ECE459: Programming for Performance Winter 2015

Lecture 6 — January 15, 2015

Patrick Lam *version 1*

Synchronization

You'll need some sort of synchronization to get sane results from multithreaded programs. We'll start by talking about how to use mutual exclusion in Pthreads.

Mutual Exclusion. Mutexes are the most basic type of synchronization. As a reminder:

- Only one thread can access code protected by a mutex at a time.
- All other threads must wait until the mutex is free before they can execute the protected code.

Here's two examples of using mutexes:

```
pthread_mutex_t m1_static = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t m2_dynamic;
pthread_mutex_init(&m2_dynamic, NULL);
...
pthread_mutex_destroy(&m1_static);
pthread_mutex_destroy(&m2_dynamic);
```

```
#include <mutex>

std::mutex m1, m2;
```

You can initialize Pthreads mutexes statically (as with `m1_static`) or dynamically (`m2_dynamic`). If you want to include attributes, you need to use the dynamic version. C++11 mutexes don't need to be explicitly destroyed; resources are freed when they go out of scope. They don't seem to have attributes.

Mutex Attributes. Both threads and mutexes use the notion of attributes. We won't talk about mutex attributes in any detail, but here are the three standard Pthreads ones.

- **Protocol**: specifies the protocol used to prevent priority inversions for a mutex.
- **Prioceiling**: specifies the priority ceiling of a mutex.
- **Process-shared**: specifies the process sharing of a mutex.

You can specify a mutex as *process shared* so that you can access it between processes. In that case, you need to use shared memory and `mmap`, which we won't get into.

Mutex Example. Let's see how this looks in practice. It is fairly simple:

```
// code
pthread_mutex_lock(&m1);
// protected code
pthread_mutex_unlock(&m1);
// more code
```

```
// code
m1.lock();
// protected code
m1.unlock();
// more code
```

- Everything within the `lock` and `unlock` is protected.
- Be careful to avoid deadlocks if you are using multiple mutexes (always acquire locks in the same order across threads).
- Another useful primitive is `pthread_mutex_trylock`, also known as `::try_lock()` in C++11 threads. We may come back to this later.

Data Races

Why are we bothering with locks? Data races. A data race occurs when two concurrent actions access the same variable and at least one of them is a **write**. (This shows up on Assignment 1!)

```
...
static int counter = 0;

void* run(void* arg) {
    for (int i = 0; i < 100; ++i) {
        ++counter;
    }
}

int main(int argc, char *argv[])
{
    // Create 8 threads
    // Join 8 threads
    printf("counter = %i\n", counter);
}
```

Is there a datarace in this example? If so, how would we fix it?

```
...
static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
static int counter = 0;

void* run(void* arg) {
    for (int i = 0; i < 100; ++i) {
        pthread_mutex_lock(&mutex);
        ++counter;
        pthread_mutex_unlock(&mutex);
    }
}

int main(int argc, char *argv[])
{
    // Create 8 threads
    // Join 8 threads
    pthread_mutex_destroy(&mutex);
    printf("counter = %i\n", counter);
}
```

(I'll leave expressing this in C++11 as an exercise to the reader.)