ECE459: Programming for Performance Winter 2018

Lecture 15 — C/C++11 Memory Model

Patrick Lam & Jeff Zarnett

C/C++11 Memory Model

We have talked about memory models in the context of OpenMP. Let's talk about the core languages now—that is, C and C++ [Lov14, BA08]—when not using OpenMP.

What outputs are possible from this example?

Thread 1:

```
foo = 7;
bar = 42;
```

Thread 2:

```
printf("%d\n", foo);
printf("%d\n", bar);
```

You might think "undefined", but actually it's worse than that. The C11 and C++11 language definitions don't even say what a thread is. Of course, there is Pthreads, but that is a library, not the language itself. So you can't even ask this question when talking about pre-C11/C++11 versions of C and C++. Well. Okay. You can ask, but the question makes no sense. Sort of like putting your hand up in class and saying "Where did you bury your oranges?". It's a syntactically valid question and it describes something that's possible, but it still makes no sense.

The older standards made no reference to any kind of CPU, memory architecture, cache strategy, or anything like that. Which on the one hand is nice and general, but on the other hand, it leads to problems. The "abstract machine" that the C and C++ standards refer to is inherently single threaded, making it actually impossible to write a portable multithreaded C or C++ program [Lov14]. The part that's impossible is that word "portable" – people write multithreaded C and C++ programs all the time (in this class, even) but they have system specific code and implementation-defined behaviour. Open up a pthreads library or some equivalent and sure enough, you will find something architecture specific in there.

C++11 (and C11) have improved the situation, though. There is actually a memory model (based on an abstract machine) and threading primitives such as mutexes, atomics, and memory barriers—the concepts that we have seen in this course. Now there are rules! Yes. We like rules. Okay. I, at least, like rules. C++11 defines how a compiler can generate code that accesses memory even when there is concurrency. There are also standard mutex operations and atomics and barriers and all those lovely things.

Now, we can ask the question about the behaviour of the above example. It does have undefined behaviour, since there is contended access to the variables `foo` and `bar`. How can we fix that?

Atomics. A good exam question: if `foo` is atomic, what are the possible outputs?

Thread 1:

```
foo.store(7);
bar.store(42);
```

Thread 2:

```
printf("%d\n", foo.load());
printf("%d\n", bar.load());
```

Alright, we have some defined behaviour now. Honestly, it depends how these things are scheduled, but the answer is one of the following set: {0/0, 7/42, 7/0, 0/42}. The answer depends on how they are interleaved. But at least we get some certainty that the output will be one of those four things and there's no chance of garbage because the print takes place during an assignment operation.

We probably still don't like this because we don't have mutual exclusion here and we can get several different answers, some of which are probably "wrong" (for whatever definition of a correct answer is), but at least our set

of potential wrong answers is smaller. So that's a start. Compilers have to follow the new rules in generating code, so their output will behave as if the architecture followed the standard memory model. That's something.

Good C++ Practice

Lots of people use postfix (`i++`) out of habit, but prefix (`++i`) is better.

In C, this isn't a problem. In some languages (like C++), it can be.

Why? Overloading. In C++, you can overload the `++` and `-` operators.

```
class X {
public:
  X& operator++();
  const X operator++(int);
...
};

X x;
++x; // x.operator++();
x++; // x.operator++(0);
```

Prefix is also known as **increment and fetch**, and might be implemented like this:

```
X& X::operator++() {
  *this += 1;
  return *this;
}
```

Postfix is also known as **fetch and increment**. Note that you have to make a copy of the old value:

```
const X X::operator++(int) {
  const X old = *this;
  ++(*this);
  return old;
}
```

So, if you're the least concerned about efficiency (and why else would you be taking programming for performance?), always use *prefix* increments/decrements instead of defaulting to postfix. This isn't really an issue if the operator is in a statement all on its own (e.g. a standalone line, or the last part of a for loop) because the compiler is (presumably) smart enough to know that this can be optimized as the return value is not assigned. Only use `postfix` when you really mean it, to be on the safe side.

Mind you, if you're doing something like `array[i++]` or something similarly "clever", you might want to think twice about this. There's a lot of potential for error or misunderstanding in a code review. Remember, clever is hard to grep for.

Locking Granularity

Alright, we already know that locks prevent data races. If this is news to you, how did you pass the operating systems course?! So we need to use them to prevent data races, but it's not as simple as it sounds. We have choices about the granularity of locking, and it is a trade-off (like always).

Coarse-grained locking is easier to write and harder to mess up, but it can significantly reduce opportunities for parallelism. *Fine-grained locking* requires more careful design, increases locking overhead and is more prone to bugs (deadlock etc). Locks' extents constitute their *granularity*. In coarse-grained locking, you lock large sections of your program with a big lock; in fine-grained locking, you divide the locks and protect smaller sections with multiple smaller locks.

We'll discuss three major concerns when using locks:

- overhead;
- contention; and
- deadlocks.

We aren't even talking about under-locking (i.e., remaining race conditions). We'll assume there are adequate locks and that data accesses are protected.

Lock Overhead. Using a lock isn't free. You pay:

- allocated memory for the locks;
- initialization and destruction time; and
- acquisition and release time.

These costs scale with the number of locks that you have.

Lock Contention. Most locking time is wasted waiting for the lock to become available. We can fix this by:

- making the locking regions smaller (more granular); or
- making more locks for independent sections.

Deadlocks. Finally, the more locks you have, the more you have to worry about deadlocks.

As you know, the key condition for a deadlock is waiting for a lock held by process X while holding a lock held by process X'. ($X = X'$ is allowed).

Okay, in a formal sense, the four conditions for deadlock are:

1. **Mutual Exclusion**: A resource belongs to, at most, one process at a time.
2. **Hold-and-Wait**: A process that is currently holding some resources may request additional resources and may be forced to wait for them.
3. **No Preemption**: A resource cannot be "taken" from the process that holds it; only the process currently holding that resource may release it.
4. **Circular-Wait**: A cycle in the resource allocation graph.

Consider, for instance, two processors trying to get two locks.

Thread 1	**Thread 2**
`Get Lock 1`	`Get Lock 2`
`Get Lock 2`	`Get Lock 1`
`Release Lock 2`	`Release Lock 1`
`Release Lock 1`	`Release Lock 2`

Processor 1 gets Lock 1, then Processor 2 gets Lock 2. Oops! They both wait for each other. (Deadlock!).

To avoid deadlocks, always be careful if your code **acquires a lock while holding one**. You have two choices: (1) ensure consistent ordering in acquiring locks; or (2) use trylock.

As an example of consistent ordering:

```
void f1() {
    lock(&l1);
    lock(&l2);
    // protected code
    unlock(&l2);
    unlock(&ll);
}
```

```
void f2() {
    lock(&l1);
    lock(&l2);
    // protected code
    unlock(&l2);
    unlock(&ll);
}
```

This code will not deadlock: you can only get **l2** if you have **l1**. Of course, it's harder to ensure a consistent deadlock when lock identity is not statically visible.

Or another example, with threads P and Q attempting to acquire `a` and `b`. Thread Q requests `b` first and then `a`, while P does the reverse. The deadlock would not take place if both threads requested these two resources in the same order, whether `a` then `b` or `b` then `a`. Of course, when they have names like this, a natural ordering (alphabetical, or perhaps reverse alphabetical) is obvious.

To generalize and formalize this principle, if the set of all resources in the system is $R = \{R_0, R_1, R_2, ...R_m\}$, we assign to each resource R_k a unique integer value. Let us define this function as $f(R_i)$, that maps a resource to an integer value. This integer value is used to compare two resources: if a process has been assigned resource R_i, that process may request R_j only if $f(R_j) > f(R_i)$. Note that this is a strictly greater-than relationship; if the process needs more than one of R_i then the request for all of these must be made at once (in a single request). To get R_i when already in possession of a resource R_j where $f(R_j) > f(R_i)$, the process must release any resources R_k where $f(R_k) \geq f(R_i)$. If these two protocols are followed, then a circular-wait condition cannot hold [SGG13].

Alternately, you can use trylock. Recall that Pthreads' `trylock` returns 0 if it gets the lock. But if it doesn't, your thread doesn't get blocked. Checking the return value is important, but at the very least, this code also won't deadlock: it will give up **l1** if it can't get **l2**.

```
void f1() {
    lock(&l1);
    while (trylock(&l2) != 0) {
        unlock(&l1);
        // wait
        lock(&l1);
    }
    // protected code
    unlock(&l2);
    unlock(&ll);
}
```

(Incidentaly, using trylocks can also help you measure lock contention.)

This prevents the hold and wait condition, which was one of the four conditions. A process attempts to lock a group of resources at once, and if it does not get everything it needs, it releases the locks it received and tries again. Thus a process does not wait while holding resources.

Coarse-Grained Locking

One way of avoiding problems due to locking is to use few locks (perhaps just one!). This is *coarse-grained locking*. It does have a couple of advantages:

- it is easier to implement;
- with one lock, there is no chance of deadlocking; and
- it has the lowest memory usage and setup time possible.

It also, however, has one big disadvantage in terms of programming for performance: your parallel program will quickly become sequential.

Example: Python (and other interpreters). Python puts a lock around the whole interpreter (known as the *global interpreter lock*). This is the main reason (most) scripting languages have poor parallel performance; Python's just an example.

Two major implications:

- The only performance benefit you'll see from threading is if one of the threads is waiting for IO.
- But: any non-I/O-bound threaded program will be **slower** than the sequential version (plus, the interpreter will slow down your system).

You might think "this is stupid, who would ever do this?" - a lot of OS kernels do in fact have (or at least had) a "big kernel lock", including Linux and the Mach Microkernel. This lasted in Linux for quite a while, from the advent of SMP support up until sometime in 2011. As much as this ruins performance, correctness is more important. We don't have a class "programming for correctness" (software testing? Hah!) because correctness is kind of assumed. What we want to do here is speed up our program as much as we can while maintaining correctness...

Fine-Grained Locking

On the other end of the spectrum is *fine-grained locking*. The big advantage: it maximizes parallelization in your program.

However, it also comes with a number of disadvantages:

- if your program isn't very parallel, it'll be mostly wasted memory and setup time;
- plus, you're now prone to deadlocks; and
- fine-grained locking is generally more error-prone (be sure you grab the right lock!)

Examples. Databases may lock fields / records / tables. (fine-grained $\rightarrow$ coarse-grained).

You can also lock individual objects (but beware: sometimes you need transactional guarantees.)

References

[BA08] Hans J. Boehm and Sarita V. Adve. Foundations of the c++ concurrency memory model, 2008. Online; accessed 16-December-2015. URL: http://rsim.cs.illinois.edu/Pubs/08PLDI.pdf.

[Lov14] Robert Love. How are the threading and memory models different in c++ as compared to c?, 2014. Online; accessed 16-December-2015. URL: http://www.quora.com/C++-programming-language/How-are-the-threading-and-memory-models-different-in-C++-as-compared-to-C.

[SGG13] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts (9th Edition)*. John Wiley & Sons, 2013.