6.826 Principles of Computer Systems	Fall Semester, 1993

Lecture 8: October 6, 1993

Lecturer: Bill Weihl *Scribe: Carlo Maley & Pawan Sinha*[1]

1 Administrivia

Handout	Title
21	Revised version of earlier handout
24	Proving Concurrent Modules Correct
25	Lamport's paper — "A Simple Approach to Specifying Concurrent Systems"

2 Lecture Overview

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

1. Condition Variables
2. Implementation of Mutexes
3. Implementation of Condition Variables
4. Proving Correctness of Concurrent Modules
5. Concurrent System Transitions (*additional*)
6. AF and Rep. Invariant for Concurrent Systems (*additional*)

3 Condition Variables

Conditional variables, descendants of 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.

For each process that needs to wait in a given module, there is some condition that it needs to wait for (e.g. the printer needs to wait for the buffer to be not empty and the formatter needs to wait for the buffer to be not full). The three procedures (actually four when the "create" routine is included) that are necessary to support condition variables are:

[1] Acknowledgments: The notes of last year's scribe, Tony Eng, from Lecture 8, 1992, were extremely useful, both as an explanation and, in some cases, for elaboration of the material covered in the 1993 lecture. Parts of this document are almost his words verbatim.

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

Figure 1: Specification of `Condition.Wait`

- Create
- Wait
- Signal
- Broadcast

Condition variables are used in conjunction with Mutex locks. The Mutex must be used to assure integrity of the data being enqueued, dequeued, or tested. So the two simple scenarios are:

Acquire Mutex	Acquire Mutex
Perform Test	Perform Test
If true - perform work	If not true - release Mutex and Wait
Release Mutex and Return	Release Mutex and Return

The Mutex must be released atomically with the "Wait" so that between the Mutex release and the Wait another thread does not perform an operation and a signal on the condition variable that we are about to Wait on. We term the decisive step or transition as the point in the body of the implementation when we abstractly think of the work of the operation as happening or being "committed". There may actually be several choices as to which step is chosen as the decisive step.

A specification of condition variables is provided in Section 4 of Handout 20 where `M` is a mutex.

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:

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

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

Figure 2: Style in use of Condition Variables

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`. It is to be noted that the correctness of a program is unaffected if each instance of "`Signal`" is replaced by "`Broadcast`"; its efficiency, however, might take a plunge.

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 de-scheduling 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 re-checking 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 condition variables, it is important to re-check 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 re-checked, 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.

3.1 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 implementation of read/write locks on page 5 of Handout 20. 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 1                   Thread 2

 Block
                            Acquire
                            Signal
                            Release
                     |
                     | Time window in which someone else invalidates
                     | condition satisfied by Thread 2.
                     |
 Acquire
```

Figure 3: Illustrating the Need to Re-check the Condition in the Do Loop

thread must be waiting for a `write` lock, therefore signal and not broadcast is invoked when the last read lock is released. The use of condition variables permits multiple reads (thus allowing for higher concurrency which a simple mutex would have been unable to provide).

4 An Implementation of a Mutex

Refer to Handout 20, 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.
- `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.

5 The Implementation of Condition Variables

See Handout 20, page 8, for a Spec program to implement condition variables.

The `eventCount` is a monotonically increasing variable that can be incremented atomically and read atomically. Without the `eventCount`, releasing the mutex in the procedure `Wait` could allow another thread to signal before the `KernelBlock` procedure is invoked. This would mean that the thread that is about to wait on the condition would miss the signal. However, before that thread puts itself to sleep, it checks on the `eventCount` to see if anything happened in the interim between releasing the mutex and this point in `KernelBlock`. If the `eventCount` is different from when it was previously recorded, a signal or broadcast must have occurred, so the thread does not have to block, but can instead act as if it was signalled.

This is the aspect of signalling that might wake up more than one thread. If there is already at least one thread waiting on the condition variable, and another one is just about to put itself to sleep, both the first thread in the condition queue, and the thread that was about to enter the queue, will be signalled. It is possible to refine the implementation of the condition variable to ensure that only one thread is awaken by a signal, but it is significantly more complicated. Specifically, it is difficult to handle interrupts.

Generally, conditional waiting happens in the context of:

```
DO P => condition.wait() OD
```

so that a predicate, `P`, is checked before waiting and after waking up. This ensures that there will be no adverse effects of waking up more than one thread with a signal or broadcast.

6 Specifications and the Correctness of Implementations

In the sequential case we had a bunch of operations for changing state. This allowed us to describe the operation of a system as a state machine with a set of transitions.

We model a concurrent system with the same basic conception, a state machine, except that now procedures are no longer atomic transitions of the state machine. Every atomic step is a transition in the state machine. The fact that procedures are made up of a series of small atomic steps means that our description of the state machine for a concurrent system is more fine grained than our description of the sequential state machine.

We also need to distinguish between external and internal transitions. An external transition involves the invocation or return of an external procedure in the module, used by the client.

There are at least three atomic actions for every external procedure:

1. Moving the program counter (pc) from the point of the procedure invocation to the body of that procedure (an external action).

2. Executing the body of that procedure (one or more internal actions).

3. Returning from that procedure (an external action).

An external transition in the implementation corresponds directly with an external transition in the specification. An internal transition is an atomic action within the body of a procedure of the module. It can correspond to a sequence, possibly empty, of internal transitions in the specification (see Section 6.1 below).

Definition: The *behavior* of a module is the set of sequences of external transitions that the module can produce.

Again, the client does not care about what happens inside the module. They only care about the external transitions, the parts of the module that the clients can get their hands on.

Our rule for correctness remains the same: $B_{Impl} \subseteq B_{Spec}$ where B_{Impl} is the behavior of the implementation, and B_{Spec} is the behavior of the specification.

6.1 Proving an Implementation Correct

We still use rep invariants and abstraction functions as our main tools. Proving that a property is a rep invariant remains the same as before. The property must be true of the initial states, and every atomic action must preserve that property. The abstraction function is now more complicated. In sequential systems, there was a one-to-one correspondence between transitions in the implementation and transitions in the specification. However, in concurrent systems, the atomic actions inside the body of the implementation have a generally finer grain than the transitions in the specification. Thus, if we are to map transitions in the implementation to transitions in the specification, it will often be the case that an implementation transition will have no obvious analogue in the specification. The solution to this problem is to map a single implementation transition to a series, α, of transitions in the specification. This series will often be empty.

$$\begin{array}{ccc} & \alpha & \\ AF(s_1) & \rightsquigarrow & AF(s_2) \\ \Uparrow & & \Uparrow \\ s_1 & \rightsquigarrow & s_2 \\ & T & \end{array}$$

In the abstraction function for a concurrent system, we must show that for all states s_1, s_2, and transitions T in the implementation, there is a (possibly empty) sequence of transitions α in the specification such that

1. α takes $AF(s_1)$ to $AF(s_2)$.

2. α is "externally the same" as T.

The second condition means that if T is an internal transition then α is a sequence of purely internal transitions; otherwise (if T is an external transition) α has exactly one (the same) external transition.

Typically α is one of two forms in practice, when T is an internal transition. Either α is empty, when T does not change the state in the specification, or α has one transition. Sometimes you need an arbitrary sequence for α, e.g., when you try to prove the *equivalence* of a specification and its implementation, so that you must map specification $\rightarrow$ implementation.

7 Concurrent System Transitions

[2]

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.

8 Abstraction Functions and Rep Invariants for Concurrent Systems

[3]

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

[2] Additional section from last year's notes
[3] Additional section from last year's notes

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.