inst.eecs.berkeley.edu/~cs61c/su05

CS61C : Machine Structures

Lecture #12: CALL

2005-07-11

Andy Carle

CALL Overview

- Interpretation vs Translation
- Translating C Programs
  - Compiler
  - Assembler
  - Linker
  - Loader
- An Example

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

Language Continuum

Scheme
Java
C++
C
Assembly
machine language

Easy to write
Inefficient to run

Difficult to write
Efficient to run

- Interpret a high level language if efficiency is not critical
- Translate (compile) to a lower level language to improve performance
- Scheme example …

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.

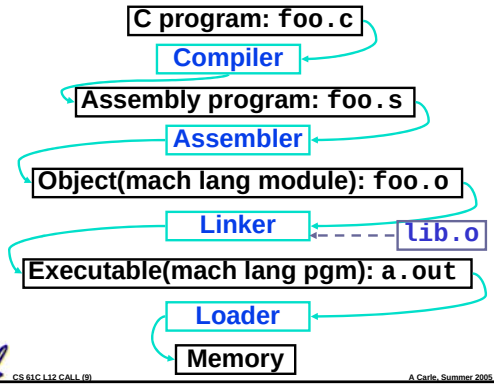
Interpretation

- 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 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
  - See Apple example

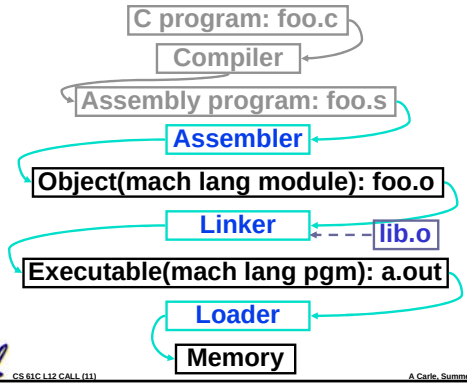
Steps to Starting a Program



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`

Where Are We Now?



Assembler

- Input: MAL 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
  - `.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

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`	`mult $t6,$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)

- Constraint on Assembler:
  - The object file output (foo.o) may be only one of many object files in the final executable:
    - C: #include “my_helpers.h”
    - C: #include <stdio.h>
- Consequences:
  - Object files won’t know their base addresses until they are linked/loaded!
  - References to addresses will have to be adjusted in later stages

Producing Machine Language (2/3)

- Simple Case
  - Arithmetic, Logical, Shifts, and so on.
  - All necessary info is within the instruction already.
- What about Branches?
  - PC-Relative and in-file
  - In TAL, we know by how many instructions to branch.
- So these can be handled easily.

Producing Machine Language (3/3)

- 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 for use by linker/loader…

1: Symbol Table

- List of “items” provided by this file.
  - What are they?
    - Labels: function calling
    - Data: anything in the `.data` section; variables which may be accessed across files
  - Includes base address of label in the file.

2: Relocation Table

- List of "items" needed by this file.
  - Any label jumped to: `j` or `jal`
    - internal
    - external (including lib files)
  - Any named piece of data
    - Anything referenced by the `la` instruction
    - static variables
  - Contains base address of instruction w/dependency, dependency name

Question

- Which lines go in the symbol table and/or relocation table?

```
my_func:
  lui $a0 my_arrayh       # a (from la)
  ori $a0 $a0 my_arrayl   # b (from la)
  jal add_link            # c
  bne $a0,$v0, my_func    # d
```

A:	Symbol: my_func	relocate: my_array
B:	-	relocate: my_array
C:	-	relocate: add_link
D:	-	-

Peer Instruction 1

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.

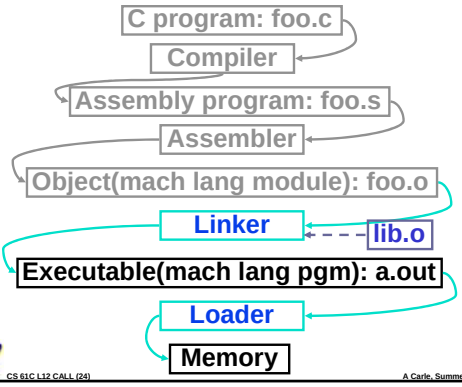
Administrivia

- Congratulations to everyone for making it through what turned out to be a fairly difficult exam
  - Midterms will be handed back either today in section or tomorrow in lab
  - Statistics will be on the website once everyone has taken the test and they have all been graded (soon)
- HW 45
  - The unholy concatenation of HW 4 and HW 5
  - Will be released today and be due Monday in lecture (paper hand-in)
  - Will be worth 20 points rather than the normal 10 points

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

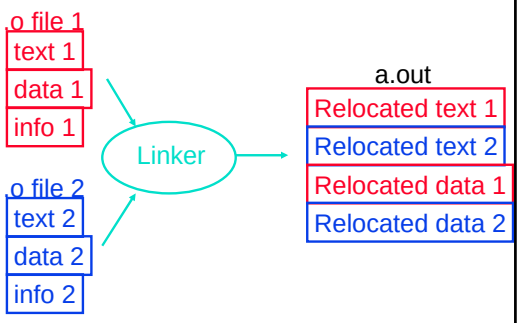
Where Are We Now?



Link Editor/Linker (1/3)

- Input: Object Code, information tables (e.g., `foo.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 is >40 M lines of code!
  - Link Editor name from editing the "links" in jump and link instructions

Link Editor/Linker (2/3)



Link Editor/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 and handle each entry
  - That is, fill in all absolute addresses

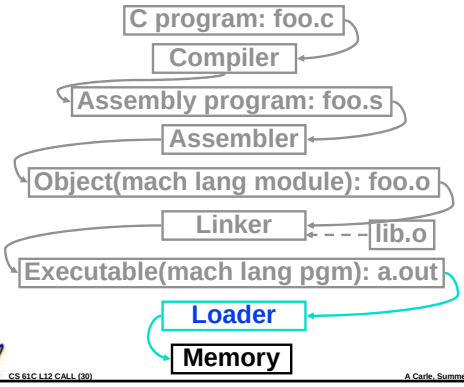
Resolving References (1/2)

- Linker *assumes* first word of first text segment is at address 0x00000000.
- 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 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?



Loader (1/3)

- 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 (2/3)

- So what does a loader 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 (this may be anywhere in memory)

Loader (3/3)

- 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 and sets the PC
  - If main routine returns, start-up routine terminates program with the exit system call

Peer Instruction 2

Which of the following instr. may need to be edited during link phase?

```
Loop: lui $at, 0xABCD      }# A
      ori $a0,$at, 0xFEDC  }
      jal add_link         # B
      bne $a0,$v0, Loop    # C
```

Things to Remember (2/3)

- Compiler converts a single HLL file into a single assembly language file.
- Assembler removes pseudoinstructions, converts what it can to machine language, and creates a checklist for the linker (relocation table). This changes each .s file into a .o file.
- Linker combines several .o files and resolves absolute addresses.
- Loader loads executable into memory and begins execution.

Things to Remember 3/3

- Stored Program concept mean instructions just like data, so can take data from storage, and keep transforming it until load registers and jump to routine to begin execution
  - Compiler ⇒ Assembler ⇒ Linker (⇒ Loader)
- Assembler does 2 passes to resolve addresses, handling internal forward references
- Linker enables separate compilation, libraries that need not be compiled, and resolves remaining addresses