CSE 303: Concepts and Tools for Software Development

Dan Grossman

Spring 2005

Lecture 28— Concurrency, Threads

You have grading to do

I am going to distribute course evaluation forms so you may rate the quality of this course. Your participation is voluntary, and you may omit specific items if you wish. To ensure confidentiality, do not write your name on the forms. There is a possibility your handwriting on the yellow written comment sheet will be recognizable; however, I will not see the results of this evaluation until after the quarter is over and you have received your grades. Please be sure to use a No. 2 PENCIL ONLY on the scannable form.

I have chosen *(name)* to distribute and collect the forms. When you are finished, he/she will collect the forms, put them into an envelope and mail them to the Office of Educational Assessment. If there are no questions, I will leave the room and not return until all the questionnaires have been finished and collected. Thank you for your participation.

I'll come back at 2:47. Please remember the back.

Our Model

So far, a process (a running program) has:

- a stack
- a heap
- code
- global variables

Other processes have a separate *address space*. The O/S takes turns running processes on one or more processors.

Interprocess communication happens via the file system, pipes, and things we don't know about.

Inter-process races

Forgetting about other processes can lead to programming mistakes:

```
echo "hi" > someFile
set foo = `cat someFile`
# assume foo holds the string hi??
```

A *race condition* is when this might occur.

Processes sharing resources must *synchronize*; no time today to show you how.

But enough about processes; we'll focus on *intra-process threads* instead and how you use *locks* in Java.

“Lightweight” Threads

One process can have multiple threads!

Each thread has its own stack.

A scheduler runs threads one-or-more at a time.

The difference from multiple processes is the threads *share an address space* – same heap, same globals.

“Lightweight” because it’s easier for threads to communicate (just read/write to shared data).

But easier to communicate means easier to mess each other up.

(Also there are tough implementation issues about where to put multiple stacks.)

Shared Memory

Now races can happen if *two threads could access the same memory at the same time, and at least one access is a write*.

```
class A { String s; }
class C {
private A a;
void m1() {
  if(a != null) // "dangerous" race
    a.s = "hi";
}
void m2() { a = null; }
...
}
```

If you naively try to code away races, you will just add other races!!!

Concurrency primitives

Different languages/libraries for multithreading provide different features, but here are the basics you can expect these days:

- A way to create a new thread
  - See the `run` method of Java's `Thread` class.
- Locks (a way to acquire and release them).
  - A lock is *available* or *held by a thread*.
  - *Acquiring* a lock makes the acquiring thread hold it, but the acquisition *blocks* (does not continue!) until the lock is available.
  - *Releasing* a lock makes the lock available.
  - Advanced note: Java locks are *reentrant*: reacquisition doesn't block, instead increments a hidden counter that release decrements...

Locks in Java

Java makes every object a lock and combines acquire/release into one language construct:

```
syncrhonized (e) { s }
```

- Evaluate `e` to an object.
- "Acquire" the object (blocking until available).
- Execute `s`.
- Release the lock. The implementation of locks ensures no races on acquiring and releasing.

Fixing our example

If a `C` object might have `m1` and `m2` called simultaneously, then *both* must *guard* their access to a with the *same* lock.

```
class C {
private A a;
void m1() {
  synchronized (this) {
  if(a != null) // "dangerous" race
    a.s = "hi";
  }
}
void m2() { synchronized (this) { a = null; } }
}
```

Note: There is more convenient syntax for this.

Note: What if a is public and/or there are subclasses.

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)

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

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.

Summary

Multithreaded programming is harder:

- there are multiple stacks in one address space
- there are potential races and deadlocks

Locks are a useful concept: only one thread holds a lock at a time. There are other useful concepts; see CSE451 or come talk to me.

Why have threads?

- Performance
- Structure of certain code (e.g., event-handling)
- Robustness of certain code (e.g., thread-failure $\neq$ program-failure)

Example you have seen but which mostly hides threads: Java EventListeners