Research shows laptops and tablets in class lower performance of people around them. Ban? Make ‘em sit in the back? EECS faculty mulling over!

wapo.st/1rd6LOR
Administrivia...

- **Midterm Exam** - You get to bring
  - Your study sheet
  - Your green sheet
  - Pens & Pencils

- **What you don’t need to bring**
  - Calculator, cell phone, pagers

- **Conflicts? DSP accomodations?** Email Head TA
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 Compiler is a translator from Scheme to machine language.
- The processor is a hardware interpreter of machine language.
Steps to Starting a Program (translation)

1. C program: foo.c
2. Compiler
3. Assembly program: foo.s
4. Assembler
5. Object (mach lang module): foo.o
6. Linker
7. Executable (mach lang pgm): a.out
8. Loader
9. 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
  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

lib.o
Assembler

- Input: Assembly Language Code (MAL) (e.g., `foo.s` for MIPS)
- Output: Object Code, information tables (TAL) (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><code>subu $sp,$sp,32</code></td>
<td><code>addiu $sp,$sp,-32</code></td>
</tr>
<tr>
<td><code>sd $a0, 32($sp)</code></td>
<td><code>sw $a0, 32($sp)</code></td>
</tr>
<tr>
<td><code>addu $t0,$t6,1</code></td>
<td><code>addiu $t0,$t6,1</code></td>
</tr>
<tr>
<td><code>ble $t0,100,loop</code></td>
<td><code>slti $at,$t0,101</code></td>
</tr>
<tr>
<td><code>la $a0, str</code></td>
<td><code>lui $at,left(str)</code></td>
</tr>
<tr>
<td><code>mul $t7,$t6,$t5</code></td>
<td><code>mul $t6,$t5</code></td>
</tr>
<tr>
<td><code>mflo $t7</code></td>
<td></td>
</tr>
<tr>
<td><code>ble $t0,100,loop</code></td>
<td><code>bne $at,$0,loop</code></td>
</tr>
<tr>
<td><code>ori $a0,$at,right(str)</code></td>
<td></td>
</tr>
</tbody>
</table>
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.
“Forward Reference” problem

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

```assembly
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
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…
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
Relocation Table

- List of “items” this file needs the address later.
- 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**

A standard format is ELF (except MS)

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
Linker (1/3)

- Input: Object Code files, information tables (e.g., foo.o, libc.o for MIPS)
- Output: Executable Code (e.g., a.out for MIPS)
- Combines several object (.o) files into a single executable ("linking")
- Enable Separate Compilation of files
  - Changes to one file do not require recompilation of whole program
    - Windows NT source was > 40 M lines of code!
  - Old name "Link Editor" from editing the "links" in jump and link instructions
Linker (2/3)

.o file 1
  text 1
  data 1
  info 1

.o file 2
  text 2
  data 2
  info 2

Linker

.a.out
  Relocated text 1
  Relocated text 2
  Relocated data 1
  Relocated data 2
Linker (3/3)

- Step 1: Take text segment from each `.o` file and put them together.
- Step 2: Take data segment from each `.o` file, put them together, and concatenate this onto end of text segments.
- Step 3: Resolve References
  - Go through Relocation Table; handle each entry
  - That is, fill in all absolute addresses
Four Types of Addresses we’ll discuss

- PC-Relative Addressing (beq, bne)
  - never relocate
- Absolute Address (j, jal)
  - always relocate
- External Reference (usually jal)
  - always relocate
- Data Reference (often lui and ori)
  - always relocate
Absolute Addresses in MIPS

- Which instructions need relocation editing?
  - J-format: jump, jump and link
    
    | j/jal | xxxxx |
    |
  - Loads and stores to variables in static area, relative to global pointer
    
    | lw/sw | $gp | $x | address |
    |
  - What about conditional branches?
    
    | beq/bne | $rs | $rt | address |
    |
  - PC-relative addressing preserved even if code moves
Resolving References (1/2)

- Linker assumes first word of first text segment is at address \(0x00000000\).
  - (More later when we study “virtual memory”)
- Linker knows:
  - length of each text and data segment
  - ordering of text and data segments
- Linker calculates:
  - absolute address of each label to be jumped to (internal or external) and each piece of data being referenced
Resolving References (2/2)

- To resolve references:
  - search for reference (data or label) in all "user" symbol tables
  - if not found, search library files (for example, for `printf`)
  - once absolute address is determined, fill in the machine code appropriately

- Output of linker: executable file containing text and data (plus header)
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

CS164

lib.o
Loader Basics

- **Input**: Executable Code (e.g., `a.out` for MIPS)
- **Output**: (program is run)
- Executable files are stored on disk.
- When one is run, loader’s job is to load it into memory and start it running.
- In reality, loader is the operating system (OS)
  - loading is one of the OS tasks
Loader ... what does it do?

- Reads executable file’s header to determine size of text and data segments
- Creates new address space for program large enough to hold text and data segments, along with a stack segment
- Copies instructions and data from executable file into the new address space
- Copies arguments passed to the program onto the stack
- Initializes machine registers
  - Most registers cleared, but stack pointer assigned address of 1st free stack location
- Jumps to start-up routine that copies program’s arguments from stack to registers & sets the PC
  - If main routine returns, start-up routine terminates program with the exit system call
Conclusion

- Compiler converts a single HLL file into a single assembly language file.
- Assembler removes pseudo instructions, converts what it can to machine language, and creates a checklist for the linker (relocation table). A .s file becomes a .o file.
  - Does 2 passes to resolve addresses, handling internal forward references
- Linker combines several .o files and resolves absolute addresses.
  - Enables separate compilation, libraries that need not be compiled, and resolves remaining addresses
- Loader loads executable into memory and begins execution.

- Stored Program concept is very powerful. It means that instructions sometimes act just like data. Therefore we can use programs to manipulate other programs!
  - Compiler ⇒ Assembler ⇒ Linker (⇒ Loader)
Which of the following instr. may need to be edited during link phase?

Loop: 
1. `lui $at, 0xABCD`  
2. `ori $a0,$at, 0xFEDC`  
3. `bne $a0,$v0, Loop`  

Options: 
1. FF  
2. FT  
3. TF  
4. TT
Peer Instruction

1) Assembler will ignore the instruction \textbf{Loop: nop} because it does nothing.

2) Java designers used a translator AND interpreter (rather than just a translator) mainly because of (at least 1 of): ease of writing, better error msgs, smaller object code.

12

a) \texttt{FF}
b) \texttt{FT}
c) \texttt{TF}
d) \texttt{TT}