ECE459: Programming for Performance Winter 2011

Lecture 15 — March 1, 2011

Patrick Lam *version 1*

More on using locks

I was looking over the notes on locks and realized that there were a few more things that I should say about how to use them.

Lock Granularity

I mentioned the Linux 2.0 Big Kernel Lock. The key there was that if your lock protects too many resources, it slows down the program. I did not talk about how to actually set up locking schemes.

Recall that the goal of a locking scheme is to prevent races. At the operating-system level, you'd need to worry about two processes attempting to write to the same file or device. In userspace, the more common races are between two threads, as they try to access the same object or set of objects, and see inconsistent state: one thread is in the process of carrying out a non-atomic write, and another thread reads some of the writes of the first thread, but not all of them.

One Big Lock. Whenever you want to do anything that might cause a concurrency issue, grab the big lock. This scheme does not scale. Not so good for "programming for performance".

Object-level locking. If the object layout of your program maps well to resources that you want to protect, object-level locking is going to be an easy way to implement a reasonable locking strategy. This works best when each object encapsulates all relevant state for that object, or is a clear owner of a set of helper objects.

Example: if you have a tree of objects, you can lock the root.

Finer-grained locking. If you want to enable more contention without paying for it, you can also use finer-grained locks. For instance, when implementing a `Vector` class, you might use `size` and `capacity` integer fields as well as an `contents` array field. But the `get()` method only needs to access the `contents` and `size` fields. So you might protect those fields with one lock, and the `capacity` field with a second lock. See my ICSE 2010 paper on views for a programming language extension, "views", which allows you to specify fine-grained locking.

Re-entrancy

Another important property of your locking primitive is whether it is re-entrant or not. A thread can request a re-entrant lock multiple times, and then must release it the same number of times. Here's an example where this is useful:

```
class Foo {
  int x;

  public synchronized void f() {
    x--;
    g();
  }

  public synchronized void g() {
    x++;
  }
}
```

Because Java locks are re-entrant, you do get to call `f()` and not have it deadlock. If the locks were non-re-entrant, `f()` could successfully acquire the lock, but then the thread would always deadlock upon calling `g()`, which attempts to acquire the lock again.

glib provides both not-necessarily-reentrant locks (`GMutex`) and re-entrant locks (`GStaticRecMutex`).

Java Note: `ReentrantLock` versus built-in locks. Java 1.5 introduced the `ReentrantLock` class, which adds a few features to the built-in locking mechanism[1]. Somewhat misleadingly, both `ReentrantLock` and the default locks are re-entrant. The difference is that the `ReentrantLock` is more efficient and supports non-nested blocks. You can also write try-locks, as mentioned earlier, and timed waits using the `ReentrantLock`. One more feature is fairness: you can tell Java to wake up the thread that has been waiting the longest for a lock.

Here is a non-nested update:

```
Lock lock = new ReentrantLock();
lock.lock();
try {
  // you have the lock here; do the update.
}
finally { // you might have thrown an exception
  lock.unlock();
}
```

The `ReentrantLock` specification states that you should always put the `unlock()` call in a `finally` block, in case the lock body throws an exception.

[1]Good explanation at `http://www.ibm.com/developerworks/java/library/j-jtp10264/`.