ECE459: Programming for Performance Winter 2011

Lecture 11 — February 8, 2011

Patrick Lam *version 1*

Today we'll talk about race conditions, how to detect them, and how to avoid them. I briefly talked about locks last week, in the context of implementing a thread pool, and I expect that you will have seen them in the context of your Operating Systems class, but I'll focus on how to use them.

Race Conditions

Let's start by discussing race conditions in more detail; I mentioned them briefly in Lecture 9. I'll give you an example from Gove's book and outline another one. Recall that a race happens when you have two parallel accesses to the same state, one of which is a write.

Simple Example. Consider the following C method and its SPARC disassembly:

```
void update(int *a) {                ld   [%o0], %o1   // load *a
  *a = *a + 4;                       add  %o1, 4, %o1  // add 4
}                                    st   %o1, [%o0]   // store *a
```

If we call `update()` from two threads running in parallel on the same memory location, then we could get the wrong value at the end. (I recommend that you work out the scenario.) In brief, one of the threads updates a value based on information that becomes stale.

More Sophisticated Example. A wrong `int` value isn't catastrophic, and may actually be a harmless race. Linked list manipulations can cause bigger problems, including infinite loops. How?

Race detection

While the general problem of static race detection is still an active research area, there are a number of dynamic race detection tools that you can use today, including `helgrind`, Oracle Solaris Studio's Thread Analyzer, and the Coverity Thread Analyzer[1].

Static versus dynamic. Dynamic tools only tell you about potential races that could occur given a particular set of inputs. These might not be actual races, because you might have ensured race freedom some other way, for instance by writing lock-free code. The general problem with dynamic tools is that require good test data if they're to tell you anything useful.

[1] Coverity also produces static tools, but this isn't one of them.

Static tools analyze the code with respect to all possible inputs. They therefore flag all potential races. An excessive false positive rate (potential races that never occur in practice) is the problem for static tools, and some static tools can display the most likely actual problems first.

Helgrind example. Here's a program with a data race, again from Gove's book:

```
#include <pthread.h>

int counter = 0; // shared variable

void * func(void * params) { counter++; }

void main() {
  pthread_t thread1, thread2;
  pthread_create (&thread1, 0, func, 0); pthread_create (&thread2, 0, func, 0);
  pthread_join (thread1, 0); pthread_join(thread2, 0);
}
```

Running `helgrind` produces the following output. (Solaris Studio has a slicker interface but gives essentially the same output).

```
$ valgrind --tool=helgrind ./a.out
[...]
==27131== Possible data race during read of size 4 at 0x8049700 by thread #3
==27131==    at 0x8048487: func (race.c:5)
==27131==    by 0x4028734: mythread_wrapper (hg_intercepts.c:221)
==27131==    by 0x4048954: start_thread (pthread_create.c:300)
==27131==    by 0x4128E7D: clone (clone.S:130)
==27131==  This conflicts with a previous write of size 4 by thread #2
==27131==    at 0x804848F: func (race.c:5)
==27131==    by 0x4028734: mythread_wrapper (hg_intercepts.c:221)
==27131==    by 0x4048954: start_thread (pthread_create.c:300)
==27131==    by 0x4128E7D: clone (clone.S:130)
==27131==
```

In general, data races are hard to find, especially if you're investigating the code manually; tools do help, but come with runtime overhead.

Prevention. To avoid data races, you need to make sure that stale state doesn't propagate. One way of doing that is by using mutual exclusion (also known as locking). Atomic operations are another option. Here's some locking code:

```
void * func(void * params) {
  pthread_mutex_lock (&mutex); counter++; pthread_mutex_unlock (&mutex);
}
```

Synchronization Primitives

Operating systems and libraries typically provide a number of primitives to prevent data races. We'll investigate how you'd use mutexes/critical regions, spin locks, semaphores, reader/writer

locks, barriers, and lock-free code. Unless you have a really good reason, you're probably better off using OS primitives for synchronization.

We'll proceed in order of complexity.

Mutexes/critical regions. We've seen these just above, as well as in Java. Key idea: locks protect resources; only one thread can hold a lock at a time. A second thread trying to obtain the lock (i.e. *contending* for the lock) has to wait, or *block*, until the first thread releases the lock. So only one thread has access to the protected resource at a time. The code between the lock acquisition and release is known as the *critical region*.

Some mutex implementations also provide a "try-lock" primitive, which grabs the lock if it's available, or returns control to the thread if it's not, thus enabling the thread to do something else.

Excessive use of locks can serialize programs. Consider two resources A and B protected by a single lock ℓ. Then a thread that's just interested in B still has acquire ℓ, which requires it to wait for any other thread working with A. (The Linux kernel used to rely on a Big Kernel Lock protecting lots of resources in the 2.0 era, and Linux 2.2 improved performance on SMPs by cutting down on the use of the BKL.)

Note: in Windows, the term "mutex" refers to an inter-process communication mechanism. "Critical sections" are the mutexes we're talking about above.

Spinlocks. Spinlocks are a variant of mutexes, where the waiting thread repeatedly tries to acquire the lock instead of sleeping. Use spinlocks when you expect critical sections to finish quickly[2]. Spinning for a long time consumes lots of CPU resources. Many lock implementations use both sleeping and spinlocks: spin for a bit, then sleep for longer.

Reader/Writer Locks. Recall that data races only happen when one of the concurrent accesses is a write. So, if you have read-only ("immutable") data, as often occurs in functional programs, you don't need to protect access to that data. For instance, your program might have an initialization phase, where you write some data, and then a query phase, where you use multiple threads to read the data.

Unfortunately, sometimes your data is not read-only. It might, for instance, be rarely updated. Locking the data every time would be inefficient. The answer is to instead use a *reader/writer* lock. Multiple threads can hold the lock in read mode, but only one thread can hold the lock in write mode, and it will block until all the readers are done reading.

```
int readData(int c1, int c2) {
  g_static_rw_lock_reader_lock (&rwlock);
  int result = data[c1] + data[c2];
  g_static_rw_lock_reader_unlock (&rwlock);
}

void writeData(int c1, int c2, int value) {
```

[2]For more information on spinlocks in the Linux kernel, see `http://lkml.org/lkml/2003/6/14/146`.

```
  g_static_rw_lock_writer_lock (&rwlock);
  data[c1] += value; data[c2] -= value;
  g_static_rw_lock_writer_unlock (&rwlock);
}
```

Semaphores/condition variables. While semaphores can keep track of a counter and can implement mutexes, you should use them to support signalling between threads or processes.

In pthreads, semaphores can be used for inter-process communication, while condition variables are like Java's `wait()`/`notify()`.

Barriers. Think about the montage operation in assignment 2: you can't start writing the montage JPEG until all of the threads have contributed their thumbnails. This is a case where you want barriers: they allow you to make sure that a collection of threads all reach the barrier before finishing. In pthreads, each thread should call `pthread_barrier_wait()`, which will proceed when enough threads have reached the barrier. Enough means a number you specify upon barrier creation.

Lock-Free Code. We'll talk more about this in a few weeks. Modern CPUs support atomic operations, such as compare-and-swap, which enable experts to write lock-free code. A recent research result [McK11, AGH+11] states the requirements for correct implementations: basically, such implementations must contain certain synchronization constructs.

References

[AGH+11] Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin Vechev. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 487–498, New York, NY, USA, 2011. ACM.

[McK11] Paul McKenney. Concurrent code and expensive instructions. Linux Weekly News, `http://lwn.net/Articles/423994/`, January 2011.