CSE 303:
Concepts and Tools for Software Development

Hal Perkins

Autumn 2008

Lecture 23— Concurrency Part II: Locks

Where are we

Done:

- Basics of shared-memory multithreading
- Fork-join parallelism
- Critical sections via `atomic`

Doing:

- Critical sections via careful use of locks (a.k.a. mutexes)
- Pitfalls of using locks
- Other concurrency gotchas

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

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; if 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.

Recovery: detect deadlocks, kill off and rerun one of the processes (databases)

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

A 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.