Computer Systems Principles

Linking and Loading

UMASSCS
SCHOOL OF COMPUTER SCIENCE

Learning Objectives

- Learn about the linker and its uses
- Learn to identify types of linking
- Learn about the creation and contents of an Executable and Linkable (ELF) file
- Learn how symbol resolution is performed inside the linker
- Learn how linking is performed in Static libraries
- Learn how linking is performed in Dynamic libraries

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
    - e.g., Math library, standard C library, etc.

Why Linkers?

- Efficiency
  - Time: Separate compilation
    - Change one source file, compile, and then re-link.
    - 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 code only for the functions they actually use.

Example C Program

main.c

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

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

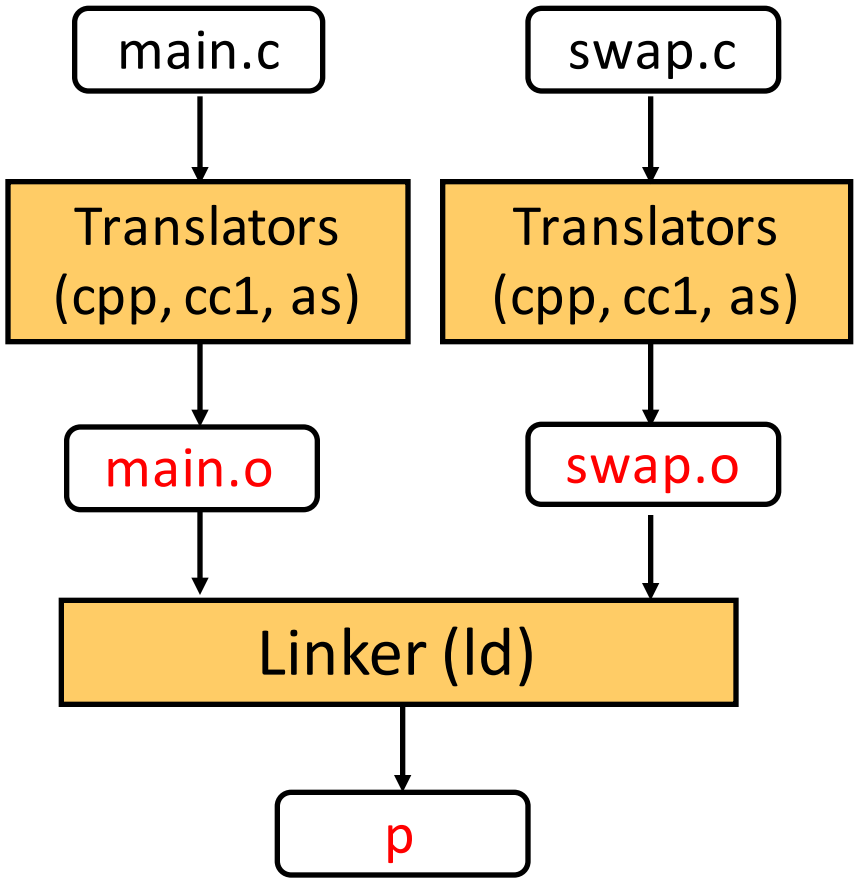
swap.c

```
extern int buf[];

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:

- > gcc main.c swap.c –o p
- > ./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)

What Do Linkers Do?

- Symbol Resolution [step 1]
  - 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 a part called the symbol table.
    - The symbol table is an array of structs
    - Each struct entry includes name, size, and location of symbol.
  - Linker associates each symbol reference with exactly one symbol definition.

What Do Linkers Do?

- Relocation [step2]
  - 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)

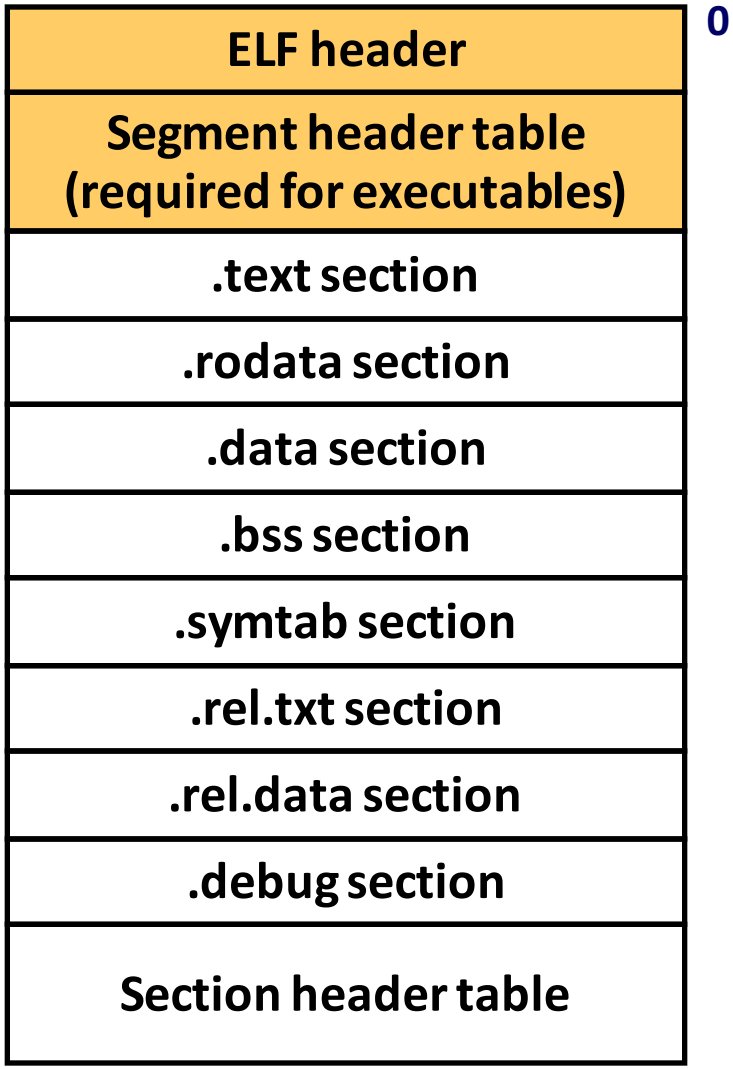
- Relocatable object file (.o file)
  - Contains code and data in a form that can be combined with other relocatable object files to form executable object file.
  - Each .o file is produced from exactly one source .c file
- Executable object file (a.out file)
  - Contains code and data in a form that 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

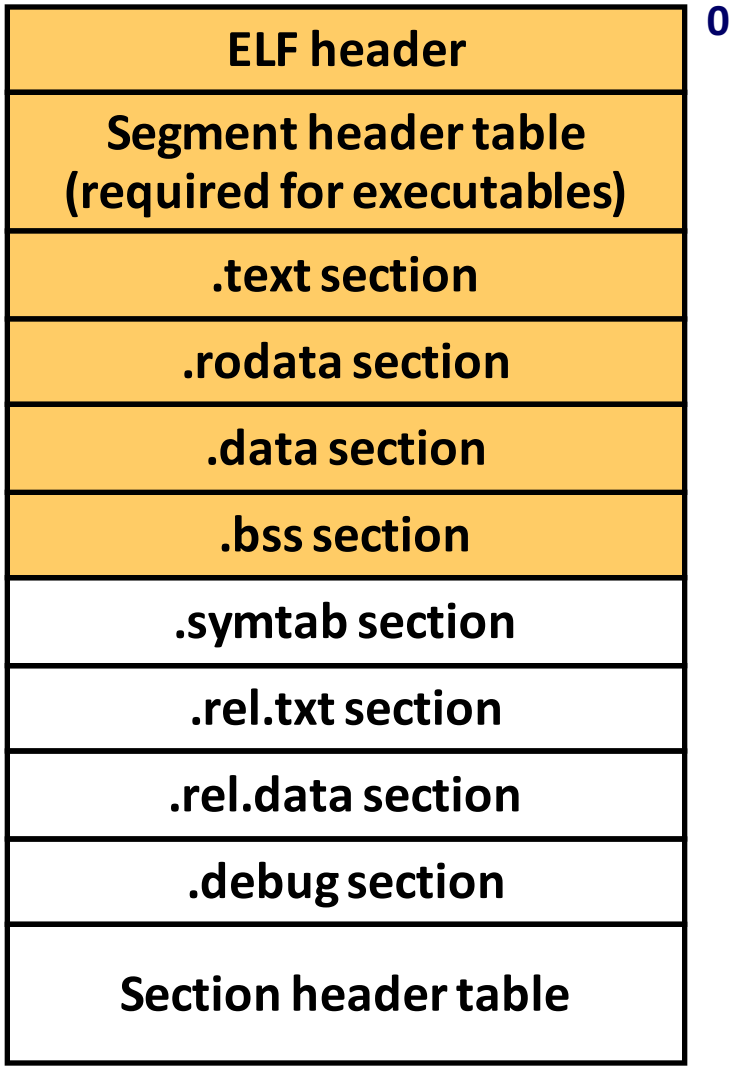
ELF Object File Format

- ELF header
  - Word size, byte ordering, file type (.o, exec, .so), machine type, etc.
- Segment header table
  - Page size, virtual addresses of memory segments (sections), segment sizes.



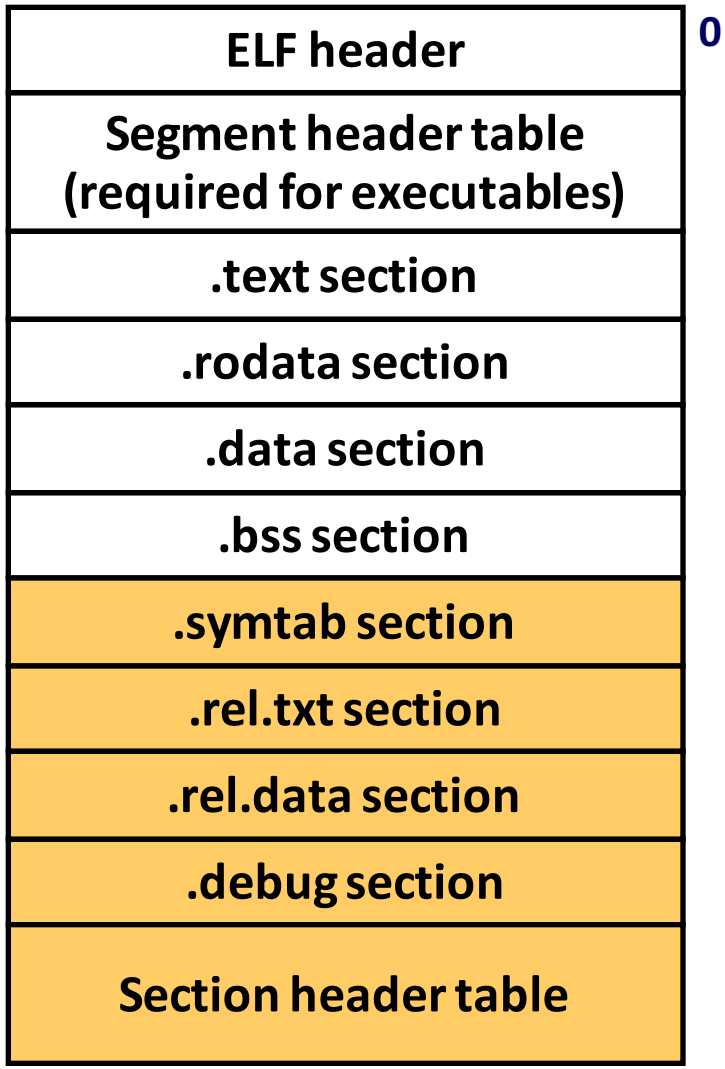
ELF Object File Format

- .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 in the ELF file (space used at run time)



ELF Object File Format

- .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
  - Directions for modifying them
- .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

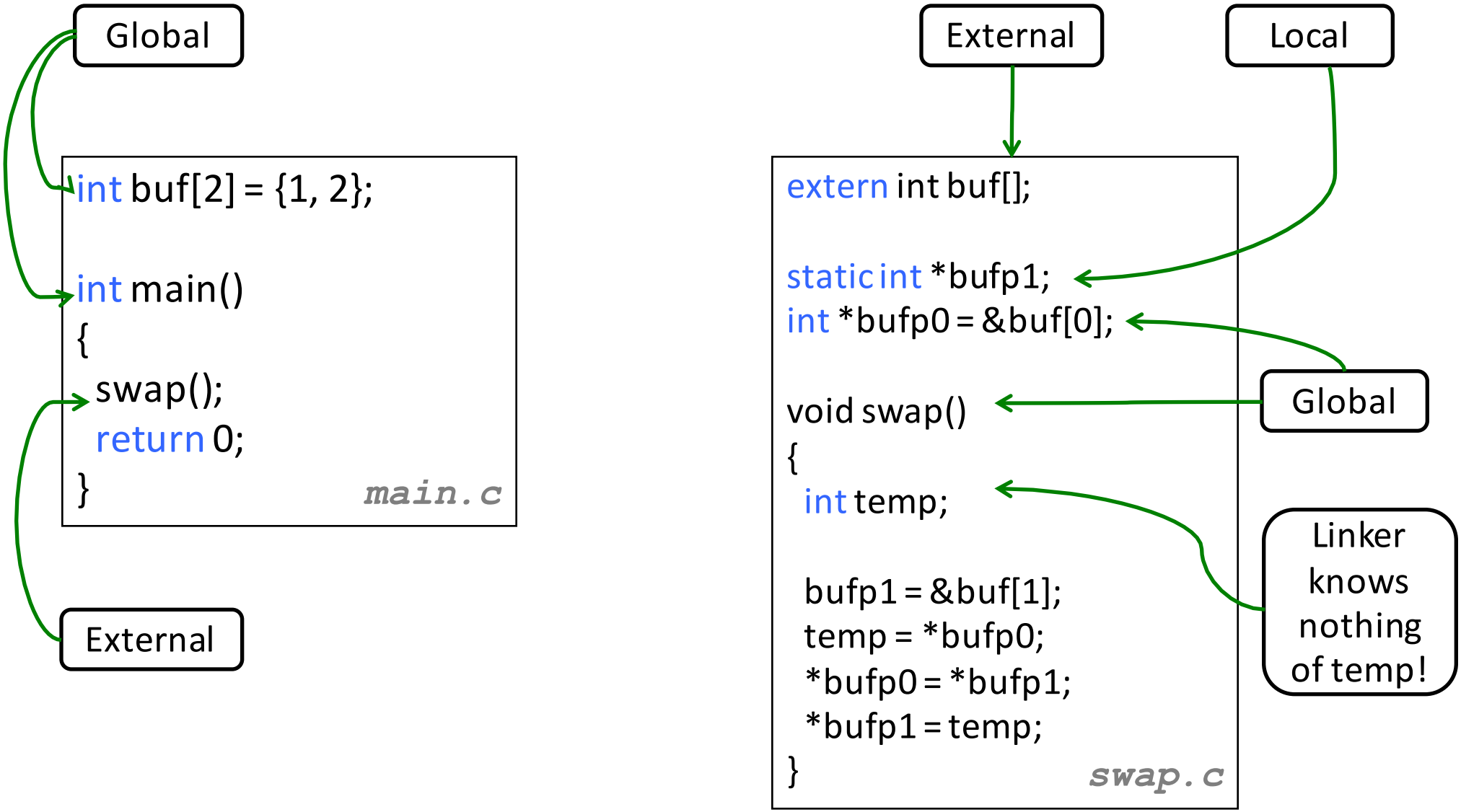


Linker Symbols

For relocatable object module m:

- 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 the same as local program variables!

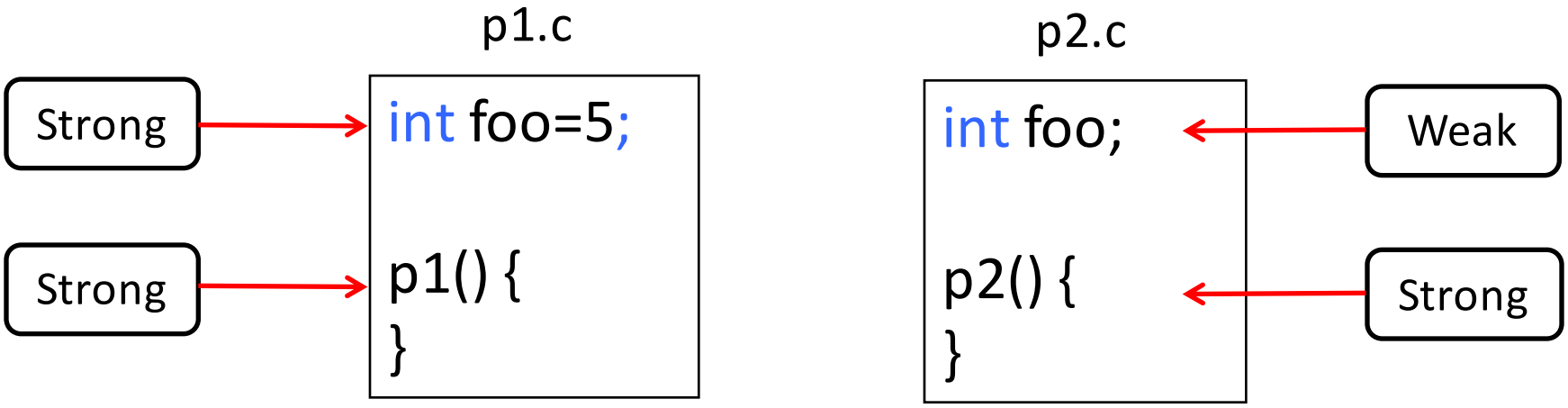
Symbol Resolution



Question: How do linkers resolve duplicate names?

Strong and Weak Symbols

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



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 symbols resolve to the strong symbol
- Rule 3: If there are multiple weak symbols, pick an arbitrary weak symbol
  - 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.

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

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

Writes to x in p2 might overwrite y!

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

```
double x=44;
p2() {}
```

Writes to x in p2 will overwrite y!

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

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

References to x will refer to the same initialized variable.

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

iClicker question

Consider alpha.c and beta.c: What will print?

```
int x;
int main () {
  x = 5;
  beta();
  printf("%d\n", x);
}
```

```
static int x;
void beta () {
  x = 10;
  return;
}
```

A) 5

B) 10

C) Something else

iClicker question

Consider alpha.c and beta.c: What will print?

```
int x;
int main () {
  x = 5;
  beta();
  printf("%d\n", x);
}
```

```
static int x;
void beta () {
  x = 10;
  return;
}
```

A) 5

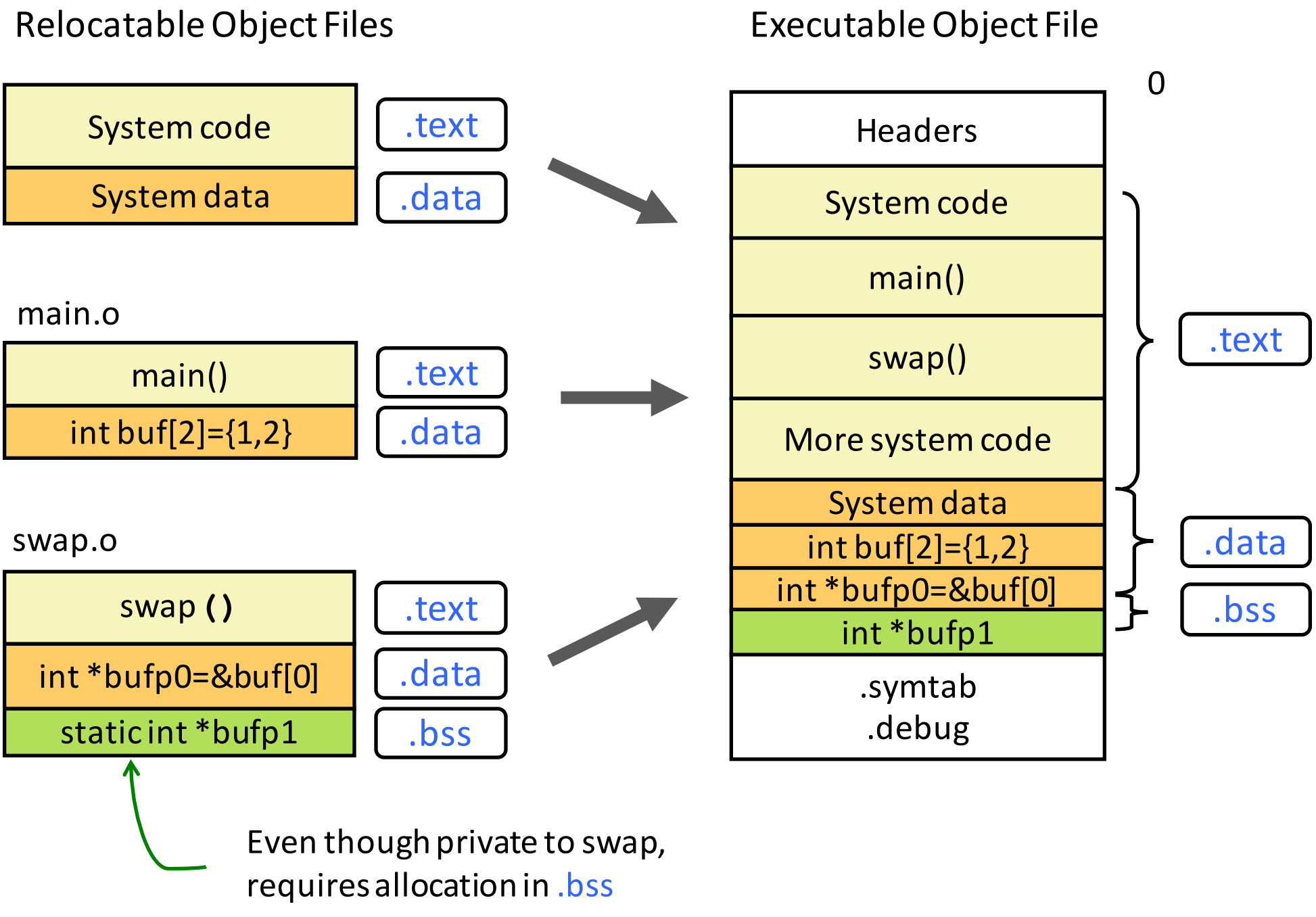
B) 10

C) Something else

Global Variables

- Avoid them if possible
- Otherwise
  - Use static when possible
  - Initialize when a global variable is defined
  - Use extern for an external global variable

Step 2: Relocation



Relocation Info (main)

main.c

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

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

```
 0:   8d 4c 24 04          lea     0x4(%esp),%ecx
 4:   83 e4 f0             and     $0xfffffff0,%esp
 7:   ff 71 fc             pushl   0xfffffffc(%ecx)
 a:   55                   push    %ebp
 b:   89 e5                mov     %esp,%ebp
 d:   51                   push    %ecx
 e:   83 ec 04             sub     $0x4,%esp
11:   e8 fc ff ff ff       call    12 <main+0x12>
                  12: R_386_PC32  swap
16:   b8 00 00 00 00       mov     $0x0,%eax
1b:   83 c4 04             add     $0x4,%esp
1e:   59                   pop     %ecx
1f:   5d                   pop     %ebp
20:   8d 61 fc             lea     0xfffffffc(%ecx),%esp
23:   c3                   ret
```

main.o

-4

Source: objdump –r –d main.o

Disassembly of section .data: Source: objdump -j .data –d main.o

Executable Before/After Relocation (.text)

```
00000000 <main>:
...
   e:      83 ec 04          sub    $0x4, %esp
  11:      e8 fc ff ff ff    call   12 <main+0x12>
                             12: R_386_PC32  swap
  16:   b8 00 00 00 00       mov    $0x0, %eax
 ...
```

Link time :
0x8048398
+ (-4)
- 0x8048386
= 0xe

```
08048374 <main>:
 8048374:      8d 4c 24 04          lea     0x4(%esp),%ecx
 8048378:      83 e4 f0             and     $0xfffffff0,%esp
 804837b:      ff 71 fc             pushl   0xfffffffc(%ecx)
 804837e:      55                   push    %ebp
 804837f:      89 e5                mov     %esp,%ebp
 8048381:      51                   push    %ecx
 8048382:      83 ec 04             sub     $0x4,%esp
 8048385:      e8 0e 00 00 00       call    8048398 <swap>
 804838a:      b8 00 00 00 00       mov     $0x0,%eax
 804838f:      83 c4 04             add     $0x4,%esp
 8048392:      59                   pop     %ecx
 8048393:      5d                   pop     %ebp
 8048394:      8d 61 fc             lea     0xfffffffc(%ecx),%esp
 8048397:      c3                   ret
```

Runtime:
0x804838a +
0xe
=
0x8048398

Relocation Info (swap, .text)

swap.c

```
extern int buf[];

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

void swap()
{
  int temp;

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

swap.o

```
00000000 <swap>:
 0:   55                         push   %ebp
 1:   89 e5                      mov    %esp,%ebp
 3:   53                         push   %ebx
 4:   c7 05 00 00 00 00 04       movl   $0x4,0x0
 b:   00 00 00
                        6: R_386_32  .bss
                        a: R_386_32  buf
 e:   8b 0d 00 00 00 00          mov    0x0,%ecx
                10: R_386_32  bufp0
14:   8b 19                      mov    (%ecx),%ebx
16:   ba 04 00 00 00             mov    $0x4,%edx
                17: R_386_32  buf
1b:   8b 02                      mov    (%edx),%eax
1d:   89 01                      mov    %eax,(%ecx)
1f:   89 1a                      mov    %ebx,(%edx)
21:   5b                         pop    %ebx
22:   5d                         pop    %ebp
23:   c3                         ret
```

Relocation Info (swap, .data)

swap.c

```
extern int buf[];

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

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
```

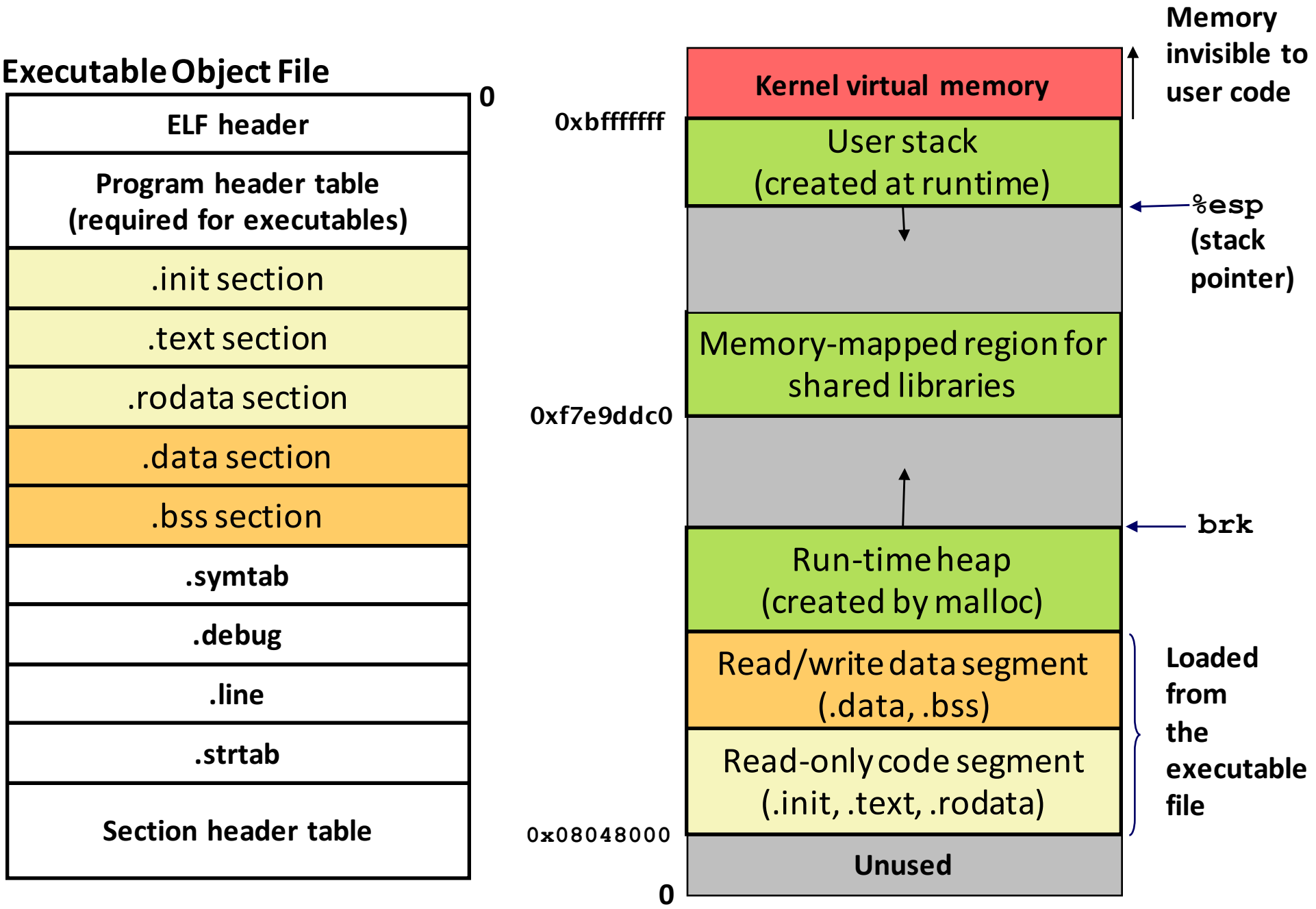
Before relocation

```
00000000 <swap>:
...
 4:   c7 05 00 00 00 00 04          movl   $0x4,0x0
 b:   00 00 00
                            6: R_386_32  .bss
                            a: R_386_32  buf
 e:   8b 0d 00 00 00 00             mov    0x0,%ecx
                      10: R_386_32  bufp0
14:   8b 19                        mov    (%ecx),%ebx
16:   ba 04 00 00 00               mov    $0x4,%edx
                      17: R_386_32  buf
```

After relocation

```
08048398 <swap>:
 8048398:      55                          push   %ebp
 8048399:      89 e5                       mov    %esp,%ebp
 804839b:      53                          push   %ebx
 804839c:      c7 05 14 96 04 08 04        movl   $0x8049604,0x8049614
 80483a3:      96 04 08
 80483a6:      8b 0d 08 96 04 08           mov    0x8049608,%ecx
 80483ac:      8b 19                       mov    (%ecx),%ebx
 80483ae:      ba 04 96 04 08              mov    $0x8049604,%edx
 80483b3:      8b 02                       mov    (%edx),%eax
 80483b5:      89 01                       mov    %eax,(%ecx)
 80483b7:      89 1a                       mov    %ebx,(%edx)
 80483b9:      5b                          pop    %ebx
 80483ba:      5d                          pop    %ebp
 80483bb:      c3                          ret
```

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

Commonly Used Libraries

libc.a (the C standard library)

- 8 MB archive of 1392 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 401 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
…
```

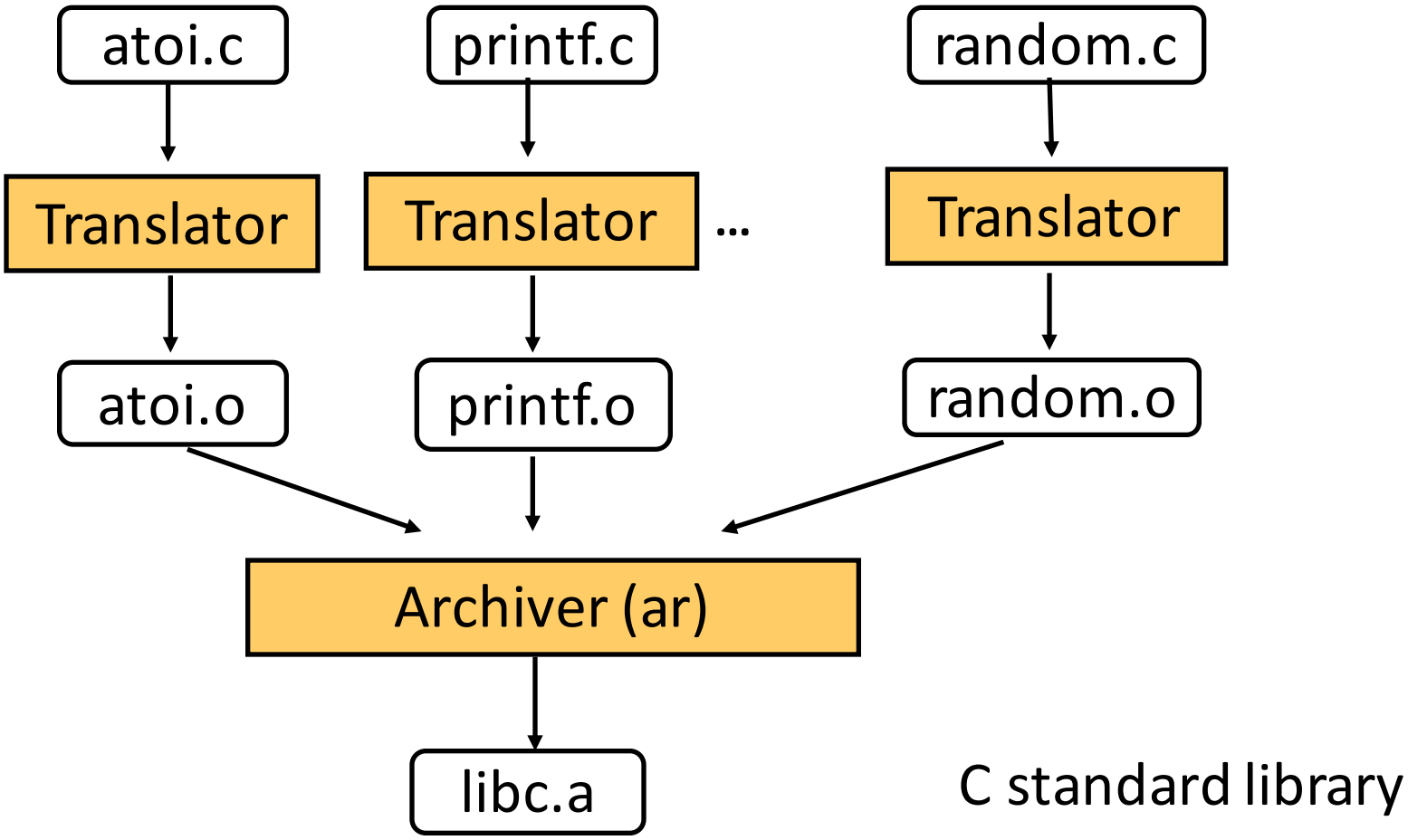
```
% ar -t /usr/lib/libm.a | sort
…
e_acos.o
e_acosf.o
e_acosh.o
e_acoshf.o
e_acoshl.o
e_acosl.o
e_asin.o
e_asinf.o
e_asinl.o
…
```

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.

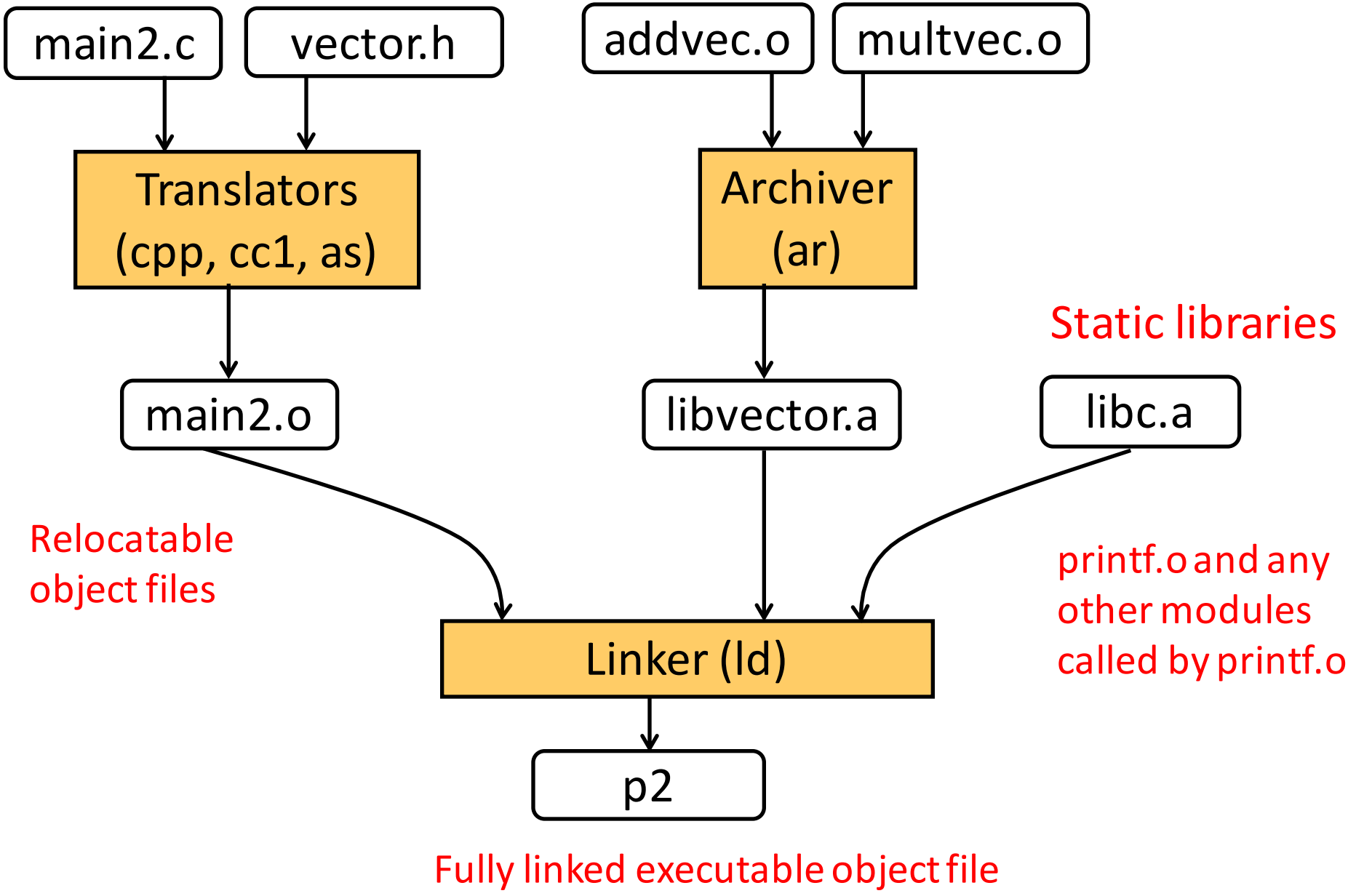
Creating Static Libraries



> ar rcs 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



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.

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

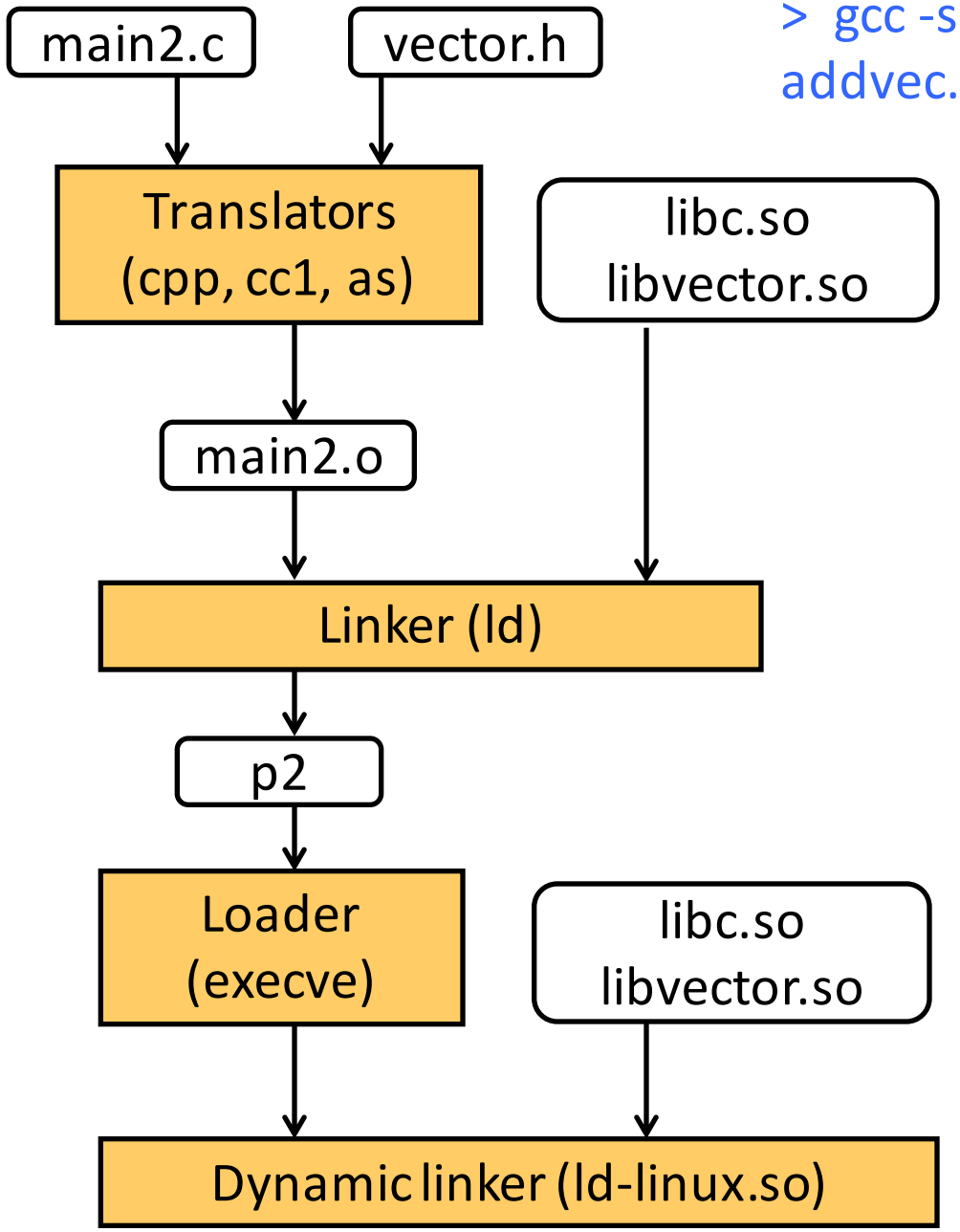
Shared Libraries

- 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

- 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 inter-positioning.
- Shared library routines can be shared by multiple processes.
  - More on this when we learn about virtual memory

Dynamic Linking at Load-time



> gcc -shared -o libvector.so \
addvec.c multvec.c

Relocatable object file

Relocation and symbol table info

Partially linked executable object file

Code and data

Fully linked executable in memory

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() 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;
}
```

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.

Adapted from slides by Bryant and O'Hallaron