CMPSCI 377 Operating Systems **Fall 2013**

Lecture 9: October 2

Lecturer: Prashant Shenoy *Scribe: Armand Halbert*

9.1 Monitors

We start with a quick review of monitors. A monitor contains a lock and a set of condition variables. This embeds synchronization into an object oriented framework so that an object method defined as synchronized must get the object lock before executing. With monitors there can be zero or more condition variables that allow a thread to go to sleep inside of a critical section and release their lock at the same time. Condition variables use three methods:

- wait: release the lock and go to sleep, when the thread wakes up it will re-acquire the lock
- signal: wake up a waiting thread if one exists, otherwise do nothing
- broadcast: wake up all waiting threads

Java has monitors built in to the language, but C++ doesn't provide monitors directly. Monitors can be implemented in C++ by following the monitor rules for acquiring and releasing locks.

9.1.1 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 signalling thread must either yield control to the newly awoken thread, or the woken up thread may have to defer its operation until the signalling thread completes. The choice of which thread should run creates at least two types of monitors: the *Hoare* type and the *Mesa* type.

9.1.1.1 The Hoare Type Monitors

In Hoare's original 1974 monitor definition, the signaller 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 signalling 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 signaller 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 signalled and the time the released thread runs. This would simplify our programming effort somewhat.

9.1.1.2 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 signalling 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 signalling thread A continues to run in the monitor.

9.1.2 Semaphore vs Condition Variable

Semaphores, Monitors, and Condition Variables are all similar. The main difference is that semaphores are tracking state. 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.
- Semaphore Wait() and Signal() are commutative (the result is the same regardless of the order of execution).
- 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.

9.2 Readers/Writers Problem

The readers-writers problem is a classic example of a common computing problem in concurrency. The problem deals with situations in which many threads must access the same shared memory at one time. These threads are broken up into two classes: Readers who read data but never modify it and Writers who read data and modify it. Only one Writer may access the data at a time so that multiple threads cannot modify the same data at the same time. An arbitrary number of Readers may access the data at a time and

it is safe to do so because they never modify the data. Also, Readers and Writers cannot access the data at the same time. Every read or write of the shared data must be within a critical section.

This is useful in many systems in order to optimize performance–if many threads want to read a data structure there is no need to limit them to accessing the data one at a time. However, synchronization still must be insured in the case of a writer–only one writer should be able to write at a time, and no reader should run concurrently with the writer. Using a simple set of locks like we have described previously will be inefficient because we want multiple Readers reading at once.

9.2.1 Three types of solutions

There are several different solutions which can be used to build a Readers/Writers system.

1. Reader preferred: waiting readers go before waiting writers. A constraint is added so that no reader shall be kept waiting if the share is currently opened for reading.
2. Writer preferred: waiting writers go before waiting readers. A constraint is added so that no writer, once added to the queue, shall be kept waiting longer than absolutely necessary.
3. Neither preferred: try to treat readers and writers fairly (a simple queue is not good enough; we want parallel readers whenever possible).

In fact, the solutions implied by the first two approaches - *reader preferred* and *writer preferred* - result in starvation; the *reader preferred problem* may starve writers in the queue, and the *writer preferred problem* may starve readers. Therefore, *neither preferred* is sometimes proposed, which adds the constraint that no thread shall be allowed to starve. Solutions to this will necessarily sometimes require readers to wait even though the share is opened for reading, and sometimes require writers to wait longer than absolutely necessary.

Look at the lecture slides for pseudocode for implementing either Reader preferred or Writer preferred.

9.2.2 Readers/Writers in Java

The Java programming language can be used to solve the Readers/Writers problem in several different ways. Monitors can be used in order to synchronize the methods that control access for readers and writers. It is important to be careful when implementing a solution using monitors because calling a synchronized method inside of another synchronized method will result in deadlock. This is because the caller already has the lock and the called method is requesting that same lock. Alternatively, Java supports the idea of a read/write lock using the *ReadWriteLock* class. This is a special lock class that has different lock calls used by either readers or writers. The lock automatically enforces the rule that multiple readers can obtain the lock at once, but only one writer can be allowed. Note that pthread also has read/write lock constructs.