ELF1 7A Relocation Background - ELF Study 1999

Young W. Lim

2019-05-08 Wed

Outline

Based on

"Study of ELF loading and relocs", 1999
http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

Compling 32-bit program on 64-bit gcc

- `gcc -v`
- `gcc -m32 t.c`
- `sudo apt-get install gcc-multilib`
- `sudo apt-get install g++-multilib`
- `gcc-multilib`
- `g++-multilib`
- `gcc -m32`
- `objdump -m i386`

Handling inter-related referece

- linking in the old days
  - at compile time, inter-related references are not resolved
  - .o files include a reloc object that contains
    the information on these inter-related references
  - at link time, the linker would merge these informations
    in .o files building a table of where symbols are ultimately located.
  - the linker would run through the set of relocs, filling them in

`http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html`

Reloc attributes

- A reloc consists of three parts:
  - where in memory the fix is to be made
  - the symbol which is involved in the fix
  - an algorithm that the linker should use to create the fixup
- The algorithm can be as simple as `R_386_32`
  "use the symbol memory location; store it in binary"
- complicated, such as `R_ARM_PC26`
  "calculate the distance from here to the symbol, divide by 4, subtract 2 and add the result to the 3 lower bytes"

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

Static linking

- these relocs are scattered through the .o files,
  and are used at link time create the correct binary executable file.
- resolving all the relocs is necessary
- in the days of static linking

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

Dynamic linking

- run-time linking
  the designers of the ELF format enabled reloc entites
  to hold run-time resolution information.
- So now executable files may have relocs in them,
  even after linking
  - ELF implements run time linking
    by deferring function resolution
    until the function is called.

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

New algorihtm

- However, new algorithms are required to inform how these fixups are to be done.
- Hence the introduction of a new family of reloc algorithms

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

Relocation

- Binary executables often need certain bits of information *fixed up* before they execute
- ELF binaries carry a list of relocs (relocation table) which describe these *fixups*

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

Relocation entries

- Each reloc contains relocation entry
    - the address in the binary that is to get the *fixup* (offset)
    - the algorithm to calculate the fixup (type)
    - a symbol (string and object length)
- At *fixup* time,
  the algorithm (type) uses the offset & symbol,
  along with the value (addend) currently in the file,
  to calculate a new value to be stored into memory.

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

Code and Data

- One of the characteristics of the ELF binary system is a separation of code and data.
- The code of apps and libraries is marked read-only and executable
- The data is marked read-write, and not-executable.

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

Code segment (1)

- The code is read-only
  so that multiple processes can share the code,
  - the code is loaded into memory only once.
  - the code is never modified,
    and appears identical in each process space.

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

Code segment (2)

- Each process has its own page tables,
  mapping the code into its own memory.
  - therefore the code must be position independent
  - each process can load the code into a different address

- The code segment is allowed to contain
  constant pointers and strings (.rodata).

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

Data segment (1)

- The data segment is read-write and
  is mapped into each process space differently.
- In Linux, each data segment is loaded
  from the same base mmap (identical),
  but it is marked copy-on-write (own copy later)

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

Data segment (2)

- after the first write,
  each process has its own copy of the data.
  (in its own read-write segment)

- therefore, relocs can only point
  to the data segment (_identically_)

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

Relocs in code and data segments

- the relocs in the data segment are *easy* to be done
  - add relative offsets or
  - write absolute addresses
- the relocs in the code area are more *difficult*.
  - the ELF reloc design makes the code relocs *intact*
  - an GOT entry in the data area is to be filled, (Global Offset Table).

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

Relocs using GOT for a global object

- if code needs to refer to a global object,
  it refers to an entry in the `GOT[]`,
  - at run-time, the `GOT` entry is *fixed-up*
    to point to the correct address of the global object.
  - the code space need never be *fixed-up* at run time.

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

Relocs using GOT for a local object

- if the code needs to refer to a local object, it refers to it relative to the `&GOT[ 0]`;
  - no run-time *fixed-up*
  - this too is position independent

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

Relocs using PLT

- If the code needs to jump to a subroutine
  in a different module,
  the linker creates an array of *jump-stubs*
  called the PLT (procedure linkup table)

- these *jump-stubs* in the PLT jump indirect,
  using an entry in the GOT[]
  to implement the far call

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

Deferring function resolution

- ELF implements run time linking
  by deferring function resolution
  until the function is called.

- calls to library functions go through a *fix-up* process
  just after the first time call is made

`http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html`

GOT relative

- GOT-relative (GOTOFF) code is made relative to the start of the GOT table (O)
- relative to the load address of the module (X)

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

static and automatic variables

- static variables
  - *initialized* at compile time,
    since their address is known and fixed.
  - initialization to zero does not incur a run time cost
- automatic variables
  - initialized at run time
    incurs run time cost
    each time the function is called
  - different addresses for each different call
  - if you do need that initialization, then request it.

https://stackoverflow.com/questions/14049777/why-are-global-variables-always-init

global and static variables

- global and static variables are stored
  - in the .data section when initialized
  - in the .bss section when uninitialized
  - a fixed memory location is allocated at compile time.
  - thus, have '0' as their default values.
- auto variables are stored
  - on the stack, not a fixed memory location
  - stack memory is allocated at run time
  - thus, have their default value as garbage

https://stackoverflow.com/questions/14049777/why-are-global-variables-always-init

uninitialized global and static variables

- an object with automatic storage duration
  if not initialized explicitly, its value is indeterminate
- an object that has static storage duration
  if not initialized explicitly, then:
  - if it has pointer type, a null pointer;
  - if it has arithmetic type, (signed or unsigned) zero;
  - if it is an aggregate, every member is initialized
  - if it is a union, the first named member is initialized

https://stackoverflow.com/questions/13251083/the-initialization-of-static-variable

universal zero initializer

- the universal zero initializer
  - initializes everything in an object to 0,
    whether it's static or not

```
-   sometype      var = {0};
  - someothertype var[SOMESIZE] = {0};
  - anytype       var[SIZE1][SIZE2][SIZE3] = {0};
```

https://stackoverflow.com/questions/13251083/the-initialization-of-static-variable

initialized and uninitialized static variables

- static variables (or arrays)
  - Initialized static variables
    - given value from code at compile time
    - *usually* stored in `.data` though this is compiler specific
  - Uninitialized statics
    - initialized at run time
    - stored into `.bss` though again this is compiler specific

https://stackoverflow.com/questions/13251083/the-initialization-of-static-variable

.bss (1) uninitialized or initialized to zero

- The .bss section is guaranteed to be all zeros when the program is loaded into memory.
- the .bss section can have global data
  - uninitialized
  - initialized to zero
- `static int g_myGlobal = 0;      // <--- in .bss section`

https://stackoverflow.com/questions/16557677/difference-between-data-section-and-t

.bss (2) no region of zero

- the .bss section data are not included in the ELF file on disk
  - there isn't a whole region of zeros
    in the file for the .bss section
- instead, the loader knows from the section headers
  how much to allocate for the .bss section,
  and simply zero it out before transfer control

https://stackoverflow.com/questions/16557677/difference-between-data-section-and-t

.bss (3) PROGBITS vs NOBITS

- the `readelf -S` section headers output:

```
[ 3] .data PROGBITS 00000000 000110 000000 00 WA 0 0 4
[ 4] .bss  NOBITS   00000000 000110 000000 00 WA 0 0 4
```

- .data is marked as PROGBITS
  - there are "bits" of program data in the ELF file that the loader needs to read out into memory
- .bss is marked NOBITS
  - there's nothing in the file that needs to be read into memory as part of the load.

https://stackoverflow.com/questions/16557677/difference-between-data-section-and-t

.rel.text section

- a list of locations in the .text section to be modified when the linker combines this object file with others
- modify any instruction in the code section that
  - calls an external function or
  - references a global variable
- do not modify any instructions in the code section that
  - calls local functions
- executable files do not include relocation information unless the user explicitly instructs the linker

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

.rel.data section

- relocation information for any global variable that are referenced or defined by the module
- modify the initial value of any initialized global variable whose initial value is
    - the address of a global variable or
    - externally defined function

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

Global symbol relocs

- global relocs must neccessarily involve the three aspects of a reloc:
  - where in memory the reloc is to be made
  - the symbol involved in the reloc
  - the algorithm used to make the fixup.

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

Local symbol relocs

- a local symbol can be fixed in memory with respect to a memory "section",
- the object file is allowed to drop the local symbol name, and replace it with a section-plus-offset

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

ARM code example (1)

```
        .section .text
        mov     r0, r0   @sample code
.L2:    call    _do_something
        ldr     r6, .L3  @this code need a reloc!
        mov     r0, r0
.L4:    .word   Lextern
.L3:    .word   .L2      @this read-only data needs a reloc
```

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

ARM code example (2)

- the code on the 3rd line (the call) needs to be fixed up, but that's easy, since it's a PC relative fixup.

```
.L2:    call    _do_something
```

- If the .o file has no idea where .Lextern is,

```
.L4:    .word   Lextern
```

  it must neccessarily create a reloc which refers to symbol Lextern

```
.L4:    .word   0   R_ARM_32 Lextern
```

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

ARM code example (3)

- the word at .L3 needs a fixup as well.

```
.L3:    .word  .L2      @this read-only data needs a reloc
```

- If the .o file can determine the location of a local symbol, such as L2, then it is allowed to replace the symbol with a section-plus-offset
- The offset is stored in the reloc target address, and the section is an entry in the reloc symbol table

```
.L3:    .word  4  R_ARM_32 .text
```

- This reduces the number of symbols in the symbol table, making run-time linking easier.

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

ARM code example (4)

- the R_*_GOTOFF and R_*_GOT32 relocs include
  - R_386_GOTOFF : GOT-relative, local symbol address
  - R_386_GOT32 : GOT-relative, GOT entry address

  an offset from &GOT[0], which is usually about halfway through the module.
- The R_ARM_RELATIVE relocs, on the other hand, contains an offset from the beginning of the module. Tradition.

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

Lazy binding and constraints

- ELF dynamic linking defers the resolution of jump / call addresses until the last minute.
- Constraints
  - should not force a change in the assembly code produced for apps but may cause changes as an assembly code is changed for PIC
  - all executable codes must not be modified at run time any modified data must not be executed at run time

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

Three steps in a far jump

1. start in the code
2. go through the PLT
3. using a pointer from the GOT

- the GOT entries that are used for PLT execution are preloaded to default addresses
- back to the corresponding PLT entry stub
- push and jmp PLT[0] sequence

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

In the code

```
call function_call_n
```

- the *relative* jump or call
- the target is a PLT entry PLT[n+1]
  - it is (n+1)-th entry not n-th entry
  - consider that PLT[0] is a special entry
- identical to a normal call
- assume n is a number

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

(1) PLT entry : stub code

- the PLT is a synthetic area, created by the linker
- exists in both executable and libraries
- an array of stubs, one per imported function call
- through the special entry PLT[0], the resolver is called at last

```
PLT[n+1]: jmp     *GOT[n+3]
          push    #n            ; push n as a argument to the resolver
          jmp     PLT[0]
```

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

(2) indirect call through GOT

- a call to PLT[n+1] will result in *indirect call* through GOT[n+3]
  - because of three special GOT entries : GOT[0,1,2]

    ```
    jmp *GOT[n+3]    ; 6-byte long
    ```

- initially, the value at GOT[n+3] points back to PLT[n+1]+6
  - the default value at GOT[n+3] is PLT[n+1]+6
  - the next instruction after the 6 byte instruction

```
PLT[n+1]:     jmp     *GOT[n+3]     ; 6 bytes insturction
PLT[n+1]+6:   push    #n            ; push n as a argument to the resolver
              jmp     PLT[0]
```

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

(3) push, jmp sequence

- at PLT[n+1], n is pushed onto the stack
  as an argument for the resolver
- consider n as an ID for a library function
- the resolver uses the argument n on the stack
  in resolving the symbol n

```
PLT[n+1]:     jmp    *GOT[n+3]    ; 6 bytes insturction
PLT[n+1]+6:   push   #n           ; push n as a argument to the resolver
              jmp    PLT[0]
```

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

(4) overriding the default GOT[n+3]

- the resolver is called by the stub at PLT[0]
- the resolver modifies the default value at GOT[n+3] to point the correct target symbol n
- overrides PLT[n+1]+6 (the default value at G[n+3])

- thus after the first call, the control is taken directly to the correct target symbol n (function_call_n) instead of executing the push-jump sequence

```
PLT[n+1]:     jmp    *GOT[n+3]     ; 6 bytes insturction
PLT[n+1]+6:   push   #n            ; push n as a argument to the resolver
              jmp    PLT[0]
```

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

(5) the special entry PLT[0]

- the resolver needs 2 argument
  - symbol n is already on the stack
  - pointer to the relocation table : GOT[1]
  - &GOT[1] is added on the stack
- the resolver that is located in ld-linux.so.2
  can determine *which library function* is asked for its service
  using these two arguments on the stack
- GOT[2] : entry point of dynamic linker

```
PLT[0]:    push    &GOT[1]
           jmp     GOT[2]          ; entry point of dynamic linker
```

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

.dynamic section (1)

- if an object file participates in dynamic linking, its program header table will have an element of type PT_DYNAMIC.
- this segment contains the .dynamic section.
- A special symbol, _DYNAMIC, labels the section, which contains an array of the dynamic structures

https://docs.oracle.com/cd/E23824_01/html/819-0690/chapter6-42444.html

.dynamic section (2)

- _DYNAMIC symbol enables a program, such as the runtime linker, to locate its own dynamic structure without having yet processed its relocation entries
- this method is especially important for the runtime linker, because it must initialize itself without relying on other programs to relocate its memory image.

https://docs.oracle.com/cd/E23824_01/html/819-0690/chapter6-74186.html

.dynamic section (3)

- essentially holds a number of arguments that inform on influence parts of the dynamic linker's behavior
- as a component of the run-time, the dynamic linker does many other things besides just relocate functions, it also executes other house keeping functions like INIT and FINI
- see elf/elf.h

http://blog.k3170makan.com/2018/11/introduction-to-elf-format-part-vii.html

.dynamic section (4)

- .dynamic section contains information
  that the dynamic linker uses
  to bind procedure addresses
  such as the symbol table and relocation information

`Computer Architecture : A Programmer's Perspective`

(1) three types of GOT entries

- the GOT contains *helper pointers* for both PLT fixups and GOT fixups
  - the first 3 entries are special and reserved
  - the next M entries belong to the PLT fixups
  - the next D entries belong to various data fixups
- the GOT is a synthetic area, createdy by the linker exists in both executables and libraries
- each library and executable gets its own PLT and GOT array

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

(2) the three special GOT entries

- the special 3 entries in the GOT
- GOT[0] : linked list pointer used by the dynamic linker
  address of .dynamic section

- GOT[1] : pointer to the reloc table for this module
  module identification info for the linker

- GOT[2] : pointer to the fixup / resolver code, located in ld-linux.so.2
  entry point in dynamic linker

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

(3) the PLT fixup GOT entries

- when the GOT is first set up,
  all the GOT entries related to PLT fixups
- GOT[n+3] are pointing back to PLT[n+1]+6
  which jump to PLT[0] to call the resolver

```
PLT[n+1]: jmp     *GOT[n+3]
          push    #n             ; push n as a argument to the resolver
          jmp     PLT[0]
```

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

(4) the PLT/data fixup GOT entries

- M entries belong to the PLT fixups

GOT[3]	indirect function call helpers
GOT[4]	indirect function call helpers
...	...
GOT[3+M-1]	indirect function call helpers,
	one per imported function

- D entries belong to various data fixups

GOT[3+M]	indirect pointers to global data references
GOT[3+M+1]	indirect pointers to global data references
...	...
GOT[end]	indirect pointers to global data references

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

Load address

- in a typical Linux system,
  the addresses `0 - 3fff_ffff (4 GB)`
  are available for the user program space.
- exectuable binary files include header information
  that indicates a load address
- libraries, because they are position-independent,
  do not need a load address, but contain a 0 in this field.

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

i386 load addresses 1999 (increasing from the top)

Start	Len	Usage
0000_0000	4k	zero page
0000_1000	128M	not used
0800_0000	896M	app code/data space followed by small-malloc() space
4000_0000	1G	mmap space library load space large-malloc() space
8000_0000	1G	stack space working back from BFFF.FFE0

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

i386 load addresses 1999 (increasing from the bottom)

Start	Len	Usage
8000_0000	1G	stack space working back from BFFF.FFE0
4000_0000	1G	memory mapped region for shared libraries large-malloc() space
0800_0000	896M	small-malloc() space app data / code space
0000_1000	128M	not used
0000_0000	4k	zero page

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

Linux Run-time Memory Image (increasing from the bottom)

0xc000_0000	Kernel virtual memory	memory invisible to the user code
	User stack created at run time	← %esp stack ptr
	↓ ↓ ↓ ↑ ↑ ↑	
0x4000_0000	memory mapped region for shared libraries	
	↑ ↑ ↑	
	Run time heap created by `malloc`	← `brk`
	R/W segment (`.data`, `.bss`)	
0x0804_8000	RO segment (`.init`, `.text`, `.rodata`)	

mmap (1)

- mmap (2) is a POSIX-compliant Unix system call that maps files or devices into memory.
- a method of memory-mapped file I/O
- implements demand paging,
  - file contents are not read from disk directly
  - initially do not use physical RAM at all.
- The actual reads from disk are performed in a lazy manner, after a specific location is accessed.

https://en.wikipedia.org/wiki/Mmap

mmap (2)

- ```
  #include <sys/mman.h>

  void *mmap(void *addr, size_t length, int prot, int flags,
             int fd, off_t offset);
  int munmap(void *addr, size_t length);
  ```
  - creates a new mapping in the *virtual address space* of the *calling process*
  - the starting address for the new mapping is specified in addr
  - the length argument specifies the length of the mapping
  - the contents of a file mapping are initialized using length bytes starting at offset offset in the file (or other object) referred to by the file descriptor fd

http://man7.org/linux/man-pages/man2/mmap.2.html
  ```

sys_brk (1)

- the sys_brk system call is provided by the kernel,
  to allocate memory without the need of moving it later
- allocates memory right behind the application image in the memory
- allows you to set the highest available address in the data section.
    - takes one parameter (the highest memory address)

https://www.tutorialspoint.com/assembly_programming/assembly_memory_management.htm

sys_brk (2)

- ```
  #include <unistd.h>

  int brk(void *addr);
  void *sbrk(intptr_t increment);
  ```
  - brk() and sbrk() change the location of the program break, which defines the end of the process's data segment
  - the program break is the first location after the end of the uninitialized data segment
  - increasing / decreasing the program break has the effect of allocating / deallocating memory to the process;
  - sbrk() increments the program's data space by increment bytes.

http://man7.org/linux/man-pages/man2/brk.2.html
  ```

Library load addresses (1)

- The kernel has a preferred location
  for mmap data objects at 0x4000_0000.
- since the shared libraries are loaded by mmap, they end up here.
- large mallocs are realized by creating a mmap, so
  these end up in the pool at 0x4000_0000.

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

Library load addresses (2)

- the library GLIBC that is mostly used for `malloc` handles small mallocs by calling `sys_brk()`, which extends the data area after the app, at `0x0800_0000+sizeof(app)`.
- As the mmap pool grows upward, the stack grows downward. between them, they share 2G bytes.

`http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html`

Shared library address

- The shared library design usually loads app first,
  then the loader notices that it need support
  and loads the dynamic loader library (using `.interp` section)
  (usually `/lib/ld-linux.so.2`)
  at `0x4000_0000`
- other libraries are loaded after `ld.so.1`
- see which and where libraries will be loaded by `ldd`
  `ldd foo_app`

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

Dynamic loader names

- dynamic loader
- dynamic linker
- runtime linker
- interpreter

- `ld-linux.so.2`
- `ld-linux.so`
- `ld.so`

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

load address example (1)

- There is a diagnostic case where the app is invoked by
  `/lib/ld-linux.so.2 foo_app foo_arg ....`
  - the `ld-linux.so.2` is loaded as an app
  - since it was built as a library, it tries to load at 0
  - [In ArmLinux, this is forbidden,
    so the kernel pushes it up to =0x1000=]
- Once `ld-linux.so.2` loads, it reads it `argv[1]` and loads the `foo_app` at its preferred location (`0x0800.0000`)
- other libraries are loaded up a the mmap area.

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

- So, in this case, the user memory map appears as

start	Len	Usage
0000_0000	128M	ld-linux.so.2 followed by small-malloc() space
0800_0000	896M	app code/data space
4000_0000	1G	mmap space lib space large-malloc() space
8000_0000	1G	stack space, working backward from BFFF_FFE0

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

- Notice that the small malloc space
  is much smaller in this case,
  but this is supposed to be
  for load testing and diagnostics
  so it's not too bad.

http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

ld vs. ld.so (1)

- ld is a static linker
  - a static library has the suffix name .a denoting archive
  - created by the ar utility
- ld.so is a dynamic linker
  - so represents shared object
  - a suffix name of shared libraries
  - libraries that may be dynamically linked into programs
  - one library is shared among several programs

https://unix.stackexchange.com/questions/356709/difference-between-ld-and-ld-so

ld vs. ld.so (2)

- A static linker links a program or library at compile time usually as the last step in the compilation process, creating a binary executable or a library.
- A dynamic linker loads the libraries
  - into the process' address space at run time.
  - that were dynamically linked at compile time

https://unix.stackexchange.com/questions/356709/difference-between-ld-and-ld-so

Binary executable file

- a statically linked binary
  with all libraries loaded into the executable itself
- a dynamically linked binary
  with only some libraries statically linked

https://unix.stackexchange.com/questions/356709/difference-between-ld-and-ld-so

statically linked vs dynamically linked (1)

- When you statically link a file into an executable,
  the contents of that file are included at link time.

- When you dynamically link a file into an executable,
  a pointer to the file is included in the executable
  but the contents of the file are not included at link time.

https://stackoverflow.com/questions/311882/what-do-statically-linked-and-dynamica

statically linked vs dynamically linked (2)

- these dynamically linked files are not bought in until you run the executable
- they are only bought into the in-memory copy of the executable, not the one on disk.
  - no files are modified on the disk
  - a shared library is shared across several processes
- statically linked files are 'locked' to the executable at link time so they never change
- A dynamically linked file referenced by an executable can change just by replacing the file on the disk.

https://stackoverflow.com/questions/311882/what-do-statically-linked-and-dynamica

library built with -fPIC

the vast majority of pages are exactly the same for every process

- processes a and b may
  load the library at different logical addresses,
  but they will point to the same physical pages:
  thus, the memory will be shared.
  the data in RAM exactly matches what is on disk,
  so it can be loaded only when needed by the page fault handler.

https://unix.stackexchange.com/questions/116327/loading-of-shared-libraries-and-ra

library built without -fPIC

- most pages of the library will need link edits, and will be different.
- they must be separate physical pages (as they contain different data).
- that means they're not shared.
- the pages don't match what is on disk
- the entire library could be loaded
  subsequently be swapped out to disk (in the swapfile)

https://stackoverflow.com/questions/311882/what-do-statically-linked-and-dynamical

shared library and re-entrant code (1)

- the concept of re-entrant code, i.e.,
  programs that cannotmodify themselves while running.
  it is necessary to write libraries.

- re-entrant code is useful for shared libraries
- Some functions in a library may be reentrant, whereas others in the same library are non-reentrant.

- A library is reentrant if and only if
  all of the functions in it are reentrant.

http://cs.boisestate.edu/~amit/teaching/297/notes/libraries-and-plugins-handout.pd
https://bytes.com/topic/c/answers/528112-basic-doubt-shared-libraries

shared library and re-entrant code (2)

- a shared library does not need to be reentrant
- the code area of the library is shared by multiple processes
- the data area of the library is copied separately for each process

- reentrant codes are required when running in multi-thread

http://cs.boisestate.edu/~amit/teaching/297/notes/libraries-and-plugins-handout.pc
https://bytes.com/topic/c/answers/528112-basic-doubt-shared-libraries

load time dynamic linking vs. run time dynamic linking

- load-time dynamic linking is
  when symbols in the library that are referenced
  by the executable (or another library) are handled
  when the executable / library is loaded into memory, by the os

- run-time dynamic linking is
  when you use an API provided by the OS
  or through a library to load a `.so` when you need it,
  and perform the symbol resolution then.

https://stackoverflow.com/questions/2055840/difference-between-load-time-dynamic-l

At the link time (1)

- `ld` is not called at either compile or run time
- only at the link step is `/usr/bin/ld` is invoked.
- a link step is performed as a final step in producing an executable, or a shared library
  - this is called static linking, to differentiate this step from dynamic loading that will happen at run time
- on Linux, `ld` is part of the binutils package.

https://stackoverflow.com/questions/52118756/is-ld-called-at-both-compile-time-anc

At the link time (2)

- The linker records
  - which shared libraries are required at the run time,
  - possibly which versions of libraries or symbols are required.
  - which run time loader should be used
- The kernel loads executable into memory, and checks whether the run time loader was requested at static link time.
- If it was, the dynamic loader is also loaded into memory, and execution control is passed to it (instead of the main executable).

https://stackoverflow.com/questions/52118756/is-ld-called-at-both-compile-time-anc

At the link time (3)

- then the dynamic loader is to examine the executable for instructions on which other libraries are required, check whether correct versions can be found, load them into memory, and prepare symbol resolution between the main executable and the shared libraries.
- This is the run time loading step, often also called dynamic linking
- The dynamic loader can be part of the OS, but on Linux it's part of libc (GLIBC, uClibc and musl each have their own loader).

https://stackoverflow.com/questions/52118756/is-ld-called-at-both-compile-time-anc

ld-linux.so vs. ld.so

- The programs `ld.so` and `ld-linux.so` find and load
  the shared libraries require by a program,
  prepare the program to run, and then run it.

- linux binaries require dynamic linking (linking at run time)
  unless the `-static` option was given to `ld(1)` during compilation.

- `ld.so` handles `a.out` binaries (a format used long ago)
- `ld-linux.so` handles ELF
  - `/lib/ld-linux.so.1` for libc5,
  - `/lib/ld-linux.so.2` for glibc2

https://linux.die.net/man/8/ld-linux

glibc

1. C library described in ANSI,c99,c11 standards.
   - It includes macros, symbols, function implementations etc.
   - (printf(), malloc() etc)
2. POSIX standard library.
   - he "userland" glue of system calls. (open(), read() etc.
   - no actual implementations of system calls. (kernel does it)
   - but glibc provides the user land interface to the services provided by kernel so that user application can use a system call just like a ordinary function.
3. Also some nonstandard but useful stuff.

`https://linux.die.net/man/8/ld-linux`

ld-linux.so (1)

- ld.so, ld-linux.so are linux's dynamic loader / linker
- most modern programs are dynamically linked
- when a dynamically linked application is loaded by the operating system, the dynamic loader must locate and load the dynamic libraries it needs for execution.
- on linux, that job is handled by ld-linux.so.2
- you can see the libraries used by a given application with the ldd command:

https://www.cs.virginia.edu/~dww4s/articles/ld_linux.html

ld-linux.so (2)

- The dynamic linker can be executed either indirectly by running some dynamically linked program or shared object
  - the dynamic linker is specified in the .interp section of an ELF file - the program to be executed
  - no command-line options to the dynamic linker
- executed directly by the command-line
  - /lib/ld-linux.so.* [OPTIONS] [PROGRAM [ARGUMENTS]]

man ld-linux.so

ld-linux.so (3)

- run time linker for dynamic objects
- a dynamic applications
  - consist of one or more dynamic objects
  - typically a dynamic executable and one or more shared object dependencies
- In Solaris, this interpreter is referred to as the run time linker
  - dynamic linker
  - dynamic loader

https://docs.oracle.com/cd/E19253-01/816-5165/ld.so.1-1/index.html

ld-linux.so (4)

- As part of the *initialization* and *execution* of a dynamic application, an interpreter is called
- this interpreter completes the binding of the application to its shared object dependencies.

https://docs.oracle.com/cd/E19253-01/816-5165/ld.so.1-1/index.html