CSE 303:
Concepts and Tools for Software Development

Dan Grossman

Spring 2007

Lecture 22— Shared-Memory Concurrency

Where are we

- Saw what *threads* are and why we want them
  - Each has own stack and program-counter
  - With *shared-memory*, just one globals and heap
  - Can help with code-structure and performance
  - Time-slicing vs. parallel computing
- Saw how to create *threads* in C and Java
- Today: Coordinating threads that use the same memory
  - Threads make programming much more difficult

Basics

C: The POSIX Threads (pthreads) *library*

- `#include <pthread.h>`
- Link with `-lpthread`
- `pthread_create` takes a function pointer and an argument for it; runs it as a separate thread.
- Many types, functions, and macros for threads, locks, etc.

Java: Built into the language

- Subclass `java.lang.Thread` overriding `run`
- Create a Thread object and call its `start` method
- Any object can “be synchronized on” (later)

See code examples...

Simple synchronization

If one thread did nothing of interest to any other thread, why is it running?

So threads have to *communicate* and *coordinate*.

- Use each others' results; avoid messing up each other's computation.

Simplest two ways not to mess each other up (don't underestimate!):

1. Do not access the same memory.
2. Do not mutate shared memory.

Next simplest: One thread does not run until/unless another thread is done

- Called a *join*
- See examples in C and Java for summing an array

About the examples

- A common pattern for expensive computations:
  - Split the work
  - Join on all the helper threads
  - Called fork-join parallelism
- To avoid bottlenecks, each thread should have about the same amount of work (load-balancing)
  - Our example is *not* particularly good at this (why?)
  - Performance depends on number of CPUs available and will typically be less than "perfect speedup"
- C vs. Java (specific to threads)
  - Java takes an OO approach (shared data via fields of Thread)
  - Java separates creating the Thread-object and creating the running-thread

Less structure

Often you have a bunch of threads running at once and they *might* need the same mutable memory at the same time but *probably not*.

Want to be *correct* without sacrificing parallelism.

Example: A bunch of threads processing bank transactions:

- withdraw, deposit, transfer, currentBalance, ...
- chance of two threads accessing the same account at the same time very low, but not zero.
- want *mutual exclusion* (a way to keep each other out of the way when there is *contention*)

Another example: Parallel search through an arbitrary graph

The Issue

```
struct Acct { int balance; /* ... other fields ... */ };

int withdraw(struct Acct * a, int amt) {
  if(a->balance < amt) return 1; // 1==failure
    a->balance -= amt;
  return 0; // 0==success
}
```

This code is correct in a sequential program.

It may have a *race condition* in a concurrent program, allowing a negative balance.

Discovering this bug is very hard with testing since the interleaving has to be “just wrong”.

atomic

Programmers must indicate what must *appear to happen all-at-once*.

```
int withdraw(struct Acct * a, int amt) {
  atomic {
    if(a->balance < amt) return 1; // 1==failure
    a->balance -= amt;
  }
  return 0; // 0==success
}
```

Reasons not to do “too much” in an atomic:

- Correctness: If another threads needs an intermediate result to compute something you need, must “expose” it.
- Performance: Parallel threads must access disjoint memory
  - Actually read/read conflicts can happen in parallel

Getting it "just right"

This code is probably wrong because critical sections too small:

```
atomic { if(a->balance < amt) return 1; }
atomic { a->balance -= amt; }
```

This code (skeleton) is probably wrong because critical section too big:

- Assume other guy does not compute until the data is set.

```
atomic {
  data_for_other_guy = 42; // set some global
  ans = wait_for_other_guy_to_compute();
  return ans;
}
```

So far

Shared-memory concurrency where multiple threads might access the same mutable data at the same time is tricky

- Must get size of critical sections just right

It's worse because

- `atomic` does not yet exist in languages like C and Java
- (Your instructor is working on that.)

Instead programmers must use *locks* (a.k.a. mutexes) or other mechanisms, usually to get the behavior of critical sections

- But misuse of locks will violate the "all-at-once" property
- Or lead to other 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 holder releases it and the blocked thread acquires it
  - Many threads might be waiting; one will “win”.
  - The lock-implementor avoids race conditions on the lock-acquire
- So to keep two things from happening at the same time, surround them with the same lock-acquire/lock-release

Locks in C/Java

(See example code.)

C: Need to *initialize* and *destroy* mutexes (a synonym for locks).

- The joys of C

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 block (return, break, exception, ...)
- "Synchronized methods" just save keystrokes.

Choosing how to lock

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

- As before, if critical sections are too small we have races and too big we may not communicate enough to get our work done efficiently.
- But now, if two "synchronized blocks" grab different locks, they can be interleaved even if they access the same memory
  - A "data race"
- Also, a lock-acquire blocks until a lock is available and only the current-holder can release it.
  - Can have "deadlock" ...

Deadlock

```
Object a;
Object b;
void m1() {            void m2() {
  synchronized a {       synchronized b {
  synchronized b {       synchronized a {
     ...                      ...
 }}                    }}
}
```

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.

Rules of Thumb

Any one of the following are *sufficient* for avoiding races:

- Keep data *thread-local* (an object is *reachable*, or at least only accessed by, one thread).
- Keep data *read-only* (do not assign to object fields after an object's constructor)
- Use locks consistently (all accesses to an object are made while holding a particular lock)
- Use a partial-order to avoid deadlock (over-simple example: do not hold multiple locks at once?)

These are tough invariants to get right, but that's the price of multithreaded programming today.

But... one way to do all the above is to have "one lock for all shared data" and that is inefficient...

False sharing

“False sharing” refers to not allowing separate things to happen in parallel.

Example:

```
synchronized x {        synchronized x {
  ++y;                    ++z;
}                       }
```

More realistic example: one lock for all bank accounts rather than one for each account

On the other hand, acquiring/releasing locks is not so cheap, so “locking more with the same lock” can improve performance.

This is the “locking granularity” question

- Coarser vs. finer granularity

Very challenging situation

My favorite example for ridiculing locks:

If each bank account has its own lock, how do you write a “transfer” method such that no other thread can see the “wrong total balance”?

```
// race (not data race)
void xfer(int a,Acct other){
 synchronized(this) {
   balance += a;
   other.balance -= a;
 }
}
```

```
// potential deadlock
void xfer(int a,Acct other){
 synchronized(this) {
 synchronized(other) {
   balance += a;
   other.balance -= a;
 }}}
```

The problem is there is no relative order among accounts, so “inverse transfers” could deadlock

A final gotcha

You would naturally assume that all memory accesses happen in "some consistent order" that is "determined by the code".

Unfortunately, compilers and chips are often allowed to cheat (reorder)! The assertion in the right thread may fail!

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

To disallow reordering the programmer must:

- Use lock acquires (no reordering across them), or
- Declare `flag` to be `volatile` (for experts, not us)

Conclusion

Threads make a lot of otherwise-correct approaches incorrect.

- Writing “thread-safe” libraries can be excruciating.
- Use an expert implementation, e.g., Java’s ConcurrentHashMap?

But they are increasingly important for efficient use of computing resources (“the multicore revolution”).

Locks and shared-memory are (just) one common approach.

Learn about other useful synchronization mechanisms (e.g., condition variables) in CSE451.