ECE459: Programming for Performance Winter 2019

Lecture 7 — Race Conditions & Synchronization

Patrick Lam and Jeff Zarnett *2018-12-10*

Race Conditions

Previous courses (ECE 254 or equivalent) 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**. In earlier courses you probably just considered any shared accesses or shared data at all.

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. It's a write, but not necessarily to the same location. 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.

Dependencies. Let's now consider two sequential operations that we would like to execute in parallel. In other situations (e.g., processor design) we might say that there are dependencies between the operations. The problem 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 If we do them out of order, the final value may be out of date or otherwise incorrect.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!

	Read 2nd	**Write 2nd**
Read 1st	Read after read (RAR)	Write after read (WAR)
	No dependency	Antidependency
Write 1st	Read after write (RAW)	Write after write (WAW)
	True dependency	Output dependency

The no-dependency case (RAR) is clear. Declaring data immutable in your program is a good way to ensure no dependencies. In fact, making your data structures immutable is a good idea in general.

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

```c
#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;
}
```

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

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

```c
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` 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:

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

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

Let's look at an antidependency (WAR) example.

```
void antiDependency(int z) {
  int y = f(x);
  x = z + 1;
}
```

```
void fixedAntiDependency(int z) {
  int x_copy = x;
  int y = f(x_copy);
  x = z + 1;
}
```

Why is there a problem?

Finally, WAWs can also inhibit parallelization:

```
void outputDependency(int x, int z) {
  y = x + 1;
  y = z + 1;
}
```

```
void fixedOutputDependency(int x, int z) {
  y_copy = x + 1;
  y = z + 1;
}
```

In both of these cases, renaming or copying data can eliminate the dependence and enable parallelization. Of course, copying data also takes time and uses cache, so it's not free. One might change the output locations of both statements and then copy in the correct output. These are usually more useful when it's not just one access, but some sort of longer computation.

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 a bit. In a previous course you should have learned about semaphores at least, and hopefully a mutex as well. If not, you should take a look at the file `synchronization.pdf` in the course repository as it contains a nice recap of the basics.

Mutual Exclusion. Mutexes are an extremely common form of synchronization. When used properly, 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.

Probably you have only used the pthread kind of mutex before. Here's a side by side comparison of the pthread mutex usage as well as the C++11:

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. Mutexes use the notion of attributes. We won't talk about mutex attributes in any detail, but here are the three standard ones. In previous courses we probably said just use `NULL` for the attributes and that's quite fine. But you might want some of these things:

- **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.

There is also the idea of trylock: you attempt to lock the mutex in a way that you won't get blocked whether we acquire the lock or not. The function returns a value to indicate if we succeeded and it is mandatory that we check. If successful, proceed; if unsuccessful then we'll have to try again at some point. Again, see the reference document if you need a recap of this.

My turn! No, my turn! 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* or *critical section*.

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 us ing the Mach microkernel, which is a whole other story.). We will come back to the subject of how to avoid this problem soon.

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.

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.

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 soon. 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.

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

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.

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.

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.

1. Any number of readers may be in the critical section simultaneously.
2. Only one writer may be in the critical section (and when it is, no readers are allowed).

Or, to sum that up, a writer cannot enter the critical section while any other thread (whether reader or writer) is there. While a writer is in the critical section, neither readers nor writers may enter the critical section [**?**]. This is very often how file systems work: a file may be read concurrently by any number of threads, but only one thread may write to it at a time (and to prevent reading of inconsistent data, no thread may read during the write).

The type for the lock is `pthread_rwlock_t`. It is analogous, obviously, to the mutex type `pthread_mutex_t`. Let's consider the functions that we have:

```
pthread_rwlock_init( pthread_rwlock_t * rwlock, pthread_rwlockattr_t * attr )
pthread_rwlock_rdlock( pthread_rwlock_t * rwlock )
pthread_rwlock_tryrdlock( pthread_rwlock_t * rwlock )
pthread_rwlock_wrlock( pthread_rwlock_t * rwlock )
pthread_rwlock_trywrlock( pthread_rwlock_t  *  rwlock )
pthread_rwlock_unlock( pthread_rwlock_t * rwlock )
pthread_rwlock_destroy( pthread_rwlock_t * rwlock )
```

In general our syntax very much resembles that of the mutex (attribute initialization and destruction not shown but they do exist). There are some small noteworthy differences, other than obviously the different type of the structure passed. Whereas before we had functions for lock and trylock, we now have those split into readlock and writelock (each of which has its own trylock function). As before, we will return to the subject of how trylock works soon.

In theory, the same thread may lock the same rwlock n times; just remember to unlock it n times as well.

And speaking of unlock, there's no specifying whether you are releasing a read or write lock. This is because it is unnecessary; the implementation unlocks whatever type the calling thread was holding. Much like `close()`, if we can figure out what we're closing we don't need the caller of the function to specify what to do.

As for whether readers get priority, the specification says this is implementation defined. If possible, for threads of equal priority, a writer takes precedence over a reader. But your system may vary.

Consider the following example of the simple readers-writers (without writer priority and with risk of starvation) using the "old" way:

```
int readers;
pthread_mutex_t mutex;
sem_t roomEmpty;

void init( ) {
  readers = 0;
  pthread_mutex_init( &mutex, NULL );
  sem_init( &roomEmpty, 0, 1 );
}

void cleanup( ) {
  pthread_mutex_destroy( &mutex );
  sem_destroy( &roomEmpty );
}
```

```
void* writer( void* arg ) {
  sem_wait( &roomEmpty );
  write_data( arg );
  sem_post( &roomEmpty );
}

void* reader( void* read ) {
  pthread_mutex_lock( &mutex );
  readers++;
  if ( readers == 1 ) {
    sem_wait( &roomEmpty );
  }
  pthread_mutex_unlock( &mutex );
  read_data( arg );
  pthread_mutex_lock( &mutex );
  readers--;
  if ( readers == 0 ) {
    sem_post( &roomEmpty );
  }
  pthread_mutex_unlock( &mutex );
}
```

Now see it as it would be with the use of a rwlock!

```
pthread_rwlock_t rwlock;

void init( ) {
  pthread_rwlock_init( &rwlock, NULL );
}

void cleanup( ) {
  pthread_rwlock_destroy( &rwlock );
}
```

```
void* writer( void* arg ) {
  pthread_rwlock_wrlock( &rwlock );
  write_data( arg );
  pthread_rwlock_unlock( &rwlock );
}

void* reader( void* read ) {
  pthread_rwlock_rdlock( &rwlock );
  read_data( arg );
  pthread_rwlock_unlock( &rwlock );
}
```

The `volatile` qualifier

We'll continue by discussing C language features and how they affect the compiler. The `volatile` qualifier in C 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. This is different from the Java `volatile`.

Here's an example.

```
int i = 0;

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

`volatile` prevents this from being optimized to:

```
int i = 0;

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

Most of the time, `volatile` 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.