6.826 Principles of Computer Systems **Fall Semester, 1992**

Lecture 8: October 7, 1992

Lecturer: Bill Weihl *Scribe: Tony Eng*

1 Administrivia

Handout	Title
23	Proving Concurrent Modules Correct
24	Solutions to Problem Set #2
25	"An Introduction to Programming with Threads" by Andrew Birrell

2 Lecture Overview

Today's lecture is the second of a series on concurrent modules and systems. The following topics are discussed:

1. Conditional Variables
2. Implementation of Mutexes
3. Implementaiton of Conditional Variables
4. Proving Correctness of Concurrent Modules

Handouts 19 and 23 are referenced and form the basis for much of the material presented in this lecture.

3 Conditional Variables

Conditional variables, descendents of Hoare's monitors, can be used as a means of reducing busy-waiting. They are a more sophisticated mechanism for synchronization than mutexes and they allow threads waiting for a certain condition to become true to block until awakened.

Let us examine the specification of condition variables in Handout 19 (see Figure 1). `M` is a mutex which the wait procedure uses to make sure that the addition of a thread to the waiting list happens atomically.

It is assumed that the caller of `Wait` has already acquired the mutex. The first guard shows the consequence of calling `Wait` without first acquiring the mutex: havoc results. Otherwise, the procedure performs the following sequential atomic operations:

```
PROC Wait () =
  Mutex.m # SELF => HAVOC                                          1
 [*]                                                               2
  << c := c ++ SELF; Mutex.m := nil >>;                            3
  << (Mutex.m = nil) /\ ~(SELF IN c) => Mutex.m := SELF >>         4
```

Figure 1: Specification of `Condition.Wait`

1. Add the calling thread to the waiting list and release the mutex (line 3).

2. Waits on the next atomic statement until the guard is true; namely until the mutex is free and this thread has been removed from the waiting list (line 4). At this point, the mutex is acquired and the calling procedure can continue.

The two atomic steps in `Wait` look rather strange because we release the mutex in one step and then test to see if it is released in the next. The important thing to note here is that we are *depending* on the intervention of another process during the semicolon between the two atomic statements. This intervention is simply the removal of this thread from the waiting list. This removal is accomplished by one of the two remaining procedures in this module. The first, `Signal`, picks some subset of waiters to wake up. The second, `Broadcast`, has the effect of removing all the waiting threads. The reason `Signal` picks some subset of waiters to resume rather than just a single waiter (this is reflected in the nondeterminism in the spec for Signal) is to allow for a more efficient implementation of `Signal`.

With the ability to signal and broadcast, the user has some measure of control over thread scheduling; however, several race conditions can occur during the scheduling and descheduling of threads, so the specification must be carefully constructed to avoid them. One such race condition is the *wakeup-waiting race*. This occurs when a thread has decided to block, but before it can add itself to the wait queue, another thread makes a signal which goes unnoticed because the queue is empty. Having missed the signal, when the blocking thread finally gets queued, it will need to wait for an arbitrarily long time until some other thread acquires and releases the mutex so that it can be awakened. This problem is avoided by rechecking and verifying the blocking condition after initiating the blocking sequence, but before actually adding the thread to the queue.

The general style for using condition variables is shown in Figure 2. With Hoare's monitors, the condition is guaranteed to be true after a process is awakened; however, with condition variables, it is important to recheck the condition after being signalled to account for the fact that some other process may have acquired and released the mutex in the interim. If the condition is not rechecked, the following sequence of events can occur (Figure 3).

Having acquired the mutex, thread 1 incorrectly thinks the condition is satisfied when in fact it has been invalidated during the interim between awakening and acquisition of the mutex by thread 1.

```
Mutex.Acquire(m)
DO ~condition => Wait(m, c) OD
      :
      :  (do work)
      :
Mutex.Release(m)
```

Figure 2: Style in use of Condition Variables

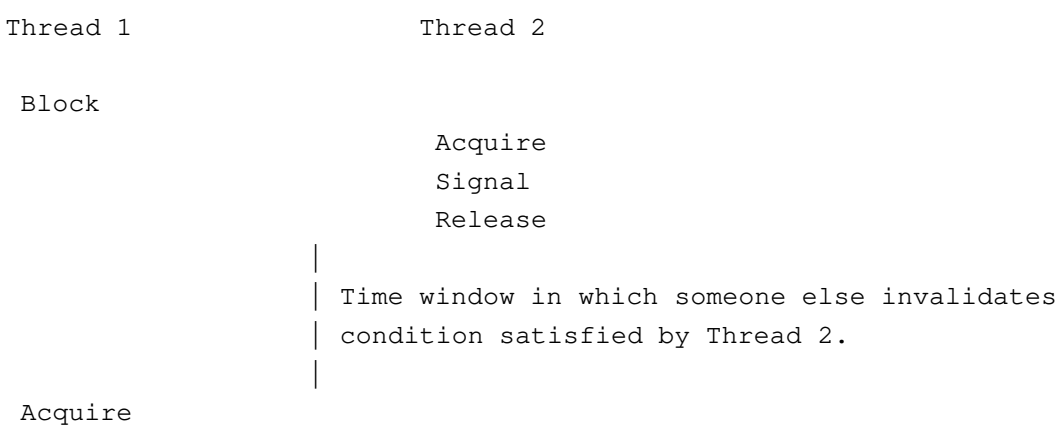

Figure 3: Illustrating the Need to Recheck the Condition in the Do Loop

4 To Spin or To Block

On a uniprocessor, a process that spins while waiting for a mutex to become available wastes cycles. On a multiprocessor however, if there are many processes contending for a mutex and if the mutex is held for long periods of time, then spinning is also not be desireable as the process may need to wait a long time before the mutex is obtained. Hence, it is often preferable to *block* rather than spin. Blocking is used to stop an otherwise spinning process from executing until there is a chance that the condition it is waiting for becomes true. This prevents the spinning process from stealing processor time while it waits.

Blocking, however, is an expensive process in most systems as it requires among other things, entering the kernel, scheduler interaction, and queue manipulation. If t_b is the amount of time needed for a process to block, a process is generally allowed to spin if the expected mutex wait time is shorter than t_b. However, in practice, the expected wait time is often unknown. A hybrid scheme allows a process to spin a duration of t_b before blocking. This 2-phase scheme is no more than a constant factor worse than the best possible scheme, and so, is a competitive algorithm.

5 Application: Read/Write Locks

Given mutexes and condition variables as building blocks, we can easily implement read/write locks (also known as shared/exclusive locks) for example. If we consider the set of threads holding the lock in read mode and the set of threads holding the lock in write mode, then an invariant is that if there is a thread in the write set, then there can be no other thread in either the write or the read set.

Consider the specification of read/write locks on page 5 of handout #19. Note that the integer rw represents the number of readers or writers currently holding the lock and not the actual identities of the holders. Also observe that in procedure **EndRd()**, when a thread holds a read lock, any waiting thread must be waiting for a *write* lock, therefore signal and not broadcast is invoked when the last read lock is released.

6 An Implementation of a Mutex

Refer to Handout 19, page 6. There are several things to note about the implementation presented here.

- The rep has a lock bit that is 'false' when the mutex is unheld and 'true' when available.
- TestAndSet() is an atomic operation that grabs a value, sets it to 'true', and returns the *old* value.
- A global lock exists for SpinLock(), ReleaseSpinLock(), and all other procedures that use the kernel. This lock allows only one procedure to execute at a time to protect critical sections involving the kernel and kernel data structures.

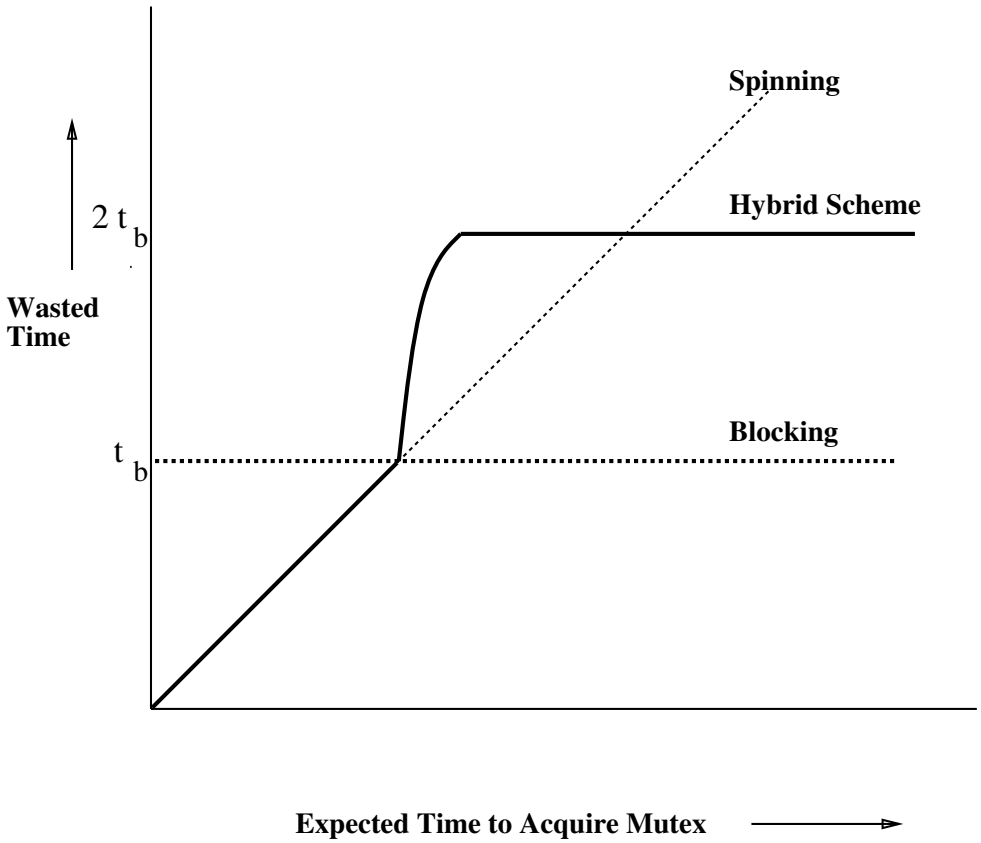

Figure 4: Performance of Hybrid Scheme Compared to Spinning and Blocking.

- Deschedule() is a kernel operation that moves a thread from the ready list onto the kernel's waiting list so that the kernel does not try to schedule it. Deschedule() also releases the global lock.

This mutex implementation maintains a queue of blocked threads inside the kernel. Although entering the kernel is expensive, in general critical sections of code are short and not executed very often, so a process requesting a mutex will find it available with high probability. In other words, the probability of a process blocking and consequently involving the kernel is low.

Nevertheless, we still wish to avoid the kernel whenever possible. Some "fast paths" in the implementation attempt to accomplish this. The kernel is avoided when a thread attempts to acquire a mutex and finds it free. If Release() finds the queue empty, there is no one waiting for the mutex so there is no need to inform the kernel.

7 An Implementation of a Conditional Variable

Refer to Handout 19, page 8. There are several things to note about the implementation of conditional variables presented in Handout 19. Notice that unlike the mutex implementation this implementation always involves the kernel; **Wait(), Signal(),** and **Broadcast** all go into the kernel. A plausible implementation of `Wait` follows:

```
% runs in the kernal
Proc Wait () =
    SpinLock ();
    queue := queue ++ SELF;
    Mutex.Release ();
    Deschedule (SELF);
    Mutex.Acquire ()
```

While this is correct, it requires that the kernel be entered and the spinlock obtained before the mutex can be released. The implementation in the handout avoids this by using an *event counter*. Introduced by Reed and Kanodia, event counter is used as an indicator of when certain events occur. Bascially, the value of the counter is noted at some time x. At some later time y, if the counter displays a larger value, then the event in question has occurred (perhaps more than once) during that time interval between x and y.

8 Concurrent System Transitions

In the sequential case, some external module will call a procedure that will execute sequentially, return a value and complete the state transition. Whether these procedures are "atomic" does not matter in the sequential case since once entered, a procedure must run until it relinquishes control (i.e., until completion).

The sequential case is equivalent to having atomicity brackets around the entire body of every procedure, while the concurrent case allows a finer grain of control.

In the concurrent case the possible state transitions of the state machine are all of the atomic transitions that the module can make rather than the transitions comprising entire procedure (viewed as a single transition).

There are two types of transitions to be considered:

external transitions include the invocations and returns(responses) of a module's external operations used by the client. An external transition in the implementation must correspond directly to a similar transition in the specification.

internal transitions include atomic actions within procedure bodies themselves. The abstraction function maps an internal transition in the implementation to a sequence, possibly empty, of internal transitions in the specification.

9 Abstraction Functions and Rep Invariants for Concurrent Systems

As in the sequential case, we still use *Abstraction Functions* and *Representation Invariants* to prove correctness, but it is more complicated and subtle. The *Rep Invariant* is the same as before– you prove that it holds for the initial state and that every transition preserves it, i.e., if the *rep invariant* is true in the state in which the transition starts and the state in which the transition ends, then it holds for every *reachable* state by induction.

The *Abstraction Function* is more complicated. In the sequential setting we must show that if the implementation can take us through a transition (i.e., an invocation can occur and take us through some initial state, some final state, and some result), then the specification of of the operation allows that same transition from the abstraction of the initial state to the abstraction of the final state. So, whenever we have a transition tr (that takes us from state r to r') in the implementation, we must have a corresponding transition in the spec. That shows that every trace of the implementation is a trace of the spec.

In the sequential case the transitions are operations and are the same for the spec and the implementation. In the concurrent case, the transitions in the implementation and the spec may be different. This is because a transition in the implementation may have no affect on the abstract state at all, or it may atomically perform a sequence of transitions allowed by the spec. Therefore, we use the abstraction function to map it to a sequence of transitions in the specification.

It will, however, always be the case that the *external* transitions correspond one–to–one. In other words, if the there is an external transition in the implementation, then the sequence should contain only one external transition; but for an internal transition, the only constraint on the sequence is that it cannot have any external transitions. In the latter case, the sequence may be empty, which means that the abstract state remained unchanged.

Further, in the case of concurrent programs, the state is more complicated. It includes the global variables, and the local state of each thread, which in turn includes the thread's local variables and its program counter.