ECE459: Programming for Performance Winter 2013

Lecture 14 — February 28, 2013

Patrick Lam *version 1*

1 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);
}
```

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 a thread is waiting for IO.
- 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.)

Reentrancy versus Thread-Safety

On a different note, we're going to discuss the distinction between reentrant functions and thread-safe functions. There is overlap, but these terms are actually different.

A function is *reentrant* if it can be suspended in the middle and re-entered, or called again, before the previous execution returns.

Reentrant does not always mean **thread-safe** (although it usually is). Recall: **thread-safe** is essentially "no data races".

The distinction is moot if the function only modifies local data, e.g. `sin()`. Those functions are both reentrant and thread-safe.

Example. Courtesy of Wikipedia (with modifications), here's a program (to the left) and its trace (to right):

```
int t;

void swap(int *x, int *y) {
    t = *x;
    *x = *y;
    // 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);
```

```
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
```

We can fix the example by storing the global variable in stack variable `s`, as follows.

```
int t;

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

    s = t;  // save global variable
    t = *x;
    *x = *y;
    // 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);
```

```
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
```

The obvious question: is the previous reentrant code also thread-safe? (This is more what we're concerned about in this course.)

Let's see. Consider two calls to the reentrant swap:

`swap(a, b)`, `swap(c, d)` with `a = 1, b = 2, c = 3, d = 4`.

```
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
```

To recap what we know so far: re-entrant does not always mean thread-safe. (But, for most sane implementations, reentrant is also thread-safe.)

But, are **thread-safe** functions reentrant? Nope! Consider:

```
int f() {
    lock();
    // protected code
    unlock();
}
```

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

`f()` obviously isn't reentrant. Plus, it will deadlock.

Interrupt handling is more for systems programming, so the topic of reentrancy may or may not come up again.

To sum up, here's the difference between reentrant and thread-safe functions:

Reentrancy.

- Has nothing to do with threads—assumes a **single thread**.
- Reentrant means the execution can context switch at any point in in a function, call the **same function**, and **complete** before returning to the original function call.
- Function's result does not depend on where the context switch happens.

Thread-safety.

- Result does not depend on any interleaving of threads from concurrency or parallelism.
- No unexpected results from multiple concurrent executions of the function.

If it helps, here's another definition of thread-safety.

> "A function whose effect, when called by two or more threads, is guaranteed to be as if the threads each executed the function one after another, in an undefined order, even if the actual execution is interleaved."

Good Example of an Exam Question. Consider the following function.

```
void swap(int *x, int *y) {
    int t;
    t = *x;
    *x = *y;
    *y = t;
}
```

- Is the above code thread-safe?
- Write some expected results for running two calls in parallel.
- Argue these expected results always hold, or show an example where they do not.

Good Programming Practices: Inlining

We have seen the notion of inlining:

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

OK, so inlining removes overhead. Sounds like better performance! Let's inline everything! There are two ways of 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. Or, we can be explicit:

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

The Other Side of Inlining. Inlining has one big downside:

- Your program size is going to increase.

This is worse than you think:

- Fewer cache hits.
- More trips to memory.

Some inlines can grow very rapidly (C++ extended constructors). Just from this your performance may go down easily.

Note also that inlining is merely a suggestion to compilers. They may ignore you. For example:

- taking the address of an "inline" function and using it; or
- virtual functions (in C++),

will get you ignored quite fast.

Implications of inlining. Inlining can make your life worse in two ways. First, debugging is more difficult (e.g. 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 it's more of a pain.

Second, it can be a problem for library design:

- If you change any inline function in your library, any users of that library have to **recompile** their program if the library updates. (Congratulations, you made a non-binary-compatible change!)

This would not be a problem for non-inlined functions—programs execute the new function dynamically at runtime.

Some Notes on Benchmarking

- Make sure your results are consistent (nothing else is running).
- Follow the 10 second guideline (60 second runs are no fun).

- Since we are assuming 100% parallel, the runtime should decrease by a factor of `physicalcores`.
- Results should be close to predicted, therefore our assumption holds (could estimate P in Amdahl's law and find it's 0.99).
- Overhead of threading (create, joining, mutex?) is insignificant for this program.
- Hyperthreading results were weird, slower the majority of the time.
- It's better to have a number of threads that match the number of virtual CPUs than an unbalanced number.
- If it's unbalanced, one thread will constantly be context switching between virtual CPUs.
- Worst case: 9 threads on 8 virtual CPUs. 8 threads complete, each doing a ninth of the work in parallel, last ninth of the work runs only on one CPU.

High-Level Language Performance Tweaks

So far, we've only seen C—we haven't seen anything complex, and C is low level, which is good for learning what's really going on.

Writing compact, readable code in C is hard, especially when #define macros and `void *` beckon.

C++11 has made major strides towards readability and efficiency—it provides light-weight abstractions. We'll look at a couple of examples.

Sorting. Our goal is simple: we'd like to sort a bunch of integers. In C, you would usually just use qsort from `stdlib.h`.

```
void qsort (void* base, size_t num, size_t size,
            int (*comparator) (const void*, const void*));
```

This is a fairly ugly definition (as usual, for generic C functions). How ugly is it? Let's look at a usage example.

```
#include <stdlib.h>

int compare(const void* a, const void* b)
{
    return (*((int*)a) - *((int*)b));
}

int main(int argc, char* argv[])
{
    int array[] = {4, 3, 5, 2, 1};
    qsort(array, 5, sizeof(int), compare);
}
```

This looks like a nightmare, and is more likely to have bugs than what we'll see next.

C++ has a sort with a much nicer interface[1]:

```
template <class RandomAccessIterator>
void sort (
    RandomAccessIterator first,
    RandomAccessIterator last
);

template <class RandomAccessIterator, class Compare>
void sort (
    RandomAccessIterator first,
    RandomAccessIterator last,
    Compare comp
);
```

It is, in fact, easier to use:

```
#include <vector>
#include <algorithm>

int main(int argc, char* argv[])
{
    std::vector<int> v = {4, 3, 5, 2, 1};
    std::sort(v.begin(), v.end());
}
```

Note: Your compare function can be a function or a functor. (Don't know what functors are? In C++, they're functions with state.) By default, `sort` uses `operator`< on the objects being sorted.

- Which is less error prone?
- Which is **faster**?

The second question is empirical. Let's see. We generate an array of 2 million ints and sort it (10 times, taking the average).

- qsort: 0.49 seconds
- C++ sort: 0.21 seconds

The C++ version is **twice** as fast. Why?

- The C version just operates on memory—it has no clue about the data.
- We're throwing away useful information about what's being sorted.
- A C function-pointer call prevents inlining of the compare function.

OK. What if we write our own sort in C, specialized for the data?

- Custom C sort: 0.29 seconds

[1]. . . well, nicer to use, after you get over templates.

Now the C++ version is still faster (but it's close). But, this is quickly going to become a maintainability nightmare.

- Would you rather read a custom sort or 1 line?
- What (who) do you trust more?

Lesson

Abstractions will not make your program slower.

They allow speedups and are much easier to maintain and read.

Vectors vs Lists

Consider two problems.

1. Generate **N** random integers and insert them into (sorted) sequence.

   Example: 3 4 2 1

   - 3
   - 3 4
   - 2 3 4
   - 1 2 3 4

2. Remove **N** elements one-at-a-time by going to a random position and removing the element.

   Example: 2 0 1 0

   - 1 2 4
   - 2 4
   - 2
   -

For which **N** is it better to use a list than a vector (or array)?

Complexity analysis. As good computer scientists, let's analyze the complexity.

Vector:

- Inserting
  - $O(\log n)$ for binary search
  - $O(n)$ for insertion (on average, move half the elements)
- Removing
  - $O(1)$ for accessing
  - $O(n)$ for deletion (on average, move half the elements)

List:

- Inserting
  - $O(n)$ for linear search
  - $O(1)$ for insertion
- Removing
  - $O(n)$ for accessing
  - $O(1)$ for deletion

Therefore, based on their complexity, lists should be better.

Reality. OK, here's what happens.

```
$ ./vector_vs_list 50000
Test 1
======
vector: insert 0.1s   remove 0.1s   total 0.2s
list:   insert 19.44s   remove 5.93s   total 25.37s
Test 2
======
vector: insert 0.11s   remove 0.11s   total 0.22s
list:   insert 19.7s   remove 5.93s   total 25.63s
Test 3
======
vector: insert 0.11s   remove 0.1s   total 0.21s
list:   insert 19.59s   remove 5.9s   total 25.49s
```

Vectors dominate lists, performance wise. Why?

- Binary search vs. linear search complexity dominates.
- Lists use far more memory. **On 64 bit machines:**
  - Vector: 4 bytes per element.
  - List: At least 20 bytes per element.

- Memory access is slow, and results arrive in blocks:
  - Lists' elements are all over memory, hence many cache misses.
  - A cache miss for a vector will bring a lot more usable data.

So, here are some tips for getting better performance.

- Don't store unnecessary data in your program.
- Keep your data as compact as possible.
- Access memory in a predictable manner.
- Use vectors instead of lists by default.
- Programming abstractly can save a lot of time.
- Often, telling the compiler more gives you better code.
- Data structures can be critical, sometimes more than complexity.
- **Low-level code != Efficient**.
- Think at a low level if you need to optimize anything.
- Readable code is good code—different hardware needs different optimizations.

Summary

In this lecture, we saw:

- Fine vs. Coarse-Grained locking tradeoffs.
- Ways to prevent deadlocks.
- Difference between reentrant and thread-safe functions.
- Limit your inlining to trivial functions:
  - makes debugging easier and improves usability;
  - won't slow down your program before you even start optimizing it.
- Tell the compiler high-level information but think low-level.