CSCI-UA.0201

Computer Systems Organization

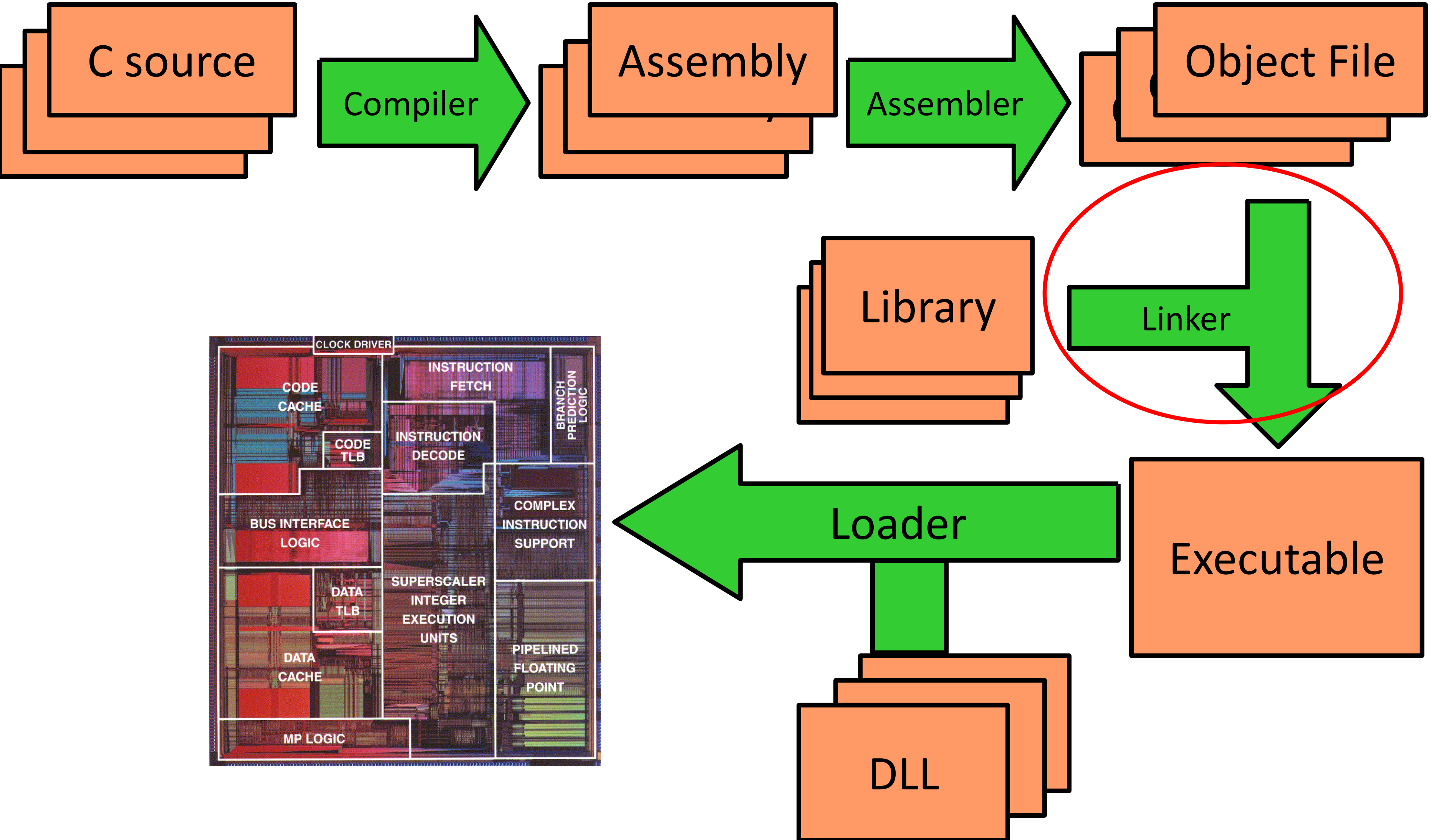
Machine Level – Linking and Loading

Thomas Wies

wies@cs.nyu.edu

https://cs.nyu.edu/wies

Source Code to Execution



C source

Compiler

Assembly

Assembler

Object File

Library

Linker

Executable

Loader

DLL

Linking Is ..

The process of collecting and combining various pieces of code and data into a single file that can be loaded into memory and executed.

Understanding Linkers Will Help You …

- build large programs
- avoid dangerous programming errors
- understand how language scoping rules are implemented
- understand other important systems concepts (virtual memory, paging, …)
- use shared libraries

Example C Program

main.c

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

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

swap.c

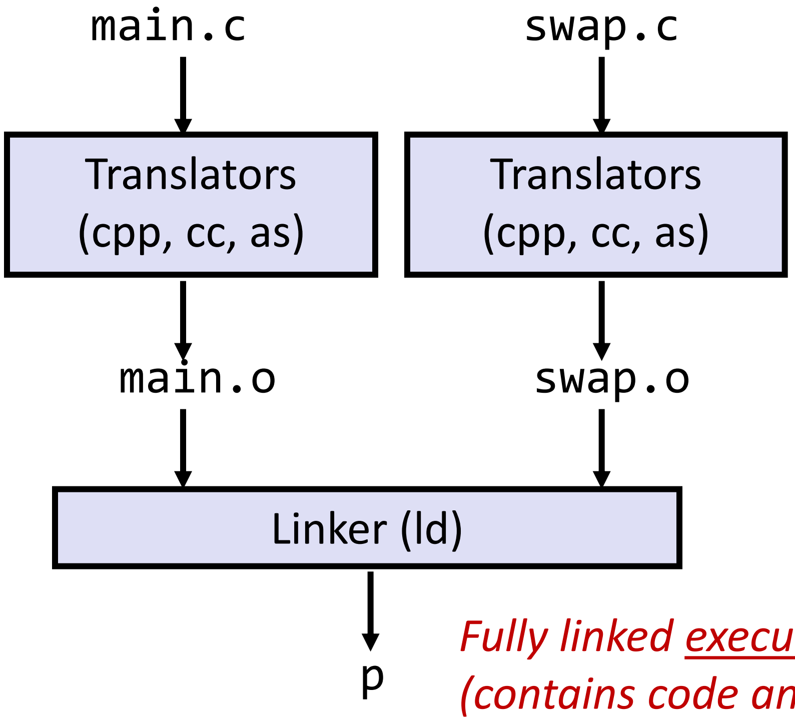
```
extern int buf[];

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

void swap()
{
   int temp;

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

The word *static* for global variable means it can only be accessed within its own module.

Module = a single file in the linker's lingo. So above, we have two modules.

Static Linking

- Programs are translated and linked using a *compiler driver*:
  - *$ gcc -O2 -g -o p main.c swap.c*
  - *$ ./p*



Source files

Separately compiled re-locatable object files

Fully linked <u>executable</u> object file (contains code and data for all functions defined in `main.c` and `swap.c`)

Why Linkers?

- Modularity
  - Write program as a set of smaller source files, rather than one giant file
  - Allow for libraries of common functions (more on this later)
    - e.g., math library, standard C library
- Efficiency
  - Separate compilation saves time
    - Change one source file, compile that file only, and then relink.
  - Libraries save memory space
    - Common functions can be aggregated into a single file...
    - Yet executable files 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, and reference x */
  ```

  – Symbol definitions are stored (by compiler) in *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 (one for code and one for data)
  - 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** (`a.out` file)
  - Contains code and data in a form that can be copied directly into memory and then executed.

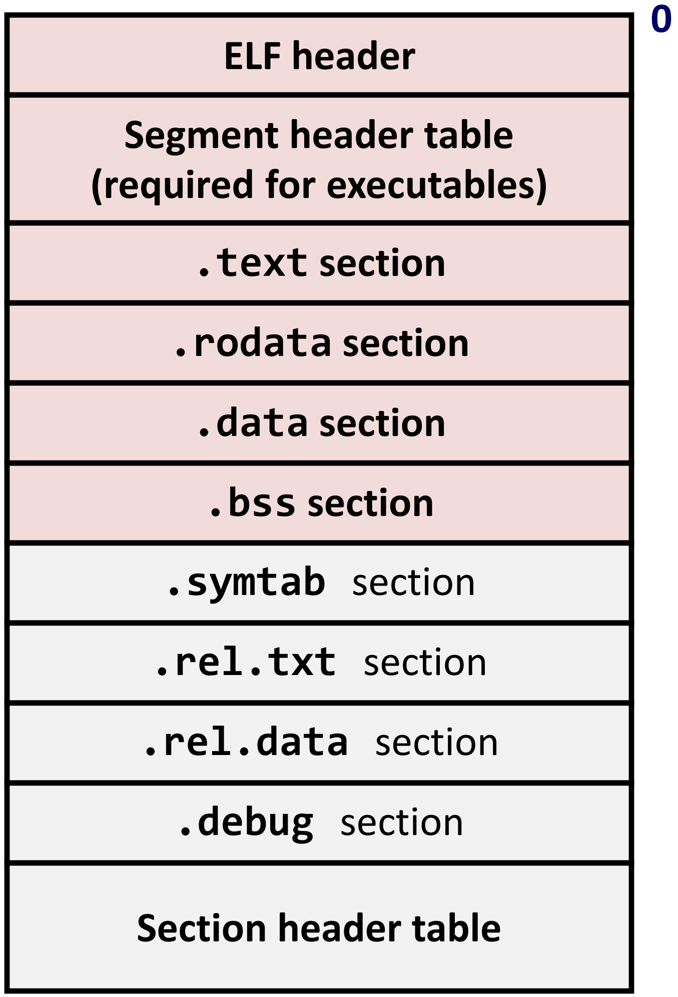
- **Shared object file** (`.so` file)
  - Special type of relocatable object file that can be loaded into memory and linked dynamically, at either load time or run-time.
  - Called *Dynamic Link Libraries* (DLLs) by Windows

Executable and Linkable Format (ELF)

- Standard binary format for object files
  - 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 `(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: jump tables, ...
- `.data` section
  - Initialized global variables
- `.bss` section (**B**lock **S**tarted by **S**ymbol)
  - Uninitialized global variables
  - Variables that are 0-initialized
  - Only the length but no data
  - Later, the program loader will allocate memory for it and 0-initialize all of it.



0

ELF header

Segment header table
(required for executables)

`.text` section

`.rodata` section

`.data` section

`.bss` section

`.symtab` section

`.rel.txt` section

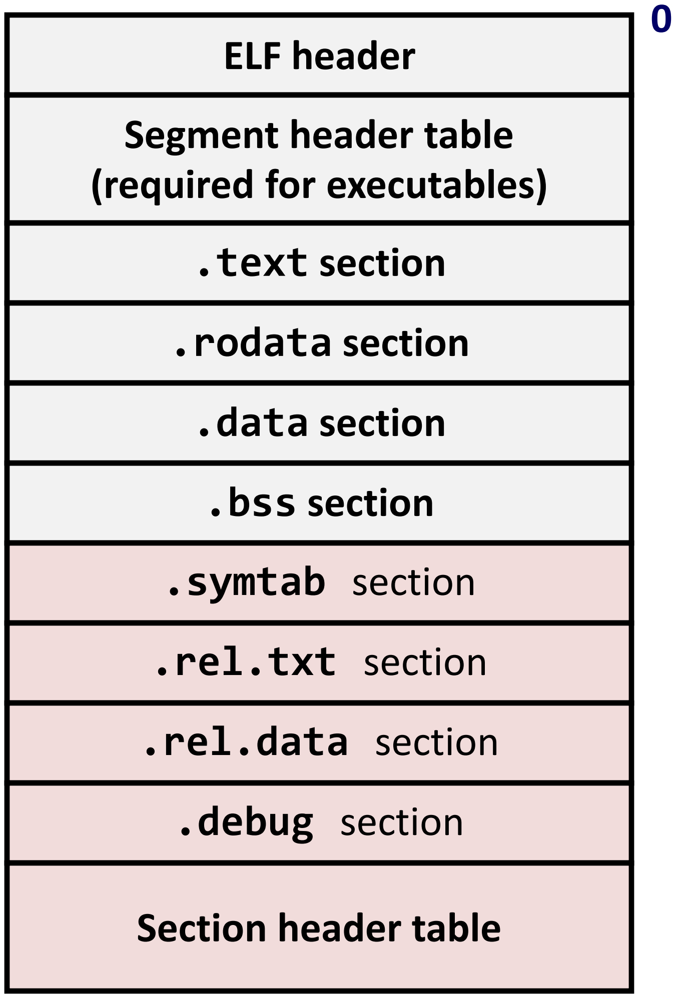
`.rel.data` section

`.debug` section

Section header table

ELF Object File Format (cont.)

- `.symtab` section
  - Symbol table
  - Procedure and global variable names
- `.rel.text` section
  - Relocation info for **`.text`** section
  - Addresses of instructions that will need to be modified in the executable
- `.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



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

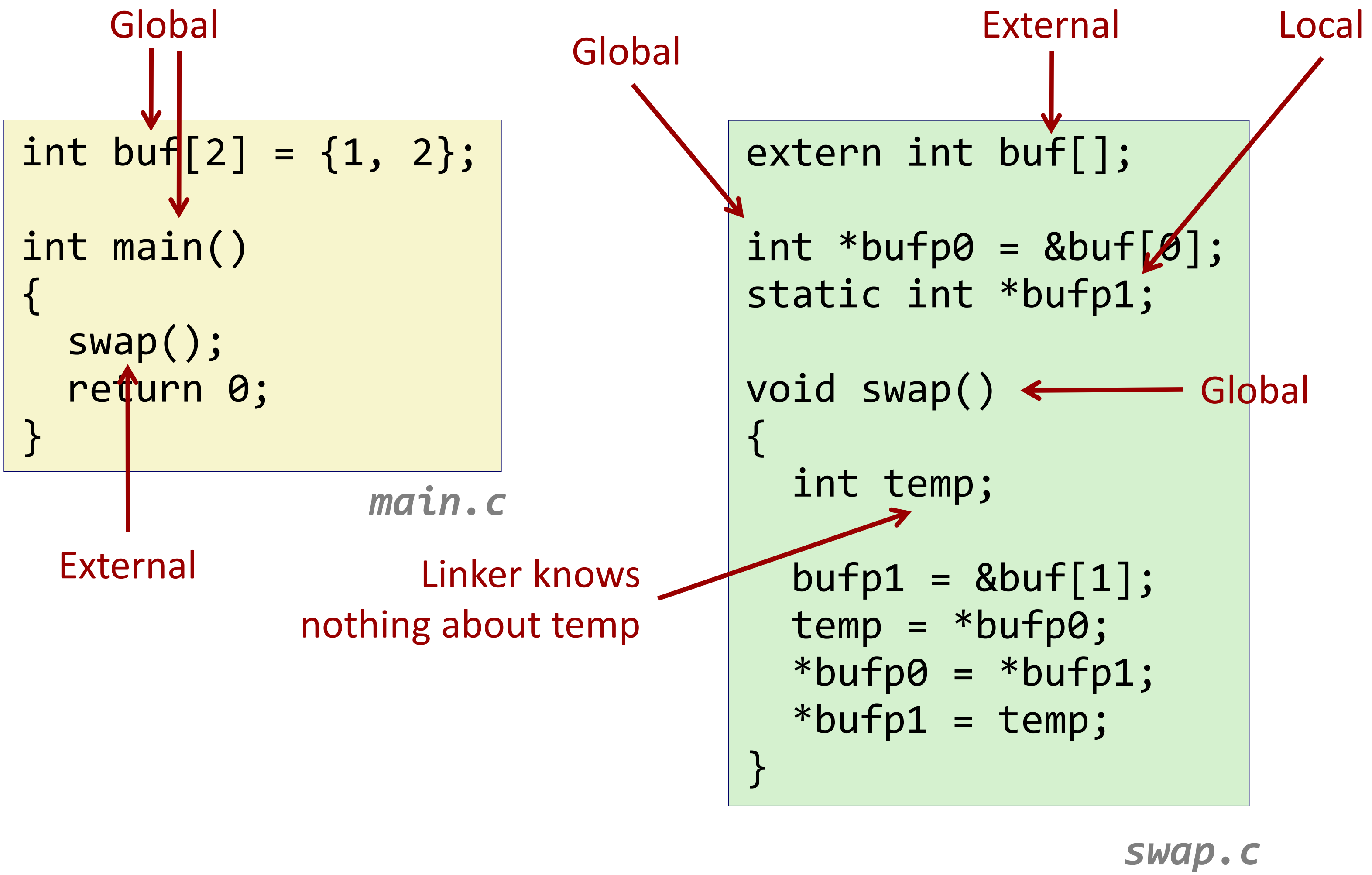
`.debug` section

Section header table

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.
  - Be careful: Local linker symbols are *not* local program variables (linker does not deal with the local variables of a function).

Resolving Symbols



Global

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

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

main.c

External

Global

External

Local

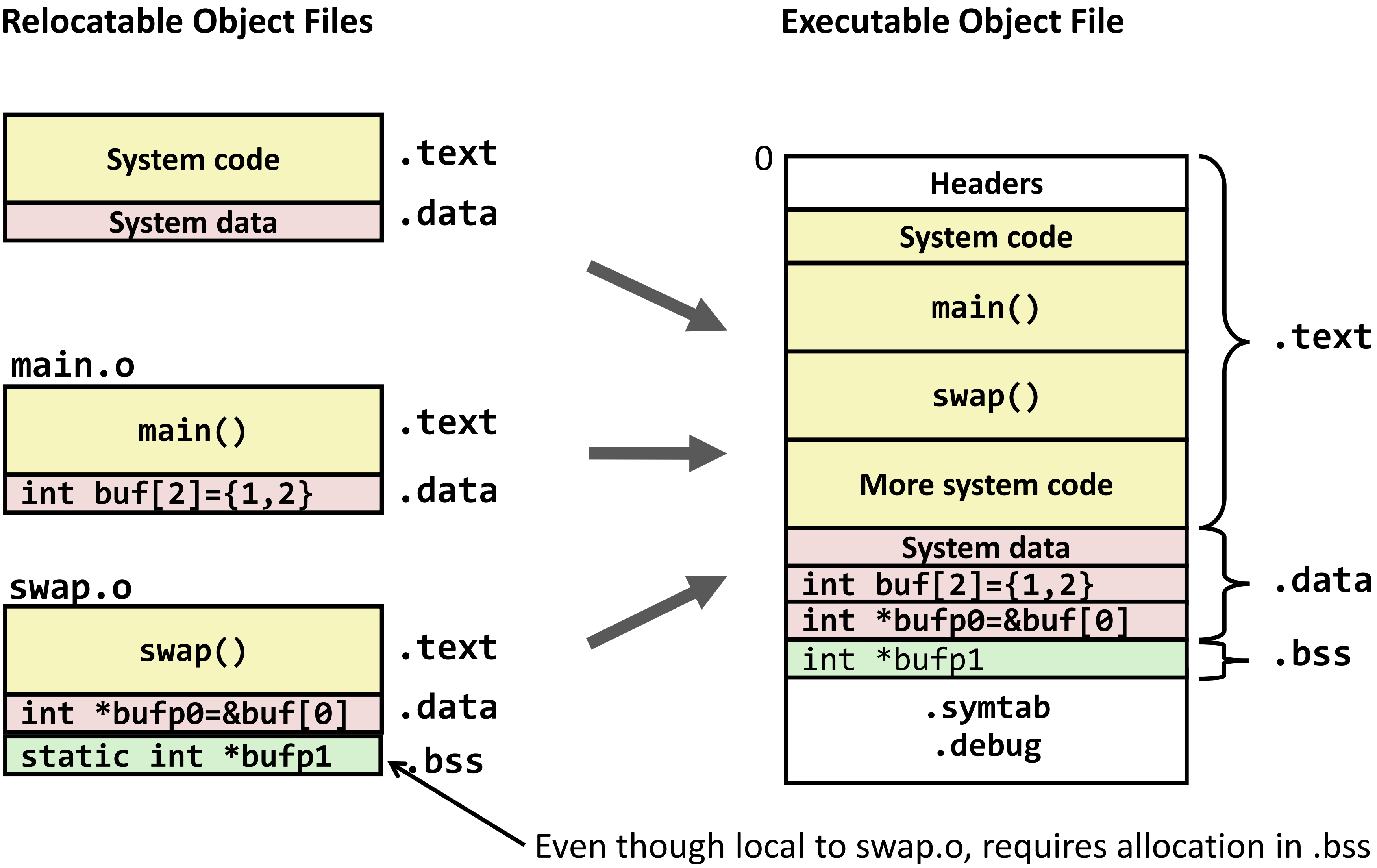
```
extern int buf[];

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

void swap()
{
  int temp;

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

Global

Linker knows nothing about temp

swap.c

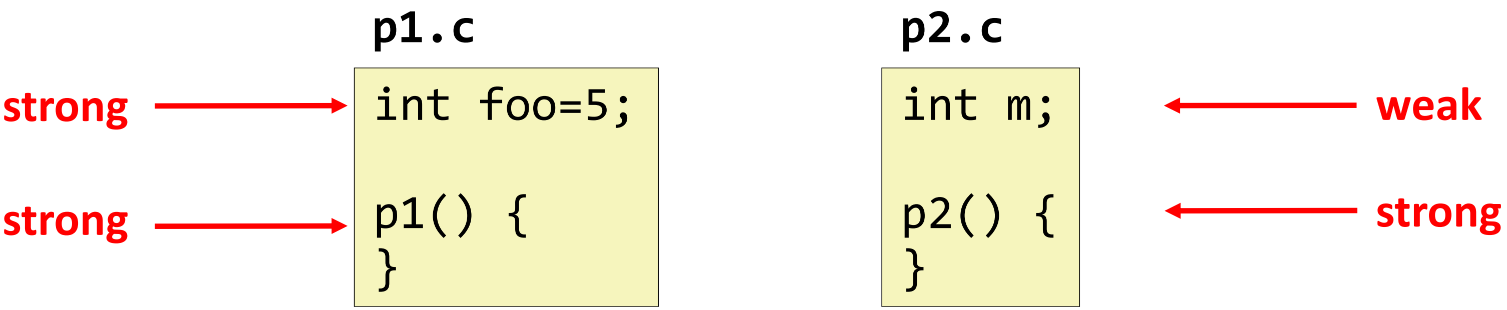
Relocating Code and Data



Even though local to swap.o, requires allocation in .bss

Strong and Weak Symbols

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



strong

strong

p1.c

```
int foo=5;

p1() {
}
```

p2.c

```
int m;

p2() {
}
```

weak

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

```
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.

Packaging Commonly Used Functions

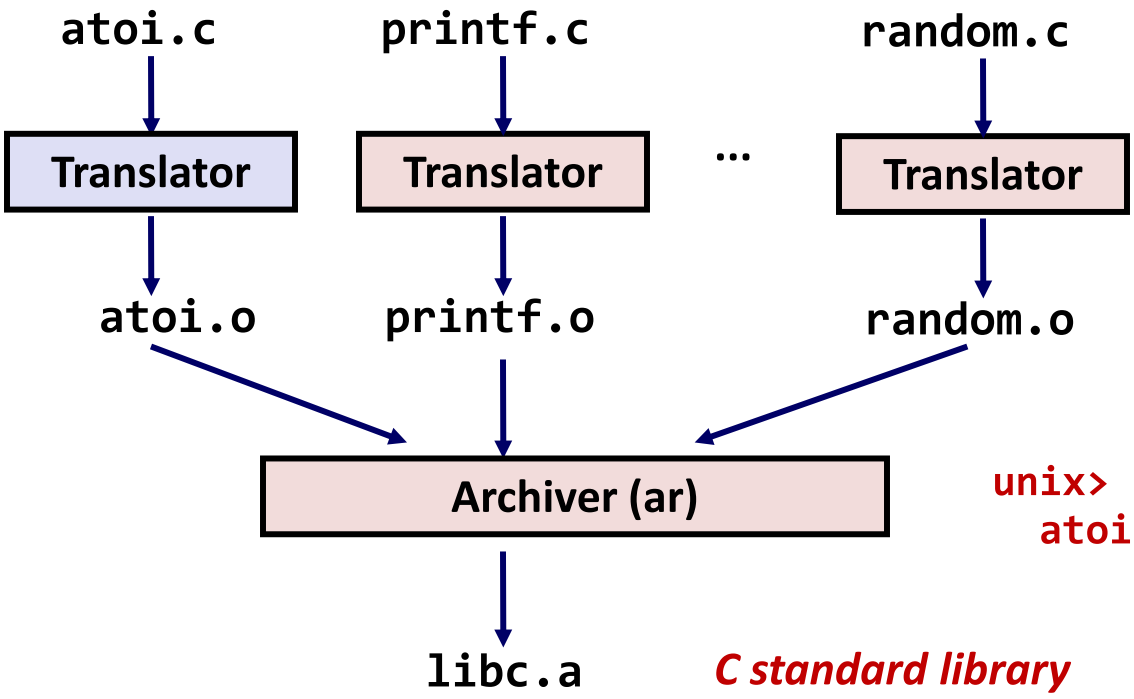
- How to package functions commonly used by programmers?
  - Math, I/O, memory management, string manipulation, etc.

- Awkward, given the linker framework so far:
  - **Option 1:** Put all functions into a single source file
    - Inefficient: programmers link big object file into their programs
  - **Option 2:** Put each function in a separate source file
    - Burdensome: programmers explicitly link appropriate binaries into their programs

Solution: Static Libraries

- Static libraries (`.a` archive files)
  - Concatenate related relocatable object files into a single file with an index (called an *archive*).
  - Linker 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
```

C standard library

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

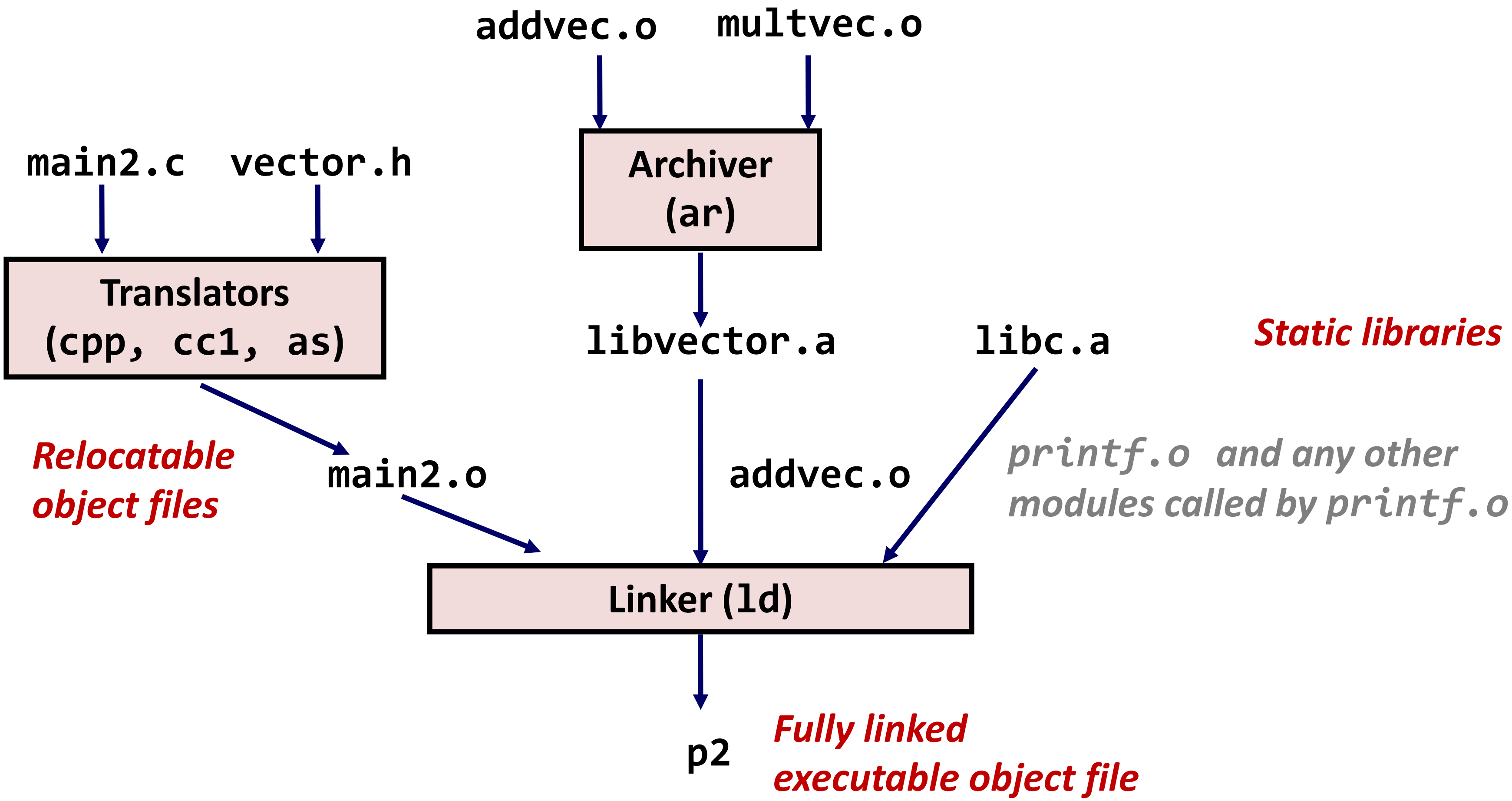
`libm.a` (the C math library)

- 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
…
```

Linking with Static Libraries



Using Static Libraries

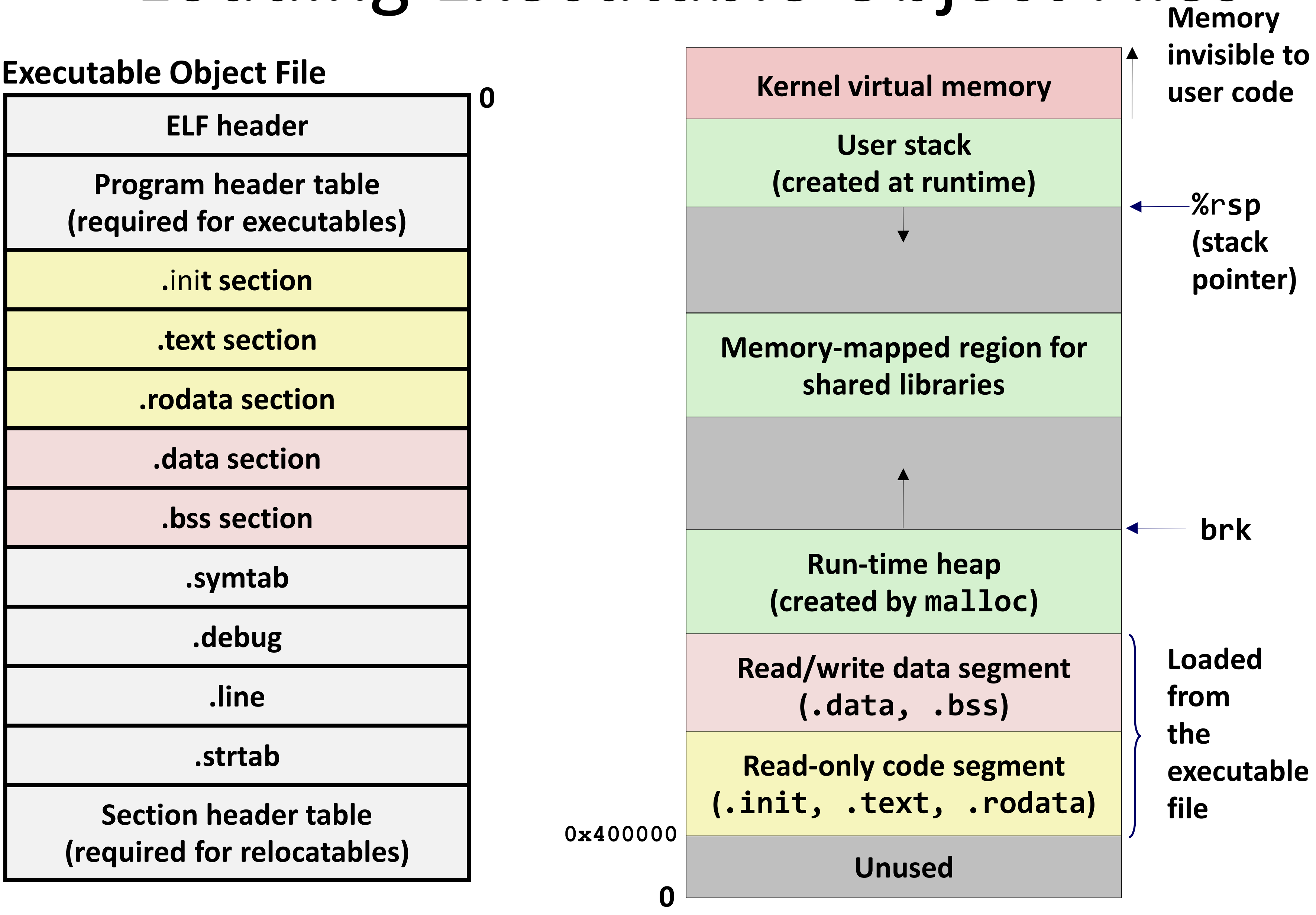
- Linker's algorithm for resolving external references:
  - Scan **`.o`** files and **`.a`** files in the command line order.
  - During the scan, keep a list of the current unresolved references.
  - As each new **`.o`** or **`.a`** file is encountered, try to resolve each unresolved reference in the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

fun is defined in `mine` and called by `libtest`

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)

Kernel virtual memory

Memory invisible to user code

User stack
(created at runtime)

`%rsp` (stack pointer)

Memory-mapped region for shared libraries

`brk`

Run-time heap
(created by `malloc`)

Read/write data segment
(`.data`, `.bss`)

Read-only code segment
(`.init`, `.text`, `.rodata`)

Loaded from the executable file

`0x400000`

Unused

0

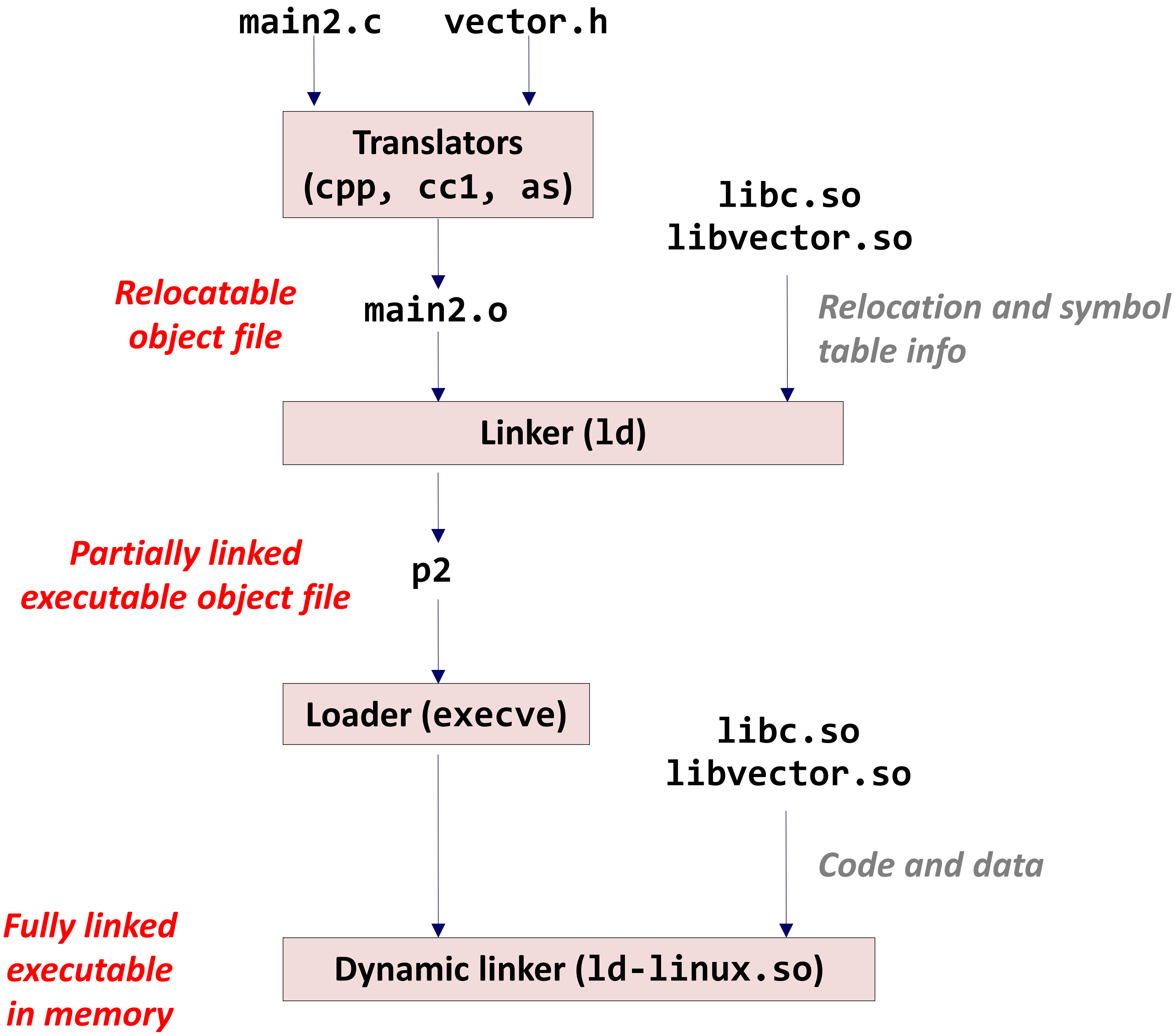
Shared Libraries

- Static libraries have the following disadvantages:
  - Duplication in the stored executables (e.g. every program needs `libc`)
  - Duplication in the running executables
  - Minor bug fixes of system libraries require each application to relink
- Modern solution: Shared Libraries
  - Object files 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.
  - 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 .
- Shared library routines can be shared by multiple processes.
  - More on this when we learn about virtual memory

Dynamic Linking at Load-time



main2.c

vector.h

Translators
(cpp, cc1, as)

Relocatable object file

main2.o

libc.so
libvector.so

Relocation and symbol table info

Linker (ld)

Partially linked executable object file

p2

Loader (execve)

libc.so
libvector.so

Code and data

Fully linked executable in memory

Dynamic linker (ld-linux.so)

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;
}
```

Conclusions

- source code (one or more modules) → preprocesser → compiler → assembler → linker → loader
- Now you can see the relationship among C code, assembly code, object code, and final executable