FOR APPROVAL

Identification

Interlocks for Access to Shared Data

V. A. Vyssotsky, 1/21/66

Introduction

In Multics a data base may have to be referenced by two or more processes, and modified by one or more of those processes, under constraints which preclude explicit advance planning of the order in which the various references and modifications will occur. Among the data bases with this property are page tables, file directories, facilities assignment tables, the active segment table and the ready list, as well as various data bases belonging to users. Unless a systematic technique is used to control the sequence of accesses to such a data base, errors are apt to occur. For example, without control of the access sequence it is possible for one process to reference a data item while another process is modifying that item so that the item is referenced at a time when its content is erroneous or inconsistent. This section specifies the interlock techniques used for controlling sequence of accesses to shared data of the operating system, and recommended for shared user data. A discussion of various alternative approaches to the problem may be found in the memorandum 'Controlling Independent Asynchronous Seizures of Shared

Facilities' by V. A. Vyssotsky.

The stac Instruction

It is expected that a new instruction, Store A Conditional, will be added to the repertoire of the 645 some time after initial delivery. If C(Y)≠0, stac Y merely sets the zero indicator OFF. If C(Y)=0, stac Y sets the zero indicator ON, and also replaces C(Y) by C(A).

At initial delivery of 645s not equipped with the stac instruction, the operation code 007 (mme4) will be reserved for the same purpose, and the function of stac will be performed by a fault-handling routine. This routine is described in detail in section .

Interlock Specification

Let d be a data base shared by two or more processes, and modified by one or more of them. Associated with d there will be a single word dlock, shared by the same processes, used to interlock access to d. The privileges required for access to dlock (e.g. master mode, chinese wall) will be the same as those required for access to d. Thus, unprivileged data has an unprivileged lock. The content of dlock at any time is either zero or the process id of some process. No process may access the data d unless dlock contains the process id of that process. A process may alter the content of dlock in only three ways:

A) If the content of dlock is zero, any process may write its own process id into dlock.

B) If the content of dlock is the process id of a process, that process may set the content of dlock to zero.

C) In error recovery situations described in section , and only in those situations, a process may replace by its own process id a process id already contained in dlock.

Action C will not be discussed further in this section. Action B will be programmed only at points where it is known that the procedure being programmed is running as part of the process whose process id is contained in dlock (e.g. because this same procedure running in this same process put the process id into dlock). Action B will be performed by the sequence of code:

```
lda dlock
cmpa my_process_id
tnz error_routine
stz dlock
```

The stz will not be programmed without the preceding test, because of the serious consequences of improper access to shared data, and the difficulty of tracking down such errors.

Action A will in most cases be performed by the sequence of code:

```
lda my_process_id
stac dlock
```

```
tnz already_locked
```

but in a few cases will be coded as

```
lda my_process_id
stac dlock
tnz *-1
```

This second form will be employed only in those cases where both of the following conditions apply:

a) The data base which is currently locked by some other process is required for the standard already-locked action described below, so that the standard action cannot be taken, and

b) It is 'known' (i.e. specifications claim) that the elapsed calendar time during which any one process will keep the data locked is bounded and short (e.g. 200 microseconds). In particular, condition b implies that no interrupts and no process faults can occur in any procedure of a process while that process has the data base locked.

In the standard sequence for locking data bases, the transfer

```
tnz already_locked
```

leads to code which performs the following tests and actions:

1) If the data base is locked by this same process, take appropriate action (see below). Otherwise,

2) Place a wakeup request with the process which has the data base locked.

3) If appropriate, place an alarm-clock wakeup request for some time so far in the future that failur to receive the wakeup of item 2 by then would be evidence of error.

4) Is the data base still locked by the process with whom a wakeup request was placed in item 2. If so, block. Otherwise,

5) Remove the wakeup calls of items 2 and 3, or the software interrupts which may have resulted from those calls. Then go back and perform action A.

On awakening from block with an item 2 wakeup signal, go back and perform action A.

It can happen in some circumstances (e.g. in a user routine for dealing with process faults) that a procedure attempting to access shared data will discover that the lock is already set by this same process. The action to be taken in these cases is not standardized. Two extreme cases, however, are specifiable. One extreme is the case where it is known (somehow) that neither the access for which the lock was set nor the one now desired can modify the data base d. In this case the desired reference can be made, and the lock left set, to be unlocked upon completion of the reference for which the lock was originally set. The other extreme is the case where it was not foreseen in planning that a process might attempt to lock the data when that same process had already set the lock. This is an error, and should be treated according to the guidelines of section .

Multiple Shared Data Blocks

It happens in some cases that a process must have two or more shared data areas locked simultaneously. In order to analyze these cases and establish standard treatment for them, we must first establish some definitions and terminology. Let us regard a process for the moment as a sequence of accesses to addressible storage. If we focus on the accesses to a particular shared data area d, we observ that they occur in some fixed order, a1d,...,amd. (Recall that a process is the execution of a collection of procedures, not the collection of procedures, so that the sequence of accesses to d during a process is unique, although perhaps not calculable in advance.)

Considering any pair of accesses (aid,ajd) to d during the process such that i is less than or equal to j, it either is or is not permissible for another process to access d between aid and ajd. We shall say that d is being used by p (or p is using d) in the closed interval given by any pair of accesses (akd,ald), k less than or equal to l, such that:

1) It <u>is not</u> permissible for another process to access d between akd and ald, and

2) For all i less than k it <u>is</u> permissible for another process to access d at some point between aid and akd, and

3) For all i greater than 1 it is permissible for another process to access d at some point between a1d and aid.

If any process p uses d at a time when d is not locked by p, an error is likely. However, it is clearly desirable that d should not be locked when it is not in use. Considering only one shared data area d, the previous two sentences specify the points in each process at which the process should lock and unlock d. However, this does not suffice to determine when locking and unlocking should be done if a particular process p uses (with the above definition of 'uses') data areas d1 and d2 simultaneously.

Let d1,d2,...,dn be shared data areas. We shall say that di *requires* dj if there is any process p which accesses dj while p is using di. Observe that for each di, di requires di. Observe also that di may require dj even though dj does not require di. We extend the definition by induction: if di requires dj and dj requires dk, then di requires dk. (Note that di may require dj only because of process p, and dj may require dk only because of process q, yet we still say that di requires dk.)

The set of data areas d1,...,dn can now be partitioned into subsets S1,...,Sm such that:

1) If di and dj are in the same subset Sk, then di requires dj and dj requires di, and

2) If di and dj are in different subsets Sk, Sl, then either di does not require dj or else dj does not require di.

We shall call each subset Sk a <u>data class</u>. (Graph theorists will observe that we have defined data classes to be the strongly connected subsets of a directed graph.)

Our rule for interlocking multiple shared data areas can now be stated as follows: Each data class will be treated as a single data area, with one lock. Two distinct data classes will be given separate locks. In many cases, of course, it is difficult to determine exactly what data areas constitute a data class. In such cases the criterion to be used is that treating several data classes as if they wer one may cause inefficiency; separating two system data areas of the same class and using separate locks for them may cause the system to loop; separating two user data areas of the same class and using different locks for them may cause the user processes to block and not restart.

In order to reduce the chances that the system will loop, three guidelines should be followed by system programmers:

1) Do not access customers' shared data areas while using shared system data base.
2) When specing and programming a procedure, analyze the requirements of each shared system data area used by the procedure, determine the data class of each such data area, and ensure that the class is

locked and unlocked as a unit.

3) To reduce the complexity of analysis, enhance the chances of doing it correctly, and improve the performance of the operating system, program to minimize the number of shared data areas that the procedure uses simultaneously.

Interlocks on Acess to Page Tables

A special case of shared data base which requires extra attention is the problem of how to interlock page tables. We shall consider the problem of changing a page table entry to 'directed fault'. when it was previously of some other class. All references to the page tables as data, of course, can be considered as like references to any other shared data area, but the page table may also be accessed directly by the appending hadware. For this reason, it is sometimes necessary to force one or more CPUs to clear associative memory. To get a CPU to clear associative memory, you have to attract its attention, via an interrupt or a connect fault. After the connect is issued or the interrupt cell set, the page must be left inviolate long enough to be sure that no associative memory contains a pointer to it. This must be done either by counting off 'enough' time, or by waiting for a response from the affected CPU(s), or by some combination. The safe strategy, and the one to be employed, is to loop until a positive response is returned.

For each CPU there will be a signal word reserved for response to requests for clearing of associative memory. A process, in order to set a page table entry to directed fault, will perform the following actions in sequence before making available for reuse the core block to which the page table entry points.

1) Set the lock cell appropriate to the page table regarded as a shared data base.
2) Set the page table entry to directed fault.
3) Set non-zero the signal words for those CPUs which must clear associative memory.
4) Issue connects, or set interrupt cells, to those CPUs.
5) Wait until each of the relevant signal words has reset to zero.
6) Reset the lock cell which was set in step 1.

A CPU which receives a fault or interrupt signal to clear associative memory will first do so, then set its signal word to zero.