CMPSCI 691W Parallel and Concurrent Programming	**Spring 2006**
Lecture 3: February 8	
Lecturer: Emery Berger	*Scribe: Kevin Grimaldi*

3.1 Overview

This lecture introduces various tools available for synchronization in concurrent programs. Synchronization is used to avoid race conditions and to coordinate the actions of threads. Locks can be used to allow multiple threads to safely access shared data. Common pitfalls and problems that arise from using locks are also covered.

3.2 Thread Safety

One of the primary motivations for synchronization is to allow multiple threads to safely share state. When multiple accesses to a shared resource are made simultaneously, they are only safe if:

- All accesses are read-only
- All accesses are idempotent, or can be done repeatedly and yield the same result
- Only one thread accesses the resource at any given time

The "too much milk" problem illustrates the need for coordination amongst multiple threads accessing a resource. If you arrive home, look in the fridge and find no milk and then head to the store to buy milk, your roommate may in the meantime come home and also go to the store after noticing the lack of milk. Without some sort of synchronization you can end up with too much milk which then goes bad. Simply checking for a note does not solve the problem either, as if your roommate comes home and checks for milk and a note in between when you check for milk and a note and then leave a note, you can still end up with too much milk. Locks are needed to solve this problem.

3.3 Locks

Locks allow mutual exclusion, or preventing more than one thread from entering a section of code, called a *critical section*, at the same time. To implement locks hardware-level atomic updates are required, such as test-and-set or compare-and-swap. Test-and-set atomically sets the value of a word to 1 and returns the previous value.

These atomic operations, unfortunately are very slow especially on Intel architectures. They generally require the entire pipeline to be flushed, making them orders of magnitude slower than the equivalent non-atomic operations.

3.3.1 Blocking Locks

Different types of locks have different semantics for how they interact with thread scheduling when the lock is contended for. The first type of locks, blocking locks, simply suspend the current thread immediately when they try to acquire a lock held by another thread, allowing other threads to run. This minimizes processor time spent waiting for the lock to be released, but is guaranteed to cause a context switch any time a thread tries to acquire a lock that is already held. The cost of the context switch along with the effects that it has on cache can be nontrivial.

3.3.2 Spin Locks

As the name implies, spin locks just spin in a loop waiting for the lock to be released instead of suspending the thread. This can sometimes avoid the cost of a context switch, but unfortunately can sometimes waste a lot of processor time doing nothing. Note that spin locks only make sense in the context of multiprocessor systems. On a uniprocessor system they are guaranteed to result in doing nothing until the expiration of the thread's quanta.

3.3.3 Hybrid Locks

Instead of taking an all-or-nothing approach, hybrid locks spin for some period of time in the hopes of avoiding unnecessary context switches, but then yield to another thread after a certain period of time to avoid wasting processor time. Different variants either use a fixed timeout or an exponential backoff algorithm.

Another variant of traditional locks, called queueing locks, maintain a FIFO queue of threads waiting for a lock to ensure fairness and scalability. These were a hot research topic in the nineties but due to performance issues have not been used in real systems.

3.4 Problems with Locks

Locks can be used to enfore mutual exclusion but also introduce the possibility of various errors.

3.4.1 Forgetting to Unlock

One very common error encountered in the use of locks, especially in C or C++ is forgetting to unlock when done with a lock. This can sometimes be hard to spot especially when there are several ways for a function to return, or for example if an exception occurs in C++.

A solution to this problem is to use the resource acquisition is initialization paradigm. An object is created on the stack that acquires a lock during construction and then releases the lock when the destructor is called which will occur when the object goes out of scope, no matter how the code block is exited. This is similar to the way that locks are handled in Java, where all of the built in locks are scoped.

3.4.2 Double Locking

Certain other problems such as leaving the *un* in `unlock` out can occur in C. This results in deadlock since the thread is waiting for itself to release the lock. This particular problem can be found relatively easily

using static analysis tools.

A solution to the double locking problem is recursive locks. When recursive locks are acquired, the current thread id is recorded, and a counter is started to track the number of times the lock has been acquired. If the same thread tries to acquire the lock again, instead of deadlocking it just increments the count. When `unlock` is called, the count is decremented and the lock is only really released if the counter reaches 0.

3.4.3 Deadlock

Another common problem encountered in locks is deadlock. If a cycle occurs in the locking graph, where edges are drawn from one thread to another if the first is waiting for a lock that is currently held by the second, then the threads will deadlock. In the following example deadlock occurs since each thread is waiting on the other to release a lock:

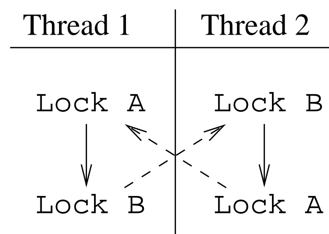

Figure 3.1: Threads Deadlocking

The standard solution to this problem is to enforce some canonical ordering on locks. As long as locks are always acquired in increasing order, deadlock freedom is guaranteed. It is also good practice to release them in decreasing order, but the order in which they are released is not important in preventing deadlock. Adhering to this ordering in acquiring locks can be difficult, however.

3.4.4 Priority Inversion

The final problem encountered in using locks is priority inversion. While this is not technically an error, it can cause problems. Priority inversion occurs when a thread with low priority acquires a lock that a thread with higher priority also wants. The thread with lower priority makes more progress even though it has lower priority because the thread with higher priority cannot run until the lock is released by the low priority thread.

3.4.5 Increasing Concurrency

Sometimes mutexes can be overly restrictive of concurrency. In certain cases objects are shared amongst a large group of threads, with the majority of them only performing reads and a certain subset actually updating the data. In these instances mutexes can be used to serialize access to the object, but this prevents multiple readers from making progress simultaneously even though it is safe for multiple threads to read the object at the same time.

To increase concurrency in such situations, read-write locks can be used. A read-write lock allows multiple threads to acquire the lock for read access but ensures exclusive access for write operations.

These locks introduce several options in terms of what to do when both readers and writers are queued up to obtain the lock. If readers are favored over writers, the writers can be starved while favoring the writers can result in readers being starved. A more complicated solution is to alternate between readers and writers every time the lock is acquired and released. This prevents either type of thread from starving.

3.5 Coordination

While safety is provided by mutexes and read-write locks, threads can coordinate with each other using other primitives such as semaphores or condition variables.

3.5.1 Semaphores

The general definition of a semaphore is a visual signaling apparatus, such as a traffic light. This is vaguely related to the computer science definition of a sempahore which is "a non-negative integer counter with atomic increment and decrement" that will block whenever decrementing would make it go negative.

Given a semaphore threads can increment it and decrement it. If a thread tries to decrement the semaphore when it is already at zero, the thread will go to sleep until another thread increments the semaphore. This can be used to signal between threads, allowing a thread for example to wait for an event to occur.

Semaphores can also be used when a certain maximum number of threads should be given access to a resource simultaneously. To be used in this manner the semaphore is initialized to the number of threads that can use the resource and then threads decrement the semaphore before using the resource and increment the semaphore again when done.

3.5.2 Condition Variables

Suppose that you want to make a blocking queue such that threads go to sleep if they try to dequeue an item from an empty queue. To check if the queue is empty a thread must hold a lock for the queue, but then to wait for something to be put in the queue the thread needs to go to sleep until it is signalled by a thread placing something in the queue. If this whole process is not done atomically, an item might be placed in the queue between when the thread checks the queue and goes to sleep and then the thread will never be signalled. On the other hand if the thread goes to sleep while holding the lock, no one else will ever be able to get the lock to put an item in the queue.

To solve this problem, condition variables are used. They are neither conditions nor variables. Condition variables allow a thread to atomically release a lock and then go to sleep. Other threads can then either wake up a single waiting thread or wake up all waiting threads. Waking up all waiting threads should be used with caution as it can result in what is known as the thundering herd problem, where a large group of threads wake up contending for the same object.