CMPSCI 377 Operating Systems **Fall 2008**

Lecture 10: October 2

Lecturer: Prashant Shenoy *Scribe: Shashi Singh*

10.1 Monitor

You might get an experience from studying all semaphore examples that signal and wait calls may scatter everywhere in your program in a not-so-well structured way. If you really get such a feeling, the concept of monitor comes to rescue. A monitor has *four* components: initialization, private data, monitor procedures, and monitor entry queue. The *initialization* component contains the code that is used exactly once when the monitor is created, The *private data* section contains all private data, including private procedures, that can only be used within the monitor. Thus, these private items are not visible from outside of the monitor. The *monitor procedures* are procedures that can be called from outside of the monitor. The *monitor entry queue* contains all threads that called monitor procedures but have not been granted permissions. Therefore, a monitor looks like a class with the initialization, private data and monitor procedures corresponding to constructors, private data and methods of that class. The only major difference is that classes do not have entry queues.

Monitors are supposed to be used in a multithreaded or multiprocess environment in which multiple threads/processes may call the monitor procedures at the same time asking for service. Thus, a monitor guarantees that at any moment at most one thread can be executing in a monitor! When a thread calls a monitor procedure, we can view the called monitor procedure as an extension to the calling thread. If the called monitor procedure is in execution, we will say the calling thread is in the monitor executing the called monitor procedure. Now, if two threads are in the monitor (i.e., they are executing two, possibly the same, monitor procedures), some private data may be modified by both threads at the same time causing race conditions to occur. Therefore, to guarantee the integrality of the private data, a monitor enforces mutual exclusion implicitly. More precisely, if a thread calls a monitor procedure, this thread will be blocked if there is another thread executing in the monitor. Those threads that were not granted the entering permission will be queued to a monitor entry queue outside of the monitor. When the monitor becomes empty (i.e., no thread is executing in it), one of the threads in the entry queue will be released and granted the permission to execute the called monitor procedure. Although we say "entry queue," you should not view it literally. More precisely, when a thread must be released from the entry queue, you should not assume any policy for which thread will be released. In summary, monitors ensure mutual exclusion automatically so that there is no more than one thread can be executing in a monitor at any time. This is a very usably and handy capability.

10.2 Condition Variables

A thread in a monitor may have to block itself because its request may not have completed immediately. This waiting for an event (e.g., I/O completion) to occur is realized by condition variables. A condition variable indicates an event and has no value. More precisely, one cannot store a value into nor retrieve a value from a condition variable. If a thread must wait for an event to occur, that thread waits on the corresponding condition variable. If another thread causes an event to occur, that thread simply signals the corresponding condition variable. Thus, a condition variable has a queue for those threads that are waiting the corresponding event to occur to wait on, and, as a result, the original monitor is extended to

the following. The private data section now can have a number of condition variables, each of which has a queue for threads to wait on.

10.2.1 Condition Variable Operations: Wait and Signal

There are only two operations that can be applied to a condition variable: *wait* and *signal*. When a thread executes a wait call (in a monitor, of course) on a condition variable, it is immediately suspended and put into the waiting queue of that condition variable. Thus, this thread is suspended and is waiting for the event that is represented by the condition variable to occur. Because the calling thread is the only thread that is running in the monitor, it "owns" the monitor lock. When it is put into the waiting queue of a condition variable, the system will automatically take the monitor lock back. As a result, the monitor becomes empty and another thread can enter. Eventually, a thread will cause the event to occur. To indicate a particular event occurs, a thread calls the signal method on the corresponding condition variable. At this point, we have two cases to consider. First, if there are threads waiting on the signaled condition variable, the monitor will allow one of the waiting threads to resume its execution and give this thread the monitor lock back. Second, if there is no waiting thread on the signaled condition variable, this signal is lost as if it never occurs.

There is a third operation *Broadcast()* that wakes up all waiting threads in the queue, instead of just one.

10.2.2 Semaphore vs Condition Variable

- Semaphores can be used anywhere in a program, but should not be used in a monitor. Condition variables can only be used in montors.
- In semaphores, Wait() does not always block the caller (i.e., when the semaphore counter is greater than zero). In condition variables Wait() always blocks the caller.
- In semaphores Signal() either releases a blocked thread, if there is one, or increases the semaphore counter. In condition variables Signal() either releases a blocked thread, if there is one, or the signal is lost as if it never happens. So, semaphores maintain a history of all past signals whereas condition variables do not.
- In semaphores, if Signal() releases a blocked thread, the caller and the released thread both continue. In condition variables, if Signal() releases a blocked thread, the caller yields the monitor (Hoare type) or continues (Mesa Type). Only one of the caller or the released thread can continue, but not both.

10.2.3 Monitor types

A signal on a condition variable causes a waiting thread of that condition variable to resume its execution. Note that the released thread will have its monitor lock back. So, what happens to the lock owned by the signaling thread? The signaling thread must be running so that it can signal. Consequently, the signaling thread must own the monitor lock! Now, once the released thread runs, this thread and the signaling thread would both own the monitor lock and both be running in monitor! This violates the mutual exclusion requirement of monitors. Therefore, to make sure that mutual exclusion is guaranteed for monitors, either the released thread or the signaling thread can run, but not both. The choice of which thread should run creates at least two types of monitors: the *Hoare* type and the *Mesa* type.

10.2.3.1 The Hoare Type Monitors

In Hoare's original 1974 monitor definition, the signaler yields the monitor to the released thread. More specifically, if thread A signals a condition variable CV on which there are threads waiting, one of the waiting threads, say B, will be released immediately. Before B can run, A is suspended and its monitor lock is taken away by the monitor. Then, the monitor lock is given to the released thread B so that when it runs it is the only thread executing in the monitor. Sometime later, when the monitor becomes free, the signaling thread A will have a chance to run. This type of monitor does have its merit. If thread B is waiting on condition variable CV when thread A signals, this means B entered the monitor earlier than A did, and A should yield the monitor to a "senior" thread who might have a more urgent task to perform. Because the released thread runs immediately right after the signaler indicates the event has occurred, the released thread can run without worrying about the event has been changed between the time the condition variable is signaled and the time the released thread runs. This would simplify our programming effort somewhat.

10.2.3.2 The Mesa Type Monitors

Mesa is a programming language developed by a group of Xerox researchers in late 70s, and supports multithreaded programming. Mesa also implements monitors, but in a different style for efficiency purpose. With Mesa, the signaling thread continues and the released thread yields the monitor. More specifically, when thread A signals a condition variable CV and if there is no waiting thread, the signal is lost just like Hoare's type. If condition variable CV has waiting threads, one of them, say B, is released. However, B does not get his monitor lock back. Instead, B must wait until the monitor becomes empty to get a chance to run. The signaling thread A continues to run in the monitor.