HARVARD
School of Engineering
and Applied Sciences

Linking and Loading

CS61, Lecture 16

Prof. Stephen Chong

October 25, 2011

Announcements

- Midterm exam in class on Thursday
  - 80 minute exam
  - Open book, closed note. No electronic devices allowed
  - Please be punctual, so we can start on time
- CS 61 Infrastructure maintenance
  - SEAS Academic Computing need to do some maintainance
  - All CS 61 VMs will be shut down on Thursday
  - You will need to re-register (i.e., run "seas-cloud register") next time you use the infrastructure
  - If this will cause problems for you, please contact course staff

Today

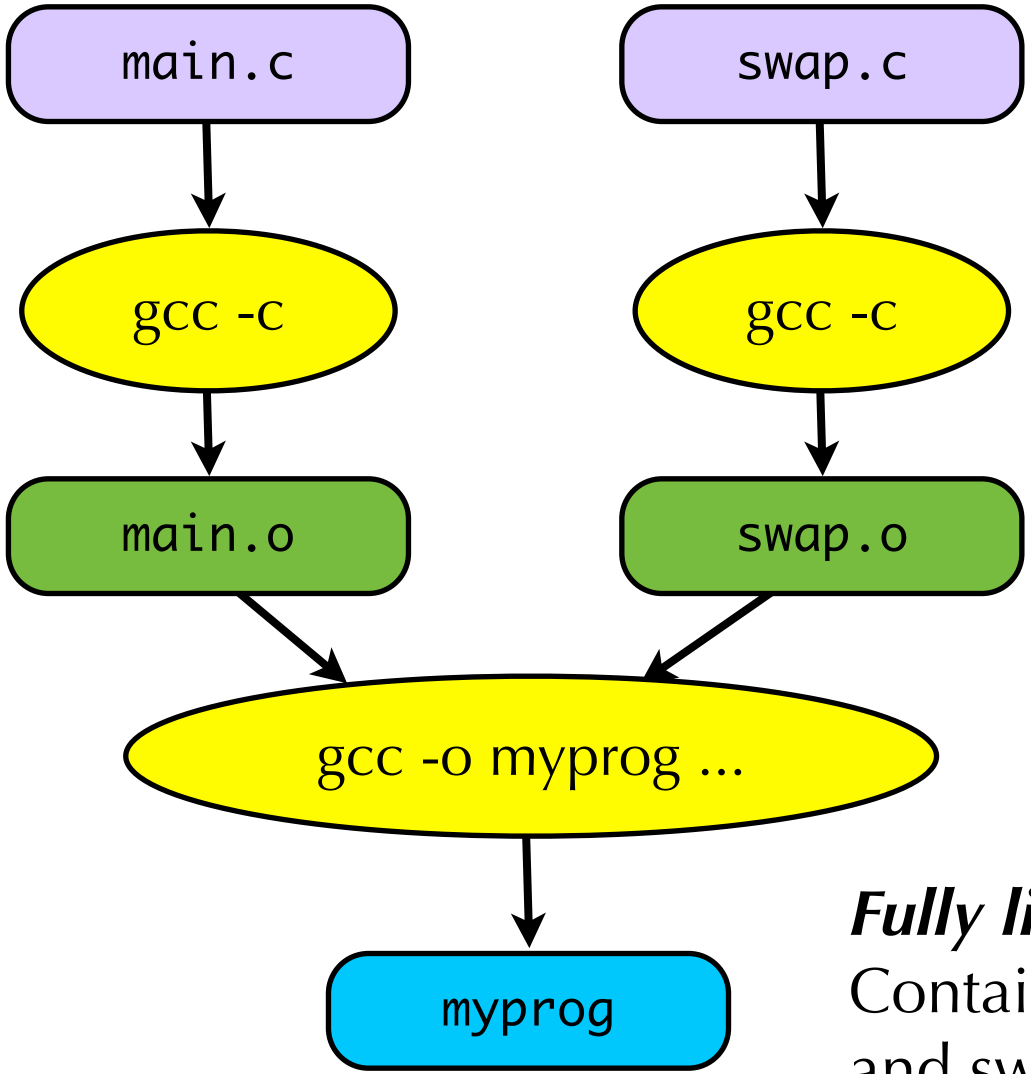
- Getting from C programs to executables
  - The what and why of linking
  - Symbol relocation
  - Symbol resolution
  - ELF format
  - Loading
  - Static libraries
  - Shared libraries

Linking

- **Linking** is the process of combining pieces of data and code into a single file than can be **loaded** into memory and executed.
- Why learn about linking?
    - Help you build large programs
    - Avoid dangerous programming errors
    - Understand how to use shared libraries
- Linking can be done
    - at compile time (aka statically)
    - at load time
    - at run time

Static linking

- Compiler driver coordinates translation and linking
  - `gcc` is a compiler driver, invoking C preprocessor (`cpp`), C compiler (`cc1`), assembler (`as`), and linker (`ld`)



Source files

Separately compiled relocatable object files

Fully linked executable object file
Contains code and data from main.o and swap.o

Example program with two C files

Definition of buf[]

main.c

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

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

Reference to swap()

Definition of main()

Declaration of buf[]

swap.c

```
extern int buf[];

void swap() {
  int temp;

  temp = buf[1];
  buf[1] = buf[0];
  buf[0] = temp;
}
```

Definition of swap()

References to buf[]

The linker

- The linker takes multiple object files and combines them into a single executable file.
- Three basic jobs:
    - 1) **Copy** code and data from each object file to the executable
    - 2) **Resolve** references between object files
    - 3) **Relocate** symbols to use absolute memory addresses, rather than relative addresses.

Resolving and relocating

main.c

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

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

swap.c

```
extern int buf[];

void swap() {
  int temp;

  temp = buf[1];
  buf[1] = buf[0];
  buf[0] = temp;
}
```

- Resolve: Find the definition of each **undefined reference** in an object file
  - E.g., The use of `swap()` in main.o needs to be resolved to definition found in swap.o
- Relocate: Assign code and data into absolute locations, and update references
  - swap.c compiled without knowing where buf will be in memory.
  - Linker must allocate `buf` (defined in main.o) into some memory location, and update uses of `buf` in swap.o to use this location.

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?

- Reason 2: **Efficiency**
  - Time efficiency: Separate Compilation
    - Change one source file, recompile just that file, and then relink the executable.
    - No need to recompile other source files.
  - Space efficiency: Libraries
    - Common functions can be combined into a single library file...
    - Yet executable files contain only code for the functions they actually use.

Today

- Getting from C programs to executables
  - The what and why of linking
  - Symbol relocation
  - Symbol resolution
  - ELF format
  - Loading
  - Static libraries
  - Shared libraries

Disassembly of main.o

main.c

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

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

```
$ objdump -d main.o
...
00000000 <main>:
  ...
   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>
  16:   b8 00 00 00 00         mov    $0x0,%eax
  ...
```

This is a bogus address!

Just a placeholder for `swap` (which we don't know the address of ... yet!)

main.o object file

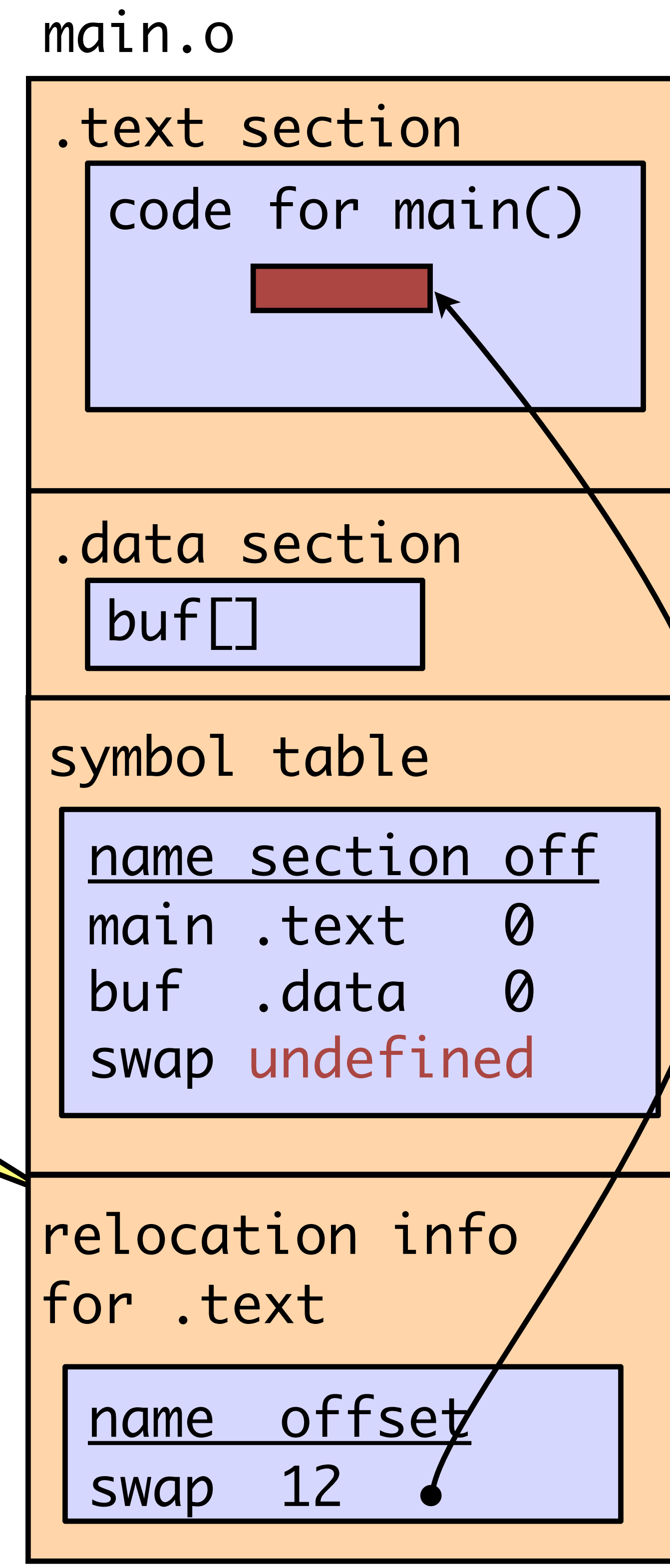
main.c

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

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

Relocation info tells the linker where references to external symbols are in the code

```
$ objdump -d main.o
...
00000000 <main>:
  ...
   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>
  16:   b8 00 00 00 00      mov    $0x0,%eax
  ...
```



Disassembly of swap.o

swap.c

```
extern int buf[];

void swap()
{
  int temp;

  temp = buf[1];
  buf[1] = buf[0];
  buf[0] = temp;
}
```

```
$ objdump -d swap.o
...
00000000 <swap>:
0:   55                        push   %ebp
   1:   89 e5                        mov     %esp,%ebp
   3:   8b 15 04 00 00 00            mov     0x4,%edx
   9:   a1 00 00 00 00               mov     0x0,%eax
   e:   a3 04 00 00 00               mov     %eax,0x4
  13:   89 15 00 00 00 00            mov     %edx,0x0
  19:   5d                           pop     %ebp
  1a:   c3                           ret
```

These are again just placeholders.

0x0 refers to buf[0]
0x4 refers to buf[1]

swap.o object file

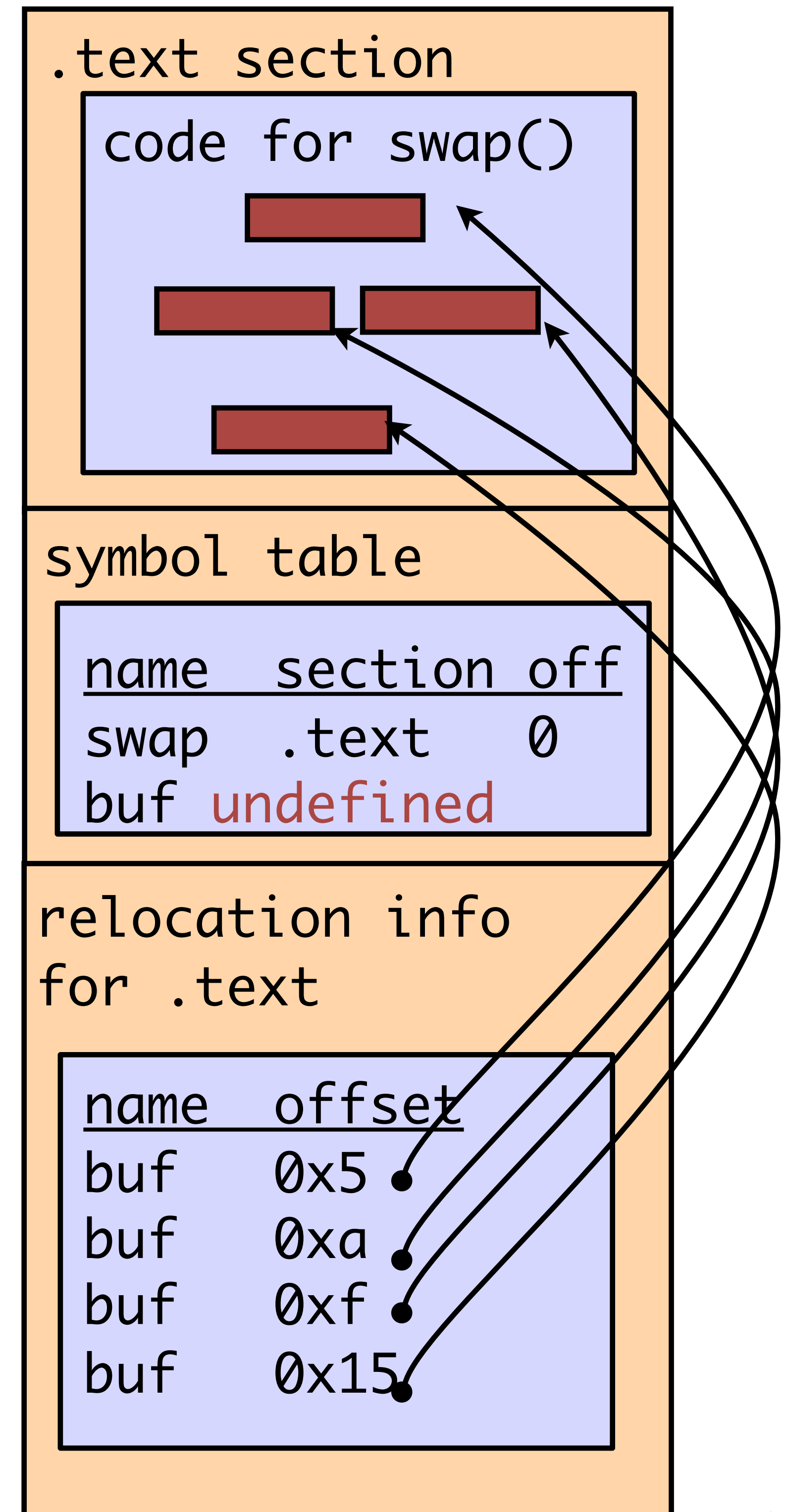
swap.c

```c
extern int buf[];

void swap()
{
  int temp;

  temp = buf[1];
  buf[1] = buf[0];
  buf[0] = temp;
}
```

swap.o

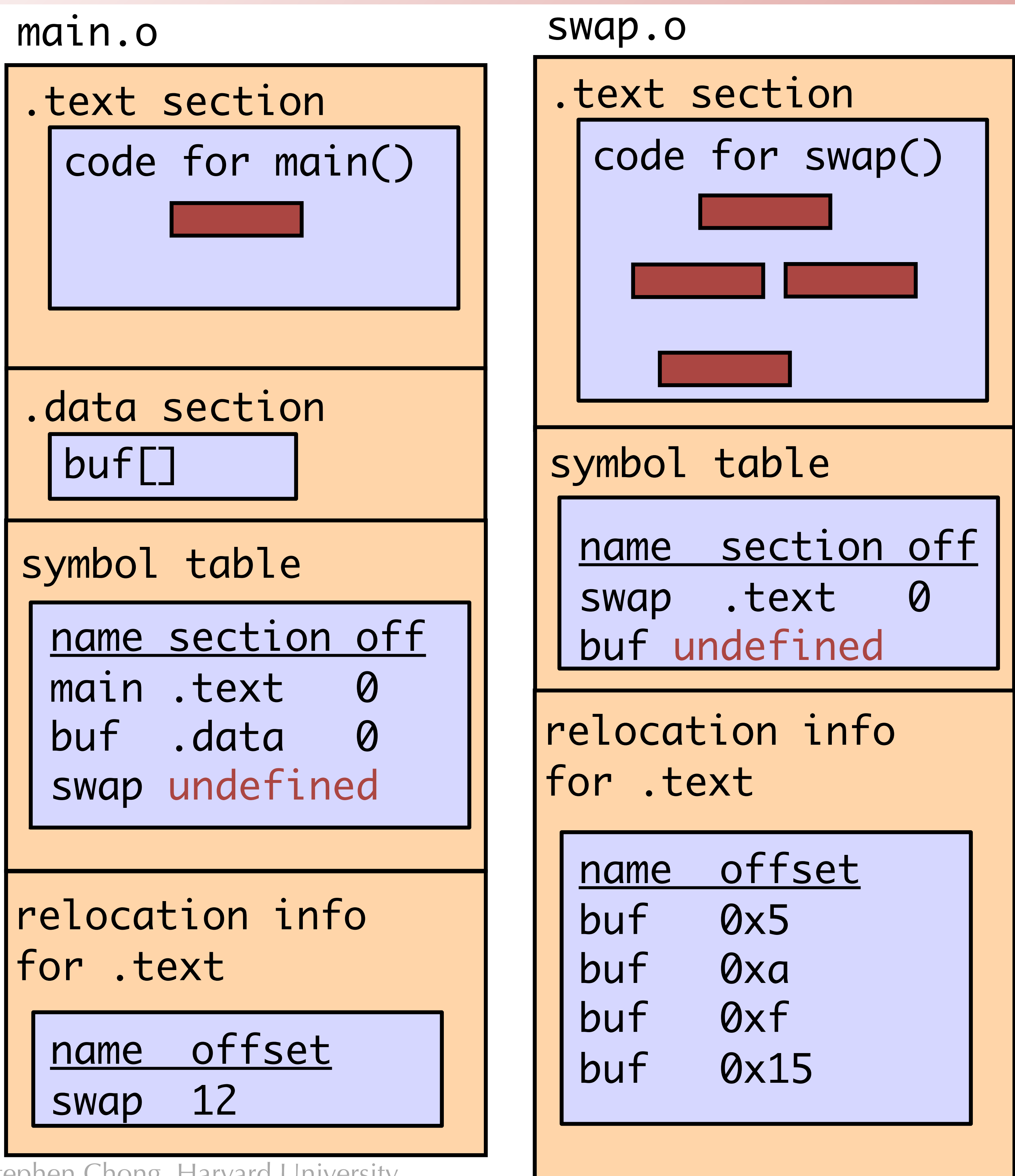


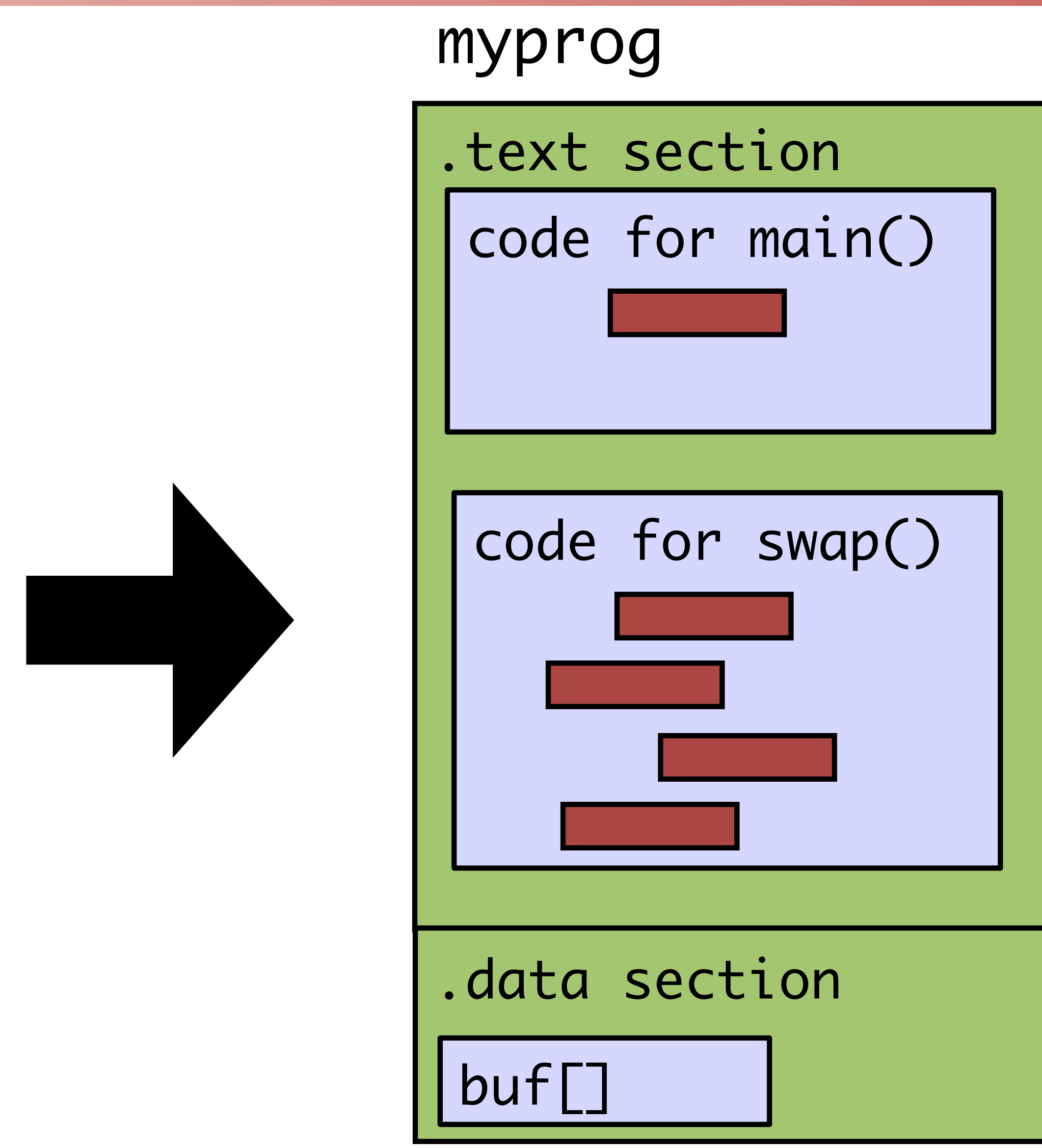

```
$ objdump -d swap.o
...
00000000 <swap>:
0:   55                        push   %ebp
   1:   89 e5                      mov    %esp,%ebp
   3:   8b 15 04 00 00 00          mov    0x4,%edx
   9:   a1 00 00 00 00             mov    0x0,%eax
   e:   a3 04 00 00 00             mov    %eax,0x4
  13:   89 15 00 00 00 00          mov    %edx,0x0
  19:   5d                         pop    %ebp
  1a:   c3                         ret
```

The linker

- The linker takes multiple object files and combines them into a single executable file.
- Three basic jobs:
  - 1) **Copy** code and data from each object file to the executable
  - 2) **Resolve** references between object files
  - 3) **Relocate** symbols to use absolute memory addresses, rather than relative addresses.

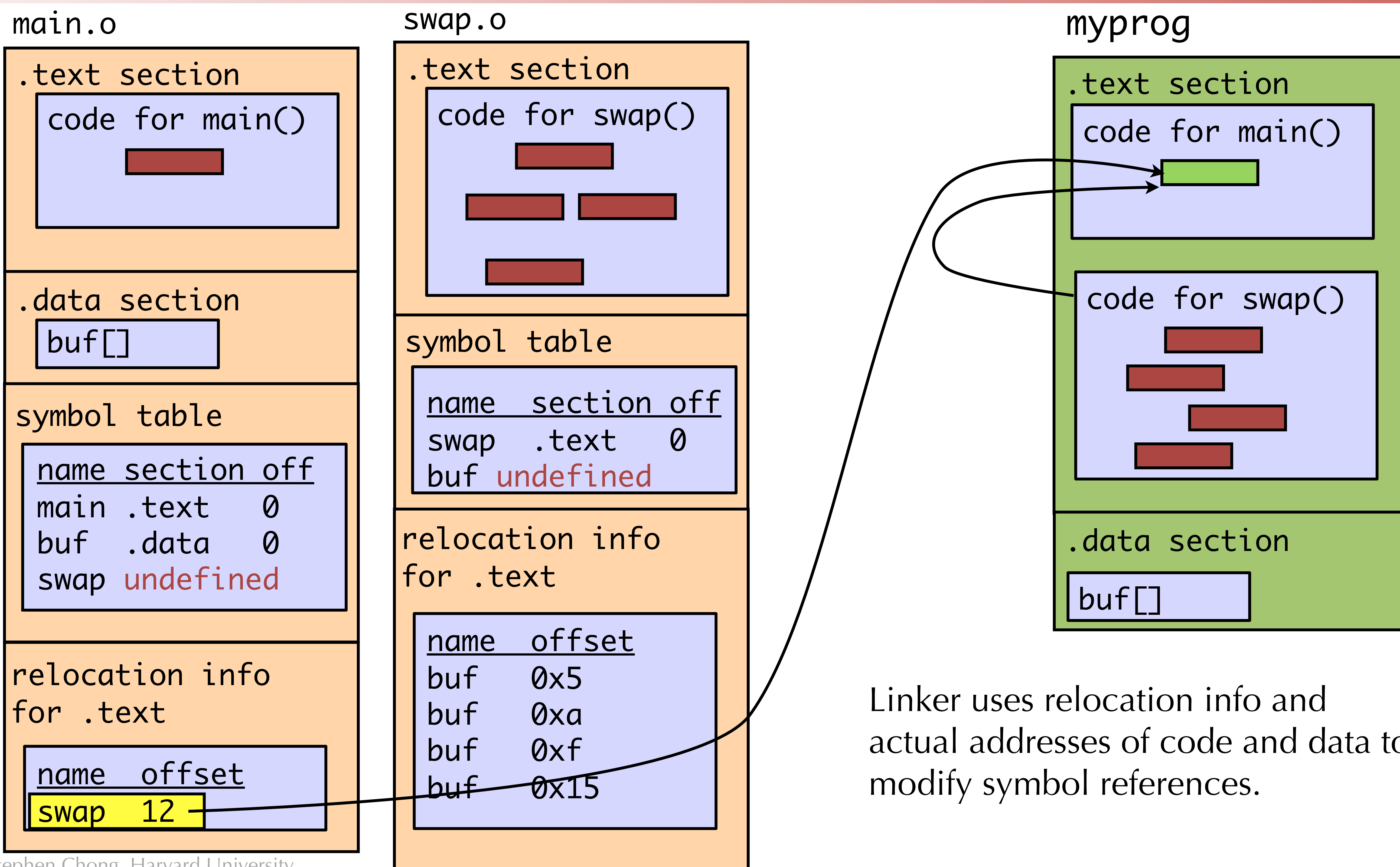
Linker operation





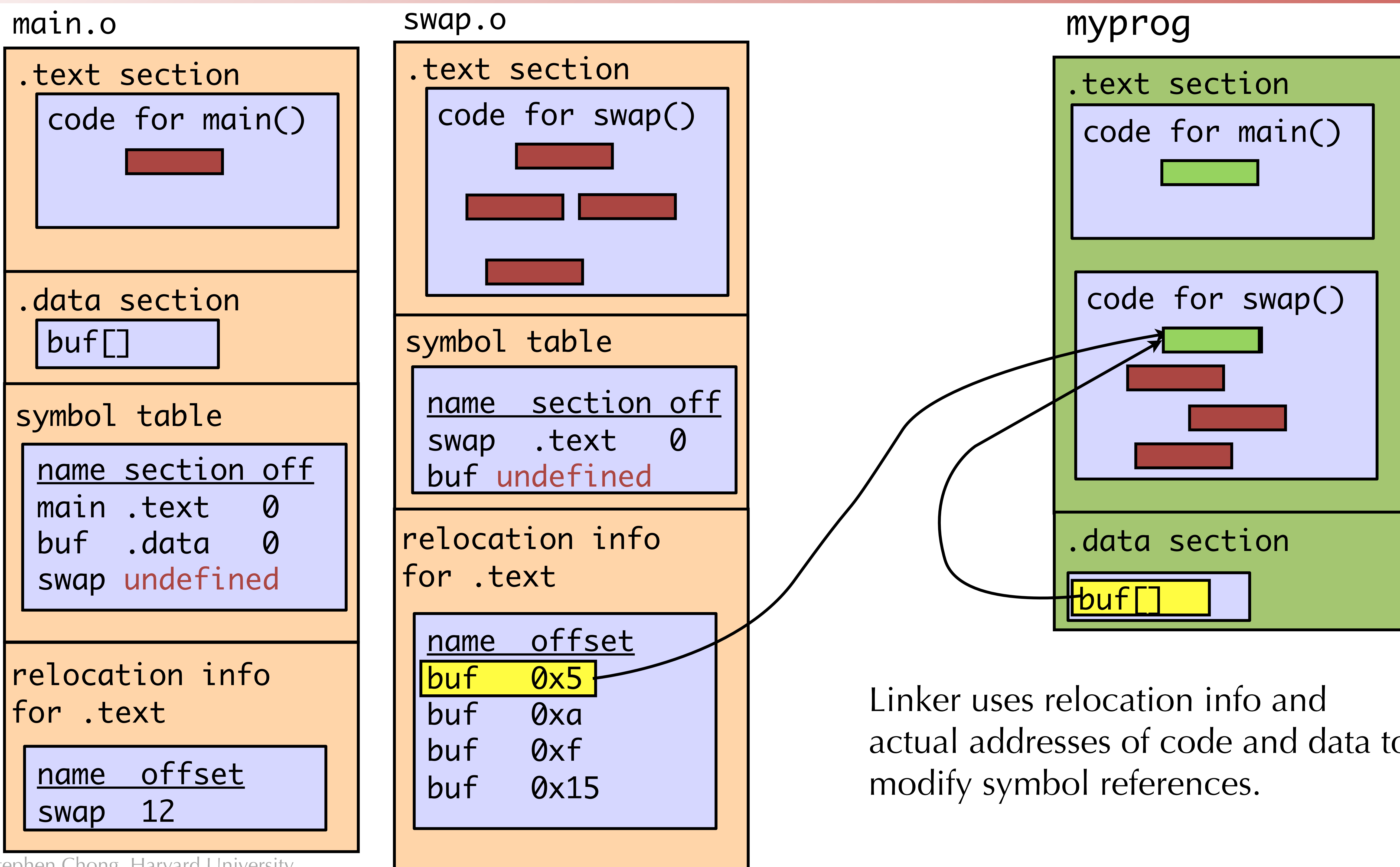
Executable file must contain the **absolute** memory addresses of the code and data (that is, where they will be located when the program actually runs!)

Linker operation



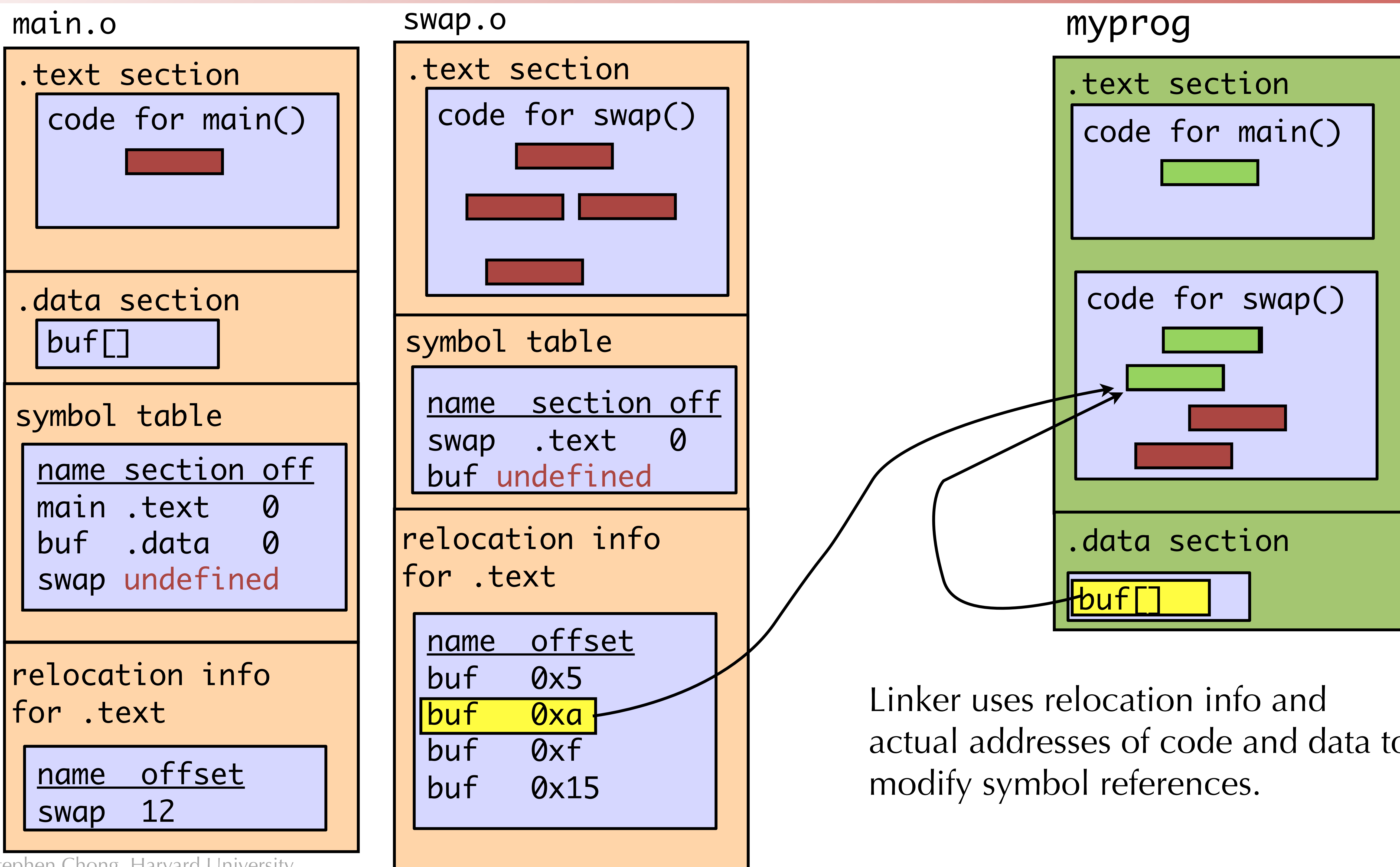
Linker uses relocation info and actual addresses of code and data to modify symbol references.

Linker operation



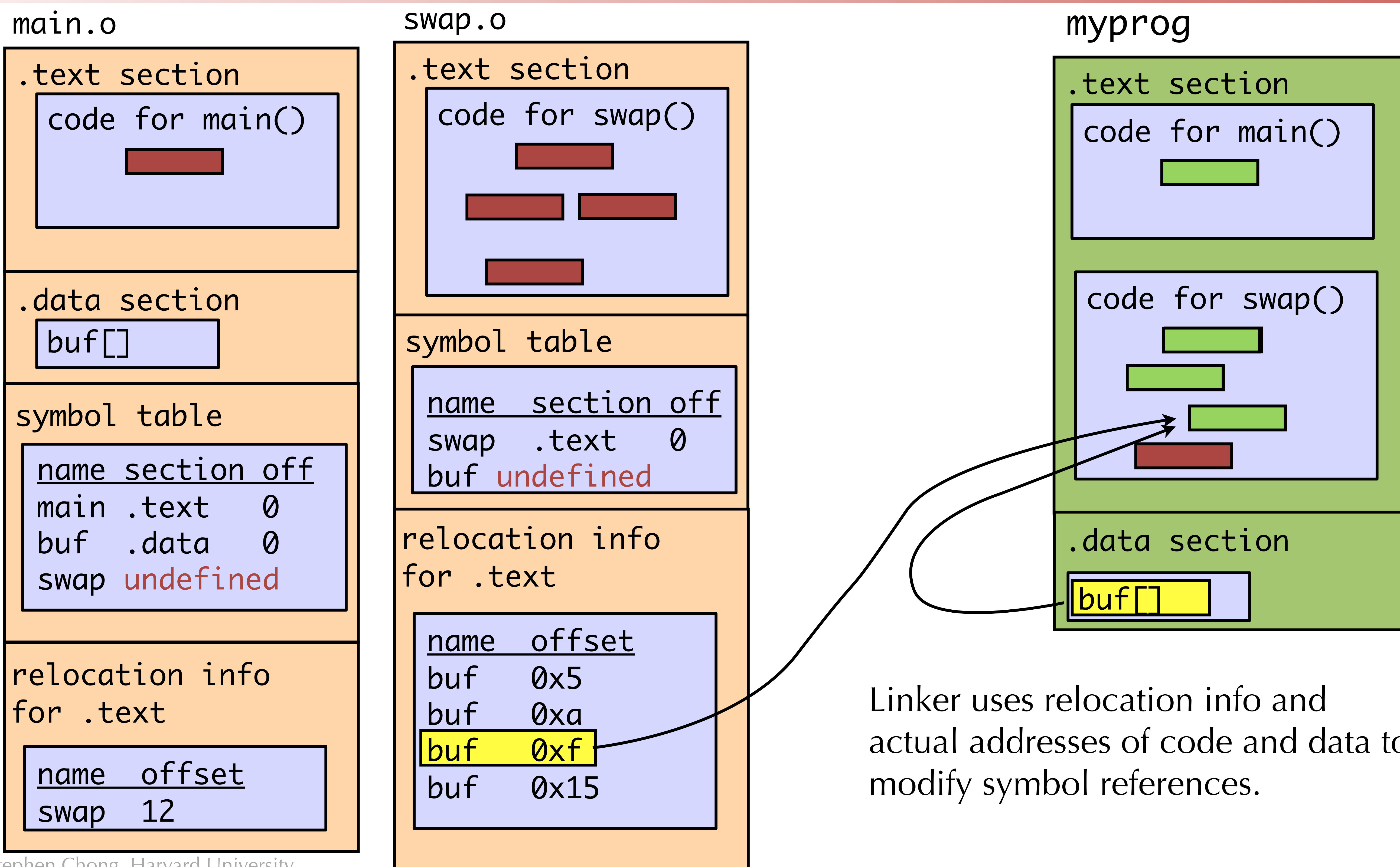
Linker uses relocation info and actual addresses of code and data to modify symbol references.

Linker operation



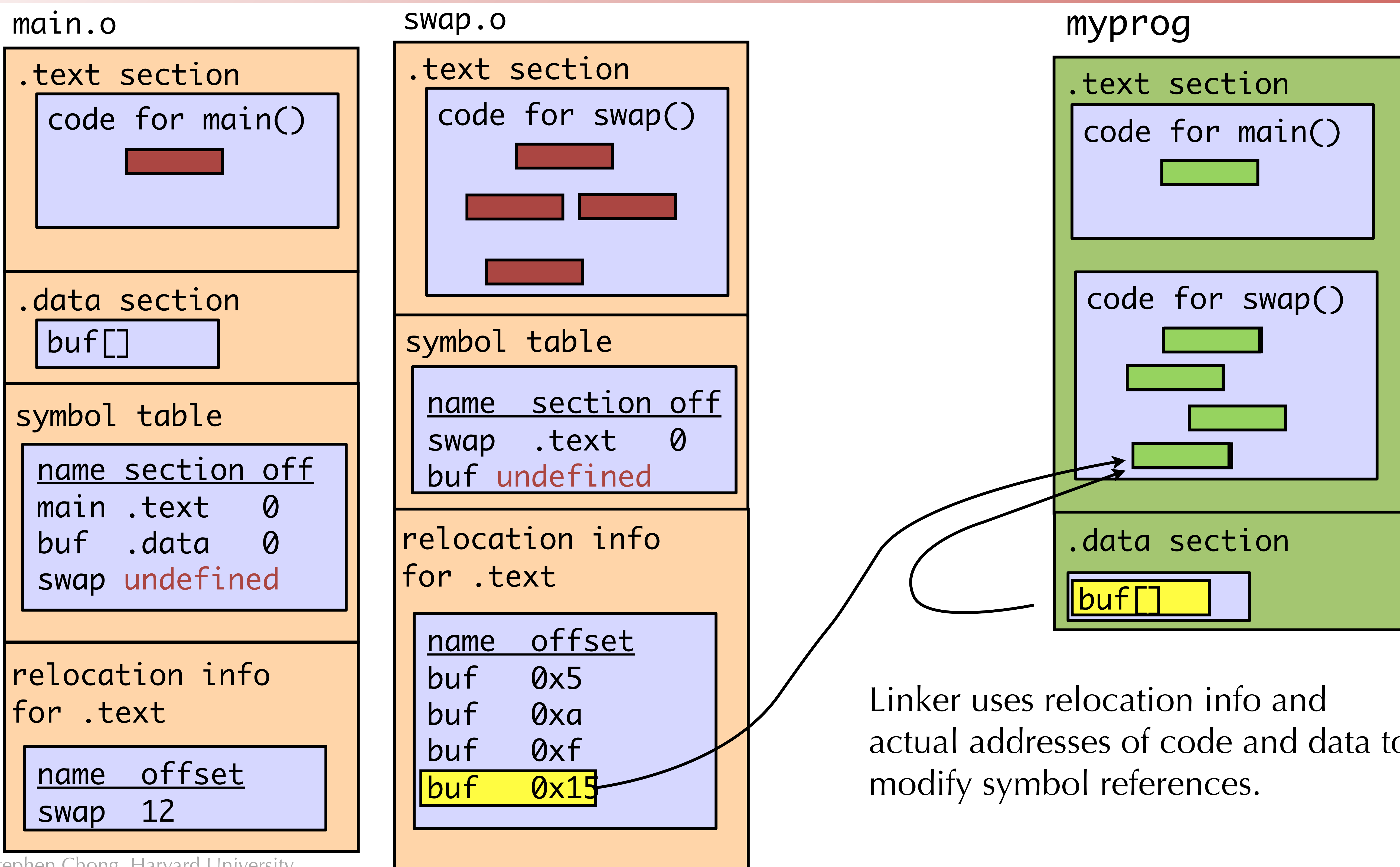
Linker uses relocation info and actual addresses of code and data to modify symbol references.

Linker operation



Linker uses relocation info and actual addresses of code and data to modify symbol references.

Linker operation



Linker uses relocation info and actual addresses of code and data to modify symbol references.

Disassembly after linking

```
08048344 <main>:
 ...
 804834e:       55                      push   %ebp
 804834f:       89 e5                   mov    %esp,%ebp
 8048351:       51                      push   %ecx
 8048352:       83 ec 04                sub    $0x4,%esp
 8048355:       e8 0e 00 00 00          call   8048368 <swap>
 804835a:       b8 00 00 00 00          mov    $0x0,%eax
 ...

08048368 <swap>:
 8048368:       55                      push   %ebp
 8048369:       89 e5                   mov    %esp,%ebp
 804836b:       8b 15 5c 95 04 08       mov    0x804955c,%edx
 8048371:       a1 58 95 04 08          mov    0x8049558,%eax
 8048376:       a3 5c 95 04 08          mov    %eax,0x804955c
 804837b:       89 15 58 95 04 08       mov    %edx,0x8049558
 8048381:       5d                      pop    %ebp
 8048382:       c3                      ret
```

Disassembly after linking

PC relative addressing:
%eip = 0x804835a
%eip + 0xe = 0x8048368

```
08048344 <main>:
 ...
 804834e:       55
 804834f:       89 e5                   mov    %esp,%ebp
 8048351:       51                      push   %ecx
 8048352:       83 ec 04                sub    $0x4,%esp
 8048355:       e8 0e 00 00 00          call   8048368 <swap>
 804835a:       b8 00 00 00 00          mov    $0x0,%eax
 ...

08048368 <swap>:
 8048368:       55                      push   %ebp
 8048369:       89 e5                   mov    %esp,%ebp
 804836b:       8b 15 5c 95 04 08       mov    0x804955c,%edx
 8048371:       a1 58 95 04 08          mov    0x8049558,%eax
 8048376:       a3 5c 95 04 08          mov    %eax,0x804955c
 804837b:       89 15 58 95 04 08       mov    %edx,0x8049558
 8048381:       5d                      pop    %ebp
 8048382:       c3                      ret
```

Disassembly after linking

```
$ objdump -t myprog
...
08049558 g     O .data  00000008                buf
...
```

Address Section Size Symbol name

```
08048368 <swap>:
 8048368:       55                      push   %ebp
 8048369:       89 e5                   mov    %esp,%ebp
 804836b:       8b 15 5c 95 04 08       mov    0x804955c,%edx
 8048371:       a1 58 95 04 08          mov    0x8049558,%eax
 8048376:       a3 5c 95 04 08          mov    %eax,0x804955c
 804837b:       89 15 58 95 04 08       mov    %edx,0x8049558
 8048381:       5d                      pop    %ebp
 8048382:       c3                      ret
```

Today

- Getting from C programs to executables
  - The what and why of linking
  - Symbol relocation
  - Symbol resolution
  - ELF format
  - Loading
  - Static libraries
  - Shared libraries

The linker

- Recall:
  - The linker takes multiple object files and combines them into a single executable file.
  - Three basic jobs:
    - 1) **Copy** code and data from each object file to the executable
    - 2) **Resolve** references between object files
    - 3) **Relocate** symbols to use absolute memory addresses, rather than relative addresses.
- We have looked at copying and relocation.
- How does linker resolve references?

Strong vs. Weak Symbols

- The compiler exports each global symbol as either strong or weak
  - **Strong symbols**:
    - Functions
    - Initialized global variables

  - **Weak symbols**:
    - Uninitialized global variables

Strong vs. Weak Symbols

Strong

main.c

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

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

Strong

Weak

swap.c

Not a global symbol (just a reference)

```
int myvar;
extern int buf[];

void swap() {
  int temp;

  temp = buf[1];
  buf[1] = buf[0];
  buf[0] = temp;
}
```

Strong

Not a global symbol

Linker rules

- Rule 1: Multiple strong symbols with the same name are not allowed.

foo1.c

```
int somefunc(){
  return 0;
}
```

foo2.c

```
int somefunc() {
  return 1;
}
```

```
$ gcc foo1.c foo2.c
/tmp/ccACj1wn.o: In function `somefunc':
foo2.c:(.text+0x0): multiple definition of `somefunc'
/tmp/ccaef3uE.o:foo1.c:(.text+0x1e): first defined here
collect2: ld returned 1 exit status
```

Linker rules

- Rule 2: Given a strong symbol and multiple weak symbols, choose the strong symbol.

foo1.c

```
void f(void);
int x = 38;

int main() {
  f();
  printf("x = %d\n", x);
  return 0;
}
```

Strong

foo2.c

```
int x;

void f() {
  x = 42;
}
```

Weak

```
$ gcc -o myprog foo1.c foo2.c
$ ./myprog
x = 42
```

Potentially unexpected behavior!

Linker rules

- This can lead to some pretty weird bugs!!!

foo1.c

```
void f(void);
int x = 38;
int y = 39;

int main() {
  f();
  printf("x = %d\n", x);
  printf("y = %d\n", y);
  return 0;
}
```

Strong

foo2.c

```
double x;

void f() {
  x = 42.0;
}
```

Weak

```
$ gcc -o myprog foo1.c foo2.c
$ ./myprog
x = 0
y = 1078263808
```

`double x` is 8 bytes in size!
But resolves to address
of `int x` in `foo1.c`.

Linker rules

- Rule 3: Given multiple weak symbols, pick any one of them

foo1.c

```
void f(void);
int x;

int main() {
  x = 38;
  f();
  printf("x = %d\n", x);
  return 0;
}
```

Weak

foo2.c

```
int x;

void f() {
  x = 42;
}
```

Weak

```
$ gcc -o myprog foo1.c foo2.c
$ ./myprog
x = 42
```

Static keyword

- In C, the keyword `static` affects the lifetime and linkage (visibility) of a variable
- A `static` global variable, declared at the top of a source file, is visible only within the source file
  - Linker will not resolve any reference from another object file to it

foo1.c

```
static int x = 38;

int main() {
  g();
  printf("x = %d\n", x);
  return 0;
}
```

foo2.c

```
int x;

void g() {
  x = 42;
}
```

```
$ gcc -o myprog foo1.c foo2.c
$ ./myprog
x = 38
```

Today

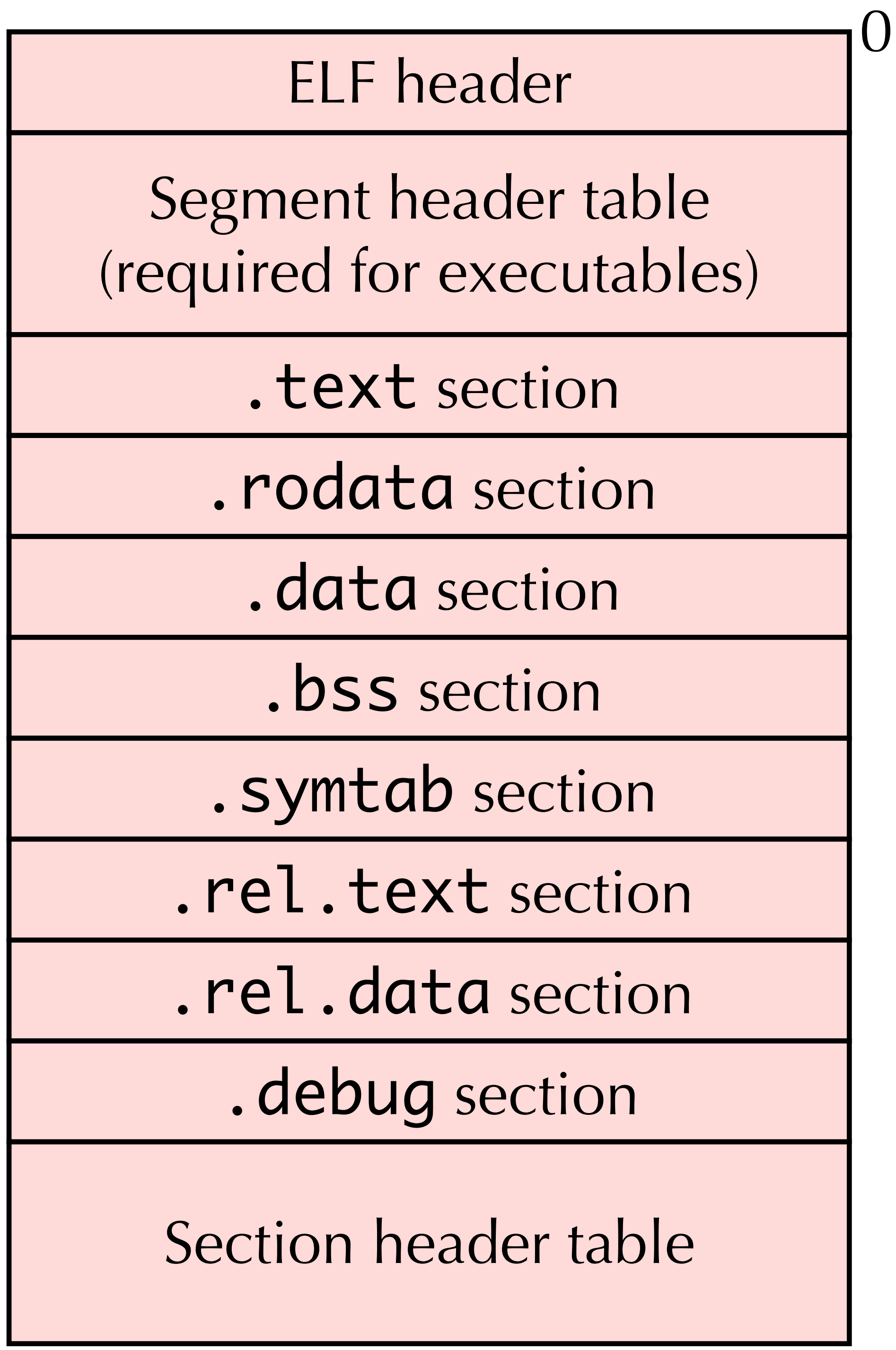
- Getting from C programs to executables
  - The what and why of linking
  - Symbol relocation
  - Symbol resolution
  - ELF format
  - Loading
  - Static libraries
  - Shared libraries

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)
  - Shared object files (.so)
  - Executable files

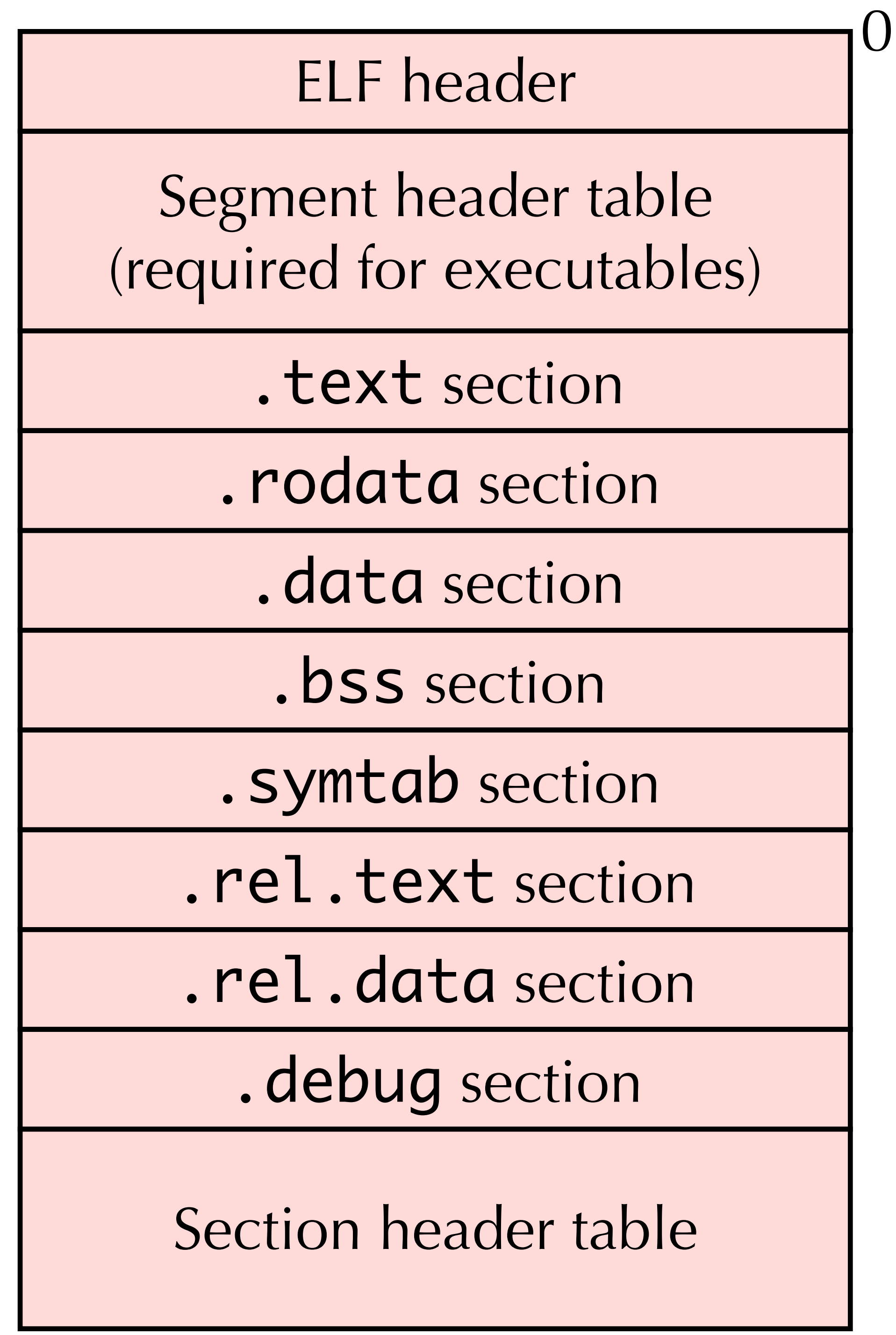
ELF Object File Format

- Elf header
  - Magic number, type (.o, exec, .so), machine, byte ordering, etc.
- Segment header table
  - Maps contiguous file sections to runtime memory segments
- `.text` section
  - Code
- `.rodata` section
  - Read-only data, e.g. jump tables, constant strings
- `.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

- `.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 modification in merged executable
- `.debug` section
  - Info for debugging (`gcc -g`)
- Section header table
  - Offsets and sizes of each section



objdump: Looking at ELF files

- Use the `objdump` tool to peek inside of ELF files (.o, .so, and executable files)
- `objdump -h`: print out list of sections in the file.

```
$ objdump -h myprog
myprog:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
...
 11 .text         000001c8  080482a0  080482a0  000002a0  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
...
 22 .data         0000000c  080495f8  080495f8  000005f8  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 23 .bss          0000000c  08049604  08049604  00000604  2**2
                  ALLOC
```

objdump: Looking at ELF files

- Use the `objdump` tool to peek inside of ELF files (.o, .so, and executable files)
- `objdump -s`: print out full contents of each section.

```
$ objdump -s myprog
myprog:     file format elf32-i386


...
Contents of section .text:
 80482a0 31ed5e89 e183e4f0 50545268 c0830408  1.^.....PTRh....
 80482b0 68d08304 08515668 74830408 e8c7ffff  h....QVht.......
 80482c0 fff49090 5589e553 83ec04e8 00000000  ....U..S........
 ...
```

objdump: Looking at ELF files

- Use the `objdump` tool to peek inside of ELF files (.o, .so, and executable files)
- `objdump -t`: print out contents of symbol table

```
$ objdump -t myprog
myprog:     file format elf32-i386

SYMBOL TABLE:
...
0804839c g     F .text   00000022      swap
08049604 g       *ABS*   00000000      __bss_start
08049610 g       *ABS*   00000000      _end
080495fc g     O .data   00000008      buf
08049604 g       *ABS*   00000000      _edata
08048439 g     F .text   00000000      .hidden __i686.get_pc_thunk.bx
08048374 g     F .text   00000028      main
08048250 g     F .init   00000000      _init
```

Today

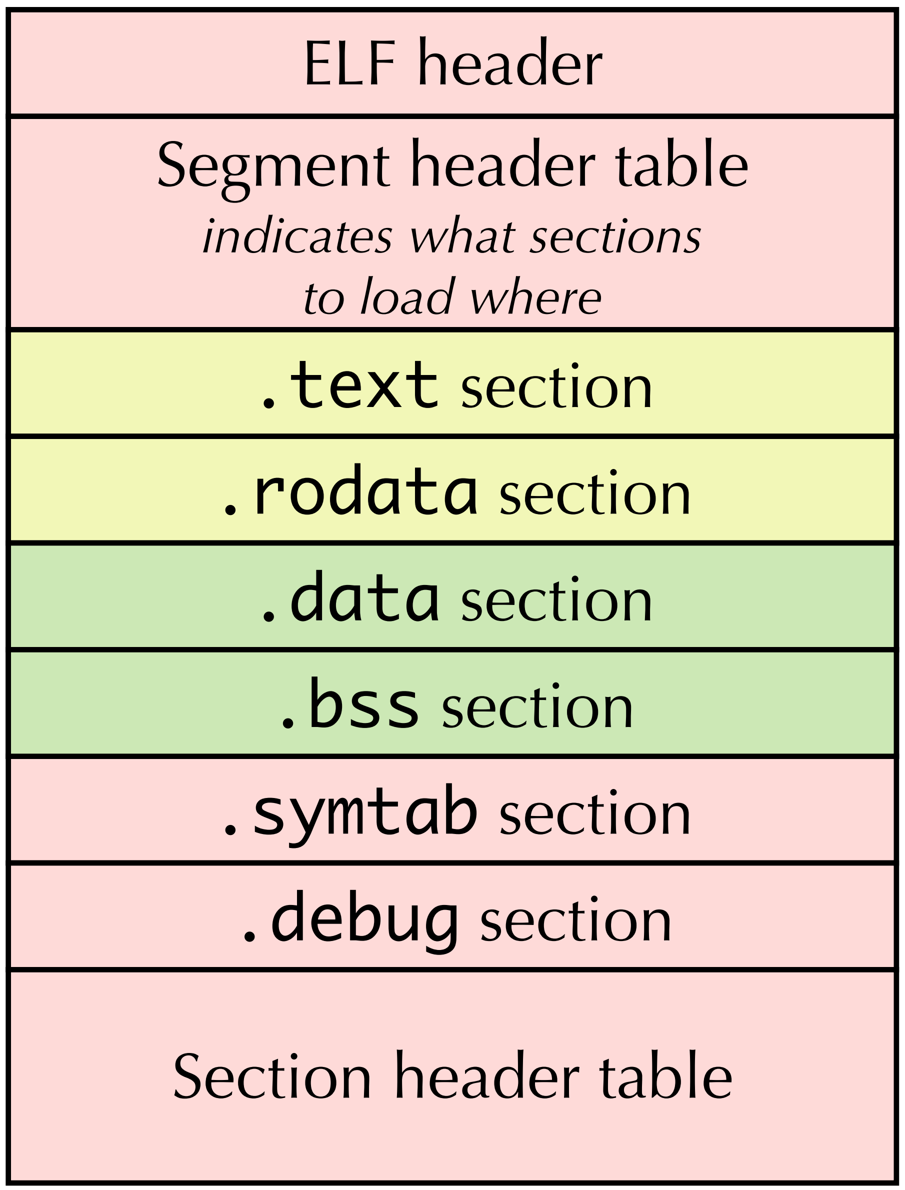
- Getting from C programs to executables
  - The what and why of linking
  - Symbol relocation
  - Symbol resolution
  - ELF format
  - Loading
  - Static libraries
  - Shared libraries

Loading an executable

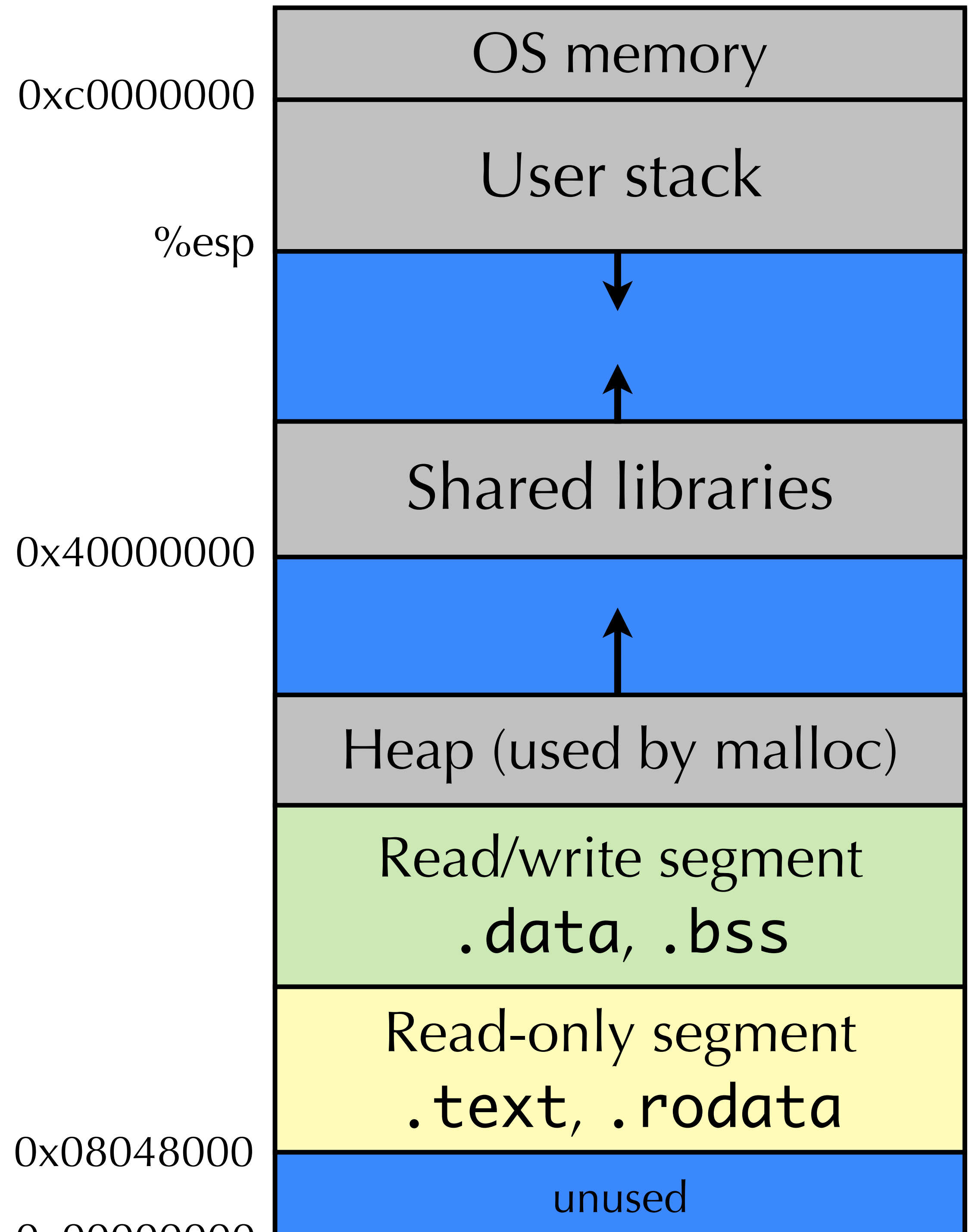
- **Loading** is the process of reading code and data from an executable file and placing it in memory, where it can run.
  - This is done by the operating system.
  - In UNIX, you can use the `execve()` system call to load and run an executable.
- What happens when the OS runs a program:
  - 1) Create a new **process** to contain the new program (more later this semester!)
  - 2) Allocate memory to hold the program code and data
  - 3) Copy contents of executable file to the newly-allocated memory
  - 4) Jump to the executable's **entry point** (which calls the `main()` function)

Address space layout

Executable file

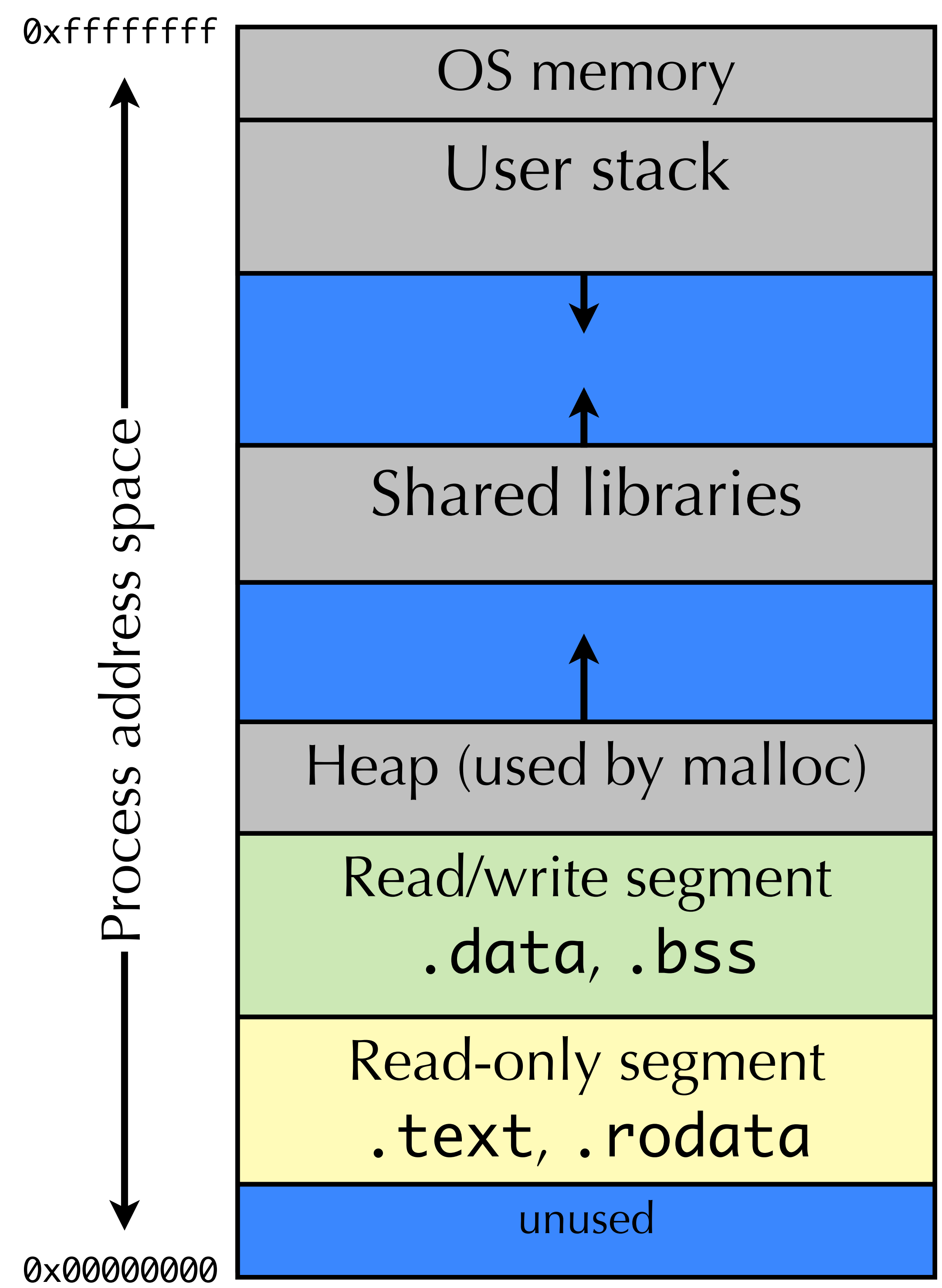


Process address space



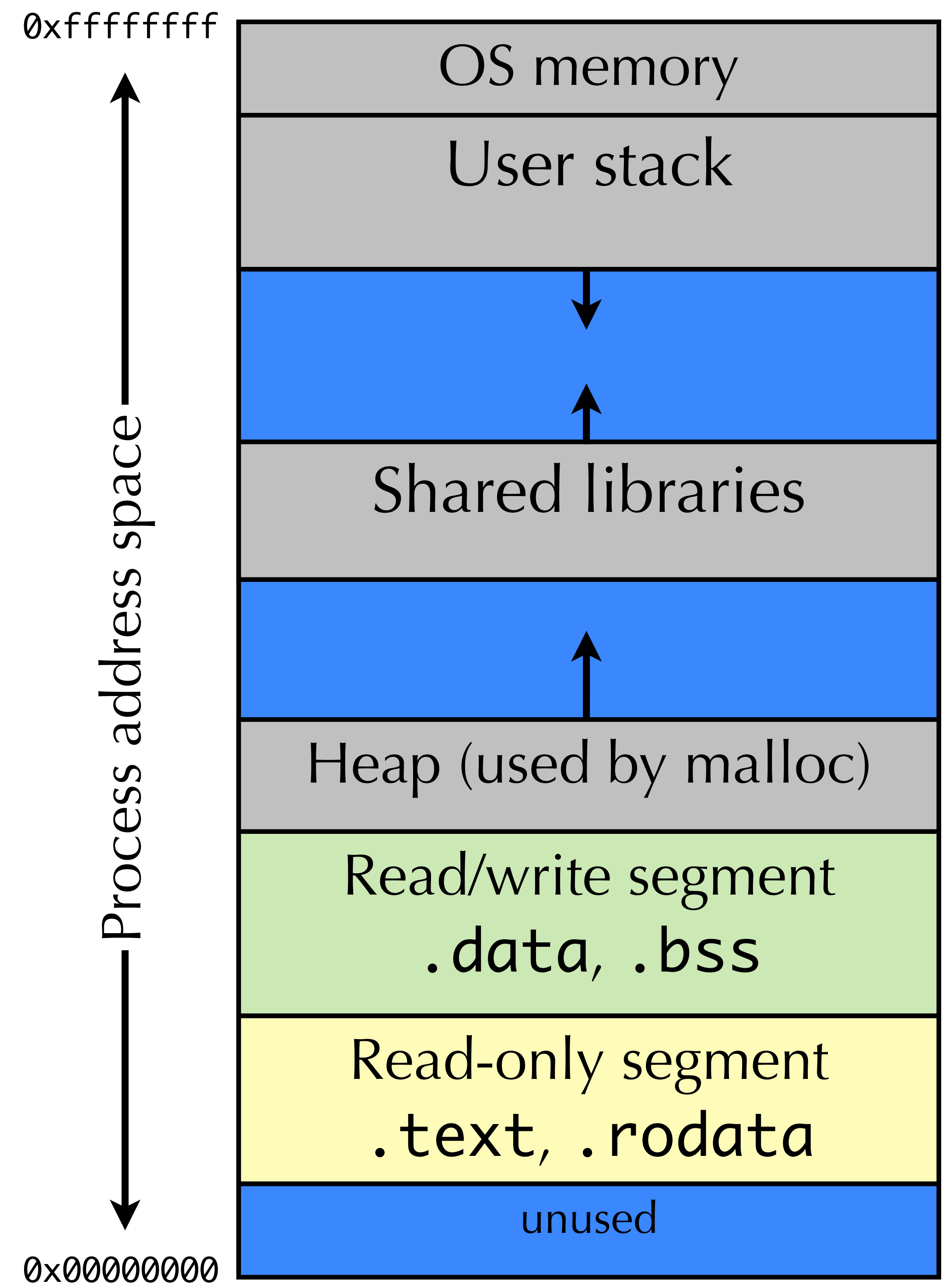
Virtual memory simplifies linking

- Linking
  - Each program has similar virtual address space
  - Code, stack, and shared libraries always start at the same address



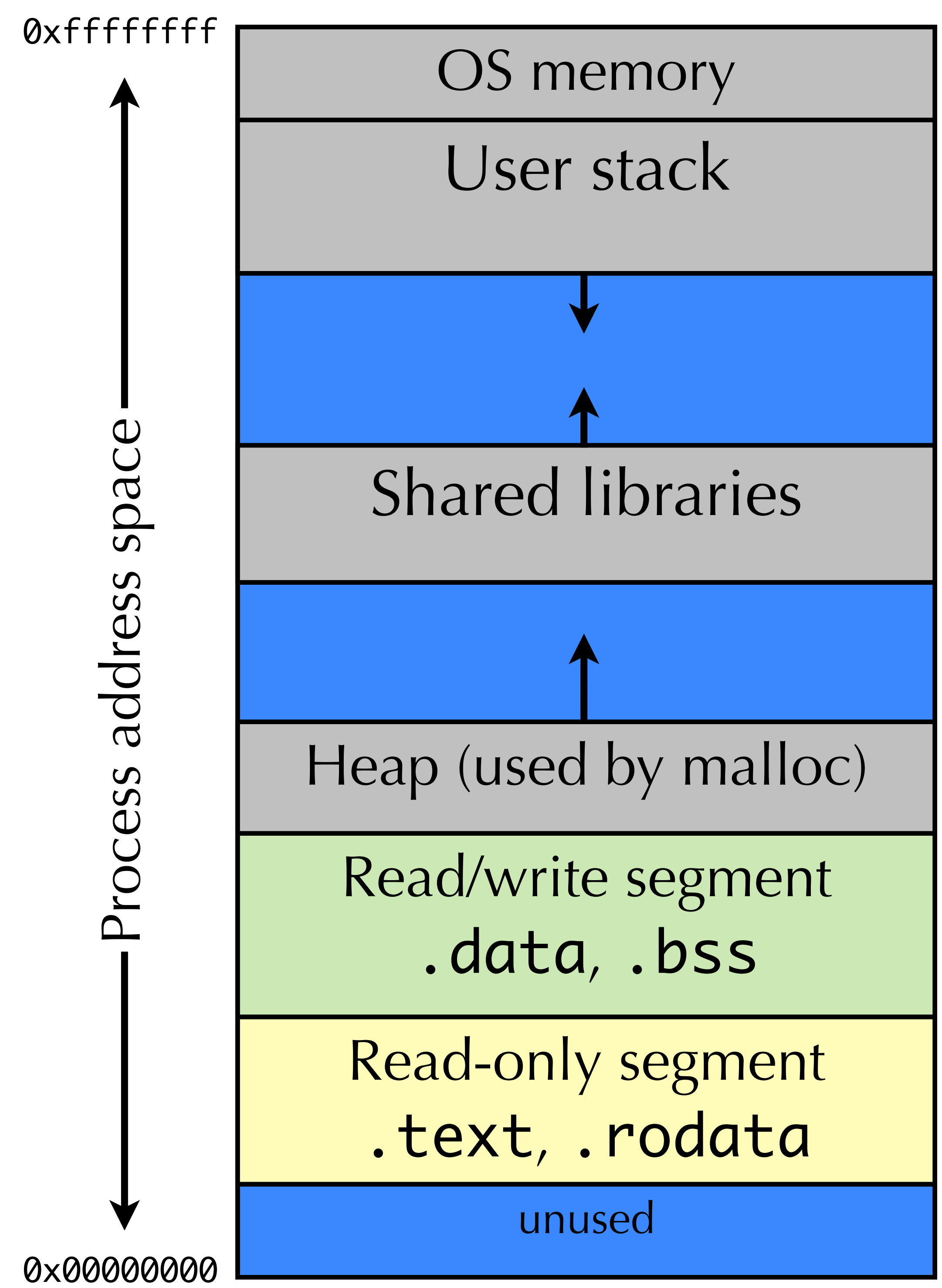
Virtual memory simplifies loading

- Doesn't make sense to immediately load all of a program into physical memory
  - Lots of code and data may not be used
  - E.g., rarely used functionality
- Initially, page table maps data, text, rodata, etc., segments to disk
  - i.e., PTE are marked as invalid
- As segments are used (e.g., code accessed for the first time), page fault will bring them into physical memory
- **Demand paging**: pages copied into memory only when they are needed



Demand-zero pages

- When page from bss segment (and new memory for the heap, stack, etc.) faulted for the first time, physical page of all zeros allocated.
  - Once page is written, like any other page.
  - **"Demand-zero" pages**



Today

- Getting from C programs to executables
  - The what and why of linking
  - Symbol relocation
  - Symbol resolution
  - ELF format
  - Loading
  - Static libraries
  - Shared libraries

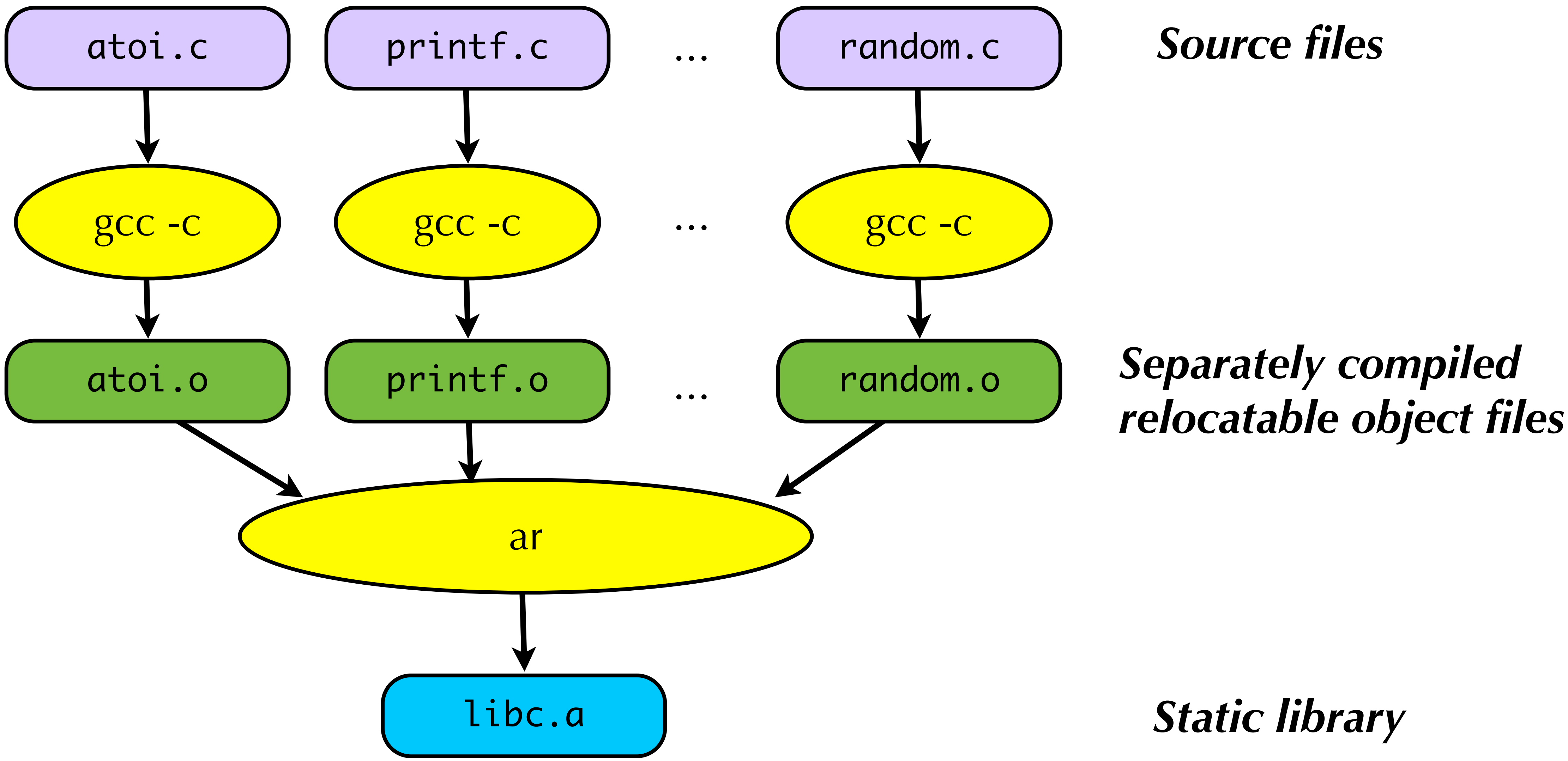
Packaging Commonly Used Functions

- How to package functions commonly used by programmers?
  - `printf`, `malloc`, `strcmp`, all that stuff?
- 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:
  - `gcc -o myprog myprog.o somebighonkinglibraryfile.o`
    - This would be really inefficient!
- **Option 2:** Put each routine in a separate object file
  - Programmers explicitly link appropriate object files into their programs
  - `gcc -o myprog myprog.o printf.o malloc.o free.o strcmp.o strlen.o strerr.o .....`
    - More efficient, but a real pain to the programmer

Static Libraries

- **Solution: Static libraries**
  - Combine multiple object files into a single archive file (file extension "`.a`")
  - Example: `libc.a` contains a whole slew of object files bundled together.
- Linker can also take archive files as input:
  - `gcc -o myprog myprog.o /usr/lib/libc.a`
  - Linker searches the `.o` files within the `.a` file for needed references and links them into the executable.

Creating Static Libraries



- Create a static library file using the UNIX `ar` command
  - `ar rs libc.a atoi.o printf.o random.o ...`
- Can list contents of a library using `ar t libc.a`

Commonly Used Libraries

- `libc.a` (the C standard library)
  - 2.8 MB archive of 1400 object files.
  - I/O, memory allocation, signal handling, string handling, data and time, random numbers, integer math
- `libm.a` (the C math library)
  - 0.5 MB archive of 400 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
…
```

Using Static Libraries

- Linker's algorithm for resolving external references:
  - Scan `.o` files and `.a` files in 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 in the new file.
  - If any entries unresolved when done, then return an error.
- Problem:
  - Command line order matters!
  - Moral: put libraries at the end of the command line.

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

unix> gcc libtest.o mylib.a
# works fine!!!
```

Shared Libraries

- Static libraries have the following disadvantages:
  - Lots of code duplication in the resulting executable files
    - Every C program needs the standard C library.
    - e.g., Every program calling `printf()` would have a copy of the `printf()` code in the executable. Very wasteful!
  - Lots of code duplication in the memory image of each running program
    - OS would have to allocate memory for the standard C library routines being used by every running program!
  - Any changes to system libraries would require relinking every binary!
- Solution: **Shared libraries**
  - Libraries that are linked into an application dynamically, when a program runs!
  - On UNIX, "`.so`" filename extension is used
  - On Windows, "`.dll`" filename extension is used

Shared Libraries

- When the OS runs a program, it checks whether the executable was linked against any shared library (`.so`) files.
  - If so, it performs the linking and loading of the shared libraries on the fly.
- Example: `gcc -o myprog main.o /usr/lib/libc.so`
- Can use UNIX `ldd` command to see which shared libs an executable is linked against

```
unix> ldd myprog

linux-gate.so.1 =>  (0xffffe000)
libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7df5000)
/lib/ld-linux.so.2 (0xb7f52000)
```

- Can create your own shared libs using `gcc -shared`
  - `gcc -shared -o mylib.so main.o swap.o`

Runtime dynamic linking

- You can also link against shared libraries at run time!
- Three UNIX routines used for this:
  - `dlopen()` – Open a shared library file
  - `dlsym()` – Look up a symbol in a shared library file
  - `dlclose()` – Close a shared library file
- Why would you ever want to do this?
  - Can load new functionality on the fly, or extend a program after it was originally compiled and linked.
  - Examples include things like browser plug-ins, which the user might download from the Internet well after the original program was compiled and linked.

Dynamic linking example

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

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

    /* dynamically load the shared
     * lib that contains somefunc() */
    handle = dlopen("./libvector.so", RTLD_LAZY);
    if (!handle) {
     fprintf(stderr, "%s\n", dlerror());
     exit(1);
    }
    /* get a pointer to the somefunc()
     * function we just loaded */
    somefunc = dlsym(handle, "somefunc");
    if ((error = dlerror()) != NULL) {
     fprintf(stderr, "%s\n", error);
     exit(1);
    }

    /* Now we can call somefunc() just
     * like any other function */
    somefunc(42, 38);

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