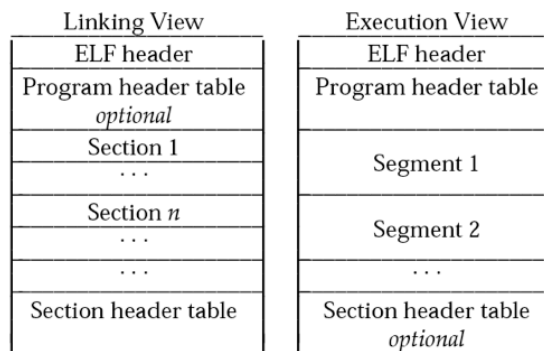


ELF

- Executable and Linkable Format
- Standard file format for
 - executables
 - object code
 - shared libraries
 - core dumps

1

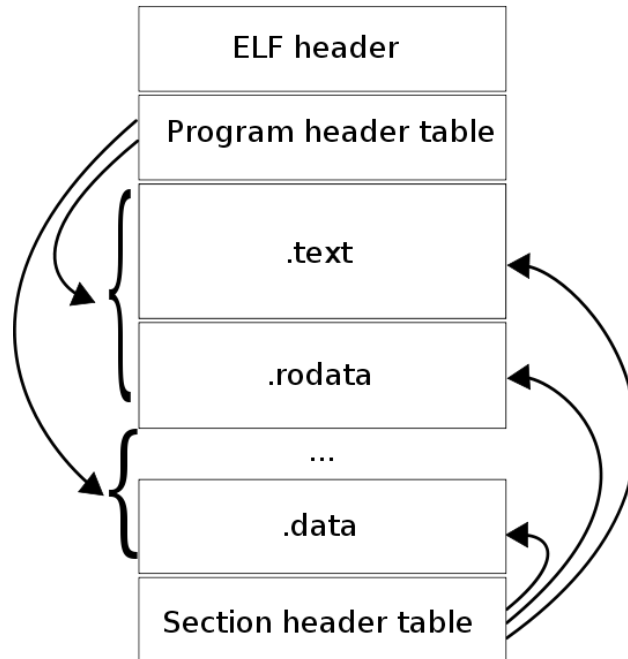
Figure 1-1: Object File Format



An *ELF header* resides at the beginning and holds a “road map” describing the file’s organization. *Sections* hold the bulk of object file information for the linking view: instructions, data, symbol table, relocation information, and so on. Descriptions of special sections appear later in Part 1. Part 2 discusses *segments* and the program execution view of the file.

source: Tool Interface Standards (TIS) Portable Formats Specification, Version 1.1

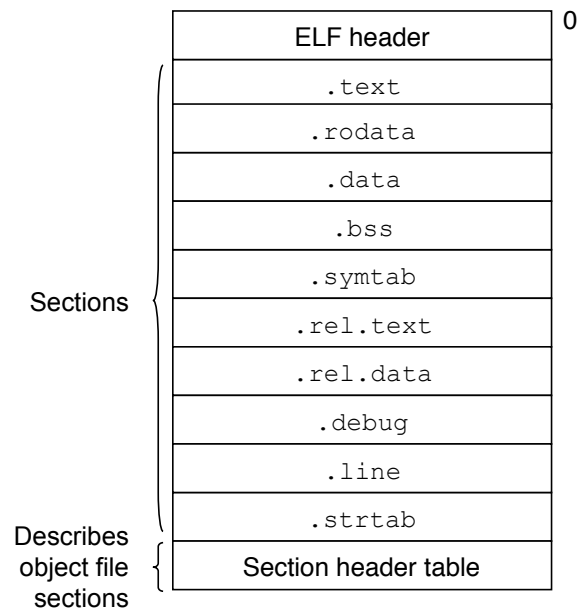
Another view
source: Wikipedia



3

Relocatable Object Files

- Format of typical ELF relocatable object file



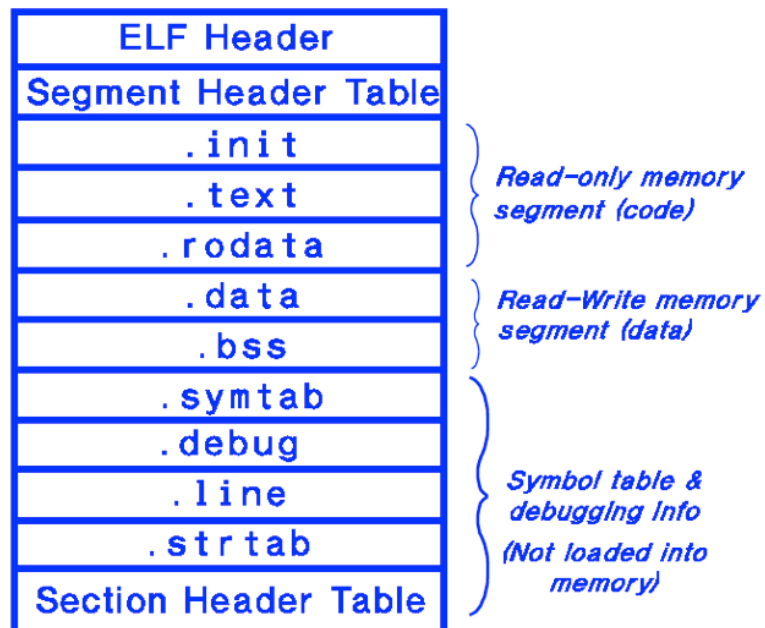
4

ELF Section examples

- .text: machine code of compiled program
- .data: initialized global C variables
- .bss: uninitialized data
- .symtab: a symbol table with info about functions and global variables
- .line: mapping between line numbers in the original C source program and machine code instructions in .text section. Only exists if -g compile option was used
- .debug: debugging symbol table

5

ELF – Executable File



6

ELF

- What is the motivation for having sections?
- What is a process image?
- Why don't we just load everything jumbled together into the process image, e.g., as in DOS?

7

ELF

- Motivation is
 - how modern machine architectures allocate memory, e.g., 4kB pages (frames)
 - memory manager can set attributes on pages, e.g., read-only. What happens when you write to read-only memory?
 - allocating memory for initialized vs uninitialized variables

8

ELF

- What happens when kernel loads & runs an executable?
 - its starts looking at image header to see how it should load the image
 - locates .text section with executable, load it in read-only pages of memory
 - then it loads .data section of the executable into user space (read-write memory)
 - locates .bss section from image header and adds appropriate pages of memory, zeroing out the pages.

9

readelf with file header option -h

```
[krings@eternium /bin]$ readelf -h ./ls
ELF Header
  Magic:                               7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00
  Class:                               ELF64
  Data:                               2's complement, little endian
  Version:                             1 (current)
  OS/ABI:                              UNIX - System V
  ABI Version:                         0
  Type:                                EXEC (Executable file)
  Machine:                             Advanced Micro Devices X86-64
  Version:                              0x1
  Entry point address:                 0x402460
  Start of program headers:            64 (bytes into file)
  Start of section headers:            89256 (bytes into file)
  Flags:                               0x0
  Size of this header:                 64 (bytes)
  Size of program headers:             56 (bytes)
  Number of program headers:           8
  Size of section headers:             64 (bytes)
  Number of section headers:           31
  Section header string table index:   30
```

10

What does this tell us?

- executable was created for AMD X86-64 architecture
- when executed it will start running from virtual address 0x402460. That is not main() though, but a `_start` procedure, created by the linker
- program has 31 sections, 8 segments
- ...

11

```
readelf -S ./ls
```

There are 31 section headers, starting at offset 0x15ca8:

Section Headers:

[Nr]	Name	Type	Address	Offset
	Size	EntSize	Flags Link Info Align	
[0]	0000000000000000	NULL	0000000000000000	00000000
	0000000000000000	0000000000000000	0 0	0
...				
[13]	.text	PROGBITS	000000000402460	0002460
	000000000000c228	0000000000000000	AX 0 0	16
[14]	.fini	PROGBITS	00000000040e688	0000e688
	000000000000000e	0000000000000000	AX 0 0	4
[15]	.rodata	PROGBITS	00000000040e6a0	0000e6a0
	000000000000382f	0000000000000000	A 0 0	32
...				
[26]	.bss	PROGBITS	000000000614820	00014820
	0000000000000538	0000000000000000	WA 0 0	32
...				

Key to Flags:

W (write), A (alloc), X (execute), M (merge), S (strings)

I (info), L (link order), G (group), x (unknown)

O (extra OS processing required) o (OS specific), p (processor specific)

12

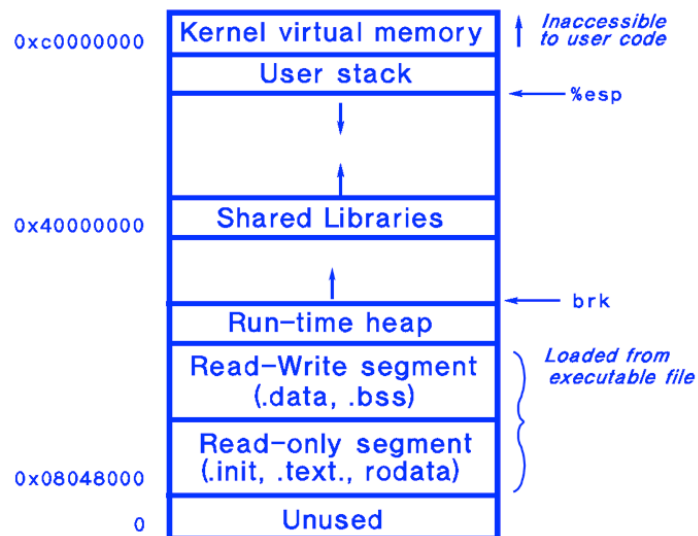
```
[krings@eternium /bin]$ objdump -d -j .text ls |more

ls:      file format elf64-x86-64

Disassembly of section .text:

0000000000402460 <close_stdout-0x6440>:
402460:  31 ed                xor    %ebp,%ebp
402462:  49 89 d1             mov    %rdx,%r9
402465:  5e                  pop    %rsi
402466:  48 89 e2             mov    %rsp,%rdx
402469:  48 83 e4 f0         and    $0xfffffffffffffff0,%rsp
40246d:  50                  push  %rax
40246e:  54                  push  %rsp
40246f:  49 c7 c0 90 e5 40 00 mov    $0x40e590,%r8
402476:  48 c7 c1 a0 e5 40 00 mov    $0x40e5a0,%rcx
40247d:  48 c7 c7 60 69 40 00 mov    $0x406960,%rdi
402484:  e8 1f fb ff ff     callq 401fa8 <_libc_start_main@plt>
402489:  f4                  hlt
40248a:  90                  nop
40248b:  90                  nop
40248c:  48 83 ec 08         sub    $0x8,%rsp
402490:  48 8b 05 09 1e 21 00 mov    2170377(%rip),%rax    # 6142a0 <quoting_style_args+0x200>
402497:  48 85 c0             test  %rax,%rax
40249a:  74 02              je     40249e <acl_from_text@plt+0x56>
40249c:  ff d0              callq *%rax
40249e:  48 83 c4 08         add    $0x8,%rsp
4024a2:  c3                 retq
4024a3:  90                  nop
4024a4:  90                  nop
```

LINUX Runtime Memory Image



LINUX Startup Pseudo-Code

```
/* crt1.o */

_start:                                /* entry point in .text */
    call __libc_init_first /* startup code in .text */
    call _init                /* startup code in .init */
    call atexit                /* startup code in .text */
    /* set up argument list for main here */
    call main                    /* application main code */
    call _exit                    /* returns control to shell */
```