Lecture 14 - Lock Granularity, Reentrancy and Inlining

ECE 459: Programming for Performance

Jon Eyolfson

University of Waterloo

February 3, 2012

Previous Lecture

- Memory ordering
- Memory fences / barriers

- Compare and swap atomic operation

- Using prefix instead of postfix

Locking

You need locks to prevent data races

- The extent of how you apply your locks is called it's **granularity**
- Do you lock large sections of your program, or divide the locks and use smaller sections?

Things to consider about locks:

- Overhead
- Contention
- Deadlocks

Locking Overhead

- Memory allocated
- Initialization and destruction time
- Time to acquire and release locks

The more locks you have, the greater each cost is going to be

Locking Contention

- Most locking time is wasted waiting for the lock
- Reduced by:
  - Making the locking region smaller (more granular)
  - Making more locks for independent sections

Locking Deadlocks

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

Conditions for deadlocking:

1. Mutual Exclusion (of coruse for simple locks)
2. Hold and Wait (you have a lock and try to acquire another)
3. No Preemption (we can't take simple locks away)
4. Circular Wait (waiting for a lock held by another process)

From the First Lecture

- Consider 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, now they both wait for eachother (deadlock)

Preventing Deadlocks

Always be careful if your code **acquires a lock while holding one**
Ways to prevent a deadlock:

- Ensure ordering in acquiring locks
- Using `trylock`

Preventing Deadlocks - Ensuring Order

```
void f1() {
    locktype_lock(&l1);
    locktype_lock(&l2);
    // protected code
    locktype_unlock(&l2);
    locktype_unlock(&ll);
}

void f2() {
    locktype_lock(&l1);
    locktype_lock(&l2);
    // protected code
    locktype_unlock(&l2);
    locktype_unlock(&ll);
}
```

- This code will not deadlock, you can only get **l2** if you have **l1**

Preventing Deadlocks - Using `trylock`

Remember, for Pthreads, `trylock` returns 0 if it gets the lock

```
void f1() {
    locktype_lock(&l1);
    while (locktype_trylock(&l2) != 0) {
        locktype_unlock(&l1);
        // wait
        locktype_lock(&l1);
    }
    // protected code
    locktype_unlock(&l2);
    locktype_unlock(&ll);
}
```

- This code will not deadlock, it will give up **l1** if it can't get **l2**

Coarse-Grained Locking (1)

(with one lock)

Coarse-Grained Locking (2)

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 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 none IO bound program will be **slower** than the sequential version (and slow down your system)

Fine-Grained Locking (1)

(with all different locks)

Fine-Grained Locking (2)

Advantages

- Maximizes parallelization in your program

Disadvantages

- May be mostly wasted memory / setup time
- Have to consider deadlocks
- More error prone (being sure you grab the right lock)

Fine-Grained Locking Examples

- The Linux kernel use to have **one big lock** that essentially made kernel mode sequential
- Now consists of finer-grained locks for performance
- Databases could lock either fields / records / tables (fine-grained to coarse-grained)
- You could also lock individual objects, etc

Reentrancy

- Means a function can be suspended in the middle and **re-entered** (called again) before the previous execution completes
- Does not always mean **thread-safe** (although it usually is)
  - Recall, **thread-safe** is essentially no data races

Avoided if the function only modifies local data

Reentrancy Example

Courtesy of Wikipedia (with modifications):

```
int t;

void swap(int *x, int *y) {
    t = *x;
    *x = *y;
    // hardware interrupt might invoke isr() here!
    *y = t;
}

void isr() {
    int x = 1, y = 2;
    swap(&x, &y);
}
...
int a = 3, b = 4;
...
    swap(&a, &b);
```

Reentrancy Example Explained

```
call swap(&a, &b);
t = *x;                     // t = 3 (a)
*x = *y;                    // a = 4 (b)
call isr();
    x = 1; y = 2;
    call swap(&x, &y)
    t = *x;                 // t = 1 (x)
    *x = *y;                // x = 2 (y)
    *y = t;                 // y = 1
*y = t;                     // b = 1

Final values:
a = 4, b = 1

Expected values:
a = 4, b = 3
```

Reentrancy Example Fixed

```
int t;

void swap(int *x, int *y) {
    int s;

    s = t;  // save global variable
    t = *x;
    *x = *y;
    // hardware interrupt might invoke isr() here!
    *y = t;
    t = s;  // restore global variable
}

void isr() {
    int x = 1, y = 2;
    swap(&x, &y);
}
...
int a = 3, b = 4;
...
    swap(&a, &b);
```

Reentrancy Example Fixed Explained

```
call swap(&a, &b);
s = t;                    // s = UNDEFINED
t = *x;                   // t = 3 (a)
*x = *y;                  // a = 4 (b)
call isr();
    x = 1; y = 2;
    call swap(&x, &y)
    s = t;                // s = 3
    t = *x;               // t = 1 (x)
    *x = *y;              // x = 2 (y)
    *y = t;               // y = 1
    t = s;                // t = 3
*y = t;                   // b = 3
t = s;                    // t = UNDEFINED

Final values:
a = 4, b = 3

Expected values:
a = 4, b = 3
```

Previous Example Thread-safety

Is the previous reentrant code thread safe?
(This is more what we're concerned about in this course)

Again:

```
int t;

void swap(int *x, int *y) {
    int s;

    s = t;  // save global variable
    t = *x;
    *x = *y;
    // hardware interrupt might invoke isr() here!
    *y = t;
    t = s;  // restore global variable
}
```

Possibly consider two calls: `swap(a, b)`, `swap(c, d)` with
`a = 1, b = 2, c = 3, d = 4`

Previous Example Thread-safety Explained

```
global: t

/* thread 1 */                          /* thread 2 */
a = 1, b = 2;
s = t;      // s = UNDEFINED
t = a;      // t = 1
                                        c = 3, d = 4;
                                        s = t;      // s = 1
                                        t = c;      // t = 3
                                        c = d;      // c = 4
                                        d = t;      // d = 3
a = b;      // a = 2
b = t;      // b = 3
t = s;      // t = UNDEFINED
                                        t = s;      // t = 1

Final values:
a = 2, b = 3, c = 4, d = 3, t = 1

Expected values:
a = 2, b = 1, c = 4, d = 3, t = UNDEFINED
```

Reentrancy vs Thread-Safety (1)

- Re-entrant does not always mean thread-safe (as we saw)
  - But, for most sane implementations, it is thread-safe

- Are **thread-safe** functions reentrant?

Reentrancy vs Thread-Safety (2)

- Are **thread-safe** functions reentrant? Nope. Consider:

```
int f() {
    locktype_lock();
    // protected code
    locktype_unlock();
}
```

Remember: **Reentrant functions can be suspended in the middle of execution and called again before the previous execution completes**

So this obviously isn't reentrant, and it will deadlock

Interrupt handling is more for systems programming, so it may or may not come up again

Inlining

You may be familiar with inlining

- Instructs the compiler to just insert the function code in-place, instead of calling the function
- Therefore, no overhead of a function call!
- Compilers can also do better, context-sensitive operations it couldn't have before

No overhead... sounds like better performance... let's inline everything!

Inlining in C++

Implicit inlining (defining a function inside a class definition):

```
class P {
public:
    int get_x() const { return x; }
...
private:
    int x;
};
```

Explicit inlining:

```
inline max(const int& x, const int& y) {
    return x < y ? y : x;
}
```

The Other Side of Inlining

One big downside:

- Your program size is going to increase
- This is worse than you think
  - Less cache hits
  - More trips to memory
- Some inlines can grow very rapidly (C++ extended constructors)

Just from this your performance may go down easily

Compilers on Inlining

Also, inlining is merely a suggestion to compilers, they can ignore you, for example:

- Taking the function pointer of an "inline" function and using it
- Virtual functions (for C++)

will get you ignored quite fast

From a Usability Point-of-View

Debugging is more difficult (you can't set a breakpoint in a function that doesn't actually exist)

- Most compilers simply won't inline code with debugging symbols on
- Some do, but typically its more of a pain

Library design:

- If you change any inline function, any users of that library have to **recompile** their program if the library updates
- Avoided for non-inlined functions (executes the new function dynamically at runtime)

Summary

- Limit your inlining to trival things
  - Makes debugging easier and better usability
  - Won't slow down your program before you even start optimizing it
- Fine vs. Coarse-Grained locking tradeoffs
- Preventing deadlocks
- Difference between reentrant and thread-safe functions

Monday's Lecture

Here's the plan for Monday:

- Take up Assignment 1 and point out common mistakes and things to improve
- Discuss Assignment 2 (probably going to be useful)