 (May 1968) 323-333.

4. Dennis, J. B. Segmentation and the design of multiprogrammed computer systems. J. ACM. 12, 4 (Oct. 1965) 143-155.

5. Dennis, J. B., and Van Horn, E. C. Programming semantics for multiprogrammed computations. Comm. ACM 9, 3 (Oct. 1966) 143-155.

6. Dijkstra, E. W. The structure of the "THE"-multiprogramming system. Comm. ACM 11, 5 (May 1968) 341-346.

7. Evans, D. C., and LeClerc, J. Y. Address mapping and the control of access in an interactive computer. Proc. AFIPS 1967 Spring Joint Comput. Conf., Vol. 30. Thompson Book Co., Washington, D. C., pp. 23-30.

8. Fenichel, R. R. On implementation of label variables. Comm. ACM 14, 5 (May 1971) 349-350.

9. Gray, J. Locking. Record of the Project MAC conference on concurrent systems and parallel computation. ACM, New York, 1970, pp. 169-176.

10. Lampson, B. W. Dynamic protection structures. Proc. AFIPS 1969 Fall Joint Comput. Conf., Vol. 35. AFIPS Press, Montvale, New Jersey, pp. 27-38.

11. Lampson, B. W. A scheduling philosophy for multiprogramming systems. Comm. ACM 11, 5 (May 1968) 347-360.

12. Saltzer, J. H. Traffic control in a multiplexed computer system. Project MAC Technical Report 30 (thesis). Cambridge, Mass., July 1966.

13. Saltzer, J. H., and Gintell, J. W. The instrumentation of MULTICS. Comm. ACM 13, 8 (Aug. 1970) 495-500.

14. Watson, R. W. Timesharing system design concepts. McGraw-Hill, New York, 1970.