Slide 1

Administrivia

- Reminder: Homework 3 supposed to be due today. Okay to turn in through Friday.
- Homework 4 on the Web. Due in a week.
- No in-class lecture today; video lecture online at `echo360.org`. Following slides are for that.

Slide 2

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 library functions, since a “library” basically consists of previously-compiled code.)
- A lot of the software conventions we’ve looked — how procedures are called, memory use, etc. — exist to make this work.

Slide 3

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. (They both compile to object code, right?)
- In practice some details can get messy, e.g.:

  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 4

Assembling Revisited

- Job of the assembler is to produce “object code”. Details vary among platforms (“platform” here means combination of architecture and operating system).
- 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.

Slide 5

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. These are absolute addresses so depend on where in memory the program is loaded. How can that work?

Slide 6

Assemblers — How They (Could?) Work

- (I admit I haven't looked at actual code for an assembler, but the job seems straightforward.)
- First start by establishing starting addresses for code and data segments.
- As each instruction or data declaration is encountered, add to appropriate segment and increment "next" address (by 4 for instructions, by size of data for data).

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

  Also build "symbol table" of labels versus addresses and list of references to labels, and make note of any declared as "global".

Slide 7

Assemblers — How They (Could?) Work, Continued

- Resolve references to labels using symbol table and "patch" instructions accordingly. (Or maybe you make one pass through the code that only builds the symbol table and another that actually converts instructions. I think two passes are needed in any case.)

  Also keep track of any uses of absolute addresses, since these depend on where in memory the program gets loaded.

- When done, should have text and data segments, symbol table, and list of instructions that aren't right yet (because they reference external symbols or use absolute addresses).

  Output all of that; format is part of platform's ABI.

Slide 8

Linkers — How They (Could?) Work

- 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 9

Linkers — How They (Could?) Work, Contined

- So linker must do some things:
- Merge code segments, data segments.
- Merge tables of “global” symbols into combined symbol table.
- 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; format is part of platform’s ABI.

Slide 10

Sidebar: Dynamic Linking

- In earlier times linkers behaved as just described, linking in all needed library code. But this may not be efficient: It 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 11

Loaders — How They Work (Textbook)

- Nice explanation in Appendix A. Summary on p.129.
- 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 12

Loaders — How They Work, 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.

Slide 13

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, in which case it (in principle) does a link step to combine them.

- 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!)

- 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 14

Linking — Example

- Textbook presents an example starting on p. 127. Some details seem a bit murky, so let's work through it . . .
- One source of possible confusion is the handling of `lw` and `sw` instructions, which apparently . . .

Slide 15

`lw`, `sw` Revisited

- Strictly speaking, these instructions specify a memory address using a register and a fixed displacement.
- However, seems useful to be able to be able to load and store from address specified via label. Assembler could support that . . .

Slide 16

`lw`, `sw` With Labels — SPIM Way

- SPIM apparently defines pseudoinstructions for `lw` and `sw`. Based on some experiments . . .
- Just referencing a label, e.g.,

  ```
  lw  $t0, A
  ```

  assembles into an `lui` to put the top 16 bits of the address of SPIM's data segment into `$at` and then a `lw` that uses `$at` for the register and the offset to `A` as the displacement (calculated using symbol table).

  (Try it!)

  (To me it seems wrong not explicitly set the low 16 bits of `$at` as well, but observation says those are always zero.)

Slide 17

`lw`, `sw` With Labels — SPIM Way, Continued

- Referencing a label and a register, e.g.,

```
lw  $t0, A($t1)
```

assembles similarly, except that the `lui` to set `$at` to the address of the data segment is followed an `addu` (unsigned add) to add the contents of `$t1`. Note that if `$t1` is an index into an array of "words" this won't do what you might want.

Slide 18

`lw`, `sw` With Labels — Textbook's Way

- The textbook's example presupposes a different scheme:
- Register `$gp` points into the data segment, at an address that will allow addressing as much of the data segment as is possible using a 16-bit signed value (which is what displacement is in `lw` and `sw`).

  (I.e., `$gp` plus the most negative value you can get with 16 bits points to the start of the data segment.)
- `lw` and `sw` are assembled into code that uses `$gp`, e.g.,

```
lw  $t0, X
```

is assembled into

```
lw  $t0, D($gp)
```

where `D` is the displacement from `$gp` to `X`, calculated during linking.

Slide 19

Linking — Textbook Example Continued

- What happens at link time — reasonably well explained on p. 128, except for computing displacements for `lw` and `sw`:
- Goal is to come with up displacements, call them `DX` and `DY`, that when added to address in `$gp` (specified as `0x10008000`) gives addresses of `X` and `Y`.

  (We know what those are based on positioning data segments one after the other starting at `0x10000000`.)
- Some simple algebra says that, e.g.,

  `DX` is `0x10000000 - 0x10008000`

  Calculating and writing result in 16-bit two's complement form gives their result.

Slide 20

Homework 4 — Example of Assembling / Linking

- For the written problem in Homework 4, I ask you to do something along the same lines, but writing things in a way that I think makes more sense.
- Given two fairly meaningless source-code files, work through steps . . .
- (Files with code etc. linked from homework writeup.)

Slide 21

Minute Essay

- Questions?