CS61C – Machine Structures

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

10/8/2007

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.

Scheme		Java bytecode	
Java			
C++	C	Assembly	machine language

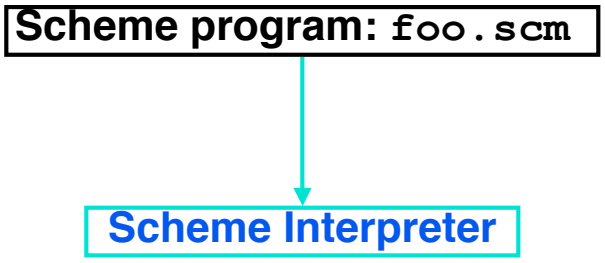
Easy to program	Efficient to interpret
Inefficient to interpret	Difficult to program

- 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

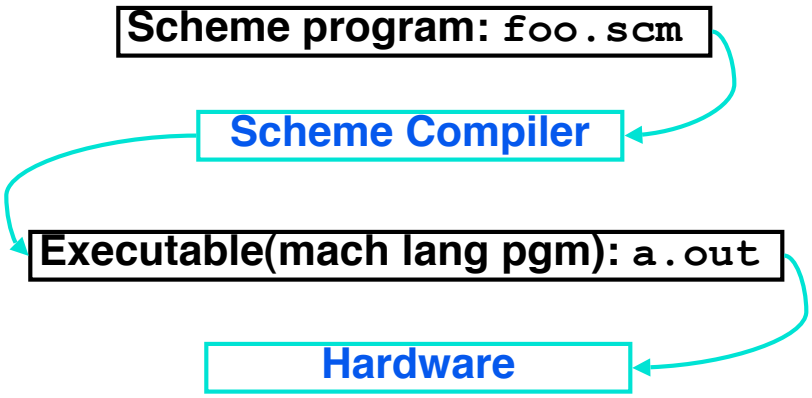
- How do we run a program written in a source language?
- Interpreter: Directly executes a program in the source language
- Translator: Converts a program from the source language to an equivalent program in another language
- For example, consider a Scheme program `foo.scm`

Interpretation



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

Translation



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

Machine Language Interpretation

- Any good reason to interpret machine language in software?
- MARS (& SPIM) – useful for learning / debugging
- Apple Macintosh conversion example
  - Switched from Motorola 680x0 instruction architecture to PowerPC.
  - Now similar issue with switch to x86.
  - Avoids all programs to be re-translated from high level language
  - Instead, let set of executables contain old and/or new machine code, interpret old code in software if necessary
- For faster execution this is often done through translation to native ISA

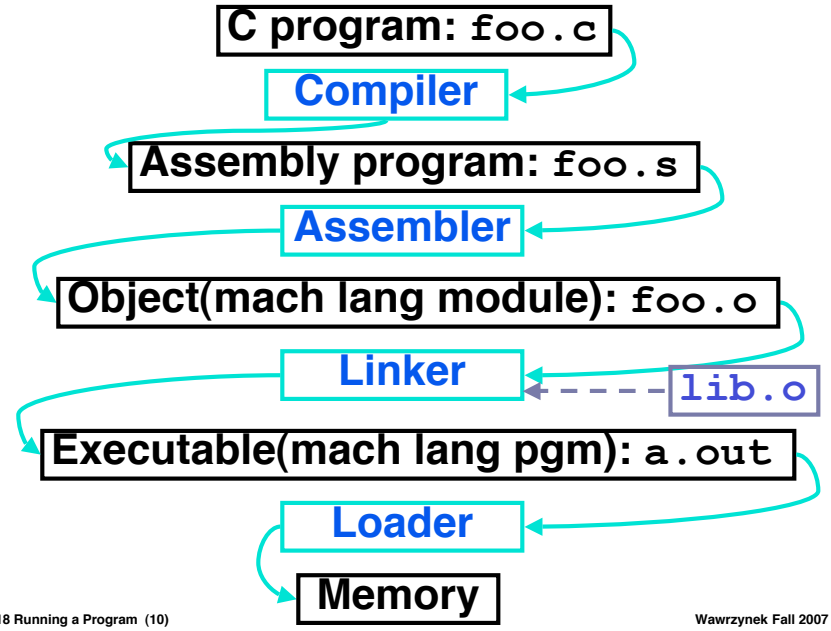
Interpretation vs. Translation? (1/2)

- Generally easier to write interpreter
- Interpreter closer to high-level, so can give better error messages (e.g., MARS)
  - 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
  - Assuming interpreter written in a protable form

Interpretation vs. Translation? (2/2)

- Translated/compiled code almost always more efficient and therefore <u>higher performance</u>:
  - 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. (eg. Linux)

Steps to Starting a Program (translation)



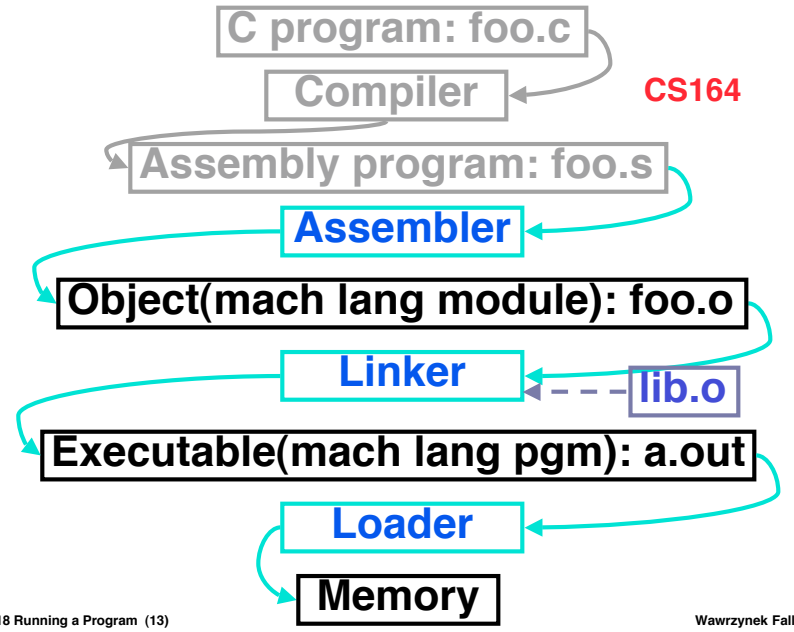
Compiler

- 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`

Administrivia

- Midterm 1, 10/16 (Week from tomorrow)
- 155 Dwinelle, 7-9pm sharp
  - Material Through 10/5
  - Open Book/Notes, but no electronic devices of any kind!
- Review session
  Friday 10/12, 4-9pm, The Woz, 430/434 Soda
  - Pizza will be served!

Where Are We Now?



Wawrzynek Fall 2007 © UCB

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
- Allow programmers to associate arbitrary names (*labels* or *symbols*) with memory locations.
- Produce Machine Language
- Creates Object File

Wawrzynek Fall 2007 © UCB

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.

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

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 usually will require the full 32-bit address of the data.
- These can't be determined yet, so we create two tables…

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 global part of the `.data` section; variables which may be accessed across files

Relocation Table

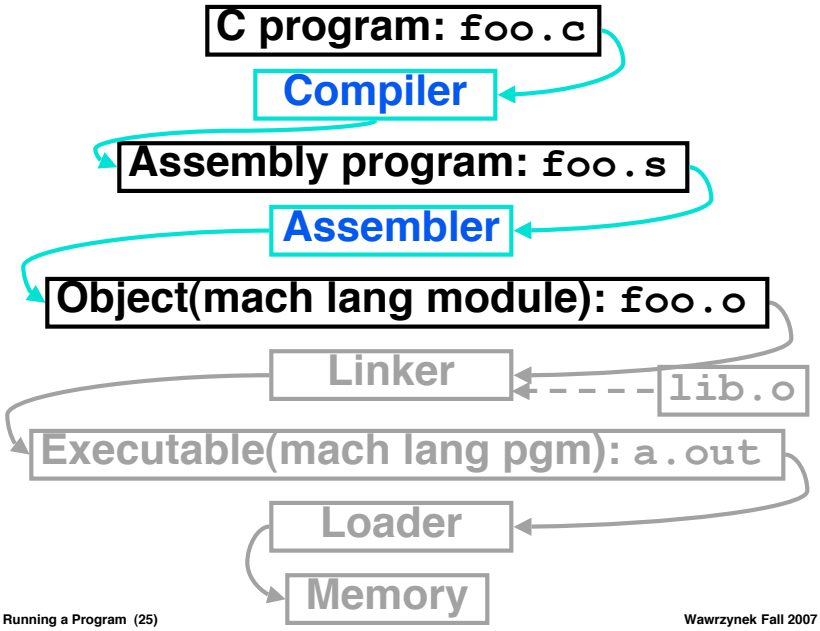
- List of "items" for which this file needs the address.
- What are they?
  - Any label jumped to: `j` or `jal`
    - internal
    - external (including lib files)
  - Any data label reference
    - such as the `la` instruction

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

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.

And in conclusion…