CSE 374: Programming Concepts and Tools

Eric Mullen
Spring 2017
Lecture 24: Concurrency
Administrivia

- Homework 6 due Tomorrow (midnight)
  - Late days: only if both partners have them to use
- Homework 7 out Friday
- Final on Wed of Finals week
  - When should review session be?
- Extra Office Hours Today 3pm (CSE 218)
Concurrency

• Computation where “multiple things happen at the same time” is inherently more complicated than “one at a time”
• Entirely new kinds of bugs!
• Two forms of concurrency:
  • time-slicing: only one thing actually happening at a time
  • parallelism: use more than one CPU at the same time
• No problem unless different computations need to communicate or use the same resources
Processes

• Multiple processes run “at once”
• Why? (Convenience, efficient use of resources, responsiveness, performance, etc…)
• No problem: address spaces separate
• They can communicate/share with files (and pipes)
• Things can go wrong, e.g. a race condition
  • echo “hi” > someFile
  • foo=`cat somefile`
• The O/S provides synchronization mechanisms to avoid this
The old story

- We said a running Java or C program had code, heap, global variables, a stack, and “where is execution right now” (program counter)
- C, Java support parallelism similarly (other languages can be different)
  - one pile of code, globals, heap
  - multiple “stack + program counter”s — called threads
  - threads are run or pre-empted by a scheduler
  - threads all share the same memory
- Various synchronization mechanisms control when threads run
  - “don’t run until I’m done with this”
Threads in C/Java

C: the POSIX Threads (pthreads) library
- \#include <pthread.h>
- pass -lpthread to gcc (when linking)
- pthread_create takes a function pointer and an argument for it, runs as a separate thread

Java: built into the language
- Subclass java.lang.Thread, and override the run method
- Create a Thread object and call its start method
- Any object can “be synchronized on” (later today)
Why?

- Convenient structure of code
  - failure isolation
  - fairness
- Performance
  - take advantage of multiple cores
  - hide I/O latency
Simple Synchronization

- If one thread did nothing of interest to any other thread, why bother running?
- Threads must communicate and coordinate
  - Use results from other threads, and coordinate access to shared resources
- Simplest ways to not mess each other up:
  - Don’t access same memory (complete isolation)
  - Don’t write to shared memory (write isolation)
- Next simplest: One thread doesn’t run until/unless another is done
Using Parallel Threads

- Common pattern for expensive computations
  - split the work up, give each piece to a thread (*fork*)
  - wait until all are done, then combine answers (*join*)
- To avoid bottlenecks, each thread should have about the same amount of work
  - Performance will always be less than perfect speedup
Less Structure

• Often you have a bunch of threads running at once and they might need the same mutable (writable) memory at the same time but probably not

• Want to be correct, but not sacrifice parallelism

• Example: bunch of threads processing bank transactions
  • withdraw, deposit, transfer, currentBalance, etc…
  • unlikely two will overlap, but there’s a chance
  • very important that answer is correct when they overlap
The issue

struct Acct {int balance; /*etc...*/ };  
int withdraw(struct Acct* a, int amt) {
    if (a->balance < amt)
        return FAIL;
    a->balance -= amt;
    return SUCCESS;
}

• This code is correct in a sequential program
• It may have a race condition in a concurrent program, allowing for a negative balance
• Discovering this bug with testing is very hard
atomic

- Program construct which indicates “all at once”
- Everything in an atomic block must appear to any other threads as having not yet started, or having already finished

```c
int withdraw(struct Acct* a, int amt) {
    atomic {
        if (a->balance < amt)
            return FAIL;
        a->balance -= amt;
    }
    return SUCCESS;
}
```

- Don’t just wrap your whole program in an atomic, then just like running sequentially
Critical Section

- The part of your program that would have races if not synchronized properly is the *critical section*

- You must make it the right size! (this is hard)
  - Too big: program runs sequentially, no parallelism
  - Too small: program has races, is incorrect
So far

- Shared memory concurrency where multiple threads might access the same mutable data at the same time is tricky
- It’s worse because atomic isn’t in C or Java
- Instead, programmers must use locks (or other mechanisms) which are lower level and harder to use
  - Misuse of locks will violate the “all at once” property
  - Can also lead to bugs we haven’t seen yet
Lock Basics

- A lock is *acquired* and *released* by a thread
- At most one thread “holds it” at any moment
- Acquiring it “blocks” until the current holder releases it
  - Many threads might be waiting, will only go to one at a time
- Lock implementor avoids race conditions
- To keep two things from happening at the same time, surround them with a lock-acquire/lock-release
Locks in C/Java

C: Need to initialize and destroy mutexes (i.e. locks)
  • An initialized (pointer to a) mutex can be locked or unlocked via library function calls

Java: A synchronized statement is an acquire/release
  • Any object can serve as a lock
  • Lock is released on any control transfer out of the synchronized block
  • “Synchronized methods” just save keystrokes
Choosing how to lock

• Now we know what locks are (how to make them, what acquire/release means), but programming with them correctly and efficiently is difficult.

• As before, if critical sections aren’t the right size, it’s not great.

• Now, if two “synchronized blocks” grab different locks, they can both run at the same time (even if they access the same memory).

• Also, a lock-acquire blocks until a lock is available, and only the current holder can release.
Deadlock

- A cycle of threads waiting on locks means none will ever run again

- Avoidance: All code acquires locks in the same order (very hard to do). Ad hoc: Don’t hold onto locks too long or while calling into unknown code
Best Practices

- Any one of the following will avoid races
  - Keep data thread local
  - Keep data read-only
  - Use locks consistently (lock A corresponds to some data, all accesses to that data are locked with that lock)
  - Use partial order of locks to avoid deadlock (simpler: only ever have one lock at a time)
- These are tough, but what you have to do
- One lock for everything satisfies above, but is inefficient
Locking Granularity

• How much data should one lock guard?
  • In Java the suggested answer is obvious: one object
  • In C you get to pick
  
  • Coarser granularity: less likely to deadlock, can improve performance (lock acquire is expensive)
  
  • Finer granularity: allows for more parallelism, thus can improve performance
Bank Accounts

• If we gave each account its own lock, how would we write our transfer method?

• Need to lock both accounts, make sure both are updated atomically, want to make sure there’s no deadlock
It’s actually a lot worse…

- You would naturally assume that what we just discussed is as bad as it gets
- Turns out that on the trip from C code to executable instructions, compilers will re-order memory accesses. Thread on right might have assertion failure.

**initially: data = 0, flag = false**

```c
data = 42;
flag = true;
while (!flag) {} assert(data==42);
```

- To disallow reordering, use lock acquire (compiler will not reorder across lock acquire), or use volatile (for experts only, not this class)
Conclusion

- Threads make a lot of otherwise-correct approaches incorrect
  - writing “thread-safe” libraries is hard
  - use an expert implementation if you can: e.g. Java’s ConcurrentHashMap and others
- Threads are increasingly important for efficient use of today's computers
- Locks with shared memory is just one common approach