ECE459: Programming for Performance Winter 2015

Lecture 8 — January 21, 2015

Patrick Lam *version 1*

Race Conditions

Previous courses should have introduced the concept of a race condition. We'll be talking about them in greater detail in this course.

Definition. A race occurs when you have two concurrent accesses to the same memory location, at least one of which is a **write**.

When there's a race, the final state may not be the same as running one access to completion and then the other. (But it "usually" is. It's nondeterministic.) Race conditions typically arise between variables which are shared between threads.

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

void* run1(void* arg)
{
    int* x = (int*) arg;
    *x += 1;
}

void* run2(void* arg)
{
    int* x = (int*) arg;
    *x += 2;
}

int main(int argc, char *argv[])
{
    int* x = malloc(sizeof(int));
    *x = 1;
    pthread_t t1, t2;
    pthread_create(&t1, NULL, &run1, x);
    pthread_join(t1, NULL);
    pthread_create(&t2, NULL, &run2, x);
    pthread_join(t2, NULL);
    printf("%d\n", *x);
    free(x);
    return EXIT_SUCCESS;
}
```

Question: Do we have a data race? Why or why not?

Example 2. Here's another example; keep the same thread definitions.

```
int main(int argc, char *argv[])
{
    int* x = malloc(sizeof(int));
    *x = 1;
    pthread_t t1, t2;
    pthread_create(&t1, NULL, &run1, x);
    pthread_create(&t2, NULL, &run2, x);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("%d\n", *x);
    free(x);
    return EXIT_SUCCESS;
}
```

Now do we have a data race? Why or why not?

Tracing our Example Data Race. What are the possible outputs? (Assume that initially `*x` is 1.) We'll look at compiler intermediate code (three-address code) to tell.

```
run1                            run2
D.1 = *x;                       D.1 = *x;
D.2 = D.1 + 1;                  D.2 = D.1 + 2
*x = D.2;                       *x = D.2;
```

Memory reads and writes are key in data races.

Let's call the read and write from `run1` R1 and W1; R2 and W2 from `run2`. Assuming a sane[1] memory model, R_n must precede W_n. **C and C++ do not guarantee such a memory model in the presence of races.** This reasoning would actually only work if we declared `x` as `atomic` (see Lecture 10) and did the individual three-address code operations. Or, you could avoid this whole mess by using read-modify-write instructions.

Here are all possible orderings:

Order				`*x`
R1	W1	R2	W2	4
R1	R2	W1	W2	3
R1	R2	W2	W1	2
R2	W2	R1	W1	4
R2	R1	W2	W1	2
R2	R1	W1	W2	3

Detecting Data Races Automatically

Dynamic and static tools exist. They can help you find data races in your program. `helgrind` is one such tool. It runs your program and analyzes it (and causes a large slowdown).

Run with `valgrind --tool=helgrind <prog>`.

[1]sequentially consistent; sadly, many widely-used models are wilder than this.

It will warn you of possible data races along with locations. For useful debugging information, compile your program with debugging information (`-g` flag for `gcc`).

```
==5036== Possible data race during read of size 4 at
         0x53F2040 by thread #3
==5036== Locks held: none
==5036==    at 0x400710: run2 (in datarace.c:14)
...
==5036==
==5036== This conflicts with a previous write of size 4 by
         thread #2
==5036== Locks held: none
==5036==    at 0x400700: run1 (in datarace.c:8)
...
==5036==
==5036== Address 0x53F2040 is 0 bytes inside a block of size
         4 alloc'd
...
==5036==    by 0x4005AE: main (in datarace.c:19)
```

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 an example of using mutexes:

PThreads

```
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);
```

C++11

```
mutex m1;
mutex * m2;

m2 = new mutex();
// ...

delete (m2);
```

You can initialize mutexes statically (as with `m1_static`) or dynamically (`m2_dynamic`). If you want to include attributes, you need to use the dynamic version.

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 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:

PThreads

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

C++11 Threads

```
// 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`. 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?

Solution: use mutexes.

```
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);
}
```