

15-410

“Nobody reads these quotes anyway...”

Executables **Oct. 7, 2013**

Dave Eckhardt

Todd Mowry

**Some slides taken from 15-213 S'03 (Goldstein, Maggs).
Original slides authored by Randy Bryant and Dave O'Hallaron.**

Pop Quiz

Q0. What is “BSS”?

Q1. What does the Unix “ld” program do?

Q2. What does “ld” stand for?

Outline

Where addresses come from

Executable files vs. Memory Images

- Conversion by “program loader”
- You will write one for `exec()` in Project 3

Object file linking (answer to Q2)

- Loader bugs make programs execute *half*-right
- You will need to characterize what's broken
 - (*Not*: “every time I call `printf()` I get a triple fault”)
- You will need to how the parts *should* fit together

Who emits addresses?

Program linking, program loading

- ... means getting bits in memory at the right addresses

Who *uses* those addresses?

- (Where did that “wild access” come from?)

Code addresses: program counter (%cs:%eip)

- Straight-line code
- Loops, conditionals
- Procedure calls

Stack area: stack pointer (%ss:%esp, %ss:%ebp)

Data regions (data/bss/heap)

- Most pointers in general purpose registers (%ds:%ebx)

Initialized how?

Program counter

- Set to “entry point” by OS program loader

Stack pointer

- Set to “top of stack” by OS program loader

Registers

- How does my code know the address of `thread_table[]`?
- Some pointers are stored in the instruction stream

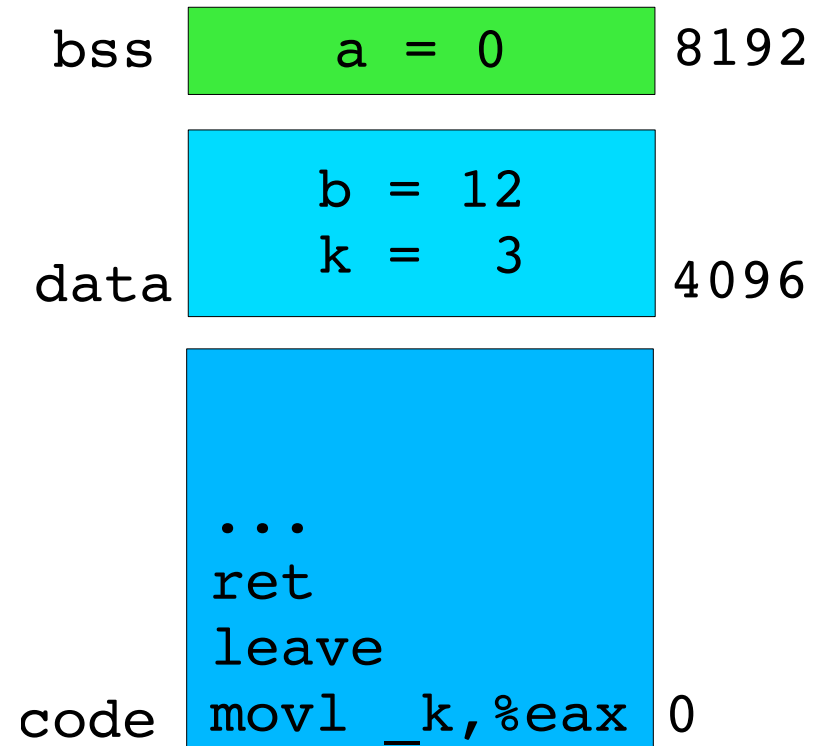
```
for (tp = thread_table,
     tp < &thread_table[n_threads], ++tp)
```
- Some pointers are stored in the data segment

```
struct thread *thr_base = &thread_table[0];
```
- How do these all point to the right places?

Where does an int live?

```
int k = 3;
int foo(void) {
    int shh = 99;
    return (k);
}

int a = 0;
int b = 12;
int bar (void) {
    return (a + b);
}
```



Loader: Image File \Rightarrow Memory Image

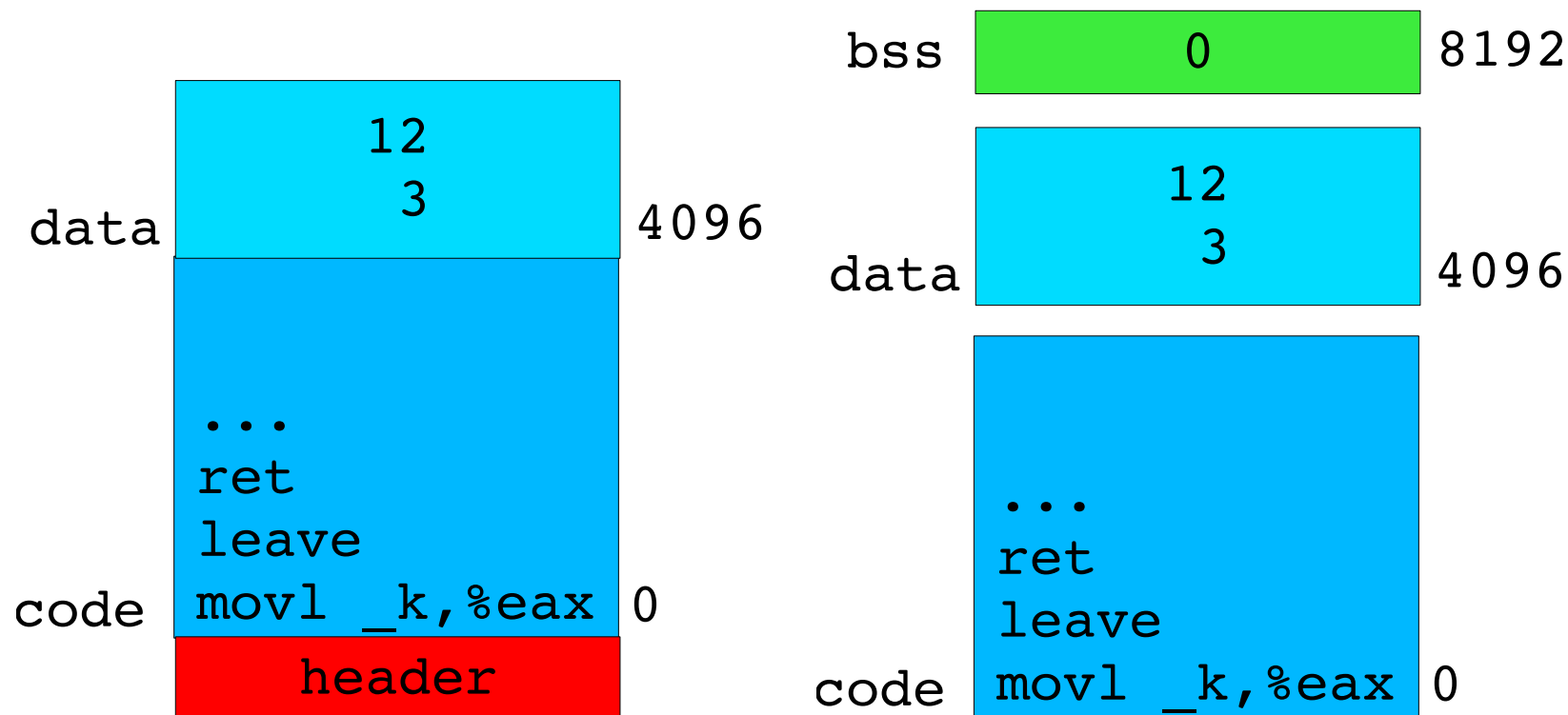


Image file has header (tells loader what to do)
Memory image has bss segment!

Programs are Multi-part

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

Efficiency (time)

- Change one source file, compile, and then relink.
- No need to recompile other source files.

“Link editor” combines objects into one image file

- Unix “link editor” called “ld”

Linker Todo List

Merge object files

- Merges multiple relocatable (.o) object files into a single executable object file that can be loaded and executed by the loader.

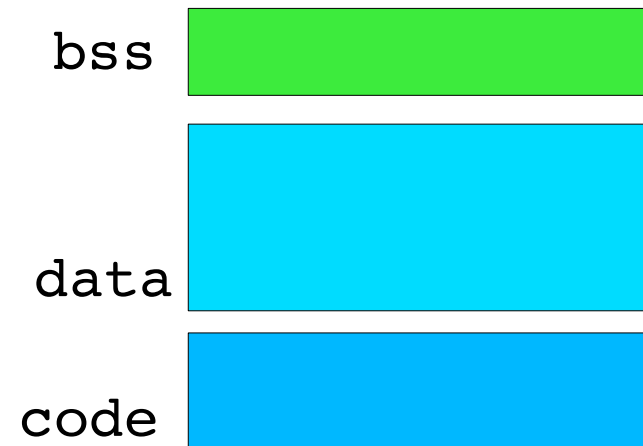
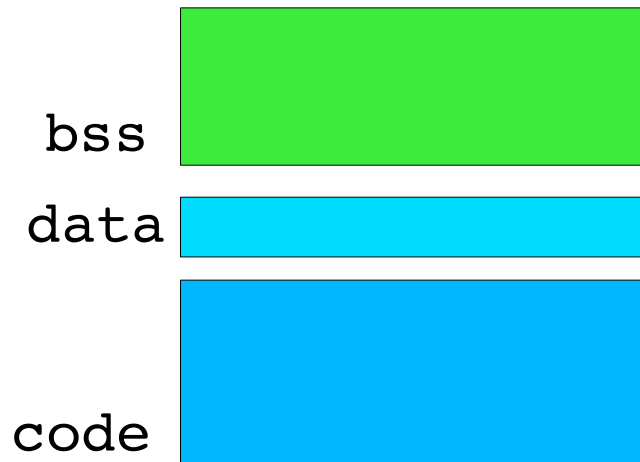
Resolve external references

- *External reference*: reference to a symbol defined in another object file.

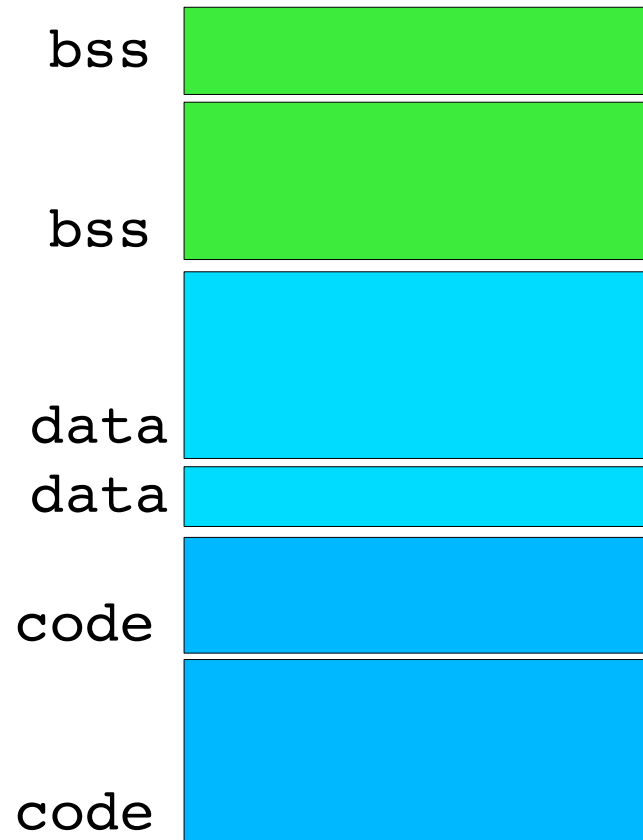
Relocate symbols

- Relocates symbols from their relative locations in the .o files to new absolute positions in the executable.
- Updates all references to these symbols to reflect their new positions.
- What does this mean??

Every .o uses same address space

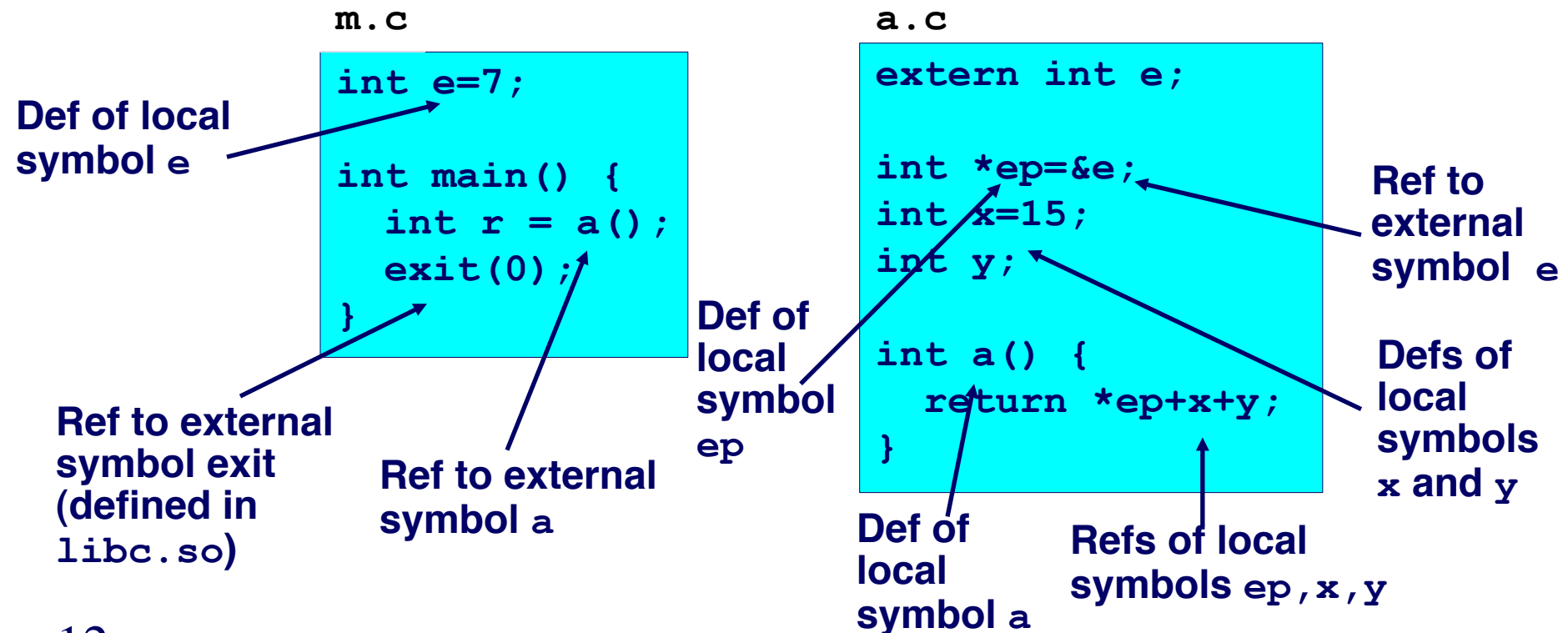


Combining .o's Changes Addresses



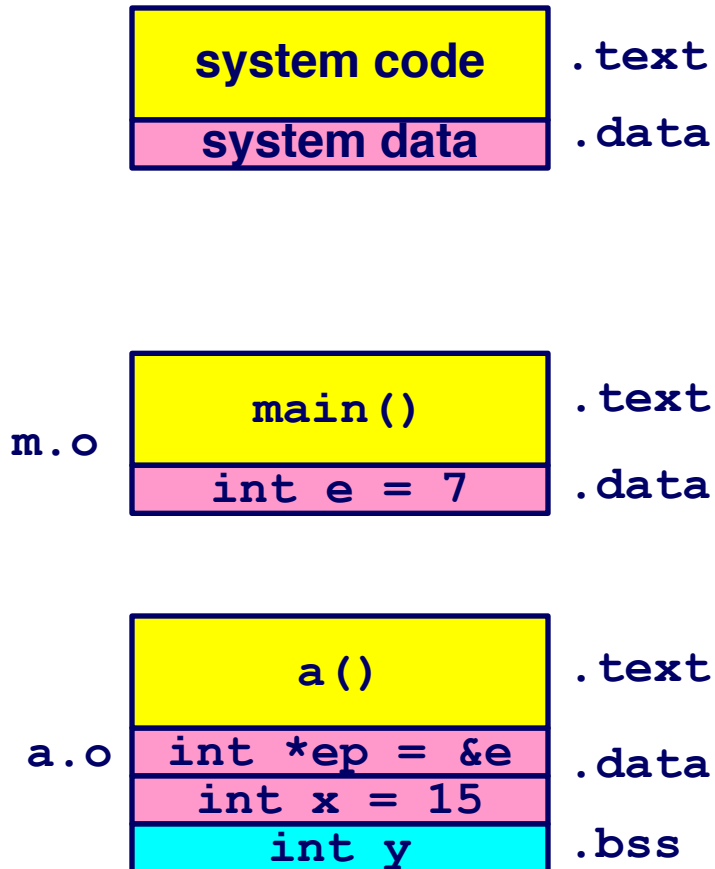
Relocating Symbols and Resolving External References

- **Symbols** are lexical entities that name functions and variables.
- Each symbol has a **value** (typically a memory address).
- Code consists of symbol **definitions** and **references**.
- References can be either **local** or **external**.

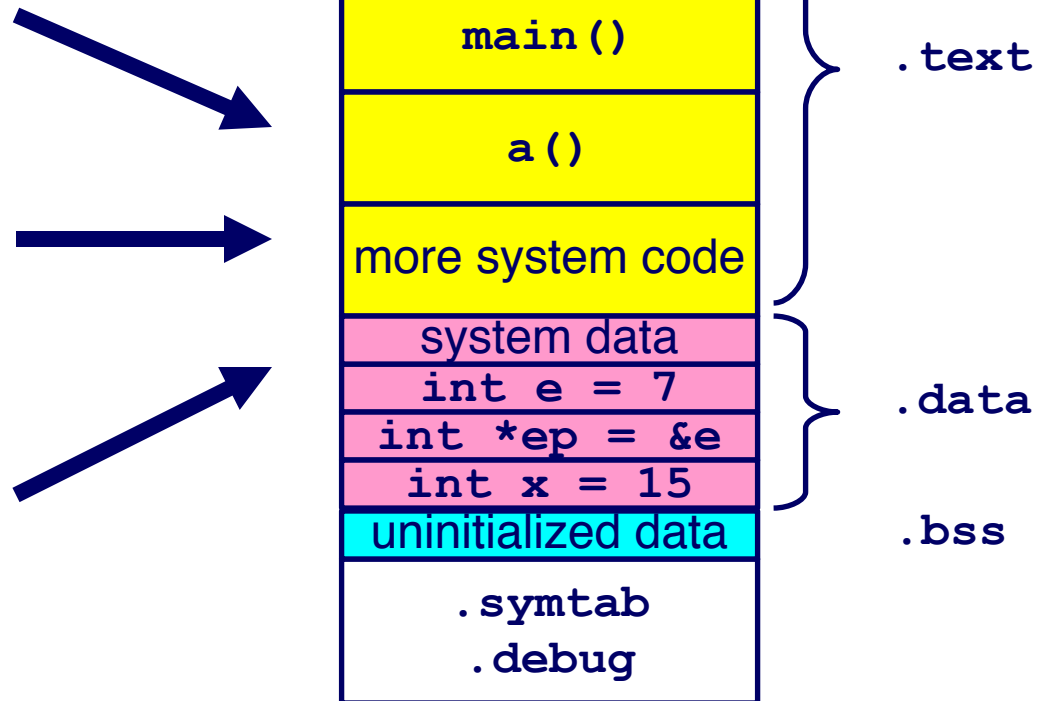


Merging Relocatable Object Files ⇒ Executable Object File

Relocatable Object Files



Executable Object File



Linker uses *relocation information*

Object file contains a “relocation table”

Each table entry specifies one “relocation field” in object

Example

- “Bytes 1024..1027 of foo.o refer to absolute address of `_main`”
 - `_main` may be a file-local symbol or an external symbol
- If `_main` moves, all fields referring to it must be patched

Field properties

Location

- offset, bit field size

Type

- relative, absolute

Reference

- symbol name (string)

Executable File / Image File

Linked program consists of multiple “sections”

- Section properties
 - Type
 - Memory address

Common Executable File Formats

- a.out - “assembler output” (primeval Unix format: 70's, 80's)
- Mach-O – Mach Object (used by MacOS X)
- ELF – Executable and Linking Format

Executable File / Image File

Linked program consists of multiple “sections”

- Section properties
 - Type
 - Memory address

Common Executable File Formats

- a.out - “assembler output” (primeval Unix format: 70's, 80's)
- Mach-O – Mach Object (used by MacOS X)
- ELF – Executable and Linking Format
 - (includes “DWARF” - Debugging With Attribute Record Format)

Executable and Linkable Format (ELF)

Standard binary format for object files

Derives from 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

Better support for shared libraries than old a.out formats.

ELF Object File Format

ELF header

- Magic number, type (.o, exec, .so), machine, byte ordering, etc.

Program header table

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

.text section

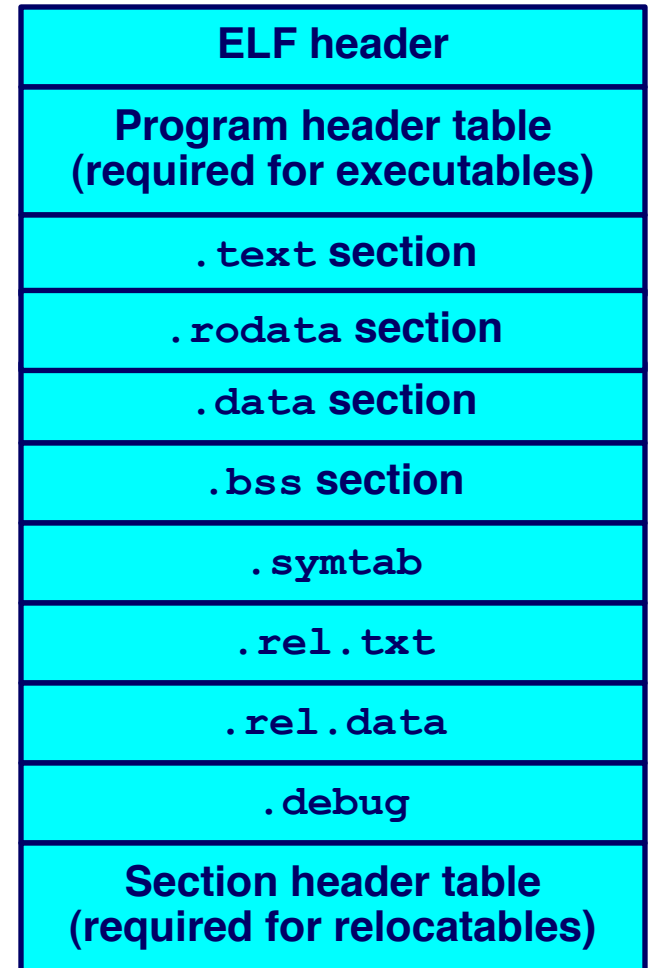
- Code

.rodata, .data section

- Initialized (static) data (ro = “read-only”)

.bss section

- Uninitialized (static) data
- “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

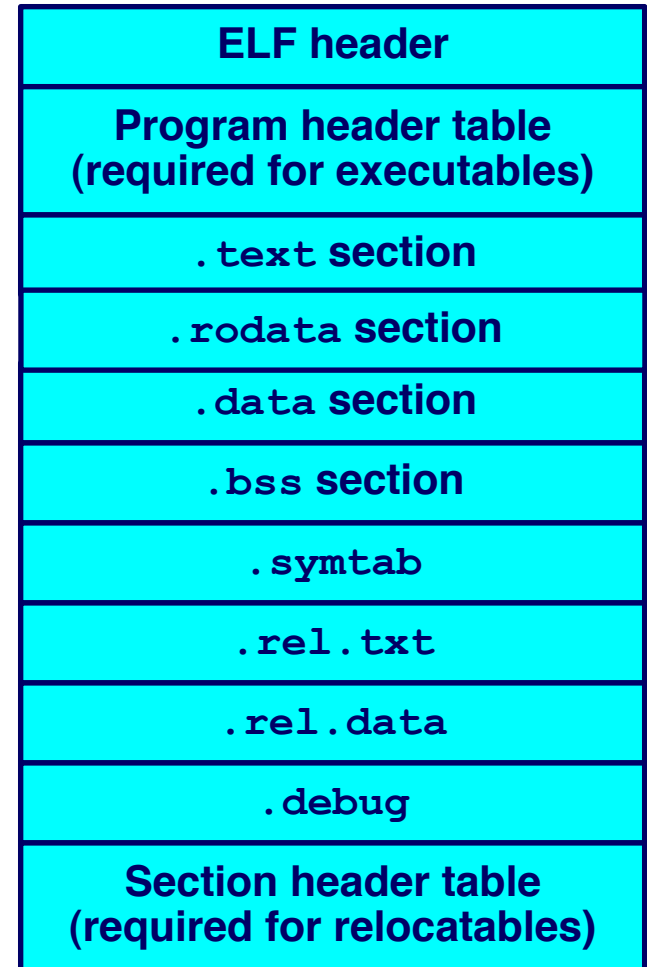
- Relocation info for .text section
- Addresses of instructions that will need to be modified in the executable
- Instructions for modifying.

.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`)



“Not Needed on Voyage”

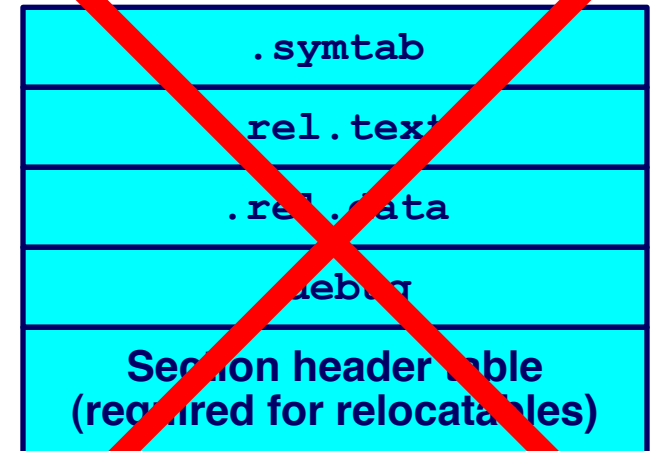
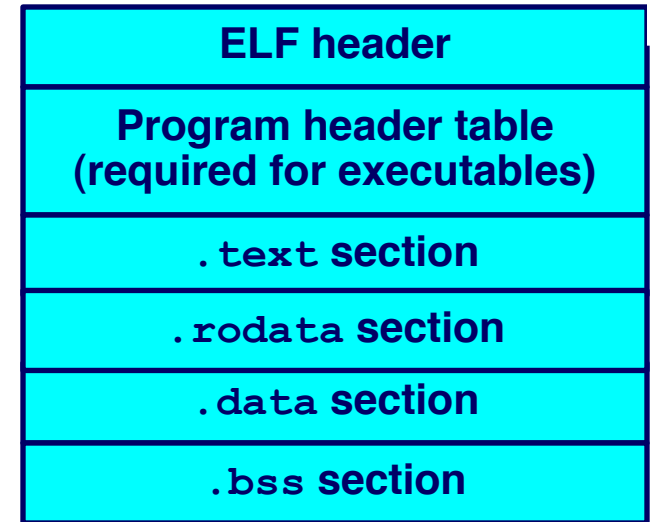
Some sections not needed for execution

- Symbol table
- Relocation information
- Symbolic debugging information

These sections not loaded into memory

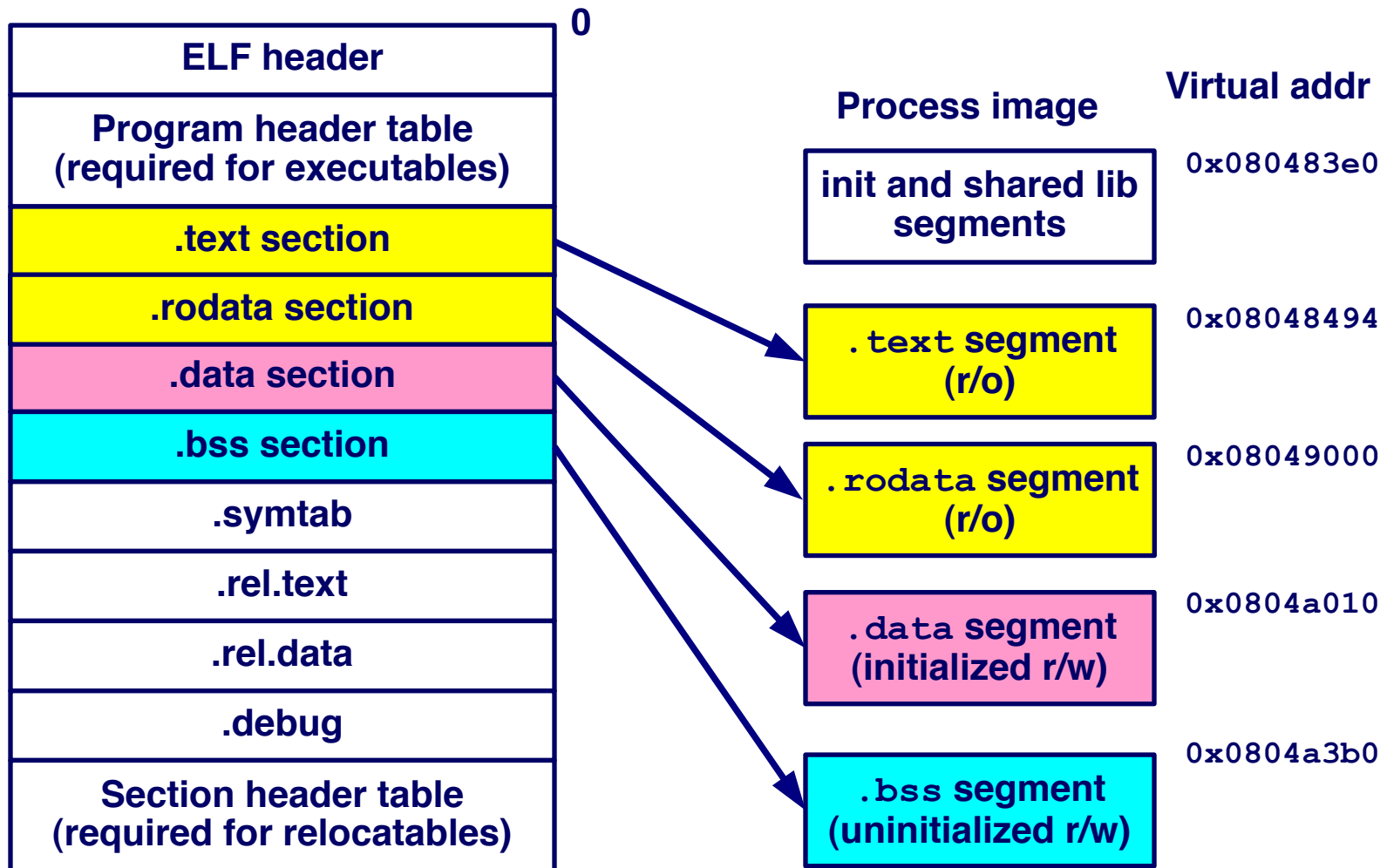
May be removed with “strip” command

- Or retained for future debugging



Loading ELF Binaries

Executable object file for
example program p



Getting Help

Writing your first loader should be fun

- But some parts might be “fun” instead

A tool you can use

- `gdb`
 - `% gdb 410user/progs/init`
 - `(gdb) x/i main`
`0x1000020 <main>: push %ebp`
 - `(gdb) x/x main`
`0x1000020 <main>: 0x83e58955`
- Ok, now you have a cross-check!

Other tools which tell you where executable parts belong

- `nm -n`
- `objdump`

Summary

Where do addresses come from?

Where does an int live?

Image file vs. Memory image

Linker

- What, why
- Relocation

ELF structure

- The pieces which need to be loaded into memory by somebody
 - Somebody whose name is a lot like yours...