

Code Generation for Data Processing

Lecture 8: Object Files, Linker, and Loader

Alexis Engelke

Chair of Data Science and Engineering (I25)
School of Computation, Information, and Technology
Technical University of Munich

Winter 2022/23

Overview: Post-compilation

- ▶ Compiler emits object file
 - ▶ Somehow? Some format?
- ▶ Linker merges object files and determines required shared libraries
 - ▶ Somehow resolves missing symbols?
- ▶ Linker creates executable file
 - ▶ Somehow? Some format the OS understands?
- ▶ Kernel loads executable file into memory
- ▶ Someone loads shared libraries

Code Model and Position Independent Code

- ▶ Code Model = address constraints
- ▶ Allows for better code
 - ▶ Long addrs/offsets = more instrs.
- ▶ Exact constraints arch-specific

- ▶ x86-64 SysV ABI:
 - ▶ Small: code and data max. 2 GiB
 - ▶ Medium: code max. 2 GiB
 - ▶ Large: no restrictions

- ▶ non-PIC: absolute addresses fixed at link-time
 - ▶ Addrs can be encoded directly
 - ▶ Sometime slightly faster
 - ▶ Not possible for shared libs

- ▶ PIC: address random at load time
 - ▶ Offsets need be PC-relative
 - ▶ Addresses need fixup at load time (e.g., in jump tables)

Compiler needs to know code model

Section 1

Object Files

Executable and Linkable Format (ELF)

- ▶ Widely used format for code
 - ▶ REL: relocatable/object file
 - ▶ EXEC: executable (non-PIE)
 - ▶ DYN: shared library/PIE
 - ▶ CORE: coredump
- ▶ ELF header: general information
- ▶ Program headers: used for execution
- ▶ Section headers: used for linking

ELF Header
Program Headers (not for REL)
.text
.rodata
.data
...
e.g., symtab, debug
Section Headers (primarily for REL)

ELF Header

```
// from glibc's elf.h
typedef struct {
    unsigned char e_ident[EI_NIDENT]; /* Magic number and other info */
    Elf64_Half e_type; /* Object file type */
    Elf64_Half e_machine; /* Architecture */
    Elf64_Word e_version; /* Object file version */
    Elf64_Addr e_entry; /* Entry point virtual address */
    Elf64_Off e_phoff; /* Program header table file offset */
    Elf64_Off e_shoff; /* Section header table file offset */
    Elf64_Word e_flags; /* Processor-specific flags */
    Elf64_Half e_ehsize; /* ELF header size in bytes */
    Elf64_Half e_phentsize; /* Program header table entry size */
    Elf64_Half e_phnum; /* Program header table entry count */
    Elf64_Half e_shentsize; /* Section header table entry size */
    Elf64_Half e_shnum; /* Section header table entry count */
    Elf64_Half e_shstrndx; /* Section header string table index */
} Elf64_Ehdr;
```

ELF Sections

- ▶ Structures content of object files for linker
 - ▶ Linker later merges content sections of same “type”
- ▶ Some sections have “meta” information (e.g., symbols)

- ▶ `.text` – program text/code, executable
- ▶ `.rodata` – read-only data
- ▶ `.data` – initialized data, writable
- ▶ `.bss` – zero-initialized data, no storage, writable
- ▶ `.symtab` – symbol table, references string table for names
- ▶ `.strtab` – string table for symbol names
- ▶ `.shstrtab` – string table for section header

ELF Section Header

```
typedef struct {
    Elf64_Word sh_name; /* Section name (string tbl index) */
    Elf64_Word sh_type; /* Section type */
    // SHT_{NULL,PROGBITS,SYMTAB,STRTAB,RELA,HASH,NOBITS,...}
    Elf64_Xword sh_flags; /* Section flags */
    // SHF_{WRITE,ALLOC,EXECINSTR,MERGE,STRINGS,...}
    Elf64_Addr sh_addr; /* Section virtual addr at execution */
    Elf64_Off sh_offset; /* Section file offset */
    Elf64_Xword sh_size; /* Section size in bytes */
    Elf64_Word sh_link; /* Link to another section */
    Elf64_Word sh_info; /* Additional section information */
    Elf64_Xword sh_addralign; /* Section alignment */
    Elf64_Xword sh_entsize; /* Entry size if section holds table */
} Elf64_Shdr;
// first section is always undefined/SHT_NULL
```


Example: Section Headers

```
void external(void);
static void bar(void) {}
void foo(void) { bar(); }
void func(void) {
    foo(); external(); }
```

Section Headers:

[Nr]	Name	Type	ES	Flg	Lk	Inf	Al
[0]		NULL	00		0	0	0
[1]	.text	PROGBITS	00	AX	0	0	1
[2]	.rela.text	RELA	18	I 10	1	8	
[3]	.data	PROGBITS	00	WA	0	0	1
[4]	.bss	NOBITS	00	WA	0	0	1
[5]	.comment	PROGBITS	01	MS	0	0	1
[6]	.note.GNU-stack	PROGBITS	00		0	0	1
[7]	.note.gnu.property	NOTE	00	A	0	0	8
[8]	.eh_frame	PROGBITS	00	A	0	0	8
[9]	.rela.eh_frame	RELA	18	I 10	8	8	
[10]	.symtab	SYMTAB	18		11	4	8
[11]	.strtab	STRTAB	00		0	0	1
[12]	.shstrtab	STRTAB	00		0	0	1

Symbol Table

- ▶ Describes symbolic reference to object/function
- ▶ Names in associated string table, referenced by byte offset

```
typedef struct {  
    Elf64_Word st_name; /* Symbol name (string tbl index) */  
    unsigned char st_info; /* Symbol type and binding */  
    unsigned char st_other; /* Symbol visibility */  
    Elf64_Section st_shndx; /* Section index */  
    Elf64_Addr st_value; /* Symbol value */  
    Elf64_Xword st_size; /* Symbol size */  
} Elf64_Sym;
```

Example: Symbol Table

```
void external(void);
static void bar(void) {}
void foo(void) { bar(); }
void func(void) {
    foo(); external(); }
```

- ▶ Ndx=UND: undefined
 - ▶ value is zero
- ▶ Ndx=ABS: no section base
 - ▶ value is absolute
- ▶ Ndx=num: section idx.
 - ▶ value is offset into sec.
 - ▶ later refers to address

Section Headers:

[Nr]	Name	Type	Size	ES	Flg	Lk	Inf	Al
[0]		NULL	000000	00		0	0	0
[1]	.text	PROGBITS	00001a	00	AX	0	0	1
	// ...							
[10]	.symtab	SYMTAB	0000a8	18		11	4	8
		sizeof(Elf64_Sym) --/						
		link to strtab -----/						
		first non-local sym -----/						
[11]	.strtab	STRTAB	00001f	00		0	0	1
[12]	.shstrtab	STRTAB	00006c	00		0	0	1

Symbol table '.symtab' contains 7 entries:

Num:	Val	Size	Type	Bind	Vis	Ndx	Name
0:	000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	000	0	FILE	LOCAL	DEFAULT	ABS	<stdin>
2:	000	0	SECTION	LOCAL	DEFAULT	1	.text
3:	000	1	FUNC	LOCAL	DEFAULT	1	bar
4:	001	6	FUNC	GLOBAL	DEFAULT	1	foo
5:	007	19	FUNC	GLOBAL	DEFAULT	1	func
6:	000	0	NOTYPE	GLOBAL	DEFAULT	UND	external

Example: Writing Code to .text

```
void external(void);
static void bar(void) {}
void foo(void) { bar(); }
void func(void) {
    foo(); external(); }
```

▶ Symbol may be unknown

▶ Linker needs to resolve
offset later

↪ Relocations

```
0000000000000000 <bar>:
    0:  c3                ret
0000000000000001 <foo>:
    1:  e8 fa ff ff ff call    0 <bar>
    6:  c3                ret
0000000000000007 <func>:
    7:  48 83 ec 08      sub     rsp,0x8
    b:  e8 00 00 00 00 call   10 <func+0x9>
    c:  R_X86_64_PC32a  foo-0x4
   10:  e8 00 00 00 00 call   15 <func+0xe>
   11:  R_X86_64_PLT32  external-0x4
   15:  48 83 c4 08      add     rsp,0x8
   19:  c3                ret
```

^aRecent GAS emits R_X86_64_PLT32, which is equivalent for local symbols.

Relocations

- ▶ Problem: symbol values unknown before linking
 - ▶ External symbols: unavailable; other section: distance unknown
- ▶ Idea: store *relocations* \Rightarrow linker patches code/data
- ▶ Relocation: quadruple of (offset in sec., type, symbol idx, addend)
- ▶ Contained in REL/RELA/RELR sections

Static Relocation

ET_REL

- ▶ For static linker (ld)
- ▶ Either: resolve or emit dyn. reloc

Dynamic Relocation

ET_EXEC/ET_DYN

- ▶ For dynamic linker/loader
- ▶ Shall be fast, outside code

Relocation Types

► Types and meaning defined by psABI³⁷

P: address of place being relocated; **S**: symbol address; **L**: PLT addr. for symbol; **Z**: sym. size;
A: addend; **B**: dynamic base address of shared obj.; **G**: GOT offset; **GOT**: GOT address

Name	Field	Calculation
R_X86_64_64	64	$S + A$
R_X86_64_PC32	32	$S + A - P$
R_X86_64_GOT32	32	$G + A$
R_X86_64_PLT32	32	$L + A - P$
R_X86_64_GLOB_DAT	addr	S
R_X86_64_JUMP_SLOT	addr	S
R_X86_64_RELATIVE	addr	$B + A$
R_X86_64_GOTPCREL	32	$G + GOT + A - P$
R_X86_64_GOTPCRELX		
R_X86_64_REX_GOTPCRELX		

Name	Field	Calculation
R_X86_64_32	32	$S + A$ (zext)
R_X86_64_32S	32	$S + A$ (sext)
R_X86_64_GOTOFF64	64	$S + A - GOT$
R_X86_64_GOTPC32	32	$GOT + A - P$
R_X86_64_GOT64	64	$G + A$
R_X86_64_GOTPCREL64	64	$G + GOT + A - P$
R_X86_64_GOTPC64	64	$GOT + A - P$
R_X86_64_PLTOFF64	64	$L - GOT + A$
R_X86_64_SIZE32	32	$Z + A$
R_X86_64_SIZE64	64	$Z + A$

Relocation Section

Section Headers:

[Nr]	Name	Type	Size	ES	Flg	Lk	Inf	Al
[1]	.text	PROGBITS	00001a	00	AX	0	0	1
[2]	.rela.text	RELA	000030	18	I	10	1	8
		sizeof(Elf64_Rela)	--/					
		I: info is section link	-----/					
		link to symtab	-----/					
		target sec. for relocations	-----/					
[10]	.symtab	SYMTAB	0000a8	18		11	4	8

Relocation section '.rela.text' at offset 0x1e0 contains 2 entries:

Offset	Info	Type	Symbol's Name + Addend
0000000000000000c	0000000400000002	R_X86_64_PC32	foo - 4
00000000000000011	0000000600000004	R_X86_64_PLT32	external - 4

Relocations on RISC Architectures

- ▶ RISC architectures typically have *more* relocation types
 - ▶ Example: AArch64³⁸ has >50 relocations
- ▶ Building a 64-bit address requires several instructions
(AArch64: one for bits 0–15, 16–31, ...)
 - ▶ Each instruction needs a different relocation to patch in the bits!

```
movz x0, #:abs_g0_nc:globalVariable
movk x0, #:abs_g1_nc:globalVariable
movk x0, #:abs_g2_nc:globalVariable
movk x0, #:abs_g3:globalVariable
```

- ▶ Often: page-granular address with added offset for low bits
 - ▶ `adrp` for ± 4 GiB range, add or load offset for low bits
 - ▶ Scaled load offsets require different relocations for each scale

Branch Relocations

- ▶ Branches (often) have limited range; compiler must assume max. distance
- ▶ x86-64: ± 2 GiB range, if larger use `mov` and indirect jump
- ▶ AArch64: ± 128 MiB range \rightsquigarrow executable sections must be < 127 MiB
linker will insert veneer between different `.text` sections
 - ▶ Veneer allowed to clobber inter-procedural scratch registers `x16/x17`
- ▶ *badly designed ISA*: ± 1 MiB range \rightsquigarrow needs ind. jump *often*
 - ▶ Construct 20 bit first with `auipc`, insert low 12 bit in `jalr`
 - ▶ Add new *relax* reloc: linker optimizes/relaxes code if possible
 - ▶ Changes code size, all relative offsets now need relocations, too
 - ▶ Alignment guarantees need new, special `align` relocations

Section 2

Executable Files

- ▶ Goal: combine multiple input files (.o/.so/.a) into executable or shared lib.
 1. Find and load all input files
 2. Scan input, store symbols, resolve symbols on-the-fly
 3. Create synthetic section (GOT, PLT, relocations for output file)
 4. Process relocations: create PLT/GOT entry and dynamic reloc.
 5. Optimize and deduplicate sections
 6. Write section to output file
 - ▶ Apply relocations which are now known; compress sections; etc.

ELF Executable File

- ▶ Entry in ELF header: entry address of the program
 - ▶ Typically provided by libc to call `__libc_start_main`
- ▶ Program headers: instructions for loading the program
- ▶ `PT_PHDR`: described program headers
- ▶ `PT_LOAD`: loadable segment
 - ▶ Specifies virtual address, file offset, file size/memory size, permission
 - ▶ `vaddr&(pagesize-1)==offset&(pagesize-1)` – kernel will just `mmap` the file
 - ▶ memory size > file size \Rightarrow filled up with zeros (for `.bss`)
- ▶ `PT_INTERP/PT_DYNAMIC`: when PIE or with shared libraries
- ▶ `PT_GNU_STACK`: permissions indicate whether stack is non-executable

Example: Program Headers

Program Headers:

Type	Offset	VirtAddr	FileSiz	MemSiz	Flg	Align
LOAD	0x000000	0x00400000	0x0a0d5e	0x0a0d5e	R E	0x1000
LOAD	0x0a17d8	0x004a27d8	0x005ab8	0x00b2e8	RW	0x1000

offset in file -/ | |
virtual address -----/ | |
bytes provided in file -----/ | |
segment size in mem -----/ |
(memsz > filesz = zero-filled) |
mmap protection -----/ |
// ...
GNU_STACK 0x000000 0x00000000 0x000000 0x000000 RW 0x10

- ▶ Note: the kernel always maps full pages from the file cache
- ▶ Note: first segment includes ELF header and program headers

Loading a Binary to Memory

- ▶ Load ELF header and program header
- ▶ If `ET_DYN` (\rightsquigarrow PIE), set random base added to all addresses
- ▶ Look if `PT_INTERP` is present
 - ▶ If present, load interpreter using same algorithm (but no nested interpreters)
- ▶ Iterate over `PT_LOAD` and `mmap` segments
 - ▶ May needs zeroing of last page and mapping extra zero pages
- ▶ Setup initial stack frame and auxiliary vector (e.g., with `phdr` address)
- ▶ Start execution at (the interpreter's) entry

This is the kernel's job

Section 3

Linker Optimizations

Eliminating Duplicate Strings/Constants

- ▶ Sections in different object may contain same data, e.g. strings
 - ▶ Critical for debug info (file names, function names, etc.)
- ▶ Idea: linker finds and deduplicates strings and other constant data
- ▶ Precondition: relative order of entries irrelevant

- ▶ SHF_MERGE – fixed-size entries, size stored in header
 - ▶ Collect all entries in hash map; afterwards emit all keys

- ▶ SHF_MERGE|SHF_STRINGS – NUL-terminated strings, entsize is char width
 - ▶ Precondition: strings must not contain NUL-byte
 - ▶ Tail merging: `foobar\0 + bar\0` \rightsquigarrow `foobar\0`
 - ▶ Sort strings from tail (e.g., radix sort), deduplicate neighbors

Linker Garbage Collection

- ▶ Problem: objects may contain unused functions
 - ▶ Compiler can't know whether function is used
- ▶ Idea: put all function into separate sections, drop unused sections
- ▶ Sections are considered as inseparable units
- ▶ GC roots: exported symbols, init functions, ...
- ▶ Iteratively mark all referenced sections, drop unmarked sections
- ▶ Downside: may need longer relocations \rightsquigarrow possibly less efficient code
- ▶ GCC/Clang `-ffunction-sections`, `ld --gc-sections`

Identical Code Folding

- ▶ Problem: objects may contain duplicate code
 - ▶ Same function compiled in many objs, e.g. template instantiation
- ▶ Idea: deduplicate read-only sections (same flags, contents, relocations(!))
- ▶ Hash all sections and their relocations, remove duplicates
- ▶ Repeat until convergence
 - ▶ Only after folding `foo1` and `foo2`, these become equivalent:

```
int funcA(void) { foo1(); } int funcB(void) { foo2(); }
```
- ▶ Caution: function pointers may be guaranteed to be different
- ▶ LLD has more aggressive deduplication

Link-Time Optimization

- ▶ Problem: no optimizations across object files
 - ▶ Inlining, constant propagation+cloning, specialized call conv., ...
 - ▶ Optimization across language boundaries
- ▶ Idea 1: glue all source code together, compile with `-fwhole-program`
 - ▶ Downside: single core, problematic with same-name `static` functions
- ▶ Idea 2: Use static binary optimization during linking (severely limited)
- ▶ Idea 3: dump IR into object, glue IR together (`-flto`)
 - ▶ Done as very first step at link-time
- ▶ LTO is widely used and highly effective

Section 4

Static Libraries

Static Libraries

- ▶ Archive of relocatable object files
 - ▶ Header often contains index mapping symbol to object file
 - ▶ Linker takes only object files that are needed
 - ▶ Code/data copied into final executable
-
- + Simple and fast, no ABI problems, no extra library needed at run-time
 - Larger executable files, library changes need relinking

Section 5

Shared Libraries

Shared Libraries

- ▶ Problem: code duplication, large executables, recompile needed for changes
- ▶ Idea: *share* code between different executables

- ▶ Executable references functions/objects in shared library
 - ▶ Shared libraries can refer to other shared libraries, too
 - ▶ Linker needs to retain dynamic relocations and symbols (dynamic symbol = externally visible symbol)
- ▶ Run-time loader links executable and libraries program start
 - ▶ Find and load libraries from different paths, resolve all relocations

Shared Libraries: Changes in Compiler

None 😊
(almost)

- ▶ When building a shared library, code must be position-independent

Shared Libraries: Changes in Linker

- ▶ Relocations to symbols in shared libraries must be retained
 - ▶ Store dynamic relocations and symbols in separate sections (`.dynsym`, `.rela.dyn`)
- ▶ Create table (GOT) for pointers to external function/objects
 - ▶ Allocate space where loader puts addresses, add relocations
- ▶ Create stub functions for external functions (PLT)
 - ▶ Compiler still creates near call, which gets redirected to stub
 - ▶ Stub jumps to address stored in table
- ▶ Emit `PT_DYNAMIC` segment with info for loader
 - ▶ Point loader to needed libs, relocations, `symtab`, `strtab`, ...

Global Offset Table (GOT) and Procedure Linkage Table (PLT)

- ▶ Global Offset Table: pointer table filled by loader
 - ▶ Linker emits dynamic relocations for GOT; loader fills addresses
 - ▶ Often subject to RELRO: after relocations are applied, GOT becomes read-only
- ▶ Procedure Linkage Table: stubs that perform jump using GOT

```
00401030 <func@plt>:
```

```
401030: ff 25 8a 2f 00 00 jmp     QWORD PTR [rip+0x2f8a] # GOT slot
```

- ▶ PLT can be disabled (`-fno-plt`): indirect jump is duplicated
 - ▶ Compiler emits indirect calls/jumps instead of near calls to PLT
 - ▶ Linker cannot convert into near jump if target is in same DSO

PT_DYNAMIC segment

- ▶ Loader needs to know needed libraries, flags, locations of relocations, etc.
 - ▶ Sections headers might be unavailable and more info is needed
- ▶ Info for loader stored in dynamic section

Type	Name/Value
(NEEDED)	Shared library: [libm.so.6]
(NEEDED)	Shared library: [libc.so.6]
(GNU_HASH)	0x4003c0
(STRTAB)	0x4004b8
(SYMTAB)	0x4003e0
(STRSZ)	259 (bytes)
(SYMENT)	24 (bytes)
// ...	
(NULL)	0x0

Symbol Lookup

- ▶ Symbol lookup using linear search + strcmp is slow
- ▶ Idea: linker creates hash table
 - ▶ Hash symbol names and store them in hash table
 - ▶ Dynamic symbols grouped by hash bucket
 - ▶ Additional bloom filter to avoid useless walks for absent symbols
- ▶ Lookup:
 - ▶ Compute hash of target symbol string
 - ▶ Check bloom filter, if absent: abort
 - ▶ Iterate through symbols in bucket, compare names (and version)
- ▶ Documentation unfortunately sparse⁴⁰

⁴⁰A Roenky. *ELF: better symbol lookup via DT_GNU_HASH*. [🌐 \(visited on 12/14/2022\)](#)

Miscellaneous Things

- ▶ Purpose of all these dynamic entries
- ▶ Symbols: versioning and visibility
- ▶ Constructors/destructors: called at load/unload of DSO
- ▶ Indirect functions (ifunc)
 - ▶ Function to dynamically determine actual address of symbol
 - ▶ Used e.g. for determining `memcpy` variant based on CPU features
- ▶ Dynamic loading of DSOs (`dlopen`)

Object Files, Linker, and Loader – Summary

- ▶ Compiler needs to know code model to emit proper asm code/relocations
- ▶ ELF format used for relocatable files, executables and shared libraries
- ▶ ELF relocatables structured in sections and have static relocations
- ▶ ELF dynamic executables grouped in segments and have dynamic relocations
 - ▶ Need dynamic loader to resolve dynamic relocations and shared libraries
- ▶ Linker combines relocatable files into executables or shared libraries
- ▶ Linker can perform further optimizations

Object Files, Linker, and Loader – Questions

- ▶ Which ELF file types exist? What is different?
- ▶ What are typical sections found in an ELF relocatable file?
- ▶ What information is contained in a symbol table?
- ▶ What information is required for a relocation?
- ▶ What are typical differences between static and dynamic relocations?
- ▶ Which steps and possible optimization does a linker perform?
- ▶ How does the OS load a binary into memory?
- ▶ What is the difference between static and shared libraries?
- ▶ How are symbols from other shared libraries resolved?