CS61: Systems Programming and Machine Organization
Harvard University, Fall 2009

Lecture 8:
Linking and Loading

Prof. Matt Welsh
September 29, 2009

Topics for today

- Getting from C programs to running executables
- Linking:
  - Process of combining multiple code modules into a runnable program
  - Example: Using standard library code (printf, malloc, etc.) in your own program
- Loading:
  - Process of getting an executable running on the machine
  - Requires resolving references to external symbols in the program (**dynamic linking**)

Example program with two C files

main.c

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

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

swap.c

```c
extern int buf[];

void swap()
{
  int temp;

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

Example program with two C files

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;
}
```

Definition of buf[]

Definition of main()

__Declaration__ of buf[]

Definition of swap()

Example program with two C files

main.c

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

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

Reference to swap()

swap.c

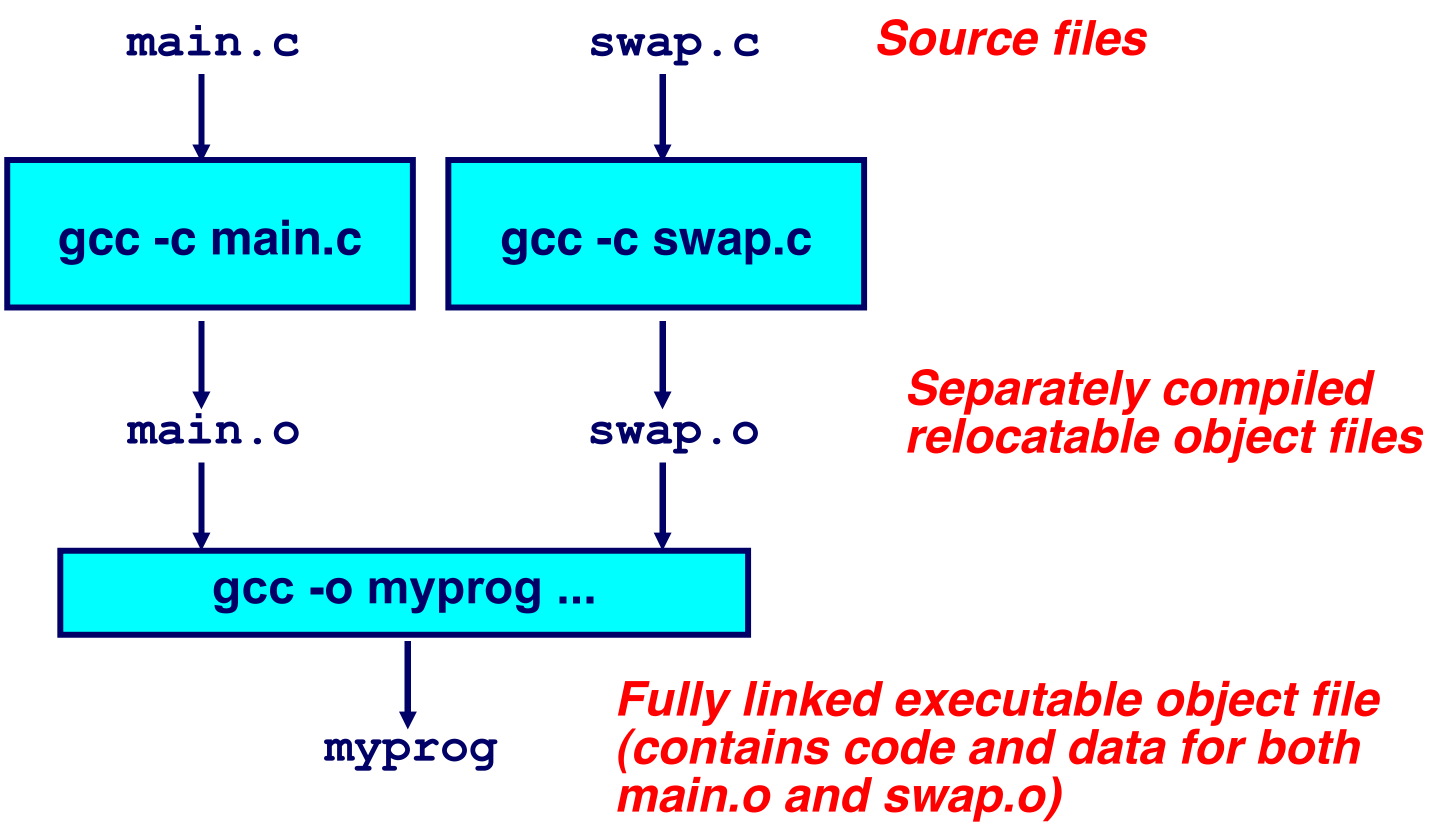
```
extern int buf[];

void swap()
{
  int temp;

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

References to buf[]

Static Linking

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

- `unix> gcc -c main.c                      # Compile .c to .o`
- `unix> gcc -c swap.c`
- `unix> gcc -o myprog main.o swap.o        # This runs ld with`
  `                                         # the right flags`



The linker's problem...

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;
}
```

- Find the definition of each **undefined reference** in an object file
  - Example: The use of “swap()” in main.o needs to be resolved to the definition found in swap.o
- **Resolve** all missing references in the code
  - We can compile main.c to main.o without knowing the memory location of swap()
  - So, main.o will contain a “dummy” address in the `call` instruction.
  - The linker must patch the assembly code in main.o to reference the correct location of swap() at link time.

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 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.

Disassembly of main.o

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

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

```
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;
}
```

main.o

.text section

code for main()

.data section

buf[]

symbol table

name	section	off
main	.text	0
buf	.data	0
swap	**undefined**	

relocation info for .text

name	offset
swap	12

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

Disassembly of swap.o

```
extern int buf[];

void swap()
{
  int temp;

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

}
```

Again, these are placeholders.

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

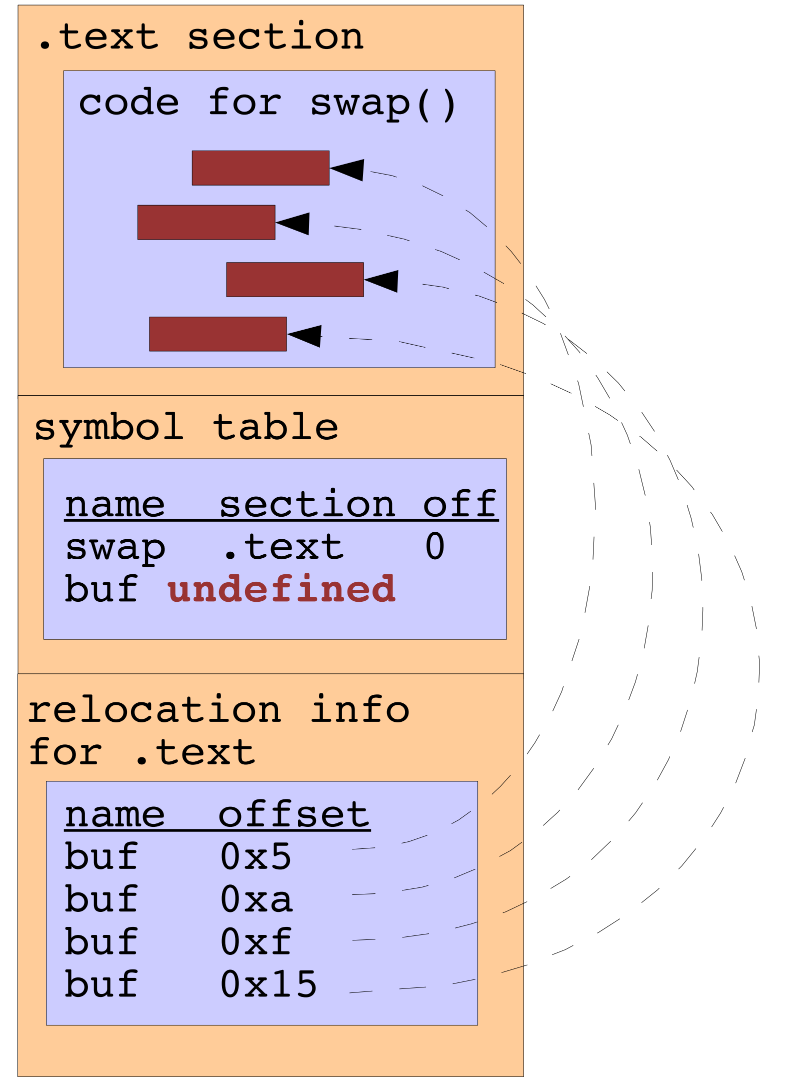
```
00000000 <swap>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   8b 15 04 00 00 00       mov    0x4,%edx
   9:   a1 00 00 00 00          mov    0x0,%eax
   e:   a3 04 00 00 00          mov    %eax,0x4
  13:   89 15 00 00 00 00       mov    %edx,0x0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

swap.o object file

swap.c

```c
extern int buf[];

void swap()
{
  int temp;

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

}
```

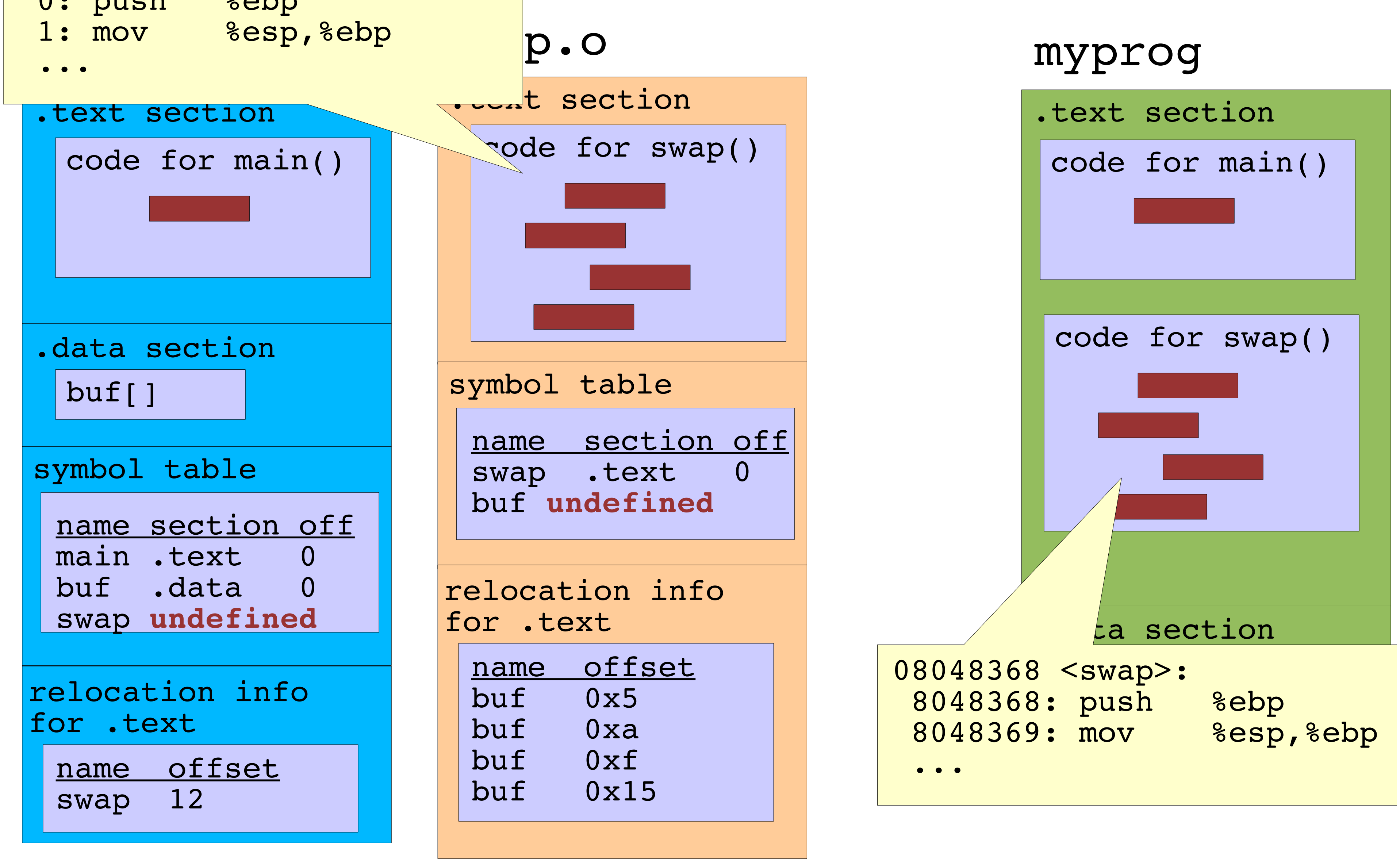
swap.o



The linker

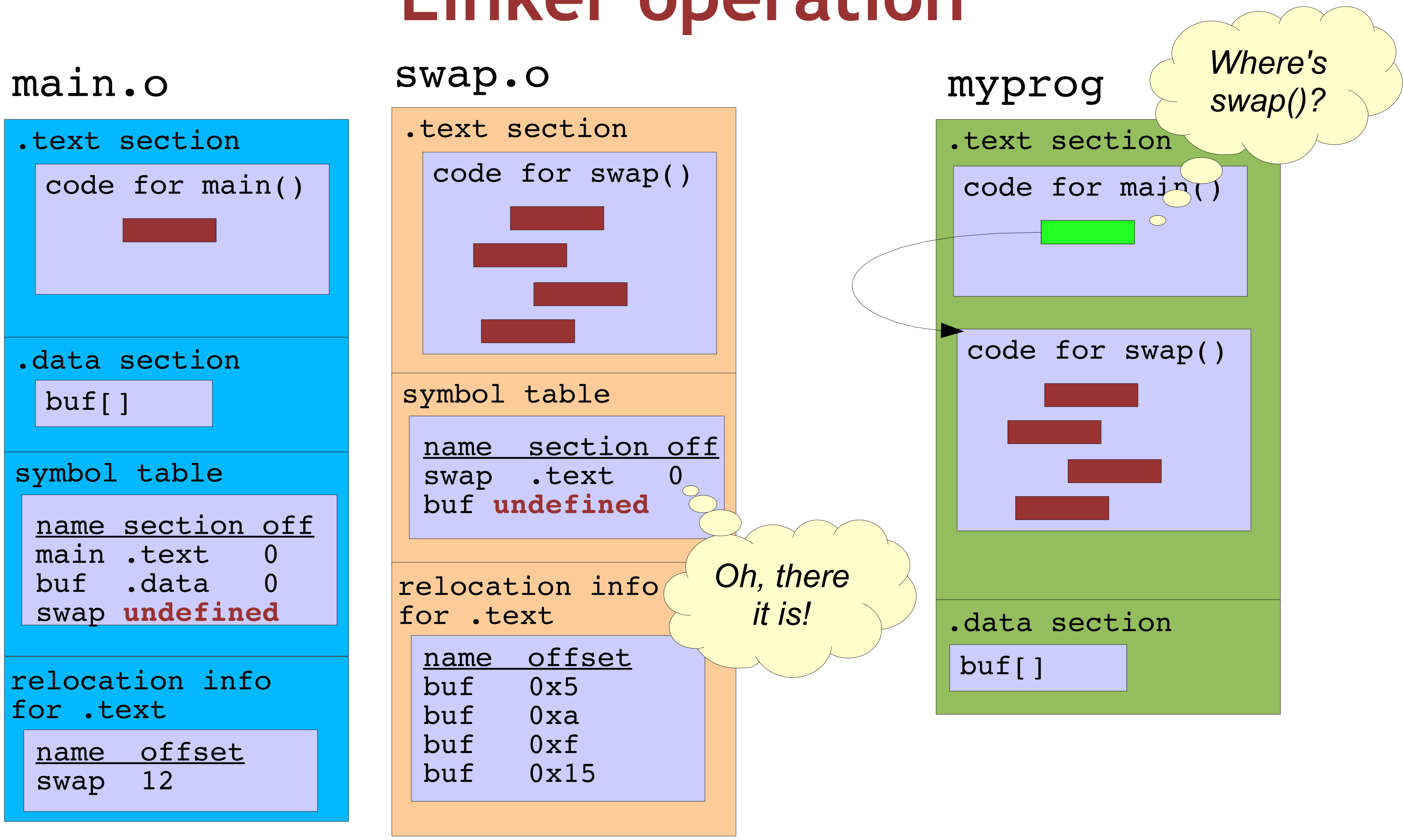
- The linker takes multiple object files and combines them into a single executable file.
- Three basic jobs:
- 1) **Copy** code and data from each object file to the executable
- 2) **Resolve** references between object files
- 3) **Relocate** symbols to use absolute memory addresses, rather than relative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

Linker operation

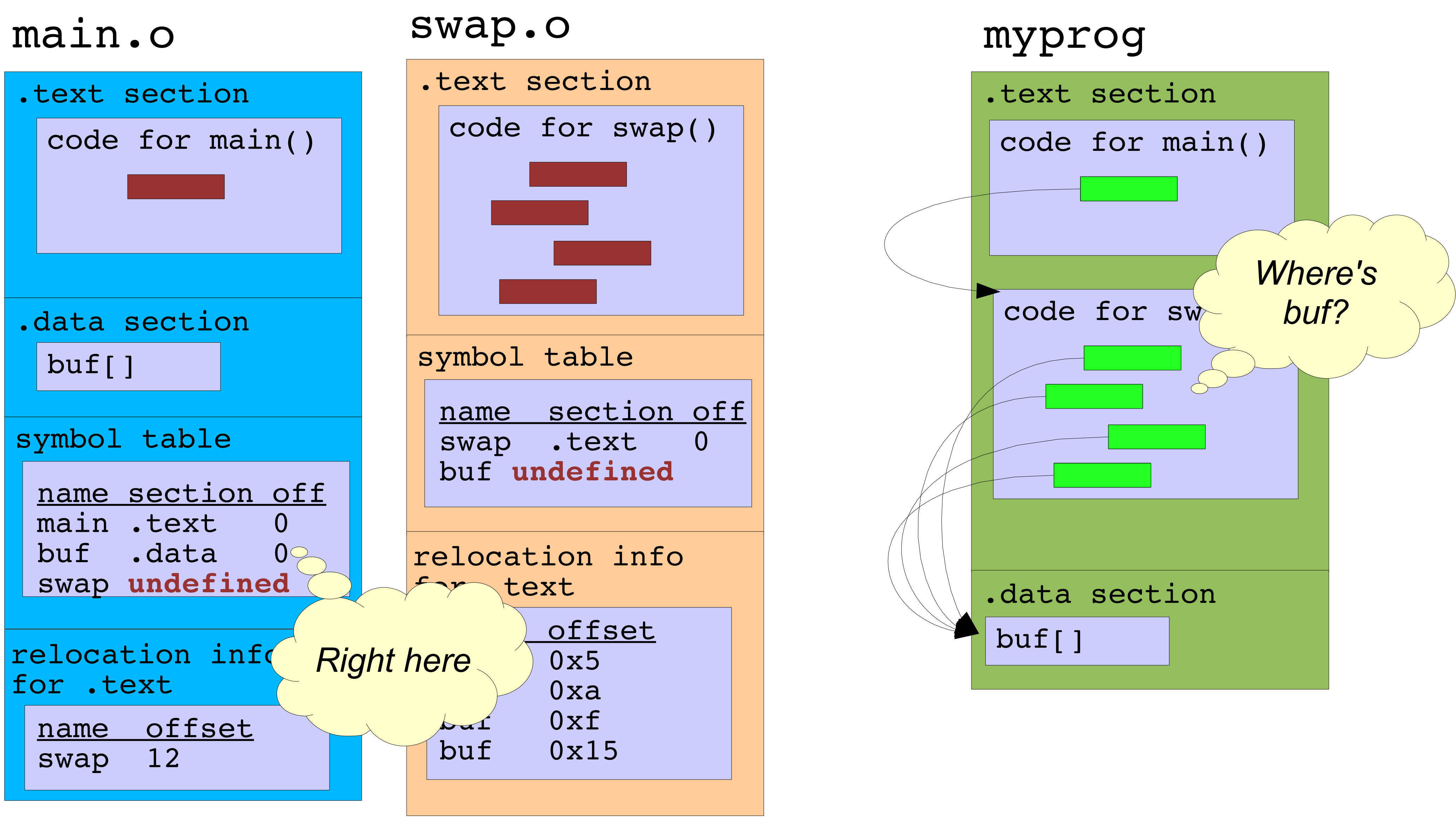


*The executable file contains the **absolute** memory addresses of the code and data (that is, where they will be located when the program actually runs!)*

Linker operation



Linker operation



Disassembly after linking

What's up with this? Not the same as 0x8048368...

```
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
```

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

Address Section Size Symbol name

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

main.c

Strong

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

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

Strong

Weak

Not a global symbol!
(Just a reference.)

swap.c

```
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
ld: duplicate symbol _somefunc in /var/folders/Yt/Yta+
ZMkTGjWQDZZ0edwfCE++UHg/-Tmp-//ccAsmhEJ.o and
/var/folders/Yt/Yta+ZMkTGjWQDZZ0edwfCE++UHg/-Tmp-//ccVxxcKX.o
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
```

You might not expect this to happen but it does!

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
```

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

- Page size, virtual addresses memory segments (sections), segment sizes.

`.text` section

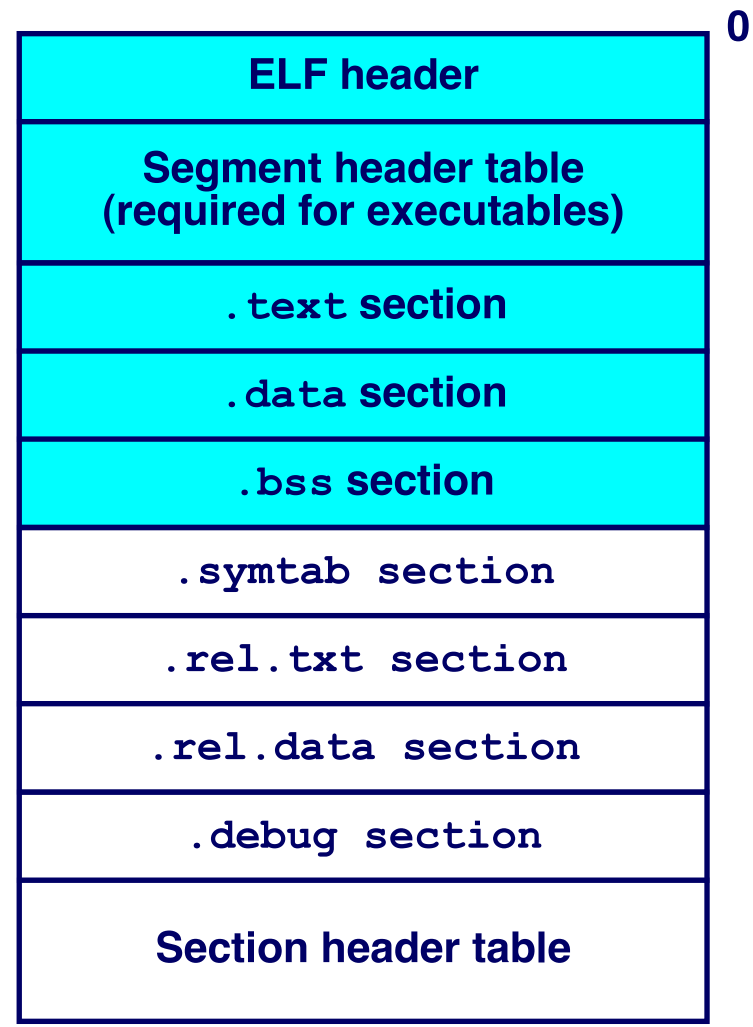
- Code

`.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 Object File Format (cont)

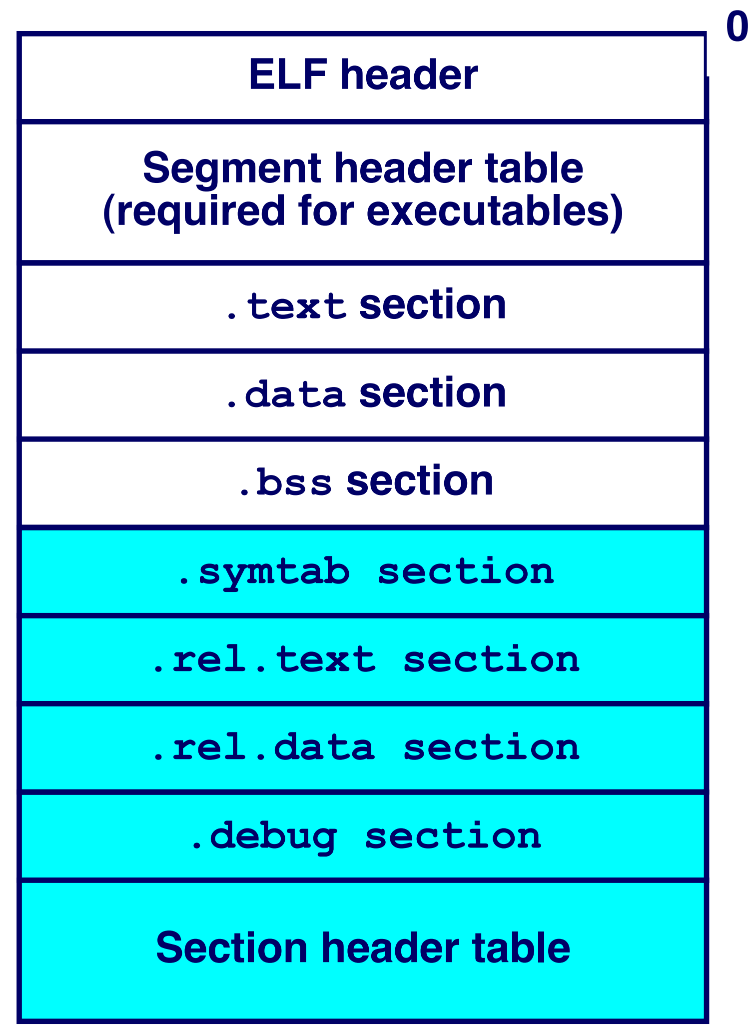
`.symtab` section
- Symbol table
- Procedure and static variable names
- Section names and locations

`.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



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
  0 .interp       00000013  08048134  08048134  00000134  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .note.ABI-tag 00000020  08048148  08048148  00000148  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .hash         00000028  08048168  08048168  00000168  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
...
```

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 .interp:
 8048134 2f6c6962 2f6c642d 6c696e75 782e736f  /lib/ld-linux.so
 8048144 2e3200                               .2.
Contents of section .note.ABI-tag:
 8048148 04000000 10000000 01000000 474e5500  ............GNU.
 8048158 00000000 02000000 06000000 08000000  ................
Contents of section .hash:
 8048168 03000000 05000000 01000000 02000000  ................
 8048178 03000000 00000000 00000000 00000000  ................
```

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:
08048134 l    d  .interp   00000000                    .interp
08048148 l    d  .note.ABI-tag   00000000              .note.ABI-tag
08048168 l    d  .hash  00000000                       .hash
08048190 l    d  .gnu.hash 00000000                    .gnu.hash
...
```

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

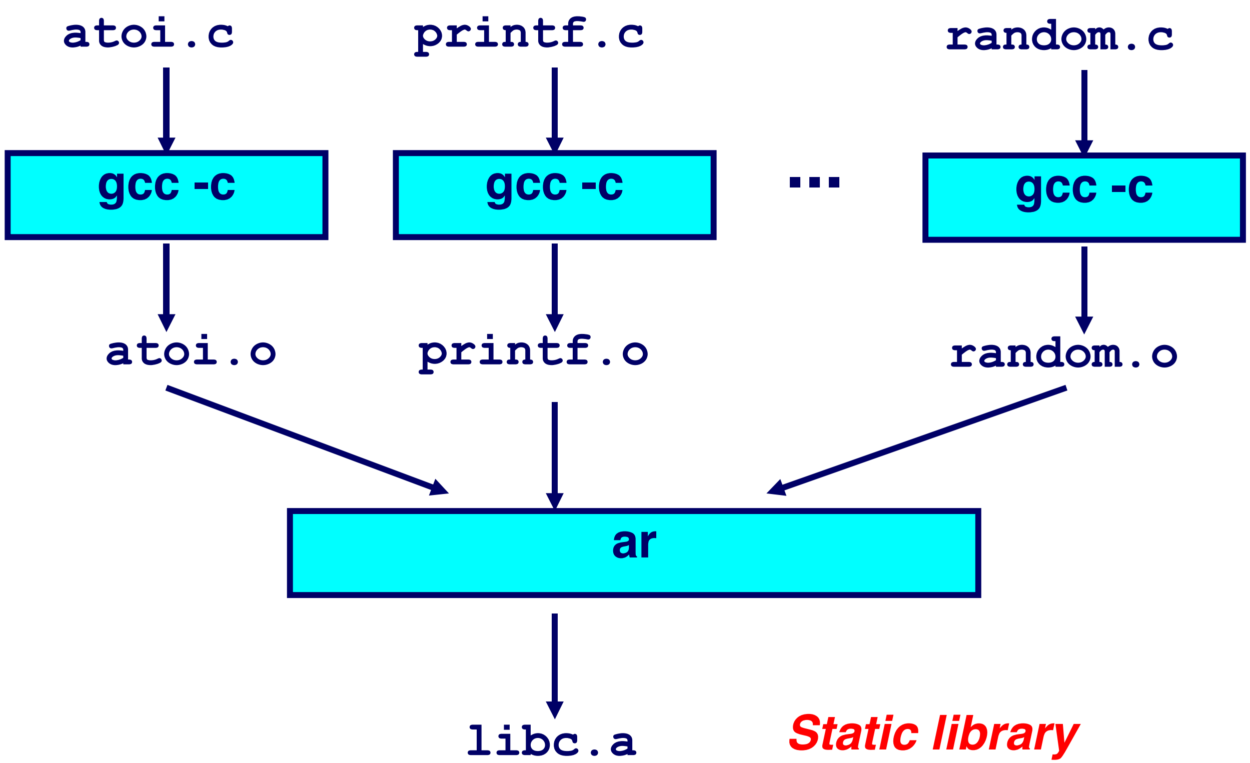
Solution: **Static libraries**

- 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:

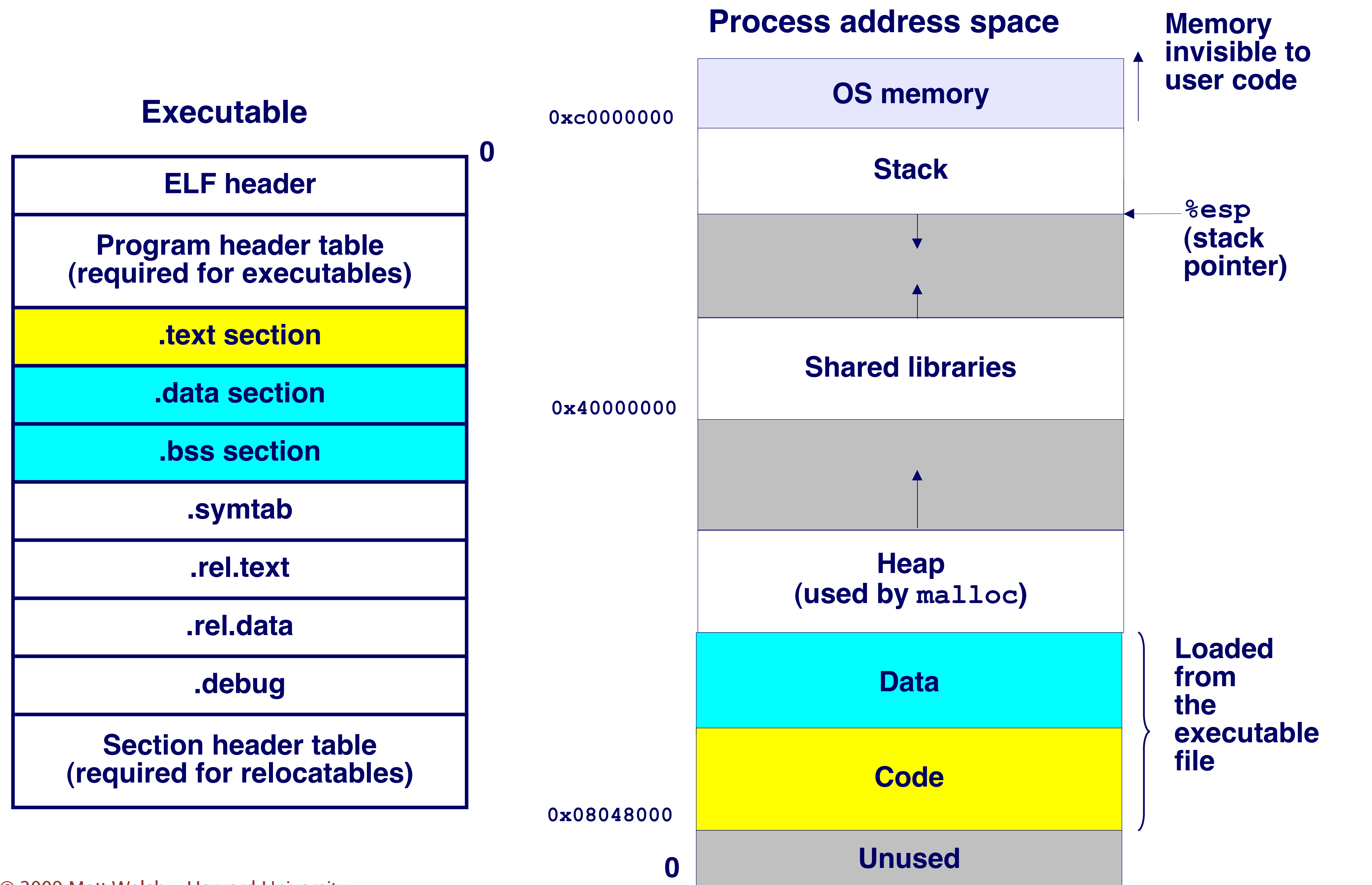
- Command line order matters!
- Moral: put libraries at the end of the command line.

```
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!!!
```

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



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;
}
```

Next time

- Memory and storage technologies
- All about SRAM and DRAM
- How disks work