

Lecture 9: Static and Dynamic Linking

Instructor: Mitch Neilsen
Office: N219D

Outline

- Reading:
 - Ch. 1-7 - Process/Thread Management - Quiz #1 – 10/9
 - Ch. 8 - Memory Management
- Homework 2: Due 10/2
- Project 1: Scheduling/Synchronization: **New Due Date 10/7**
 - Alarm Clock (finish this week)
 - Priority-based Scheduler (finish this week)
 - Synchronization and Priority Inheritance (start this week)
 - [Extra Credit] MLFQ Scheduler

Quote of the Day

"You cannot depend on your eyes when your imagination is out of focus."

- Mark Twain

Chapter 8: Memory Management

- **Background**
- **Linking**
- Swapping
- Contiguous Memory Allocation
- Paging
- Structure of the Page Table
- Segmentation
- Example: The Intel Pentium

Chapter Objectives

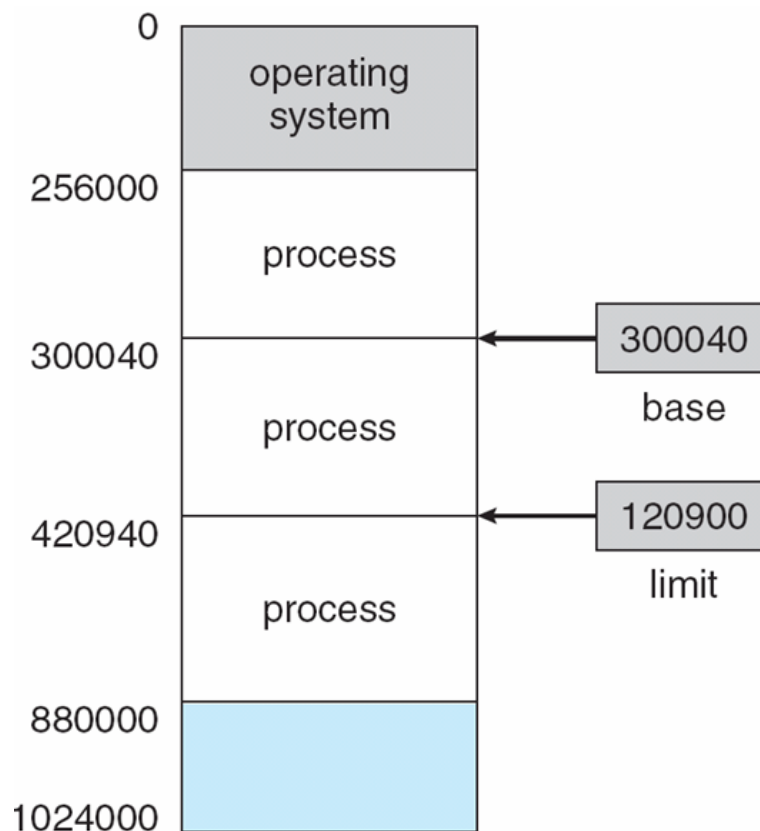
- To provide a detailed description of various ways of organizing memory hardware
- To discuss various memory-management techniques, including paging and segmentation
- To provide a detailed description of the Intel Pentium, which supports both pure segmentation and segmentation with paging

Background

- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Register access in one CPU clock cycle (or less)
- Main memory can take many cycles
- **Cache** sits between main memory and CPU registers
- Protection of memory and cache consistency is required to ensure correct operation

Base and Limit Registers

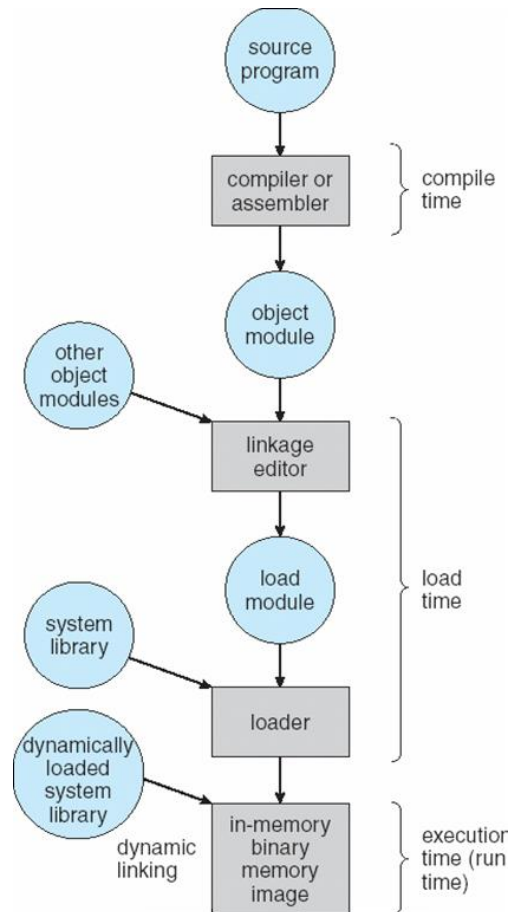
- A pair of **base** and **limit** registers define the logical address space



Binding of Instructions and Data to Memory

- Address binding of instructions and data to memory addresses can happen at three different stages
 - **Compile time:** If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
 - **Load time:** Must generate **relocatable code** if memory location is not known at compile time
 - **Execution time:** Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., base and limit registers)

Multistep Processing of a User Program



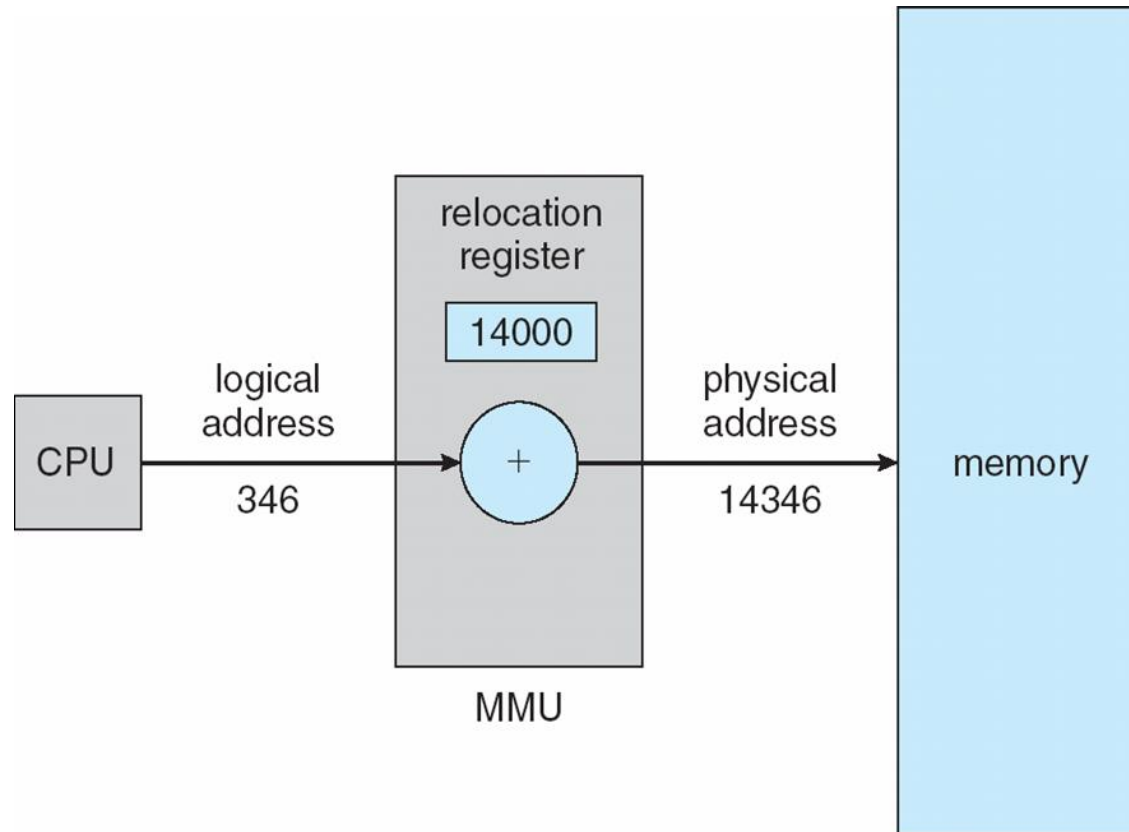
Logical vs. Physical Address Space

- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
 - **Logical address** – generated by the CPU; also referred to as **virtual address**
 - **Physical address** – address seen by the memory unit
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme

Memory-Management Unit (MMU)

- Hardware device that maps virtual to physical address
- In MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
- The user program deals with *logical* addresses; it never sees the *real* physical addresses

Dynamic relocation using a relocation register



Dynamic Loading

- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded
- Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required implemented through program design

Dynamic Linking

- Linking postponed until execution time
- Small piece of code, *stub*, used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and executes the routine
- Operating system needed to check if routine is in processes' memory address
- Dynamic linking is particularly useful for libraries
- System also known as **shared object libraries (.so)** in Linux or **dynamic link libraries (.dll)** in Windows

Static vs. Dynamic Linking

- Static linking
- Dynamic linking
- Case study: Library interpositioning

Example C Program

main.c

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

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

swap.c

```
extern int buf[];

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

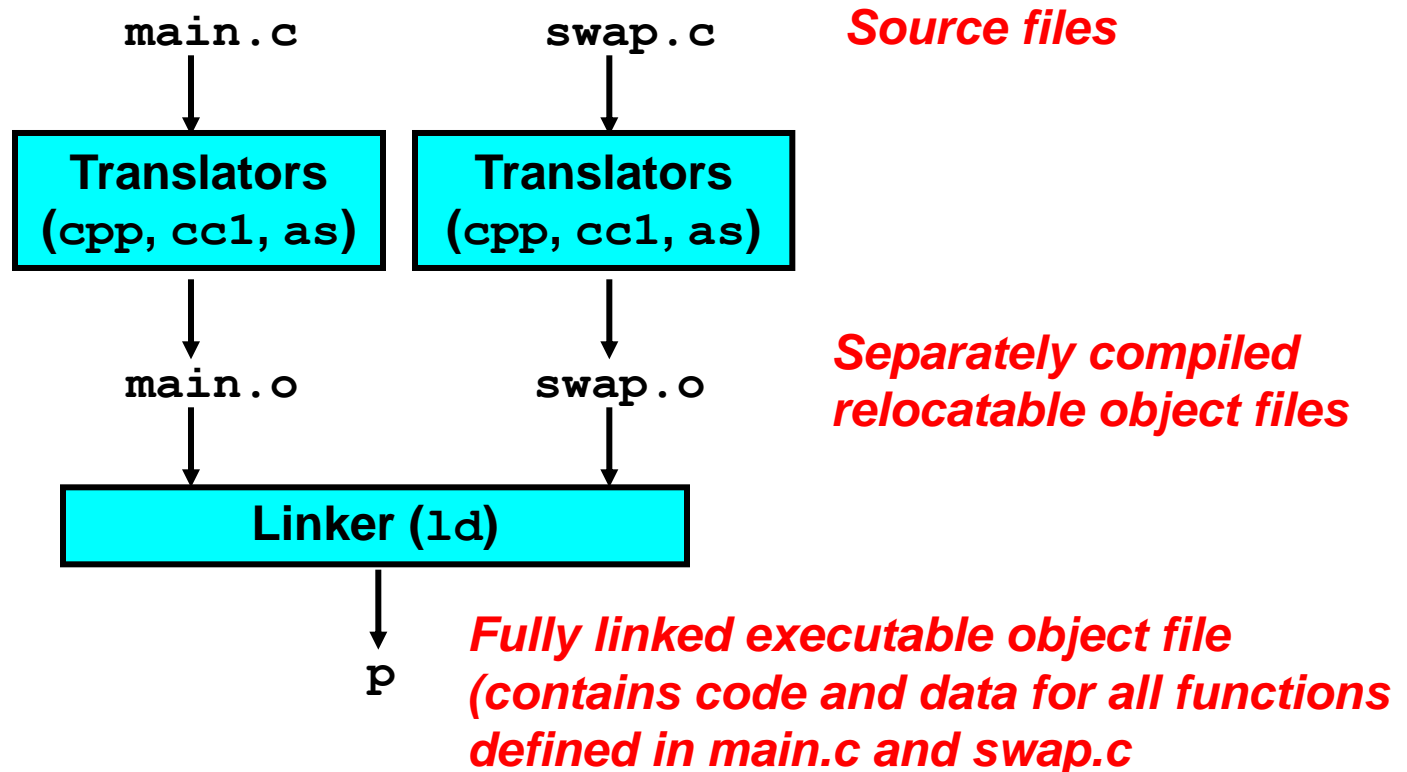
void swap()
{
    int temp;

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


Static Linking

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

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



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

- 1. 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
- 2. Executable object file
 - Contains code and data in a form that can be copied directly into memory and then executed.
- 3. 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)*** in Windows Op. Systems

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
 - Magic number, type (.o, exec, .so), machine, byte ordering, etc.
- Segment header table
 - Page size, virtual addresses memory segments (sections), segment sizes.
- .text section
 - Code
- .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 header
Segment header table (required for executables)
.text 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

ELF header
Segment header table (required for executables)
<code>.text</code> section
<code>.data</code> section
<code>.bss</code> section
<code>.symtab</code> section
<code>.rel.text</code> section
<code>.rel.data</code> section
<code>.debug</code> section
Section header table

Linker Symbols

- Global symbols
 - Symbols defined by module *m* that can be referenced by other modules.
 - Ex: `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*.
 - Ex: C functions and variables defined with the `static` attribute.
- Key Point: Local linker symbols are *not* local program variables

Resolving Symbols

Def of global
symbol buf

main.c

```
int buf[2] = {1,2};  
  
int main()  
{  
    swap();  
    return 0;  
}
```

Ref to external
symbol swap

Def of local
symbol bufp0

Def of global
symbol swap

swap.c

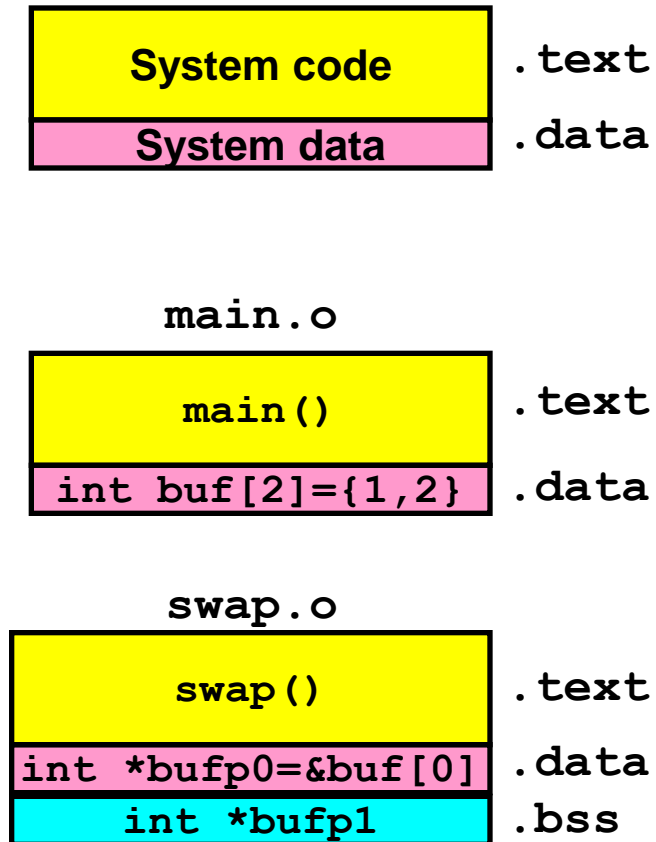
```
extern int buf[];  
  
static int *bufp0 = &buf[0];  
static int *bufp1;  
  
void swap()  
{  
    int temp;  
  
    bufp1 = &buf[1];  
    temp = *bufp0;  
    *bufp0 = *bufp1;  
    *bufp1 = temp;  
}
```

Ref to external
symbol buf

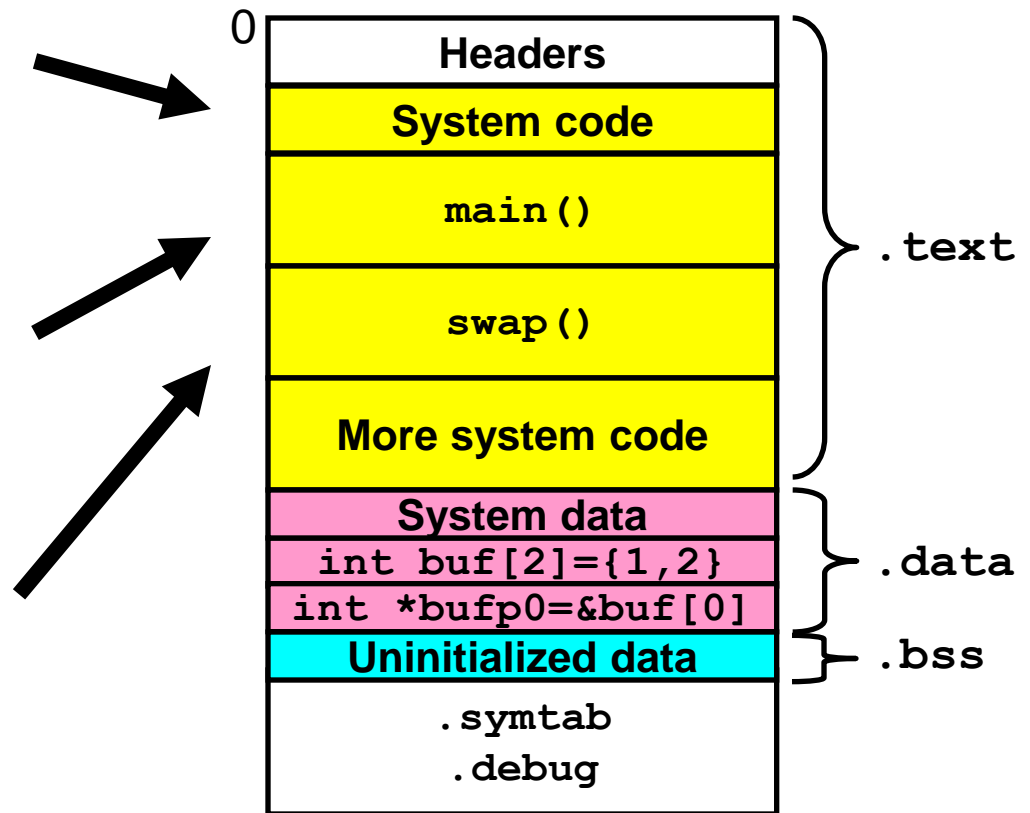
Linker knows
nothing of temp

Relocating Code and Data

Relocatable Object Files



Executable Object File



main.o Relocation Info

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

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

```
00000000 <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
```

swap.o Relocation Info (.text)

```
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 .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 04	movl	\$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	

a.o Relocation Info (.data)

```
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() {  
        }  
}
```

p2.c

```
int foo; ← weak  
p2() ← { ← strong  
}
```

Linker's Symbol Rules

- Rule 1. A strong symbol can only appear once.
- Rule 2. A weak symbol can be overridden by a strong symbol of the same name.
 - references to the weak symbol resolve to the strong symbol.
- Rule 3. If there are multiple weak symbols, the linker will 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.

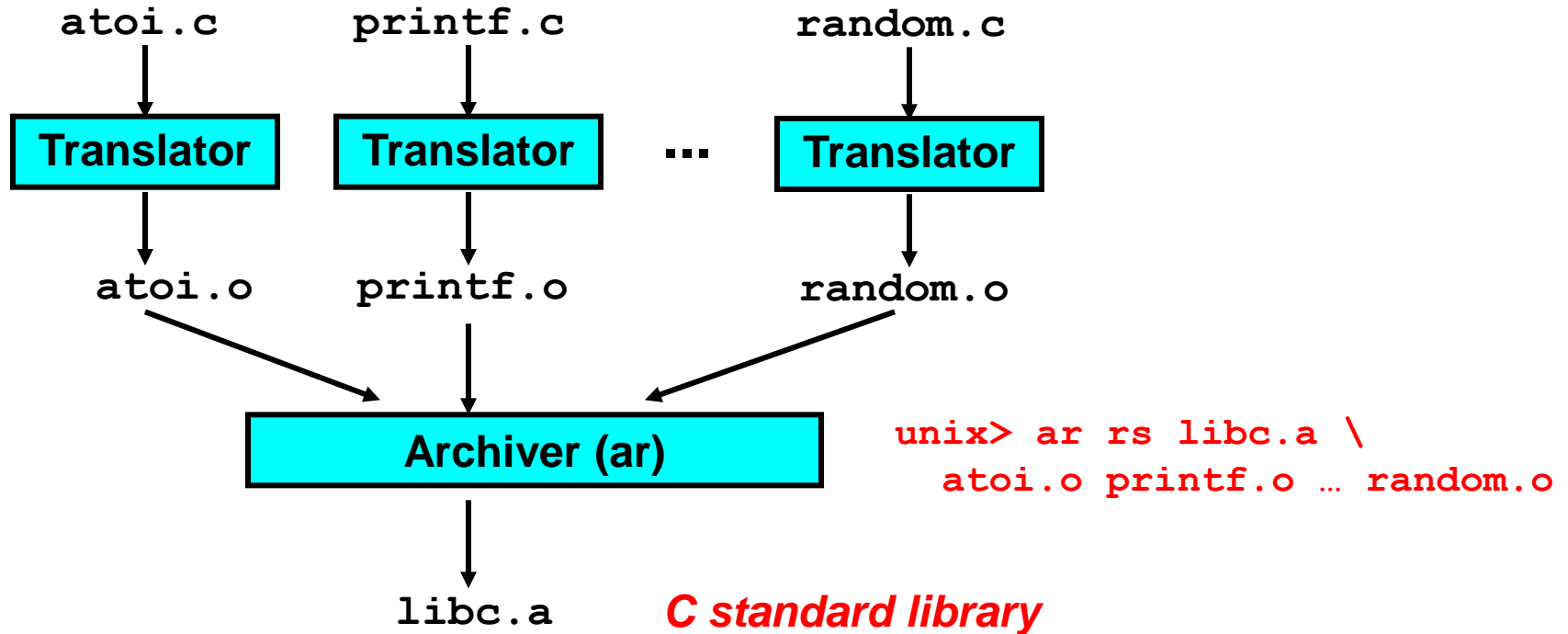
Packaging Commonly Used Functions

- 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 in 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

Static Libraries

- Solution: ***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



Archiver allows incremental updates:

- Recompile function that changes and replace .o file in archive.

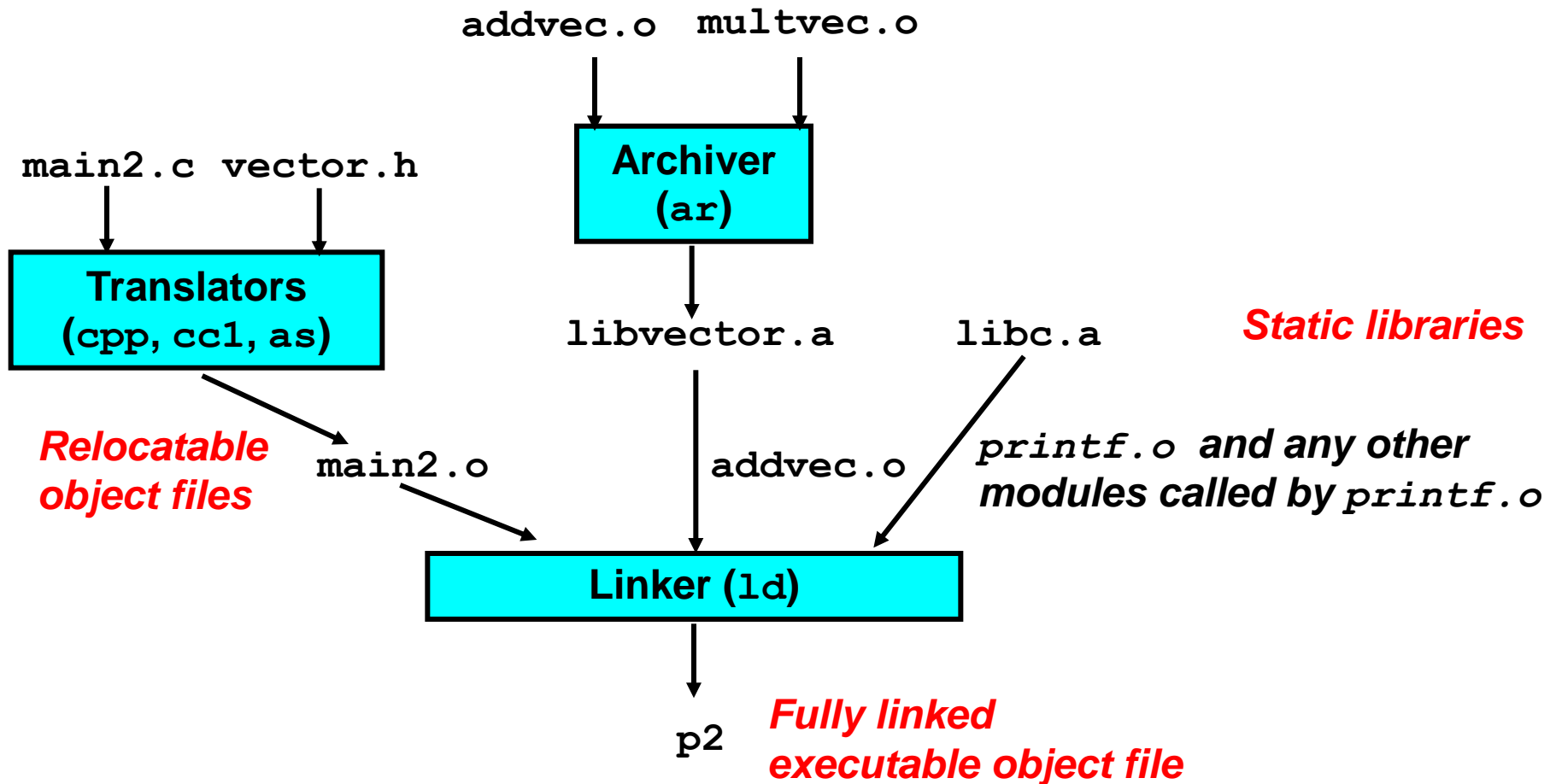
Commonly Used Libraries

- **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.

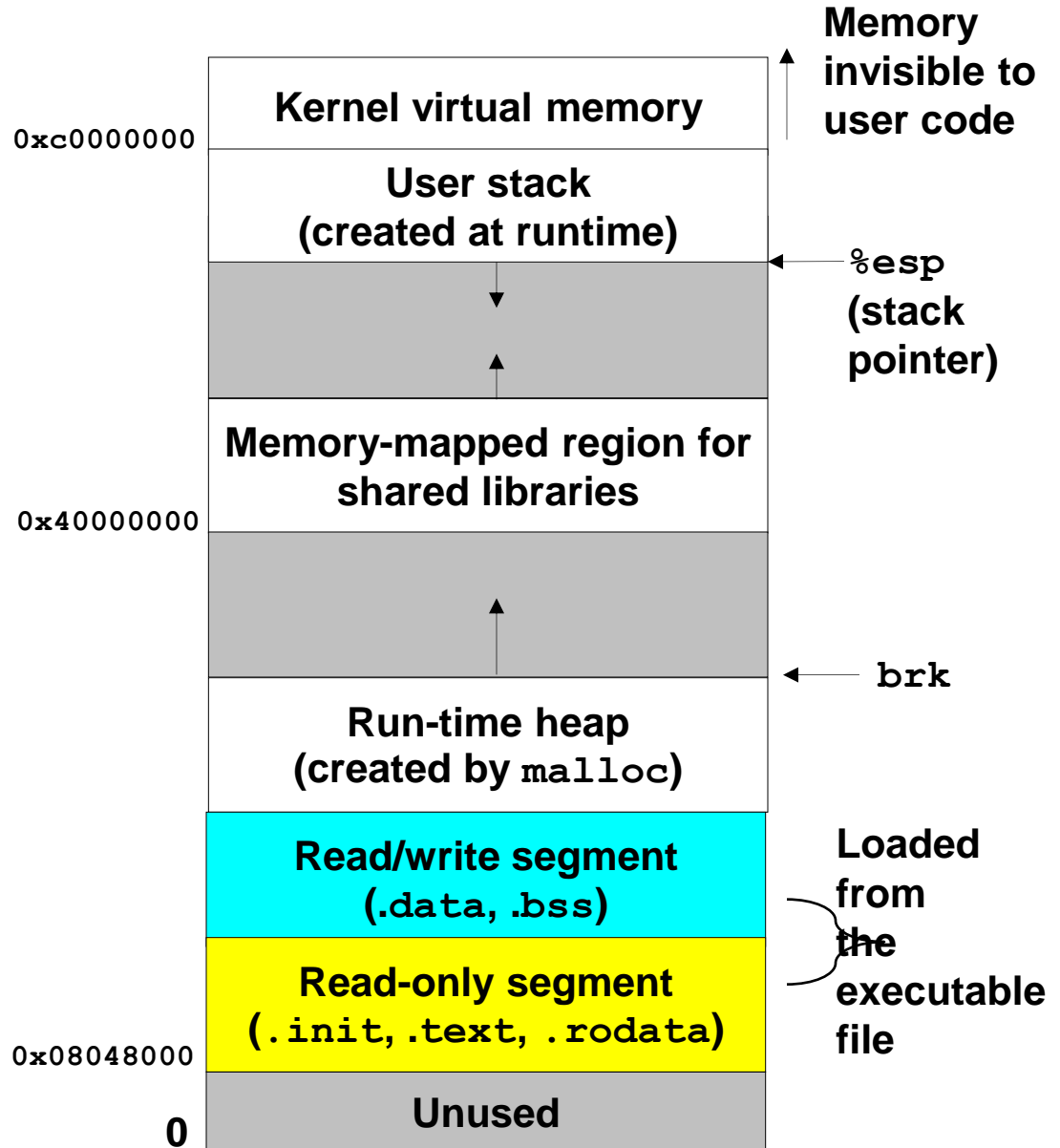
```
bass> gcc -L. libtest.o -lmine
bass> gcc -L. -lmine libtest.o
libtest.o: In function `main':
libtest.o(.text+0x4): undefined reference to `libfun'
```

Loading Executable Object Files

Executable Object File

ELF header
Program header table (required for executables)
.text section
.data section
.bss section
.symtab
.rel.text
.rel.data
.debug
Section header table (required for relocatables)

0



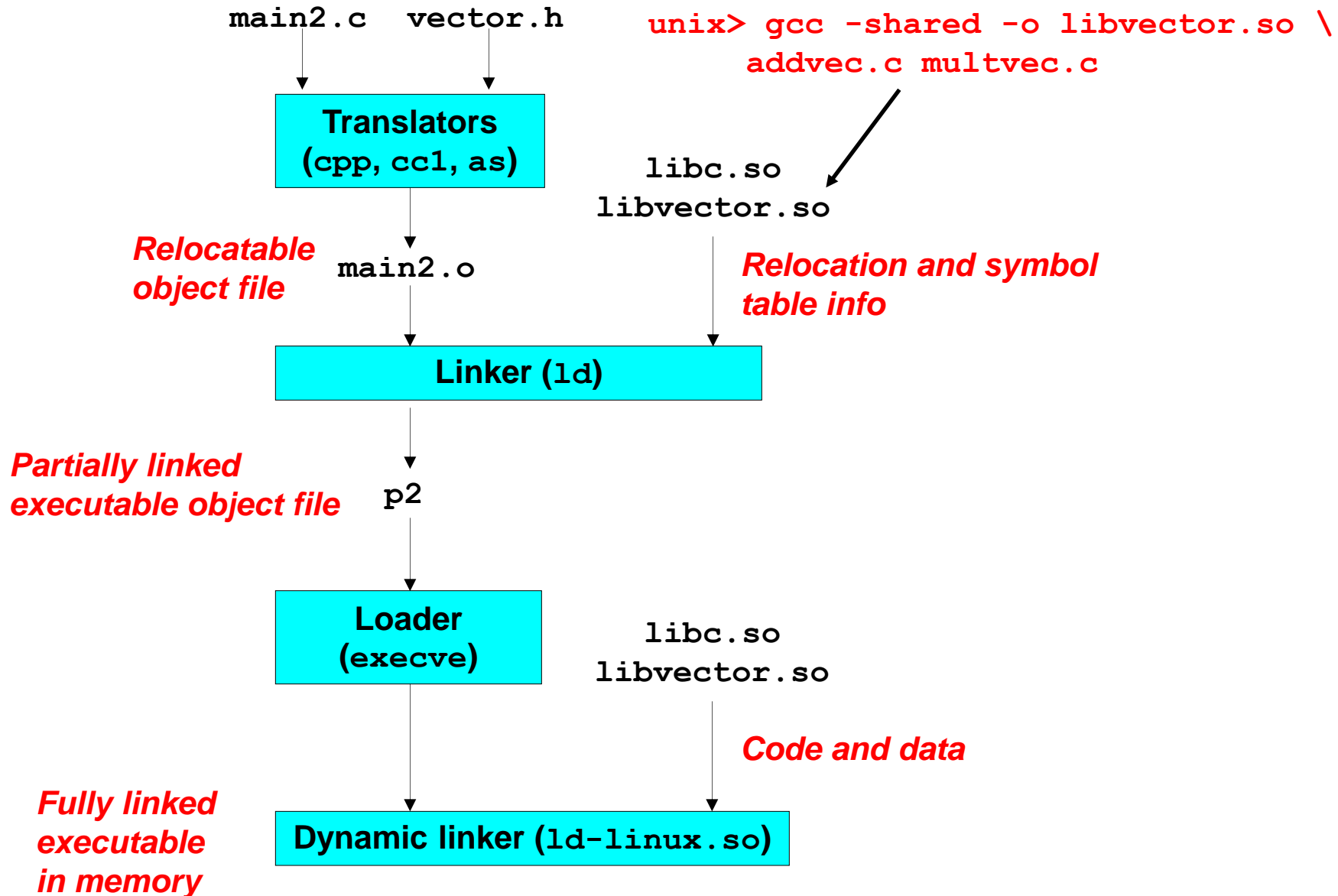
Shared Libraries

- Static libraries have the following disadvantages:
 - Potential for duplicating lots of common code in the executable files on a filesystem.
 - e.g., every C program needs the standard C library
 - Potential for duplicating lots of code in the virtual memory space of many processes.
 - 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 Run-time

```
#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
 - Detect memory leaks
 - **Generate 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;
}
```

Example: /pub/cis520/programs/malloc

- `$ make`
- `$ make runc` → `./helloc` – compile-time version
- `$ make runl` → `./hellol` – link-time version
- `$ make runr` → `(LD_PRELOAD="/usr/lib/libdl.so ./mymalloc.so" ./hellor)` – run-time version

Summary

- Read Ch. 1-8
- Processes and Threads (Ch. 4)
- Process Scheduling (Ch. 5)
- Synchronization (Ch. 6)
- Deadlock (Ch. 7)
- Memory Management (Ch. 8)
- Project 1 – Scheduling and Synchronization