Race Conditions

Previous courses (ECE 155, ECE 254) should have introduced the concept of a race condition. We'll be talking about them in greater detail in this course.

> *"Knock knock."*
> *"Race Condition."*
> *"Who's there?"*

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

This definition is a little bit strict. We could also say that there is a race condition if there is some form of output, such as writing to the console. If one thread is going to write "1" to the console and another is going to write "2", then we could have a race condition. If there is no co-ordination, we could get output of "12" or "21". If the order here is unimportant, there's no issue; but if one order is correct, then the appearance of the other is a bug.

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. The fact that the output is often "12" and only very occasionally "21" may make it very difficult to track down the source of the problem. Furthermore, if we end up adding additional logging statements or use the debugger or anything to that effect, we will change the timing and possibly suppress (cover up) the behaviour of the bug.

In other situations (e.g., processor design) these are sometimes referred to as data hazards or dependencies. The problem there is that we have something that needs to wait for something else. There are four basic possibilities to consider:

1. **RAW** (Read After Write) - The classic form of dependency. The read has to take place after the write, otherwise there's nothing to read, or an incorrect value will be read.

2. **WAR** (Write After Read) - A write cannot take place until the read has happened, to ensure the read takes the correct value.

3. **WAW** (Write After Write) - A write cannot take place because an earlier write needs to happen first. If we do them out of order, the final value may be out of date or otherwise incorrect.

4. **RAR** (Read After Read) - No such hazard!

Race conditions typically arise between variables which are shared between threads.

Example.

```
#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

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.

---

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

- **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	C++11 Threads

```
// 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`. 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);
}
```

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 `unlocks` it.

- Other calls to `lock` ℓ_1 will not return until `m1` is available.

For background on implementing mutual exclusion, see `http://en.wikipedia.org/wiki/Lamport\%27s_bakery_algorithm` Lamport's bakery algorithm. Implementation details are not in scope for this course. You did take ECE 254, right?

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.) Mac OS also used to have problems with this, using the big and small kernel locks (but this is something they got from using the Mach microkernel, which is a whole other story.).

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

When would we ever want to use a spinlock? After all, we spend so much time talking about how we would never ever want to wait in a busy loop. Well. What we normally expect is to block until the lock becomes available. But that means a process switch, and then a switch back in the future when the lock is available. This takes nonzero time so it's optimal to use a spinlock if the amount of time we expect to wait for the lock is less than the amount of time it would take to do two process switches. As long as we have a multicore system.

---

[2]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.

The "typical" use case for volatile is having some variable like `quit` as the condition of your infinite loop (`while (!quit)...`) and when something happens, say, you catch the Ctrl-C signal, you change the value of `quit` so your infinite loop will exit and the program will clean itself up nicely. The compiler doesn't necessarily notice that some other thread or signal handler or what have you will make changes to the value of `quit` and so it will probably conclude that it can optimize something like `!quit` to `true`, which is right the vast majority of the time, but wrong in that important scenario...

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. URL: `http://doi.acm.org/10.1145/1926385.1926442`.

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