Linking and Loading: Linking

These slides adapted from materials provided by the textbook authors.

Linking and Loading

- **Linking**
- Loading
- Case study: Library interpositioning

Example C Program

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

int sum(int *a, int n);

int main(){
    int val = sum(array, 2);
    return val;
}
```

main.c

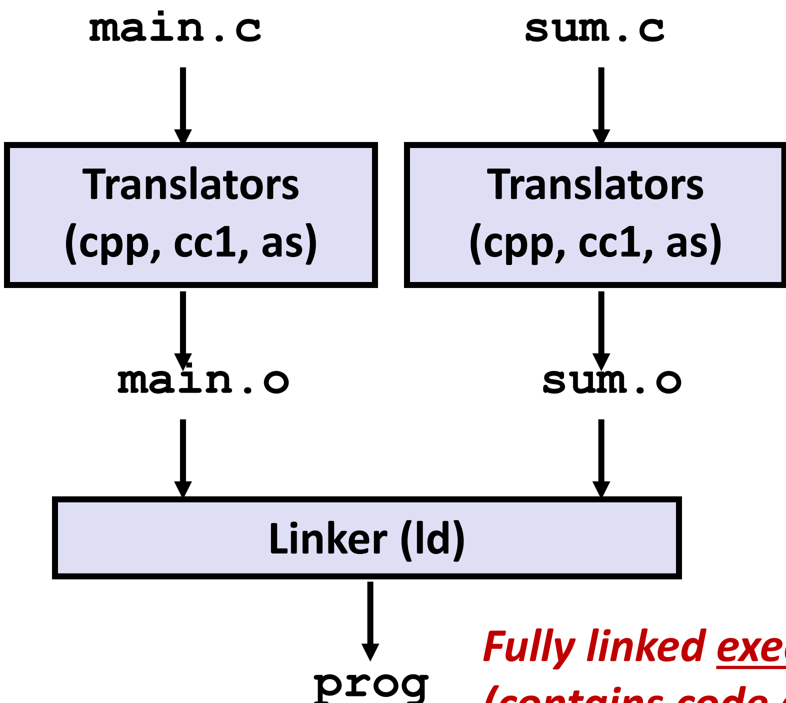
```c
int sum(int *a, int n)
{
    int i, s = 0;

    for (i = 0; i < n; i++) {
        s += a[i];
    }
    return s;
}
```

sum.c

Static Linking

- **Programs are translated and linked using a *compiler driver*:**
  - `linux> gcc -Og -o prog main.c sum.c`
  - `linux> ./prog`



Source files

Separately compiled relocatable object files

Fully linked executable object file (contains code and data for all functions defined in `main.c` and `sum.c`)

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

  - Space: Libraries
    - 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.

What Do Linkers Do?

- **Step 1: Symbol resolution**

  - Programs define and reference *symbols* (global variables and functions):
    - `void swap() {…}     /* define symbol swap */`
    - `swap();             /* reference symbol swap */`
    - `int *xp = &x;       /* define symbol xp, reference x */`

  - Symbol definitions are stored in object file (by assembler) in *symbol table*.
    - Symbol table is an array of `struct`s
    - Each entry includes name, size, and location of symbol.

  - **During symbol resolution step, the 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

  - 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.

Let’s look at these two steps in more detail….

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)

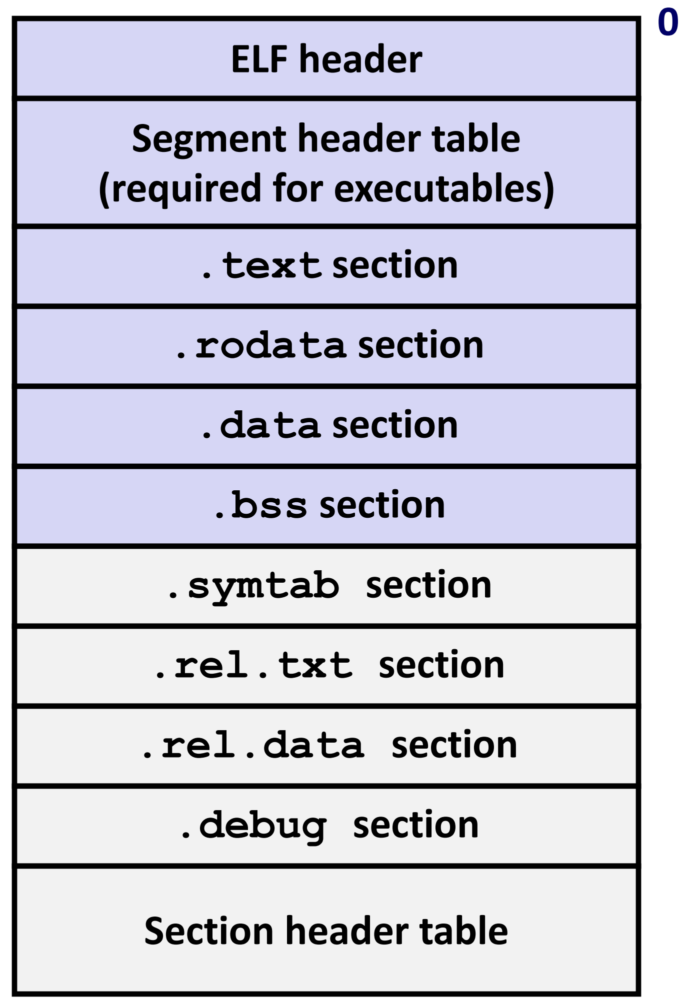
- **Standard binary format for object files**

- **One unified format for**
  - Relocatable object files (`.o`),
  - Executable object files `(a.out`)
  - Shared object files (`.so`)

- **Generic name: ELF binaries**

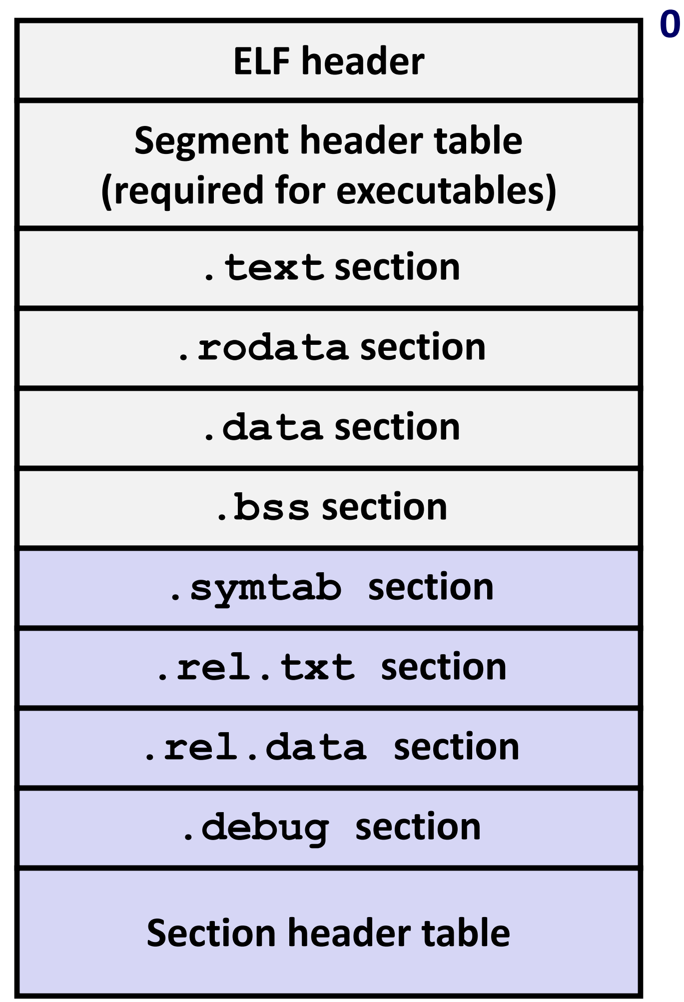
ELF Object File Format

- **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**
  - Uninitialized global variables
  - “Block Started by Symbol”
  - 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



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 global variables defined with the `static` attribute.
  - **Local linker symbols are *not* local program variables – those are allocated on the stack at runtime & not managed by linker**

Step 1: Symbol Resolution

Referencing a global…

…that's defined here

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

int sum(int *a, int n);

int main(){
    int val = sum(array, 2);
    return val;
}
```

main.c

```
int sum(int *a, int n)
{
    int i, s = 0;

    for (i = 0; i < n; i++) {
        s += a[i];
    }
    return s;
}
```

sum.c

Defining a global

Linker knows nothing of `val`

Referencing a global…

…that's defined here

Linker knows nothing of `i` or `s`

Local Symbols

- **Local non-static C variables vs. local static C variables**
  - local non-static C variables: stored on the stack
  - local static C variables: stored in either `.bss,` or `.data`

```
int f()
{
    static int x = 0;
    return x;
}

int g()
{
    static int x = 1;
    return x;
}
```

Compiler allocates space in `.data` for each definition of `x`

Creates local symbols in the symbol table with unique names, e.g., `x.1` and `x.2`.

How Linker Resolves Duplicate Symbol Definitions

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

	p1.c	**p2.c**	
strong	`int foo=5;`	`int foo;`	weak
strong	`p1() { }`	`p2() { }`	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 symbols, 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.

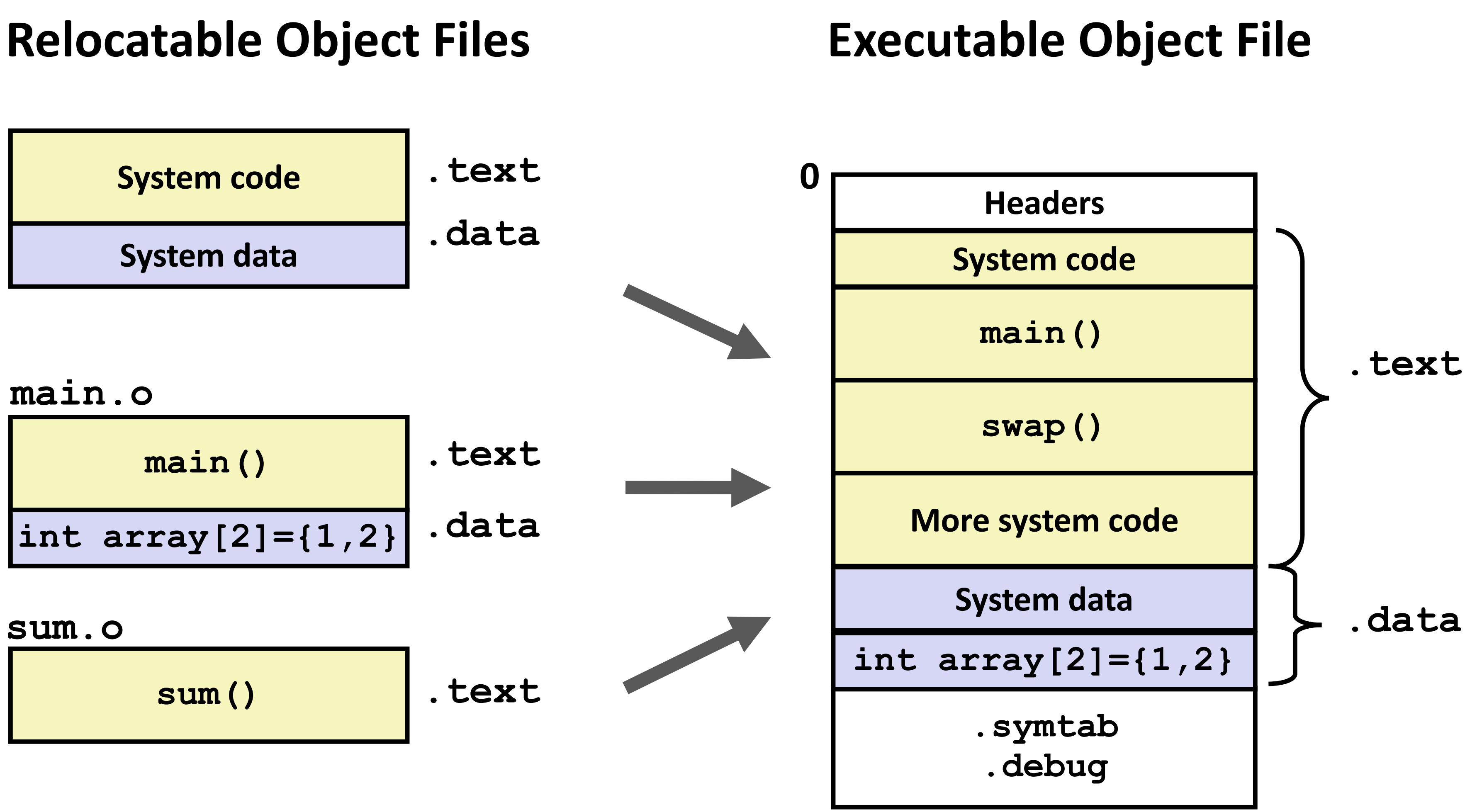
Nightmare scenario: two identical weak structs, compiled by different compilers with different alignment rules.

Global Variables

- **Avoid if you can**

- **Otherwise**
  - Use **`static`** if you can
  - Initialize if you define a global variable
  - Use **`extern`** if you reference an external global variable

Step 2: Relocation



Relocation Entries

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

int main()
{
    int val = sum(array, 2);
    return val;
}
```
main.c

```
0000000000000000 <main>:
   0:   48 83 ec 08             sub    $0x8,%rsp
   4:   be 02 00 00 00          mov    $0x2,%esi
   9:   bf 00 00 00 00          mov    $0x0,%edi      # %edi = &array
                        a: R_X86_64_32 array          # Relocation entry

   e:   e8 00 00 00 00          callq  13 <main+0x13> # sum()
                        f: R_X86_64_PC32 sum-0x4      # Relocation entry
  13:   48 83 c4 08             add    $0x8,%rsp
  17:   c3                      retq
```
main.o

Relocated .text section

```
00000000004004d0 <main>:
  4004d0:       48 83 ec 08         sub    $0x8,%rsp
  4004d4:       be 02 00 00 00      mov    $0x2,%esi
  4004d9:       bf 18 10 60 00      mov    $0x601018,%edi  # %edi = &array
  4004de:       e8 05 00 00 00      callq  4004e8 <sum>    # sum()
  4004e3:       48 83 c4 08         add    $0x8,%rsp
  4004e7:       c3                  retq

00000000004004e8 <sum>:
  4004e8:       b8 00 00 00 00            mov    $0x0,%eax
  4004ed:       ba 00 00 00 00            mov    $0x0,%edx
  4004f2:       eb 09                     jmp    4004fd <sum+0x15>
  4004f4:       48 63 ca                  movslq %edx,%rcx
  4004f7:       03 04 8f                  add    (%rdi,%rcx,4),%eax
  4004fa:       83 c2 01                  add    $0x1,%edx
  4004fd:       39 f2                     cmp    %esi,%edx
  4004ff:       7c f3                     jl     4004f4 <sum+0xc>
  400501:       f3 c3                     repz retq
```

Using PC-relative addressing for sum(): 0x4004e8 = 0x4004e3 + 0x5