

Lecture 19 – Running a Program II
aka Compiling, Assembling, Linking, Loading (CALL)



2004-10-13

Lecturer PSOE Dan Garcia

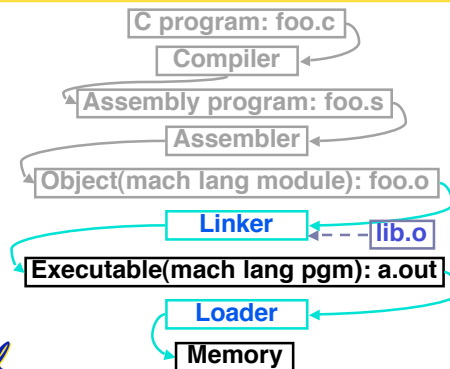
www.cs.berkeley.edu/~ddgarcia

Holiday present? →

Segway's new idea in transportation is called the Centaur, which allows for lean-forward acceleration, wheelie turns, and an enviable ride. Be the first on your block! www.segway.com/centaur



Where Are We Now?

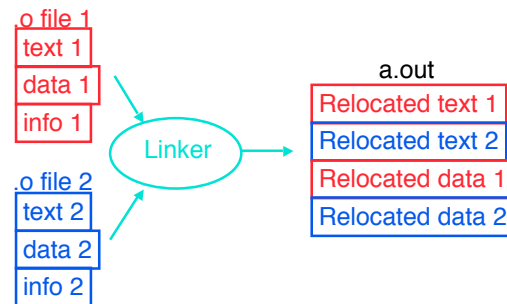


Link Editor/Linker (1/3)

- **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**



Four Types of Addresses

- **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	XXXXXXXX
-------	----------

- 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.

- 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)



Static vs Dynamically linked libraries

- What we've described is the traditional way to create a static-linked approach

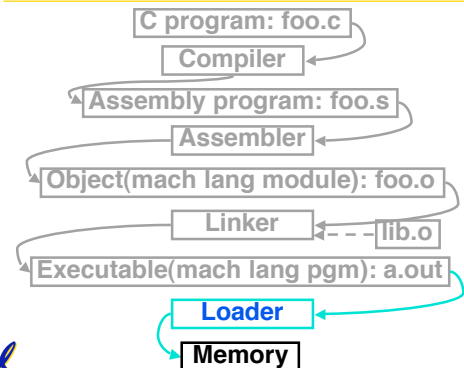
- The library is now part of the executable, so if the library updates we don't get the fix (have to recompile if we have source)
- In includes the entire library even if not all of it will be used.

- An alternative is **dynamically linked libraries** (DLL), common on Windows & UNIX platforms

- 1st run overhead for dynamic linker-loader
- Having executable isn't enough anymore!



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



Administrivia

- If you have points taken off for “not enough comments” by your reader for HW2 or HW3, then email your reader before next Monday (freeze day).
- Friday will be Intro to Synchronous Digital Systems (not Caches)
- Anonymous Survey in lab this week



Example: C ⇒ Asm ⇒ Obj ⇒ Exe ⇒ Run

```
#include <stdio.h>
int main (int argc, char *argv[]) {
    int i, sum = 0;
    for (i = 0; i <= 100; i++)
        sum = sum + i * i;
    printf ("The sum from 0 .. 100 is %d\n",
           sum);
}
```



Example: C ⇒ Asm ⇒ Obj ⇒ Exe ⇒ Run

```
.text
.align 2
.globl main
main:
    subu $sp,$sp,32
    sw $ra, 20($sp)
    sd $a0, 32($sp)
    sw $0, 24($sp)
    sw $0, 28($sp)
loop:
    lw $t6, 28($sp)
    mul $t7, $t6,$t6
    lw $t8, 24($sp)
    addu $t9,$t8,$t7
    sw $t9, 24($sp)
    addu $t0, $t6, 1
    sw $t0, 28($sp)
    ble $t0, 100, loop
    la $a0, str
    lw $a1, 24($sp)
    jal printf
    move $v0, $0
    lw $ra, 20($sp)
    addiu $sp,$sp,32
    jr $ra
    .data
    .align 0
str:
    .asciiz "The sum
from 0 .. 100 is
%d\n"
```



Example: C ⇒ Asm ⇒ Obj ⇒ Exe ⇒ Run

```
.text
.align 2
.globl main
main:
    subu $sp,$sp,32
    sw $ra, 20($sp)
    sd $a0, 32($sp)
    sw $0, 24($sp)
    sw $0, 28($sp)
loop:
    lw $t6, 28($sp)
    mul $t7, $t6,$t6
    lw $t8, 24($sp)
    addu $t9,$t8,$t7
    sw $t9, 24($sp)
    addu $t0, $t6, 1
    sw $t0, 28($sp)
    ble $t0, 100, loop
    la $a0, str
    lw $a1, 24($sp)
    jal printf
    move $v0, $0
    lw $ra, 20($sp)
    addiu $sp,$sp,32
    jr $ra
    .data
    .align 0
str:
    .asciiz "The sum
from 0 .. 100 is
%d\n"
```



Symbol Table Entries

• Symbol Table

Label Address

```
main:
loop:
str:
printf:
```

?

• Relocation Table

Address Instr. Type Dependency



Example: C ⇒ Asm ⇒ Obj ⇒ Exe ⇒ Run

• Remove pseudoinstructions, assign addresses

```
00 addiu $29,$29,-32 30 addiu $8,$14, 1
04 sw $31,20($29) 34 sw $8,28($29)
08 sw $4, 32($29) 38 slti $1,$8, 101
0c sw $5, 36($29) 3c bne $1,$0, loop
10 sw $0, 24($29) 40 lui $4, l.str
14 sw $0, 28($29) 44 ori $4,$4,r.str
18 lw $14, 28($29) 48 lw $5,24($29)
1c multu $14, $14 4c jal printf
20 mflo $15 50 add $2, $0, $0
24 lw $24, 24($29) 54 lw $31,20($29)
28 addu $25,$24,$15 58 addiu $29,$29,32
2c sw $25, 24($29) 5c jr $31
```



Symbol Table Entries

• Symbol Table

Label Address

```
main: 0x00000000
loop: 0x00000018
str: 0x10000430
printf: 0x000003b0
```

• Relocation Information

Address	Instr. Type	Dependency
0x00000040	lui	l.str
0x00000044	ori	r.str
0x0000004c	jal	printf



Example: C ⇒ Asm ⇒ Obj ⇒ Exe ⇒ Run

• Edit Addresses: start at 0x0040000

```
00 addiu $29,$29,-32 30 addiu $8,$14, 1
04 sw $31,20($29) 34 sw $8,28($29)
08 sw $4, 32($29) 38 slti $1,$8, 101
0c sw $5, 36($29) 3c bne $1,$0, -10
10 sw $0, 24($29) 40 lui $4, 4096
14 sw $0, 28($29) 44 ori $4,$4,1072
18 lw $14, 28($29) 48 lw $5,24($29)
1c multu $14, $14 4c jal 812
20 mflo $15 50 add $2, $0, $0
24 lw $24, 24($29) 54 lw $31,20($29)
28 addu $25,$24,$15 58 addiu $29,$29,32
2c sw $25, 24($29) 5c jr $31
```



Example: C ⇒ Asm ⇒ Obj ⇒ Exe ⇒ Run

```
0x004000 0010011110111101111111111100000
0x004004 10101111101111111000000000010100
0x004008 10101111101001000000000000010000
0x00400c 101011111010010100000000000100100
0x004010 10101111101000000000000000011000
0x004014 10101111101000000000000000011100
0x004018 10001111101011100000000000011100
0x00401c 10001111101110000000000000011000
0x004020 00000001110011100000000000011001
0x004024 0010010111001000000000000000001
0x004028 00101001000000010000000001100101
0x00402c 10101111101010000000000000011100
0x004030 000000000000000001110000010010
0x004034 0000001100001111110100000100001
0x004038 000101000010000011111111110111
0x00403c 10101111101110010000000000011000
0x004040 0011100000001000001000000000000
0x004044 10001111101001010000000000011000
0x004048 00001000000100000000000001110100
0x00404c 00100100100001000000010000110000
0x004050 10001111101111110000000000010100
0x004054 00100111101111010000000000100000
0x004058 0000001111100000000000000001000
0x00405c 0000000000000000000100000100001
```



Peer Instruction

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

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

- | | ABC |
|----|-----|
| 1: | FFF |
| 2: | FTT |
| 3: | FTF |
| 4: | FTT |
| 5: | TFE |
| 6: | FTT |
| 7: | TFE |
| 8: | TTT |



Peer Instruction Answer

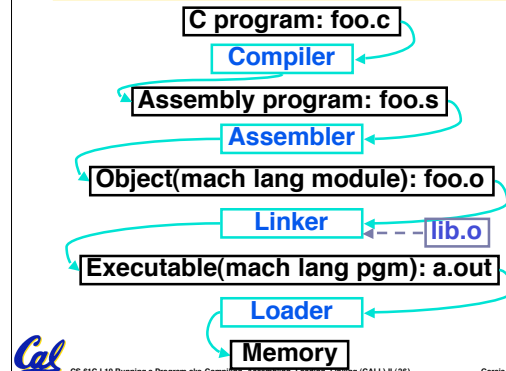
Which of the following instr. may need to be edited during link phase?

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

- 1: ABC
- 2: FFT
- 3: FFT
- 4: FFT
- 5: TFF
- 6: TFF
- 7: TFF
- 8: TTT



Things to Remember (1/3)



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

