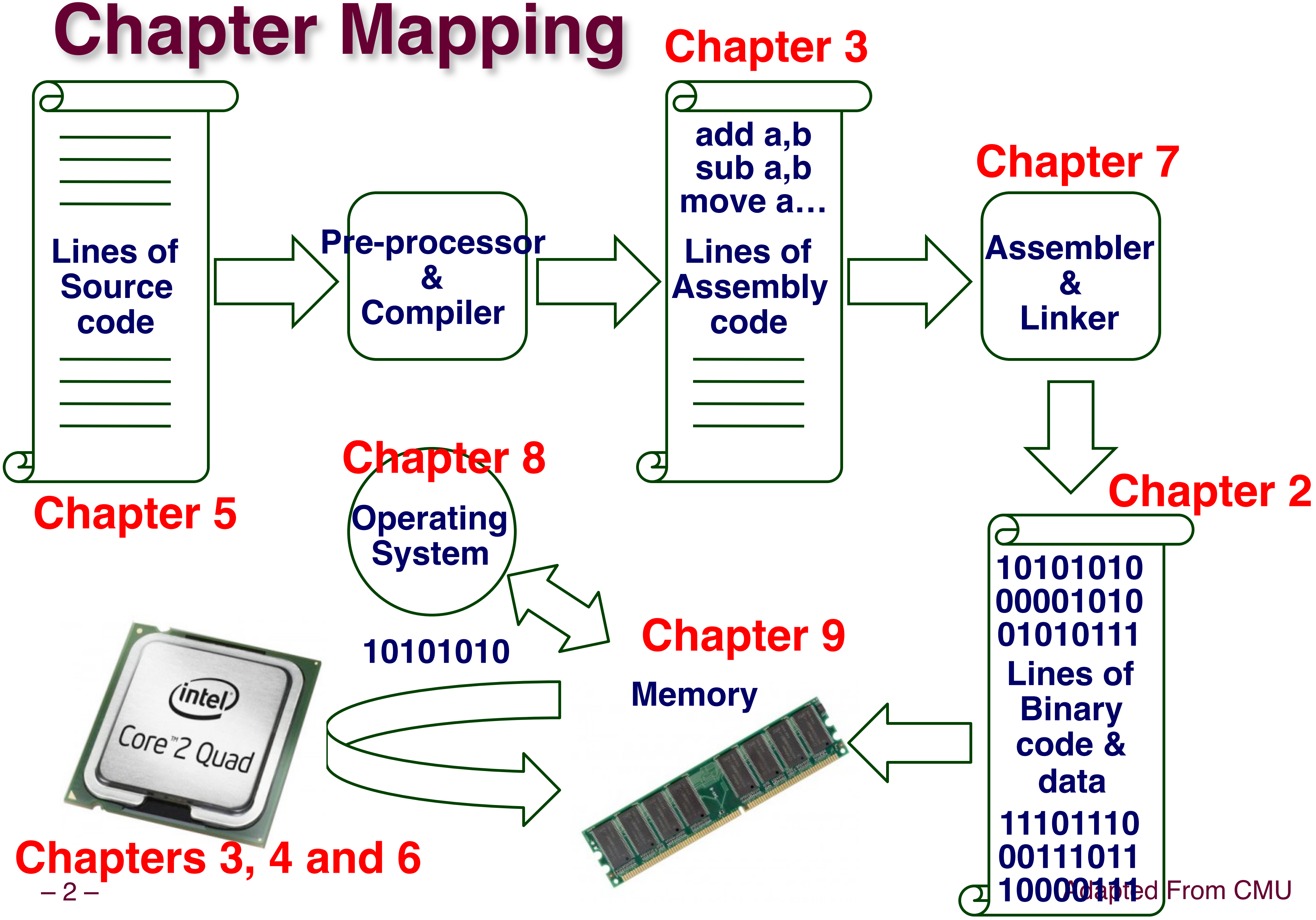
Chapter 7: Linking

Topics

- Static linking
- Object files

Chapter Mapping



Chapter 3

Lines of Source code

Chapter 5

Pre-processor & Compiler

add a,b
sub a,b
move a…

Lines of Assembly code

Chapter 7

Assembler & Linker

Chapter 2

10101010
00001010
01010111

Lines of Binary code & data

11101110
00111011
10000111

Chapter 8

Operating System

Chapter 9

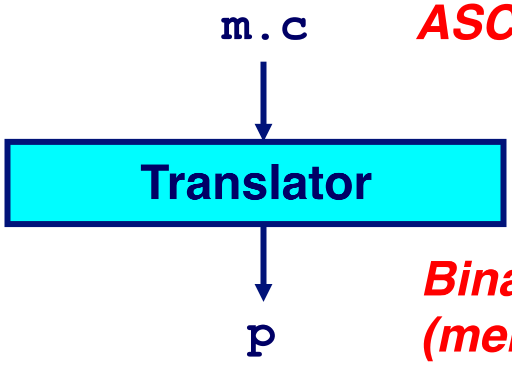
Memory

10101010

Chapters 3, 4 and 6

Adapted From CMU

A Simplistic Program Translation Scheme



ASCII source file

Binary executable object file (memory image on disk)

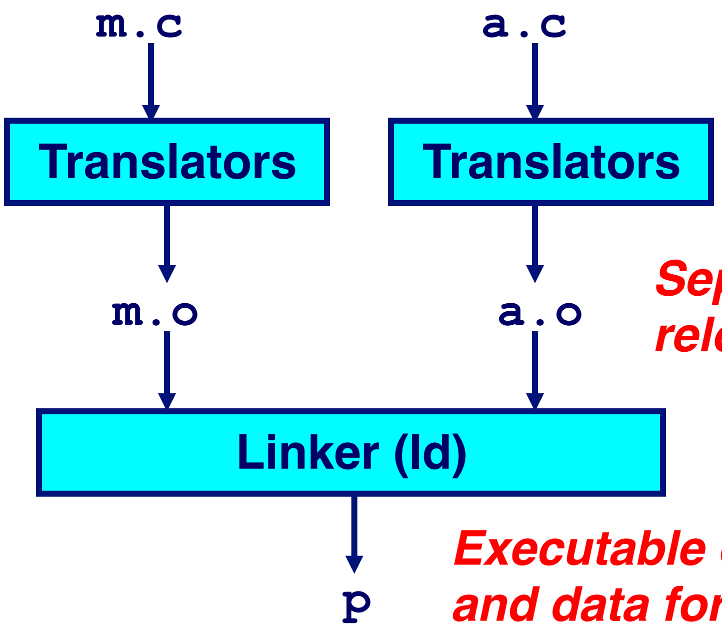
Problems:

- **Efficiency: small change requires complete recompilation**
- **Modularity: hard to share common functions (e.g. `printf`)**

Solution:

- ***Static linker (or linker)***

A Better Scheme Using a Linker



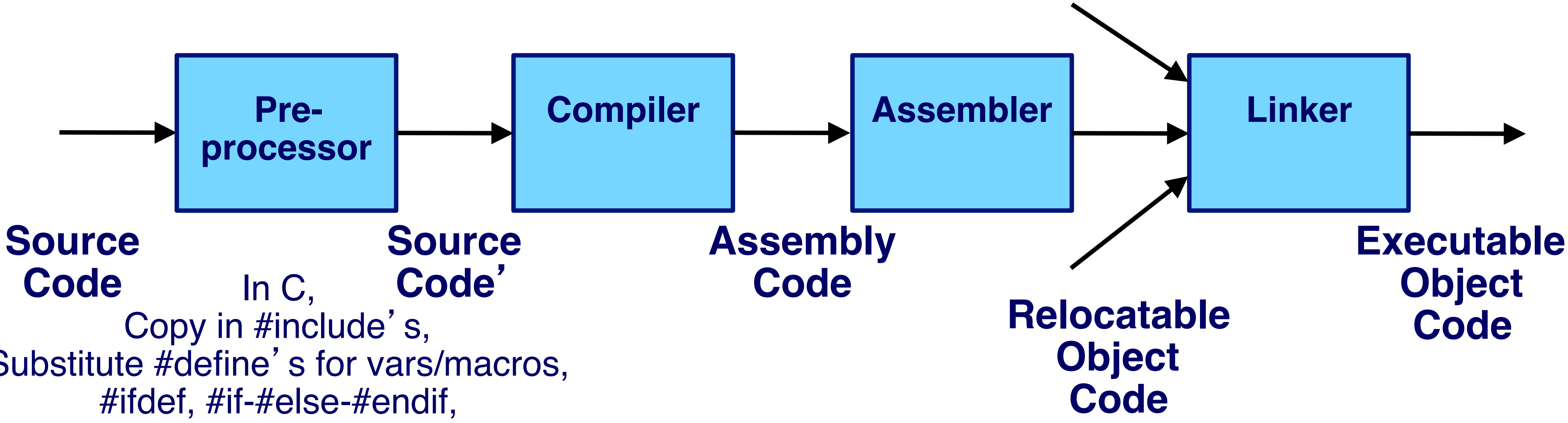
Separately compiled relocatable object files

Executable object file (contains code and data for all functions defined in `m.c` and `a.c`)

Compiling a Program

Compiler driver **coordinates all steps in the translation and linking process.**

- Typically included with each compilation system (e.g., `gcc`)
- Invokes preprocessor (`cpp`), compiler (`cc1`), assembler (`as`), and linker (`ld`).
- Passes command line arguments to appropriate phases



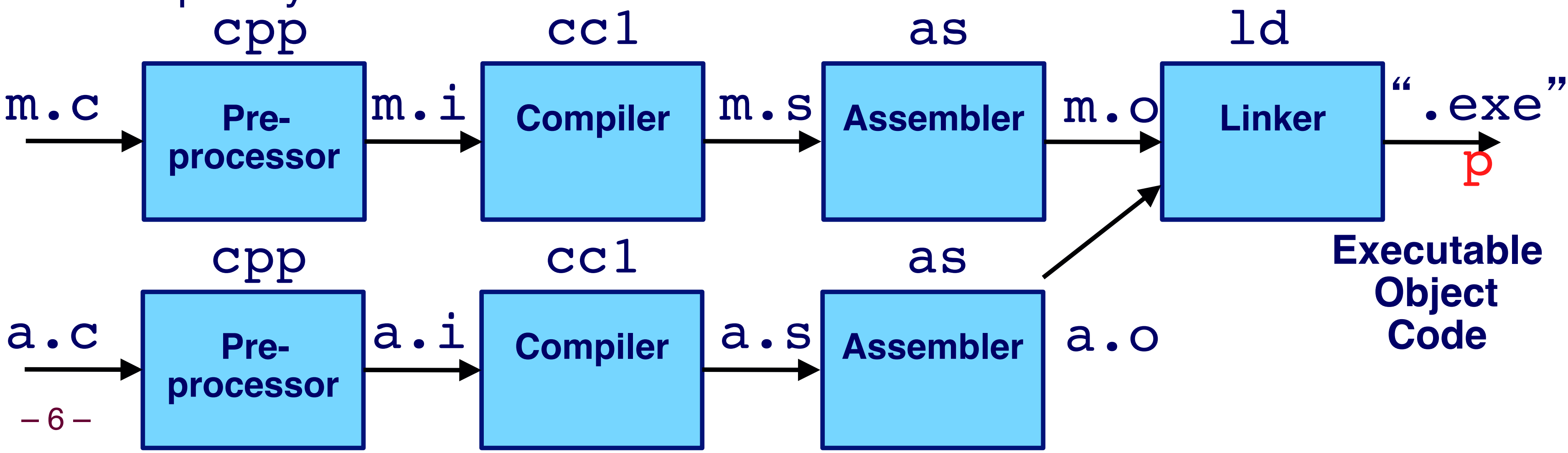
In C,
Copy in #include's,
Substitute #define's for vars/macros,
#ifdef, #if-#else-#endif,
etc.

Translating the Example Program

Example: create executable p from `m.c` and `a.c`:

```
bass> gcc -O2 -v -o p m.c a.c
cpp [args] m.c /tmp/cca07630.i
cc1 /tmp/cca07630.i m.c -O2 [args] -o /tmp/cca07630.s
as [args] -o /tmp/cca076301.o /tmp/cca07630.s
<similar process for a.c>
ld -o p [system obj files] /tmp/cca076301.o /tmp/cca076302.o
bass>
```

Conceptually:



cpp cc1 as ld

m.c → Pre-processor → m.i → Compiler → m.s → Assembler → m.o → Linker → ".exe" p

cpp cc1 as

a.c → Pre-processor → a.i → Compiler → a.s → Assembler → a.o → Linker

Executable Object Code

What Does a Linker Do?

Merges object files

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

1. Resolves external references, i.e. symbol resolution

- As part of the merging process, resolves external references.
  - *External reference*: reference to a symbol defined in another object file.

2. Relocates symbols, i.e. code relocation

- 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.
  - References can be in either code or data
    - » code: `a();           /* reference to symbol a */`
    - » data: `int *xp=&x;    /* reference to symbol x */`

Why Linkers?

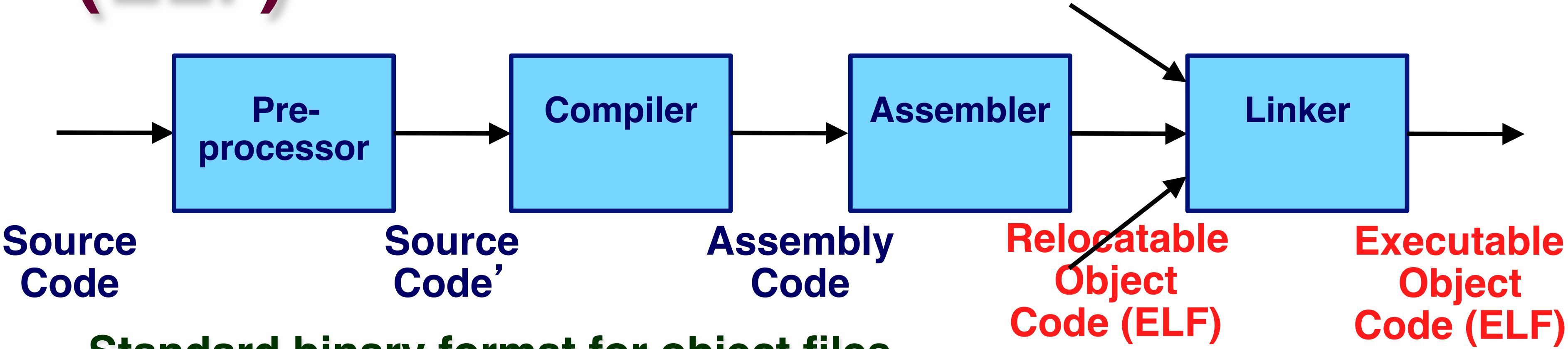
Modularity

- Program can be written as a collection of smaller source files, rather than one monolithic mass.
- Can build libraries of common functions (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.
- Space:
  - Libraries of common functions can be aggregated into a single file...
  - Yet executable files and running memory images contain only code for the functions they actually use.

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

One unified format for

- Relocatable object files (`.o`),
- Executable object files
- Shared object files (`.so`)

Generic name: ELF binaries

`readelf` is a Unix binary utility that displays info about ELF files

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

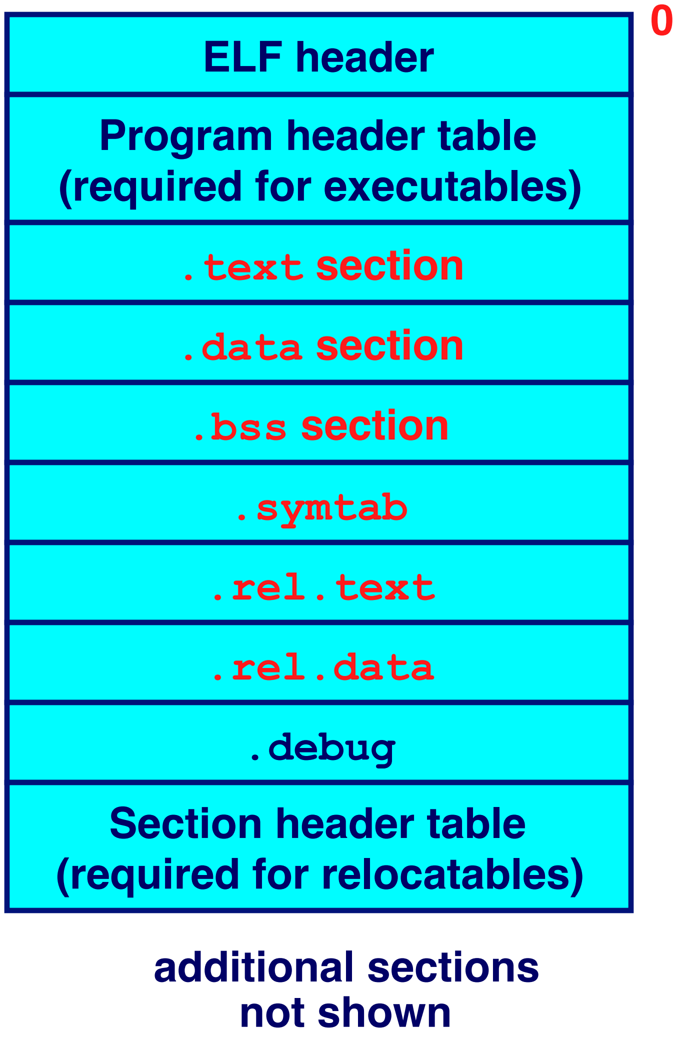
- Code!

`.data` section

- Initialized (static) data – global variables

`.bss` section

- Uninitialized (static) data – global variables
- “Blank Storage Segment”
- “Better Save Space”
- Has section header but occupies no space



0

ELF header

Program header table
(required for executables)

`.text` section

`.data` section

`.bss` section

`.symtab`

`.rel.text`

`.rel.data`

`.debug`

Section header table
(required for relocatables)

additional sections
not shown

ELF Object File Format (cont)

`.symtab` section

- Symbol table
- Procedure and static variable names
- Section names and locations

`.rel.text` section

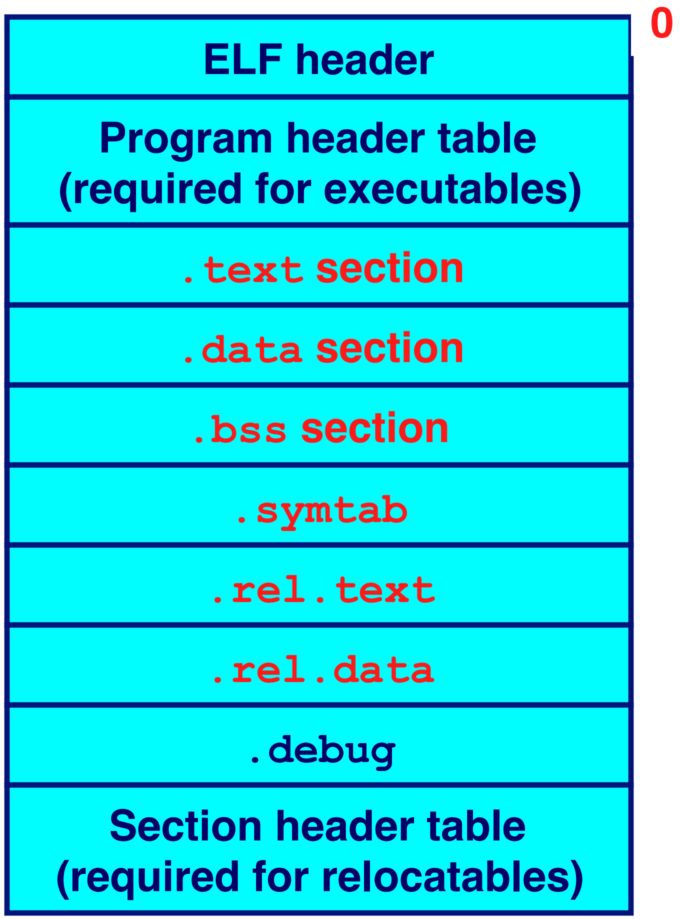
- Relocation info for `.text` section
- Addresses of instructions that will need to be modified in the executable
- Instructions for modifying.

`.rel.data` section

- Relocation info for `.data` section
- Addresses of pointer data that will need to be modified in the merged executable

`.debug` section

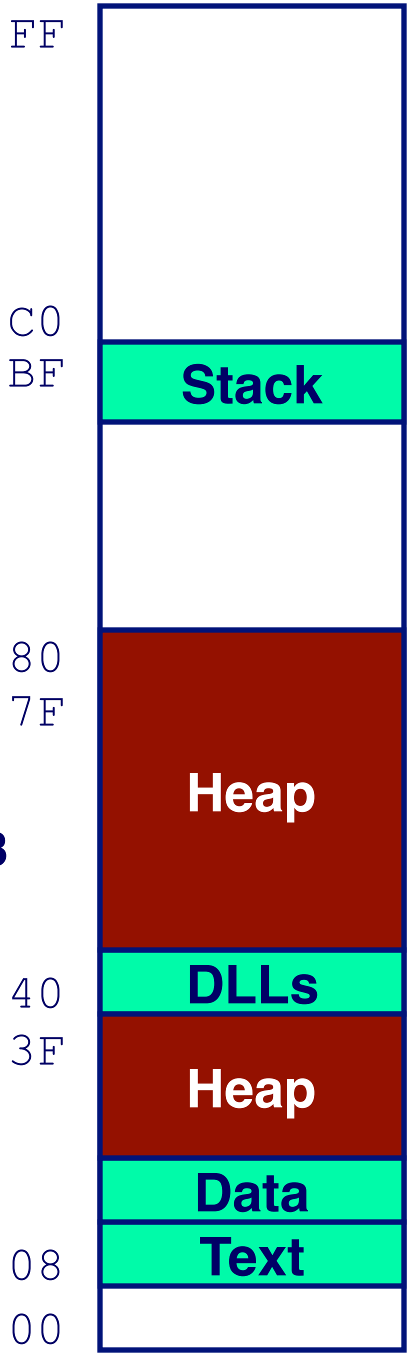
- Info for symbolic debugging (`gcc -g`)



additional sections
not shown

Supplementary Slides

Linux Memory Layout



Upper 2 hex digits of address

Red Hat v. 6.2 ~1920MB memory limit

Stack

- Runtime stack (8MB limit)

Heap

- Dynamically allocated storage
- When call `malloc`, `calloc`, `new`

DLLs

- Dynamically Linked Libraries
- Library routines (e.g., `printf`, `malloc`)
- Linked into object code when first executed

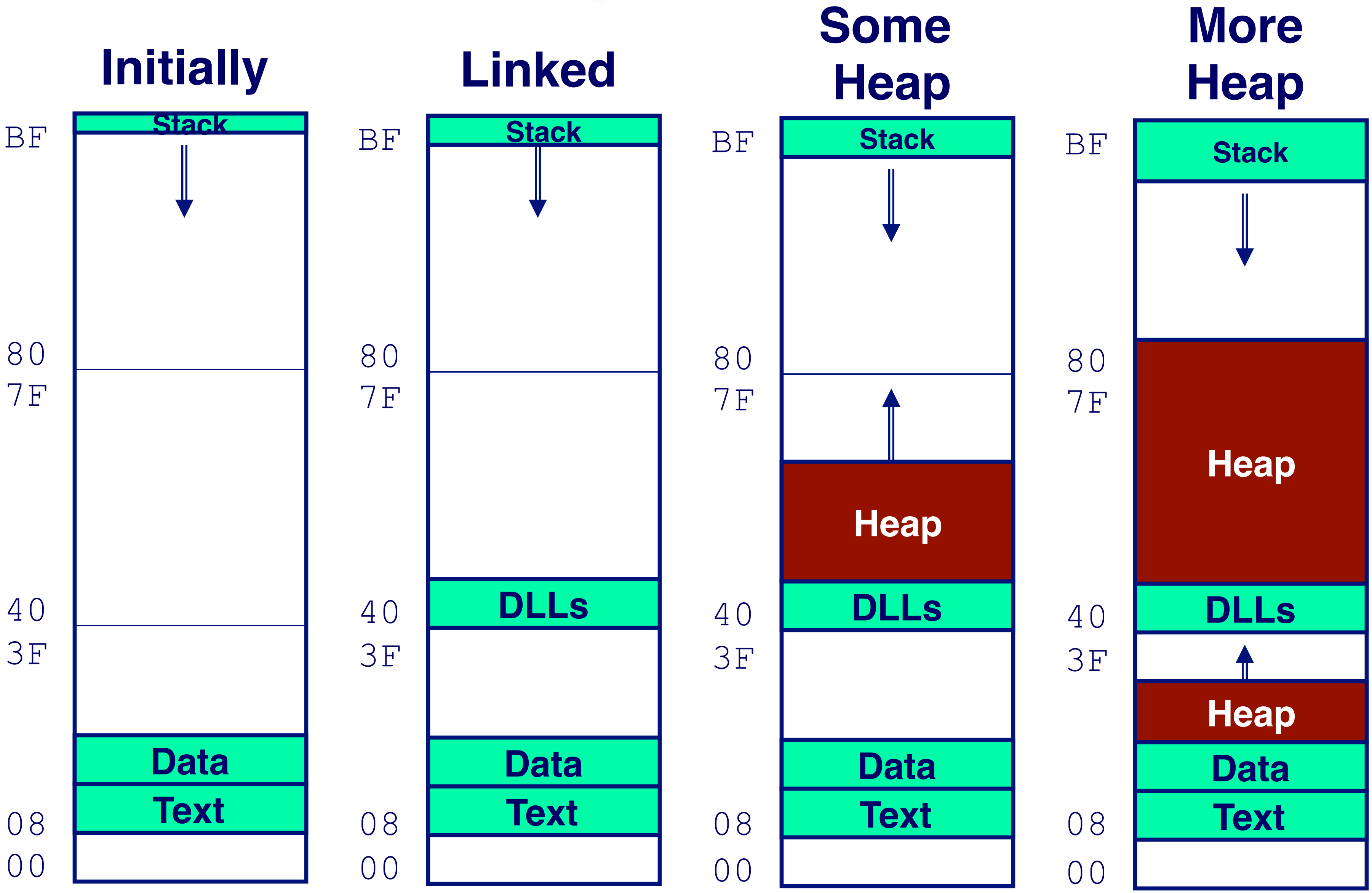
Data

- Statically allocated data
- E.g., arrays & strings declared in code

Text

- Executable machine instructions
- Read-only

Linux Memory Allocation


Initially
Linked
Some Heap
More Heap
BF
80
7F
40
3F
08
00
Stack
DLLs
Heap
Data
Text