Lecture 11:
Program Translation
Overview

- Interpretation vs Translation
- Translating C Programs
  - Compiler
  - Assembler
  - Linker
  - Loader
## 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>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
- 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 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?

1) SPIM – useful for learning / debugging

2) Apple Macintosh conversion
   • Switched from Motorola 680x0 instruction architecture to PowerPC.
   • 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?

• 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`

Steps:
1. **Compiler** processes the `foo.c` program to produce an intermediate file `foo.s`.
2. **Assembler** converts `foo.s` into an object file `foo.o`.
3. **Linker** combines `foo.o` and any external libraries (`lib.o`) to create the **Executable** `a.out`.
4. **Loader** loads `a.out` into **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 machine does not
  
  • For example:
    
    • `move $s1,$s2` or `$s1,$s2,$zero`
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-14 to A-15)

• Give directions to assembler, but do not produce machine instructions
  .text: Subsequent items put in user text segment
  .data: Subsequent items put in user data segment
  .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>mult $t6,$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>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.
• 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**
Quickie Quiz

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. Assembler keeps track of all labels in symbol table…F!

3. Java designers used an interpreter mainly because of code portability…F!

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 translater) mainly because of (at least one of): ease of writing, better error msgs, smaller object code.