# CSE 303: Concepts and Tools for Software Development

Dan Grossman

Winter 2006

Lecture 22— Linking Wrap-up; Threads, concurrency

# Archives

An archive is the "`.o` equivalent of a `.jar` file" (though history is the other way around).

Create with `ar` program (lots of features, but fundamentally take `.o` files and put them in, but *order matters*).

The semantics of passing `ld` an argument like `-lfoo` is complicated and often not what you want:

- Look for what: file `libfoo.a` (ignoring shared libraries for now), when: at link-time, where: defaults, environment variables (`LIBPATH` ?) and the `-L` flags (analogous to `-I`).
- Go through the `.o` files in `libfoo.a` *in order*.
  - If a `.o` defines a *needed reference*, include the `.o`.
  - Including a `.o` may add more needed references.
  - Continue.

# The rules

A call to `ld` (or `gcc` for linking) has `.o` files and `-lfoo` options in left-to-right order.

- State: "Set of needed functions not defined" initially empty.
- Action for `.o` file:
  - Include code in result
  - Remove from set any functions defined
  - Add to set any functions used and not yet defined
- Action for `.a` file: For each `.o` in order
  - If it defines one or more functions in set, do all 3 things we do for a `.o` file.
  - Else do nothing.
- At end, if set is empty create executable, else error.

# Library gotchas

1. Position of `-lfoo` on command-line matters
   - Only resolves references for "things to the left"
   - So `-lfoo` *typically* put "on the right"
2. Cycles
   - If two `.o` files in a `.a` need other other, you'll have to link the library in (at least) twice!
   - If two `.a` files need each other, you might do `-lfoo -lbar -lfoo -lbar -lfoo ...`
3. If you include `math.h`, then you'll need `-lm`.

# Another gotcha

4. No repeated function names
   - 2 `.o` files in an executable can't have (public) functions of the same name.
   - Can get burned by library functions you do not know exist, but only if you need another function from the same `.o` file. (Solution: 1 public function per file?!)

# Beyond static linking

Static linking has disadvantages:

- More disk space (copy library portions for every application)
- More memory when programs are running (what if the O/S could have different processes magically share code).

So we can *link later*:

- Shared libraries (link in when program starts executing). Saves disk space. O/S can share actual memory behind your back (if/because code is immutable).
- Dynamically linked/loaded libraries. Even later (while program is running). Devil is in the details.

"DLL hell" – if the version of a library on a machine is not the one the program was tested with...

# Summary

Things like "standard libraries" "header files" "linkers" etc. are not magic.

But since you rarely need fine-grained control, you easily forget how to control typically-implicit things. (You don't need to know any of this until you need to. :) )

There's a huge difference between source code and compiled code (a header file and an archive are quite different).

The linker includes files from archives using strange rules.

# Our Old 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
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:

```
synchronized (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.

# Creating threads

Java separates creating a thread (an object of type `Thread`) from starting it (calling its `run` method).

- So subclass `Thread` and override `run` method.
- Share data via arguments passed to constructor.

C the language does not have threads, but there are libraries, especially `pthreads`, which provide locks and thread-creation.

- `pthread_create` takes a function-pointer and an argument to pass to it.
- Share data via the argument (like we did when studying function pointers).

# 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 (multicore is coming!)
- Structure of certain code (e.g., event-handling)
- Robustness of certain code (e.g., thread-failure $\neq$ program-failure)

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