

CSE 4105 • SYSTEM PROGRAMMING • SHORT COURSE

Class 5

Linking and the ELF Format

shared libraries, and what is inside an object file

Md Mahbub E Noor

Assistant Professor, Department of CSE

University of Barishal

CLASS

5 / 10

20 slides, taken from the full course:

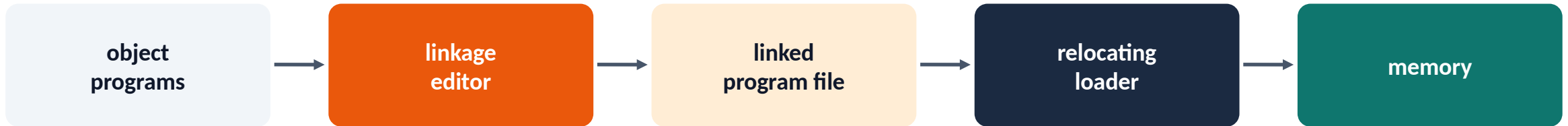
Lecture 5.2, Lecture 6

Two ways to organise linking and loading

Linking loader (Lecture 5.1): link and load together, every run



Linkage editor: link once, save the result; load it many times



SAME AS C

ld is a linkage editor

`gcc main.o counta.o data.o -o count` links once and writes the file `count`. You run it a thousand times without linking again.

DIFFERENT FROM C

But not the whole story

`count` still needs `libc.so.6`: that part is linked **every run** by `ld.so`. Linux mixes both designs (next part).

Book: Beck §3.4.1 (linkage editors)

PART 3

Dynamic linking

Link the library when the program starts, or even later: ld.so, GOT and PLT.

Make a shared library and use it

```
demo folder (real)

$ gcc -fPIC -c counta.c -o counta_pic.o
$ gcc -fPIC -c data.c -o data_pic.o
$ gcc -shared -o libcount.so counta_pic.o data_pic.o
$ gcc -o count_so main.o -L. -lcount
$ ./count_so
./count_so: error while loading shared libraries:
libcount.so: cannot open shared object file
exit status 127
$ LD_LIBRARY_PATH=. ./count_so
total = 3
$ gcc -o count_rp main.o -L. -lcount -Wl,-rpath,'$ORIGIN'
$ ./count_rp
total = 3
```

-fPIC

Position-independent code: the library may be loaded at **any** address, so its code must not contain absolute addresses.

Linking ≠ copying

The link step only checks that counta exists and writes "I need libcount.so" into count_so. The code is not copied.

The error

At **run time** ld.so must find the file. The current folder is not searched. LD_LIBRARY_PATH or a RUNPATH (\$ORIGIN = the program's own folder) fixes it.

Who links at run time? The dynamic linker ld.so

```
● ● ● readelf -l count_dyn (excerpt, real)
```

```
Entry point 0x401050
```

```
INTERP [Requesting program interpreter:  
         /lib64/ld-linux-x86-64.so.2]
```

```
LOAD ... (4 segments)
```

```
● ● ● ldd and a direct start (real)
```

```
$ LD_LIBRARY_PATH=. ldd ./count_so  
linux-vdso.so.1 (0x00007f7fa716d000)  
libcount.so => ./libcount.so (0x00007f7fa715b000)  
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (...)  
/lib64/ld-linux-x86-64.so.2 (0x00007f7fa716f000)  
$ /lib64/ld-linux-x86-64.so.2 ./count_dyn  
total = 3
```

1. The kernel

loads count_dyn's segments, sees INTERP, and also loads ld.so. It starts **ld.so**, not your program (Lecture 8.2).

2. ld.so

reads the NEEDED list, finds and maps libcount.so and libc.so.6, relocates them, binds the names. A linking loader!

3. Your program

ld.so jumps to the entry point (`_start`), which calls main.

Book: Beck §3.4.2 (dynamic linking)

Calls through the PLT: one stub per function

```
objdump -d lazy (real)
401162: e8 e9 fe ff ff  call 401050 <counta@plt>
...
0000000000401050 <counta@plt>:
401050: f3 0f 1e fa      endbr64
401054: ff 25 a6 2f 00 00  jmp *0x2fa6(%rip) # 404000
```

```
readelf -r lazy (.rela.plt, real)
404000 R_X86_64_JUMP_SLOT counta + 0
404008 R_X86_64_JUMP_SLOT printf@GLIBC_2.2.5 + 0
```

1 main calls **counta@plt**, a stub inside the program. ld could fill that call: the stub is in the same file.

2 The stub jumps to the address stored in the **GOT slot 404000**.

3 ld.so fills slot 404000 (a JUMP_SLOT relocation = an M record for the GOT).

C link: you saw `call puts@PLT` in Lecture 1's assembly. The GOT is an array of function pointers, and the stub does `goto *got[3];`.

Watch the first call to counta

Lazy binding: the first call to counta through PLT and GOT

STARTCALL 1CALL 2

```
main (lazy.C, -z lazy)

int a = counta();
401162: call 401050 <counta@plt>

int b = counta();
40116a: call 401050 <counta@plt>
```

call 1

PLT (code in the program)

```
.plt (PLT0)
401020: push [403ff0]    GOT[1]
401026: jmp  *[403ff8]   GOT[2]
401030: endbr64
401034: push $0x0       reloc 0
401039: jmp  401020     to PLT0
.plt.sec
401050: endbr64         counta@plt
401054: jmp  *[404000]  GOT[3]
```

GOT (.got.plt, writable data)

403fe8 GOT[0]	0000000000403de8	<i>_DYNAMIC</i>
403ff0 GOT[1]	00007ffff7ffe2e0	<i>link_map</i>
403ff8 GOT[2]	00007ffff7fda2f0	<i>resolver</i>
404000 GOT[3]	00007ffff7fb70f9	<i>counta</i>
404008 GOT[4]	0000000000401040	<i>printf</i>

Who writes what

PLT: read-only code, made by the linker (ld)

GOT: data, filled in at run time by ld.so

ld.so (/lib64/ld-linux-x86-64.so.2)

```
7ffff7fda2f0 _dl_runtime_resolve_xsavec

reads reloc 0 in .rela.plt:
JUMP_SLOT counta at 404000
looks up counta, writes GOT[3]
```

libcount.so (loaded at 7ffff7fb6000)

```
7ffff7fb70f9 counta
= base 7ffff7fb6000 + 10f9
(nm -D libcount.so: counta at 10f9)
```

The resolver looks up counta, finds it at 7ffff7fb70f9 in libcount.so, writes that into GOT[3] and jumps there. A relocation done at run time, once.

Watch for

1. At start GOT[3] points **back into the PLT** (401030).
2. push \$0x0: which relocation to fix.
3. PLT0 jumps into **ld.so** through GOT[2].
4. ld.so **writes GOT[3]**: it turns orange.
5. The second call skips ld.so completely.

Click the video to play or pause. About 1½ minutes, no sound.

Static or dynamic linking? The trade-offs

Question	Static (-static)	Dynamic (default)
File size (count)	785 504 bytes	16 064 bytes
Copies of printf in memory, 50 programs	50	1, shared
libc security fix	relink every program	update libc.so.6 once
Start-up	fast: nothing to find	ld.so must find, map and bind libraries
Can fail at run time?	no missing libraries	yes: "cannot open shared object file"
Each call to printf	direct call	through the PLT/GOT: one extra jump
Runs on another Linux?	usually yes	needs compatible library versions

No winner: desktops and servers use dynamic linking; small tools for rescue disks and containers are often static. Know the trade-off, then choose.

Four files, one format (real output of file and readelf)

```
file main.o count count_pie libcount.so
```

```
main.o:      ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
count:       ELF 64-bit LSB executable, x86-64, ... dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2 ...
count_pie:   ELF 64-bit LSB pie executable, x86-64, ... dynamically linked ...
libcount.so: ELF 64-bit LSB shared object, x86-64, ... dynamically linked, not stripped
```

File	Made by	e_type	Sections?	Segments?	Size
main.o	gcc -c	REL (1)	yes, 14	no	1 656
count	gcc -no-pie	EXEC (2)	yes, 31	yes, 13	16 064
count_pie	gcc (default)	DYN (3)	yes, 31	yes, 13	16 160
libcount.so	gcc -shared	DYN (3)	yes, 25	yes, 11	15 248
tiny	nasm, by hand	EXEC (2)	none	yes, 1	132

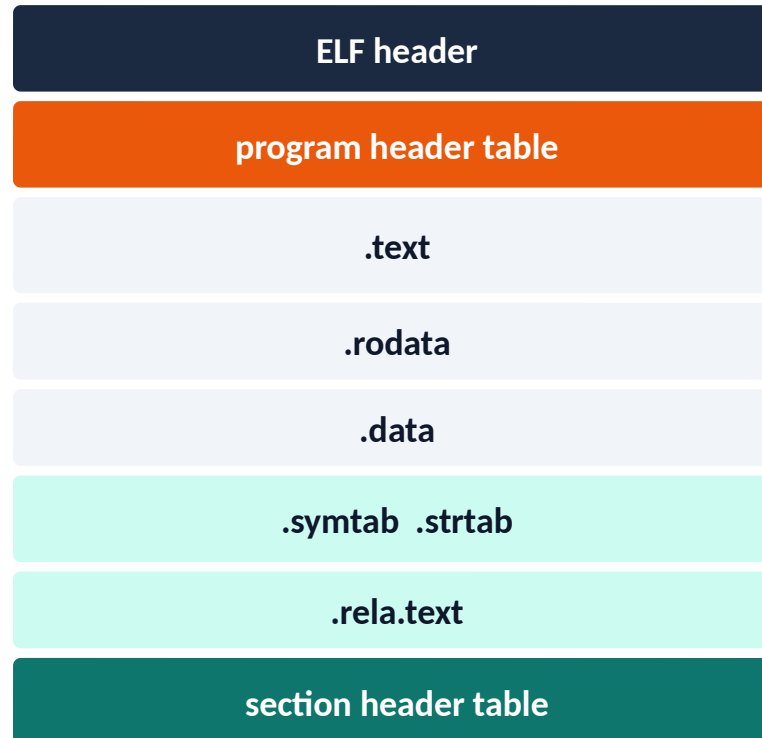
A .o file has no segments: it cannot be run, only linked. tiny has no sections: it can be run, but not linked. Two views of one format (next slide).

Book: CS:APP §7.3–7.4; man 5 elf

Two views of one file: linker and loader

Loader's view: segments

The **program header table** lists a few big pieces: "map file bytes X-Y at address A with permission r-x". The kernel reads only this. A .o file has none.



Linker's view: sections

The **section header table** (at the end) lists many small named pieces: code, data, symbols, relocations. `ld`, `readelf` and `gdb` read it. The kernel ignores it.

C link: like one array seen two ways: as `char text[16064]` (bytes) or as an array of structs. Same bytes, two tables of contents for two different readers.

Book: CS:APP §7.4 (relocatable object files), §7.9 (loading executables)

PART 2

The ELF header: the first 64 bytes

A C struct written to disk.

The first 64 bytes of main.o, and the C struct they fill

```
od -A x -t x1 -N 64 main.o (real)
```

```
000000 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
000010 01 00 3e 00 01 00 00 00 00 00 00 00 00 00 00 00
000020 00 00 00 00 00 00 00 00 00 f8 02 00 00 00 00 00
000030 00 00 00 00 40 00 00 00 00 00 40 00 0e 00 0d 00
```

7f 45 4c 46

the magic: 0x7f then "ELF". Every ELF file starts like this.

02 01 01

64-bit, little-endian, ELF version 1.

01 00 at 16

e_type = 1 = REL. Little-endian: read 01 00 as 0x0001.

f8 02 ... at 40

e_shoff = 0x2f8 = 760: the section headers start at byte 760.

```
/usr/include/elf.h
```

```
typedef struct {
    unsigned char e_ident[16]; /* 0 */
    Elf64_Half e_type; /* 16 */
    Elf64_Half e_machine; /* 18 */
    Elf64_Word e_version; /* 20 */
    Elf64_Addr e_entry; /* 24 */
    Elf64_Off e_phoff; /* 32 */
    Elf64_Off e_shoff; /* 40 */
    Elf64_Word e_flags; /* 48 */
    Elf64_Half e_ehsize; /* 52 */
    Elf64_Half e_phentsize; /* 54 */
    Elf64_Half e_phnum; /* 56 */
    Elf64_Half e_shentsize; /* 58 */
    Elf64_Half e_shnum; /* 60 */
    Elf64_Half e_shstrndx; /* 62 */
} Elf64_Ehdr; /* 64 bytes */
```

Book: *man 5 elf*; `/usr/include/elf.h`

The sections of main.o (readelf -S, real)

Nr	Name	Type	Offset	Size	Holds
1	.text	PROGBITS	0x40	0x36	the machine code of main (54 bytes)
2	.rela.text	RELA	0x1f0	0x78	5 relocations for .text (5×24 bytes)
3	.data	PROGBITS	0x76	0	initialised globals: none in main.c
4	.bss	NOBITS	0x76	0	zero globals: size only, never bytes
5	.rodata	PROGBITS	0x76	0xc	"total = %d\n" (12 bytes with \0)
11	.symtab	SYMTAB	0x108	0xc0	8 symbols (8×24 bytes)
12	.strtab	STRTAB	0x1c8	0x21	the symbol names
13	.shstrtab	STRTAB	0x280	0x74	the section names

Check the sizes: $0x78 = 120 = 5 \times 24$ (sizeof Elf64_Rela); $0xc0 = 192 = 8 \times 24$ (sizeof Elf64_Sym). Tables in ELF are just arrays of structs.

The symbol table: D and R records in one list

```
readelf -s main.o ; readelf -s data.o (real, shortened)
```

Num:	Value	Size	Type	Bind	Ndx	Name
4:	0000	54	FUNC	GLOBAL	1	main
5:	0000	0	NOTYPE	GLOBAL	UND	counta
6:	0000	0	NOTYPE	GLOBAL	UND	total
7:	0000	0	NOTYPE	GLOBAL	UND	printf
--- data.o						
2:	0000	7	OBJECT	GLOBAL	2	text
3:	0008	4	OBJECT	GLOBAL	2	len

Ndx = a section number

main lives in section 1 (.text) at offset 0: a D record.

Ndx = UND

Defined elsewhere: an R record. Value 0, nobody knows it yet.

```
struct Elf64_Sym (24 bytes)
```

```
Elf64_Word  st_name; /* offset in .strtab */
unsigned char st_info; /* type (low 4) + bind (high 4) */
unsigned char st_other; /* visibility */
Elf64_Half  st_shndx; /* section number, 0 = UND */
Elf64_Addr  st_value; /* offset in that section */
Elf64_Xword st_size; /* bytes */
```

Size and Type

SIC had neither. ELF knows main is a 54-byte function and text a 7-byte object: gdb and the linker use this.

Relocations: the M records of ELF

```
● ● ● readelf -r main.o (real, first 3 of 5)
```

Offset	Info	Type	Sym + Addend
0009	0000000500000004	R_X86_64_PLT32	counta - 4
000f	0000000600000002	R_X86_64_PC32	total - 4
0015	0000000600000002	R_X86_64_PC32	total - 4

```
● ● ● struct Elf64_Rela (24 bytes)
```

```
Elf64_Addr  r_offset; /* where to patch          */  
Elf64_Xword r_info;  /* symbol << 32 | type */  
Elf64_Sxword r_addend; /* a number to add      */
```

1

r_info 0x0000000500000004: symbol 5 (counta), type 4 (PLT32)

2

Patch the 4 bytes at offset 9: the hole in e8 00 00 00 00

3

Value = **S + A - P**: symbol + addend - place

The numbers from Lecture 5.1: S = counta = 0x40116c, A = -4, P = 0x401136 + 9 = 0x40113f. 0x40116c - 4 - 0x40113f = **0x29**: exactly the e8 29 00 00 00 ld wrote. A relocation is an M record that also says **how** to compute the value.

Book: CS:APP §7.7 (relocation)

Program headers of count: four LOAD segments

Type	Offset	VirtAddr	FileSiz	MemSiz	Flags	Sections inside
INTERP	0x318	0x400318	0x1c	0x1c	R	.interp: "/lib64/ld-linux-x86-64.so.2"
LOAD	0x0	0x400000	0x500	0x500	R	headers, .dynsym, .rela.plt, ...
LOAD	0x1000	0x401000	0x1bd	0x1bd	R E	.init .plt .plt.sec .text .fini
LOAD	0x2000	0x402000	0x114	0x114	R	.rodata .eh_frame
LOAD	0x2df8	0x403df8	0x22c	0x238	RW	.dynamic .got .got.plt .data .bss
DYNAMIC	0x2e08	0x403e08	0x1d0	0x1d0	RW	.dynamic: the table for ld.so

Grouped by permission

Many sections, few segments: all code together (R E), all writable data together (RW). The kernel maps each LOAD with mmap.

MemSiz > FileSiz

$0x238 - 0x22c = 12$ bytes of .bss: no bytes in the file, zero-filled in memory.

Offset $\equiv$ address (mod 0x1000)

0x2df8 and 0x403df8 end the same: pages of the file map onto pages of memory.

Book: CS:APP §7.8–7.9 (executable object files, loading)

e_entry points to _start, not to main

```
objdump -d count (real, excerpt)
```

```
0000000000401050 <_start>:
```

```
401054: 31 ed      xor    %ebp,%ebp
40105d: 48 83 e4 f0 and    $0xfff...f0,%rsp
```

```
...
```

```
401068: 48 c7 c7 36 11 40 00 mov    $0x401136,%rdi
40106f: ff 15 63 2f 00 00    call  *... <__libc_start_main>
401075: f4          hlt
```

Entry point 0x401050 = _start

A small piece of code from the C library (crt1.o), linked into every C program.

0x401136 = main

_start passes main's address to __libc_start_main, which sets up the C library, calls main, then calls exit with main's return value.

C link: who calls main? You always wrote `int main(void)` as if it were the start. It is the start of **your** code only. `return 0;` in main becomes `exit(0)` in `__libc_start_main`.

hlt is never reached

`__libc_start_main` never returns. If it did, `hlt` (a privileged instruction) would crash the program.

Watch the kernel read tiny, byte by byte

How Linux reads a 132-byte program

ELF HEADERPROGRAM HEADERLOADRUN

the file tiny (od -A x -t x1)

0000	7f	45	4c	46	02	01	01	00	00	00	00	00	00	00	00
0010	02	00	3e	00	01	00	00	00	78	00	40	00	00	00	00
0020	40	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0030	00	00	00	00	40	00	38	00	01	00	00	00	00	00	00
0040	01	00	00	00	05	00	00	00	00	00	00	00	00	00	00
0050	00	00	40	00	00	00	00	00	00	00	40	00	00	00	00
0060	84	00	00	00	00	00	00	00	84	00	00	00	00	00	00
0070	00	10	00	00	00	00	00	00	bf	2a	00	00	00	b8	3c
0080	00	00	0f	05											

ELF header 0-63

program header 64-119

code 120-131

struct Elf64_Phdr (56 bytes)

p_type1 = PT_LOAD

p_flags5 = R + X

p_offset

p_vaddr

p_paddr

p_filesz

p_memsz

p_align

$p_flags = 5 = 4 \text{ (read)} + 1 \text{ (execute)}$. No 2 (write): the code cannot change itself.

Watch for

1. Every number is read **backwards** (little-endian).
2. **e_phoff** sends the kernel to byte 64.
3. **e_shoff = 0**: sections are never needed.
4. **PT_LOAD** maps the file at 0x400000, r-x.
5. Three instructions end in **exit(42)**.

Click the video to play or pause. About 2 minutes, no sound.

Key ideas from Class 5

- 1 Static linking copies library code into the program; dynamic linking leaves it in a shared library (.so).
- 2 ld.so, the dynamic linker, loads the libraries and connects the calls when the program starts.
- 3 A call to a library function goes through the PLT and a GOT slot that is filled on the first call.
- 4 ELF is one format for .o files, executables and libraries: header, sections for the linker, segments for the kernel.
- 5 The symbol table and relocations in ELF are the D, R and M records of SIC.

Homework and practice

Homework 5 (bring it to the next class)

1. Give one advantage and one disadvantage of dynamic linking.
2. What is stored in .text, .data, .bss and .rodata? Give one line of C for each.
3. Why does `e_entry` point to `_start` and not to `main`?

Try it in the browser

ELF explorer: click the bytes of a real ELF header

[labs/lab06.html](#)

On the course website: open the lab page and choose "Try it here".

Next class: The Linux Kernel and Booting