Slide 1

Administrivia

- Via e-mail, except . . .

Slide 2

Why the Delay

- I was all set to refresh my notes on assembly and linking when one of you asked a question I couldn't really answer, and I figured I should try to improve my own understanding, and . . . .
- Hours of Web-searching didn't help. Finally I posted a query to the SIGCSE mailing list (SIGCSE being ACM's special-interest group for CS educators). Where the moderators are overwhelmed, so a delay . . .
- Eventually over a dozen replies, some *very* helpful. After some back-and-forth, finally I think I know enough . . .

  Also it sounds like I'm not alone in thinking section 2.12 has some problems. I think the authors are sound on concepts but not so clear about details.

  Worth noting that many were using other simulators — including two who had written their own! (Side project that could get out of hand?)

Slide 3

From Source to Execution — Big Picture Revisited

- Goal is to be able to translate programs written in a HLL or assembler language into something that the operating system can load into memory and run.
- Usually want this to be done in a way that supports separate compilation/assembly of source code files, possibly in different languages. (That sort of implies support for function libraries too, since a "library" basically consists of previously-compiled code.)
- A lot of the software conventions we've looked at — how procedures are called, memory use, etc. — exist to make this work.

Slide 4

Semi-Sidebar: Compilers Revisited

- In principle compilers all generate assembly-language code that follows these conventions, so it should be possible to call a function in one language from another language — assuming both compile to object code.
- In practice some details can get messy. Examples:

  C lays out 2D arrays in "row-major" order (by rows), Fortran in column-major order.

  Some language support overloading of functions. How to implement that might involving having a different name (think MIPS label) for each version (e.g., "name mangling" in C++ — Wikipedia article seems good).

  Usually, though, calling one language from another can be made to work.

Slide 5

Assembling Revisited

- Job of the assembler is to produce "object code". Details vary among platforms ("platform" here means combination of architecture and operating system — think ABI as defined in Chapter 1).
- Keeping in mind the big picture, object code needs to contain:
  - Machine language for instructions, typically collected into "code segment", a.k.a. "text segment".
  - Binary representation of any variables (`.word`, `.asciiz`, etc., in MIPS), typically collected into "data segment".
  - Something that will make it possible for code in one object file to reference a global symbol (procedure or data) in another, and to make absolute addresses work given that they aren't known at assembly time.

Slide 6

Assembling, Continued

- Even without the complication of referencing a label in another object file, though . . .
- You know that MIPS assembly language has a notion of labels that let you branch or jump to another place in the code, or load the address of a variable. Some are position-independent (PC-relative addressing), but some are absolute addresses and thus depend on where in memory program is loaded. How can that work?

Slide 7

Assemblers

- Most are two-pass (like most compilers — C a notable exception!). First pass builds table of symbols; second pass uses that to assemble.
- Start by establishing starting addresses for code and data segments.
- First pass just goes through the code, adding entries to symbol table and keeping track of "next" address.

  (*NOTE* that labels themselves occupy no space, but pseudoinstructions might expand to multiple real instructions.)

  Also make a note of any symbols declared as "global".
- Second pass uses table to generate code and data, plus . . .

Slide 8

Assemblers, Continued

- Note that if separate compilation is going to work, we need more information for next step ("linking" to combine object files). What do we need? Two things:

  "Relocation information" — which instructions use absolute addresses and what label they reference.

  Symbol table — global labels and unresolved references.
- Output all of that; format is part of platform's ABI.
- (A bit more about this in the section on linking.)

Slide 9

Linkers

- Job of linker is combine one or more object files into "executable file" — something the operating system can load into memory and execute.

  What does that imply . . .
- Instructions that aren't complete yet because they reference procedures or data in another object file need to be corrected.
- Instructions that aren't complete/correct because they use absolute addresses need to be corrected.

  Note that absolute addresses could still not be right, if it's not known at link time where in memory the program will be loaded.

Slide 10

Linkers, Continued

- So linker must do some things:
- Merge code segments, data segments.
- Merge symbol table into combined symbol table. (Error if unresolved symbols — which you may have seen in linking programs?)
- Use it to resolve unresolved references.
- Modify any absolute addresses, keeping track of the instructions that use them if they will need to be changed when the program is loaded.
- Output all of that (as an "executable"); format is part of platform's ABI.

Slide 11

Sidebar: Function Libraries

- Worth noting that link step is where code from system and other libraries is merged in.
- May explain why `gcc` sometimes fails with error messages starting `ld:` — linker isn't looking in all the needed libraries.
- In times past, code from library merged in directly, and sometimes that still makes sense. Alternately . . .

Slide 12

Sidebar Continued: Dynamic Linking

- Copying library code into executable may not be efficient — may result in pulling in code for unused procedures. Also, if the system supports concurrent execution of multiple threads/applications/etc., might be nice to allow them to share a single copy in memory of library code.
- "Dynamic linking" supports this, and has the side benefit(?) of allowing updates to library code without relinking all applications that use it. (Is this always a benefit?)
- Implementations have different names ("DLL" in Windows, "shared library" in UNIX/Linux). How it works is similar: At link time, link in "stub" routine that at runtime locates the desired code, loads it into memory (if necessary!) and patches references.

Slide 13

Loaders (Textbook)

- Nice explanation in Appendix A. Summary on p. 135.
- Operating system (loader) must:
- Read executable file to get sizes of text and data segments.
- "Create address space" big enough for text, data, and stack segments. (Details vary by O/S.)
- Initialize text and data segments from executable file.

  (Appendix doesn't mention this, but if the program isn't always loaded at the same address, somewhere in here any references to absolute addresses need to be modified.)

Slide 14

Loaders, Continued

- Set up registers — stack pointer, global pointer, etc.
- Push any arguments to program onto stack. (Think command-line arguments?)
- Jump to start-up code that copies arguments to registers and calls program's `main()`. On return, makes a system call to terminate program.
- Note in passing that code invoked by "system calls" is not part of the program; the `syscall` instruction jumps to code in the O/S's part of memory, in a way that allows it to execute with raised privileges.

Slide 15

From Source to Execution in SPIM

- SPIM combines assemble, link, and load steps:

  Assembles (in some way that lets it show source code lines).

  Loads resulting object code into memory. Can load more than one source file, but note that even so it doesn't really have to link; can just continue assembling.

- Always loads into memory at the same address, right after some code that . . .

  This is the start-up code just mentioned: Remember parameters to C's `main()`? `argc`, `argv`? and there's an optional third one, a list of environment variables. This sets that up. (I'm not sure where values come from for SPIM!)

- IMO, called `main` should start by pushing `$ra` onto stack, end by popping it off and using `jr` to return to SPIM code.

  (Many examples online don't do that. Not sure why not!)

Slide 16

Sidebar: Cross Assembling

- Very possible to generate, on one architecture, object code for another — "cross compiler/assembler".
- I've never tried it (on my long long list of things to do sometime), but one of my SIGCSE replies was from someone who had an oldish one, and they sent me some listings showing a dump of output. (I don't think I should post this but can show it here.) It's much as you might think!

Slide 17

Linking — Example

- Textbook presents an example starting on p. 132. Details are frankly confusing, especially for SPIM users — and trying to clarify is what sent me off on my days-long attempt to understand better!
- Part of the problem — discussion doesn't seem to mesh well with how SPIM does things . . .
- To be continued . . .

Slide 18

Minute Essay

- Do you feel like you understand the basic concepts here — the role of assemblers, linkers, and loaders — even if the details are a bit vague?