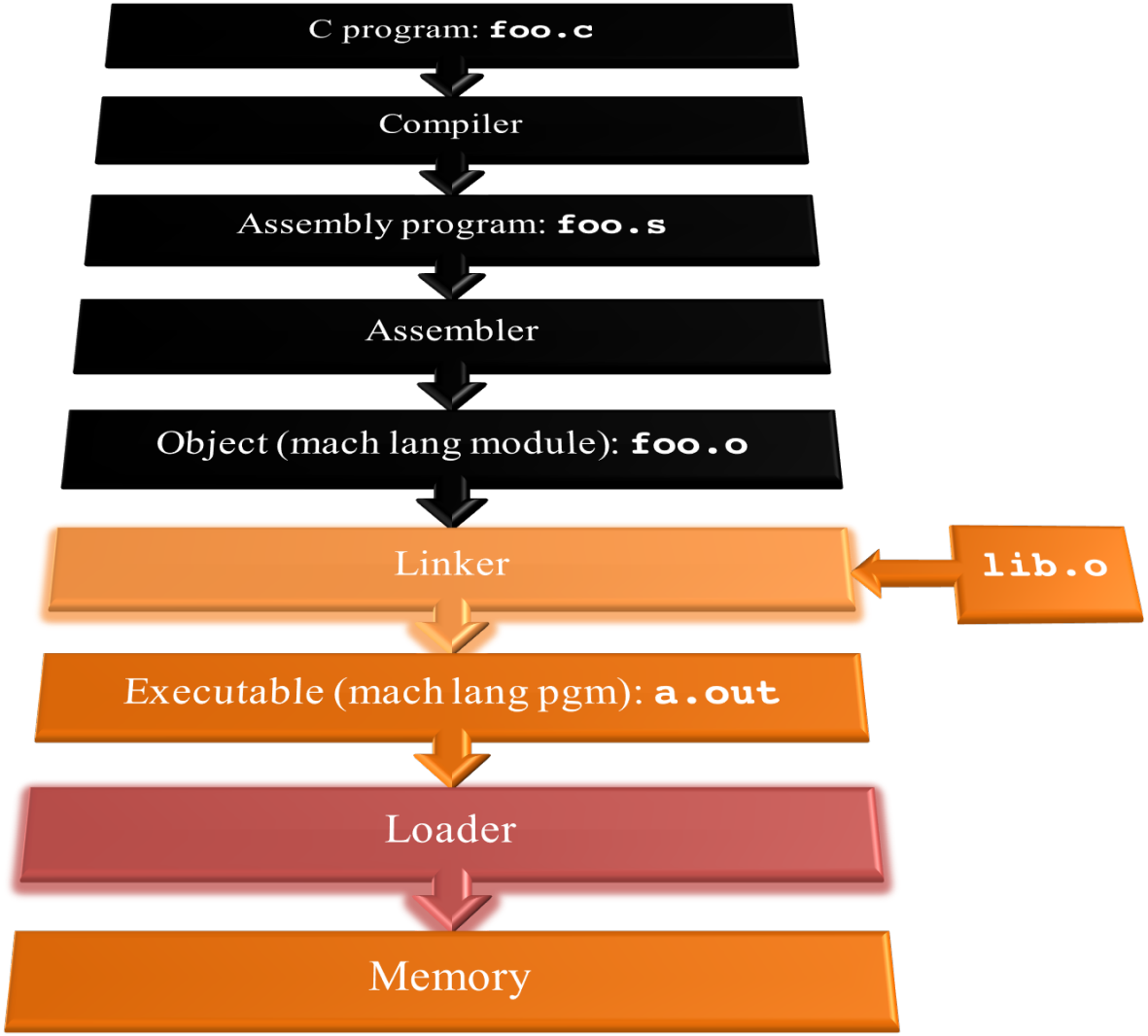
CS 61C:
Great Ideas in Computer Architecture
CALL continued
(Linking and Loading)

Instructors:

Nicholas Weaver & Vladimir Stojanovic

http://inst.eecs.Berkeley.edu/~cs61c/sp16

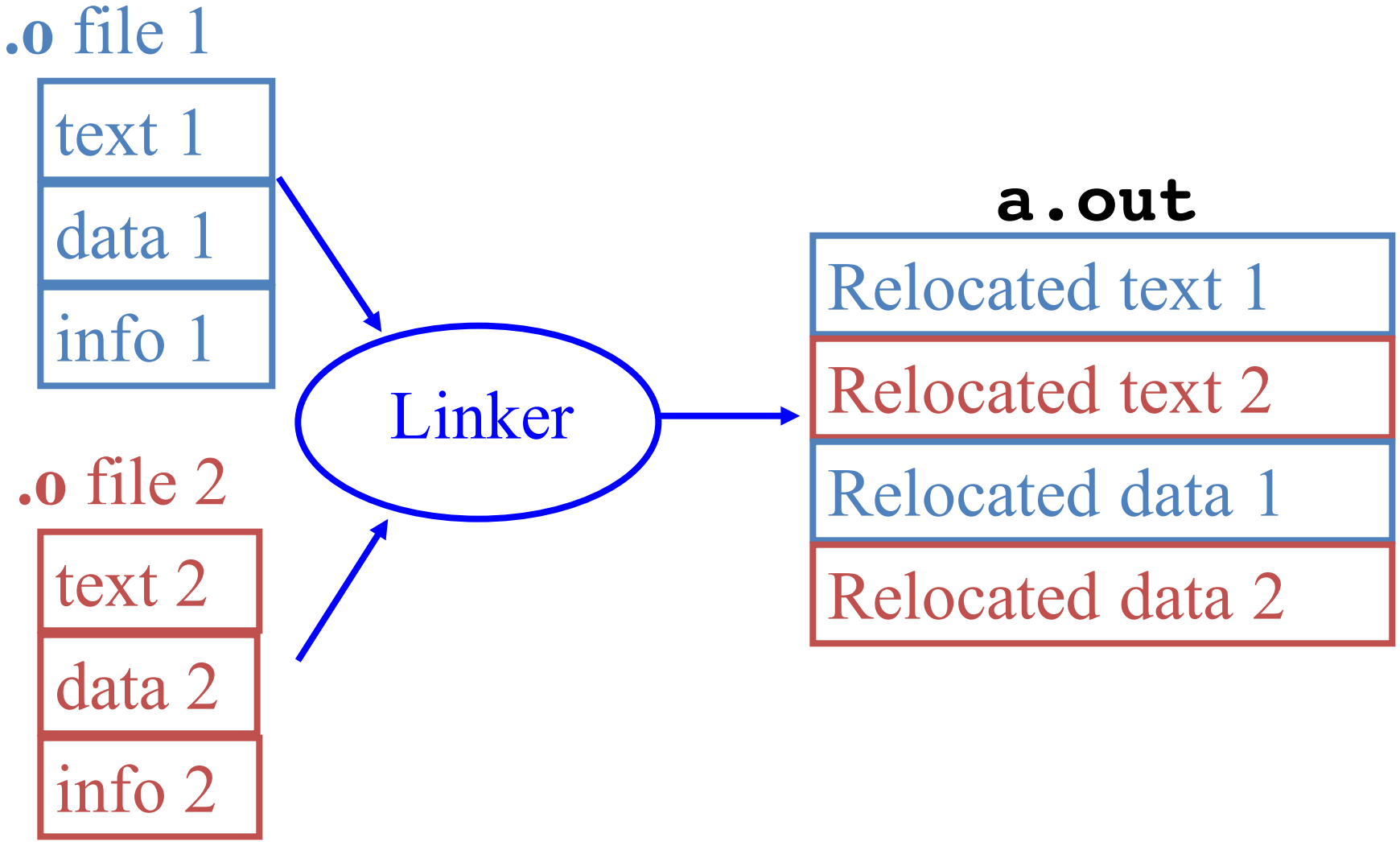
Where Are We Now?



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 the whole program
    - Windows 7 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

- PC-Relative Addressing (`beq`, `bne`)
  - never relocate
- Absolute Function Address (`j`, `jal`)
  - always relocate
- External Function Reference (usually `jal`)
  - always relocate
- Static 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

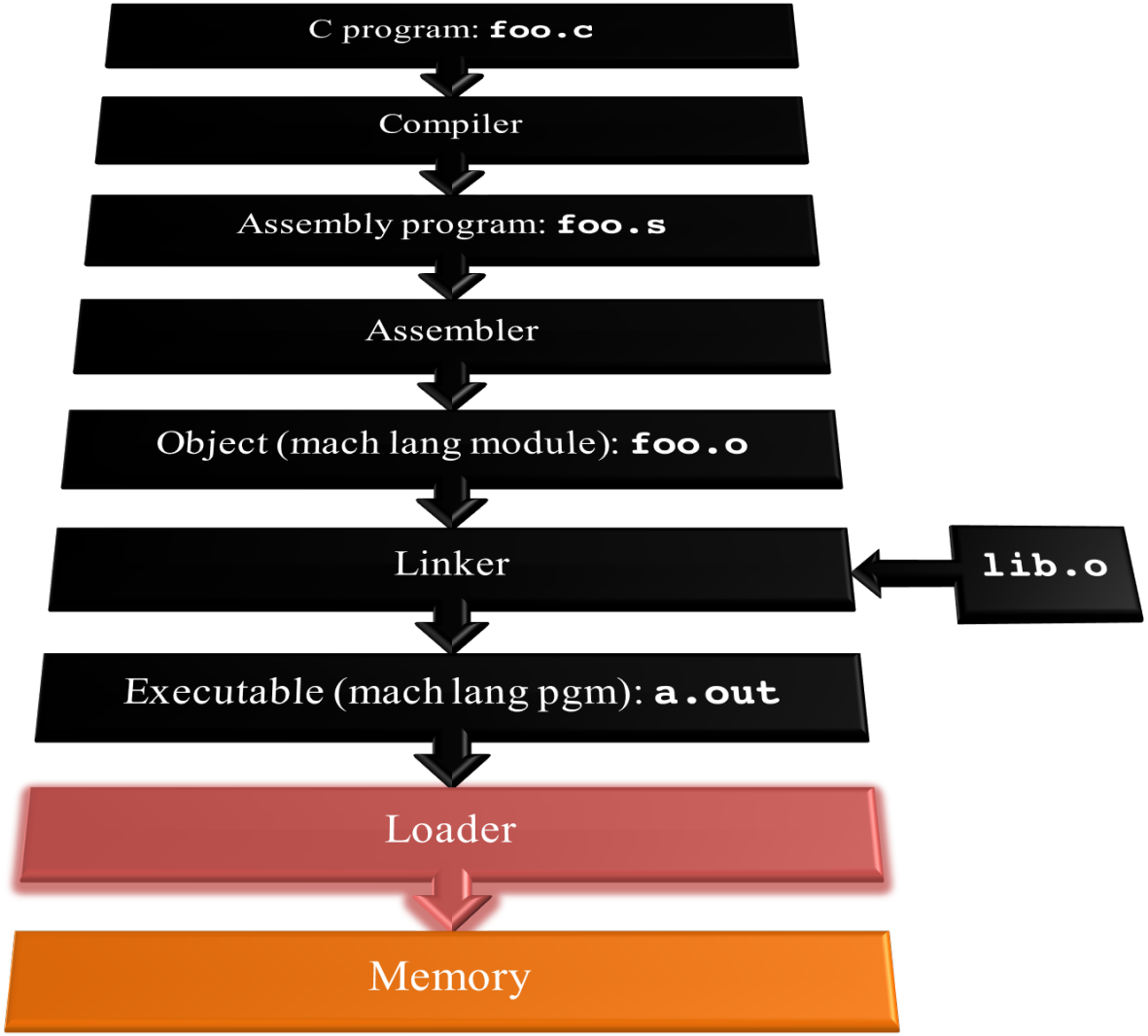
Resolving References (1/2)

- Linker assumes first word of first text segment is at address **`0x04000000`**.
  - (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?



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
  - And these days, the loader actually does a lot of the linking

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

Clicker/Peer Instruction

At what point in process are all the machine code bits determined for the following assembly instructions:

1) `addu $6, $7, $8`
2) `jal fprintf`

A: 1) & 2) After compilation

B: 1) After compilation, 2) After assembly

C: 1) After assembly, 2) After linking

D: 1) After compilation, 2) After linking

E: 1) After compilation, 2) After loading

Example: C ⇒ Asm ⇒ Obj ⇒ Exe ⇒ Run

C Program Source Code: ***prog.c***

```
#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 of sq from 0 .. 100 is
  %d\n",    sum);
}
```

*"**printf**" lives in "**libc**"*

Compilation: MAL

```
   .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
   of sq from 0 ..
   100 is %d\n"
```

Where are 7 pseudo-instructions?

Compilation: MAL

```
   .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
   of sq from 0 ..
   100 is %d\n"
```

7 pseudo-instructions underlined

Assembly step 1:

Remove pseudoinstructions, assign addresses

```
00 addiu $29,$29,-32
04 sw$31,20($29)
08 sw$4, 32($29)
0c sw$5, 36($29)
10 sw     $0, 24($29)
14 sw     $0, 28($29)
18 lw    $14, 28($29)
1c multu $14, $14
20 mflo  $15
24 lw    $24, 24($29)
28 addu $25,$24,$15
2c sw    $25, 24($29)
```

```
30 addiu $8,$14, 1
34 sw$8,28($29)
38 slti $1,$8, 101
3c bne  $1,$0, loop
40 lui  $4, l.str
44 ori  $4,$4,r.str
48 lw$5,24($29)
4c jal  printf
50 add  $2, $0, $0
54 lw     $31,20($29)
58 addiu $29,$29,32
5c jr $31
```

Assembly step 2

Create relocation table and symbol table

- Symbol Table

Label	address (in module)	type
main:	0x00000000	global text
loop:	0x00000018	local text
str:	0x00000000	local data

- Relocation Information

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

Assembly step 3

Resolve local PC-relative labels

```
00 addiu $29,$29,-32
04 sw    $31,20($29)
08 sw    $4, 32($29)
0c sw    $5, 36($29)
10 sw    $0, 24($29)
14 sw    $0, 28($29)
18 lw    $14, 28($29)
1c multu $14, $14
20 mflo  $15
24 lw    $24, 24($29)
28 addu  $25,$24,$15
2c sw    $25, 24($29)
```

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

Assembly step 4

- Generate object (`.o`) file:
  - Output binary representation for
    - text segment (instructions)
    - data segment (data)
    - symbol and relocation tables
  - Using dummy "placeholders" for unresolved absolute and external references

Text segment in object file

```
0x000000    00100111101111011111111111100000
0x000004    10101111101111110000000000010100
0x000008    10101111101001000000000000100000
0x00000c    10101111101001010000000000100100
0x000010    10101111101000000000000000011000
0x000014    10101111101000000000000000011100
0x000018    10001111101011100000000000011100
0x00001c    10001111101110000000000000011000
0x000020    00000001110011100000000000011001
0x000024    00100101110010000000000000000001
0x000028    00101001000000010000000001100101
0x00002c    10101111101010000000000000011100
0x000030    00000000000000001111000000010010
0x000034    00000011000011111110010000100001
0x000038    00010100001000001111111111110111
0x00003c    10101111101110010000000000011000
0x000040    00111100000000100000000000000000
0x000044    10001111101001010000000000000000
0x000048    00001100000100000000000011101100
0x00004c    00100100000000000000000000000000
0x000050    10001111101111110000000000010100
0x000054    00100111101111010000000000100000
0x000058    00000011111000000000000000001000
0x00005c    00000000000000000001000000100001
```

Link step 1: combine `prog.o`, `libc.o`

- Merge text/data segments
- Create absolute memory addresses
- Modify & merge symbol and relocation tables
- 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   …
```

Link step 2:

- Edit Addresses in relocation table
  - (shown in TAL for clarity, but done in binary)

```
00 addiu $29,$29,-32
04 sw$31,20($29)
08 sw$4, 32($29)
0c sw$5, 36($29)
10 sw    $0, 24($29)
14 sw    $0, 28($29)
18 lw   $14, 28($29)
1c multu $14, $14
20 mflo $15
24 lw   $24, 24($29)
28 addu $25,$24,$15
2c sw   $25, 24($29)
```

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

Link step 3:

- Output executable of merged modules
  - Single text (instruction) segment
  - Single data segment
  - Header detailing size of each segment

- NOTE:
  - The preceeding example was a much simplified version of how ELF and other standard formats work, meant only to demonstrate the basic principles.

Static vs Dynamically linked libraries

- What we've described is the traditional way: statically-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)
  - It includes the entire library even if not all of it will be used
  - Executable is self-contained
- An alternative is dynamically linked libraries (DLL), common on Windows & UNIX platforms

en.wikipedia.org/wiki/Dynamic_linking

Dynamically linked libraries

- Space/time issues
  - + Storing a program requires less disk space
  - + Sending a program requires less time
  - + Executing two programs requires less memory (if they share a library)
  - – At runtime, there's time overhead to do link
- Upgrades
  - + Replacing one file (`libXYZ.so`) upgrades every program that uses library "XYZ"
  - – Having the executable isn't enough anymore

Overall, dynamic linking adds quite a bit of complexity to the compiler, linker, and operating system. However, it provides many benefits that often outweigh these

Dynamically linked libraries

- The prevailing approach to dynamic linking uses machine code as the "lowest common denominator"
  - The linker does not use information about how the program or library was compiled (i.e., what compiler or language)
  - This can be described as "linking at the machine code level"
  - This isn't the only way to do it ...
- Also these days will ***randomize layout*** (Address Space Layout Randomization)
  - Acts as a defense to make exploiting C memory errors substantially harder, as modern exploitation requires jumping to pieces of existing code ("Return oriented programming") to counter another defense (marking heap & stack unexecutable, so attacker can't write code into just anywhere in memory).

Update Your Linux Systems!!!

- The GNU glibc has a catastrophically bad bug
  - A stack overflow in getaddrinfo()
    - Function that turns "DNS name" into "IP address"
    - **CVE-2015-7547**
      - "Common Vulnerabilities and Exposures"
- If ”bad guy” can make your program look up a name of their choosing…
  - And their bad name has a particularly long reply…
- With static linking, there would be a need to recompile and update ***hundreds*** of different programs
- With dynamic linking, "just" need to update the operating system

In 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.