CSE 303:
Concepts and Tools for Software Development

Dan Grossman

Spring 2007

Lecture 21— Linking Wrap-Up; Concurrency Part 1

Where are we

- Saw Java's "very late" class-loading
- In the middle of ld's static-linking
  - Need to learn how archives (`.a` files) work
- A bit on shared-libraries and dynamic-linking

Then concurrency:

- Multiple threads of execution (call-stacks) at once!
  - Why, how, what goes wrong, how to control it

Linking

If a C file uses but does not define a function (or global variable) `foo`, then the `.o` has "unresolved references". *Declarations don't count; only definitions.*

The linker takes multiple `.o` files and "patches them" to include the references. (It literally *moves code* and *changes instructions* like function calls.)

An executable must have no unresolved references (you have seen this error message).

What: Definitions of functions/variables

When: The linker creates an executable

Where: Other `.o` files on the command-line (and much more...)

More about where

The linker and O/S don't know anything about `main` or the C library.

That's why `gcc` "secretly" links in other things.

We can do it ourselves, but we would need to know a lot about how the C library is organized. Get `gcc` to tell us:

- `gcc -v -static hello.c`
- Should be largely understandable soon.
- `-static` (stick with the simple "get all the code you need into `a.out` story)
- the secret `*.o` files: (they do the stuff before `main` gets called, which is why `gcc` gives errors about `main` not being defined).
- `-lc`: complicated story about finding the *library* (a.k.a. "archive") `libc.a` and including any *files* that provide still-unresolved references.

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 ...`
   - (There are command-line options to do this for you, but not the default.)
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.

Concurrency

Computation where "multiple things happen at the same time" is inherently more complicated than *sequential* computation.

- Entirely new kinds of bugs and obligations

Two forms of concurrency:

- *time-slicing*: only one computation at a time but *pre-empt* to provide *responsiveness* or *mask I/O latency*.
- *true parallelism*: more than one CPU (e.g., the lab machines have two, the attu machines have 4, ...)

No problem unless the different computations need to *communicate* or use the same *resources*.

Example: Processes

The O/S runs multiple processes "at once".

Why? (Convenience, efficient use of resources, performance)

No problem: keep their address-spaces separate.

But they do communicate/share via files (and pipes).

Things can go wrong, e.g., a *race condition*:

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

The O/S provides *synchronization mechanisms* to avoid this

- See CSE451; we will focus on *intraprocess* concurrency.

The Old Story

We said a running Java or C program had code, a heap, global variables, a stack, and "what is executing right now" (in assembly, a *program counter*).

C, Java support parallelism similarly (other languages can be different):

- One pile of code, global variables, and heap.
- Multiple "stack + program counter"s — called *threads*
- Threads can be *pre-empted* whenever by a *scheduler*
- Threads can communicate (or mess each other up) via *shared memory*.
- Various *synchronization mechanisms* control what *thread interleavings* are possible.
  - "Do not do your thing until I am done with my thing"

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...

Why do this?

- Convenient structure of code
  - Example: 2 threads using information computed by the other
  - Example: Failure-isolation – each "file request" in its own thread so if a problem just "kill that request".
  - Example: Fairness – one slow computation only takes some of the CPU time without your own complicated timer code. Avoids *starvation*.
- Performance
  - Run other threads while one is reading/writing to disk (or other slow thing that can happen in parallel)
  - Use more than one CPU at the same time
    - The way computers will get faster over the next 10 years
    - So no parallelism means no faster.