ECE459: Programming for Performance Winter 2015

Lecture 9 — January 23, 2015

Patrick Lam *version 1*

More synchronization primitives

We'll proceed in order of complexity.

Recap: Mutexes. Recall that our goal in this course is to be able to use mutexes correctly. You should have seen how to implement them in an operating systems course. Here's how to use them.

- Call `lock` on mutex ℓ_1. Upon return from `lock`, your thread has exclusive access to ℓ_1 until it `unlock`s it.
- Other calls to `lock` ℓ_1 will not return until `m1` is available.

For background on implementing mutual exclusion, see Lamport's bakery algorithm. Implementation details are not in scope for this course.

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. (Kind of like non-blocking I/O!)

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[1]. 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. At some point, we saw a live coding example comparing spinlocks to normal mutexes.

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

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) {              // glib usage example
  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) {
  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 also be used for inter-process communication, while condition variables are like Java's `wait()/notify()`.

Barriers. This synchronization primitive allows 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.

Semaphores

As you learned in previous courses, semaphores have a `value` and can be used for signalling between threads. When you create a semaphore, you specify an initial value for that semaphore. Here's how they work.

- The value can be understood to represent the number of resources available.
- A semaphore has two fundamental operations: wait and post.
- wait reserves one instance of the protected resource, if currently available—that is, if value is currently above 0. If value is 0, then wait suspends the thread until some other thread makes the resource available.
- post releases one instance of the protected resource, incrementing value.

Semaphore Usage. Here are the relevant API calls.

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_destroy(sem_t *sem);
int sem_post(sem_t *sem);
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
```

This API is a lot like the mutex API:

- must link with -pthread (or -lrt on Solaris);
- all functions return 0 on success;
- same usage as mutexes in terms of passing pointers.

How could you use a semaphore as a mutex?

Semaphores for Signalling. Here's an example from the book. How would you make this always print "Thread 1" then "Thread 2" using semaphores?

```
#include <pthread.h>
#include <stdio.h>
#include <semaphore.h>
#include <stdlib.h>

void* p1 (void* arg) { printf("Thread 1\n"); }

void* p2 (void* arg) { printf("Thread 2\n"); }

int main(int argc, char *argv[])
{
    pthread_t thread[2];
    pthread_create(&thread[0], NULL, p1, NULL);
    pthread_create(&thread[1], NULL, p2, NULL);
    pthread_join(thread[0], NULL);
    pthread_join(thread[1], NULL);
    return EXIT_SUCCESS;
}
```

Proposed Solution. Is it actually correct?

```
sem_t sem;
void* p1 (void* arg) {
  printf("Thread 1\n");
  sem_post(&sem);
}
void* p2 (void* arg) {
  sem_wait(&sem);
  printf("Thread 2\n");
}

int main(int argc, char *argv[])
{
    pthread_t thread[2];
    sem_init(&sem, 0, /* value: */ 1);
    pthread_create(&thread[0], NULL, p1, NULL);
    pthread_create(&thread[1], NULL, p2, NULL);
    pthread_join(thread[0], NULL);
    pthread_join(thread[1], NULL);
    sem_destroy(&sem);
}
```

Well, let's reason through it.

- `value` is initially 1.
- Say `p2` hits its `sem_wait` first and succeeds.
- `value` is now 0 and `p2` prints "Thread 2" first.
- It would be OK if `p1` happened first. That would just increase `value` to 2.

Fix: set the initial `value` to 0. Then, if `p2` hits its `sem_wait` first, it will not print until `p1` posts, which is after `p1` prints "Thread 1".

The `volatile` qualifier

We'll continue by discussing C language features and how they affect the compiler. The `volatile` qualifier notifies the compiler that a variable may be changed by "external forces". It therefore ensures that the compiled code does an actual read from a variable every time a read appears (i.e. the compiler can't optimize away the read). It does not prevent re-ordering nor does it protect against races.

Here's an example.

```
int i = 0;

while (i != 255) { ... }
```

`volatile` prevents this from being optimized to:

```
int i = 0;

while (true) { ... }
```

Note that the variable will not actually be `volatile` in the critical section; most of the time, it only prevents useful optimizations. `volatile` is usually wrong unless there is a *very* good reason for it.

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.