

Lecture 19 — Single Thread Performance, Compiler Optimizations

Patrick Lam p.lam@ece.uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

December 6, 2017

Before C/C++11:
no language-level definition of threads.

Not even a well-formed question to ask what this means:

Thread 1:

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

Thread 2:

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

Before C/C++11:

no language-level definition of threads.

Not even a well-formed question to ask what this means:

Thread 1:

Thread 2:

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

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

pre C/C++11: no such thing as a thread!

Language Support in C++11: Defining the Question

Now¹:

- a memory model
- primitives: mutexes, atomics, memory barriers.

Previous example has undefined behaviour
per C++11. (why?)

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

Language Support in C++11: Atomics

We've seen the notion of atomics. Here's the C++11 notation:

```
atomic<int> foo, bar;
```

Thread 1:

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

Thread 2:

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

What are the possible outputs? (good exam question!)

Lots of people use postfix out of habit, but prefix is better.

In C, this isn't a problem.

In some languages (like C++), it can be.

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);
```

Common Increment Implementations

Prefix is also known as **increment and fetch**.

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

Postfix is also known as **fetch and increment**.

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

If you're the least concerned about efficiency, always use **prefix** increments/decrements instead of defaulting to postfix.

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

Digression: The Wayside



(Daderot, Wikimedia Commons)

Nathaniel Hawthorne wrote:

I have been equally unsuccessful in my architectural projects; and have transformed a simple and small old farm-house into the absurdest anomaly you ever saw; but I really was not so much to blame here as the ~~programmer~~ village-carpenter, who took the matter into his own hands, and produced an unimaginable sort of thing instead of what I asked for. (January 1864)

Original budget: \$500 (\$7540 inflation-adjusted)

Actual cost: \$2000 (\$30160 inflation-adjusted)

Locks prevent data races.

- Locks' extents constitute their **granularity**—do you lock large sections of your program with a big lock, or do you divide the locks and protect smaller sections?

Concerns when using locks:

- overhead;
- contention; and
- deadlocks.

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.

Most locking time is wasted waiting for the lock to become available.

How can we fix this?

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

The more locks you have, the more you have to worry about deadlocks.

Key condition:

 waiting for a lock held by process X
while holding a lock held by process X' . ($X = X'$ allowed).

The four conditions for deadlock are:

- 1 Mutual Exclusion**
- 2 Hold-and-Wait**
- 3 No Preemption**
- 4 Circular-Wait**

Consider two processors trying to get two *locks*:

Thread 1

Get Lock 1

Get Lock 2

Release Lock 2

Release Lock 1

Thread 2

Get Lock 2

Get 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**).

Always be careful if
your code **acquires a lock while holding one**.

Here's how to prevent a deadlock:

- Ensure consistent ordering in acquiring locks; or
- Use `trylock`.

Preventing Deadlocks—Ensuring 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**.

If the set of all resources in the system is $R = \{R_0, R_1, R_2, \dots, R_m\}$:

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 $f(R_j) > f(R_i)$.

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.

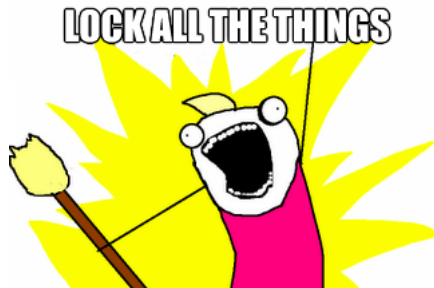
Preventing Deadlocks—Using `trylock`

Recall: Pthreads' `trylock` returns 0 if it gets the lock.

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

This code also won't deadlock: it will give up **l1** if it can't get **l2**.

(BTW: trylocks also enable measuring lock contention.)



(with one lock)

Advantages:

- Easier to implement;
- No chance of deadlocking;
- Lowest memory usage / setup time.

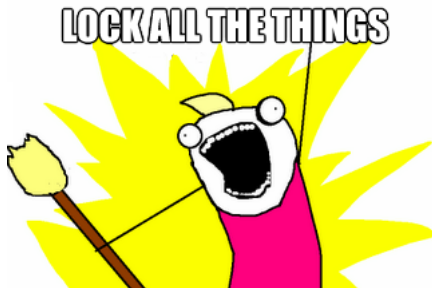
Disadvantages:

- Your parallel program can quickly become sequential.

Coarse-Grained Locking Example—Python GIL

This is the main reason (most) scripting languages have poor parallel performance; Python's just an example.

- Python puts a lock around the whole interpreter (global interpreter lock).
- Only performance benefit you'll see from threading is if a thread is waiting for IO.
- Any non-I/O-bound threaded program will be **slower** than the sequential version (plus, it'll slow down your system).



(with all different locks)

Advantages:

- Maximizes parallelization in your program.

Disadvantages

- May be mostly wasted memory / setup time.
- Prone to deadlocks.
- Generally more error-prone (be sure you grab the right lock!)

The Linux kernel used to have **one big lock** that essentially made the kernel sequential.

- (worked fine for single-processor systems!)

Now uses finer-grained locks for performance.

Databases may lock fields / records / tables.
(fine-grained → coarse-grained).

Can lock individual objects.

C++11 memory model.
Good increment practice.
Lock granularity.