Lecture 14—Lock granularity; Reentrant vs. Thread-safe; Inlining; Benchmarking A2; High-Level Language Perf Tweaks

ECE 459: Programming for Performance

February 28, 2013

Previous Lecture

- Cache coherency;
- Implementing servers to handle many connections.

Oh, and a midterm.

Part I

Locking Granularity

Locking

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.

Locking: 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.

Locking: Contention

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.

Locking: Deadlocks

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).

Flashback: From Lecture 1

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. Oops! They both wait for each other (deadlock).

Key to Preventing 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(&ll);
}

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

This code will not deadlock: you can only get **l2** if you have **l1**.

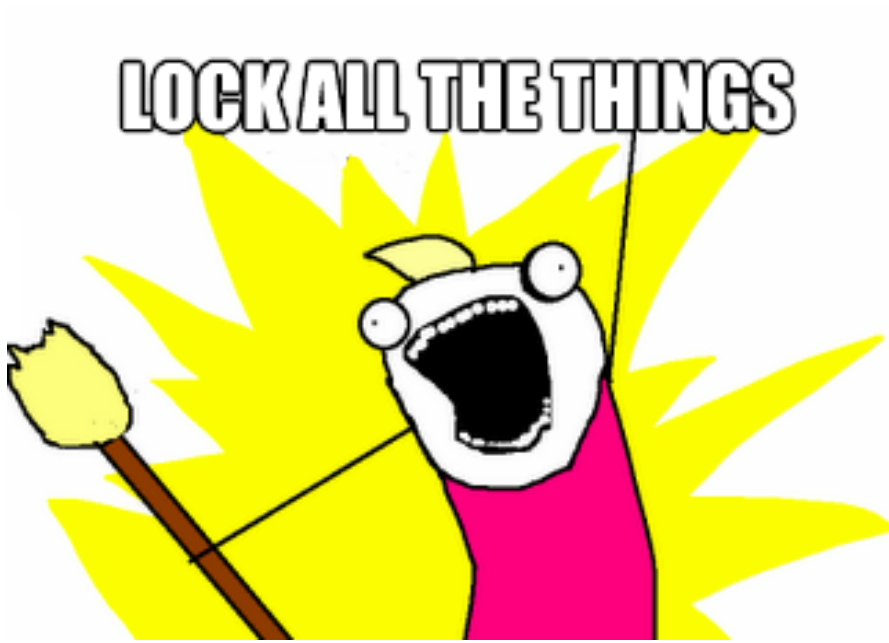
Preventing Deadlocks—Using `trylock`

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(&ll);
}
```

This code also won't 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:

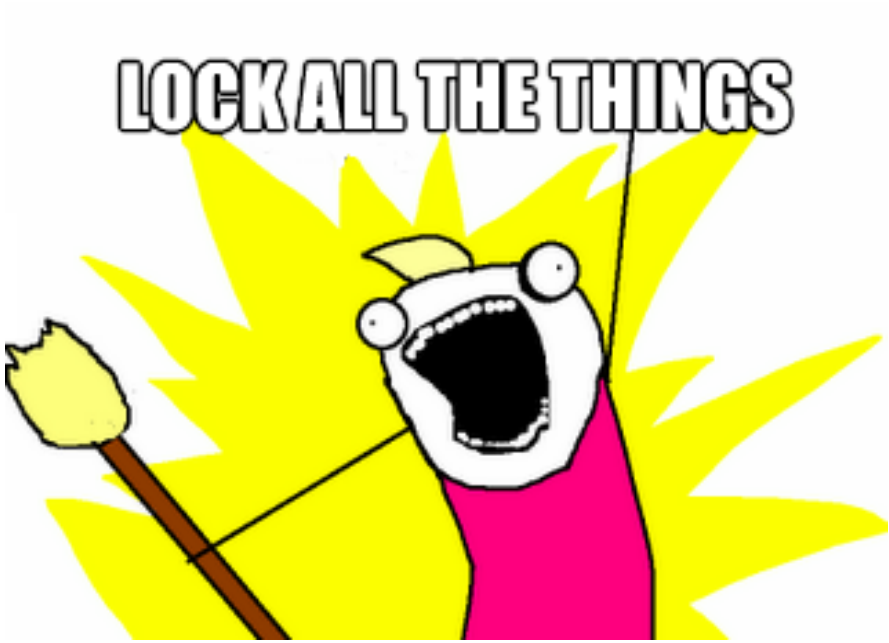
- Your parallel program can quickly become sequential.

Coarse-Grained Locking Example—Python GIL

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

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.
- Prone to deadlocks.
- Generally more error-prone (be sure you grab the right lock!)

Fine-Grained Locking Examples

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 $\rightarrow$ coarse-grained).

Can lock individual objects.

Part II

Reentrancy

Reentrancy

$\Rightarrow$ A function can be suspended in the middle and **re-entered** (called again) before the previous execution returns.

Does not always mean **thread-safe** (although it usually is).

- Recall: **thread-safe** is essentially "no data races".

Moot point if the function only modifies local data, e.g. `sin()`.

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 (a trace)

```
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 (a trace)

```
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 also thread-safe?
(This is more what we're concerned about in this course.)

Let's see:

```
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
}
```

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

Previous Example: thread-safety trace

```
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

Ok, but are **thread-safe** functions reentrant?

Reentrancy vs Thread-Safety (2)

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.

Summary of Reentrancy vs Thread-Safety

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.

Another Definition of Thread-Safe Functions

"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

```
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.

Part III

Good 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.

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:

- 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.

Compilers on Inlining

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.

From a Usability Point-of-View

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.

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. (non-binary-compatible change!)

Not a problem for non-inlined functions—programs execute the new function dynamically at runtime.

Notes on Benchmarking A2—Sequential and Parallel Physical Cores

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.

A2—Parallel Virtual CPUs vs Virtual CPUs + 1

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.

Part IV

High-Level Language Performance Tweaks

Introduction

So far, we've only seen C—we haven't seen anything complex.

C is low level, which is good for learning what's really going on.

Writing compact, readable code in C is hard.
Common C sights:

- **#define macros**
- **void***

C++11 has made major strides towards readability and efficiency (it provides light-weight abstractions).

Goal

Sort a bunch of integers.

In **C**, usually use qsort from `stdlib.h`.

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

- A fairly ugly definition (as usual, for generic C functions)

How ugly? qsort usage

```
#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.

C++ sort

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
);
```

[1] ...nicer to use, after you get over templates (they're useful, I swear).

C++ sort Usage

```
#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.
By default, `sort` uses `operator<` on the objects being sorted.

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

Timing Various Sorts

[Shown: actual runtimes of `qsort` vs `sort`]

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 Sort

[Shown: actual runtimes of custom sort vs `sort`]

- The C++ version is still faster (although it's close).
- However, 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?

Abstractions will not make your program slower.

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

Lecture Fun

Let's throw Java in the mix and see what happens.

Vectors vs. Lists: Problem

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

- **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

[Shown: actual runtimes of vectors and lists]

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.

Performance Tips: Bullets

- 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.

Programming for Performance with the Compiler

- 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

- 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.