Elf Formats in Linux

ELF: Executable and Linking Format

- ELF is a portable object file format defining the composition and organization of the object file. Kernel and binary loader looks at this format to know how to load this file and find various pieces of information like code, initialized data, dependencies on shared libraries etc.

Elf File Types

- Elf defines the format of executable binary files. There are four different types:
  - Relocatable
    - Created by compilers or assemblers. Need to be processed by the linker before running.
    - holds code and data suitable to link with other object files
  - Executable
    - Have all relocation done and all symbol resolved except perhaps shared library symbols that must be resolved at run time.
    - suitable for execution
  - Shared object
    - Shared library containing both symbol information for the linker and directly runnable code for run time.
    - holds code and data suitable to link with other relocatable object or shared objectsObject files are created by the assembler and link editor, object files are binary representations of programs
  - Core file
    - A core dump file.

ELF Structure

- Elf files have a dual nature:
  - Compilers, assemblers, and linkers treat the file as a set of logical sections described by a section header table.
  - The system loader treats the file as a set of segments described by a program header table.

ELF Structure

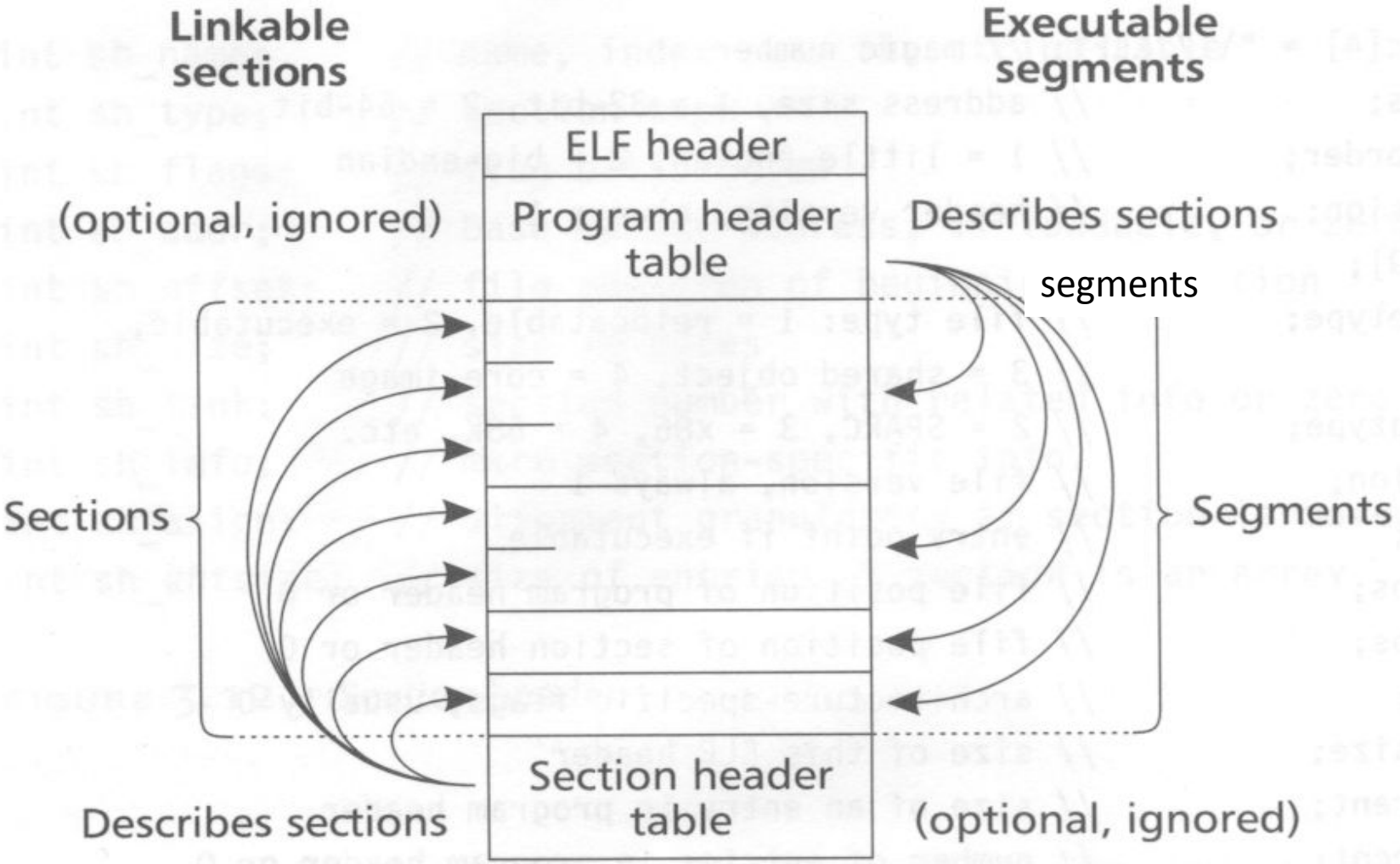


FIGURE 3.10 • Two views of an ELF file.

ELF File Format

- An ELF file has:
  - A header
  - A program header
  - A section header
  - Sections
  - More stuff which matters less (string table, debug symbols, …)

Sections are used by the linker (compile time)

Segments are used by the loader (runtime)

ELF Structure

- A single segment usually consist of several sections. E.g., a loadable read-only segment could contain sections for executable code, read-only data, and symbols for the dynamic linker.
- Relocatable files have section header tables. Executable files have program header tables. Shared object files have both.
- Sections are intended for further processing by a linker, while the segments are intended to be mapped into memory.

An Example C Program

```
#include <stdio.h>
    int main() {
       printf("hello world!n");
    }
```

Save the code in a file named test.c, and then compile the source code into a *.o file and an executable with the commands below.

```
$ gcc test.c -c
$ gcc test.o -o test
```

We can check if a file is ELF file by the file command as shown below.

```
$ file test.o
```

Program segments/sections

$ size /usr/bin/test.o

text	data	bss	dec	hex
245138	3696	2832	251666	3d712

filename

/usr/bin/cc

$ *$ readelf -h test*

The -h option means to display the ELF file header. The Magic number is used to indicate the file is ELF file.

ELF Header

- The Elf header is always at offset zero of the file.
- The program header table and the section header table's offset in the file are defined in the ELF header.
- The header is decodable even on machines with a different byte order from the file's target architecture.

$ *readelf -h test.o*

The -h option means to display the ELF file header.

The Magic number is used to indicate the file is ELF file.

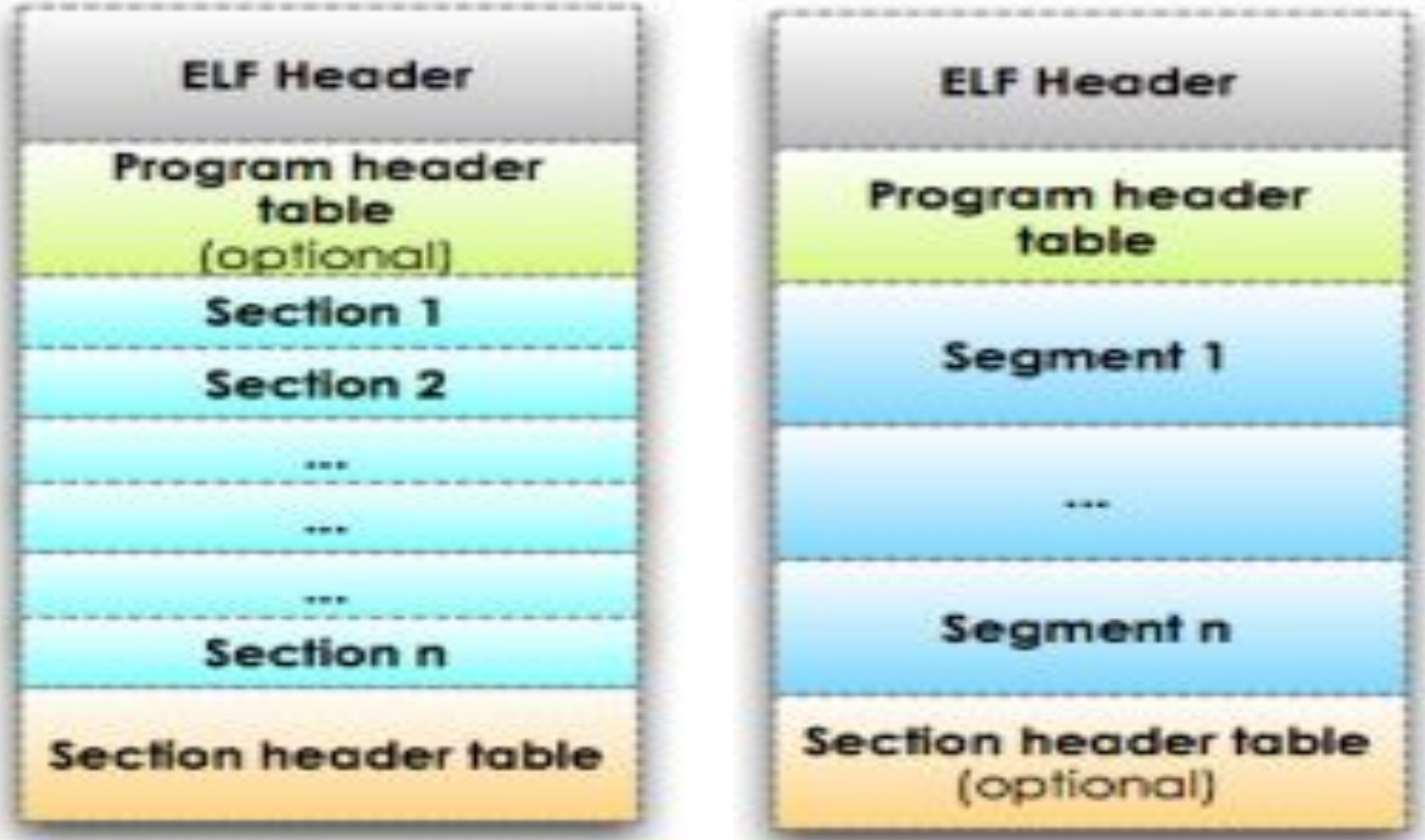
```
char magic[4] = "\177ELF"; // magic number
char class;              // address size, 1 = 32-bit, 2 = 64-bit
char byteorder;          // 1 = little-endian, 2 = big-endian
char hversion;           // header version, always 1
char pad[9];
short filetype;          // file type: 1 = relocatable, 2 = executable,
                         // 3 = shared object, 4 = core image
short archtype;          // 2 = SPARC, 3 = x86, 4 = 68K, etc.
int fversion;            // file version, always 1
int entry;               // entry point if executable
int phdrpos;             // file position of program header or 0
int shdrpos;             // file position of section header or 0
int flags;               // architecture-specific flags, usually 0
short hdrsize;           // size of this ELF header
short phdrent;           // size of an entry in program header
short phdrcnt;           // number of entries in program header or 0
short shdrent;           // size of an entry in section header
short shdrcnt;           // number of entries in section header or 0
short strsec;            // section number that contains section name strings
```

FIGURE 3.11 • ELF header.

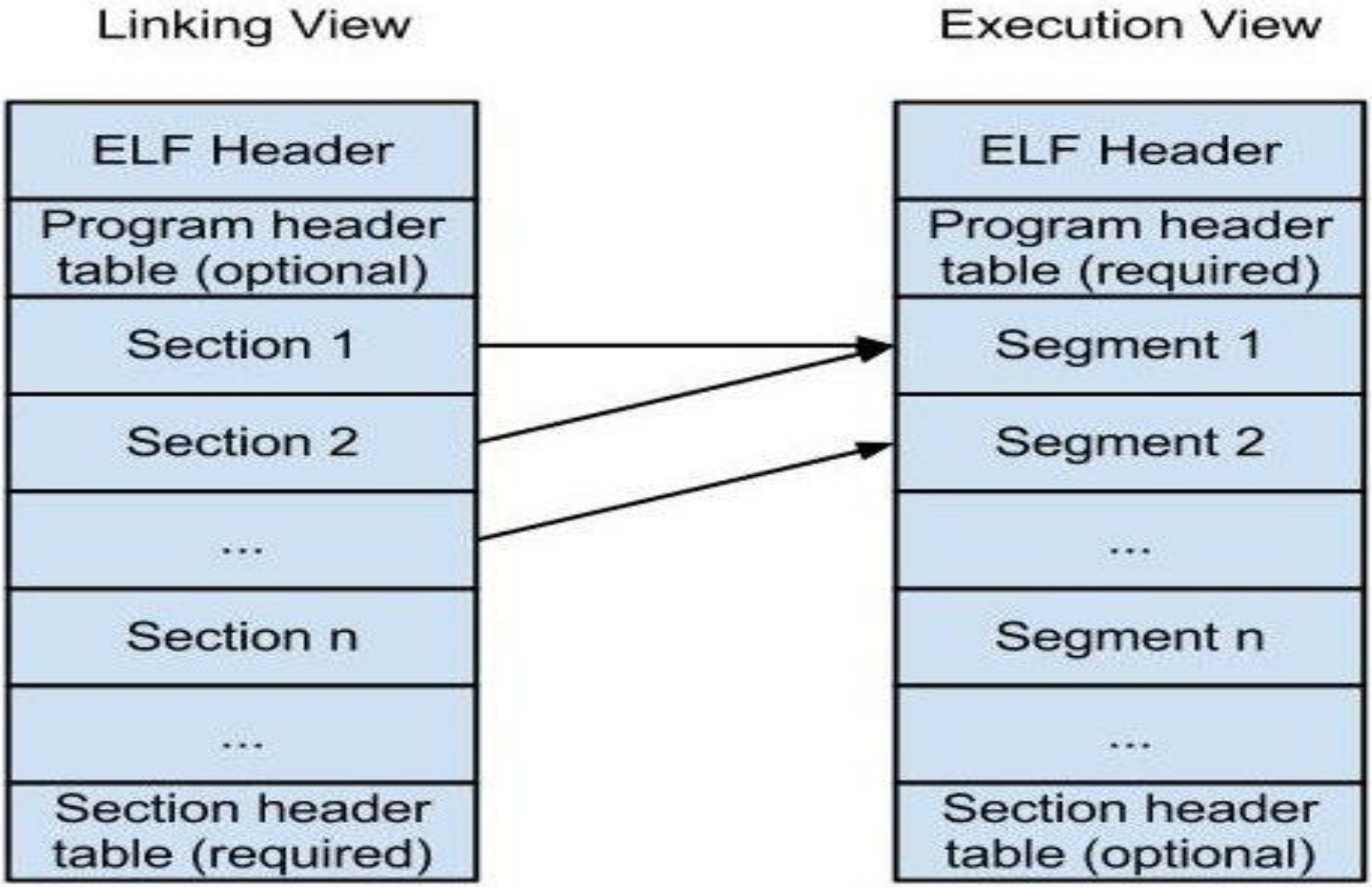
Display ELF header: readelf -h

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 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:               0x400440
  Start of program headers:          64 (bytes into file)
  Start of section headers:          4424 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         9
  Size of section headers:           64 (bytes)
  Number of section headers:         31
```

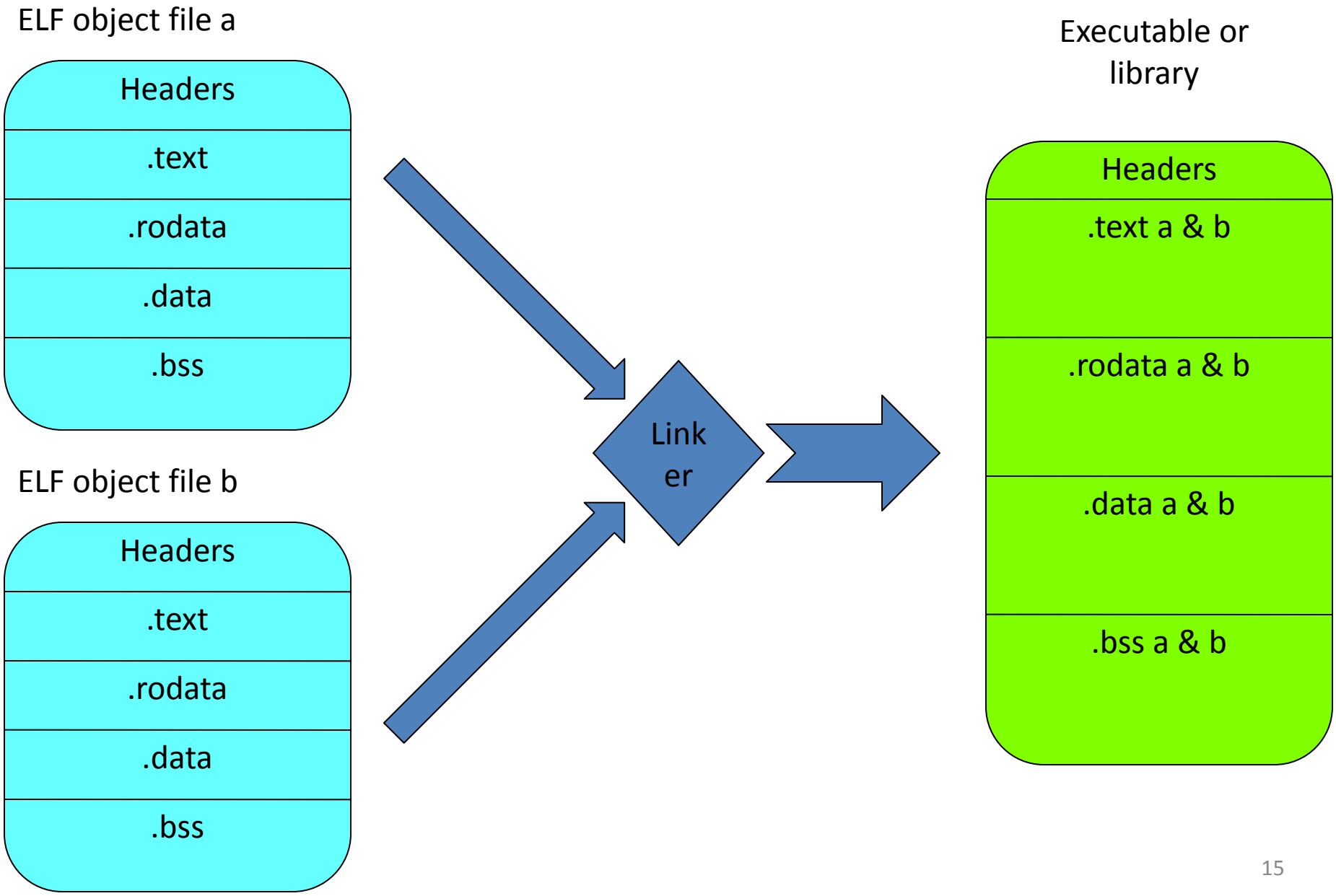
ELF File Format



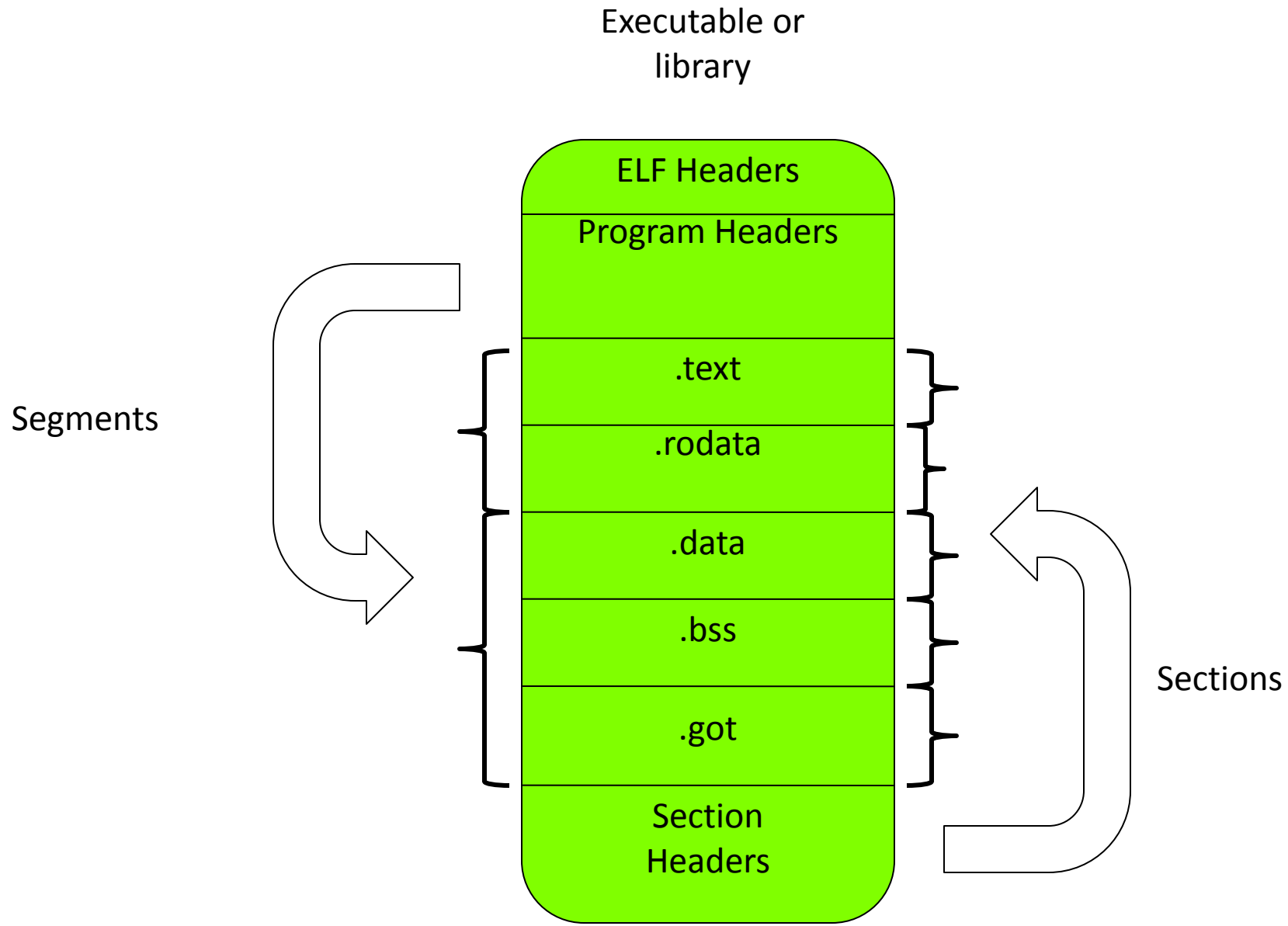
ELF Files: Dual Views



Linker Overview



Headers Overview



Relocatable Files

- A relocatable or shared object file is a collection of sections.
- Each section contains a single type of information, such as program code, read-only data,or read/write data, relocation entries,or symbols.
- Every symbol's address is defined relative to a section.
  - Therefore, a procedure's entry point is relative to the program code section that contains that procedure's code.

Section Header

```
int sh_name;      // name, index into the string table
int sh_type;      // section type
int sh_flags;     // flag bits, below
int sh_addr;      // base memory address, if loadable, or zero
int sh_offset;    // file position of beginning of section
int sh_size;      // size in bytes
int sh_link;      // section number with related info or zero
int sh_info;      // more section-specific info
int sh_align;     // alignment granularity if section is moved
int sh_entsize;   // size of entries if section is an array
```

FIGURE 3.12 • Section header.

Types in Section Header

- PROGBITS: This holds program contents including code, data, and debugger information.
- NOBITS: Like PROGBITS. However, it occupies no space.
- SYMTAB : These hold symbol table.
- STRTAB: This is a string table, like the one used in a.out.
- REL and RELA: These hold relocation information.
- DYNAMIC and HASH: This holds information related to dynamic linking.

Flags in Section Header

- WRITE: This section contains data that is writable during process execution.
- ALLOC: This section occupies memory during process execution.
- EXECINSTR: This section contains executable machine instructions.

Various Sections

- .text:
  - This section holds executable instructions of a program.
  - Type: PROGBITS
  - Flags: ALLOC + EXECINSTR
  - $ readelf -S vipi | grep '.text'
- .data:
  - This section holds initialized data that contributes to the program's image.
  - Type: PROGBITS
  - Flags: ALLOC + WRITE
- .rodata:
  - This section holds read-only data.
  - Type: PROGBITS
  - Flags: ALLOC

$objdump -s test.o

Various Sections

- .bss :
  - This section holds uninitialized data that contributed to the program's image. By definition, the system will initialize the data with zero when the program begins to run.
  - Type: NOBITS
  - Flags: ALLOC + WRITE
- .rel.text, .rel.data, and .rel.rodata:
  - These contain the relocation information for the corresponding text or data sections.
  - Type: REL
  - Flags: ALLOC is turned on if the file has a loadable segment that includes relocation.
- .symtab:
  - This section hold a symbol table.
- .strtab:
  - This section holds strings.

Various Sections

- .init:
  - This section holds executable instructions that contribute to the process initialization code.
  - Type: PROGBITS
  - Flags: ALLOC + EXECINSTR
- .fini:
  - This section hold executable instructions that contribute to the process termination code.
  - Type: PROGBITS
  - Flags: ALLOC + EXECINSTR
- C does not need these two sections. However, C++ needs them.

Various Sections

- .interp:
  - This section holds the pathname of a program interpreter.
  - Type: ALLOC
  - Flags: PROGBITS
  - If this section is present, rather than running the program directly, the system runs the interpreter and passes it the elf file as an argument.
  - For many years (used in a.out), UNIX has had self-running interpreted text files, using
  - #! /bin/csh as the first line of the file.
  - Elf extends this facility to interpreters that run nontext programs.
  - In practice, this is used to run the run-time dynamic linker to load the program and to link in any required shared libraries.

Various Sections

- .debug:
  - This section holds symbolic debugging information.
  - Type: PROGBIT
- .line:
  - This section holds line number information for symbolic debugging, which describes the correspondence between the program source and the machine code (ever used gdb?)
  - Type: PROGBIT
- .comment
  - This section may store extra information.
- .got:
  - This section holds the global offset table.
    - Related toshared library.
  - Type: PROGBIT
- .plt:
  - This section holds the procedure linkage table.
  - Type: PROGBIT
- .note:
  - This section contains some extra information.

Various Sections

- .debug:
  - This section holds symbolic debugging information.
  - Type: PROGBIT
- .line:
  - This section holds line number information for symbolic debugging, which describes the correspondence between the program source and the machine code (ever used gdb?)
  - Type: PROGBIT
- .comment
  - This section may store extra information.
- .got:
  - This section holds the global offset table.
    - Related toshared library.
  - Type: PROGBIT
- .plt:
  - This section holds the procedure linkage table.
  - Type: PROGBIT
- .note:
  - This section contains some extra information.

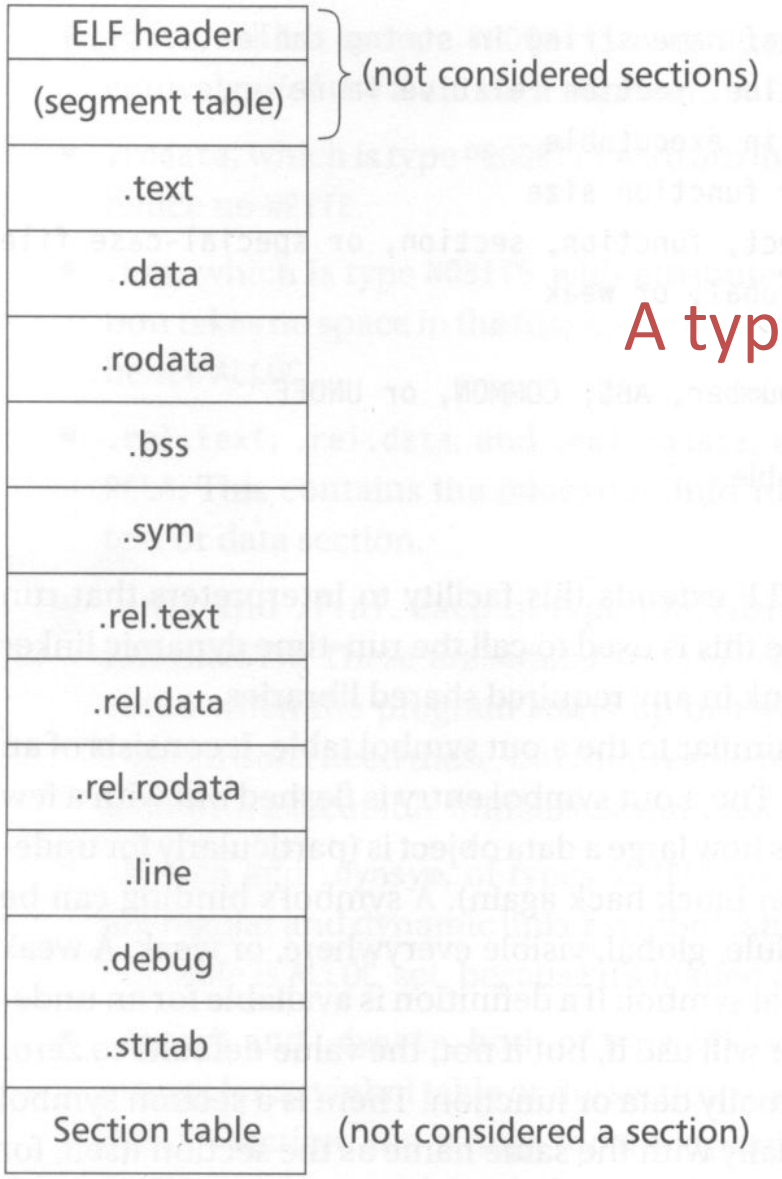


FIGURE 3.14 • Sample relocatable ELF file.

A typical relocatable file.

String Table

- Like the format used in a.out.
- String table sections hold null-terminated character sequences, commonly called strings.
- The object file uses these strings to represent symbol and section names.
- We use an index into the string table section to reference a string.
- The reason why we separate symbol names from symbol tables is that in C or C++, there is no limitation on the length of a symbol.

String Table

Index	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9
0	\0	n	a	m	e	.	\0	V	a	r
10	i	a	b	l	e	\0	a	b	l	e
20	\0	\0	x	x	\0					

$ readelf -p '.symtab' test.o

Index	String
0	*none*
1	name.
7	Variable
11	able
16	able
24	*null string*

Symbol Table

- An object file's symbol table holds information needed to locate and relocate a program's symbolic definition and references.
- A symbol table index is a subscript into this array.

$ readelf -s test.o

Symbol Table

```
int name;     // position of name string in string table
int value;    // symbol value, section relative in reloc,
              // absolute in executable
int size;     // object or function size
char type:4;  // data object, function, section, or special-case file
char bind:4;  // local, global, or weak
char other;   // spare
short sect;   // section number, ABS, COMMON, or UNDEF
```

FIGURE 3.13 • ELF symbol table.

e.g., int, double

If a definition is available for an undefined weak symbol, the linker will use it. Otherwise, the value defaults to 0.

The section relative to which the symbol is defined. (e.g., the function entry points are defined relative to .text)

Relocation Table

- Relocation is the process of connecting symbolic references with symbolic definitions.
- Relocatable files must have information that describes how to modify their section contents.
- A relocation table consists on many relocation structures.

- ***$ readelf -l test.o*** *to read program Headers*
- ***$ readelf -S test.o*** view the section headers and program headers with readelf command.

Executable Files

- An executable file usually has only a few segments. E.g.,
  - A read-only one for the code.
  - A read-only one for read-only data.
  - A read/write one for read/write data.
- All of the loadable sections are packed into the appropriate segments so that the system can map the file with just one or two operations.
  - E.g., If there is a .init and .fini sections, those sections will be put into the read-only text segment.
  - Segment are Page Aligned

Segments

- Executable and shared object files contain segments. A segment is a grouping of one or more sections in the object file. For example, the *code segment*includes the .text and .rodata sections while the *data segment* contains the .bss and .data sections.
- It Defines:
- Permission applied in segments
- Size & Section Information
- Sections with identical permission can mapped to one segments
- **$ readelf –l test.o**

Dynamic linking

- Dynamic linking allows to resolve library functions at runtime
- Relocatable File: Holds code and data suitable for linking with other object files to create an executable or shared object file
  - Executable File: Holds a program suitable for execution
  - Shared Object File: Holds code and data suitable for linking in two contexts:

The **Link Editor** may process it with other relocatable and shared object files to create another object file

The **Dynamic Linker** combines it with an executable file and other shared objects to create a process image

Program Header

```
int type;      // loadable code or data, dynamic linking info, etc.
int offset;    // file offset of segment
int virtaddr;  // virtual address to map segment
int physaddr;  // physical address, not used
int filesize;  // size of segment in file
int memsize;   // size of segment in memory (bigger if contains bss)
int flags;     // Read, Write, Execute bits
int align;     // required alignment, invariably hardware page size
```

FIGURE 3.15 • ELF program header.

The Types in Program Header

- This field tells what kind of segment this array element describes:
  - PT_LOAD: This segment is a loadable segment.
  - PT_DYNAMIC: This array element specifies dynamic linking information.
  - PT_INTERP: This element specified the location and size of a null-terminated path name to invoke as an interpreter.

Executable File Example

	File offset	*Load address*	*Type header*
ELF header	0	0x8000000	
Program header	0x40	0x8000040	
Read-only text (size 0x4500)	0x100	0x8000100	LOAD, read/execute
Read/write data (file size 0x2200, memory size 0x3500)	0x4600	0x8005600	LOAD, read/write/ execute
Nonloadable information and optional section headers			

FIGURE 3.16 • ELF loadable segments.

Elf Linking

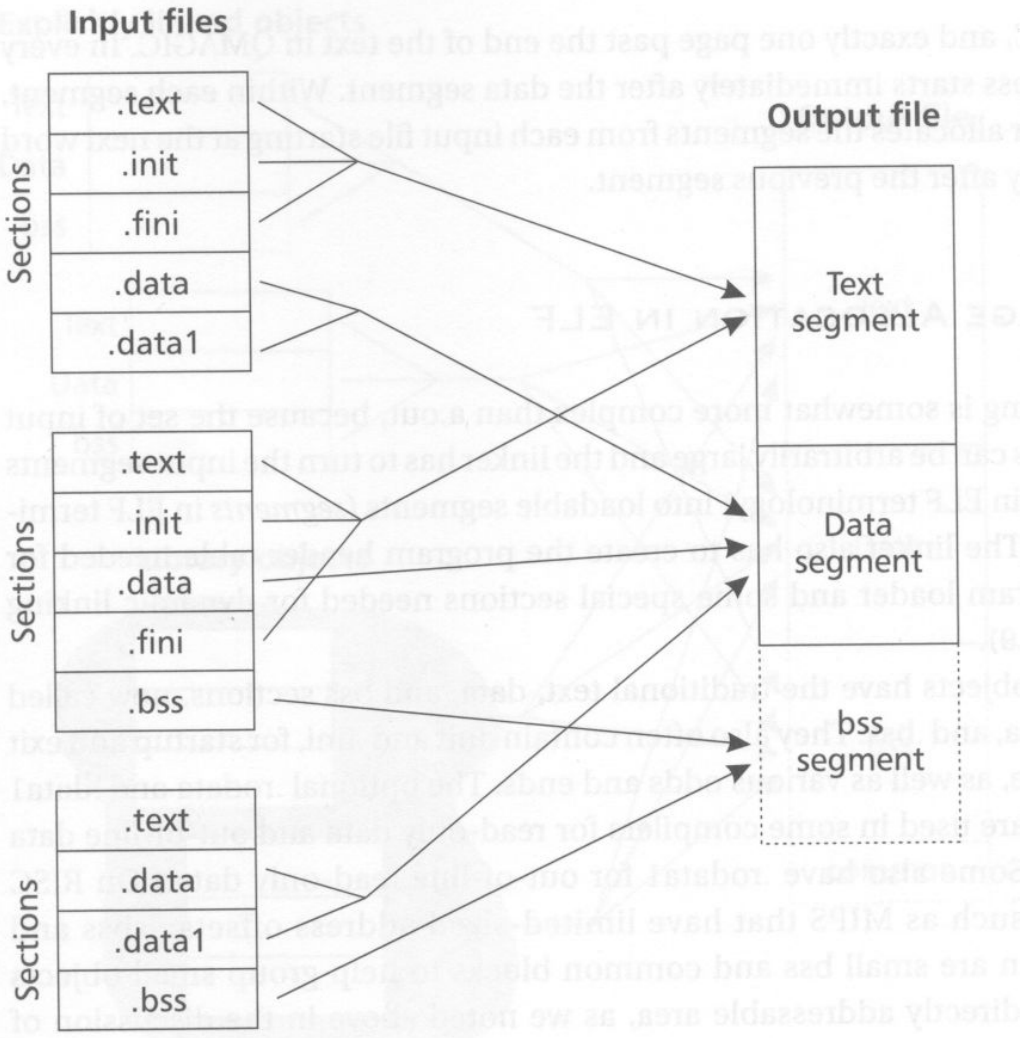


FIGURE 4.9 • ELF linking.

Elf File Trace

(We can use the objdump or nm command)

Program Header

```
Program Header:
    PHDR off    0x00000034 vaddr 0x08048034 paddr 0x08048034 align 2**2
         filesz 0x000000c0 memsz 0x000000c0 flags r-x
  INTERP off    0x000000f4 vaddr 0x080480f4 paddr 0x080480f4 align 2**0
         filesz 0x00000019 memsz 0x00000019 flags r--
    LOAD off    0x00000000 vaddr 0x08048000 paddr 0x08048000 align 2**12
         filesz 0x00000564 memsz 0x00000564 flags r-x
    LOAD off    0x00000564 vaddr 0x08049564 paddr 0x08049564 align 2**12
         filesz 0x000000a8 memsz 0x000000cc flags rw-
 DYNAMIC off    0x0000059c vaddr 0x0804959c paddr 0x0804959c align 2**2
         filesz 0x00000070 memsz 0x00000070 flags rw-
    NOTE off    0x00000110 vaddr 0x08048110 paddr 0x08048110 align 2**2
         filesz 0x00000018 memsz 0x00000018 flags r--
```

Dynamic Section

```
Dynamic Section:
  NEEDED       libc.so.4
  INIT         0x8048390
  FINI         0x8048550
  HASH         0x8048128
  STRTAB       0x80482c8
  SYMTAB       0x80481b8
  STRSZ        0xad
  SYMENT       0x10
  DEBUG        0x0
  PLTGOT       0x8049584
  PLTRELSZ     0x18
  PLTREL       0x11
  JMPREL       0x8048378
```

Need to link this shared library for printf()

Section Header

```
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .interp       00000019  080480f4  080480f4  000000f4  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .note.ABI-tag 00000018  08048110  08048110  00000110  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .hash         00000090  08048128  08048128  00000128  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .dynsym       00000110  080481b8  080481b8  000001b8  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .dynstr       000000ad  080482c8  080482c8  000002c8  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  5 .rel.plt      00000018  08048378  08048378  00000378  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  6 .init         0000000b  08048390  08048390  00000390  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  7 .plt          00000040  0804839c  0804839c  0000039c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  8 .text         00000174  080483dc  080483dc  000003dc  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
```

Section Header (cont'd)

```
 9 .fini         00000006  08048550  08048550  00000550  2**2
                 CONTENTS, ALLOC, LOAD, READONLY, CODE
10 .rodata       0000000e  08048556  08048556  00000556  2**0
                 CONTENTS, ALLOC, LOAD, READONLY, DATA
11 .data         0000000c  08049564  08049564  00000564  2**2
                 CONTENTS, ALLOC, LOAD, DATA
12 .eh_frame     00000004  08049570  08049570  00000570  2**2
                 CONTENTS, ALLOC, LOAD, DATA
13 .ctors        00000008  08049574  08049574  00000574  2**2
                 CONTENTS, ALLOC, LOAD, DATA
14 .dtors        00000008  0804957c  0804957c  0000057c  2**2
                 CONTENTS, ALLOC, LOAD, DATA
15 .got          00000018  08049584  08049584  00000584  2**2
                 CONTENTS, ALLOC, LOAD, DATA
16 .dynamic      00000070  0804959c  0804959c  0000059c  2**2
                 CONTENTS, ALLOC, LOAD, DATA
17 .bss          00000024  0804960c  0804960c  0000060c  2**2
                 ALLOC
18 .stab         000001bc  00000000  00000000  0000060c  2**2
                 CONTENTS, READONLY, DEBUGGING
19 .stabstr      00000388  00000000  00000000  000007c8  2**0
                 CONTENTS, READONLY, DEBUGGING
20 .comment      000000c8  00000000  00000000  00000b50  2**0
```

Symbol Table

```
SYMBOL TABLE:
080480f4 l    d  .interp        00000000
08048110 l    d  .note.ABI-tag  00000000
08048128 l    d  .hash  00000000
080481b8 l    d  .dynsym        00000000
080482c8 l    d  .dynstr        00000000
08048378 l    d  .rel.plt       00000000
08048390 l    d  .init  00000000
0804839c l    d  .plt   00000000
080483dc l    d  .text  00000000
08048550 l    d  .fini  00000000
08048556 l    d  .rodata        00000000
08049564 l    d  .data  00000000
08049570 l    d  .eh_frame      00000000
08049574 l    d  .ctors 00000000
0804957c l    d  .dtors 00000000
08049584 l    d  .got   00000000
0804959c l    d  .dynamic       00000000
```

Symbol Table (cont'd)

```
0804960c l    d  .bss   00000000
00000000 l    d  .stab  00000000
00000000 l    d  .stabstr       00000000
00000000 l    d  .comment       00000000
00000000 l    d  .note  00000000
00000000 l    d  *ABS*  00000000
00000000 l    d  *ABS*  00000000
00000000 l    d  *ABS*  00000000
00000000 l    df *ABS*  00000000 crtstuff.c
08048460 l       .text  00000000 gcc2_compiled.
08049568 l     O .data  00000000 p.3
0804957c l     O .dtors 00000000 __DTOR_LIST__
0804956c l     O .data  00000000 completed.4
08048460 l     F .text  00000000 __do_global_dtors_aux
08049570 l     O .eh_frame       00000000 __EH_FRAME_BEGIN__
```

Symbol Table (cont'd)

```
080484b4 l     F .text  00000000 fini_dummy
0804960c l     O .bss   00000018 object.11
080484bc l     F .text  00000000 frame_dummy
080484e0 l     F .text  00000000 init_dummy
08049570 l     O .data  00000000 force_to_data
08049574 l     O .ctors 00000000 __CTOR_LIST__
00000000 l    df *ABS*  00000000 crtstuff.c
08048520 l        .text  00000000 gcc2_compiled.
08048520 l     F .text  00000000 __do_global_ctors_aux
08049578 l     O .ctors 00000000 __CTOR_END__
08048548 l     F .text  00000000 init_dummy
08049570 l     O .data  00000000 force_to_data
08049580 l     O .dtors 00000000 __DTOR_END__
08049570 l     O .eh_frame      00000000 __FRAME_END__
00000000 l    df *ABS*  00000000 p10.c
080483ac       F *UND*  00000031 printf
0804959c g     O *ABS*  00000000 _DYNAMIC
08048550 g     O *ABS*  00000000 _etext
08048390 g     F .init  00000000 _init
08049624 g     O .bss   00000004 environ
00000000  w      *UND*  00000000 __deregister_frame_info
08049630 g     O *ABS*  00000000 end
08049628 g     O .bss   00000004 xx
```

Symbol Table (cont'd)

```
08049564 g     O .data  00000004 __progname
080483dc g     F .text  00000083 _start
0804960c g     O *ABS*  00000000 __bss_start
080484e8 g     F .text  00000038 main
08048550 g     F .fini  00000000 _fini
0804962c g     O .bss   00000004 yy
080483bc       F *UND*  00000070 atexit
0804960c g     O *ABS*  00000000 _edata
08049584 g     O *ABS*  00000000 _GLOBAL_OFFSET_TABLE_
08049630 g     O *ABS*  00000000 _end
080483cc       F *UND*  0000005b exit
00000000  w      *UND*  00000000 __register_frame_info
```

Dynamic Symbol Table

```
DYNAMIC SYMBOL TABLE:
080483ac      DF *UND*  00000031 printf
0804959c g    DO *ABS*  00000000 _DYNAMIC
08048550 g    DO *ABS*  00000000 _etext
08048390 g    DF .init  00000000 _init
08049624 g    DO .bss   00000004 environ
00000000  w   D  *UND*  00000000 __deregister_frame_info
08049630 g    DO *ABS*  00000000 end
08049564 g    DO .data  00000004 __progname
0804960c g    DO *ABS*  00000000 __bss_start
08048550 g    DF .fini  00000000 _fini
080483bc      DF *UND*  00000070 atexit
0804960c g    DO *ABS*  00000000 _edata
08049584 g    DO *ABS*  00000000 _GLOBAL_OFFSET_TABLE_
08049630 g    DO *ABS*  00000000 _end
080483cc      DF *UND*  0000005b exit
00000000  w   D  *UND*  00000000 __register_frame_info
```

Dynamic Relocation Table

- DYNAMIC RELOCATION RECORDS
- OFFSET TYPE VALUE
- 08049590 R_386_JUMP_SLOT printf
- 08049594 R_386_JUMP_SLOT atexit
- 08049598 R_386_JUMP_SLOT exit

Library

Library is a collection of resources used to develop software.

These may include pre-written code and subroutines, classes, values or type specifications.

Libraries contain code and data that provide services to independent programs.

Adv- sharing and changing of code and data in a modular fashion, and eases the distribution of the code and data.

Some executables are both standalone programs and libraries, but most libraries are not executable. Executables and libraries make references known as links to each other through the process known as linking, which is typically done by a linker.

Shared Libraries

shared library or shared object is a file that is intended to be shared by executable files and further shared objects files.

Modules used by a program are loaded from individual shared objects into memory at load time or run time, rather than being copied by a linker when it creates a single monolithic executable file for the program.

Shared Libraries

Shared libraries remove the common library routines from the executable file

Maintaining a single copy of the library routine somewhere in memory that all processes reference.

This reduces the size of each executable file (But may add some runtime overhead,either when the program is first executed or the first time each shared library function is called)

Another advantage of shared libraries is that library functions can be replaced with new versions without having to relink edit every program that uses the library.

```
$ cc -static hello1.c            prevent gcc from using shared libraries

$ ls -l a.out
-rwxrwxr-x 1 sar        475570 Feb 18 23:17 a.out

$ size a.out
   text     data     bss     dec     hex    filename
 375657     3780    3220  382657   5d6c1    a.out
```

If we compile this program to use shared libraries, the text and data sizes of the executable file are greatly decreased:

```
$ cc hello1.c                    gcc defaults to use shared libraries

$ ls -l a.out
-rwxrwxr-x 1 sar         11410 Feb 18 23:19 a.out

$ size a.out
   text     data     bss   dec       hex    filename
    872      256       4  1132       46c    a.out
```

http://tldp.org/HOWTO/Program-Library-HOWTO/shared-libraries.html