Lecture 18 – Running a Program I
aka Compiling, Assembling, Linking, Loading (CALL)

Cloak of invisibility?!

Researchers at U Penn have discovered a type of “invisibility shielding” to camouflage an object with a “plasmonic” screen that suppresses scattering of single-\(\lambda\) light. Star Trek?

Overview

• Interpretation vs Translation

• Translating C Programs
  • Compiler
  • Assembler
  • Linker (next time)
  • Loader (next time)

• An Example (next time)
## Language Continuum

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Java bytecode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td></td>
</tr>
<tr>
<td>C++</td>
<td>C</td>
</tr>
<tr>
<td>C</td>
<td>Assembly</td>
</tr>
<tr>
<td>Assembly</td>
<td>machine language</td>
</tr>
</tbody>
</table>

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

- In general, we interpret a high level language if efficiency is not critical or translated 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
Translation

Scheme program: foo.scm

Scheme Compiler

Executable (mach lang pgm): a.out

Hardware

° Scheme Compiler is a translator from Scheme to 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.
  • Could require all programs to be retranslated from high level language
  • Instead, let executables contain old and/or new machine code, interpret old code in software if necessary
Interpretation vs. Translation?

• Easier to write interpreter

• Interpreter closer to high-level, so gives 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
Steps to Starting a Program

- **C program**: `foo.c`
- **Assembly program**: `foo.s`
- **Object (mach lang module)**: `foo.o`
- **Executable (mach lang pgm)**: `a.out`

---

**Compilers and Assemblers**

- **Compiler**
- **Assembler**

**Linker and Loader**

- **Linker**
- **Loader**

**Memory**
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` $\Rightarrow$ or `$s1,$s2,$zero`
# Upcoming Calendar

<table>
<thead>
<tr>
<th>Week #</th>
<th>Mon</th>
<th>Wed</th>
<th>Thurs Lab</th>
<th>Fri</th>
</tr>
</thead>
<tbody>
<tr>
<td>#7 This week</td>
<td>MIPS III</td>
<td>Running Program I</td>
<td>Running Program</td>
<td>Running Program II</td>
</tr>
<tr>
<td>#8 Midterm week</td>
<td>Intro to SDS I</td>
<td>Intro to SDS II</td>
<td>SDS</td>
<td>Intro to SDS III</td>
</tr>
<tr>
<td></td>
<td>Midterm @ 7pm 1 Le Conte</td>
<td></td>
<td></td>
<td>Midterm grades out</td>
</tr>
<tr>
<td></td>
<td>(review Sun @ 2pm 10 Evans)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Administrivia…Midterm in 5 days!

- 2005-03-07 @ 7-10pm in 1 Piminitel
- Covers labs, hw, proj, lec up to SDS
- Last sem midterm + answers on www
- Bring…
  - NO backpacks, cells, calculators, pagers, PDAs
  - 2 Pens (we’ll provide write-in exam booklets)
  - One handwritten (both sides) 8.5”x11” paper
  - One green sheet (corrections below to bugs from “Core Instruction Set”)

1) Opcode wrong for Load Word. It should say 23hex, not 0 / 23hex.

2) sll and srl should shift values in R[rt], not R[rs] i.e. sll/srl: \( R[rd] = R[rt] \ll \text{shamt} \)
Where Are We Now?

C program: foo.c
Compiler
Assembly program: foo.s
Assembler
Object (mach lang module): foo.o
Linker
Executable (mach lang pgm): a.out
Loader
Memory
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

<table>
<thead>
<tr>
<th>Pseudo</th>
<th>Real</th>
</tr>
</thead>
<tbody>
<tr>
<td>subu $sp,$sp,32</td>
<td>addiu $sp,$sp,-32</td>
</tr>
<tr>
<td>sd $a0, 32($sp)</td>
<td>sw $a0, 32($sp)</td>
</tr>
<tr>
<td></td>
<td>sw $a1, 36($sp)</td>
</tr>
<tr>
<td>mul $t7,$t6,$t5</td>
<td>mul $t6,$t5</td>
</tr>
<tr>
<td></td>
<td>mflo $t7</td>
</tr>
<tr>
<td>addu $t0,$t6,1</td>
<td>addiu $t0,$t6,1</td>
</tr>
<tr>
<td>ble $t0,100,loop</td>
<td>slti $at,$t0,101</td>
</tr>
<tr>
<td></td>
<td>bne $at,$0,loop</td>
</tr>
<tr>
<td>la $a0, str</td>
<td>lui $at,left(str)</td>
</tr>
<tr>
<td></td>
<td>ori $a0,$at,right(str)</td>
</tr>
</tbody>
</table>
Producing Machine Language (1/2)

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

• What about Branches?
  • PC-Relative
    • So once pseudoinstructions are replaced by real ones, we know by how many instructions to branch.

• So these can be handled easily.
Producing Machine Language (2/2)

• What about jumps (j and jal)?
  • Jumps require absolute address.

• 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...
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

• First Pass: record label-address pairs

• Second Pass: produce machine code
  • Result: can jump to a later label without first declaring it
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
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**
Peer Instruction

1. Assembler **knows where** a module’s data & instructions are in relation to other modules.

2. Assembler will **ignore the instruction** `Loop: nop` because it does nothing.

3. Java designers used an interpreter (rather than a translator) **mainly** because of (at least one of): ease of writing, better error msgs, smaller object code.

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>4</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>5</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>6</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>7</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>8</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>
Peer Instruction Answer
And in conclusion...

C program: foo.c

Compiler

Assembly program: foo.s

Assembler

Object (mach lang module): foo.o

Linker

Executable (mach lang pgm): a.out

Loader

Memory