CS61C – Machine Structures

Lecture 18 - Running a Program I
aka Compiling, Assembling, Linking, Loading

3/1/2006

John Wawrzynek

(www.cs.berkeley.edu/~johnw)

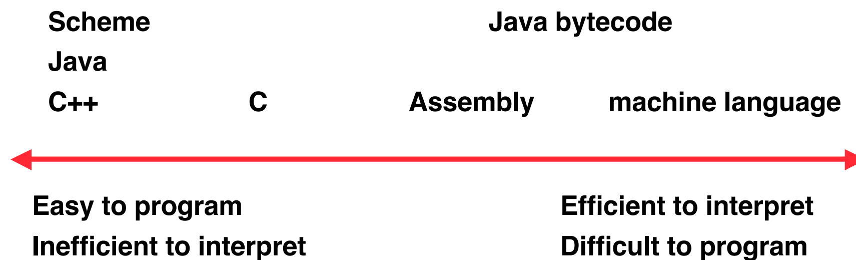
www-inst.eecs.berkeley.edu/~cs61c/

Overview

- Interpretation vs Translation
- Translating C Programs
  - Compiler
  - Assembler
  - Linker (next time)
  - Loader (next time)
- An Example (next time)

Language Execution Continuum

- An *Interpreter* is a program that executes other programs.



- Language *translation* gives us another option.
- In general, we interpret a high level language when efficiency is not critical and translate to a lower level language to improve performance

Interpretation vs Translation

- How do we run a program written in a source language?
- Interpreter: Directly executes a program in the source language
- Translator: Converts a program from the source language to an equivalent program in another language
- For example, consider a Scheme program `foo.scm`

Interpretation

Scheme program: `foo.scm`

Scheme Interpreter

- Scheme Interpreter is just a program that reads a scheme program and performs the functions of that scheme program.

Translation

Scheme program: `foo.scm`

Scheme Compiler

Executable(mach lang pgm): `a.out`

Hardware

- Scheme Compiler is a translator from Scheme to machine language.
- The processor is a hardware interpeter of machine language.

Interpretation

- Any good reason to interpret machine language in software?
- SPIM – useful for learning / debugging
- Apple Macintosh conversion
  - Switched from Motorola 680x0 instruction architecture to PowerPC.
  - Now similar issue with switch to x86.
  - Could require all programs to be re-translated from high level language
  - Instead, let executables contain old and/or new machine code, interpret old code in software if necessary

Interpretation vs. Translation? (1/2)

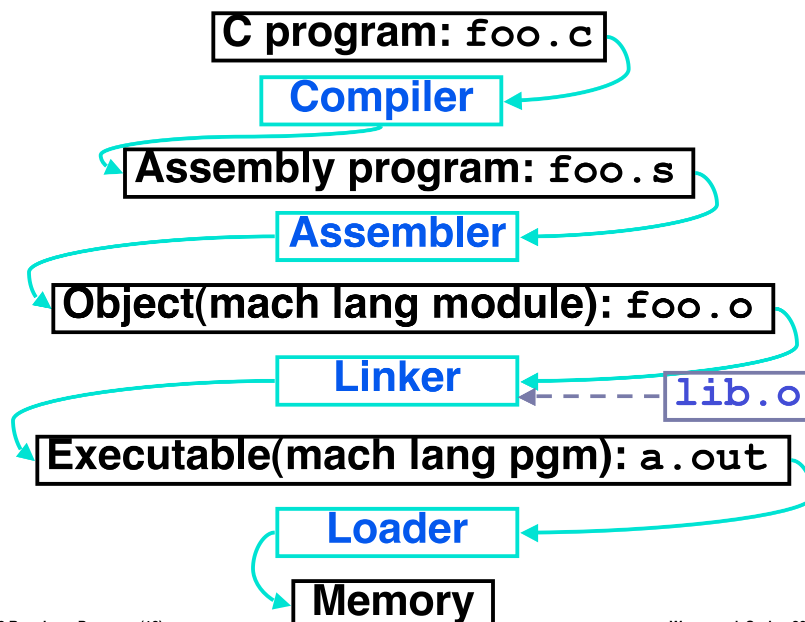
- Generally easier to write interpreter
- Interpreter closer to high-level, so can give better error messages (e.g., SPIM)
  - Translator reaction: add extra information to help debugging (line numbers, names)
- Interpreter slower (10x?) but code is smaller (1.5X to 2X?)
- Interpreter provides instruction set independence: run on any machine
  - Apple switched to PowerPC. Instead of retranslating all SW, let executables contain old and/or new machine code, interpret old code in software if necessary

Interpretation vs. Translation? (2/2)

- Translated/compiled code almost always more efficient and therefore higher performance:
  - Important for many applications, particularly operating systems.
- Translation/compilation helps "hide" the program "source" from the users:
  - One model for creating value in the marketplace (eg. MicroSoft keeps all their source code secret)
  - Alternative model, "open source", creates value by publishing the source code and fostering a community of developers.

Wawrzynek Spring 2006 © UCB

Steps to Starting a Program (translation)



Wawrzynek Spring 2006 © UCB

Compiler

- Input: High-Level Language Code (e.g., C, Java such as `foo.c`)
- Output: Assembly Language Code (e.g., `foo.s` for MIPS)
- Note: Output *may* contain pseudoinstructions
- Pseudoinstructions: instructions that assembler understands but not in machine (last lecture) For example:
- `mov $s1,$s2` ⇒ `or $s1,$s2,$zero`

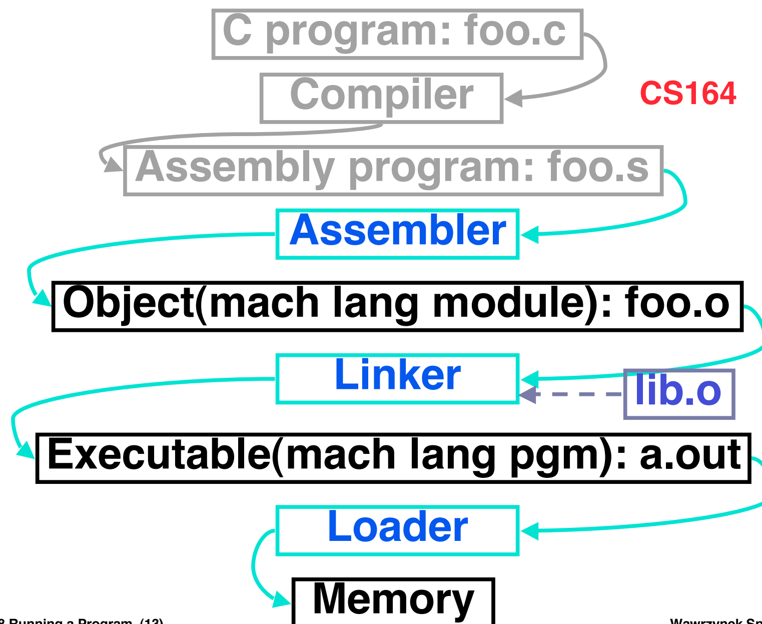
Wawrzynek Spring 2006 © UCB

Administrivia…

- Graded exams:
  - If you are not in the lecture hall today come to lecture Monday to pick it up.
- Exam Regrade requests must be in writing.
  - Attach a written cover-sheet with your exam, explaining your concern.
  - Turn-in in class, no later than Monday.
- Start working on project 3: MIPS instruction interpreter.
- Impending Grade Freeze!
  - HW 1-6, Project 1&2 grades must be settled before Spring break.
  - Use glookup to verify your grades.

Wawrzynek Spring 2006 © UCB

Where Are We Now?



Assembler

- Input: Assembly Language Code (e.g., `foo.s` for MIPS)
- Output: Object Code, information tables (e.g., `foo.o` for MIPS)
- Reads and Uses Directives
- Replace Pseudoinstructions
- Produce Machine Language
- Creates Object File

Assembler Directives (p. A-51 to A-53)

° Give directions to assembler, but do not produce machine instructions

`.text`: Subsequent items put in user text segment (machine code)

`.data`: Subsequent items put in user data segment (binary rep of data in source file)

`.globl sym`: declares `sym` global and can be referenced from other files

`.asciiz str`: Store the string `str` in memory and null-terminate it

`.word w1…wn`: Store the *n* 32-bit quantities in successive memory words

Pseudoinstruction Replacement

° Asm. treats convenient variations of machine language instructions as if real instructions

Pseudo:	Real:
`subu $sp,$sp,32`	`addiu $sp,$sp,-32`
`sd $a0, 32($sp)`	`sw $a0, 32($sp)` `sw $a1, 36($sp)`
`mul $t7,$t6,$t5`	`mul $t6,$t5` `mflo $t7`
`addu $t0,$t6,1`	`addiu $t0,$t6,1`
`ble $t0,100,loop`	`slti $at,$t0,101` `bne $at,$0,loop`
`la $a0, str`	`lui $at,left(str)` `ori $a0,$at,right(str)`

Producing Machine Language (1/3)

° Simple Case

- Arithmetic, Logical, Shifts, and so on.
- All necessary info is within the instruction already.

° What about Branches?

- PC-Relative
- So once pseudo-instructions are replaced by real ones, we know by how many instructions to branch.

° So these can be handled.

Wawrzynek Spring 2006 © UCB

Producing Machine Language (2/3)

"Forward Reference" problem

- Branch instructions can refer to labels that are "forward" in the program:

```
            or     $v0,$0,$0
L1:         slt    $t0,$0,$a1
            beq    $t0,$0,L2
            addi   $a1,$a1,-1
            j      L1
L2:         add     $t1,$a0,$a1
```

- Solved by taking 2 passes over the program.
  - First pass remembers position of labels
  - Second pass uses label positions to generate code

Wawrzynek Spring 2006 © UCB

Producing Machine Language (3/3)

- What about jumps (`j` and `jal`)?
  - Jumps require absolute address.
  - So, forward or not, still can't generate machine instruction without knowing the position of instructions in memory.
- What about references to data?
  - `la` gets broken up into `lui` and `ori`
  - These will require the full 32-bit address of the data.
- These can't be determined yet, so we create two tables…

Wawrzynek Spring 2006 © UCB

Symbol Table

- List of "items" in this file that may be used by other files.
- What are they?
  - Labels: function calling
  - Data: anything in the `.data` section; variables which may be accessed across files

Wawrzynek Spring 2006 © UCB

Relocation Table

- List of “items” for which this file needs the address.
- What are they?
  - Any label jumped to: `j` or `jal`
    - internal
    - external (including lib files)
  - Any piece of data
    - such as the `la` instruction

Wawrzynek Spring 2006 © UCB

Object File Format

- object file header: size and position of the other pieces of the object file
- text segment: the machine code
- data segment: binary representation of the data in the source file
- relocation information: identifies lines of code that need to be “handled”
- symbol table: list of this file’s labels and data that can be referenced
- debugging information
- A standard format is ELF (except MS)

http://www.skyfree.org/linux/references/ELF_Format.pdf

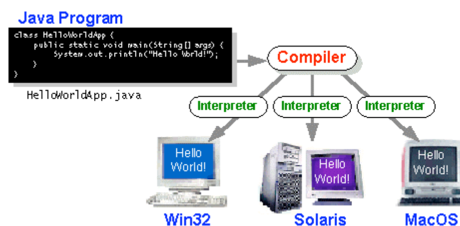
Wawrzynek Spring 2006 © UCB

Quiz

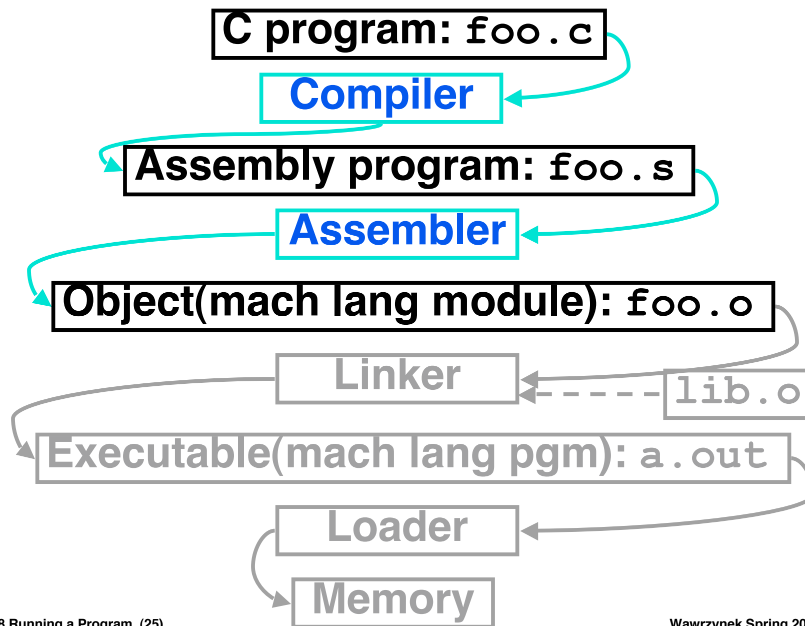
1. T/F. Assembler knows where a module's data & instructions are in relation to other modules.
2. T/F. Assembler will ignore the instruction `Loop:nop` because it does nothing.
3. Java systems uses a) a translator, b) a interpreter, or c) both.

Quiz Answer

1. Assembler only sees one compiled program at a time, that's why it has to make a symbol & relocation table. It's the job of the linker to link them all together…F!
2. Nop might be important to the programmer and therefore the assembler will convert it to machine language…F!
3. Both! Java uses
   a compiler from Java source to Java byte codes and an interpreter to execute Java byte codes.



And in conclusion…



Wawrzynek Spring 2006 © UCB