CMPSCI 377 Operating Systems **Spring 2010**

Lecture 9: February 18

Lecturer: Prashant Shenoy *TA: Vimal Mathew & Tim Wood*

9.1 Monitors

While semaphores can be a powerful synchronization mechanism, they have some drawbacks which make them difficult to use in practice. Semaphores are a low level mechanism, and improperly placed signal and wait calls can lead to incorrect behavior. Additionally, semaphores are a global data structure which are not explicitly tied to a critical section–they can be called from any thread at any point in the program. These characteristics can cause semaphores to be difficult to use, and improper usage can easily lead to bugs.

A monitor is a higher level synchronization mechanism that tries to resolve some of these issues. A monitor can be viewed as a class that encapsulates a set of shared data as well as the operations on that data (e.g. the critical sections). The monitor is implemented in such a way so as to guarantee mutual exclusion–only one thread calling a method in the monitor class can run at a time. We consider a thread to be "in a monitor" if it has acquired control of the monitor; the monitor automatically ensures that only one thread can be in the monitor at a time.

A monitor contains a lock and a set of *condition variables*. The lock is used to enforce mutual exclusion. The condition variables are used as wait queues so that other threads can sleep while the lock is held. Thus condition variables make it so that if a thread can safely go to sleep and be guaranteed that when they wake up they will have control of the lock again. In Java, monitors can be used by adding the *synchronized* keyword to a method declaration. This will make it so that only one thread can execute the method at a time. This eliminates the need to manually acquire and release locks or call semaphore operations–the mutual exclusion is provided through the monitor system.

9.2 Condition Variables

A thread in a monitor may have to block itself so it can wait for an event, but this can cause a problem since the thread has already acquired the monitor lock–if it were to go to sleep, no other thread could enter the monitor. To allow a thread to wait for an event (e.g., I/O completion) without holding on to the lock, condition variables are used. 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 for the corresponding event to occur. Each monitor may have multiple condition variables.

9.2.1 Condition Variable Operations: Wait and Signal

There are only two operations that can be applied to a condition variable: *wait* and *signal*, which are similar to the semaphore calls. When a thread executes a wait call 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.

9.2.2 Monitor types

A signal on a condition variable causes a waiting thread of that condition variable to resume its execution. However, this is a potential problem because the thread which called signal must already be running code within the monitor! To deal with this, the signaling thread must either yield control to the newly awoken thread, or the woken up thread may have to defer its operation until the signaling thread completes. The choice of which thread should run creates at least two types of monitors: the *Hoare* type and the *Mesa* type.

9.2.2.1 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.

9.2.2.2 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– because of this the Mesa style is used in most recent operating systems, as well as programming languages such as Java. 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.

9.2.3 Semaphore vs Condition Variable

Semaphores, Monitors, and Condition Variables are all similar. The main difference is that semaphores are tracking a history. This is because a semaphore is implemented as a counter. Condition variables, however, do not naturally track any history. This means that repeated calls to signal in a semaphore will repeatedly increment the counter, allowing multiple threads to enter the critical section if they are started at a later date. With a condition variable, a signal call will only impact any threads currently waiting–any threads started at a later date will not be impacted. Here are some of the key differences between the two approaches.

- Semaphores can be used anywhere in a program, while condition variables can only be used in monitors.
- 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.

Despite these differences, condition variables and semaphores can each be used to implement the other. This can be done by adding additional state to the condition variable so that it tracks the history of signal and wait calls.