CMPSCI 377 Operating Systems **Fall 2005**

Lecture 9: October 13

Lecturer: Emery Berger *Scribes: Billy Dean and Zeid Rusan*

Last time: Locks, Semaphores, and Busy Waiting, etc.

Today we're going to learn about more synchronization primitives:

- Read/Write Locks
- Monitors
- Condition Variables

9.1 More Synchronization Primitives

9.1.1 Read/Write Locks

The problem with readers and writers is that they are a group of objects that cannot run simultaneously. This can be solved using single locks, but that does not allow groups of reads at once. Another way to approach this is to have two types of locks: Reader locks and Writer locks.

How to build Reader/Writer blocks:

- As long as there are no writers, let a reader in
- Writers queue up
- When all readers are done, allow one writer (readers queue up when that writer is executed)

WARNING: This can starve the writers.

Two solutions to this:

1. Waiting writer always gets served
2. No reader waits unless writer is already being run

Good technique is to switch between those two variant modes to ensure equalty.

9.2 Using Semaphores

You can implement these locks with semaphores. One is a Reader/Semaphore, the other is a Writer/Semaphore.

- readerSeaphore: protects number of readers allowed

- writerSemaphore: controls scheduling of writers (allows one in at a time)

[See the first Java Code example in the Lecture Slides for more details]

Basically, the code behaves in the following way:
If there are no readers, then the writerSemaphore allows a writer to execute. A readerSemaphore increments the number of readers when it is run. If the reader is the first one, then it blocks the writers. If the reader is the last one, then the writers are unblocked after it completes. Writers wait for an empty slot, then write (they wait for readers and other writers to complete).

9.2.1 Problems with Semaphores and Locks

They do two things:

1. Mutual exclusion
2. Scheduling

But, those require shared global variables. The program can be broken; it's not structured and it does not have any relationship with its protective data. We want a synchronization operating that does not suffer from this.

9.3 Monitors

Monitors tie mutal exclusion to a particular set of objects. Monitors are similar to Java classes in that all their data is private and all their methods are synchronized.

We define the Java keyword 'synchronize' as more than just a lock; it's a recursive lock that a thread can keep reusing. It contains a thread ID, the lock itself, and a count to keep track of the number of recursive calls (used to know when to release the lock).

9.3.1 Problems with Monitors

What to do with when the queue is empty:

- Return special error value (Ex. null)
- Throw an exception
- Wait for something to appear in queue (go to sleep for now). However, sleep is inside synchronize, so it will hold the lock when it sleeps.

The solution to this problem is to use Condtion Variables.

9.4 Conditon Variables

- Has queue of threads waiting in critical section
- Thread must hold back when performing operations
- method: wait(lock 1) - atomically releases lock, then goes to sleep
- reaquires lock when awakened
- method: notify() - wakes up one waiting

[See the R/W lock Java Code example in the Lecture Sldes for more details]

9.5 Summary

Benifits of Reader/Writer locks:

- Permit concurrent reads
- Implementable with Semaphores

Benifits of Monitors:

- Tie data and methods with synchronization

Benifits of Condition Variables:

- They release the lock temporarily, waiting inside critical sections

For more information see: Doug Lea's book "Concurrent Programming in Java"