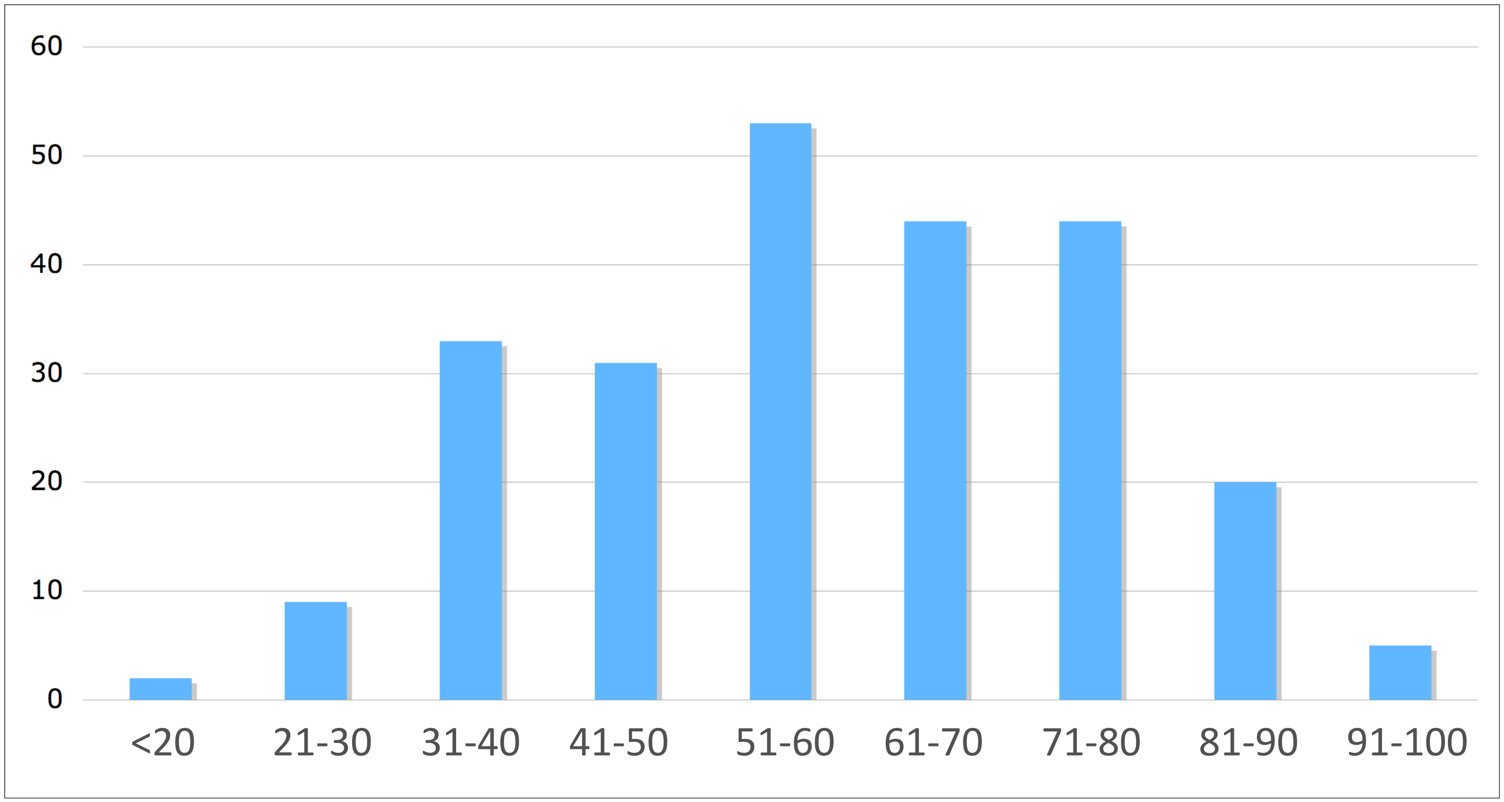
Introduction to Computer Systems

15-213/18-243, fall 2009
9th Lecture, Sep 29th

Instructors:

Roger B. Dannenberg and Greg Ganger

First Exam



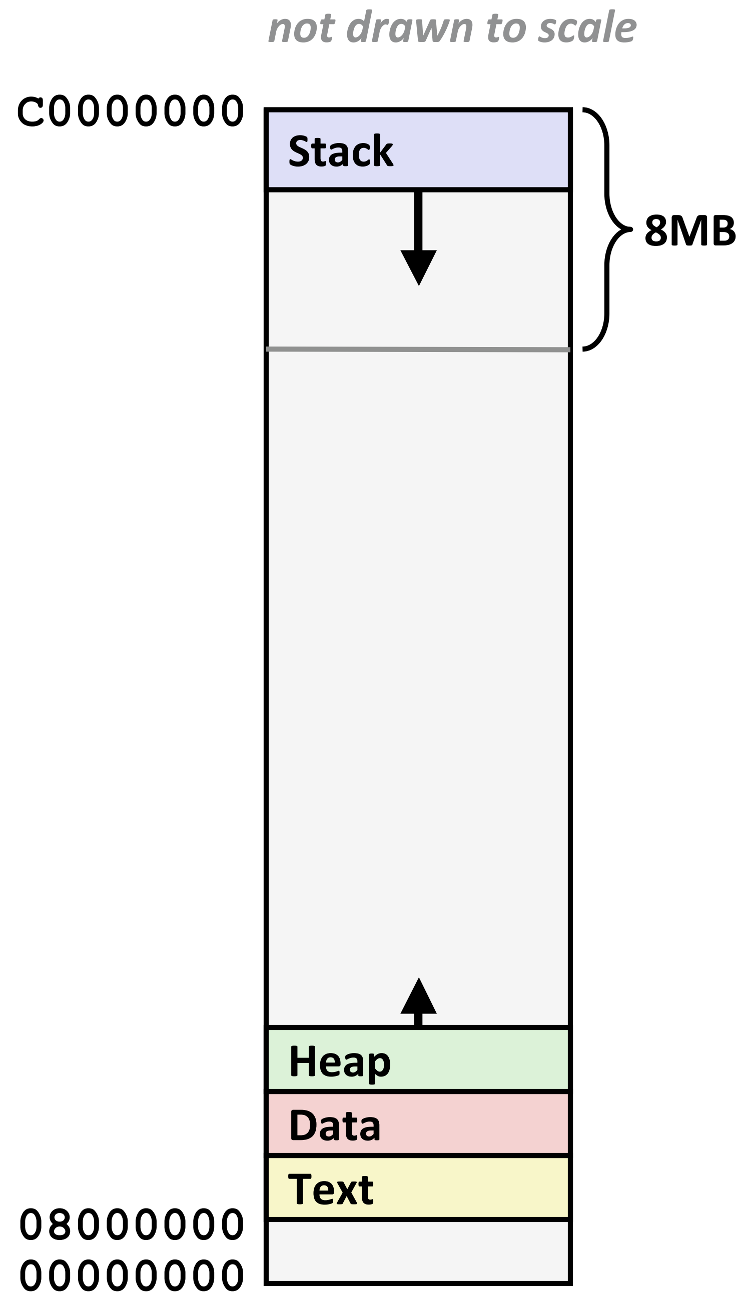
Exam1 Grade as *Percent*

Today

- Address Space at Run Time
- Linking

IA32 Linux Memory Layout

- **Stack**
  - Runtime stack (8MB limit)
- **Heap**
  - Dynamically allocated storage
  - When call `malloc()`, `calloc()`, `new()`
- **Data**
  - Statically allocated data
  - E.g., arrays & strings declared in code
- **Text**
  - Executable machine instructions
  - Read-only

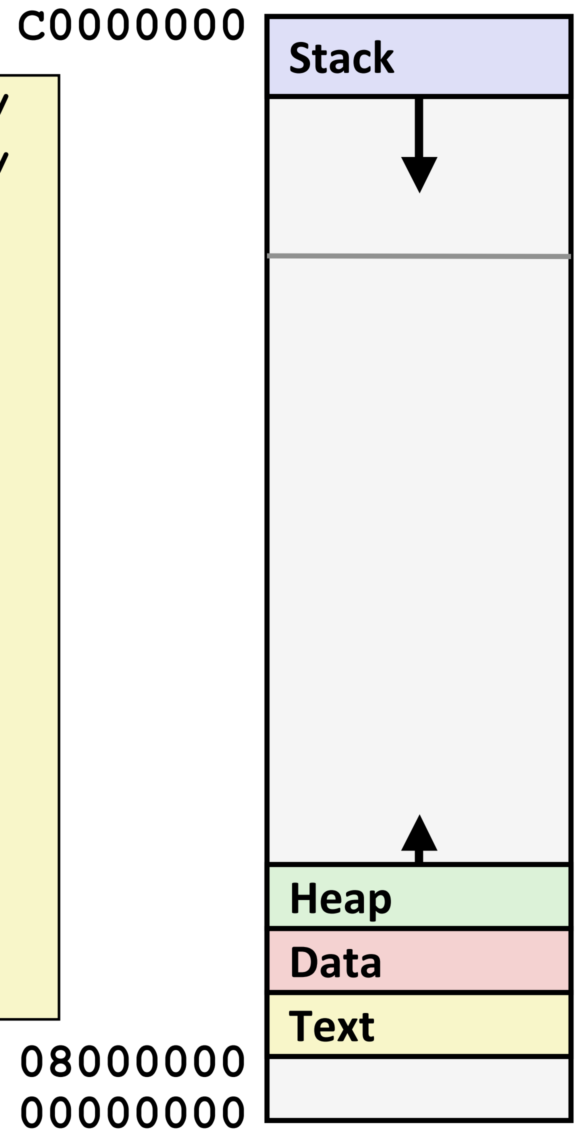


not drawn to scale

Memory Allocation Example

```
char big_array[1<<24];  /*  16 MB */
char huge_array[1<<28]; /* 256 MB */

int beyond;
char *p1, *p2, *p3, *p4;

int useless() {  return 0; }

int main()
{
 p1 = malloc(1 <<28);  /* 256 MB */
 p2 = malloc(1 << 8);  /* 256 B  */
 p3 = malloc(1 <<28);  /* 256 MB */
 p4 = malloc(1 << 8);  /* 256 B  */
 /* Some print statements ... */
}
```

Where does everything go?



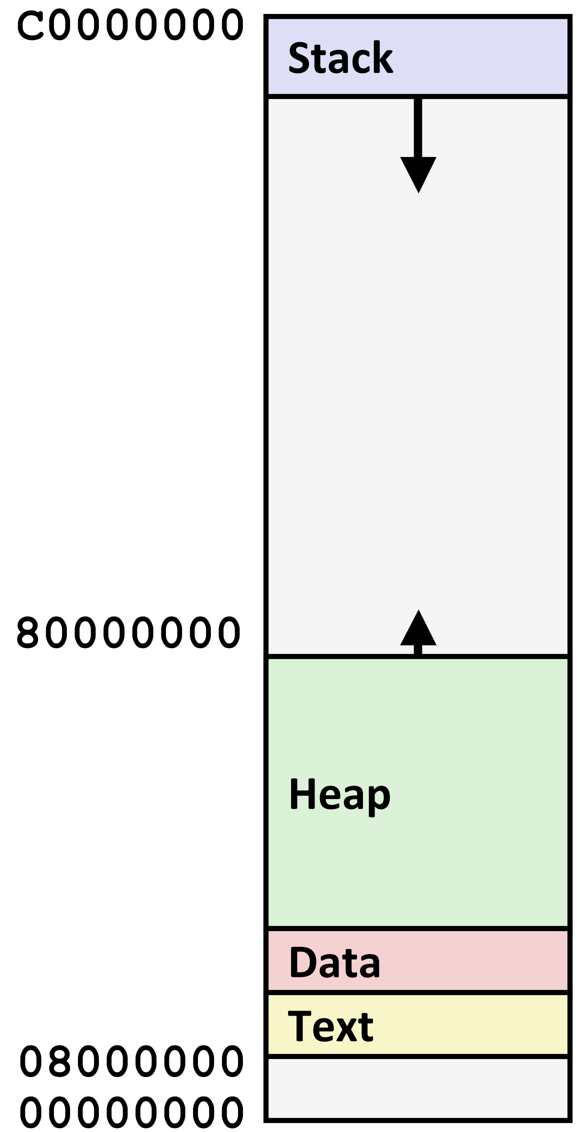
not drawn to scale

IA32 Example Addresses

address range ~2^{32}

`$esp`	`0xffffbcd0`
`p3`	`0x65586008`
`p1`	`0x55585008`
`p4`	`0x1904a110`
`p2`	`0x1904a008`
`&p2`	`0x18049760`
`beyond`	`0x08049744`
`big_array`	`0x18049780`
`huge_array`	`0x08049760`
`main()`	`0x080483c6`
`useless()`	`0x08049744`
final `malloc()`	`0x006be166`

`malloc()` is dynamically linked
address determined at runtime



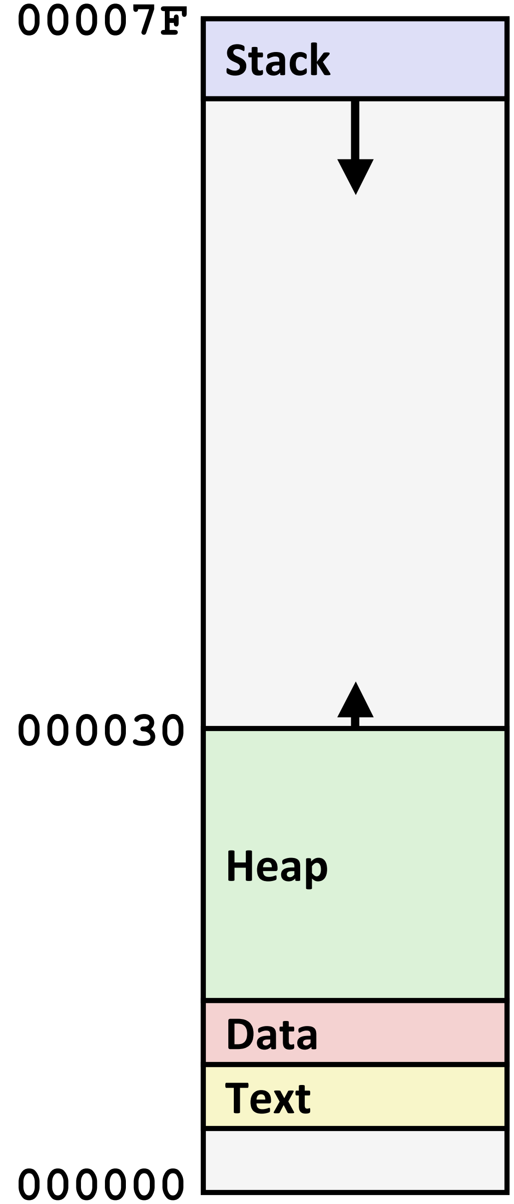
x86-64 Example Addresses

address range ~2^{47}

`$rsp`	`0x7fff,7576,0050`
`p1`	`0x2b7f,e3f7,1010`
`p3`	`0x2b7f,f3f7,2010`
`p4`	`0x0000,1150,1120`
`p2`	`0x0000,1150,1010`
`beyond`	`0x0000,1150,0a28`
`big_array`	`0x0000,1050,0a20`
`&p2`	`0x0000,1050,0a00`
`huge_array`	`0x0000,0050,0a00`
`main()`	`0x0000,0040,0510`
`useless()`	`0x0000,0040,0500`
final `malloc()`	`0x0038,6ae6,a170`

`malloc()` is dynamically linked
address determined at runtime

not drawn to scale



Example C Program

main.c

```c
int buf[2] = {1, 2};

int main()
{
   swap();
   return 0;
}
```

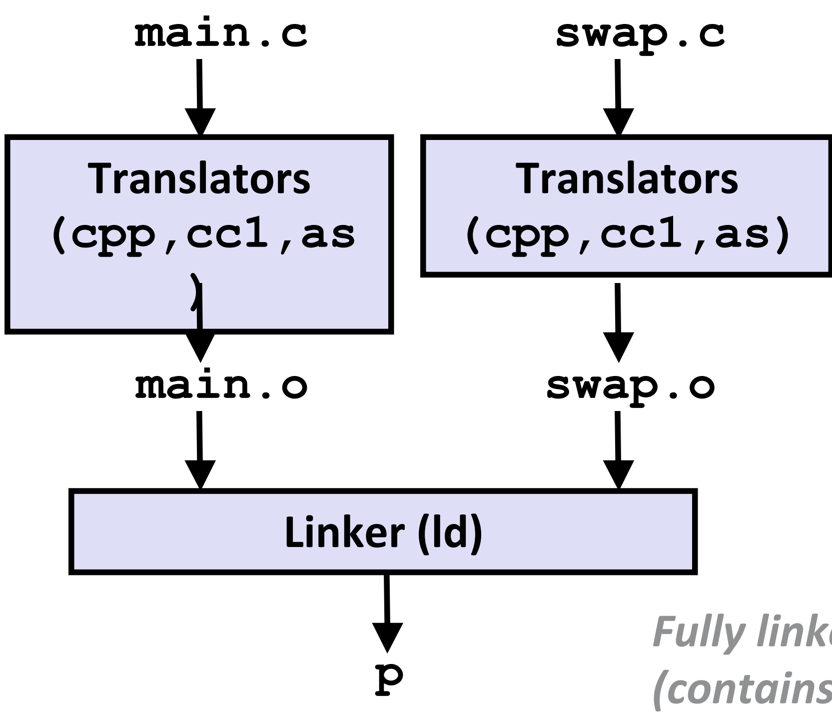
swap.c

```c
extern int buf[];

static int *bufp0 = &buf[0];
static int *bufp1;

void swap()
{
   int temp;

   bufp1 = &buf[1];
   temp = *bufp0;
   *bufp0 = *bufp1;
   *bufp1 = temp;
}
```

Static Linking

- **Programs are translated and linked using a *compiler driver*:**

```
unix> gcc -O2 -g -o p main.c swap.c
unix> ./p
```



Source files

Separately compiled relocatable object files

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

Why Linkers? 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? 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* (variables and functions):
    - `void swap() {…}    /* define symbol swap */`
    - `swap();            /* reference symbol swap */`
    - `int *xp = &x;      /* define xp, reference x */`
  - Symbol definitions are stored (by compiler) in *symbol table*.
    - Symbol table is an array of structs
    - Each entry includes name, type, size, and location of symbol.
  - 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.

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**
  - 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**
- **Originally proposed by AT&T System V Unix**
  - Later adopted by BSD Unix variants and Linux
- **One unified format for**
  - Relocatable object files (`.o`),
  - Executable object files
  - Shared object files (`.so`)

- **Generic name: ELF binaries**

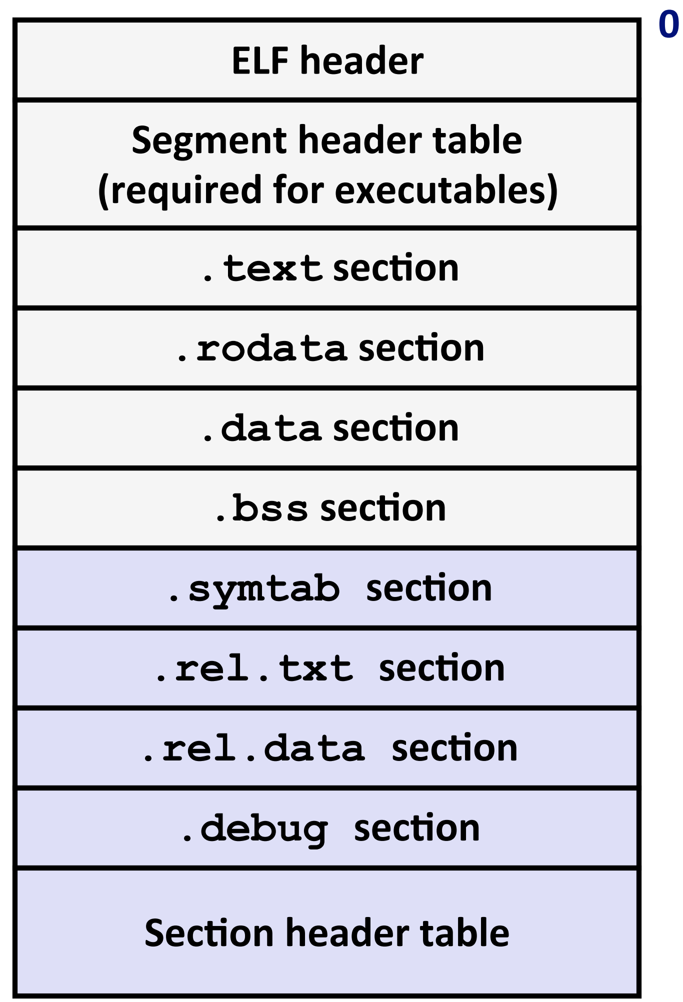
ELF Object File Format

- **Elf header**
  - Word size, byte ordering, file type (.o, exec, .so), machine type, etc.
- **Segment header table**
  - For executables: virtual address, segment size, alignments
- **`.text` section**
  - Code
- **`.rodata` section**
  - Read only data: 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

0
ELF header
Segment header table (required for executables)
`.text` section
`.rodata` section
`.data` section
`.bss` section
`.symtab` section
`.rel.txt` section
`.rel.data` section
`.debug` section
Section header table

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 variables defined with the `static` attribute.
  - **Local linker symbols are *not* local program variables**

Resolving Symbols

Global

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

int main()
{
   swap();
   return 0;
}
```

main.c

External

External   Local

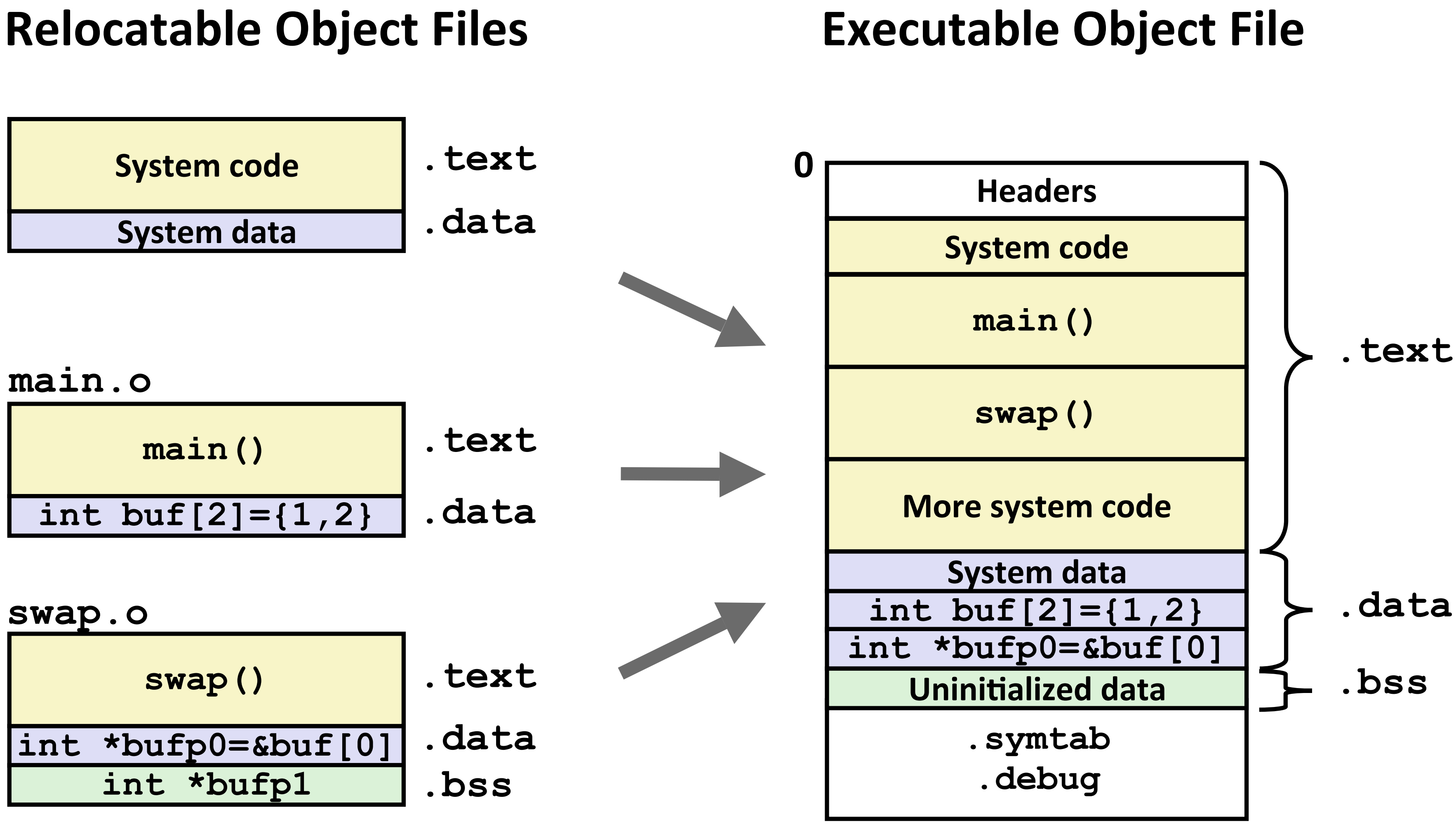
```
extern int buf[];

static int *bufp0 = &buf[0];
static int *bufp1;

void swap()
{
   int temp;

   bufp1 = &buf[1];
   temp = *bufp0;
   *bufp0 = *bufp1;
   *bufp1 = temp;
}
```

swap.c

Global

Linker knows nothing of temp

Relocating Code and Data



Relocation Info (main)

`main.c`

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

int main()
{
  swap();
  return 0;
}
```

`main.o`

```
0000000 <main>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   83 ec 08                sub    $0x8,%esp
   6:   e8 fc ff ff ff          call   7 <main+0x7>
                                7: R_386_PC32 swap
   b:   31 c0                   xor    %eax,%eax
   d:   89 ec                   mov    %ebp,%esp
   f:   5d                      pop    %ebp
  10:   c3                      ret
```

```
Disassembly of section .data:

 00000000 <buf>:
   0:   01 00 00 00 02 00 00 00
```

Source: `objdump`

Relocation Info (swap, .text)

swap.c

```
extern int buf[];

static int *bufp0 =
           &buf[0];
static int *bufp1;

void swap()
{
  int temp;

  bufp1 = &buf[1];
  temp = *bufp0;
  *bufp0 = *bufp1;
  *bufp1 = temp;
}
```

swap.o

```
Disassembly of section .text:

00000000 <swap>:
 0: 55                        push   %ebp
 1: 8b 15 00 00 00 00         mov    0x0,%edx
                              3: R_386_32 bufp0
 7: a1 0 00 00 00             mov    0x4,%eax
                              8: R_386_32 buf
 c: 89 e5                     mov    %esp,%ebp
 e: c7 05 00 00 00 00 04movl  $0x4,0x0
15: 00 00 00
                              10: R_386_32 bufp1
                              14: R_386_32 buf
18: 89 ec                     mov    %ebp,%esp
1a: 8b 0a                     mov    (%edx),%ecx
1c: 89 02                     mov    %eax,(%edx)
1e: a1 00 00 00 00            mov    0x0,%eax
                              1f: R_386_32 bufp1
23: 89 08                     mov    %ecx,(%eax)
25: 5d                        pop    %ebp
26: c3                        ret
```

Relocation Info (swap, .data)

swap.c

```
extern int buf[];

static int *bufp0 =
           &buf[0];
static int *bufp1;

void swap()
{
  int temp;

  bufp1 = &buf[1];
  temp = *bufp0;
  *bufp0 = *bufp1;
  *bufp1 = temp;
}
```

```
Disassembly of section .data:

00000000 <bufp0>:
   0:   00 00 00 00

        0: R_386_32 buf
```

Executable After Relocation (.text)

```
080483b4 <main>:
 80483b4:       55                      push   %ebp
 80483b5:       89 e5                   mov    %esp,%ebp
 80483b7:       83 ec 08                sub    $0x8,%esp
 80483ba:       e8 09 00 00 00          call   80483c8 <swap>
 80483bf:       31 c0                   xor    %eax,%eax
 80483c1:       89 ec                   mov    %ebp,%esp
 80483c3:       5d                      pop    %ebp
 80483c4:       c3                      ret
080483c8 <swap>:
 80483c8:       55                      push   %ebp
 80483c9:       8b 15 5c 94 04 08       mov    0x804945c,%edx
 80483cf:       a1 58 94 04 08          mov    0x8049458,%eax
 80483d4:       89 e5                   mov    %esp,%ebp
 80483d6:       c7 05 48 95 04 08 58    movl   $0x8049458,0x8049548
 80483dd:       94 04 08
 80483e0:       89 ec                   mov    %ebp,%esp
 80483e2:       8b 0a                   mov    (%edx),%ecx
 80483e4:       89 02                   mov    %eax,(%edx)
 80483e6:       a1 48 95 04 08          mov    0x8049548,%eax
 80483eb:       89 08                   mov    %ecx,(%eax)
 80483ed:       5d                      pop    %ebp
 80483ee:       c3                      ret
```

Executable After Relocation (.data)

```
Disassembly of section .data:

08049454 <buf>:
 8049454:       01 00 00 00 02 00 00 00

0804945c <bufp0>:
 804945c:       54 94 04 08
```

Strong and Weak Symbols

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

`p1.c`

strong → `int foo=5;`

strong → `p1() {`

```
int foo=5;

p1() {
}
```

`p2.c`

```
int foo;

p2() {
}
```

`int foo;` ← weak

`p2() {` ← strong

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 symbol, 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

```c
int x;
p1() {}
```

```c
p1() {}
```

Link time error: two strong symbols (**`p1`**)

```c
int x;
p1() {}
```

```c
int x;
p2() {}
```

References to **`x`** will refer to the same uninitialized int. Is this what you really want?

```c
int x;
int y;
p1() {}
```

```c
double x;
p2() {}
```

Writes to **`x`** in **`p2`** might overwrite **`y`**! Evil!

```c
int x=7;
int y=5;
p1() {}
```

```c
double x;
p2() {}
```

Writes to **`x`** in **`p2`** will overwrite **`y`**! Nasty!

```c
int x=7;
p1() {}
```

```c
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 use external global variable

Packaging Commonly Used Functions

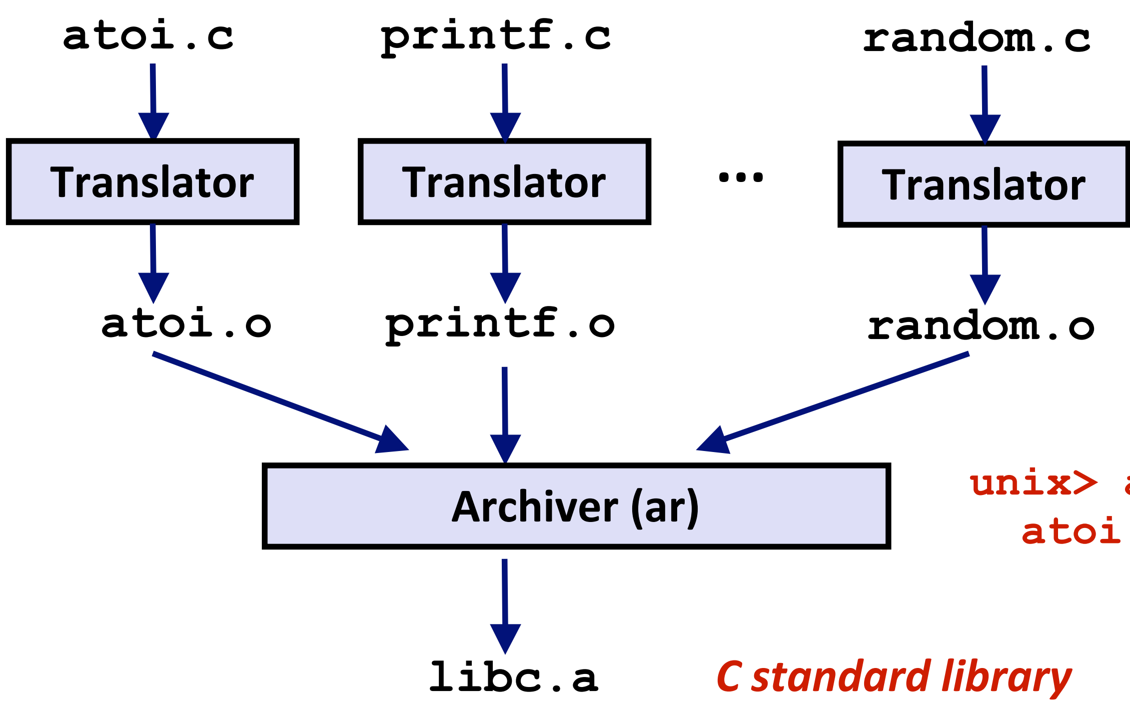
- **How to package functions commonly used by programmers?**
  - Math, I/O, memory management, string manipulation, etc.

- **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

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 into executable.

Creating Static Libraries




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

C standard library

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

Commonly Used Libraries

`libc.a` (the C standard library)

- 8 MB archive of 900 object files.
- I/O, memory allocation, signal handling, string handling, data and time, random numbers, integer math

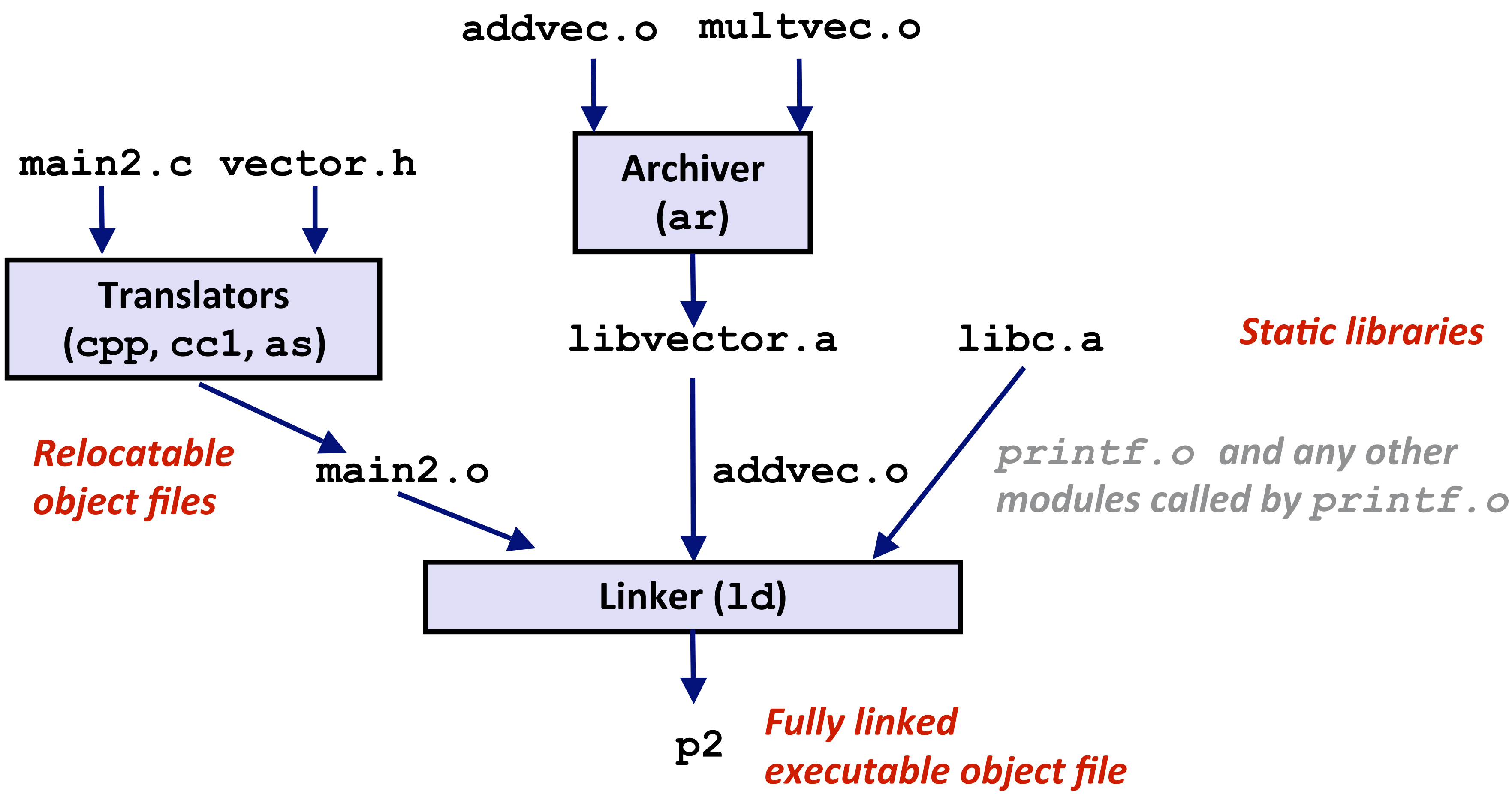
`libm.a` (the C math library)

- 1 MB archive of 226 object files.
- floating point math (sin, cos, tan, log, exp, sqrt, …)

```
% ar -t /usr/lib/libc.a | sort
…
fork.o
…
fprintf.o
fpu_control.o
fputc.o
freopen.o
fscanf.o
fseek.o
fstab.o
…
```

```
% ar -t /usr/lib/libm.a | sort
…
e_acos.o
e_acosf.o
e_acosh.o
e_acoshf.o
e_acoshl.o
e_acosl.o
e_asin.o
e_asinf.o
e_asinl.o
…
```

Linking with Static Libraries



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.

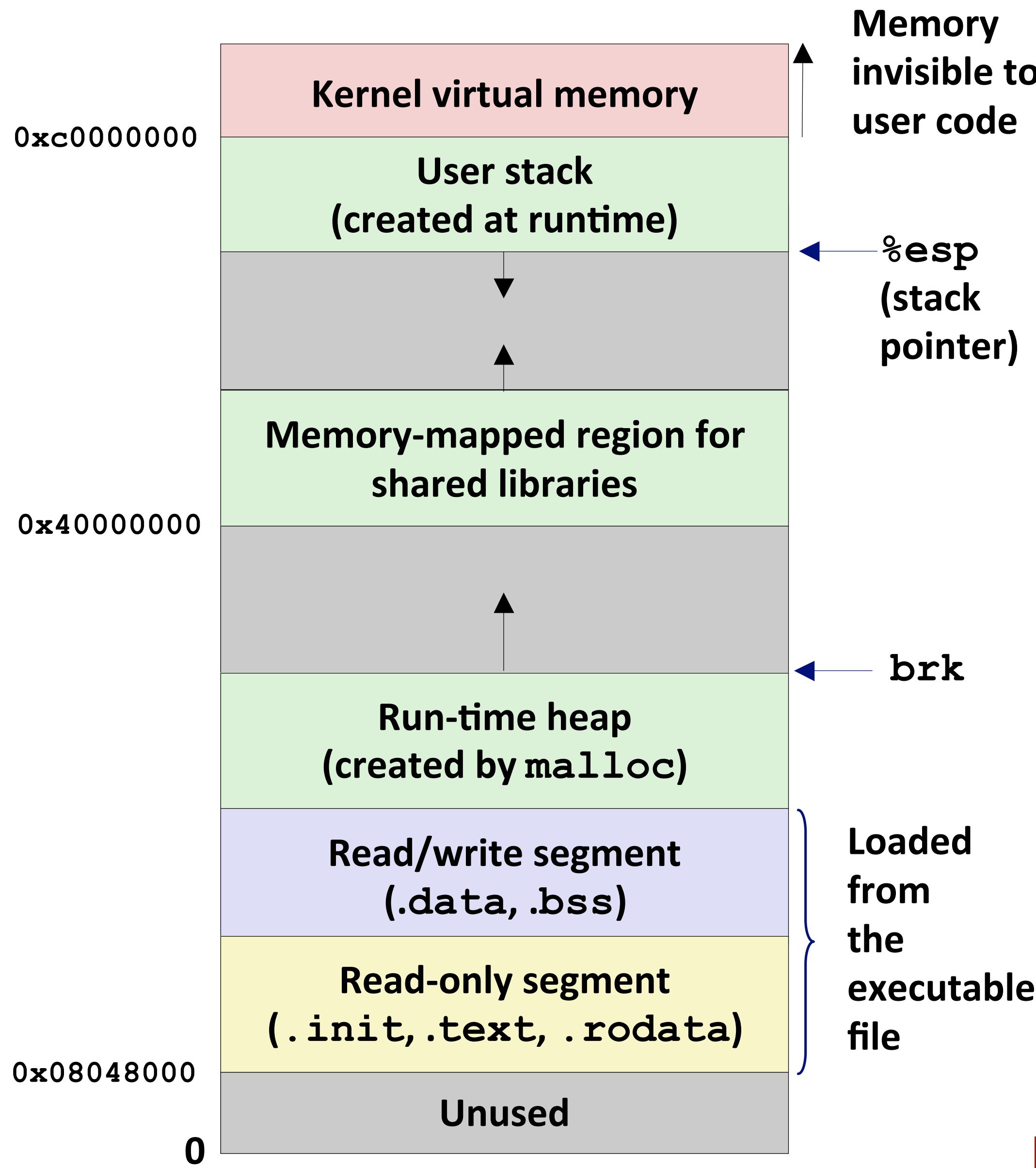
- **Problem:**
  - Command line order matters!
  - Moral: put libraries at the end of the command line.

```
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'
```

Loading Executable Object Files

Executable Object File

0
ELF header
Program header table (required for executables)
.init section
.text section
.rodata section
.data section
.bss section
.symtab
.debug
.line
.strtab
Section header table (required for relocatables)



Shared Libraries

- **Static libraries have the following disadvantages:**
  - Duplication in the stored executables (every function need std 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

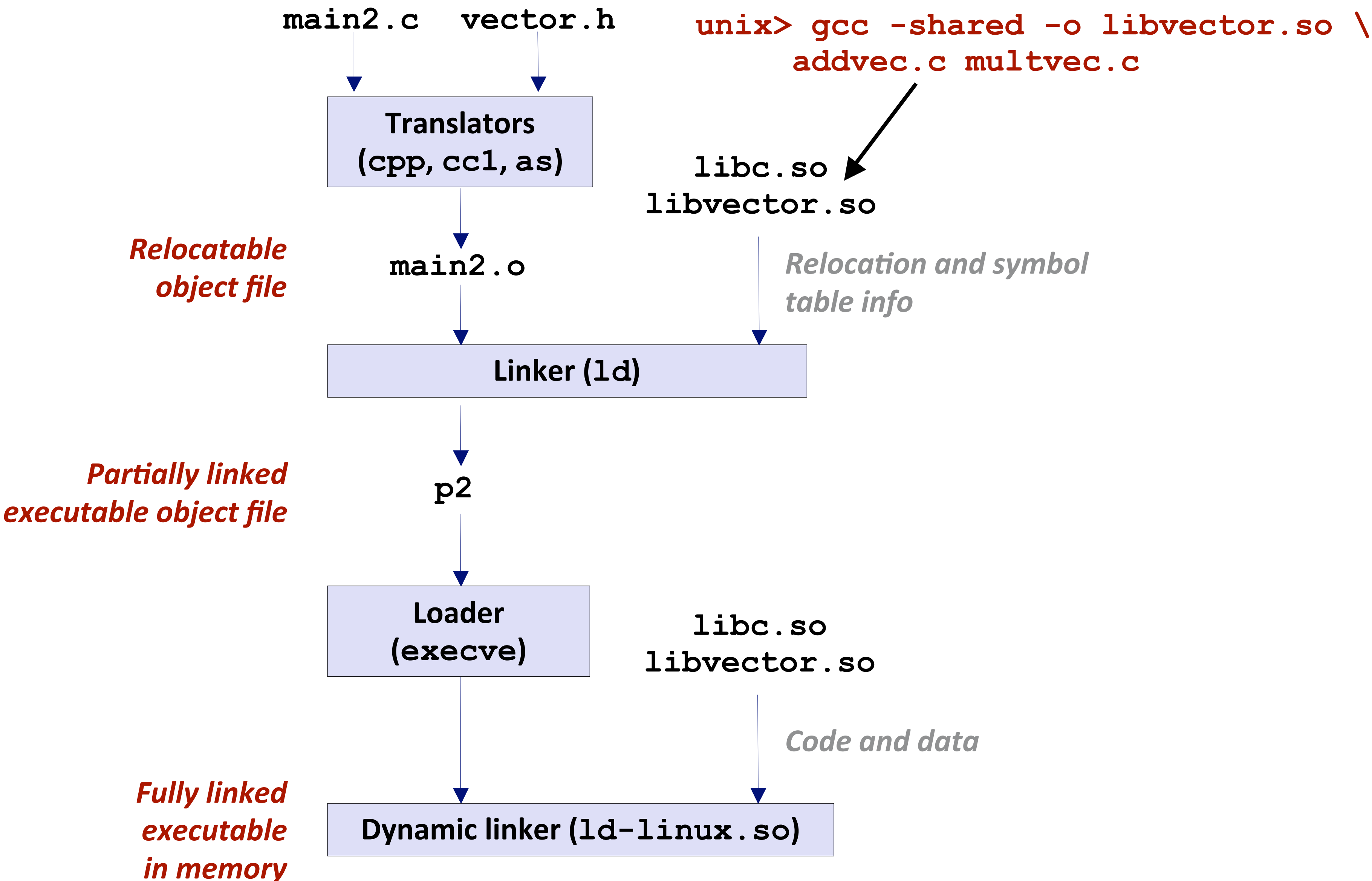
Shared Libraries (cont.)

- **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 Unix, this is done by calls to the **`dlopen()`** interface.
    - 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 Runtime

```
#include <stdio.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 lib that contains addvec() */
    handle = dlopen("./libvector.so", RTLD_LAZY);
    if (!handle) {
            fprintf(stderr, "%s\n", dlerror());
            exit(1);
    }
```

Dynamic Linking at Run-time

```
    ...

    /* 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() it 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;
}
```

Case Study: Library Interpositioning

Library interpositioning is a powerful linking technique that allows programmers to intercept calls to arbitrary functions

Interpositioning can occur at:

- compile time
  - When the source code is compiled
- link time
  - When the relocatable object files are linked to form an executable object file
- load/run time
  - When an executable object file is loaded into memory, dynamically linked, and then executed.

Some Interpositioning Applications

Security

- Confinement (sandboxing)
  - Interpose calls to libc functions.
- Behind the scenes encryption
  - Automatically encrypt otherwise unencrypted network connections.

Monitoring and Profiling

- Count number of calls to functions
- Characterize call sites and arguments to functions
- Malloc tracing
  - Detecting memory leaks
  - Generating malloc traces

Example: `malloc()` Statistics

Count how much memory is allocated by a function

```
void *malloc(size_t size){
    static void *(*fp)(size_t) = 0;
    void *mp;
    char *errorstr;

    /* Get a pointer to the real malloc() */
    if (!fp) {
        fp = dlsym(RTLD_NEXT, "malloc");
        if ((errorstr = dlerror()) != NULL) {
            fprintf(stderr, "%s(): %s\n", fname, errorstr);
            exit(1);
        }
    }

    /* Call the real malloc function */
    mp = fp(size);

    mem_used += size;

    return mp;
}
```

Summary

- **ELF files contain**
  - Object files
  - Libraries
  - Executables
- **Linking**
- **Loading**
- **Dynamic Linking**
- **Details:**
  - How are globals, externals, static symbols handled?
  - How are names searched and resolved by linkers?
  - How can you interpose your own library implementation?