Published: 11/15/67

Identification

The Locker facility
Michael J. Spier

Purpose

Shared data bases (common to more than one process) have to be protected from possible damage which could be caused by concurrent parallel manipulation. By a systemwide convention, access to such data bases is subject to the state of an associated "lock." A process which does access such a shared data base "locks" it prior to any manipulation and "unlocks" it as soon as the data base can once more be safely accessed by another process. A process refrains from accessing a locked data base; it "waits" for it to become unlocked. The Locker facility handles these locking, unlocking and waiting functions.

The Locker is a user of the Interprocess Communication Facility (MSPM section BQ.6.00). A thorough knowledge of Interprocess Communication is required of those who wish to understand how the Locker functions. However, no such knowledge is required in as far as the use of the Locker is concerned.

The Locker resides in the Administrative ring. It is invoked to handle the locking of any data base within the system regardless of its process group affiliation. Hardcore tables are managed by the Standard Interlock Mechanism (MSPM section BG.15.03).

The Locker has no special tables associated with it. It may be invoked from rings 1-63.

Introduction

We define "shared data base" to be a shared data segment (accessible to more than one process) or any data structure within that segment. Data bases can be nested within one another. The data base's size and depth of nesting are arbitrary. However, a nested data base must be accessed progressively, by locking and unlocking its containing data bases.

The "lock" is a word of memory having a bit(36) EPL attribute and known to be associated with a data base. (Due to the 645-dependent method of locking, "lock" has to have

a machine-dependent definition). It is considered to be "unlocked" if its value is zero and "locked" if its value is non-zero. By convention, a bit(36) process-id is always used for the locking constant. The locking is done by means of the STAC instruction which --within the same read-write memory cycle-- reads the contents of "lock" and stores the accumulator into it only if "lock"'s previous value was zero (we name this condition "stac successful"). If "lock" contains any other value its contents remains unmodified and this condition made known to the program. (Note that the hardware allows only one processor at a time to access a given word of core).

Unlocking is done by conventionally storing a zero-constant in "lock". Normally, unlocking is permitted only to the process which has previously set the lock.

By a systemwide convention, the Locker facility considers a lock to have the following structure:

```
declare  1 x based(lock_ptr)  /* lock structure */,
         2 lock bit(36)       /* lock word */,
         2 channel bit(70)    /* event channel name */;
```

In practice, however, the user must reserve a zone of 4 contiguously addressable words per lock. When corresponding with the Locker, the user refers to the first of these 4 words. (Example: dcl lock(4);)

When using the PL/I language, care must be taken to avoid the possible packing of the lock structure with adjoining data items. It is therefore recommended to have the lock structure declared as a substructure (or array) of 4 fixed binary variables.

When the Locker fails in its attempt to set a lock (STAC unsuccessful), it invokes its waiting mechanism which sends the current locking process an event signal, asking to be notified when the data base is once more unlocked, then blocks itself in the Wait Coordinator.

A detailed description of the Locker's internal logic and its waiting mechanism is given under "implementation". The following paragraph discusses general problems associated with the locking of shared data bases.

Possible error conditions

Normally a process may block itself in its Wait Coordinator, awaiting a specific lock to be reset. If, for some reason, the awaited event signal does not arrive, the process will remain blocked indefinitely.

This may happen because of two possible conditions:

1. The process which currently locks the data base attempts to set the same lock recursively (i.e. it goes blocked waiting to be awakened by itself).

2. The process which has set the lock has been quit or destroyed.

To intercept the first error condition, the Locker compares its own process-id to the lock's current value. An error-status return is made if both values are found to be equal.

The second error condition makes it necessary for the Locker to know whether or not the locking process has been quit. This information is available to the Interprocess Communication Facility (the wakeup switch in the Event Channel Table). Set_event returns a status indicator if the target process was found to have been quit. The user of the Locker determines (see "implementation") whether or not this actually is an error condition, and how it is to be handled. Within a process group, for instance, it can, in most cases, be disregarded; for normally, if one working process of that group is quit, all other working processes of that group are quit as well. Similarly, certain processes which do not belong to the same process group (notably in the I/O system) might choose to wait for a data base which is currently locked by a quit process.

The user may wish to insure himself against the eventuality that he be blocked indefinitely in the Wait Coordinator awaiting an event signal from a quit (and possibly destroyed) process. He specifies that unless an event signal arrives --within a given time limit-- from the process which currently locks the data base, the locker is to try that lock again. A status indicator returned by set_event to the effect that the target process was quit or destroyed causes an error-status return from the Locker.

A number of shared data bases --in the Administrative ring-- must always be correctly updated; the system does not tolerate a process being quit while manipulating such a data base, as this would result in the data base being indefinitely locked as well as being in an unpredictable state.

Hardcore data bases are automatically protected in that a process is never quit while executing in ring 0. Likewise, the Quit Inhibit Counter (see MSPM section BQ.3.06) inhibits quitting in the administrative ring until no more "sensitive" data bases are endangered.

Another problem is that of accessing the lock structure. A process that succeeds in setting the lock (storing its process-id in "x.lock") proceeds to store an event channel name in "x.channel". A unique identifier (event channel name) is known to never contain a zero value in its leftmost 36-bits. Consequently, a second process which reads the (locked) lock structure accepts "x.channel" as a valid event channel name only if the first 36 bits of the event channel name are unequal to "0"b. If this condition is encountered, the Locker reads "x.channel" again.

Implementation

The Locker is invoked by calling:

```
locker$wait(lock_ptr,wait_sw,time_limit,status);
declare lock_ptr pointer, wait_sw bit(1),
        time_limit fixed bin(17), status bit(36);
```

lock_ptr is a pointer to the 3-word lock structure
wait_sw is a switch which indicates whether the Locker is to wait indefinitely for a lock to reset ("0"b) or whether it should wait for a given time interval only ("1"b).
time_limit (provided that wait_sw="1"b) is the time in seconds to be waited before the Locker abandons the lock.

The logic of the wait-sw is the following:

"0"b= The Locker waits for the lock to be reset, ignoring the possibility that the (current) locking process might be quit. Locking is abandoned and an error-status return is made only if the locking process is known to have been destroyed.
"1"= The Locker is to "mistrust" the locking process. If an event signal is not received from that process within a given time limit, an error status return is made. Likewise, an error-status return is made if the other process is found to have been quit.

When the Locker (executing in process A) is invoked, it goes through the following steps:

a. It creates an event-queue-mode channel, having event channel name "a".

b. It attempts to set the lock, using its own process-id (A) for locking value.

If the STAC is successful, it puts event channel name "a" into "x.channel" as explained above. The Locker returns to its caller.

If the STAC is unsuccessful (lock set by process B), it tries to read event channel name "b" out of "x.channel". If successful (the event channel name's leftmost 36 bits are non-zero) it proceeds to step c. Otherwise, it resumes step b after having blocked itself for a certain length of time. Note: Functionally, this amounts to a transfer to step d. However, the "time_limit" argument may be either absent or of too high a value. The Locker therefore sets its own time limit.

c. The Locker (still executing in process A) sends process B an event signal over event channel "b". It uses event channel name "a" as event indicator (see unlocking step d). Set_event may return four status indicators which, if present, are interpreted as follows:

1. The target process, B, cannot be reached (process destroyed). The Locker makes an error-status return to its caller.

2. The target process has been previously quit. The Locker makes an error return to its caller if wait_sw="1"b.

3. Event channel "b" cannot be found. The Locker reads once more "x.channel". If it still contains event channel "b", an error return is made. Otherwise, the Locker loops back to step b.

4. Event channel "b" has been inhibited (process B called ecm$cutoff(b), see unlocking step b). The Locker loops back to step b after waiting for a certain length of time.

d. The Locker tests "wait_sw". If it is set to "1"b, it calls upon the Calendar Clock Coordinator (MSPM section BD.10.03):

```
call set_wakeup_interval(interval,a)
```

where interval is a function of "time_limit" and a is event channel "a". This insures that either process B or the Calendar Clock Coordinator will signal over event channel "a" within (time_limit) seconds.

If wait_sw is set to "0"b, the Locker calls set_wakeup_interval and specifies an arbitrary time constant (e.g. a minute). This insures that, even though the Locker will wait until the lock has been reset, it still retains the capability of checking every once in a while to see whether or not the other process has, in the mean time, been destroyed.

e. The Locker calls the Wait Coordinator to wait on event channel "a".

f. Upon return from the Wait Coordinator, the Locker loops back to step b.

To unlock a data base, locker$reset(lock_ptr) is called which does the following:

(The Locker is executing in process B, process A is waiting for the lock to be reset).

a. Compares its process-id (B) to the value of "x.lock". An error return is made if the process which tries to reset the lock is not the one that has set it.

b. Inhibits the event channel ("b") which is associated with that lock by calling ecm$cutoff(b).

c. Resets the lock structure.

d. Reads event channel "b". Reading is done by calling wc$test_event. The received event-id "a" is interpreted as being an event channel name (see locking step c). It sends an event signal over that channel using the channel's name as event-id. Reading the event channel is repeated until it is found to be empty.

e. Deletes event channel "b".

f. The Locker returns to its caller.

(Note: By matching the procedures of locking and unlocking, it becomes evident why the Locker interprets the four status indicators returned by set_event in four different ways.)

Locker$try

Certain users of the Locker facility do not wish to wait for a lock (i.e. process their locks sequentially). Such users, notably the Device Manager Process Group, lock their data bases by calling locker$try rather than locker$wait. Locker$try is a stripped-down version of locker$wait, with the automatic waiting functions (notably the interface with the Interprocess Communication Facility) removed. It is intended for the more sophisticated user which wants to access several shared data bases "in parallel."

It is invoked by calling:

```
locker$try(lock_ptr,ev_chn,wait_sw,time_limit,status)
declare ev_chn bit(70);
```

where ev_chn is the name of an event-queue-mode channel.

The basic difference between locker$wait and locker$try is that the latter never calls the Wait Coordinator. It always returns to its caller, whether or not the attempt to set the lock was successful. It performs, basically, the same way lock$wait does. The following lists the differences between the two procedures, with reference to the "locking steps" outlined under "implementation":

step a. It does not create an event channel, but uses the event channel name which it received as argument.

step b. same

step c. The Locker tests ev_chn for a zero-value (ev_chn="0"b).

ev_chn="0"b no signalling is to take place. The Locker returns to its caller.

ev_chn≠"0"b same as locking-step c.

step d. The Locker calls the Calendar Clock Coordinator if wait_sw is set to "1"b.

step e. The Locker returns to its caller.

The unlocking of a lock is done conventionally, by calling locker$reset.

The logic of the wait_sw is the following:

"0"b = The Calendar Clock Coordinator is never invoked. An event may or may not be signalled over event channel "a".
"1"b = An error-status return is made if, while locker$try is executing, it is found out that the other process (B) has been quit. Event channel "a" will be signalled over within a maximum delay of (time_limit) seconds.

Calls to the Locker facility

```
call  locker$wait(lock_ptr,wait_sw,time_limit,status)

call  locker$try(lock_ptr,ev_chn,wait_sw,time_limit,status)

call  locker$reset(lock_ptr)

declare  lock_ptr pointer, time_limit fixed bin(17), status
bit(36), ev_chn bit(70);
```