Cheap Mutual Exclusion

William Moran, Jr. – Swiss Bank Corp Investment Banking[1]
Farnam Jahanian – IBM T. J. Watson Research Center

ABSTRACT

A new method of enforcing mutual exclusion among concurrent processes on uni-processors running *UNIX* is presented in this paper. When a process attempts to obtain a lock, no race condition will occur unless the process is preempted. The central idea is that a process can avoid a race condition if preemption is made visible to the process when it is rescheduled. Two possible implementations of this idea are discussed in depth. The proposed solutions do not require special hardware support or disabling of interrupts during a critical section.

1. Introduction

This paper presents an alternative approach to mutual exclusion on a uni-processor *UNIX* system. The motivation for this work arises from the expense frequently associated with achieving mutual exclusion among concurrent processes. On many *UNIX* systems, such protection requires several system calls per lock access; the use of semaphores[2] to protect shared resources is frequently impractical due to the expense. A common use of shared memory is for storage of data structures accessed by several processes; the contention for these data structures is infrequent enough that costly locking schemes are wasteful. On many systems, the use of semaphores requires a system call to lock the resource; another system call is required to unlock the resource. Even when there is no contention for the lock, both system calls are executed. These system calls may be very expensive depending on how much support the hardware provides, but even in the best case, system calls are expensive due to the context switches and the change from user mode to kernel mode. When there is contention among processes in accessing shared resources, it is often the case that the actual code for manipulating shared data structures is only a few lines. Hence, the expense associated with obtaining a lock is disproportionate to the actual use of the shared resource. The overhead of obtaining and releasing a lock is particularly important if a process accesses locks multiple times.

Some hardware provides support for a test and set or any of a number of similar sorts of constructs; on architectures which support such atomic operations, the Intel i486 for example, locking is trivial. However, not all architectures provide support for this sort of atomic operation; For example, the IBM RS/6000 processor intentionally does not provide any such instruction. However, there is a special operation called CS. This implements the compare and swap instruction as a pseudo system call; while this avoids the overhead of a system call, it requires both special hardware support and kernel code. So, while this is a compromise between having a special atomic hardware instruction (the hardware support required for the CS instruction is somewhat more general in nature, i.e., it is useful for other things) and having no support, it is not possible to use this technique on most other machines [7,6,8].

This paper presents a set of practical solutions to this problem. The solutions presented here are general enough that they could be used on many machines running *UNIX* ; in particular, it will work on any machine with POSIX signals or their ancestor, BSD signals. The proposed solutions do not require special hardware support or disabling of the interrupts while a lock is being acquired; the solutions presented here do require that the Operating System kernel be modified. The characteristics desired of such solutions are that they should be very inexpensive in the ordinary case (no contention), and they should be no more expensive than semaphores in the presence of contention. Ideally, the solution should require no more user code than the standard semaphore example:[3]

```
int sem_id;
...
key2 = ftok( FTOK2_FILE, FTOK2_ID );
sem_id = semget( key2, 1,
                 IPC_CREAT | 0777 );
semctl( sem_id, SEM, SETVAL, 1 );
...
p(sem_id);
/* critical section */
v(sem_id);
```

Code that uses semaphores typically requires 3 system calls to initialize the semaphore, and each

[1]William Moran was with IBM T. J. Watson Research Center when this work was performed.

[2]Semaphores means System V style semaphores in this context.

[3]This uses the Sun implementation of standard System V UNIX IPC operations.

acquisition and subsequent freeing of the semaphore requires 2 system calls. Since system calls typically have path lengths of at least 3000 instructions, system calls are considered expensive. In particular, if a program makes many lock accesses, the cost associated with getting the locks becomes extremely important.

The remainder of this paper is organized as follows: The next section discusses an overview of the proposed approach for achieving mutual exclusion in *UNIX* systems. Section 3 and Section 4 present two different implementations of this approach. Section 5 is a discussion of a few of the subtler implementation issues. The last section contains concluding remarks.

2. Approach

Mutual exclusion is needed to avoid race conditions associated with the modification of shared data by concurrent processes [4,5,2,3,11,14] The problem of avoiding a race condition can be formulated in an abstract way as imposing a shared lock (e.g., shared variable) on access of a critical section. In a uni-processor *UNIX* system, obtaining a lock may be non-atomic if the process attempting to obtain the lock is preempted while it is obtaining the lock. For example, suppose a process obtains a lock by writing its process id into a shared variable, called `shared_lock`. This variable is initially set to zero to denote that the lock is free.

```
if (shared_lock == 0)
  shared_lock = my_pid;
else ...
```

If the process is preempted after testing `shared_lock` but before writing its process id into the shared variable, another process may obtain the lock by executing the same sequence of instructions (without preemption). When the first process is rescheduled at the point of preemption, it will acquire the lock while it is being held by the second process.

In a uni-processor system, if the sequence of instructions to obtain a lock is executed without preemption by the kernel, no race condition can occur. This simple observation is very important: if a process obtains a lock by writing its process id in the lock, then if it is not preempted while so doing, the lock can be held by only one process. However, if the process is preempted while obtaining a lock, the race condition may be avoided by informing the process that it was preempted. The central idea here is very simple: make preemption visible to processes that require the service. If a process attempting to lock a shared variable is preempted, it is notified when rescheduled so that another attempt at acquiring the lock is made, avoiding the race condition. The two subsequent sections describe alternative implementations of this concept in a *UNIX* system.

3. First Version

The first solution is intended to show the viability of achieving mutual exclusion by making process preemption visible to the process after it is rescheduled. The proposed solution involves asynchronous notification of a user process when it has been preempted. The signal facility, supported in all UNIX-like systems, is used for asynchronous notification. The process receives a signal when it is rescheduled after the preemption; this signal indicates that a preemption occurred. The key point is that the process returns to the signal handler rather than to the point at which the preemption occurred. In the signal handler, the process can be forced to resume execution at a point where another attempt at acquiring the lock can be made. For example:

```
jmp_buf context;
int ret;
...
void sig_handler(int num)
{
  longjmp(context,0);
}
...
setjmp(context); ← execution resumes here
statement1;
statement2;      ← preemption occurs here
statement3;
statement4;
...
statementn;
```

The above code segment invokes two library procedures: `setjmp` and `longjmp`. The `setjmp` call saves the context of the process at the point of invocation. The process context includes information such as the values of the registers and the PC. The `longjmp` call , when invoked inside the signal handler, will cause the process to resume execution at the point where the setjmp was called. If a signal gets delivered to this code upon preemption, and the signal is caught by `sig_handler`, then execution will resume at the setjmp rather than where the code was preempted.

The preceding paragraph illustrated how normal execution of a process can be altered via the signal facility. This forms that basis for the first solution to making preemption visible to a process. Specifically, the following code implements mutual exclusion if the kernel has been modified to allow the preemption signal; specifically, it is necessary that the preemption signal handler be the first code executed upon return from a preemption[4,5]. The

[4]This code uses the AIX V 3 implementation of POSIX signals and signal handlers.

[5]This and subsequent examples assume that `sleep()` causes a process to be preempted automatically.

following code uses a new system call, `preempt_sig` which takes as an argument, the number of the signal which is to be sent to the process when it returns after being preempted. The subsequent examples in this paper will use signal number 43. A system without the extended signals added by POSIX could simply use one of the signals not normally used anymore. See Figure 1.

This code works as follows: it gets the process id of this process (1). The process id is an integer that uniquely identifies every process. The next two lines (2 & 3) define two signal handlers used for the preemption signal. The first of these will simply ignore the signal; as the result of *UNIX* semantics, ignored signals are never delivered [9,10], so while `a` is the specified handler, preemption will have no extra cost. While the handler bound to `b` is in effect, `sig_handler` will be called upon delivery of the preemption signal. (4) simply causes the preemption signal to be ignored. (5) is a new system call that tells the kernel that this process should have the specified signal, in this case 43, delivered upon preemption. The critical section follows this; the `setjmp` call saves the state of the process at this point. The `longjmp` in the signal handler will cause execution to be resumed here. (7) rebinds the preemption signal to `sig_handler`, so once this statement has been executed, the process, upon being preempted, will resume execution in `sig_handler` rather than at the point of preemption. (8-11) are the standard method of acquiring a shared lock. Finally, (12) turns off the preemption signal. Figure 2 shows what happens upon preemption. The idea here is that if, on a uniprocessor, the statements (8-11) execute atomically, then we are assured that the lock has been acquired by only one process. However, if preemption occurs anywhere in these statements, then it is possible that another process has come in and acquired the lock, so we need to check the lock to see if this has happened. If it has, we force preemption with the `nsleep` call (`nsleep()` does the same thing as usleep except in nano-seconds). There is one important assumption in this code; (11) must not be capable of leaving the variable `shared_lock` in a corrupt state, but on every architecture of which we are aware, preemption

```
#define SIG_NUM 43
jmp_buf context;
int ret;
pid_t ourpid;
struct sigaction a,b;
...
void sig_handler(int num)
{
   longjmp(context,0);
}
...
ourpid = getpid();                                          (1)
a.sa_handler = SIG_IGN;                                     (2)
b.sa_handler = sig_handler;                                 (3)
...
sigaction(SIG_NUM,&a,NULL);                                 (4)
/* The following instructions get the lock */
preempt_sig(SIG_NUM);                                       (5)
setjmp(context);                                            (6)
sigaction(SIG_NUM,&b,NULL);                                 (7)
if ((shared_lock != 0) && (shared_lock != ourpid))          (8)
   nsleep(1);                                               (9)
else                                                        (10)
   shared_lock = ourpid;                                    (11)
/* We have the lock, so we turn off the signal */
sigaction(SIG_NUM,&a,NULL);                                 (12)
/* critical section */
...
/* Free the lock */
shared_lock = 0;
```

Figure 1: Locking

during this statement either results in `shared_lock` being unchanged, or in the assignment actually succeeding. On an architecture where this assignment could result in `shared_lock` changing to something other than `ourpid`, this method of mutual exclusion will not work.

The performance of this code is about 25% better than the comparable code using semaphores when run on an RS/6000; based on the performance of semaphore operations and signal operations, it seems as if this code would be 50% as costly as the equivalent semaphore code on a Sun 4 running SunOS 4.0.3.

Sequence of Events Upon Preemption

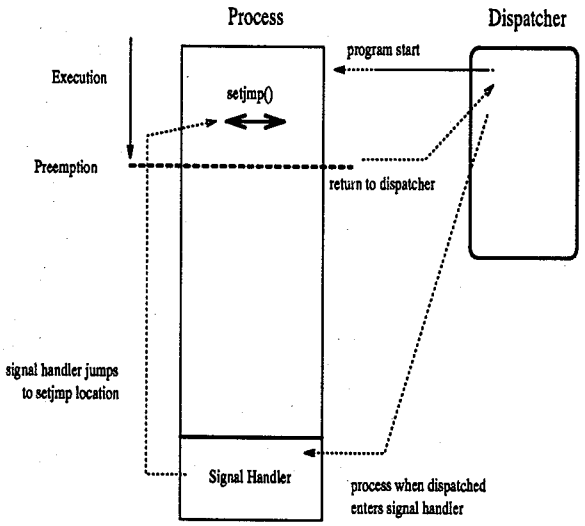

Figure 2: Preemption Events

Significant performance improvement can be obtained when several locks are being acquired and freed in one section of code. Specifically, it is possible to leave the signal delivery turned on and simply check in the signal handler whether it is necessary to perform the `longjmp`. After the first lock access, the cost of obtaining the subsequent locks is reduced to the cost of invoking `setjmp` to save the process context. The cost of releasing a lock is a single assignment. However, there is a small overhead in invoking the signal handler after the process is rescheduled. The code for obtaining and releasing several locks looks something like Figure 3.

The code segment enables the preempt signal before an attempt to acquire any lock is made. The process context is saved and a local variable `should_jump` is set prior to each lock access. If the process is preempted while acquiring the lock, the variable `should_jump` is tested inside the signal handler and the process is forced to make another attempt to obtain the lock. After obtaining the lock, the variable `should_jump` is reset. This is an effective solution since returning from a preemption into a signal handler does not slow the code down excessively. This code should be substantially faster than the equivalent semaphore code; it saves two system calls per critical section, i.e., both `sigaction` calls, but it has not been tested enough to state this definitively.

```
#define SIG_NUM 43
jmp_buf context;
int ret;
pid_t ourpid;
struct sigaction a, b;
int should_jump = 0;
...
void sig_handler(int num)
{
   if (should_jump)
      longjmp(context,0);
   else
      return;
}
...
ourpid = getpid();
a.sa_handler = SIG_IGN;
b.sa_handler = sig_handler;
...
sigaction(SIG_NUM,&a,NULL);
preempt_sig(SIG_NUM);
setjmp(context);
sigaction(SIG_NUM,&b,NULL);
should_jump = 1;
if ((shared_lock != 0) &&
          (shared_lock != ourpid))
   nsleep(1);
else
   shared_lock = ourpid;
should_jump = 0;
...
setjmp(context);
should_jump = 1;
... /* acquire lock */
should_jump = 0;
...
setjmp(context);
should_jump = 1;
... /* acquire lock */
should_jump = 0;
sigaction(SIG_NUM,&a,NULL);
```

Figure 3: Obtaining and releasing locks

This section has presented code that uses the visibility of preemption as a way of ensuring mutual exclusion. The code presented here uses this feature in a rather naive fashion, but it shows the power of this idea. The final example in this section demonstrates one of the most important aspects of this idea: by making locks very cheap to obtain, this code makes it possible to write code that shares many resources. Since serial access to these

resources can be insured cheaply, the cost of sharing resources can be dramatically reduced.

4. Second Version

In the previous section we presented a solution that uses the preempt signal in the obvious way. While acquiring a lock, the process enables the preempt signal so that it is notified of a preemption after being rescheduled. The preempt signal is disabled after the lock is obtained. As illustrated in the preceding section, the cost of enabling/disabling the signal can be amortized among multiple lock accesses. However, a solution which reduces this overhead is desirable. In this section, we present a solution that uses the signal in a slightly more subtle fashion. If we modify the system call

`preempt_sig`, that we presented in the previous section, so that it takes the address of a flag in user space, then the kernel can check this flag to determine whether to send a signal. In this case, protecting the acquisition of the lock will be the cost of setting this variable. For example:

```
#define SIG_NUM 43
jmp_buf context;
int flag = 0;
int ret;
pid_t ourpid;
struct sigaction a,b;
...
void sig_handler(int num)
{
   if (flag)
    longjmp(context,0);
}
...
ourpid = getpid();
b.sa_handler = sig_handler;
...
sigaction(SIG_NUM,&b,NULL);
preempt_sig(SIG_NUM,&flag);      (1)
/* Code to get a lock */
setjmp(context);
flag = 1;                        (2)
if ((shared_lock != 0) &&
            (shared_lock != ourpid))
    nsleep(1);
else
    shared_lock = ourpid;        (3)
/* We have lock;
           turn off the signal */
flag = 0;
/* critical section */
...
shared_lock = 0;  /* Free the lock */
```

This version uses the new system call (1) that takes the address of `flag`, and it simply sets `flag` to 1 when the preempt signal should be delivered. When the signal is not necessary, all that is needed is to set the flag to 0. Compare the critical section here (the code between (2) and (3)) with that above. Two system calls have been eliminated. As a result, we are not using any system calls to achieve the necessary protection. There are only two costs here; the first is the `setjmp` and the `longjmp` if preemption occurs, but this is very small. The other cost is slightly harder to quantify, since it is incurred in the kernel. Understanding this cost requires that the code in the kernel be understood. These costs compare favorably to the semaphore code. The cost incurred when a process blocks on a semaphore is fairly high.

Kernel code is required in two places, the first is the system call `preempt_sig`. This code needs to set a flag associated with the process that indicates that the process should be signalled when preempted; this code also needs to store the number of the signal to send. These two quantities require 7 bits (1 for the flag and 6 for the signal number); the pointer to the flag requires 32 bits, so 39 bits are associated with the process. Finally, because the flag, which resides in the users address space, is going to be checked in the kernel, the page containing this flag must be pinned into real memory; since this flag will be checked in the dispatcher, and the code in this part of the kernel is not allowed to page fault, the memory access to check this flag must not page fault. This page is pinned with the `pinu` kernel service. Unless the system is swapping (as opposed to paging), this appears to have minimal impact. Thus, the process will have one page that is always in memory.

The second place where kernel code is required is in the dispatcher; after the dispatcher has determined which process will be run next, it is necessary to check the flag that indicates whether the process should be signalled. The code to do this is something like:

```
if (p->presig)
   if (*(p->sigflagaddr))
      pidsig(p->pid,p->sigtosend);
```

where the first `if` checks to see whether this process should ever receive the preempt signal; the second `if` checks whether the process should have the signal delivered at present (`sigflagaddr` is a pointer to user address space), and the `pidsig` call sends the appropriate signal to the process.

This approach is clearly superior to the first approach presented. It saves two system calls per lock, and the code the user needs to understand is considerably simpler. This code compares very favorably with that required in the semaphore case; it is much faster (obtaining the lock is very inexpensive), and the number of system calls required for the setup is the same (3 in each case). These system calls are also slightly cheaper than those used for semaphores. The mechanism required by this approach has been implemented and tested on an RS/6000 running AIX 3.1, and the added cost in the

kernel is not measurable. This code also has the property that the complexity (as the user sees it), is no worse than for semaphores. This solution attains all the qualities that were originally hoped for. The complexity of use is slightly better than for semaphores. The performance when there is no contention for the lock is substantially better than equivalent code that uses semaphores. Finally, the performance is no worse when the lock is already held by another process; in both cases, the process blocks. The only extra cost for this code occurs when the process is preempted while obtaining the lock; this extra cost is dwarfed by the reduced cost in the ordinary case. Due to these advantages, this method of achieving mutual exclusion seems preferable to the use of semaphores.

5. Implementation Notes

The astute reader will have noticed several potential problems with the solutions presented above; this section attempts to address these. In particular, the following might be seen as potential problems: nesting of signal handlers, starvation as a result of repeated preemption, and hidden assumptions as to atomicity of operations. This section will show why each of these fails to be a problem, and a brief explanation of each is included.

UNIX makes few guarantees about what can safely be executed within a signal handler. In particular, in the case of nested signal handlers, it is the case that almost anything executed can cause problems. However, *POSIX* 1003.1 [9] makes the guarantee that `longjmp` can be executed from *a non-nested* signal handler. Since it is the case, that the signal handlers here will not be nested, the `longjmp()` should be safe. It will also be noted that this code assumes that `nsleep(1)` will force preemption. It is certain that `nsleep` with some value will force preemption, and an incremental backoff could be implemented as `nsleep(i*=2)` where `i` had some suitable initial value.

The above code contains the possibilty of starvation if the process has few enough pages. If, upon return from the signal handler, the process page faults, then the process will immediately return to the signal handler. As a result, it might be necessary that the process have two pages available to it; one might be used to hold the signal handler code, and the other would hold the code where the `setjmp` was executed. If the code to obtain the lock spans multiple pages, a similar problem may arise; it should be noted that this is only a problem if the process can obtain only one page. However, a process that cannot acquire two pages is likely to have many other problems as well, so this seems like a fairly minor drawback.

An interesting example arises when a process is preempted twice while trying to get the same lock. In particular, a problem might be thought to arise if a process is preempted in the signal handler for the preemption signal; the potential problem occurs since the second instance of the signal might not be delivered until the next trip through the kernel. Consider a process in the preempt signal handler that is preempted. When the process resumes, the new instance of the signal will not be delivered since the signal is currently being handled; the second instance of the signal will be pending. The second instance of the signal will not be received until the next time the kernel gets control. In the first example, since the instruction executed after the `setjmp` is a system call eg `sigaction`, the signal will be delivered immediately, the process will jump back to the same place, and it will continue. In the second example, since no system call is executed, the signal will be delivered some arbitrary time later. However, the `longjmp` will only occur if the flag is still set, and the correct semantics are preserved because the lock cannot be corrupted. Further, since the order of operations is: `setjmp`, flag = 1, even if the signal gets delivered at the time of the next lock use, the correct context would be used. Of course, this sequence of events is the least favorable, but it is handled correctly.

Finally, as noted above, the assumption is made that an assignment of a wordsize quantity to a word size location cannot be preempted in such a way as to leave the location with a value other than either its initial value or the value being assigned. Simply put, if the statement `a = 7` is executed with `a` having an initial value of zero, `a` cannot have a value other than zero or seven no matter what the operating system may do (other than crash the machine possibly). The reason for this restriction should be obvious, and we know of no machines that do not obey this restriction. A failure to provide this facility would make the solutions presented here invalid; however, a machine that acted this way would need to have a multi instruction assignment, and this seems untenable.

6. Conclusion

The cost of obtaining and releasing a lock is particularly important when concurrent processes require mutual exclusion for accessing shared data structures stored in memory. In such cases, the duration of a critical section is often very short, and expensive methods for acquiring locks impose unnecessary overhead on those processes which manipulate the shared data structures. The characteristics desired of a method of achieving mutual exclusion on a *UNIX* system are that it should be simple for a program to use, it should be extremely inexpensive to use when there is no contention, and it should be as cheap as possible when contention exists. The two solutions presented here can be made as simple to use as semaphores, so they meet the first criteria. When there is no contention for the

lock, and the process does not get preempted, the second method presented is essentially free. Even in the presence of contention, the second method presented is extremely inexpensive. Since much of the cost of using a semaphore is due to the overhead of making a system call, the proposed solution is considerably less expensive than semaphores on machines without hardware support. Furthermore, it is not too much more expensive, in the worst case, than semaphores even with hardware support. Combining ease of use and inexpensive operation as it does, this method of achieving mutual exclusion should make shared resources practical in many cases where they have been considered too expensive.

7. Bibliography

[1] Maurice J. Bach.*The Design of the UNIX Operating System*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[2] P.Brinch Hansen. *Operating System Priciples*. Prentice-Hall, Englewood Cliffs, N.J., 1973

[3] P.Brinch Hansen. The programming language concurrent pascal. *IEEE Transactions on SE*, SE-1(2):199-207, June 1975.

[4] E. W. Dijkstra. Co-operating sequential processes. In F.Genuys, editor, *Programming Languages*. Academic Press, London, 1968.

[5] C. A. R. Hoare. Monitors: An operating system concept. *Communications of the ACM*, pages 549-557, October 1974.

[6] IBM Corp. *AIX Version 3 for the RS/6000 Calls and Subroutines Reference: Base Operating System Volume 2*, first edition, March 1990.

[7] IBM Corp. *AIX Version 3 for the RS/6000 Calls and Subroutines Reference: Base Operating System Volume 1*, first edition, March 1990.

[8] IBM Corp. *AIX Version 3 for the RS/6000 Calls and Subroutines Reference: Kernel Volume 5*, first edition, March 1990.

[9] IEEE, New York, New York. *POSIX 1003.1*, 1989.

[10] Samuel Leffler, Marshall Kirk McKusick, Michael Karels, and John Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison Wesley, Reading, Massachusets, 1989.

[11] G. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, pages 115-116, June 1981.

[12] Marc J. Rochkind. *Advanced UNIX Programming*. Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

[13] SUN Microsystems Inc. *SUN OS 4 System Services Overview*, revision a of 9 may 1988 edition.

[14] Andrew S. Tanenbaum. *Operating Systems Design and Implementation*. Prentice-Hall, Englewood Cliffs, New Jersey, 1987.

Author Information

William Moran has BS and MS degrees in CS from Union College. He joined IBM Research in 1988 after having been a PhD student at Yale. At IBM Research he worked on real-time and distributed programming languages, real-time systems, distributed and fault-tolerant systems, and a juggling robot. He left IBM Research in 1992 to join a new proprietary trading group at Swiss Bank Corp. Investment Banking as a Senior Computer Scientist. Reach him via US Mail at SBCI; 4th Floor; 222 Broadway; NY, NY 10038. Reach him electronically at wlm@panix.com.

Farnam Jahanian received a Ph.D. in Computer Science from the University of Texas at Austin in 1989. He is currently a Research Staff Member at the IBM T.J. Watson Research Center. His reasearch interests include specification and analysis of real-time systems and fault-tolerant computing. His address is: IBM T.J. Watson Research Center; P.O. Box 704; Yorktown Heights, NY 10598. Contact him by e-mail at farnam@WATSON.IBM.COM.