CMPSCI 377 Operating Systems **Spring 2009**

Lecture 8: October 1

Lecturer: Emery Berger *Scribes: Bruno Silva,Jim Partan*

8.1 Thread safety

What does it mean for something to be thread-safe? By saying that X is thread-safe we mean that if multiple threads use X at the same time, we don't have to worry about concurrency problems. The STL, for instance, is **not** thread-safe; if we were to create an STL queue and have two threads to operate on it simultaneously, we would have to manually perform all locking operations. The *cout* instruction is also not thread-safe.

Suppose now we want to build a thread-safe queue; the methods we want the queue to have are *insert(item)*, *remove()* and *empty()*. The first question to be asked is what should *remove()* do when the queue is empty. One solution would be for it to return a special value (such as NULL or -1, etc) or to throw an exception. It would much more elegant and useful, however, to make that function call wait until something actually appears in the queue. By implementing this type of blocking system, we are in fact implementing part of a *producer-consumer* system.

Now let us think of how to make the function wait. We can spin, i.e. write something like *while (empty()) ;*. This, however, obviously doesn't work, since the test of emptiness needs to *read shared data*; we need to put locks somewhere! And if we lock around the *while(empty());* line, the program will hang forever. Several other naïve approaches do not work (see slides for examples). The conclusion is that we need some way of going to sleep and at the same time having someone to wake us up when there's something interesting to do. Let us now present several possible implementations for this system and discuss why they do *not* work. The first possibility is:

```
dequeue()
   lock()     // needs to lock before checking if it's empty
   if (queue empty)
       sleep()
   remove_item()
   unlock()

enqueue()
   lock()
   insert_item()
   if (thread waiting)
       wake up dequeuer
   unlock()
```

One problem with this approach is that the dequeuer goes to sleep while holding the lock! How about the second possible approach:

```
dequeue()
   lock()     // need to lock before checking if it's empty
   if (queue empty) {
       unlock()
       sleep()
   }
   remove_item()
   unlock()


enqueue()
   lock()
   insert_item()
   if (thread waiting)
       wake up dequeuer
   unlock()
```

One problem here is that we are using an "if" instead of a "while" when checking whether or not the queue is empty. Consider the case where the dequeuer is sleeping, waiting for the enqueuer to insert an item. The enqueuer inserts an item, sends a wake-up signal, and the dequeuer wakes up. But then another dequeuer thread runs and removes the item. The first dequeuer thread now tries to remove an item from an empty queue. This problem can be fixed by using a "while" instead of the "if" statement, like this:

```
dequeue()
   lock()     // need to lock before checking if it's empty
   while (queue empty) {
       unlock()
       sleep()
   }
   remove_item()
   unlock()


enqueue()
   lock()
   insert_item()
   if (thread waiting)
       wake up dequeuer
   unlock()
```

This presents a more subtle and harder problem: the dequeuer might unlock, and then before it actually executes the sleep() function, the enqueuer could get the CPU back. In this case, the enqueuer would acquire the lock and insert the new item, but since there would be no one sleeping, there would be no thread to wake up, so the enqueuer would simply unlock, and never wake up the dequeuer again. Then the dequeuer would get the CPU back, finish calling sleep(), and sleep forever, even though the queue is not empty. The main issue is that another thread can run between the unlock() and the sleep() calls in dequeue().

The general solution to this type of problem is to make "unlock + sleep" atomic, which we will use to build our function called wait():

```
wait(lock l, cv c) {
    unlock_and_sleep(l,c);  // in one atomic step, unlock and sleep, waiting for c.v. c
    lock(l);                // re-acquire the lock
}
```

We use *condition variables* to do exactly that: condition variables make it possible and easy to go to sleep, by atomically releasing the lock, putting the thread on the waiting queue and going to sleep. Each condition variable has a "wait queue" of threads waiting on it, managed by the threads library. There are three important functions to be used when dealing with condition variables:

- *wait(lock l, cv c)*: atomically releases the lock and goes to sleep. When calling wait, we must be holding the lock. Depending upon the threads library, wait() may or may not re-acquire the lock when awakened (pthread_cond_wait() does re-acquire the lock);
- *signal(cv c)*: wakes up one waiting thread, if there are any;
- *broadcast(cv c)*: wakes up all waiting threads.

8.1.1 Producer-consumer using condition variables

Now let us present an implementation of a producer-consumer system using condition variables. This implementation works.

```
dequeue()
   lock(A)
   while (queue empty) {
         wait(A, C)    // atomically releases lock A and sleeps, waiting for
                       // condition variable C.
                       // When the thread wakes up, it re-acquires the lock.
                       // C is the condition variable that means ‘‘queue not empty’’.
   }
   remove_item()
   unlock(A)


enqueue()
   lock(A)
   insert_item()
   signal(C)
   unlock(A)
```

In dequeue() above, if the thread wakes up and by chance the queue is empty, there is no problem: that's why we need the "while" loop.