RJ 5118 (53162) 4/23/86
Computer Science

Research Report

SYSTEMS PROGRAMMING: COPING WITH PARALLELISM

R. Kent Treiber

IBM Almaden Research Center
650 Harry Road
San Jose, California 95120-6099

RJ 5118 (53162) 4/23/86
Computer Science

Systems Programming: Coping With Parallelism

R. Kent Treiber

Almaden Research Center
K55/801
650 Harry Road
San Jose, California 95120-6099

Abstract: Creating operating system components and subsystems in today's large processors generally requires dealing with more than one CPU operating in parallel against a shared memory. While "applications" are typically shielded from the effects of parallelism, components and subsystems usually are designed in such a way that some level of understanding is required. This paper concentrates on the pitfalls awaiting the programmer in a parallel (or even a multiprogramming) environment when shared data structures (control blocks) are referenced and altered by multiple processes (tasks). The focus is on the IBM System/370 architecture because of its multiple CPU architecture and the powerful "compare and swap" instruction. The paper reviews some architectural groundrules that a parallel programmer must understand, presents problems that must often be solved in a parallel environment, then describes solutions such as usage of compare and swap, locks, and single-process schemes. Kernels of code are used to illustrate problems and solutions.

CONTENTS

INTRODUCTION

UNNATURAL, ERROR PRONE AND UNTESTABLE

Designing and coding in a parallel environment is unnatural for most programmers. Programmers tend to think in sequential, if-then-else kind of logic. In a parallel environment, the traditional if-then-else may be totally invalid if its operating against a data structure that is shared with another process (task or dispatchable unit of work). In a parallel environment, the programmer must ask "what if another process is running simultaneously?" after coding each instruction associated with a shared data structure (most commonly a control block). In fact, many of the problems discussed here can occur in a multiprogramming environment as long as at least two processes are operating with preemptive dispatching (where one process can lose control of the CPU at any instruction boundary and another process can be given control of the CPU).

In addition to being unnatural, the parallel environment is very error prone (partially due to the unnatural nature, partly due to complexity). Unless serialization is done at a very high level, the level of system understanding required to even *recognize* and then *resolve* a parallelism exposure is high.

As if unnatural and error prone weren't bad enough, parallelism "bugs" are very difficult to detect by testing. Most of these parallelism exposures only function incorrectly in very small "windows" of time that occur in a parallel environment. Often the failure results in a problem much later in processing and there is no trace of the actual failure. In many cases, the failures only come under load and are very difficult to reproduce, particularly in a real customer environment. Introducing debugging tools can alter performance such that a parallelism exposure window does not occur (an example of the Heisenburg uncertainty principle).

A fairly obvious solution to this nasty set of problems is to introduce high level serialization to avoid parallelism exposures. This is a valid approach, but such a decision must be made carefully: it may have far reaching impacts on thruput and response time. Once complex code is written depending on high level serialization it is likely to be extremely difficult to make the code work with lower level techniques that allow more parallelism.

The author's early experiences with system programming basically did not involve parallel environments and he has learned the hard way. It is hoped that this paper will make it easier for others.

ARCHITECTURE REVIEW

The System/370 (extended or not) architecture [princop] documents synchronizing and serializing that must occur in all System/370 machines. These issues are important when programming in a parallel environment with shared data structures; most are important even in a preemptively dispatched multiprogramming environment. The following portions of the principles of operation [princop] are summarized in this document, but are worth review:

- Chapter 5, section on "Sequence of Storage References". This describes what "really" happens when instructions that modify main storage are executed.

- Chapter 7, the Compare and Swap (CS), Compare Double and Swap (CDS) instruction descriptions.

- Appendix A, section on "Multiprogramming and Multiprocessing Examples".

When dealing with a parallel environment, the machine does not operate the way many programmers think it does. For example, "comparing a field to itself my yield a result other than zero". This can occur if another CPU updates the storage after the compare has fetched the first operand and before it fetches the second operand. Another example is that a test of a bit by an instruction on CPU A can show 0 even though an instruction on CPU B set the bit to 1 several microseconds earlier. This occurs when *CPU serialization* is not performed. A third example is a move instruction copying shared storage that obtains a copy that contains only part of the changes caused by an instruction running on another processor. A fourth example is code that increments a counter in shared storage - occasional increments will be lost. These are problems in serialization and consistency. The key architectural rules are:

- To obtain consistent data from shared storage that may be altered by other CPU's, you must use instructions that provide *block-concurrent reference*. This can allow you to see or copy all bytes within a halfword, word, or doubleword consistently - there will be no partial results of an instruction. Note that this does not guarantee you current data, just consistent data. Note that you also cannot get consistency on more than a doubleword. The following instructions give you block-concurrent reference for a word or doubleword: LOAD, LOAD MULTIPLE, COMPARE LOGICAL, COMPARE LOGICAL CHARACTERS UNDER MASK, INSERT CHARACTERS UNDER MASK, MOVE CHARACTER. Most RX format instructions are block-concurrent when their operand is on the appropriate boundary. When using higher level languages such as PL/AS, you generally don't control the instuctions used but you can often control storage boundaries. You can get block-concurrent instructions that alter storage, but this is misleading: these instructions will prevent the contents of a word or doubleword from containing partial results of an instruction, but they will not prevent total loss of the results of an instruction due to alteration of the same storage locations by instructions running on another CPU. Another potential problem with block-concurrent instructions that alter storage is that programs running on other CPU's may not see the change for a while - see below.

- To alter data in shared storage, access must be serialized by a software mechanism such as locks, or the Compare and Swap, Compare Double and Swap (CS|CDS) instructions must be used. A surprisingly large number of operations can be accomplished with careful use of these instructions.

  A basic description of

  $$CS \quad R_1, R_3, disp(R_2)$$

  is: The content of the first operand (R_1) is compared to the value at the main storage addressed by the second operand and, if equal, the main storage at the second operand address is replaced by the content of the third operand (R_3) This is all done "atomically": no other CPU is permitted to alter the main storage location between the compare and the alteration. If the comparison produces a not-equal condition, the value at the main storage addressed by the second operand is loaded into the first operand register. Condition codes allow branching based on the success of the instruction.

- In order to be sure that the results of storage alteration by instructions are visible to instructions executing on other CPU's, a *CPU serialization* operation must be performed by the CPU that altered the storage. Common instructions that cause serialization are: BR R0 (which causes CPU serialization but does not branch), STORE CLOCK, SUPERVISOR CALL, COMPARE AND SWAP (CS|CDS). All interruptions cause serialization. Assuming you have a "safe" technique for altering storage (such as serializing through a lock), this aspect is important - you must cause CPU serialization at the appropriate time (before unlock is complete) or you can still get into trouble. This aspect of the architecture is most likely to cause a problem with readers of shared storage on one CPU while an updater is on another CPU: the reader sees old data or an inconsistent mix of old and new data.

BASIC RULES FOR SAFE OPERATION

To survive in a parallel environment, rules for accessing and altering shared storage must be established; the specific rules must be followed by all programs. The primary rule to follow for storage alteration is: *you must alter shared storage only when holding a lock that protects it or via the CS, CDS instructions.*[1]

- If every program wishing to alter an area of shared storage obtains an exclusive lock (refer to "LOCKS AND LATCHES" on page 5) before altering the storage and frees the lock when through, consistency and integrity will be maintained.[2]
- If storage alteration is performed using the Compare and Swap (CS or CDS) instructions, consistency and integrity will be maintained. Obviously, the largest area that may be safely altered in a consistent manner with these instructions is 8 bytes on a doubleword boundary.

The primary rule to follow for storage access (reading) is: *you must read shared storage only when holding its lock or by use of block-concurrent instructions.* If you need consistent data that does not come from the same doubleword in the shared storage, you must hold a lock that prevents updates.

These rather restrictive rules are relatively simple and easy to follow once you have really learned them. Unfortunately, the basic rules drive you toward lock solutions that in turn can cause performance problems.

OTHER GENERAL SOLUTIONS

There are other approaches to dealing with parallelism, but "rule 1" strongly applies: know what you're doing; the cost of failure is high. We recommend that you keep your design as simple as possible consistent with your performance objectives. This section summarizes some of the generally applicable approaches.

- **Single Update Process**.. A single process can be used to update areas of shared storage. With a single process, there is no parallelism problem and no preemptive interrupt problem. A single operating system process may be sub-dispatched to get "multi-threading" as in CICS, IMS, or other systems, but this is OK because threads are not preempted to run other threads. Other

1 The test and set (TS) instruction available on 370 is a holdover from the System/360 MP65 machine. TS is very primitive and is, in general, no longer used. TEST AND SET can be described as: atomically set all bits in a byte to ones while evaluating bit 0 of that byte. Thus you can detect if you were the one to set a bit (bit 0 of a byte only) to one. You cannot use this instruction to directly set a bit to zero; you cannot test several bits; you cannot set/reset several bits. In addition, since TEST AND SET is carried along soleiy for compatability, its implementation is probably slow.

2 This assumes that the unlock operation will use an instruction (CS) that will cause CPU serialization (cause all storage changes to occur as viewed from other CPU's).

processes may read the shared storage as long as they understand the CPU serialization and block concurrent reference implications. This single update process can manipulate complex chains involving many shared storage areas in complete safety. Optionally, the single update process can be used for only certain areas of a shared data structure such as complex chain manipulation while other areas (such as words containing bit flags) are updated by any process via CS.

The single update process has many attributes in common with the use of a single high level lock for update - both techniques serialize to allow only one updater at a time.

There are certainly drawbacks to the single update process approach. If the need to update occurs on other processes, then passing that need to the single process can be cumbersome and expensive. A single update process cannot take advantage of multiple CPU's and potentially becomes a bottleneck. If the single update process must do I/O, a much bigger potential bottleneck exists unless the design employs multi-threading.

- **Single Owner**. Another approach, single owner, naturally fits some designs. If a given data area is shared among many processes, but is owned or allocated to one specific process at a time, that one process can update with ordinary instructions without a lock. If other processes may read the area, caution must be used to ensure that what they read is as consistent and current as they require - serialization operations may be required of the owning process.

- **Complex CS Algorithms**. There are several different "tricks" that can be performed using the CS, CDS instructions to manipulate shared data structures in ways that are not immediately obvious. These approaches have the advantage of avoiding locks, but tend to have the disadvantage of complexity. If you invent what you think is a new such approach, examine it carefully for subtle bugs.

- **Obligation Passing**. Usually combined with the use of CS, the obligation passing technique has many different applications. The basic approach is to attempt to perform the work that requires serialization, but if another process is conflicting with you, pass the obligation to perform the work to the other process.

One variation of obligation passing uses "last one out, take out the trash" type of logic. Normally, there will only be one person in the room and he will take out the trash, but the logic also handles a crowd, keeping trash off the floor.

Another variation of obligation passing uses "first one there does it all until there is no more to do" type of logic. Normally, the first one there will do his own work and leave, but the logic also handles a crowd showing up while he's doing the work.

Several examples of obligation passing, some quite complex, are shown later.

LOCKS AND LATCHES

BASIC DESCRIPTION

Function

A well-known serialization mechanism, the lock is a mechanism allowing control of access to and change of shared data structures - locks are a serialization mechanism[3]. Data base systems use locking to control access and change of data base items - that application is ignored here. Note that a lock, like other techniques, is based on a gentlepersons' agreement. Everyone must agree to perform operation *xyz* on a data structure only when lock *abc* is owned.

In its simplest form, all must agree that any access to shared data object *x* requires lock *l*. Only one process at a time is allowed to own the lock. When desiring access to *x* you request lock *l*, and when you get control back from the request, you own the lock. You may now read and alter shared data item *x*. When you are through reading and altering *x*, you release the lock. This is basically the type of lock provided and used extensively by MVS [mvslock].

What happens when you request a lock and some other process owns it? In most cases, your process "suspends" or waits until the lock is available. For certain critical operating system functions, the "spin lock" approach may be used: go into a loop trying to obtain the lock, preventing other work on this CPU by disabling interrupts while trying to get the lock and while holding the lock. Because of the "severe" action that occurs when a lock is not available, a lock service (like MVS) may allow a conditional request for a lock: get it if it's available, tell me if it's not available.

The IMS/VS product has both locks and latches. My colleague Kurt Shoens differentiates locks and latches thusly: A lock is based on a name (some bit string), a latch is based on a storage location (a word or doubleword in storage). Thus, MVS and VM have misnamed their serialization mechanisms since their "locks" are really latches because they are based on storage locations; MVS ENQ/DEQ really is lock/unlock. For this paper, we will remain unenlightened and use the term lock for both locks and latches.

Modes and Compatibility

Locks often have different "modes". Once you have modes, you need to define "compatibility" between modes. The simple lock we described above is an exclusive mode lock: only one holding process is allowed at a time (since exclusive is not compatible with exclusive). It is possible for a lock to have a shared mode: any number of processes may share it. Shared mode locks usually have an exclusive mode, exclusive mode being incompatible with exclusive and share (see Figure 1 on page 6). When a lock request is received that is incompatible with the modes of existing holders of a lock, the request waits (and all subsequent requestors of any mode are usually made to wait as well) until incompatible holders release the lock. Database lock managers often have many modes and complex compatibility tables.

3 For those with a computer science background[introos], some translation of terms: What is often called a lock, this paper would call an exclusive mode spin lock. The P and V operations on the semaphores described in computer science literature would be called lock and unlock operations against an exclusive mode suspend lock.

	SHARE	EXCLUSIVE
SHARE	compatible	incompatible
EXCLUSIVE	incompatible	incompatible

Figure 1: LOCK COMPATIBILITY TABLE

Appendix A of [princop] describes logic (and shows assembler code) for two different lock/unlock services. Both support an exclusive lock, assume the caller can issue WAIT and POST, and ignore recovery considerations. One has LIFO queueing on contention (somewhat unfair, but easy), the other FIFO queueing. Note that these routines use CS and may issue SVC instructions so CPU serialization (all changes visible to other CPU's) will be performed before any other CPU may obtain the lock.

Deadlocks

The use of locks generally requires either a deadlock avoidance or deadlock detection/resolution mechanism. A simple deadlock can occur as follows: Process A gets lock 1; process B gets lock 2; process A requests lock 2 and is suspended until it is available; process B requests lock 1 and is suspended until it is available. It is possible for deadlock to occur with combinations of locks and other "wait until available" resources.

By far the most desirable treatment of deadlocks for the parallel environment is deadlock avoidance.[4] The simplest deadlock avoidance scheme is to never attempt to hold more than one lock at a time. For the majority of applications of locking, this may be possible, but it is unlikely to cover all cases encountered when building a system. For example, a control block may be on two chains because it can be located in two ways. Most of the time, a lock on only one chain would be required, but deletion of the block would require simultaneous ownership of two locks.[5]

The MVS approach to deadlock avoidance is to define a hierarchy of locks and a rule: you may request unconditionally only those locks that are higher[6] in the hierarchy than the locks you currently hold. This sounds like it is easy to do, but it can get complicated where there are many layers of code: you may get into trouble by holding a lock and then using a function which, it turns out, uses a lower level lock to cover itself.

The basic address-space related lock in MVS is called the "local lock". It is relatively easy to get and serializes activity within an address space nicely. There is only one of these type of locks

4 Database systems have had deadlock detection and backout mechanisms since the 1970's. Building a backout mechanism can be expensive and restrictive, but database managers tend to need it anyway in case of failures.

5 You could do it in sequence: get one lock, manipulate one chain, release the lock, then repeat for the other chain. The potential problem here is the creation of a window where the block is on only one chain.

6 In VM, you must only request locks "lower" in the hierarchy. The key is that there must be an ordering and you must move in only one direction when obtaining locks.

per address space so there is no deadlock problem unless you're dealing with other "wait until available" resources. Unfortunately, the locking granularity is very coarse - only one for all activities in the address space. Many basic and unrelated MVS services require the local lock - there is more serialization than necessary. For example, if you get the local lock and then page fault, any other process in your address space that attempts GETMAIN will be serialized (because GETMAIN needs the local lock) until your pagefault is resolved and you give up the lock.

Recovery

In components and subsystems that have logic to prevent abnormal termination when a failure such as a program check or abend occurs, additional complexity is required in lock support. If an error occurs in a process that holds a lock, the recovery logic must ensure that the lock is released, otherwise your system will probably "dry up" as processes wait for the lock that they can never obtain. If an error occurs in a process that is waiting for a lock (asynchronous abend, for example), the mechanism used by the lock support to locate and resume the waiter must be cleaned up if the process in error is terminated. MVS includes recovery in its lock support, but if your code is going to retry from failures, you'll need to take locks into account.

ADVANTAGES

Building a complete lock service including recovery support is not trivial, but is also not a tremendous amount of work. You may be able to use existing operating system support. Once a lock service is available, using it to serialize and avoid parallelism problems appears quite desirable:

- Getting a lock is simple: a procedure call or a macro.
- Code written to manipulate a shared object while holding a lock does not have to use special instructions or worry about consistency and CPU serialization. To a large degree, the complexity of parallelism is eliminated (because when you have the lock you have eliminated parallelism), eliminating lots of opportunities for tricky bugs.
- A simple lock request can be satisfied in as few as 6 instructions generated by an inline macro if the lock is not already held by another process; the release of the lock can be as cheap.
- Avoiding deadlock seems easy early in a design and most of the time actually is easy.

DISADVANTAGES

- The primary disadvantages of locking lie in the performance arena: Designs that made "bad" decisions about the use of locks have caused serious bottlenecks in systems and increased pathlengths significantly:
  - The suspend/resume that occurs when a lock is requested and already held normally costs several thousand instructions.[7] This is one of the costs of "lock contention". How much contention will occur? You need to know this to decide if a particular locking design is viable. A suggested approach is to discussed under "LOCKING PERFORMANCE ANALYSIS" on page 9.

[7] When running disabled for interrupts, spin locks can be used instead of suspend locks. The cost of spinning is generally going to be less than the cost of suspend/resume assuming that lock holders do not hold the lock for long instruction sequences.

- Using an exclusive lock to cover a "large" function causes that function to be available on only one CPU at a time. On a 4-way, 4 mip machine for example, a maximum of 1 mips worth of such a function can be supplied. Using a single lock to cover lots of function is just as bad as supplying all the function under 1 and only 1 process: you cannot utilize more than 1 CPU in the machine.

- Designs that are likely to page fault or do voluntary i/o while holding locks drastically reduce the capacity of the function and increase the probability of contention. If you estimate that a random i/o may average 25 milliseconds and a page fault response may take longer, then a locked service loses responsiveness and capacity very quickly when i/o or paging occurs.

- If your lock can be used by requests from several address spaces, your users had better be non-swappable. Getting swapped out while holding the lock would reduce the availability of your function significantly.

- If your lock can be used by requests from several address spaces of differing priorities you may provide poor service to high priority users. If you are running on a low priority user's process and obtain the lock and then are preempted for higher priority work your function will not be very responsive.

- In the general case, a lock service will probably cost more than the minimum. In reality, getting and releasing a lock with no contention will cost roughly 25 instructions as opposed to the three or four extra instructions required for a simple Compare and Swap.[8] If the function is "mainline", many cpu cycles may be spent in locking and unlocking. MVS components do enough locking to justify microcode assist.

- Another disadvantage of locking is the need to avoid deadlocks. The hierarchical rule approach used by MVS is simple and may allow you the flexibility you need. You probably want to build a hierarchy validity check into your lock service - it's a whole lot easier to find hierarchy program errors that way (the alternative is to debug the resulting deadlocks).

  The strict hierarchy approach to avoiding deadlock may be too restrictive - look closely at MVS locking and you'll see that it was for them: there is an exception where three locks at the same level can be obtained with one request; there's also a new "CPU" lock that doesn't follow the hierarchy rule. If you can't follow a strict hierarchy, rule 1 applies again, know what you're doing. Remember that the more complex the scheme, the more error prone it is.

- Locking often causes complications in recovery. If your component or subsystem is to have serious recovery logic (functional recovery or ESTAE logic that tries to clean up damage and retry), your recovery code must clean up locks obtained by a failing routine. It is difficult to determine precisely whether the failing logic obtained a lock, particularly when the lock service allows one process to request the same lock many times, thus the caller of the failing routine may have owned the lock. Minimum recovery logic must free any locks obtained by a terminating process, otherwise the rest of the system will probably dry up behind them.

- Once code is built with locking as the mechanism for preventing parallelism problems, changing approaches (eliminating the lock) will normally be quite difficult unless the function

8 A lock function in a real system requires instructions to deal with several aspects: finding the lock structure; statistics; tracking for recovery; checking hierarchy violations; linkage to out-of-line routines. The 25 instruction number is from a specialized subsystem macro that runs inline unless there is lock conflict. The MVS lock function with microcode assist (one instruction performs a lot of function) takes about 20 instructions for set and release.

is simple and well contained. The primary reason is that other approaches place more restraints on what can be done and when - code written based on locks will not be structured with this in mind.

LOCKING PERFORMANCE ANALYSIS

One potential problem with locking performance is contention. Designs using locks should undergo at least a cursory analysis to spot problems. A suggested approach is to:

1. Estimate *ihl*: the average number of instructions that will be executed while holding the lock. Be sure to be pessimistic.

2. Identify your favorite machine and get an estimate of the *mips* of a single CPU.

3. Estimate *n*: the maximum number of times per second that a function using this lock could be invoked on this machine. Again, be sure to be pessimistic.

A simplistic calculation of the probability of lock contention is:

$p = n*ihl/(mips*1000000)$

A simple sensitivity analysis should also be done: what happens if the pathlength of the function doubles or the number of requests doubles or the number of CPU's per machine doubles? If any one of these causes trouble, come up with another design. If two of these cause trouble, worry about the design.

Example 1: A function that averages 500 instructions will be executed under a lock on a 4-way machine where each CPU produces 10 mips. In a high usage environment, a 40 mip machine could generate 4000 invocations per second of this function. There would then be a probability of 4000*500/(10*1000000) = .20 for lock contention assuming an equal distribution of requests among processors and across time. I would look for another design approach: On the average, 20% of the requests will require suspend/resume. Just the pathlength impact of contention assuming 2500 instructions for suspend/resume would effectively double the cost of the function (.20*2500 + 500 = 1000). Another viewpoint is that the serialization mechanism actually costs as much as the function.

Sensitivity analysis: if the number of requests doubles, then contention goes to 8000*500/(10*1000000) = .40. If, in addition, the number of CPU's per machine doubles, you would normally expect the number of requests to double again, now giving 16000*500/(10*1000000) = .80.

Example 2: A function that averages 100 instructions will be executed under a lock on a 4-way machine where each CPU produces 1 mip. In a high usage environment, a 4 mip machine could generate 50 invocations per second of this function. There would then be a probability of 50*100/(1*1000000) = .005 of lock contention assuming an equal distribution of requests among processors and across time. Use of a lock for this function appears reasonable.

Example 3: A function that averages 100 instructions will be executed under a lock on a 4-way machine where each CPU produces 1 mip. In a high usage environment, a 4 mip machine could generate 50 invocations per second of this function. Since the function accesses user storage, we estimate a probability of .05 that a request will take a page fault lasting 40 milliseconds. There would then be a probability of 50*100/(1*1000000) + 50*.05*.040 = .105 of lock contention assuming an equal distribution of requests among processors and across time. Use of a lock for this function is marginal at best; I would look for another solution.

Convoy

A performance problem that has been encountered in some systems is the "convoy phenomenon"[convoy]. This occurs when a locked function that is normally quite fast suddenly takes a long time (takes a page fault, for example), causing a large number of requests to queue up waiting for the lock. The standard lock service logic when a lock is freed and a waiter is found is to grant the lock to the waiter and resume him. Once you get a large number queued up, it is hard to get back to a "nobody queued" state:

- Queuing up costs suspend/resume instructions. The instructions for resume and the elapsed time until the resumed process is dispatched all tend to count as "lock held time", thus the lock is 100% busy until the queue is dried up even though the code needing lock protection is executing very rarely.

- If a process releases the lock and then quickly requests the lock again (quite likely in some applications - remember the process releasing the lock is already dispatched), it will queue up for the lock behind everybody else, leading to a lock driven "timeslicing" condition.[9]

The solution to the "convoy phenomenon" is to change the "unlock, waiter found" logic: a) actually unlock, do not allocate the lock to anyone; b) resume every waiting process. This gives access to the lock to the first process that can use it, including the process that issues the unlock and other processes that have not yet requested the lock.

In solving the convoy phenomenon, you have changed a "fair access" design for the lock service into a design which could allow monopolization of a lock by a process. For example, two processes running on a two processor machine both are in a loop executing 500 instructions then getting a specific lock. If re-dispatching a resumed process takes more than 500 instructions then one of the processes will be "starved" - it will never get the lock because by the time it tries to get it again, it's held again. In most cases, if lock utilization is low as it should be, starvation should not be a problem although it might be a short term problem when trying to clear up a convoy. An approach that has been suggested [convoy] is to free the lock and resume only the first waiter, thus allowing the current process, the process who has waited the longest, and any new processes to compete for the lock.

The convoy phenomenon is a lesson in how complex the performance aspects of locking can be.

9 If your logic is such that you need the lock, then execute a few instructions that don't need the lock, then need the lock again, you should normally keep the lock instead of giving it up and obtaining it again. Giving it up for a very short time will tend to cause convoys or, if the convoy resolution approach is used, giving it up for a short time can cause lots of unproductive suspend/resume processing.

PROBLEMS AND SOLUTIONS

SETTING BITS

The Problem

As described in Appendix A of [princop], the standard instructions used for setting the value of bits will not consistently work for shared data structures in a parallel environment since the effects of some of the changes will be lost.

Figure 2 provides an example of failure. The intent of the two OI instructions was a value for A of 90. In a parallel environment, different timings will produce 90, 80 or 10. In this example, it produces 10. Note that this example does not illustrate the additional "window of error" that can be caused by lack of CPU serialization.

Solutions

Holding a lock that, by agreement, covers the bits being altered will allow use of ordinary instructions to alter the bits. To obtain a lock just to set bits that reside in the same word is overkill.

When using PL/AS, the solution is relatively simple: define the bits with the "abnormal" attribute and the compiler will generate Compare and Swap instructions to alter them. This is the only case where the compiler provides meaningful assistance. If you alter byte, character, or word variables that have the "abnormal" attribute, the generated code is not correct for shared data structures in parallel environments.

When using assembler, you must use Compare and Swap to alter the word that the bits reside in, typically altering the bits in a register. Appendix A of [princop] gives an example of this; another example is given in Figure 3 on page 12.

Note one of the costs of parallelism: a simple AND IMMEDIATE instruction is replaced with five instructions that require registers and use an instruction that's relatively expensive if the machine has more than one CPU (CS). In addition, for the code to remain somewhat independent of bit locations, additional definitions such as

```
TIME   CPU 1              MAIN STORAGE    CPU 2
----  -----------------  ------------  ---------------------
 0                            A: 00
 1       OI     A,X'80'
 2         fetch A = 00                   OI     A,X'10'
 3         alter A = 80                     fetch A = 00
 4         store A = 80       A: 80         alter A = 10
 5                            A: 10         store A = 10
```

Figure 2: BIT SETTING FAILURE

```
         .
         L    R0,FWORD         GET WORD WITH BITS
CSFAIL   EQU  *
         LR   R1,R0            COPY OLD VALUE
         N    R1,FLONBIT0      CREATE NEW VALUE
         CS   R0,R1,FWORD      REPLACE IF NO VALUE CHANGE
         BNE  CSFAIL           GO DO IT AGAIN IF VALUE CHANGE


SDATA    DSECT                 SHARED DATA STRUCTURE
FWORD    DS   0F               WORD ALTERED VIA CS
FL0      DS   X                FLAGS 0
FL0BIT0  EQU  X'80'            BIT 0 LABEL FOR TESTING ONLY
FL1      DS   X
  .      DS   H

FLONBIT0 DS   0F               AND MASK FOR FL0BIT0
         DC   AL1(255-FL0BIT0),X'FFFFFF'
```

Figure 3: BIT SETTING VIA CS

FL0NBIT0 must be created for "and and or masks" in CS sequences.

TESTING AND SETTING BITS

The Problem

Commonly encountered logic is

```
IF BITx=ON THEN
  DO;
  BITy=OFF;
  other logic;
  END;
```

There are many minor variations of this logic; the common point is that based on the value of a bit or bits, you wish to change a bit or bits and perform some other action; the change of bits must occur "atomically" with the test.

In both a multiprogramming and a parallel environment with shared data structures, the "normal" code for this type of logic will occasionally fail to perform as desired even though you include code to alter the bits via Compare and Swap. There are several ways the failure can occur; one is shown in Figure 4 on page 13. The basic problem is the timing window between testing the bits and setting the bits - parallel processes can hit this window and two processes will take action where the intent is that only one process will take action. The impact of this problem is significant: you basically have to evaluate every test of a bit in a shared data structure to see if the test and set problem applies.

```
TIME   CPU 1              MAIN STORAGE     CPU 2
---- ----------------- ------------ --------------------
  0                       A: 00
  1      TM    FL0,FL0BIT0
  2      BO    DONE
  3      L     R0,FWORD
  4 RT   LR    R1,R0
  5      O     R1,FL00BIT0                  TM   FL0,FL0BIT0
  6      CS    R0,R1,FWORD A: 80            BNZ  DONE
  7      BNE   RT                           L    R0,FWORD
  8      CALL  ONETIME               RT     LR   R1,R0
  9 DONE EQU *                              O    R1,FL00BIT0
 10                       A: 80             CS   R0,R1,FWORD
 11                                         BNE  RT
 12                                         CALL ONETIME
 13                                  DONE   EQU  *
----------------------------------------------------------

SDATA    DSECT                    SHARED DATA STRUCTURE
FWORD    DS   0F                  WORD ALTERED VIA CS
FL0      DS   X                   FLAGS 0
FL0BIT0  EQU  X'80'               BIT 0 LABEL FOR TESTING ONLY
FL1      DS   X
         DS   H

FL00BIT0 DS   0F                  OR MASK FOR FL0BIT0
         DC   AL1(FL0BIT0),X'000000'
```

Figure 4: TEST AND SET FAILURE

Code using a bit to perform some function "one time" will fail and perform the function twice.

Solutions

Holding a lock that, by agreement, covers the bits being altered will allow use of ordinary instructions to alter the bits and consistently achieve the proper result. To obtain a lock just for this function is also overkill.

Basically, we need an "atomic test and set" or an "if and only if" type of operation when we are dealing with shared data structures. Do not bother to investigate the System/370 TEST AND SET instruction, it's of no general value.[1] It turns out that a slight variation on the code that gets in trouble will perform correctly - see Figure 5 on page 14.

```
         L    R0,FWORD          GET WORD WITH BITS FIRST
RT       EQU  *
         TM   FL0,FL0BIT0       IS THE BIT ON?
         BO   DONE              BRANCH NO
         LR   R1,R0             COPY OLD VALUE
         O    R1,FL00BIT0       CREATE NEW VALUE
         CS   R0,R1,FWORD       REPLACE IF NO VALUE CHANGE
         BNE  RT                GO TRY IT AGAIN IF VALUE CHANGE
         CALL ONETIME           DO ONLY IF WE SET BIT
DONE     EQU  *

SDATA    DSECT                  SHARED DATA STRUCTURE
FWORD    DS   0F                WORD ALTERED VIA CS
FL0      DS   X                 FLAGS 0
FL0BIT0  EQU  X'80'             BIT 0 LABEL FOR TESTING ONLY
FL1      DS   X
         DS   H

FL00BIT0 DS   0F                OR MASK FOR FL0BIT0
         DC   AL1(FL0BIT0),X'000000'
```

Figure 5: TEST AND SET VIA CS

By saving a copy of the word containing the bits before the test, we ensure that any change up to the point where we atomically change the bit will be detected, causing us to look again. Note that you may test the bits in storage; there is no need to test the copy of the bits in register 0.

This technique works as long as all the bits involved in the test and set are within one word (or doubleword if you use CDS). The coding for this gets tedious in assembler and PL/AS: a nice candidate for a macro.

COUNTERS

The Problem

The simplest problem solved by Compare and Swap is that of a counter in a shared data structure. Without Compare and Swap, we have the same old problem: occasional updates will be lost in very much the same way as bit updates are lost as shown in Figure 2 on page 11. For some counters used for statistics, you may decide that complete accuracy is not necessary and specifically choose to use normal instructions which are faster and simpler. If you do this, I suggest you document it to avoid later confusion.

Solution

As described in Appendix A of [princop], basic Compare and Swap logic handles counters well - see the logic in Figure 6 on page 15.

```
         L     R0,COUNTER      GET OLD VALUE
RT       EQU   *
         LA    R1,1            INCREMENT VALUE
         AR    R1,R0           NEW VALUE
         CS    R0,R1,COUNTER   REPLACE IF NO VALUE CHANGE
         BNE   RT              GO TRY IT AGAIN IF VALUE CHANGE

SDATA    DSECT                 SHARED DATA STRUCTURE
COUNTER  DS    F               WORD ALTERED VIA CS
```

Figure 6: COUNTER SET VIA CS

CPU SERIALIZATION

The Problem

The fact that storage changes done on one CPU may not be immediately seen by another CPU can cause "timing" bugs whose probability of occurrence depends on load and processor implementation. This problem is not understood by many people who understand compare and swap very well.

An example of the problem is shown in Figure 7 on page 16. In the application, the "NORM" routine runs quite often. NORM's job is to alter field CB0 to 2. The process structure is such that no other process can conflict with this, so compare and swap is not necessary. On rare occasions, routine "EXC" will run on a parallel process and must ensure that *after* CB0 is set to 2, routine "DOIT" is called. To avoid a timing problem, EXC first sets a bit via CS, then looks at CB0. If CB0 is already set to 2, it assumes that NORM is unlikely to call DOIT, thus EXC calls DOIT. If CB0 is not set to 2, EXC assumes that NORM will see the bit set via CS and call DOIT. Calling DOIT twice will use extra instructions, but not do any damage.

Even if the two routines execute in the time relationship shown in Figure 7 on page 16, DOIT may not be called at all even though EXC is run. The reason is that the alteration of CB0 at time 1 will cause the cache of CPU 1 to be altered, but there is no CPU serialization operation run on CPU 1 to cause main storage and cache for CPU 2 to reflect the alteration. Because of this, EXC at time 12 still sees an old value of CB0 and expects NORM (which is already done) to run later and call DOIT.

```
TIME   CPU 1                        TIME      CPU 2
----  -----------------             ----    ---------------------
  0
  1 NORM MVI    CB0,2
  2      TM     FL0,FLOBIT0
  3      BZ     DONE
         CALL   DOIT
         L      R0,FWORD              7     EXC  L     R0,FWORD
    RT   LR     R1,R0                 8     RT   LR    R1,R0
         N      R1,FLONBIT0           9          O     R1,FLOOBIT0
         CS     R0,R1,FWORD          10          CS    R0,R1,FWORD
         BNE    RT                   11          BNE   RT
  4 DONE EQU    *                    12          CLI   CB0,2
                                     13          BNE   DONE
                                                 CALL  DOIT
                                                 L     R0,FWORD
                                            RT1  LR    R1,R0
                                                 N     R1,FLONBIT0
                                                 CS    R0,R1,FWORD
                                                 BNE   RT1
                                     14     DONE EQU   *
-----------------------------------------------------------------

SDATA      DSECT                       SHARED DATA STRUCTURE
FWORD      DS    0F                    WORD ALTERED VIA CS
FL0        DS    X                     FLAGS 0
FLOBIT0    EQU   X'80'                 BIT 0 LABEL FOR TESTING ONLY
FL1        DS    X
           DS    H
SDATA1     DSECT                       SHARED READ, UPDATE BY "NORM" PROCESS
CB0        DS    X                     CONTROL BYTE


FLOOBIT0   DS    0F                    OR MASK FOR FLOBIT0
           DC    AL1(FLOBIT0),X'000000'
FLONBIT0   DS    0F                    AND MASK FOR FLOBIT0
           DC    AL1(255-FLOBIT0),X'FFFFFF'
```

Figure 7: CPU SERIALIZATION FAILURE

Although CB0 is set to 2 at time 1 by CPU 1, CPU 2 at time 12 can still see the prior value of CB0 if CPU 1 has not not been forced to perform CPU serialization.

Solutions

The simplest solution is to cause CPU serialization at the proper time. This can be accomplished by adding a branch register 0 (BR R0) instruction to NORM before the test of FL0BIT0. This instruction causes CPU serialization (and is otherwise a no-op). Once this is done, the logic will work - if EXC is called, DOIT will be invoked at least once, regardless of timing.

Another solution would be to have NORM set CB0 using compare and swap. While not strictly necessary because the design prevents conflicts, compare and swap will cause CPU serialization.

CHAINS, LISTS, QUEUES

There are many variations of chains, lists, and queues that are commonly used in system programming. These mechanisms become more common as components and subsystems become more dynamic. In this section, we will refer to all of these as chains. Dealing with these techniques in a parallel environment ranges in difficulty from fairly simple to very complex. In choosing the solution to a given chaining problem it often helps to understand the absolute and relative frequencies of three operations against the chain: search, add, delete. If, for example, you never delete, things get simple. If there are many searches and few adds/deletes, you'd like to make searching the most efficient. If you rarely use the chain an efficient technique is not important[10].

In all the problems discussed in this section, the use of a lock to serialize access and management of a chain is a potential solution. The pros and cons of locks have been discussed earlier and will not be repeated here. There are cases where locks or another "single process" serialization technique are the only solution.

The Problem - Free Element List

An extremely common technique is the use of a "free element list" to keep track of resources available for use. The basic approach as shown in Figure 8 on page 18 begins a chain at an "anchor" in a control block. The anchor contains 0 (empty chain) or the address of the first element; each element contains a chain word which will point to the next element or contain 0 (end of chain).

In a single process environment with no multiprogramming or parallelism considerations, dealing with such a chain is easy. It is easy to manage the chain in FIFO (first in - first out) or LIFO (last in - last out) sequence.

In a multiprogramming or parallel environment with the chain being a shared data structure, management of the chain is more difficult. Appendix A of [princop] discusses this problem and provides a solution. A modified solution is presented here.

At first glance it seems that, as long as you're willing to use LIFO processing, a simple Compare and Swap sequence as shown in Figure 9 on page 18 will work. It turns out that there is a bug in this code; on occasion you'll lose elements or even worse, have an element that is actually in use also appear on the free chain. An occurrence of the problem is shown in Figure 10 on page 19.

10 Beware of system programmer myopia - systems and components usually must last more than a decade and tremendous changes in system scale occur. Such changes often turn one year's sensible design decision into another year's performance disaster.

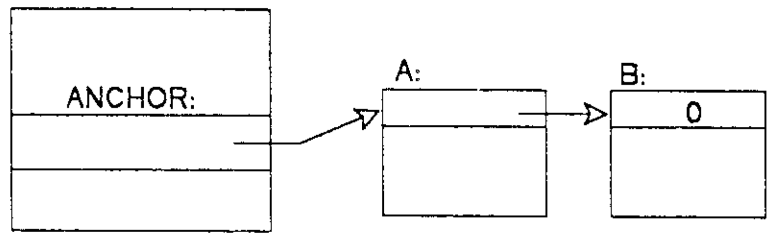

Figure 8: BASIC FREE ELEMENT LIST

```
PUTEL     EQU  *                 PUT R4 ELEMENT ON FREE CHAIN
          L    R2,ANCHOR         -> FIRST ELEMENT OR 0
PRETRY    EQU  *
          ST   R2,ELNEXT-EL(,R4) -> NEXT IN CHAIN OR 0
          CS   R2,R4,ANCHOR      ADD ELEMENT FROM CHAIN
          BNE  PRETRY            GO TRY IT AGAIN IF VALUE CHANGE

          . . . .

GETEL     EQU  *                 GET FREE ELEMENT
          L    R2,ANCHOR         -> FIRST ELEMENT OR 0
RT        EQU  *
          LTR  R2,R2             CHECK FOR EMPTY
          BZ   EMPTY             BRANCH IF IT IS
          L    R4,ELNEXT-EL(,R2) -> NEXT IN CHAIN OR 0
          CS   R2,R4,ANCHOR      REMOVE ELEMENT FROM CHAIN
          BNE  RT                GO TRY IT AGAIN IF VALUE CHANGE

          . . . .

SDATA     DSECT                  SHARED DATA STRUCTURE
ANCHOR    DS   F                 ANCHOR OF CHAIN OF FREE ELS

EL        DSECT                  ELEMENT
ELNEXT    DS   F                 POINTER TO NEXT FREE ELEMENT OR 0
```

Figure 9: ERRONEOUS FREE ELEMENT CHAIN LOGIC

This apparently good logic for putting an element on a free chain and removing it has a serious flaw that is described in Figure 10 on page 19.

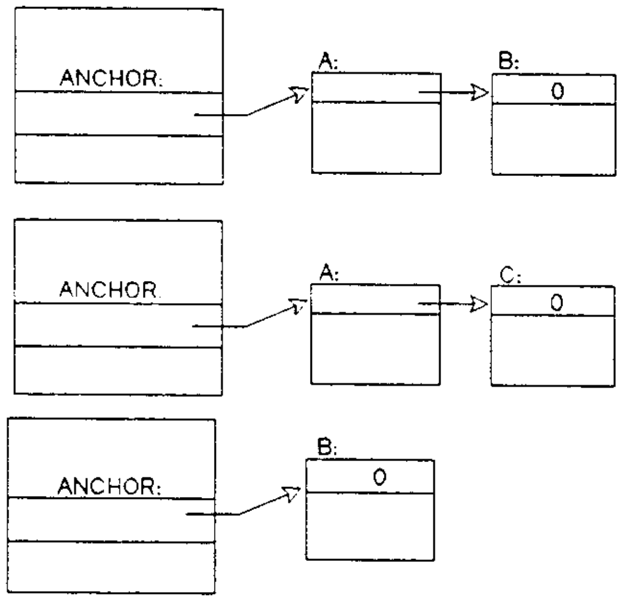

Figure 10: DESTROYING A CHAIN

The first picture shows the chain as it exists when the GETEL logic of Figure 9 on page 18 loads register 4 with the "next" pointer. This CPU is then interrupted. Before the interrupted process gets re-dispatched, the chain has been altered to that shown in the second picture by: allocate A, allocate B, free C, free A. At re-dispatch of the interrupted process, the Compare and Swap succeeds, giving the totally invalid chain shown in the third picture. Note that in addition to losing track of C, somebody could now get B as a free element when it is in fact already in use, creating a very nasty bug.

Solution - Free Element List

A slight change to the anchor structure and a change in the logic will solve the problem (for all practical purposes). We add an "allocation counter" and use Compare Double and Swap to detect the situation that gets us in trouble - where an element has been removed and replaced but it points to a different element than it used to. While [princop] uses CDS on GET and PUT, it is sufficient to do it on GET.[11] The good logic is shown in Figure 11 on page 20.

11 GET must have a way to atomically: a) Ensure that the first elements' chain field has not changed; b) Ensure that the anchor field has not changed; c) Change the anchor. To satisfy the first constraint, GET uses the CDS with allocation counter to detect that other GET's have occurred and thus the new anchor value must be re-fetched. PUT by itself cannot get into trouble. PUT must atomically: a) Ensure that the anchor field has not changed; b) Change the anchor. For this reason, a CS on the anchor field is sufficient for PUT.

```
PUTEL    EQU  *                  PUT R4 ELEMENT ON FREE CHAIN
         L    R2,ANCHORP         -> FIRST ELEMENT OR 0
PRETRY   EQU  *
         ST   R2,ELNEXT-EL(,R4) -> NEXT IN CHAIN OR 0
         CS   R2,R4,ANCHORP      ADD ELEMENT TO CHAIN (CS IS ENOUGH)
         BNE  PRETRY             GO TRY IT AGAIN IF VALUE CHANGE
         . . . .

GETEL    EQU  *                  GET FREE ELEMENT
         LM   R2,R3,ANCHOR       -> FIRST ELEMENT OR 0, COUNTER
RT       EQU  *
         LTR  R2,R2              CHECK FOR EMPTY
         BZ   EMPTY              BRANCH IF IT IS
         L    R4,ELNEXT-EL(,R2) -> NEXT IN CHAIN OR 0
         LA   R5,1               INCREMENT VALUE
         AR   R5,R3              NEW COUNTER VALUE
         CDS  R2,R4,ANCHOR       REMOVE ELEMENT FROM CHAIN
         BNE  RT                 GO TRY IT AGAIN IF VALUE CHANGE
         . . . .

SDATA    DSECT                   SHARED DATA STRUCTURE
ANCHOR   DS   0D                 ANCHOR DOUBLEWORD
ANCHORP  DS   F                  FIRST WORD, POINTER TO NEXT ELEMENT OR 0
ANCHORC  DS   F                  COUNT OF GETS

EL       DSECT                   ELEMENT
ELNEXT   DS   F                  POINTER TO NEXT FREE ELEMENT OR 0
```

Figure 11: FREE ELEMENT CHAIN LOGIC

It should be noted that the storage occupied by free elements cannot be safely freemained even though the element has been removed from the chain properly. The problem is that a process interrupted in the GETEL logic before the load of register 4 with the element's next pointer can program check (0C4) on the load instruction if, before the process was re-dispatched, the element was gotten, freemained and its page was invalidated. If its page was not invalidated, there's no problem because the CDS will fail. An intelligent recovery routine can recognize the program check, check the anchor for different values and, if so, assume the above problem and retry.

The counter used to get chain integrity may have some functional value: it will contain a count of the number of allocations. An occasionally useful variation allows you to keep an "in use" count as well - it is shown in Figure 12 on page 21. Your use count will wrap to 0 if you have more than 65,535 elements in use. There is no wrap problem with the activity count halfword.

```
PUTEL     EQU  *                 PUT R4 ELEMENT ON FREE CHAIN
          LM   R2,R3,ANCHOR      -> FIRST ELEMENT OR 0, COUNTERS
PRETRY    EQU  *
          ST   R2,ELNEXT-EL(,R4) -> NEXT IN CHAIN OR 0
          L    R5,PLUSMIN        ADD TO COUNTER, DECREMENT BUSY
          ALR  R5,R3
          CDS  R2,R4,ANCHOR      ADD ELEMENT TO CHAIN
          BNE  PRETRY            GO TRY IT AGAIN IF VALUE CHANGE

          . . . .

GETEL     EQU  *                 GET FREE ELEMENT
          LM   R2,R3,ANCHOR      -> FIRST ELEMENT OR 0, COUNTER
RT        EQU  *
          LTR  R2,R2             CHECK FOR EMPTY
          BZ   EMPTY             BRANCH IF IT IS
          L    R4,ELNEXT-EL(,R2) -> NEXT IN CHAIN OR 0
          L    R5,DOUBLE1        INCREMENT VALUE
          ALR  R5,R3             NEW COUNTER VALUE
          CDS  R2,R4,ANCHOR      REMOVE ELEMENT FROM CHAIN
          BNE  RT                GO TRY IT AGAIN IF VALUE CHANGE

          . . . .

DOUBLE1   DS   0F
          DC   X'00010001'       SIMULTANEOUS ADD 2 HALFWORDS.
PLUSMIN   DS   0F
          DC   X'0000FFFF'       INCREMENT LEFT HW, DECREMENT RIGHT HW
*                                AS LONG AS RIGHT HW > 0.

SDATA     DSECT                  SHARED DATA STRUCTURE
ANCHOR    DS   0D                ANCHOR DOUBLEWORD
ANCHORP   DS   F                 FIRST WORD, POINTER TO NEXT ELEMENT OR 0
ANCHORC   DS   H                 COUNT OF GETS,PUTS
ANCHORB   DS   H                 COUNT OF CURRENTLY "BUSY" ELEMENTS

EL        DSECT                  ELEMENT
ELNEXT    DS   F                 POINTER TO NEXT FREE ELEMENT OR 0
```

Figure 12: FREE ELEMENT CHAIN LOGIC WITH BUSY COUNT

This logic will manage the chain and give you an accurate "in use" count. The cost is the requirement to do CDS on both PUT and GET. Note that the logic at "EMPTY", if it creates an element and returns the element to the caller, must increment the busy count via CS.

The Problem - FIFO Processing

The basic operation of CS or CDS on chains results in Last-in First-out (LIFO) processing. This is probably good for free resource chains - LIFO would have better cache and paging effects than First-in First-out (FIFO) processing. In many other applications. FIFO processing is either highly desirable or required.

In a parallel or multiprogramming environment, FIFO processing requires special logic because it is generally unsafe to scan the chain to find the "first chained" element while the chain is being altered by other processes that are removing elements. This goes back to a basic problem - chain scanning when delete is allowed requires special protection.

Solutions - FIFO Processing, Single Process

In a parallel or multiprogramming environment, when only one process removes elements from a chain, it may do FIFO processing by scanning the chain to the end even though multiple processes are adding to the chain as long as all alterations to ANCHOR use CS or CDS.

Unless otherwise mentioned, all these approaches:

1. Allow code that adds to the chain to use a simple CS, as shown by the PUTEL logic in Figure 14 on page 26. CDS with an allocation counter is not required because there will be no conflict between processes trying to remove elements.
2. Require the remover of an element to remove the element via CS if it is being removed from the anchor.

Locking

The most obvious and simple approach, as usual, is to have a process get a lock before scanning the chain and free the lock after removing an element.

One application of this solution is the lock service itself. A lock service desiring to resume the first waiting requestor of a lock is already serialized - it holds the lock in question. Thus, assuming that convoy effects are to be ignored and only one waiter is to be posted, a simple chain scan is all that is required - not the complex mechanism suggested in Appendix A of [princop] under "LOCK/UNLOCK with FIFO Queueing for Contentions".

Even if the lock is not exclusive mode only, support for a shared/exclusive lock can have a single process scan the wait list if waiters queue up while many processes are sharing the lock. Assuming that all requests wait once an exclusive request is made, the process that decrements the share count to 0 is the one who performs the scan.

Single Process Design

By designing the logic such that only one specific process will scan the chain, remove elements and, presumably, process them. For many environments, this "single server" design is quite reasonable. The essential difference between this approach and locking is that this approach has only one process; with locking, the serialized work is done under many processes, but only one at any given time. Evaluation of the viability of a single server design is similar in some ways to evaluating locking performance:

- Will there be enough capacity on a single CPU of an "n" way machine to support requests generated by the entire machine?
- Will the single server be responsive enough given its use of only one CPU and the delays caused by activities like page faults and i/o?

Solutions - FIFO Processing, Parallel Processes

Approximate FIFO

If multiple processes are to perform some operation against an item, then even if each item is removed from a chain in strict FIFO sequence, actual processing of each item may not occur in strict FIFO sequence since several processes may be running in parallel subject to their own page faults, preemptions, etc. This type of environment might be called "approximate FIFO" since it approximates FIFO processing but does not guarantee it.

Recognizing the approximate nature of the processing, you may be willing to make it slightly more approximate by having two anchors for the chain: LIFO and FIFO. Logic to remove an element is shown in Figure 13 on page 24 and described below.

Processes adding to the chain do the normal Compare and Swap on the LIFO anchor.

Processes looking for work look on the FIFO chain first, attempting to remove the first element from it via Compare and Swap. If the FIFO chain is empty, the process takes the entire LIFO chain using Compare and Swap, reverses the order of the chain to FIFO sequence (taking the last one (first in) for processing), then places any remaining items carefully on the FIFO chain. While this logic requires a fair amount of code, the most probable path (one element on the LIFO chain) takes only a few more instructions than standard LIFO GETEL logic.

As you can see, there are windows where items will be selected in a non-FIFO sequence[12], thus we would process them more out of sequence than a scheme that always selected them in sequence.

12 While one process has swapped a chain off the LIFO anchor and is re-ordering it, other processes can add to the LIFO chain and yet other processes can remove either a single element from the LIFO anchor (and process it), or a chain of elements which may be re-ordered and placed on the FIFO anchor before the first process puts its chain on the FIFO anchor.

```
GETEL    EQU  *                GET FREE ELEMENT
         LM   R2,R3,FIANCHOR   FIFO -> FIRST ELEMENT OR 0
         USING EL,R2
RT       EQU  *
         LTR  R2,R2            CHECK FOR EMPTY
         BZ   FIEMPTY          BRANCH IF IT IS
         L    R4,ELNEXT-EL(,R2) -> NEXT IN CHAIN OR 0
         LA   R5,1             INCREMENT VALUE
         ALR  R5,R3            NEW COUNTER VALUE
         CDS  R2,R4,FIANCHOR   REMOVE ELEMENT FROM CHAIN
         BE   DONE             GO PROCESS ELEMENT
         BNE  RT               GO TRY IT AGAIN IF VALUE CHANGE
FIEMPTY  EQU  *                FIFO CHAIN IS EMPTY
         L    R2,LIANCHOR      LIFO -> FIRST ELEMENT OR 0
         LTR  R2,R2            CHECK FOR EMPTY
         BZ   NOWORK           PROBABLY NO WORK, BUT NOT FOR SURE
         SLR  R3,R3
         CS   R2,R3,LIANCHOR   SWAP ENTIRE CHAIN OFF
         BNE  GETEL            TRY FROM THE START IF THINGS CHANGE
         CLC  ELNEXT,=F'0'     NORMAL CASE: IS THIS THE ONLY ONE?
         BE   DONE             YES - GO PROCESS ELEMENT
*                              MUST REORDER THE LIST
         LR   R6,R2            SAVE -> TO LAST IN REORDERED CHAIN
         SLR  R4,R4            ZERO FOR FIRST PASS IN THE LOOP
RESEQ    EQU  *                LOOP TO REVERSE SEQUENCE
         L    R5,ELNEXT
         ST   R4,ELNEXT
         LR   R4,R2            NEW PREVIOUS
         LTR  R2,R5            NEW CURRENT OR DETECT END
         BNZ  RESEQ
*                              FIFO CHAIN: R4->FIRST, R6->LAST
*                              CHAIN OF LAST NEED SETTING
         L    R2,ELNEXT-EL(,R4) -> TO SECOND ELEMENT
         LM   R8,R9,FIANCHOR   FIFO ANCHOR
RESEQ20  EQU  *                CDS RETRY
*                              COVER CASE WHEN FIFO CHAIN IS NOT EMPTY
*                              BY PLACING OUR CHAIN IN FRONT OF CHAIN
*                              ALREADY THERE, POSSIBLY GETTING THINGS
*                              QUITE OUT OF SEQUENCE.
         ST   R8,ELNEXT-EL(,R6) CHAIN TO IT FROM OUR LAST
         LA   R5,1
         ALR  R5,R9
         CDS  R8,R4,FIANCHOR   MY BUNCH FIRST, THEN PRIOR BUNCH
         BNE  RESEQ20          BRANCH IF BUSY
DONE     EQU  *                R2->GOTTEN ELEMENT
NOWORK   EQU  *                R2 = 0, NO ELEMENTS AVAILABLE


SDATA    DSECT                 SHARED DATA STRUCTURE
FIANCHOR DS   0D               ANCHOR DOUBLEWORD - FIFO CHAIN
         DS   F                FIRST WORD, POINTER TO NEXT ELEMENT OR 0
         DS   F                SECOND WORD, ACTIVITY COUNT FOR SAFETY
LIANCHOR DS   F                ANCHOR FOR LIFO CHAIN

EL       DSECT                 ELEMENT
ELNEXT   DS   F                POINTER TO NEXT FREE ELEMENT OR 0
```

Figure 13: APPROXIMATE FIFO

Parallel FIFO Removal

With certain restrictions, this scheme allows parallel addition to a chain and provides for FIFO removal from the chain by parallel processes. The restrictions exist because the removal logic may be examining and even attempting to alter the chain fields of an element that has already been removed from (or even re-inserted into) the chain:

- The chain area of the element must not be altered by any code other than GETEL, PUTEL.
- After an element has been placed on the chain once, the element's storage may not be freed even though the element is no longer on the chain (if storage is freed, other processes searching the chain which may have obsolete data pointing to this element may program check when referencing this element). There is a way around this problem - described later in this section.

The code to perform parallel FIFO removal is shown in Figure 14 on page 26.

```
PUTEL    EQU  *                      PUT R4 ELEMENT ON FREE CHAIN
         L    R2,ANCHORP             -> FIRST ELEMENT OR 0
PRETRY   EQU  *
         ST   R2,ELNEXT-EL(,R4)      -> NEXT IN CHAIN OR 0
         CS   R2,R4,ANCHORP          ADD ELEMENT TO CHAIN
         BNE  PRETRY                 GO TRY IT AGAIN IF VALUE CHANGE
         . . . .

GETEL    EQU    *                    GET AN ELEMENT, FIFO
         SLR    R0,R0                0 VALUE
RESTART  EQU    *
         LM     R2,R3,ANCHOR         POINTER, COUNTER
         LTR    R2,R2                ANY ELEMENTS?
         BZ     DONE                 BRANCH NO
         LA     R4,ANCHOR            ->DOUBLEWORD
         USING EL,R2
SCAN     EQU    *                    SCAN FOR THE LAST ON THE CHAIN
         C      R0,ELNEXT            IS THERE A NEXT ELEMENT?
         BE     MAYBE                BRANCH NO
         LA     R4,ELDWORD           ->DOUBLEWORD
         LM     R2,R3,ELDWORD        POINTER, COUNTER
         LTR    R2,R2                NOW END OR OFF OF CHAIN?
         BNZ    SCAN                 BRANCH NO
         B      RESTART              BRANCH YES - START FROM BEGINNING
MAYBE    EQU    *                    AT ONE TIME, R2 EL WAS LAST
         LA     R1,1                 INCREMENT VALUE
         ALR    R1,R3                NEW COUNTER VALUE
         CDS    R2,R0,0(R4)          SWAP LAST ELEMENT OFF CHAIN
         BNE    RESTART              IF FAIL, SOMEONE ELSE GOT IT
DONE     EQU    *                    R2->ALLOCATED ELEMENT OR R2 = 0
         . . . .

SDATA    DSECT                       SHARED DATA STRUCTURE
ANCHOR   DS     0D
ANCHORP  DS     F                    A(FIRST ELEMENT) OR 0
         DS     F                    COUNT: NEXT ELEMENT REMOVED

EL       DSECT
ELDWORD  DS     0D                   AREA ONLY ALTERABLE BY GETEL, PUTEL
ELNEXT   DS     F                    A(NEXT ELEMENT) OR 0
         DS     F                    COUNT: NEXT ELEMENT REMOVED
```

Figure 14: PARALLEL FIFO REMOVAL

Note that this logic is somewhat slow if the average queue/chain length is large. With long chains, the approximate FIFO logic performs better.

It turns out that an obligation passing approach can be used to free storage for an element if that capability is required. We must add a doubleword containing a "GETEL use count" and a delete chain anchor to the shared data structure. Any process desiring to free the storage for an element first checks to see if there are any active GETEL processes by looking at the GETEL use count. If there are none, the element can actually be deleted (freemained or used for something else or whatever). If there are active GETEL processes, the obligation to delete the element is passed on by chaining the element off of the delete chain anchor using a delete chain field in the element. Before entering the scan portion of GETEL, we increment the "GETEL use count" - effectively a share mode lock. Once through the GETEL logic, we decrement the use count and, if we decrement it to zero, remove the entire chain (if any) of elements to be deleted and actually delete them.

When obligation passing delete support is added, good recovery logic becomes more important: a failure in GETEL is likely to leave the GETEL use count incremented - if it's not decremented you will be unable to actually delete any elements.

```
DELEL    EQU   *                      DELETE ELEMENT THAT ONCE WAS
*                                     ON THE CHAIN (ANCHORP). R2->EL
         USING EL,R2
         LM    R0,R1,DANCHOR          DELETE ANCHOR POINTER
DELEL10  EQU   *                      CS RETRY
         LTR   R1,R1                  ACTIVE GETEL PROCESSES?
         BP    DELEL20                BRANCH YES, PASS THE OBLIGATION
         CALL  DELETE2                ACTUALLY DELETE THE R2 ELEMENT
         B     DELEL30                DONE
DELEL20  EQU   *                      PASS ON DELETE OBLIGATION
         ST    R0,ELDNEXT             CHAIN OFF NEW ELEMENT
         LR    R3,R1                  SAME USE COUNT
         CS    R0,R2,DANCHORP         PUT NEW ELEMENT ON CHAIN
         BNE   DELEL10                BRANCH IF BUSY
         . . . .
         . . . .

GETEL    EQU   *                      GET AN ELEMENT, FIFO
         L     R1,DANCHORC            GETEL USE COUNT
GET10    EQU   *                      CS FETRY
         LA    R2,1
         ALR   R2,R1
         CS    R1,R2,DANCHORC         INCREMENT USE COUNT
         BNE   GET10

         . . . .                      OTHER PARALLEL FIFO REMOVAL CODE
                                      GOES HERE UNCHANGED

DONE     EQU   *                      R2->ALLOCATED ELEMENT OR R2 = 0
         LM    R0,R1,DANCHOR
DONE10   EQU   *                      CDS RETRY
         LR    R5,R1                  COPY USE COUNT
         BCT   R5,DONE20              DECREMENT, BRANCH IF NOT 0
         SLR   R4,R4                  NEW CHAIN ORIGIN VALUE
         CDS   R0,R4,DANCHOR          SET BOTH WORDS TO ZERO
         BNE   DONE10                 RETRY
         LTR   R0,R0                  DID WE GET A DELETE CHAIN?
         BZ    DONE40                 BRANCH NO
         CALL  DELETE1                DELETE CHAIN STARTING WITH R0
         B     DONE40
DONE20   EQU   *                      USE COUNT > 0
         CS    R1,R5,DANCHORC         APPLY MY DECREMENT
         BNE   DONE                   GO REFRESH BOTH REGISTERS
DONE40   EQU   *                      REALLY DONE, R2 STILL SET
         . . . .

SDATA    DSECT                        SHARED DATA STRUCTURE
ANCHOR   DS    0D
ANCHORP  DS    F                      A(FIRST ELEMENT) OR 0
         DS    F                      COUNT: NEXT ELEMENT REMOVED
DANCHOR  DS    0D                     DELETE ELEMENT CONTROL
DANCHORP DS    F                      A(FIRST ELEMENT) OR 0
DANCHORC DS    F                      COUNT: NEXT ELEMENT REMOVED

EL       DSECT
ELDWORD  DS    0D                     AREA ONLY ALTERABLE BY GETEL, PUTEL
ELNEXT   DS    F                      A(NEXT ELEMENT) OR 0
         DS    F                      COUNT: NEXT ELEMENT REMOVED
ELDNEXT  DS    F                      A(NEXT ELEMENT ON DELETE CHAIN) OR 0
```

Figure 15: PARALLEL FIFO REMOVAL WITH DELETION

The additional code required to support safe deletion (freemain) of elements that have once been on the chain.

The Problem - Find and Remove by Name

A more complex class of problem than FIFO processing is the case where you must be able to find a specific element on a chain and be able to remove it from the chain no matter where it is in the chain. Typically, you would be trying to find an element that was uniquely identified by some bit-string (NAME).

This problem is somewhat difficult, primarily because you need to remove items from the middle of a chain and you need to scan the chain. Figure 16 on page 30 provides an example of what can go wrong. Note that parallelism is not required to cause the problem shown - multiprogramming is sufficient.

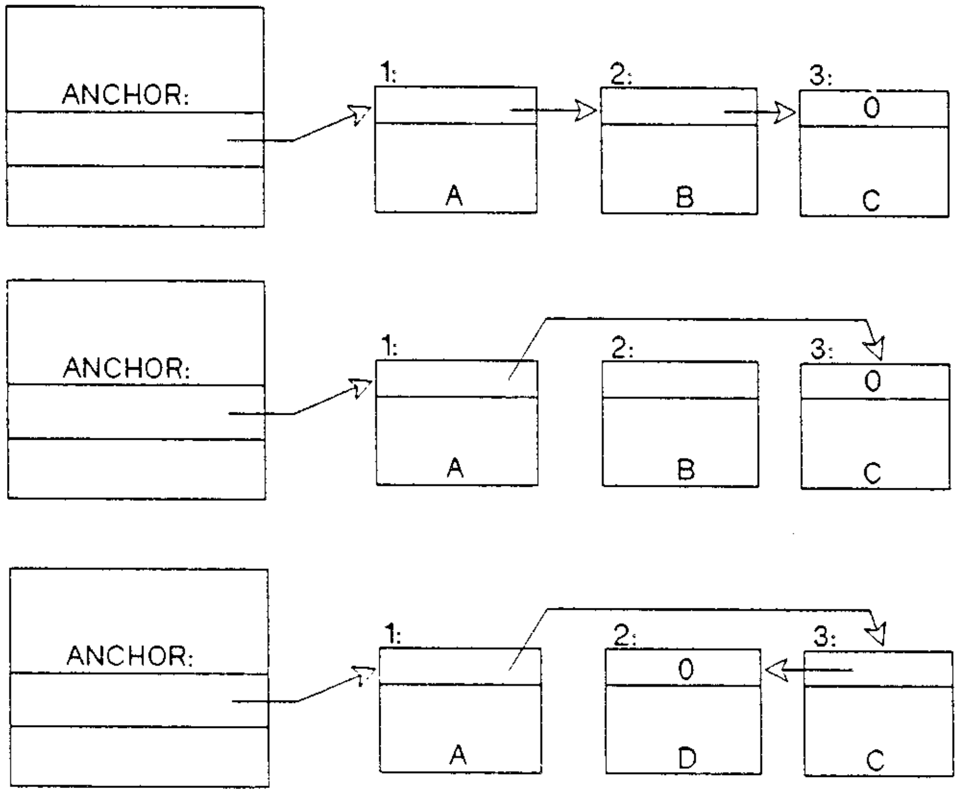

Figure 16: PROBLEM: FIND and REMOVE by NAME

This figure illustrates one possible sequence that could occur using simple logic to search a chain and remove an element, even if compare and swap is used for removal.

1. The first figure shows the chain as process 1, searching for element named C, gets the "next pointer" (value 2) from the element at location 1. This process is then suspended due to preemption or a page fault.

2. The second figure shows the chain after process 2 finds and removes the element named B located at address 2.

3. The third figure shows the chain after process 2 changes the name of the element at address 2 to D and adds the element into the chain in name sequence.

4. Process 1 then resumes and looks at the element at address 2, finds name D and concludes that C is not on the chain!

What would happen if process 2 freemained the element at address 2 (and freemain invalidated the page in the page table)?

What would happen if process 2 used the storage at address 2 for some other purpose?

Solutions - Find and Remove by Name

This problem can be resolved by the single process approach, but this eliminates parallelism for finding, adding, and removing elements. Once an element is found and either removed or protected from deletion, the element can be used by parallel processes.

If chains are to be searched, it is a good idea to use one or more search techniques such as ordering by name and/or hashing [rjhash]. Hashing is particularly nice - it is simply a bunch of lists, each of which can be handled using these serialization techniques.

Locking

As usual, locking can be used to solve the problem. A lock with shared and exclusive modes as shown in Figure 1 on page 6 is the best solution.

- If the request is merely to search for a named element but leave it on the chain, the lock is obtained in share mode and the chain is searched by merely following the chain.
- If the request is to remove a named element from the chain, the lock is obtained in exclusive mode, the chain is searched and the element may be removed with ordinary instructions.
- If the request is to add a named element then the lock is obtained in exclusive mode and the element added to the chain with ordinary instructions. If names must be unique, you should search for the name while holding the exclusive lock before adding the new element to the chain.

Obligation Passing

Obligation passing may be used in this situation if appropriate to the application. As used here, obligation passing controls processing such that one and only one process at a time will be manipulating chains for the purposes of element deletion.

If other processes want to perform chain manipulation at the same time, they pass their obligation to the last process dealing with the structure. This variation of obligation passing, developed by Ron Obermarck, is somewhat complex and will be described with words as well as an assembler language example (found in "Appendix A. ASSEMBLER EXAMPLES, BY NAME, OBLIGATION PASSING" on page 36).

One potential shortcoming: physical removal of deleted items (and the ability to re-use/free the storage) can take "a while" because it can only be triggered when the chain goes "idle" (no processes in search or delete). In most applications, this should not be a problem since there are no built-in suspends in the logic. In an extremely high usage application with lots of parallelism, it might become a problem.

The actual application being used for the example needs the following functions:

1. ADD an element to the list.
2. FIND a specific (named) element for shared usage. Support for exclusive usage could be easily added.
3. UNFIND an element from shared usage established by a prior FIND.

4. DELETE a specific (named) element - make it no longer eligible for FIND and free its storage as soon as possible.

All of these functions must be accessible from parallel processes and the functions must not suspend (or spin for a lock).

ADD is accomplished by standard compare and swap of the new element onto the chain anchor producing a LIFO ordered chain. An assembler example, essentially the same as the PUTEL logic in Figure 11 on page 20, is shown in Figure 19 on page 37.

FIND is accomplished by incrementing a use count (basically a shared-mode-only lock) that is associated with the entire chain. The chain is then searched for the requested element. If the element is found, an element use count is incremented (using Compare and Swap logic) as long as the element is not marked as "logically deleted". If the element is logically deleted, it is treated as "not found". The following case is covered: a named element is logically deleted then an element with the same name is added. Before returning to the caller, a RELEASE routine is called to decrement the chain's use count and handle any passed obligations.

An assembler example is shown in Figure 20 on page 38.

UNFIND is accomplished by decrementing an element's use count using compare and swap. The chain use count is not required for this operation.

DELETE is begun by incrementing the chain's use count. The chain is then searched for the requested element. If the element is found, the element is not marked as logically deleted (this check protects against parallel processes trying to delete the same element), and the element use count is zero (a safety check), then the element is marked as logically deleted (using Compare and Swap logic). If we successfully mark the element as logically deleted, we add the element to a delete chain. Note that we do not alter the primary element chain here since it is not safe. Before returning to the caller, a RELEASE routine is called to decrement the chain's use count, handle any passed obligations, and, if we are the only current user of the chain, perform further delete actions on the element we may have placed on the delete chain.

Results of DELETE processing are shown pictorially in Figure 17 on page 33. An assembler example is shown in Figure 21 on page 39.

RELEASE is a function invoked by FIND and DELETE to decrement the chain use count. If it finds the use count to be 1 (this is the only process working on the chain), it then looks for delete work (indicated by elements chained from the delete anchor). A first pass removes elements from the primary element chain (this is safe because only one process is doing the chain alteration; processes that start FIND or DELETE will be OK because the old elements and their chain fields are still valid).

A second pass begins if the chain use count is 1 after the first pass completes. The second pass can actually free elements that were removed from the primary chain by an earlier pass. Freeing the elements is safe because no process can be searching the primary chain with the address of these elements since they were off the primary chain before we found that no other processes were looking at the chain (use count of 1 at the beginning of pass 2).

The actual logic is a bit more complicated because it must account for DELETE action occurring during any of the passes and ensure that the chain use count never reaches 0 while there are elements on the delete chain.

Results of RELEASE processing are shown pictorially in Figure 17 on page 33. An assembler example is shown in Figure 22 on page 40 and Figure 23 on page 41.

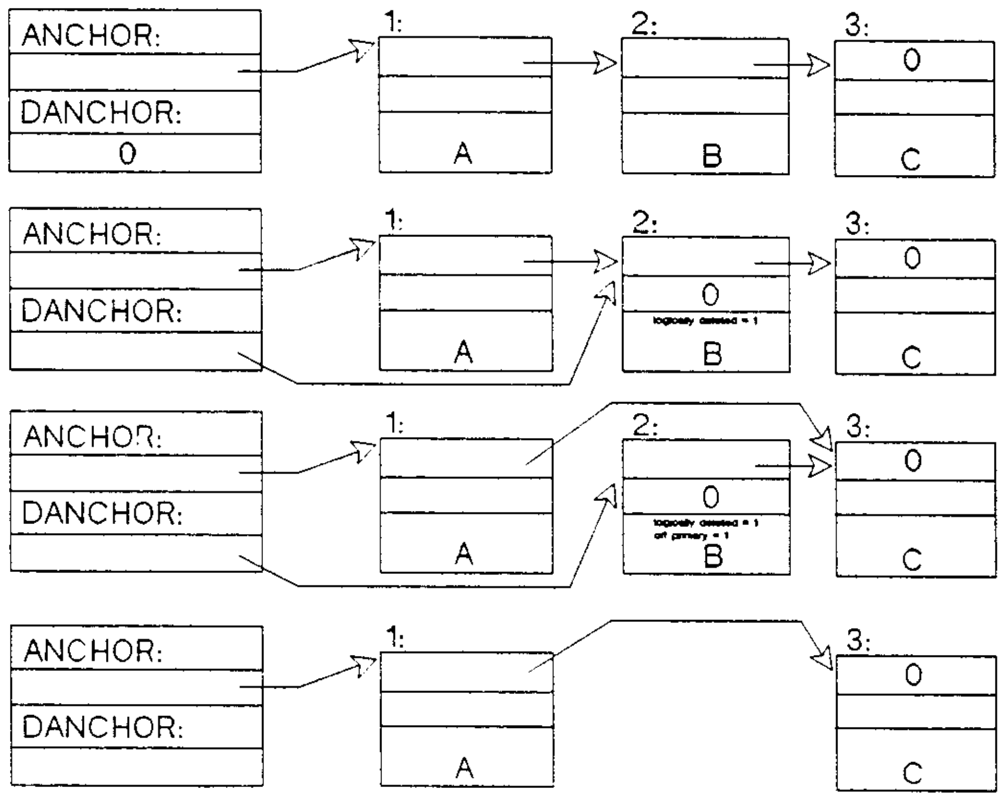

Figure 17: DELETE FROM MIDDLE VIA OBLIGATION PASSING

1. The first figure shows the chain with no delete activity, possibly many processes searching it.

2. The second figure shows the chain after a process running the DELETE function on the element named B located at address 2 has finished all but the call to RELEASE. If, when RELEASE is called by the DELETE process, there are other processes busy with the chain, the DELETE process will make no further changes to the structure and the physical delete obligation has been passed.

3. The third figure shows the chain after a process running FIND or DELETE logic calls RELEASE and the chain use count is found to be 1 and the first pass of physical delete has completed by altering the primary chain and marking the element as "off primary chain". Note that the pointer from DANCHOR to element B would not exist unless at the beginning of pass 2, RELEASE logic found a chain use count of greater than 1, in which case this process would decrement the use count and return, passing the obligation for stage 2 of physical delete.

4. The fourth figure shows the chain after the second pass of RELEASE has completed; all deletion work has completed. Note that it is possible that three different processes are involved in the deletion of element B.

SUMMARY

Coding in a parallel environment can be complex and error prone. The first step to success, hopefully provided by this paper, is understanding the problems and understanding the solutions available.

The simplest solution to problems of parallelism appears to be the use of locks, but they introduce subtle capacity and performance problems. A large number of parallelism problems can be dealt with using Compare and Swap as shown above. Use of Compare and Swap techniques to avoid locks is strongly suggested where at all possible. As of 1986, large /370 processors have 4 CPU's - parallelism is real and indiscriminate use of locking can defeat it.

ACKNOWLEDGEMENTS

The content of this document has many sources, some of which are listed in the bibliography, some are just "systems programming lore" and cannot be attributed. I am indebted to Ron Obermarck and Kurt Shoens for significant assistance in "coping with parallelism".

APPENDIX A. ASSEMBLER EXAMPLES, BY NAME, OBLIGATION PASSING

This appendix contains assembler language examples of logic to use obligation passing to support a "find by name" with delete capability.

```
* COPYRIGHT IBM CORPORATION 1986
* DATA AREAS AND EQUATES USED
*
LISTANCH DC   A(0)                LIST ANCHOR
SHRLATCH DS   0D                  DOUBLE-WORD BOUNDARY
RCOUNT   DC   F'0'                COUNT OF CURRENT USERS OF LATCH
DELTANCH DC   A(0)                ANCHOR FOR PENDING DELETES
*
* DEFINE RETURN-CODE VALUES RETURNED TO CALLER IN R15
*
SUCCESS  EQU  0                   VALUE RETURNED IF SUCCESS
NOSUCCES EQU  4                   VALUE RETURNED IF NO SUCCESS
*
* DEFINE THE LIST ELEMENT STRUCTURE AS A DUMMY SECTION
*
ELEMENT  DSECT ,                  DUMMY SECTION -
ELEMPCHN DS    A                  PRIMARY CHAIN FOR LIFO QUEUE
ELEMACHN DS    A                  ALTERNATE CHAIN
ELEMIDEN DS    F                  ELEMENT IDENTIFIER
ELEMCTFL DS    0F                 ELEMENT IN-USE COUNTER AND DELETE FLAG
USECTR1  EQU   X'00000100'        USE-COUNT 1'S POSITION VALUE IN WORD
ELEMUSCT DS    FL3                ELEMENT IN-USE COUNTER (0 = NO USERS)
ELEMUSFL DS    X                  ELEMENT DELETE FLAGS
DELETED  EQU   B'00000001'        FLAG BIT - IF 1, ELEMENT DELETED
OFFCHAIN EQU   B'00000010'        FLAG BIT - IF 1, ELEMENT UNCHAINED
ELEMDATA DS    C                  DATA BEGINS HERE
```

Figure 18: Sample data structures for List Manipulation

```
* COPYRIGHT IBM CORPORATION 1986
ADD      STM   R14,R12,12(13)        SAVE INPUT REGISTERS
         L     R9,0(,R1)             GET POINTER TO ELEMENT TO ADD
         USING ELEMENT,R9            ASSIGN BASE REGISTER
         SLR   R8,R8                 GET A ZERO
         ST    R8,ELEMCTFL           SET FLAGS AND COUNT TO NULL
*
         L     R8,LISTANCH           CURRENT ANCHOR CONTENT
ADD01    ST    R8,ELEMPCHN           LIFO CHAIN
         CS    R8,R9,LISTANCH        ATTEMPT ANCHOR UPDATE
         BNE   ADD01                 LOOP IF NOT SUCCESS
*
         LM    R14,R12,12(13)        RESTORE REGISTERS
         LA    R15,SUCCESS           ALWAYS SUCCESSFUL
         BR    R14                   RETURN
         DROP  R9                    RELEASE BASE REGISTER
```

Figure 19: Sample Implementation of ADD for List Manipulation

```
* COPYRIGHT IBM CORPORATION 1986
FIND     STM   R14,R12,12(R13)      SAVE CALLERS REGS
         L     R2,0(,R1)            POINTER TO IDENTIFIER
         L     R6,0(,R2)            GET IDENTIFIER ITSELF
         L     R5,4(,R1)            GET POINTER TO FOUND BLOCK
*
         L     R0,RCOUNT            CURRENT READER COUNT
FIND01   LA    R1,1                 COUNTER INCREMENT
         AL    R1,R0                NEW COUNTER VALUE
         CS    R0,R1,RCOUNT         ATTEMPT THE CHANGE
         BNZ   FIND01               LOOP UNTIL SUCCESS
*
         L     R9,LISTANCH          GET A(FIRST ELEMENT IN LIST)
         USING ELEMENT,R9           SET AS BASE
         B     FIND02               ENTER AT TEST FOR WHILE-LOOPS
*
FIND02L  L     R9,ELEMPCHN          TO NEXT IN CHAIN
FIND02   LTR   R9,R9                TEST FOR NULL POINTER
         BZ    FIND02X              EXIT LOOP IF NULL
         C     R6,ELEMIDEN          COMPARE FOR IDENT TO DELETE
         BNE   FIND02L              LOOP IF NOT EQUAL
FIND02X  EQU   *                    SEARCH EXIT
         ST    R9,0(,R5)            SAVE POINTER TO BLOCK OR NULL
         LTR   R9,R9                TEST FOR NULL POINTER
         BZ    FINDERRX             EXIT IF NULL(SUCCESS = OFF)
*
         L     R0,ELEMCTFL          CURRENT USER COUNT
FIND03   LA    R1,DELETED           FLAG VALUE FOR DELETED
         NR    R1,R0                AND OLD VALUE WITH FLAG
         BNZ   FINDERRX             DELETED - EXIT NO SUCCESS
         LA    R1,USECTR1           COUNTER INCREMENT
         AL    R1,R0                NEW COUNTER VALUE
         CS    R0,R1,ELEMUSCT       ATTEMPT THE CHANGE
         BNZ   FIND03               LOOP UNTIL SUCCESS
*
*    ELEMENT IS NOW RESERVED FOR THE CALLER - RETURN SUCCESS
         CALL  RELEASE              RESET SHARED LATCH
         LM    R14,R12,12(13)       RESTORE REGISTERS
         LA    R15,SUCCESS          SET SUCCESS RETURN CODE
         BR    R14                  RETURN
*
*    ELEMENT WAS NOT FOUND - RETURN NO SUCCESS
FINDERRX CALL  RELEASE              RESET SHARED LATCH
         LM    R14,R12,12(13)       RESTORE REGISTERS
         LA    R15,NOSUCCES         SET ERROR RETURN CODE
         BR    R14                  RETURN
         DROP  R9                   RELEASE BASE REGISTER
```

Figure 20: Sample Implementation of FIND for List Manipulation

```
* COPYRIGHT IBM CORPORATION 1986
DELETE   STM   R14,R12,12(R13)         SAVE CALLERS REGS
         L     R1,0(,R1)               POINTER TO IDENTIFIER
         L     R6,0(,R1)               GET IDENTIFIER ITSELF
         L     R0,RCOUNT               CURRENT READER COUNT
DLET01   LA    R1,1                    COUNTER INCREMENT
         AL    R1,R0                   NEW COUNTER VALUE
         CS    R0,R1,RCOUNT            ATTEMPT THE CHANGE
         BNZ   DLET01                  LOOP UNTIL SUCCESS
         L     R9,LISTANCH             GET A(FIRST ELEMENT IN LIST)
         USING ELEMENT,R9              SET AS BASE
         B     DLET02                  ENTER AT TEST FOR WHILE-LOOPS
DLET02L  L     R9,ELEMPCHN             TO NEXT IN CHAIN
DLET02   LTR   R9,R9                   TEST FOR NULL POINTER
         BZ    DLETERRX                EXIT LOOP IF NULL - (NOT FOUND)
         C     R6,ELEMIDEN             COMPARE FOR IDENT TO DELETE
         BNE   DLET02L                 LOOP IF NOT EQUAL
*  CODE FOR LOGICAL DELETION
         L     R8,ELEMCTFL             GET USE COUNTER AND FLAGS
         LA    R6,DELETED              FLAG FOR TEST AND SETTING
*
DLET03   LTR   R8,R8                   TEST FOR DELETED OR IN USE
         BNZ   DLETERRX                IF EITHER, EXIT WITH ERROR
         LR    R7,R8                   CURRENT COUNT AND FLAG VALUE
         OR    R7,R6                   SET DELETE FLAG ON IN NEW
         CS    R8,R7,ELEMCTFL          ATOMIC CHANGE TO DELETE FLG
         BNZ   DLET03                  LOOP IF OLD VALUE CHANGED
*
* ELEMENT MARKED DELETED - CHAIN FOR RELEASE PROCESS BY SOME PROCESS
*
         L     R8,DELTANCH             CURRENT VALUE OF PENDING DELETES
DLET04   ST    R8,ELEMACHN             LIFO QUEUE ON ALTERNATE CHAIN
         CS    R8,R9,DELTANCH          ADD TO ANCHOR
         BNZ   DLET04                  LOOP TILL SUCCESS
*
*   ELEMENT HAS BEEN DELETED (NOT FREED) - RETURN SUCCESS
         CALL  RELEASE                 RESET SHARED LATCH
         LM    R14,R12,12(13)          RESTORE REGISTERS
         LA    R15,SUCCESS             SET SUCCESS RETURN CODE
         BR    R14                     RETURN
*
*      ELEMENT TO DELETE NOT FOUND ON CHAIN - RETURN NO SUCCESS
DLETERRX CALL  RELEASE                 RESET SHARED LATCH
         LM    R14,R12,12(13)          RESTORE REGISTERS
         LA    R15,NOSUCCES            SET ERROR RETURN CODE
         BR    R14                     RETURN
         DROP  R9                      RELEASE BASE REGISTER
```

Figure 21: Sample Implementation of DELETE for List Manipulation

```
* COPYRIGHT IBM CORPORATION 1986
*
* RELEASE SHARED LATCH, AND FREE PENDING DELETES IF LAST USER.
* REGISTERS ARE NOT SAVED -
* ACTUAL FREEING OF REMOVED ELEMENTS IS NOT SHOWN.
*
RELEASE  SLR   R4,R4                   ZERO  REMAINING WORK FIC
         SLR   R5,R5                   ZERO  REMAINING WORK LIC
*
* OUTER RELEASE LOOP - DETERMINES WHETHER TO WORK OR EXIT
*
RLSE00   LM    R0,R1,SHRLATCH          COUNT AND CHAIN ANCHOR
RLSE01L  LR    R2,R0                   REPLICATE READER COUNT
         LTR   R3,R5                   REMAINING WORK LIC
         BZ    RLSE01A                 IF REMAINING WORK THEN
         ST    R1,ELEMACHN-ELEMENT(,R4) ADD ANY NEW TO FRONT (LIFO)
         B     RLSE01B
RLSE01A  LTR   R3,R1                   ELSE REPLICATE NEW WORK LIC
         BZ    RLSE01C                 IF NEW WORK OR REMAINING WORK THEN
RLSE01B  BCT   R2,RLSE01D              DECREMENT - BRANCH IF NOT TO 0
         LR    R3,R2                   WAS ZERO - ZERO NEW ANCHOR
         LR    R2,R0                   RESTORE READER COUNT TO 1
         B     RLSE01D                 TO CDS
RLSE01C  BCTR  R2,0                    ELSE RELEASE SHARED LATCH
RLSE01D  CDS   R0,R2,SHRLATCH          ATTEMPT THE CHANGE
         BNZ   RLSE01L                 LOOP IF NOT SUCCESS
*
* IF LATCH COUNT WAS DECREMENTED, RELEASE IS COMPLETE.
* OTHERWISE, THERE IS SOME WORK REMAINING
*
         CR    R0,R2                   DID I CHANGE COUNT?
         BNZR  R14                     IF YES, THEN EXIT - DONE.
*
*  CHAIN TO PROCESS EITHER IN R4 OR IN R1 IF R4 IS ZERO
*
         LTR   R9,R5                   CHECK FOR RESIDUAL CHAIN
         BNZ   RLSE02                  IF RESIDUAL IS ZERO THEN
         LR    R9,R1                     MOVE NEW CHAIN ANCHOR
RLSE02   SLR   R4,R4                   ZERO RESIDUAL LIC
         LR    R5,R4                   AND RESIDUAL FIC

         USING ELEMENT,R9              SET AS BASE
```

Figure 22: Sample Implementation of RELEASE for List Manipulation

Part-1 Outer Loop Control

```
* COPYRIGHT IBM CORPORATION 1986
RLSE03   TM    ELEMUSFL,OFFCHAIN     IF ELEMENT OFF PRIMARY CHAIN
         BNZ   RLSE04                 THEN TO FREE CHAIN, ELSE
*
*  ELEMENT TO BE UNCHAINED FROM PRIMARY CHAIN
*
         L     R8,ELEMPCHN           GET  -> NEXT ELEMENT
*
         LR    R7,R9                 MAKE CURR COMPARE VALUE FOR CS
         CS    R7,R8,LISTANCH        ATTEMPT SWAP UPDATE
         BZ    RLSE03T               IF SUCCESS EXIT, ELSE
         B     RLSE03E               BRANCH TO WHILE LOOP TEST
*
RLSE03L  L     R7,ELEMPCHN-ELEMENT(,R7) TO NEXT LIST ELEMENT
RLSE03E  C     R9,ELEMPCHN-ELEMENT(,R7) POINT TO ELEMENT TO DELETE?
         BNE   RLSE03L                  NO- LOOP
         ST    R8,ELEMPCHN-ELEMENT(,R7) CHAIN PREV AROUND DELETED ONE
*
RLSE03T  L     R8,ELEMCTFL           CURRENT VALUE OF COUNT AND FLAGS
RLSE03U  LA    R7,OFFCHAIN           FLAG - OFF PRIMARY CHAIN
         OR    R7,R8                 SET FLAG ON IN PROPOSED VALUE
         CS    R8,R7,ELEMCTFL        ATTEMPT UPDATE
         BNZ   RLSE03U               LOOP TILL SUCCESS
*
         L     R8,ELEMACHN           SAVE NEXT IN CHAIN
         LTR   R4,R4                 CHECK FOR FIRST TIME
         BNZ   RLSE03X               IF FIRST (ZERO) THEN
         LR    R4,R9                   SAVE FIC RESIDUAL
RLSE03X  ST    R5,ELEMACHN           ADD POINTER TO NEW ELEMENT
         LR    R5,R9                 MAKE IT NEW LAST
         B     RLSE05                TO COMMON
*
*  ELEMENT OFF PRIMARY CHAIN - MAY NOW BE FREED
*
RLSE04   L     R8,ELEMACHN           SAVE NEXT IN CHAIN
*
*  FREEING PROCESS OF R9->ELEMENT NOT SHOWN, BUT DONE HERE
*   WITHOUT DISTURBING R8, R4, R5, OR R14 CURRENT CONTENTS
*
RLSE05   LTR   R9,R8                 POSSIBLE NEXT TO PROCESS
         BNZ   RLSE03                IF NOT ZERO, INNER LOOP
         B     RLSE00                ELSE TO OUTER LOOP
         DROP  R9                    RELEASE ELEMENT BASE
```

Figure 23: Sample Implementation of RELEASE for List Manipulation

Part-2 Processing of Queued Work for Obligated Process.

REFERENCES

[princop] IBM System/370 Extended Architecture, Principles of Operation, IBM Pub. No. SA22-7085 (1983).

[mvslock] MVS/Extended Architecture System Programming Library: System Macros and Facilities Volume 1 and Volume 2, IBM Pub. No. GC28-1150 and GC28-1151 (1983).

[rjhash] R. L. Obermarck, R. K. Treiber, Practical Uses of Hashing For Main Storage Searching, IBM Research Report No. RJ3483 (1982).

[convoy] Mike Blasgen, Jim Gray, Mike Mitoma, Tom Price, The Convoy Phenomenon, ACM Operating Systems Review Vol 13 No 2 (April, 1979).

[introos] A. N. Habermann, Introduction To Operating System Design, Science Research Associates (1976).

[deadlock] J. W. Havender, Avoiding Deadlock in Multitasking Systems, IBM Systems Journal **7 No. 2** (1968) pp. 74-84.