18-600: Recitation #8

Oct 18th, 2016

Linking &
Midterm Review

Today

- **Linking**
- Midterm Review

Example C Program

```c
int sum(int *a, int n);

int array[2] = {1, 2};

int main()
{
    int val = sum(array, 2);
    return val;
}
```

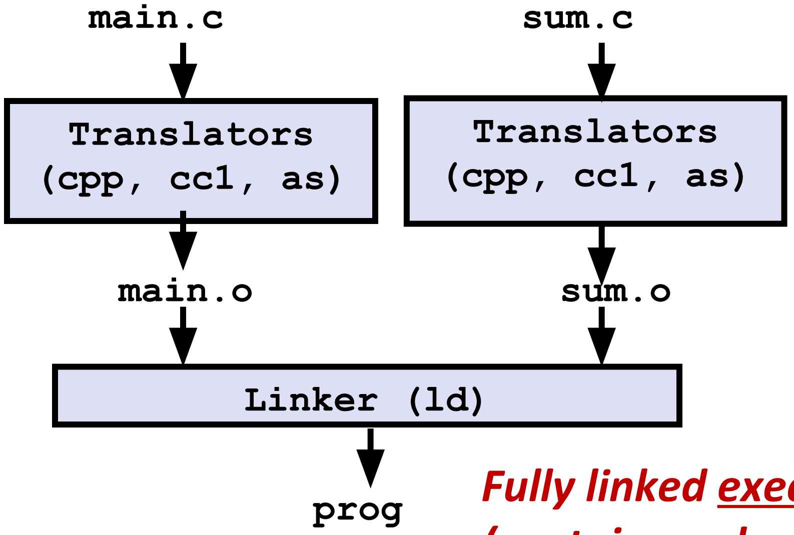
main.c

```c
int sum(int *a, int n)
{
    int i, s = 0;

    for (i = 0; i < n; i++) {
        s += a[i];
    }
    return s;
}
```

sum.c

Static Linking

- Programs are translated and linked using a *compiler driver*:
  - `linux> gcc -Og -o prog main.c sum.c`
  - `linux> ./prog`



Source files

Separately compiled relocatable object files

Fully linked executable object file (contains code and data for all functions defined in `main.c` and `sum.c`)

Why Linkers?

- **Reason 1: Modularity**
  - Program can be written as a collection of smaller source files, rather than one monolithic mass.
  - Can build libraries of common functions (more on this later)
    - e.g., Math library, standard C library

Why Linkers? (cont)

- **Reason 2: Efficiency**
  - Time: Separate compilation
    - Change one source file, compile, and then relink.
    - No need to recompile other source files.
  - Space: Libraries
    - Common functions can be aggregated into a single file...
    - Yet executable files and running memory images contain only code for the functions they actually use.

What Do Linkers Do?

Step 1: Symbol resolution

- Programs define and reference *symbols* (global variables and functions):
  - `void swap() {…}    /* define symbol swap */`
  - `swap();            /* reference symbol swap */`
  - `int *xp = &x;      /* define symbol xp, reference x */`

- Symbol definitions are stored in object file (by assembler) in *symbol table*.
  - Symbol table is an array of `structs`
  - Each entry includes name, size, and location of symbol.

- **During symbol resolution step, the linker associates each symbol reference with exactly one symbol definition.**

What Do Linkers Do? (cont)

- **Step 2: Relocation**
  - Merges separate code and data sections into single sections
  - Relocates symbols from their relative locations in the `.o` files to their final absolute memory locations in the executable.
  - Updates all references to these symbols to reflect their new positions.

Let's look at these two steps in more detail....

Three Kinds of Object Files (Modules)

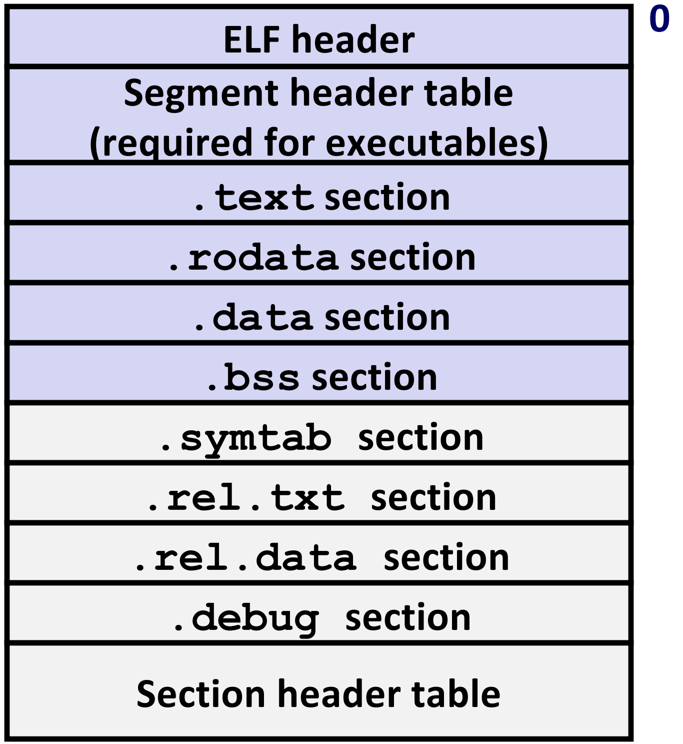
- **Relocatable object file (`.o` file)**
  - Contains code and data in a form that can be combined with other relocatable object files to form executable object file.
    - Each `.o` file is produced from exactly one source (`.c`) file
- **Executable object file (`a.out` file)**
  - Contains code and data in a form that can be copied directly into memory and then executed.
- **Shared object file (`.so` file)**
  - Special type of relocatable object file that can be loaded into memory and linked dynamically, at either load time or run-time.
  - Called *Dynamic Link Libraries* (DLLs) by Windows

Executable and Linkable Format (ELF)

- **Standard binary format for object files**
- **One unified format for**
  - Relocatable object files (`.o`),
  - Executable object files (`a.out`)
  - Shared object files (`.so`)
- **Generic name: ELF binaries**

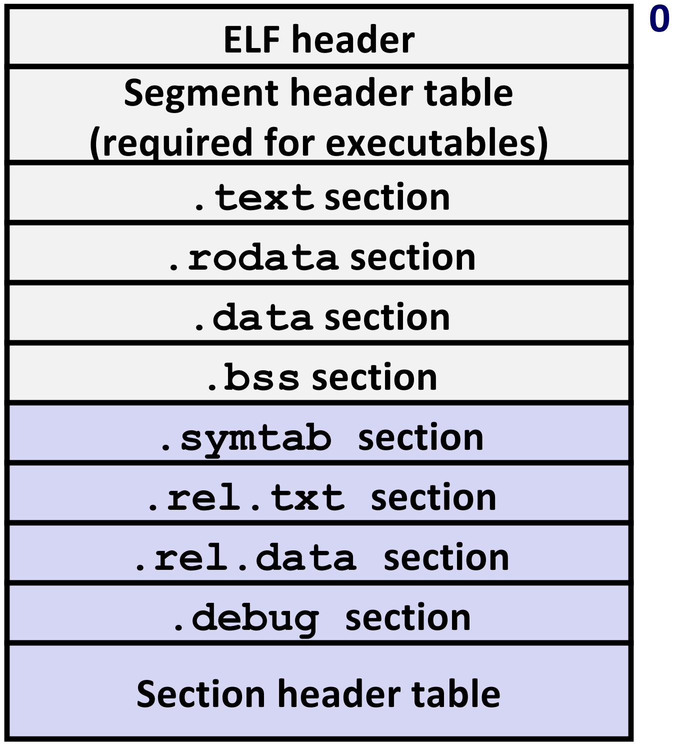
ELF Object File Format

- **Elf header**
  - Word size, byte ordering, file type (.o, exec, .so), machine type, etc.
- **Segment header table**
  - Page size, virtual addresses memory segments (sections), segment sizes.
- **.text section**
  - Code
- **.rodata section**
  - Read only data: constant strings, jump tables, ...
- **.data section**
  - Initialized global variables
- **.bss section**
  - Uninitialized global variables
  - “Block Started by Symbol”
  - “Better Save Space”
  - Has section header but occupies no space



ELF Object File Format (cont.)

- **.symtab section**
  - Symbol table
  - Procedure and static variable names
  - Section names and locations
- **.rel.text section**
  - Relocation info for **.text** section
  - Addresses of instructions that will need to be modified in the executable
  - Instructions for modifying.
- **.rel.data section**
  - Relocation info for **.data** section
  - Addresses of pointer data that will need to be modified in the merged executable
- **.debug section**
  - Info for symbolic debugging (**gcc -g**)
- **Section header table**
  - Offsets and sizes of each section



Linker Symbols

- **Global symbols**
  - Symbols defined by module *m* that can be referenced by other modules.
  - E.g.: non-**static** C functions and non-**static** global variables.

- **External symbols**
  - Global symbols that are referenced by module *m* but defined by some other module.

- **Local symbols**
  - Symbols that are defined and referenced exclusively by module *m*.
  - E.g.: C functions and global variables defined with the **static** attribute.
  - **Local linker symbols are *not* local program variables**

Step 1: Symbol Resolution

…that's defined here

Referencing a global…

```
int sum(int *a, int n);

int array[2] = {1, 2};

int main()
{
    int val = sum(array, 2);
    return val;
}
```

main.c

```
int sum(int *a, int n)
{
    int i, s = 0;

    for (i = 0; i < n; i++) {
        s += a[i];
    }
    return s;
}
```

sum.c

Defining a global

Linker knows nothing of `val`

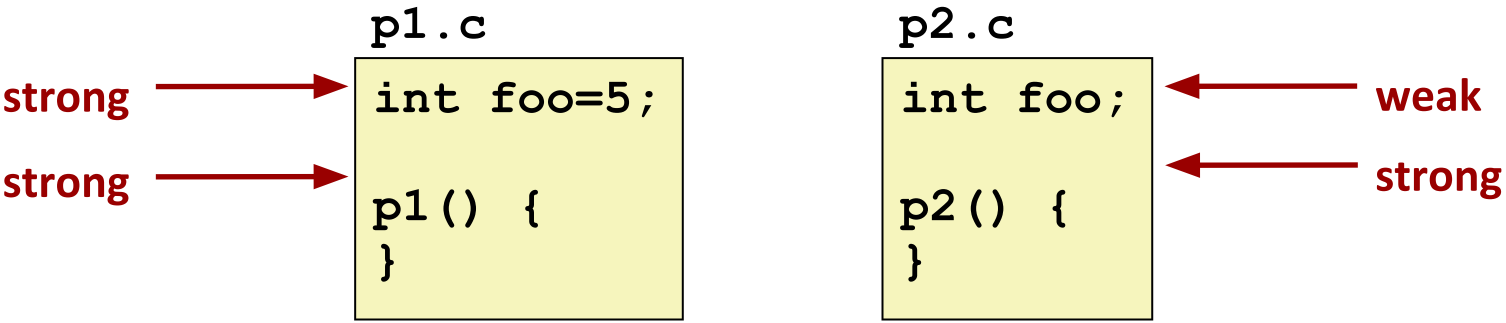
Referencing a global…

…that's defined here

Linker knows nothing of `i` or `s`

How Linker Resolves Duplicate Symbol Definitions

- **Program symbols are either *strong* or *weak***
  - ***Strong***: procedures and initialized globals
  - ***Weak***: uninitialized globals



Linker's Symbol Rules

- **Rule 1: Multiple strong symbols are not allowed**
  - Each item can be defined only once
  - Otherwise: Linker error
- **Rule 2: Given a strong symbol and multiple weak symbols, choose the strong symbol**
  - References to the weak symbol resolve to the strong symbol
- **Rule 3: If there are multiple weak symbols, pick an arbitrary one**
  - Can override this with **gcc –fno-common**

Linker Puzzles

`int x;` `p1() {}`	`p1() {}`	Link time error: two strong symbols (**`p1`**)
`int x;` `p1() {}`	`int x;` `p2() {}`	References to **`x`** will refer to the same uninitialized int. Is this what you really want?
`int x;` `int y;` `p1() {}`	`double x;` `p2() {}`	Writes to **`x`** in **`p2`** might overwrite **`y`**! Evil!
`int x=7;` `int y=5;` `p1() {}`	`double x;` `p2() {}`	Writes to **`x`** in **`p2`** will overwrite **`y`**! Nasty!
`int x=7;` `p1() {}`	`int x;` `p2() {}`	References to **`x`** will refer to the same initialized variable.

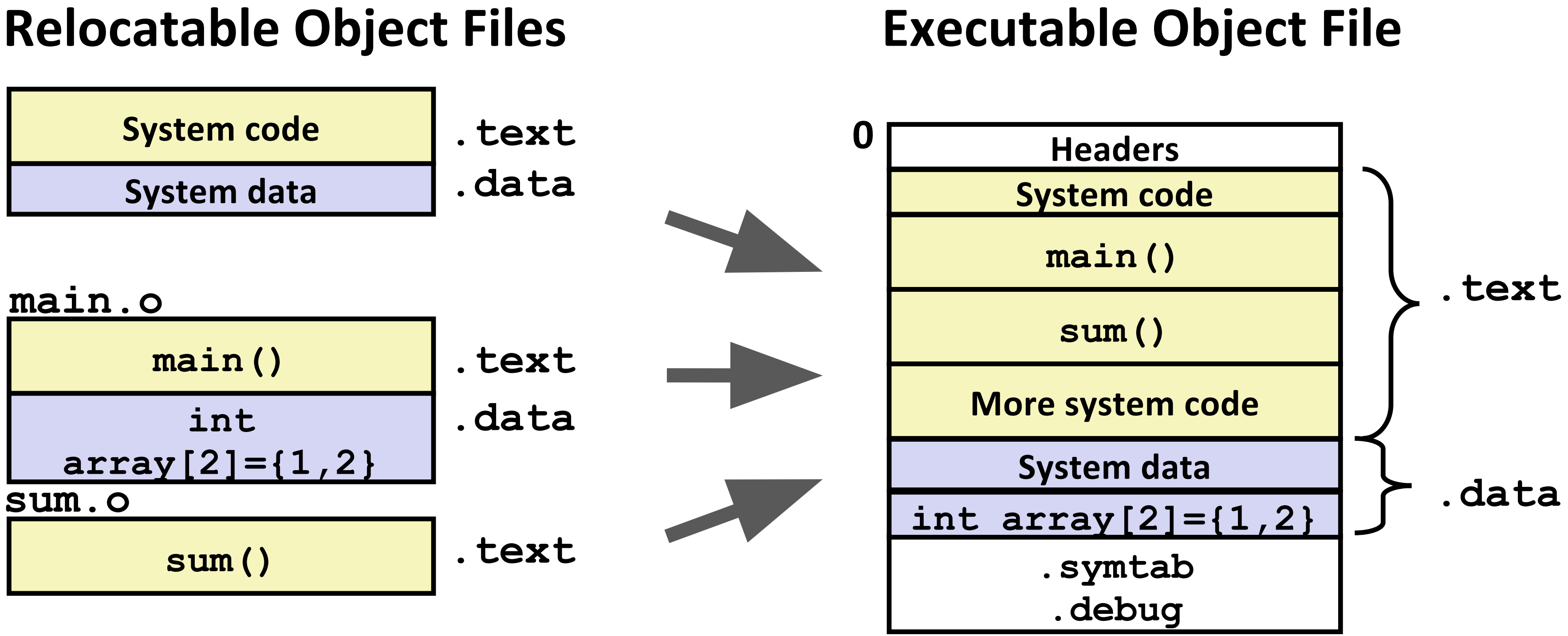
Nightmare scenario: two identical weak structs, compiled by different compilers with different alignment rules.

Global Variables

- **Avoid if you can**

- **Otherwise**
  - Use **`static`** if you can
  - Initialize if you define a global variable
  - Use **`extern`** if you reference an external global variable

Step 2: Relocation



Relocation Entries

```
int array[2] = {1, 2};

int main()
{
    int val = sum(array, 2);
    return val;
}
                                        main.c
```

```
0000000000000000 <main>:
   0:   48 83 ec 08             sub    $0x8,%rsp
   4:   be 02 00 00 00          mov    $0x2,%esi
   9:   bf 00 00 00 00          mov    $0x0,%edi       # %edi = &array
                        a: R_X86_64_32 array           # Relocation entry

   e:   e8 00 00 00 00          callq  13 <main+0x13> # sum()
                        f: R_X86_64_PC32 sum-0x4       # Relocation entry
  13:   48 83 c4 08             add    $0x8,%rsp
  17:   c3                      retq
                                                                main.o
```

Source: `objdump -r -d main.o`

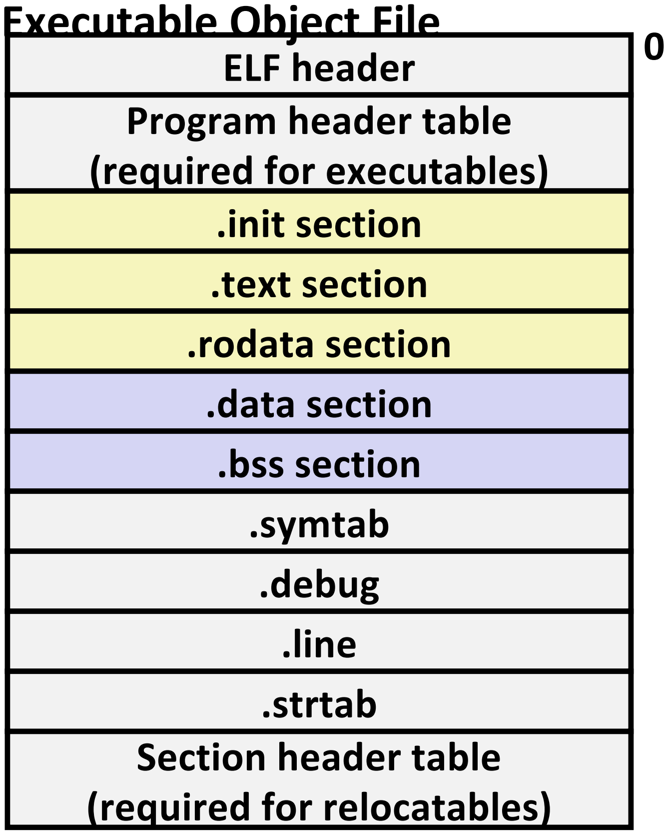
Relocated .text section

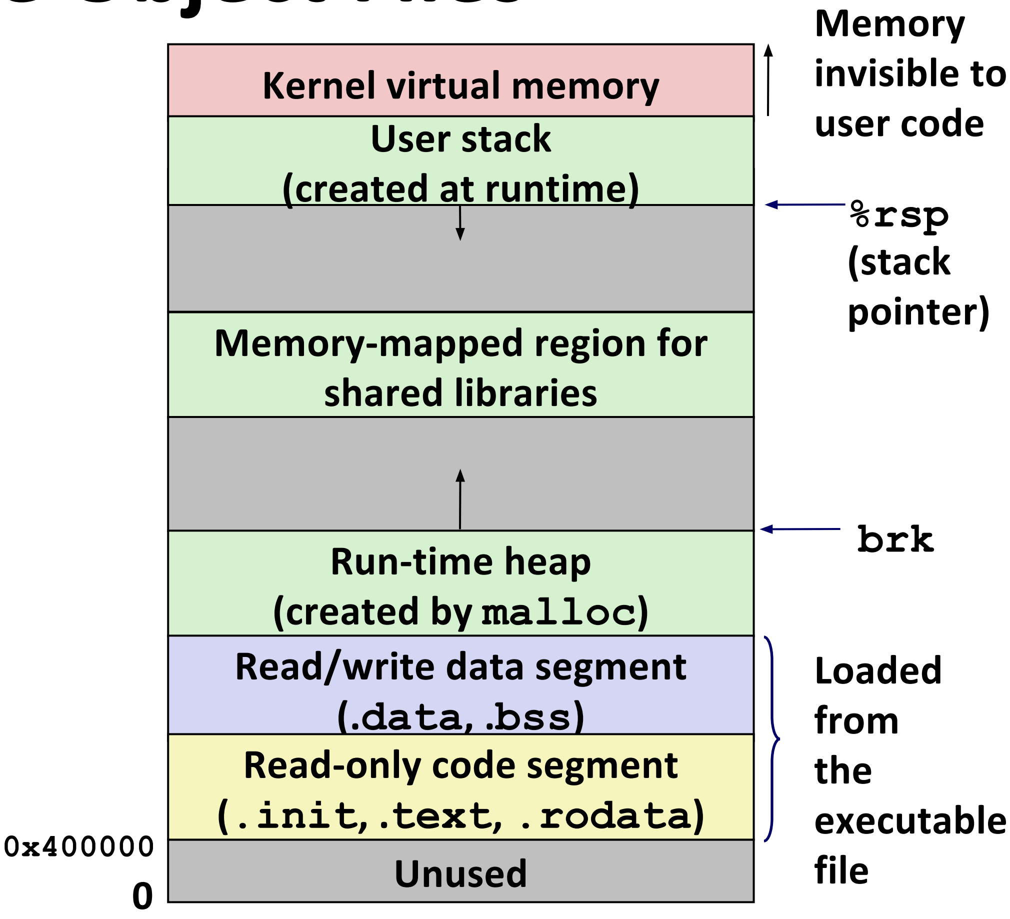
```
00000000004004d0 <main>:
  4004d0:         48 83 ec 08          sub     $0x8,%rsp
  4004d4:         be 02 00 00 00       mov     $0x2,%esi
  4004d9:         bf 18 10 60 00       mov     $0x601018,%edi  # %edi = &array
  4004de:         e8 05 00 00 00       callq   4004e8 <sum>    # sum()
  4004e3:         48 83 c4 08          add     $0x8,%rsp
  4004e7:         c3                   retq

00000000004004e8 <sum>:
  4004e8:         b8 00 00 00 00            mov     $0x0,%eax
  4004ed:         ba 00 00 00 00            mov     $0x0,%edx
  4004f2:         eb 09                     jmp     4004fd <sum+0x15>
  4004f4:         48 63 ca                  movslq  %edx,%rcx
  4004f7:         03 04 8f                  add     (%rdi,%rcx,4),%eax
  4004fa:         83 c2 01                  add     $0x1,%edx
  4004fd:         39 f2                     cmp     %esi,%edx
  4004ff:         7c f3                     jl      4004f4 <sum+0xc>
  400501:         f3 c3                     repz    retq
```

Using PC-relative addressing for sum(): 0x4004e8 = 0x4004e3 + 0x5

Source: `objdump -dx prog`

Loading Executable Object Files





Packaging Commonly Used Functions

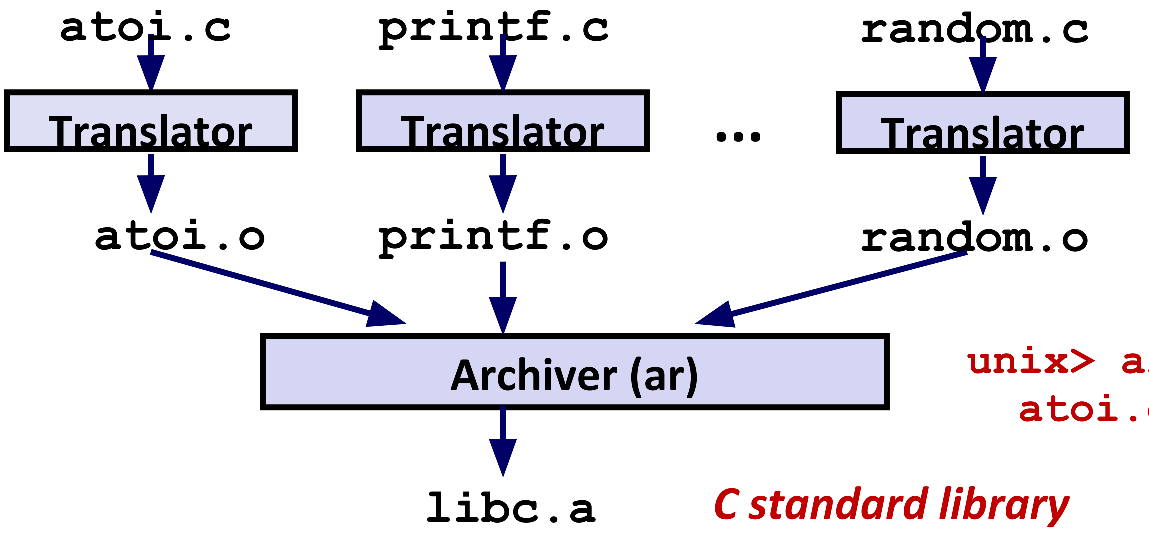
- **How to package functions commonly used by programmers?**
  - Math, I/O, memory management, string manipulation, etc.

- **Awkward, given the linker framework so far:**
  - **Option 1:** Put all functions into a single source file
    - Programmers link big object file into their programs
    - Space and time inefficient
  - **Option 2:** Put each function in a separate source file
    - Programmers explicitly link appropriate binaries into their programs
    - More efficient, but burdensome on the programmer

Old-fashioned Solution: Static Libraries

- **Static libraries (`.a` archive files)**
  - Concatenate related relocatable object files into a single file with an index (called an *archive*).
  - Enhance linker so that it tries to resolve unresolved external references by looking for the symbols in one or more archives.
  - If an archive member file resolves reference, link it into the executable.

Creating Static Libraries




```
unix> ar rs libc.a \
  atoi.o printf.o … random.o
```

- Archiver allows incremental updates
- Recompile function that changes and replace .o file in archive.

Linking with Static Libraries

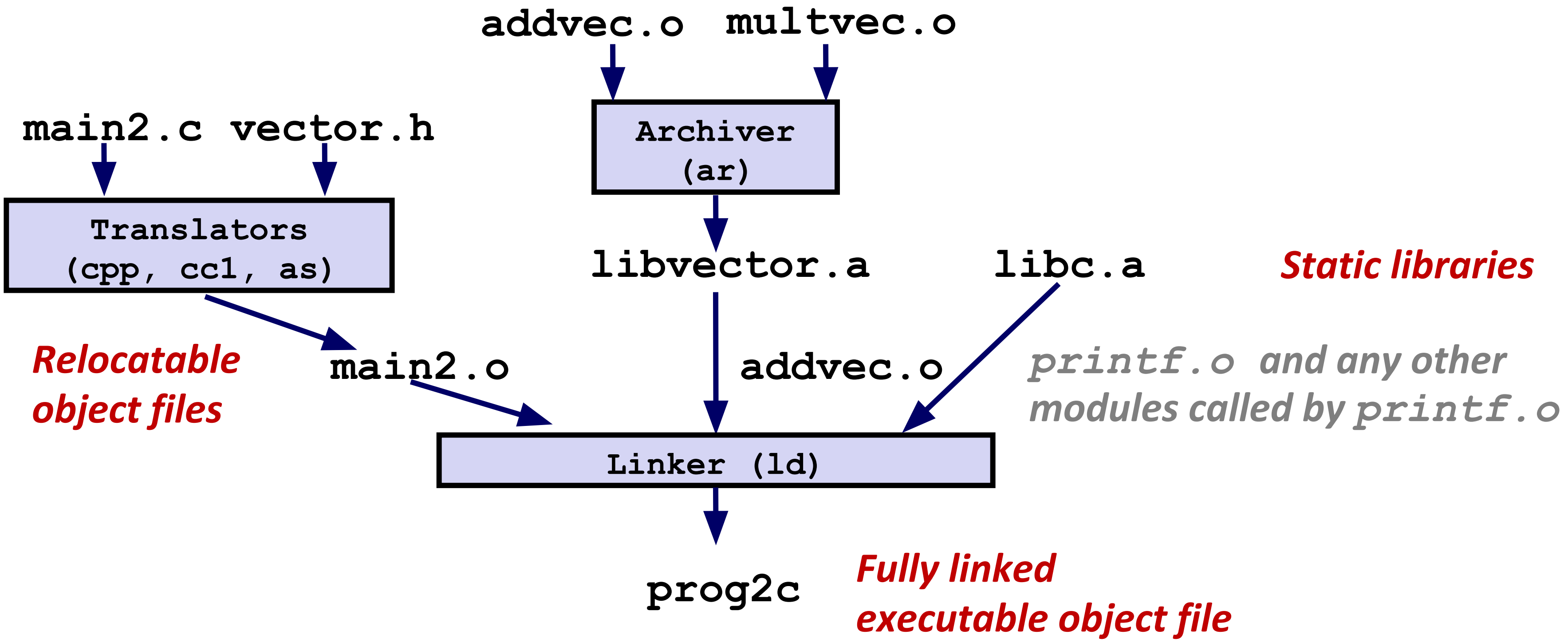
```
#include <stdio.h>
#include "vector.h"

int x[2] = {1, 2};
int y[2] = {3, 4};
int z[2];

int main()
{
    addvec(x, y, z, 2);
    printf("z = [%d %d]\n”,
           z[0], z[1]);
    return 0;
}
```
main2.c

libvector.a

```
void addvec(int *x, int *y,
            int *z, int n) {
    int i;

    for (i = 0; i < n; i++)
        z[i] = x[i] + y[i];
}
```
addvec.c

```
void multvec(int *x, int *y,
             int *z, int n)
{
    int i;

    for (i = 0; i < n; i++)
        z[i] = x[i] * y[i];
}
```
multvec.c

Linking with Static Libraries



"c" for "compile-time"

Using Static Libraries

- **Linker's algorithm for resolving external references:**
  - Scan **.o** files and **.a** files in the command line order.
  - During the scan, keep a list of the current unresolved references.
  - As each new **.o** or **.a** file, *obj*, is encountered, try to resolve each unresolved reference in the list against the symbols defined in *obj*.
  - If any entries in the unresolved list at end of scan, then error.

- **Problem:**
  - Command line order matters!
  - Moral: put libraries at the end of the command line.

```
unix> gcc -L. libtest.o -lmine
unix> gcc -L. -lmine libtest.o
libtest.o: In function `main':
libtest.o(.text+0x4): undefined reference to `libfun'
```

Modern Solution: Shared Libraries

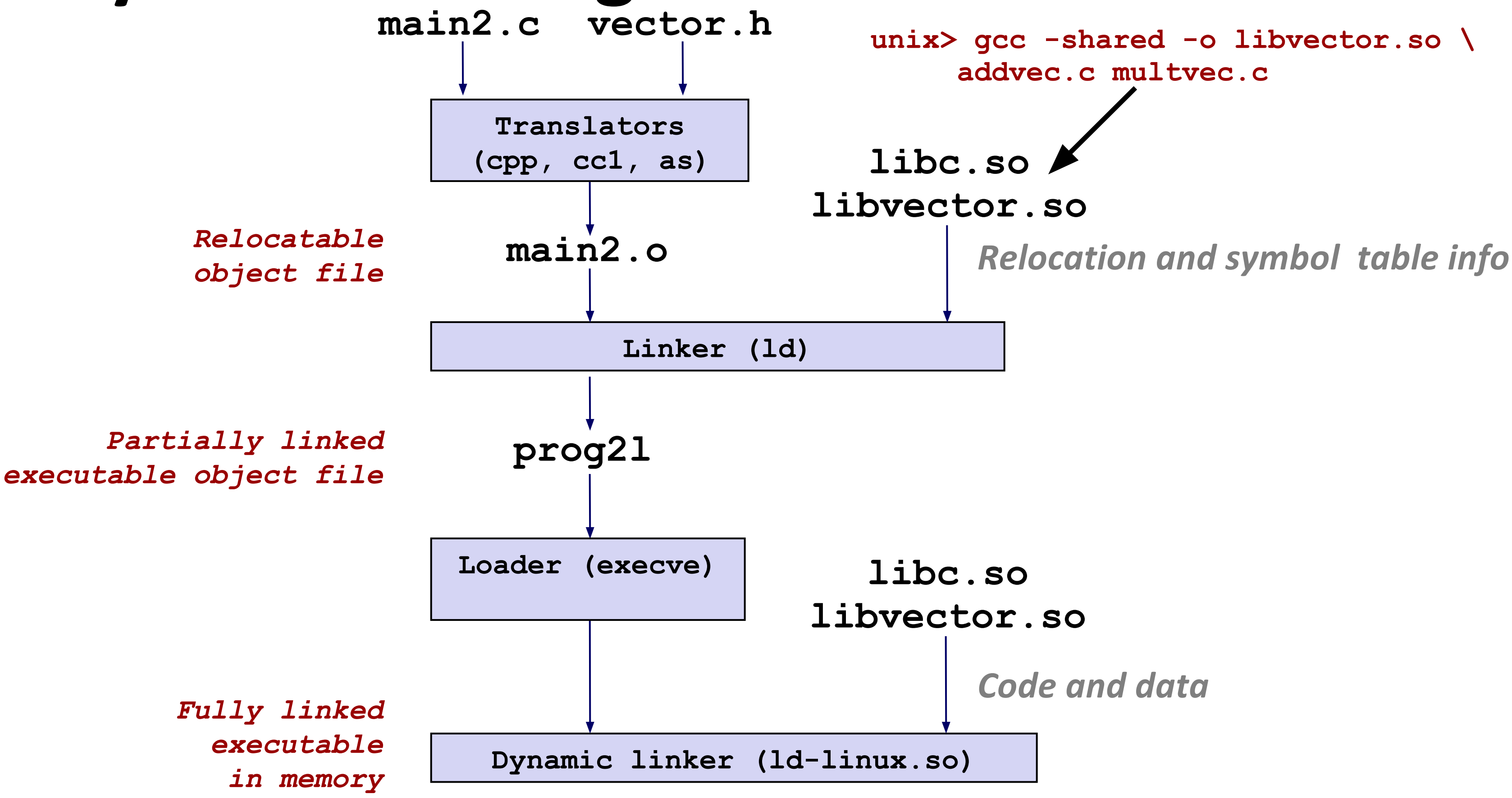
- **Static libraries have the following disadvantages:**
  - Duplication in the stored executables (every function needs libc)
  - Duplication in the running executables
  - Minor bug fixes of system libraries require each application to explicitly relink

- **Modern solution: Shared Libraries**
  - Object files that contain code and data that are loaded and linked into an application *dynamically,* at either *load-time* or *run-time*
  - Also called: dynamic link libraries, DLLs, `.so` files

Shared Libraries (cont.)

- **Dynamic linking can occur when executable is first loaded and run (load-time linking).**
  - Common case for Linux, handled automatically by the dynamic linker (**ld-linux.so**).
  - Standard C library (**libc.so**) usually dynamically linked.
- **Dynamic linking can also occur after program has begun (run-time linking).**
  - In Linux, this is done by calls to the **dlopen()** interface.
    - Distributing software.
    - High-performance web servers.
    - Runtime library interpositioning.
- **Shared library routines can be shared by multiple processes.**
  - More on this when we learn about virtual memory

Dynamic Linking at Load-time



Dynamic Linking at Run-time

```c
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

int x[2] = {1, 2};
int y[2] = {3, 4};
int z[2];

int main()
{
    void *handle;
    void (*addvec)(int *, int *, int *, int);
    char *error;

    /* Dynamically load the shared library that contains addvec() */
    handle = dlopen("./libvector.so", RTLD_LAZY);
    if (!handle) {
        fprintf(stderr, "%s\n", dlerror());
        exit(1);
    }

/* Get a pointer to the addvec() function we just loaded */
    addvec = dlsym(handle, "addvec");
    if ((error = dlerror()) != NULL) {
        fprintf(stderr, "%s\n", error);
        exit(1);
    }

    /* Now we can call addvec() just like any other function */
    addvec(x, y, z, 2);
    printf("z = [%d %d]\n", z[0], z[1]);

    /* Unload the shared library */
    if (dlclose(handle) < 0) {
        fprintf(stderr, "%s\n", dlerror());
        exit(1);
    }
    return 0;
}
```

dll.c

Linking Summary

- **Linking is a technique that allows programs to be constructed from multiple object files.**

- **Linking can happen at different times in a program's lifetime:**
  - Compile time (when a program is compiled)
  - Load time (when a program is loaded into memory)
  - Run time (while a program is executing)

- **Understanding linking can help you avoid nasty errors and make you a better programmer.**

Today

- Linking
- Midterm Review

Agenda

- Midterm Logistics
- Overview of *some* topics
- Practice Questions

Midterm

- **Wed Oct 26th**
  - Duration – 110 minutes
  - Closed book, paper exam
    - Bring your student ID with you
- **Note Sheet – ONE double sided 8 ½ x 11 paper**
  - No worked out problems on that sheet

Midterm

- **What to study?**
  - Chapters 1-4 and Chapter 6
- **How to Study?**
  - Make sure to understand the contents in the lecture slides and recitations
  - Practice problems in the text book
  - Old exam papers: http://www.cs.cmu.edu/~213/exams.html
    - Some old practice exams include questions that use the IA32 architecture (if you see %ebp, this is a good sign is 32 bit). **You will only need to know x86-64 for the midterm.**

Bits, Bytes & Integers

- **Know how to do basic bit operations by hand**
  - Shifting, addition, negation, and, or, xor, etc.
- **If you have w bits**
  - What are the largest/smallest representable signed numbers?
  - What are the largest/smallest representable unsigned numbers?
  - What happens to the bits when casting signed to unsigned (and vice versa)?
- **Distinguish between logical and bitwise operators**
- **What happens in C if you do operations on mixed types (either different size, or signedness?)**

Floating Point (IEEE Format)

- Sign, Exponent, Mantissa
  - $(-1)^s \times M \times 2^E$
  - s – sign bit
  - M – Mantissa/Fraction bits
  - E – Determined by (but not equal to) exponent bits
- Bias $(2^{k-1} - 1)$
- Three main categories of floats
  - Normalized: Large values, not near zero
  - Denormalized: Small values close to zero
  - Special Values: Infinity/NaN

Floating Point (IEEE Format)

	Normalized	Denormalized	Special Values
Represents:	Most numbers	Tiny numbers	Infinity, NaN
Exponent bits:	Not those →	000…000	111…111
E =	exp – bias	1 – bias	+/- ∞ if frac = 000…000; otherwise NaN
M =	1.frac	.frac	

- Floating Point Rounding
  - Round-up – if the spilled bits are greater than half
  - Round-down – if the spilled bits are less than half
  - Round to even – if the spilled bits are exactly equal to half

Floating point encoding. In this problem, you will work with floating point numbers based on the IEEE floating point format. We consider two different 6-bit formats:

Format A:

- There is one sign bit s.
- There are $k = 3$ exponent bits. The bias is $2^{k-1} - 1 = 3$.
- There are $n = 2$ fraction bits.

Format B:

- There is one sign bit s.
- There are $k = 2$ exponent bits. The bias is $2^{k-1} - 1 = 1$.
- There are $n = 3$ fraction bits.

For formats A and B, please write down the binary representation for the following (use round-to-even). Recall that for denormalized numbers, $E = 1 -$ bias. For normalized numbers, $E = e -$ bias.

Value	Format A Bits	Format B Bits
Zero	0 000 00	0 00 000
One		
1/2		
11/8		

- Fall 2012

Floating point encoding. In this problem, you will work with floating point numbers based on the IEEE floating point format. We consider two different 6-bit formats:

Format A:

- There is one sign bit s.
- There are $k = 3$ exponent bits. The bias is $2^{k-1} - 1 = 3$.
- There are $n = 2$ fraction bits.

Format B:

- There is one sign bit s.
- There are $k = 2$ exponent bits. The bias is $2^{k-1} - 1 = 1$.
- There are $n = 3$ fraction bits.

For formats A and B, please write down the binary representation for the following (use round-to-even). Recall that for denormalized numbers, $E = 1 - \text{bias}$. For normalized numbers, $E = e - \text{bias}$.

Value	Format A Bits	Format B Bits
Zero	0 000 00	0 00 000
One	0 011 00	0 01 000
1/2	0 010 00	0 00 100
11/8	0 011 10	0 01 011

- Fall 2012

Assembly

- Recognize common assembly instructions
- Know the uses of all registers in 64 bit systems
- Understand how different control flow is turned into assembly
  - For, while, do, if-else, switch, etc
- Be very comfortable with pointers and dereferencing
  - The use of parens in mov commands.
    - %rax vs. (%rax)
  - The options for memory addressing modes:
    - R(Rb, Ri, S)
  - lea vs. mov

Assembly Loop

```
00000000004004b6 <mystery>:
  4004b6:  mov    $0x0,%eax
  4004bb:  jmp    4004d3 <mystery+0x1d>
  4004bd:  movslq %eax,%rdx
  4004c0:  lea    (%rdi,%rdx,4),%rcx
  4004c4:  mov    (%rcx),%edx
  4004c6:  test   $0x1,%dl
  4004c9:  jne    4004d0 <mystery+0x1a>
  4004cb:  add    $0x1,%edx
  4004ce:  mov    %edx,(%rcx)
  4004d0:  add    $0x1,%eax
  4004d3:  cmp    %esi,%eax
  4004d5:  jne    4004bd <mystery+0x7>
  4004d7:  repz retq
```

```
void mystery(int *array, int n)
{
        int i;
        for(________; ________; ________)
        {
         if(____________ == 0)
             ____________;
        }
}
```

Assembly Loop

```
00000000004004b6 <mystery>:
  4004b6:  mov    $0x0,%eax
  4004bb:  jmp    4004d3 <mystery+0x1d>
  4004bd:  movslq %eax,%rdx
  4004c0:  lea    (%rdi,%rdx,4),%rcx
  4004c4:  mov    (%rcx),%edx
  4004c6:  test   $0x1,%dl
  4004c9:  jne    4004d0 <mystery+0x1a>
  4004cb:  add    $0x1,%edx
  4004ce:  mov    %edx,(%rcx)
  4004d0:  add    $0x1,%eax
  4004d3:  cmp    %esi,%eax
  4004d5:  jne    4004bd <mystery+0x7>
  4004d7:  repz retq
```

```
void mystery(int *array, int n)
{
    int i;
    for(__i = 0__; __i < n; __i++)
    {
      if((array[i] & 1) == 0)
        array[i] += 1;
    }
}
```

Array Access

- A suggested method for these problems:
  - Start with the C code
  - Then look at the assembly Work backwards!
  - Understand how in assembly, a logical 2D array is implement as a 1D array, using the width of the array as a multiplier for access

[0][0] = [0]	[0][1] = [1]	[0][2] = [2]	[0][3] = [3]
[1][0] = [4]	[1][1] = [5]	[1][2] = [6]	[1][3] = [7]
[2][0] = [8]	[2][1] = [9]	[2][2] = [10]	[2][3] = [11]

[0][2] = 0 * 4 + 2 = 2

[1][3] = 1 * 4 + 3 = 7

[2][1] = 2 * 4 + 1 = 9

[i][j] = i * width of array + j

```
int array1[H][J];
int array2[J][H];

int copy_array(int x, int y) {
    array2[y][x] = array1[x][y];

    return 1;
}
```

Find H & J

Suppose the above C code generates the following x86-64 assembly code:

```
# On entry:
#     %edi = x
#     %esi = y
#
copy_array:
        movslq  %esi,%rsi
        movslq  %edi,%rdi
        movq    %rsi, %rax
        salq    $4, %rax
        subq    %rsi, %rax
        addq    %rdi, %rax
        leaq    (%rdi,%rdi,2), %rdi
        addq    %rsi, %rdi
        movl    array1(,%rdi,4), %edx
        movl    %edx, array2(,%rax,4)
        movl    $1, %eax
        ret
```

- **Fall 2010;**

```
int array1[H][J];
int array2[J][H];

int copy_array(int x, int y) {
    array2[y][x] = array1[x][y];

    return 1;
}
```

Find H & J

Suppose the above C code generates the following x86-64 assembly code:

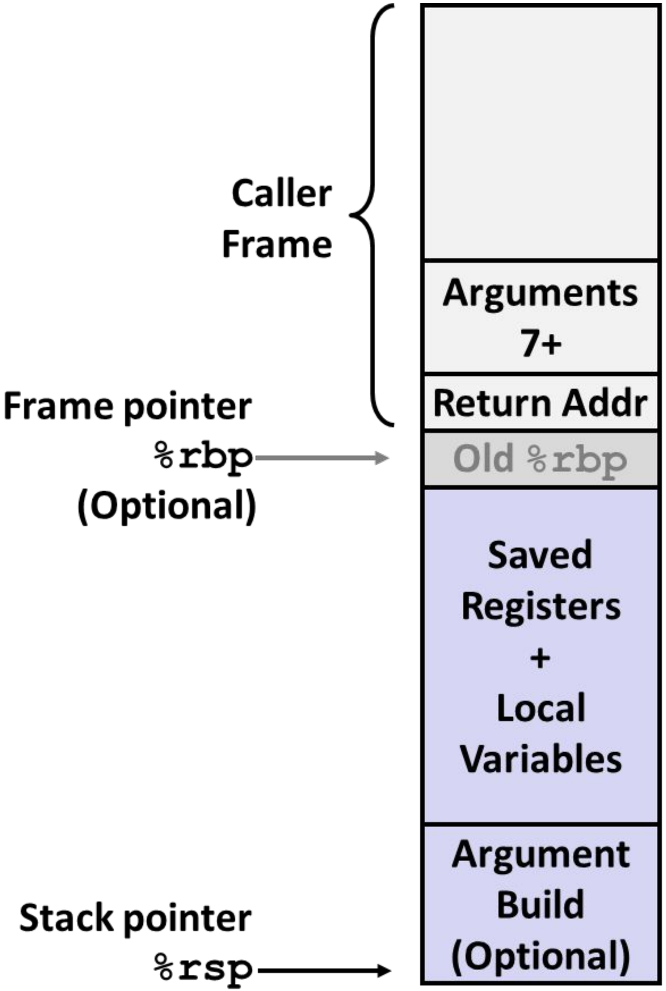
```
# On entry:
#     %edi = x
#     %esi = y
#
copy_array:
        movslq  %esi,%rsi
        movslq  %edi,%rdi
        movq    %rsi, %rax
        salq    $4, %rax
        subq    %rsi, %rax
        addq    %rdi, %rax
        leaq    (%rdi,%rdi,2), %rdi
        addq    %rsi, %rdi
        movl    array1(,%rdi,4), %edx
        movl    %edx, array2(,%rax,4)
        movl    $1, %eax
        ret
```

J = 3
H = 15

- **Fall 2010;**

Stack

- Be able to draw a stack diagram
- Recall how arguments are passed/returned from a function
  - x86-64
- How these instructions modify stack
  - call
  - ret
  - pop
  - push
  - mov



Stack

- **Review the slides on attack lab!**
- **Understand the concept of a buffer overflow.**
  - What causes it?
  - What happens to the stack?
  - How can hackers exploit this to run arbitrary code?

Pipelining

Pipeline stages

- Fetch:
  - Read instruction from instruction cache
  - Determine the registers to use
  - Update PC
- Decode:
  - Read values from registers
- Execute:
  - Perform arithmetic operations
  - Compute effective memory address
  - Check condition codes
- Memory:
  - Write to stack
  - Read from stack
  - Write/Read from computed effective memory address
- Write Back:
  - Write to registers
  - Update stack pointer

Examples of pipeline stages

Stage		OPq rA, rB	
Fetch	icode,ifun	icode:ifun ← M_1[PC]	Read instruction byte
	rA,rB	rA:rB ← M_1[PC+1]	Read register byte
	valC		[Read constant word]
	valP	valP ← PC+2	Compute next PC
Decode	valA, srcA	valA ← R[rA]	Read operand A
	valB, srcB	valB ← R[rB]	Read operand B
Execute	valE	valE ← valB OP valA	Perform ALU operation
	Cond code	Set CC	Set/use cond. code reg
Memory	valM		[Memory read/write]
Write back	dstE	R[rB] ← valE	Write back ALU result
	dstM		[Write back memory result]
PC update	PC	PC ← valP	Update PC

Stage		call Dest	
Fetch	icode,ifun	icode:ifun ← M_1[PC]	Read instruction byte
	rA,rB		[Read register byte]
	valC	valC ← M_8[PC+1]	Read constant word
	valP	valP ← PC+9	Compute next PC
Decode	valA, srcA		[Read operand A]
	valB, srcB	valB ← R[%rsp]	Read operand B
Execute	valE	valE ← valB + –8	Perform ALU operation
	Cond code		[Set /use cond. code reg]
Memory	valM	M_8[valE] ← valP	Memory read/write
Write back	dstE	R[%rsp] ← valE	Write back ALU result
	dstM		[Write back memory result]
PC update	PC	PC ← valC	Update PC

- All instructions follow same general pattern
- Differ in what gets computed on each step

Forwarding

- Purpose is to prevent stalls/bubbles
- Make results available as soon as possible to the previous pipeline stages
- Do not wait till write back stage updates the registers

Example of forwarding

- Consider the following example

```
I1: add r1, r2
I2: mrmovq d(r2), r3
I3: rmmovq r3, d(r2)
```

- Identify the data hazard
  - Case 1: I1 and I2 wrt r2
  - Case 2: I2 and I3 wrt r3
  - Case 3: I1 and I3 wrt r2
- Case 1:
  - Which is the earliest stage at which value of r2 is ready ?
  - Which is the latest by which I2 MUST receive updated r2 ?

Example of forwarding

- Consider the following example

```
I1: add r1, r2
I2: mrmovq d(r2), r3
I3: rmmovq r3, d(r2)
```

- Identify the data hazard
  - Case 1: I1 and I2 wrt r2
  - Case 2: I2 and I3 wrt r3
  - Case 3: I1 and I3 wrt r2
- Case 1:
  - Which is the earliest stage at which value of r2 is ready ? EXECUTE
  - Which is the latest by which I2 MUST receive updated r2 ?

Example of forwarding

- Consider the following example

```
I1: add r1, r2
I2: mrmovq d(r2), r3
I3: rmmovq r3, d(r2)
```

- Identify the data hazard
  - Case 1: I1 and I2 wrt r2
  - Case 2: I2 and I3 wrt r3
  - Case 3: I1 and I3 wrt r2
- Case 1:
  - Which is the earliest stage at which value of r2 is ready ? EXECUTE
  - Which is the latest by which I2 MUST receive updated r2 ? EXECUTE

Example of forwarding

- Consider the following example

```
I1: add r1, r2
I2: mrmovq d(r2), r3
I3: rmmovq r3, d(r2)
```

- Identify the data hazard
  - Case 1: I1 and I2 wrt r2
  - Case 2: I2 and I3 wrt r3
  - Case 3: I1 and I3 wrt r2
- Case 1:
  - Which is the earliest stage at which value of r2 is ready ? EXECUTE
  - Which is the latest by which I2 MUST receive updated r2 ? EXECUTE
  - Forward from EXECUTE stage of I1 to EXECUTE stage of I2

Example of forwarding from Recitation 6

- Consider the following example

```
I1: add r1, r2
I2: mrmovq d(r2), r3
I3: rmmovq r3, d(r2)
```

- Case 2:
  - Which is the earliest stage at which value of r3 is ready ?
  - Which is the latest by which I3 MUST receive updated r3 ?
- Case 3:
  - Which is the earliest stage at which value of r2 is ready ?
  - Which is the latest by which I3 MUST receive updated r2 ?

Example of forwarding from Recitation 6

- Consider the following example

```
I1: add r1, r2
I2: mrmovq d(r2), r3
I3: rmmovq r3, d(r2)
```

- Case 2:
  - Which is the earliest stage at which value of r3 is ready ? MEMORY
  - Which is the latest by which I3 MUST receive updated r3 ? MEMORY
  - Forward from MEMORY stage of I2 to MEMORY stage of I3
- Case 3:
  - Which is the earliest stage at which value of r2 is ready ? EXECUTE
  - Which is the latest by which I3 MUST receive updated r2 ? EXECUTE
  - Forward from EXECUTE stage of I1 to EXECUTE stage of I3
  - Also not late to forward from MEMORY stage of I1 to EXECUTE stage of I3

Caching Concepts

- **Dimensions: S, E, B**
  - S: Number of sets
  - E: Associativity – number of lines per set
  - B: Block size – number of bytes per block (1 block per line)
- **Given Values for S,E,B,m**
  - Find which address maps to which set
  - Is it a Hit/Miss? Is there an eviction?
  - Hit rate/Miss rate
- **Types of misses**
  - Which types can be avoided?
  - What cache parameters affect types/number of misses?

Questions/Advice

- Relax!
- Work on past exams
- Make a great cheat sheet
- Post questions on Piazza
- Come to office hours