ECE459: Programming for Performance Winter 2016

Lecture 16 — C/C++11 Memory Model

Patrick Lam

Cache Coherency

We have talked about memory models in the context of OpenMP. Let's talk about the core languages now—that is, C and C++[12]—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++.

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, 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());
```

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);
...
```

[1] http://www.quora.com/C++-programming-language/How-are-the-threading-and-memory-models-different-in-C++-as-compared-to-C

[2] http://rsim.cs.illinois.edu/Pubs/08PLDI.pdf

```
};

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.

Only use `postfix` when you really mean it, to be on the safe side.

Locking Granularity

Locks prevent data races.

However, using locks involves a trade-off between coarse-grained locking, which can significantly reduce opportunities for parallelism, and fine-grained locking, which requires more careful design, increases locking overhead and is more prone to bugs. 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.

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).

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).

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(&l1);
}
```

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

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.

Alternately, you can use trylock. Recall that Pthreads' `trylock` returns 0 if it gets the lock. So, 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(&l1);
}
```

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

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).

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. The Linux kernel used to have **one big lock** that essentially made the kernel sequential. This worked fine for single-processor systems! The introduction of symmetric multiprocessor systems (SMPs) required a more aggressive locking strategy, though, and the kernel now uses finer-grained locks for performance.

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

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

Live Coding Example. I ran the code from the midterm (augmented with code to populate the tree) using 1) a coarse-grained lock; and 2) per-item fine-grained locks.

Fine-grained locks were suspiciously fast.