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.

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`.