

ELF1 7B Loading Background - ELF Study 1999

Young W. Lim

2020-10-22 Thr

- 1 Based on
- 2 Dynamic loading and dynamic linking
 - Dynamic loading
 - Dynamic linking
 - Possible Cases of loading and linking
- 3 Load addresses
 - TOC
 - Memory Map
 - Library load addresses
- 4 Executing dynamic executables
 - Entry point
 - Execution Sequence
 - Virtual memory
 - Memory mapped I/O

"Study of ELF loading and relocs", 1999

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

I, the copyright holder of this work, hereby publish it under the following licenses: GNU head Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled GNU Free Documentation License.

CC BY SA This file is licensed under the Creative Commons Attribution ShareAlike 3.0 Unported License. In short: you are free to share and make derivative works of the file under the conditions that you appropriately attribute it, and that you distribute it only under a license compatible with this one.

Compiling 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`

- dynamic loading

Dynamic loading (1)

- suppose our program that is to be executed consist of various modules.
- not all the modules are loaded into the memory at once
- the **main** module is loaded first and then starts to execute
- some other modules are loaded only when they are *required*
- until loading them, the execution is stopped

<https://cs.stackexchange.com/questions/92484/difference-between-dynamic-loading-and-static-loading>

Dynamic loading (2)

- Assume a linker is called to link necessary modules into an executable module.
- In dynamic loading, after the linker is called, only main module is loaded into memory.
- During execution, if main module needs another module which is already linked in executable module, then calling module calls **relocatable linking loader** to load the called module into appropriate location in the process's logical address space.

<https://cs.stackexchange.com/questions/92484/difference-between-dynamic-loading-and>

Dynamic loading (3)

- loading the dependent library or routine *on-demand* or at some time at **run time** after **load time** (the time at which the main program executable is loaded).
- this is contrast to loading all dependencies with the main program. at **load-time** together
- The loading process completes when the library has been successfully loaded into main memory.

<https://cs.stackexchange.com/questions/92484/difference-between-dynamic-loading-and-static-loading>

Dynamic loading (4)

- loading the library (or any other binary executable) into the memory during **load** or **run** time.
- **dynamic loading** can be imagined to be similar to plugins
 - an executable (main module) can actually start to run **before** the **dynamic loading** happens
- The **dynamic loading** example can be created using `dlopen()` of **Dynamically Loaded (DL) libraries**

<https://stackoverflow.com/questions/10052464/difference-between-dynamic-loading-and>

Dynamic loading (5)

- Dynamic loading :
system library or other routine
is loaded during **run time** and
it is not supported by **OS**
- when your program runs, it's the programmer's job
to open that library.
such programs are usually linked with **libdl**,
which provides the ability to open a shared library.

<https://stackoverflow.com/questions/10052464/difference-between-dynamic-loading-and>

Dynamic loading (6)

- dynamic loading allows a computer program
 - to start up without loading these libraries,
 - to discover and load available libraries after starting
- a computer program can, at **run time**,
 - load a library or other binary into memory,
 - retrieve the addresses of library functions and variables
 - execute those functions or access those variables, and
 - unload the library from memory.
- the 3 mechanisms by which
 - dynamic loading
 - static linking
 - dynamic linking.

<https://stackoverflow.com/questions/10052464/difference-between-dynamic-loading-and-static-linking>

Dynamic loading (7)

- With dynamic loading a module is not loaded until it is called
 - all modules are kept on a disk in a relocatable load format.
 - the main program is loaded into memory and is executed
- when a module needs to call another module, the calling module first checks to see whether it has been loaded.
 - if not , the **relocatable linking loader** is called to load the desired module into memory and update program's address tables to reflect this change.
 - then control is passed to newly loaded module

<https://stackoverflow.com/questions/10052464/difference-between-dynamic-loading-a>

Dynamic loading (8)

- an unused module is never loaded .
 - useful when the code is large
- dynamic loading does not need special support from OS
 - it is the responsibility of a programmer

<https://stackoverflow.com/questions/10052464/difference-between-dynamic-loading-and-static-loading>

- dynamic linking

Dynamic linking (1)

- suppose a program has some function calls whose definition is located in some system library
- the header file only consists of the declarations of functions and not definitions
- during execution, if the function gets called
 - the system library is loaded into main memory
 - **link** the function call in the program with the function definition in the system library.

<https://cs.stackexchange.com/questions/92484/difference-between-dynamic-loading-and>

Dynamic linking (2)

- when a module needs to be called,
 - the called module is loaded into memory and
 - a **link** between the calling module and called module is established by the **stub** (a piece of code that is linked) in **static linking time** of the program.
 - **stub** is a piece of code that is linked
 - a temporary small function placed by the **compiler**
 - makes an indirect call to a module function
- **dynamic Linking** mostly used with shared libraries which different users may use.

<https://cs.stackexchange.com/questions/92484/difference-between-dynamic-loading-and>

Dynamic linking (3)

- When the program makes the first call to an imported function whose library may or may not have been loaded yet.
 - Initially, a **stub** gets called instead of the imported function
 - the **stub** calls into the **OS**.
 - if the library is currently not loaded, it gets loaded (this step is called **dynamic loading**).
 - then, the **stub** is modified so that it calls the imported function directly for subsequent calls (this step is called **dynamic linking**)
- The component of the **OS** that performs both steps is called the **dynamic linker** or the **dynamic linking loader**.

<https://cs.stackexchange.com/questions/92484/difference-between-dynamic-loading-and-dynamic-linking>

Dynamic linking (4)

- **dynamic linking** is done during **load** or **run** time and not when the executable is created (**compile** time)
- the **static linker** does minimal work when creating the executable (generating **stub** functions)
- the **dynamic linker** has to load the libraries so it is called **linking loader**.

<https://stackoverflow.com/questions/10052464/difference-between-dynamic-loading-a>

Dynamic linking (5)

- system library or other routine is linked during **run time** and by the support of **OS**
- when an executable is **compiled** the required shared libraries must be specified otherwise it won't even compile.
- When your program starts it's the **system**'s job to open these libraries
- the required libraries can be listed using the `ldd` command.

<https://stackoverflow.com/questions/10052464/difference-between-dynamic-loading-and>

Dynamic linking (6)

- Dynamic linker is a **run time** program that loads and binds all of the dynamic dependencies of a program before starting to execute that program.
 - find what dynamic libraries a program requires, what libraries those libraries require ... (dynamic dependencies)
 - load all those libraries and make all references to the functions point to the right places
- the "hello world" program requires the standard C library
 - the **dynamic linker** will load the standard C library before loading the hello world program and will make any calls to `printf()` go to the right place

<https://stackoverflow.com/questions/10052464/difference-between-dynamic-loading-and>

Dynamic linking (7)

- both **dynamic loading** and **dynamic linking** happen at **run time**, and load whatever they need into memory.
- The key difference is that
 - **dynamic loading** checks if the routine was loaded by the loader
 - **dynamic linking** checks if the routine is in the memory.

<https://stackoverflow.com/questions/10052464/difference-between-dynamic-loading-and-dynamic-linking>

Dynamic linking (8)

- for **dynamic linking**, there is only one copy of the library code in the memory,
 - this may be not true for **dynamic loading**
 - That's why dynamic linking needs **OS support** to check the memory of other processes.
- this feature is very important for language libraries, which are shared by many programs.

<https://stackoverflow.com/questions/10052464/difference-between-dynamic-loading-and-dynamic-linking>

Dynamic loading and dynamic linking

- **dynamic loading** refers to mapping (or less often copying) an executable or library into a process's memory after the executable has been started.
- **dynamic linking** refers to resolving symbols
 - associating their names with addresses or offsets
 - after **compile time**
- the reason it's hard to make a distinction is that the two are often done together without recognizing

<https://www.quora.com/Systems-Programming/What-is-the-exact-difference-between-dy>

(1) Dynamic loading, Static linking

- The executable has an address/offset table generated at **compile time**, but the actual code/data aren't loaded into memory at **process start**.
- old-fashioned **overlay** systems.
- some current **embedded** systems may work in this way
- to give the programmer control over memory use
- also to avoid the linking overhead at **runtime**

<https://www.quora.com/Systems-Programming/What-is-the-exact-difference-between-dy>

(2) Static loading, Dynamic linking

- when dynamic libraries specified at **compile time**
- an executable contains a reference to the dynamic/shared library, but the **symbol table** is missing or incomplete.
- both **loading** and **linking** occur at **process start**, which is considered as
 - **dynamic** for **linking**
 - **static** for **loading**.

<https://www.quora.com/Systems-Programming/What-is-the-exact-difference-between-dy>

(3) Dynamic loading, Dynamic linking

- when you call `dlopen`
- the object file is loaded dynamically under program control (i.e. after **process start**)
- symbols in the calling program and in the library are resolved based on the process's particular memory layout at that time.

<https://www.quora.com/Systems-Programming/What-is-the-exact-difference-between-dy>

(4) Static loading, Dstatic linking

- everything is resolved at **compile time**.
- everything is loaded into memory immediately at **process start**
- no further resolution (linking)
- does not require to load a single file
- but no known implementation for multiple file loading without dynamic linking

<https://www.quora.com/Systems-Programming/What-is-the-exact-difference-between-dy>

TOC: Load addresses

- Memory Map
- Library load addresses

TOC: Memory Map

- Load address
- i386 Load addresses 1999 (increasing from the top)
- i386 Load addresses 1999 (increasing from the bottom)
- Linux run-time memory image
- mmpa
- sys_brk

- in a typical Linux system, the addresses 0 - 3fff_ffff (4 GB) are available for the user program space.
- executable 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 memory mapped region for shared libraries
4000_0000	1G	large-malloc() space small-malloc() space
0800_0000	896M	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	← brk
	created by malloc	
	R/W segment	
	(.data, .bss)	
	RO segment	
0x0804_8000	(.init, .text, .rodata)	

- **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>

- 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

- `#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>

TOC: Library load addresses

- Library load addresses
- Shared library address
- Dyn loader names
- load address example

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

- 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)

- consider a diagnostic case where the app (`foo_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` is loaded, it reads its `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

load address example (2)

- 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

load address example (3)

- Notice that the small malloc space is much smaller in this case (128M),
but this is supposed to be for load testing and diagnostics

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

- the *vast majority* of **pages** are exactly the same for every process
- different processes 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-r>

library built without -fPIC

- *most* **pages** of the library will need **link edits**, and will be different
- each process has separate **physical pages** because they contain different data (as a result of execution)
- that means they're not shared.
- the **pages** don't match what is on **disk**
- in the worst case, the entire library could be loaded and then 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 cannot modify 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.pdf>
<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.pdf>

<https://bytes.com/topic/c/answers/528112-basic-doubt-shared-libraries>

- defines whether to use 32-bit or 64-bit addresses.
- contains three fields that are affected by this setting and offset other fields that follow them.
 - e_entry (entry point)
 - e_phoff (program header table offset)
 - e_shoff (section header table offset)
- The ELF header is 52 or 64 bytes long for 32-bit and 64-bit binaries respectively.

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

ELF header example

```
$ readelf -h /bin/bash
```

```
ELF Header:
```

```
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                               ELF32
  Data:                                   2's complement, little endian
  Version:                               1 (current)
  OS/ABI:                                UNIX - System V
  ABI Version:                           0
  Type:                                   EXEC (Executable file)
  Machine:                                Intel 80386
  Version:                                0x1
  Entry point address:                   0x805be30
  Start of program headers:              52 (bytes into file)
  Start of section headers:              675344 (bytes into file)
  Flags:                                  0x0
  Size of this header:                    52
  Size of program headers:                32
  Number of program headers:              8
  Size of section headers:                40
  Number of section headers:              26
  Section header string table index:     25
```

```
https://greek0.net/elf.html
```

ELF header fields

```
typedef struct {
    unsigned char    e_ident[EI_NIDENT];
    Elf32_Half       e_type;
    Elf32_Half       e_machine;
    Elf32_Word       e_version;
    Elf32_Addr       e_entry;
    Elf32_Off        e_phoff;
    Elf32_Off        e_shoff;
    Elf32_Word       e_flags;
    Elf32_Half       e_ehsize;
    Elf32_Half       e_phentsize;
    Elf32_Half       e_phnum;
    Elf32_Half       e_shentsize;
    Elf32_Half       e_shnum;
    Elf32_Half       e_shstrndx;
} Elf32_Ehdr;
// 52 bytes for 32-bit machines
```

```
typedef struct {
    unsigned char    e_ident[EI_NIDENT];
    Elf64_Half       e_type;
    Elf64_Half       e_machine;
    Elf64_Word       e_version;
    Elf64_Addr       e_entry;
    Elf64_Off        e_phoff;
    Elf64_Off        e_shoff;
    Elf64_Word       e_flags;
    Elf64_Half       e_ehsize;
    Elf64_Half       e_phentsize;
    Elf64_Half       e_phnum;
    Elf64_Half       e_shentsize;
    Elf64_Half       e_shnum;
    Elf64_Half       e_shstrndx;
} Elf64_Ehdr;
// 64 bytes for 64-bit machines
```

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

ELF header - e_ident field

0x00	4	e_ident[EI_MAG0] through e_ident[EI_MAG3]
0x04	1	e_ident[EI_CLASS]
0x05	1	e_ident[EI_DATA]
0x06	1	e_ident[EI_VERSION]
0x07	1	e_ident[EI_OSABI]
0x08	1	e_ident[EI_ABIVERSION]
0x09	7	e_ident[EI_PAD]

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

ELF header - e_entry field

- This is the memory address of the **entry point** from where the process starts executing. This field is either 32 or 64 bits long depending on the format defined earlier.

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

- an **entry point** is where control is transferred from the operating system to a computer program, at which place the processor enters a program or a code fragment and execution begins.
- This marks the transition from **load time** (and **dynamic link time**, if present) to **run time**

<https://reverseengineering.stackexchange.com/questions/18088/start-analysis-at-any>

entry (2)

- 1 In some operating systems or programming languages, the initial entry is not part of the program but of the **runtime library**
 - the **runtime library** initializes the program
 - then the **runtime library** enters the program.
- 2 In other cases, the program may call the **runtime library** before doing anything when it is entered for the first time,
 - after the **&runtime library** returns, the actual code of the program begins to execute.

<https://reverseengineering.stackexchange.com/questions/18088/start-analysis-at-any>

- entry point is used to point at the location at which the **OS loader** will start a program
 - for a given binary file (ELFBIN), use `readelf -h ELFBIN` to read the binary's header information (-h):
 - Entry point address: `0x400a80`
 - after running `objdump` on the binary
 - `0000000000400a80 <_start>`:

<https://reverseengineering.stackexchange.com/questions/18088/start-analysis-at-any>

entry (4)

- it is the `_start` function that prepares certain parameters/registers before eventually calling `main`
 - `400aa4: callq *0x20851e(%rip)` contains a program code.
 - the `_start` function is usually called after all other `sections` of the binary have been loaded in memory.
- after the `main` is done, the `hlt` instruction is executed to terminate the execution in this example.
 - the `hlt` instruction is typically never reached since `__libc_start_main` calls `exit(2)` if `main` returns normal

<https://reverseengineering.stackexchange.com/questions/18088/start-analysis-at-any>

- with gcc's `-g` flag, an executable contains debugging information.
 - for each instruction there is information which line of the source code generated it, the name of the variables in the source code is retained and can be associated to the matching memory at runtime etc.
- **strip** can remove this debugging information and other data included in the executable which is not necessary for execution in order to reduce the size of the executable.

<https://unix.stackexchange.com/questions/2969/what-are-stripped-and-not-stripped->

- gcc being a compiler/linker, its `-s` option is something done while **linking**
- it's not configurable
 - it has a set of information which it removes, no more no less.
- removes the relocation information and the symbol table which is not done by **strip**
 - Note that, removing relocation information would have some effect on address space layout randomization

<https://stackoverflow.com/questions/1349166/what-is-the-difference-between-gcc-s->

- `strip` can be run on an object file which is already compiled.
- has a variety of command-line options to configure which information will be removed.
- For example, `-g` strips only the debug information
- Note that `strip` is not a bash command, though you may be running it from a bash shell.
- It is a command totally separate from bash, part of the GNU binary utilities suite.

<https://stackoverflow.com/questions/1349166/what-is-the-difference-between-gcc-s-a>

finding main function's entry point (1)

- once a program has been stripped, there is no straightforward way to locate the function that the symbol **main** would have otherwise referenced.
- The value of the symbol **main** is not required for program **start-up**:

<https://stackoverflow.com/questions/9885545/how-to-find-the-main-functions-entry->

finding main function's entry point (2)

- in the ELF format, the start of the program is specified by the `e_entry` field of the ELF file header.
- This field normally points to the `C library`'s initialization code, and not directly to `main`.
- While the `C library`'s initialization code does call `main()` after it has set up the `C run time environment`, this call is a normal function call that gets fully resolved at link time

<https://stackoverflow.com/questions/9885545/how-to-find-the-main-functions-entry->

Execution sequence

- 1 **Kernel** does permission checks
- 2 **Kernel** attempts to determine the internal format.
It finds out it's ELF and that it's dynamically linked.
- 3 **Kernel** decodes the structure of the ELF executable,
finding the interpreter (`ld-linux.so.2` or something).
It attempts to load the interpreter,
which itself is a statically linked ELF executable.
- 4 The **interpreter**, in user space, looks for and loads
the shared object files (extension `.so`, internal format ELF)
which are needed by the executable.
Once they are all loaded and relocated, control is passed
to the executable itself, at the **entry point** established.

<https://www.quora.com/How-is-a-elf-file-executed-in-Linux>

Manual load and execution (1)

- 1 Read the **program headers**
 - to find the **LOAD** directives and
 - determine the total length of **mappings** in pages.
- 2 Map the lowest-address **LOAD** directive with the total length (which may be greater than the file length), letting **mmap** assign you an address. This will reserve contiguous **virtual address** space.
- 3 map the remainin **LOAD** directives over top of parts of this mapping using **MAP_FIXED**.

<https://stackoverflow.com/questions/6554825/how-do-i-load-and-execute-an-elf-binary>

Manual load and execution (2)

- 4 Use the **program headers** to find the **DYNAMIC vector**, which will in turn give you the address of the relocation vectors
- 5 Apply the relocations
Assuming your binary was a *static-linked PIE* binary, they should consist entirely of *RELATIVE* relocations (just adding the base load address), meaning you don't have to perform any symbol lookups or anything fancy.

<https://stackoverflow.com/questions/6554825/how-do-i-load-and-execute-an-elf-binary>

Manual load and execution (3)

- Construct an **ELF program entry stack** consisting of the following sequence of system-word-sized values in an array on the stack:

```
ARGC ARGV[0] ARGV[1] ... ARGV[ARGC-1] 0 \
```

```
ENVIRON[0] ENVIRON[1] ... ENVIRON[N] 0 0
```

- (This step requires ASM!)
Point the stack pointer at the beginning of this array and jump to the loaded program's **entry point** address (which can be found in the program headers).

<https://stackoverflow.com/questions/6554825/how-do-i-load-and-execute-an-elf-binary>

ELF program header (1)

- The **program header table** tells the system how to create a process image
- it is found at file offset `e_phoff` and consists of `e_phnum` entries each with size `e_phentsize`
- The layout is slightly different in 32-bit ELF vs 64-bit ELF, because the `p_flags` are in a different structure location for alignment reasons.

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

ELF program header (2)

- The **Program Header Table** contains information for the kernel on how to start the program.
- the LOAD directives specifies a loadable segment parts of the ELF file get mapped into memory
- The INTERP directive specifies an ELF interpreter normally `/lib/ld-linux.so.2`
- The DYNAMIC entry points to the `.dynamic` section contains information used by the ELF interpreter to setup the binary

<https://www.ics.uci.edu/~aburtsev/143A/hw/hw2/hw2-elf.html>

ELF program header example

Program Headers:

Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
PHDR	0x000034	0x08048034	0x08048034	0x00100	0x00100	R E	0x4
INTERP	0x000134	0x08048134	0x08048134	0x00013	0x00013	R	0x1
[Requesting program interpreter: /lib/ld-linux.so.2]							
LOAD	0x000000	0x08048000	0x08048000	0xa0200	0xa0200	R E	0x1000
LOAD	0x0a0200	0x080e9200	0x080e9200	0x04b44	0x09728	RW	0x1000
DYNAMIC	0x0a0214	0x080e9214	0x080e9214	0x000d8	0x000d8	RW	0x4
NOTE	0x000148	0x08048148	0x08048148	0x00020	0x00020	R	0x4
GNU_EH_FRAME	0x0a0138	0x080e8138	0x080e8138	0x0002c	0x0002c	R	0x4
GNU_STACK	0x000000	0x00000000	0x00000000	0x00000	0x00000	RW	0x4

<https://greek0.net/elf.html>

ELF program header fields (1)

```
typedef struct {
    Elf32_Word    p_type;
    Elf32_Off     p_offset;
    Elf32_Addr    p_vaddr;
    Elf32_Addr    p_paddr;
    Elf32_Word    p_filesz;
    Elf32_Word    p_memsz;
    Elf32_Word    p_flags;
    Elf32_Word    p_align;
} Elf32_Phdr;
// 52 bytes for 32-bit machines
```

```
typedef struct {
    Elf64_Word    p_type;
    Elf64_Word    p_flags;
    Elf64_Off     p_offset;
    Elf64_Addr    p_vaddr;
    Elf64_Addr    p_paddr;
    Elf64_Xword   p_filesz;
    Elf64_Xword   p_memsz;
    Elf64_Xword   p_align;
} Elf64_Phdr;
// 64 bytes for 64-bit machines
```

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

ELF program header fields (2)

- **p_type** : the kind of segment this array element describes or how to interpret the array element's information.
- **p_offset** : the offset from the beginning of the file at which the first byte of the segment resides

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

ELF program header fields (3)

- **p_vaddr** : the virtual address at which the first byte of the segment resides in memory.
- **p_paddr** : the segment's physical address for systems in which physical addressing is relevant
 - the system ignores physical addressing for application programs,
 - this member has unspecified contents for executable files and shared objects

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

ELF program header fields (4)

- **p_filesz** : the number of bytes in the file image of the segment, which can be zero.
- **p_memsz** : the number of bytes in the memory image of the segment, which can be zero.
- **p_flags** : flags relevant to the segment.

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

ELF program header fields (5)

- `p_align` : loadable process segments must have congruent values for `p_vaddr` and `p_offset`, modulo the page size.
 - this member gives the value to which the segments are aligned in memory and in the file
 - values 0 and 1 mean no alignment is required.
 - otherwise, `p_align` should be a positive, integral power of 2,
 - `p_vaddr` should equal `p_offset`, modulo `p_align`

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

ELF program header field p_type (1)

PT_NULL	0	unused
PT_LOAD	1	a loadable segment
PT_DYNAMIC	2	dynamic linking information
PT_INTERP	3	an interpreter path name
PT_NOTE	4	auxiliary information
PT_SHLIB	5	unspecified semantics
PT_PHDR	6	the program header table
PT_LOSUNW	0x6ffffffa	sun microsystems
PT_SUNWBSS	0x6ffffffb	sun microsystems
PT_SUNWSTACK	0x6ffffffa	sun microsystems
PT_HISUNW	0x6fffffff	sun microsystems
PT_LOPROC	0x70000000	a processor specific semantics
PT_HIPROC	0x7fffffff	a processor specific semantics

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

p_type = PT_LOAD segment entry

- specifies a **loadable segment**, described by
 - p_filesz (the segment's file size) and
 - p_memsz (the segment's memory size)
- The bytes from the file are mapped to the beginning of the **memory segment**
 - case 1) $p_memsz > p_filesz$, the extra bytes are defined to hold the value 0 and to follow the segment's initialized area
 - case 2) $p_memsz < p_filesz$: not possible
- **loadable segment** entries in the **program header table** appear in ascending order, sorted on the p_vaddr member.

<https://docs.oracle.com/cd/E19683-01/816-1386/chapter6-83432/index.html>

- specifies dynamic linking information

<https://docs.oracle.com/cd/E19683-01/816-1386/chapter6-83432/index.html>

p_type = PT_INTERNP segment entry

- specifies the location and size of a null-terminated path name to invoke as an **interpreter**
- this segment type is mandatory for dynamic executable files and can occur in shared objects.
but cannot occur more than once in a file.
- this type, if present,
it must precede any loadable segment entry.

<https://docs.oracle.com/cd/E19683-01/816-1386/chapter6-83432/index.html>

Sections and segments (1)

- **section**: tell the **linker** if a section is either:
 - raw data to be loaded into memory,
e.g. `.data`, `.text`, etc, or
 - formatted meta data about other sections,
that will be used by the linker, but disappear at runtime
e.g. `.symtab`, `.srctab`, `.rela.text`
- **segment**: tells the **operating system**:
 - where should a segment be loaded into *virtual memory*
 - what *permissions* the segments have (read, write, execute).

<https://cirosantilli.com/elf-hello-world>

<https://stackoverflow.com/questions/14361248/whats-the-difference-of-section-and-segment>

Sections and segments (2)

- ELF files are composed of **sections** and **segments**
- **sections** gather all needed information to link a given object file and build an executable,
- while Program Headers split the executable into **segments** with different attributes, which will eventually be loaded into memory.

<https://www.intezer.com/blog/research/executable-linkable-format-101-part1-section>

Sections and segments (3)

- **segments** can be viewed as a tool to help the **linux loader**, as they group sections by attributes into single segments for more efficient loading process of the executable, instead of loading each individual section into memory.

<https://www.intezer.com/blog/research/executable-linkable-format-101-part1-section>

Sections and segments (4)

- segments' offsets and virtual addresses must be congruent modulo the page size
- their `p_align` field must be a *multiple* of the system page size
- The reason for this alignment is to prevent the mapping of two different segments within a single memory page.

<https://www.intezer.com/blog/research/executable-linkable-format-101-part1-section>

Sections and segments (5)

- this is due to the fact that different segments usually have different access attributes,
- these cannot be enforced if two segments are mapped within the same memory page.
- therefore, the default segment alignment for PT_LOAD segments is usually a system page size
- The value of this alignment will vary in different architecture

<https://www.intezer.com/blog/research/executable-linkable-format-101-part1-section>

Virtual address and physical address (1)

- **Physical addresses** are provided directly by the machine
 - one physical address space per machine
 - addresses typically range from some minimum (sometimes 0) to some maximum,
 - though some portions of this range are usually used by the OS and/or devices, but not available for user processes

<https://www.student.cs.uwaterloo.ca/~cs350/F07/notes/mem.pdf>

Virtual address and physical address (2)

- **Virtual addresses** (or logical addresses) are addresses provided by the OS
 - one virtual address space per process
 - addresses typically start at zero, but not necessarily
 - space may consist of several segments

<https://www.student.cs.uwaterloo.ca/~cs350/F07/notes/mem.pdf>

Virtual address and physical address (3)

- **address translation** (or **address binding**) means mapping **virtual addresses** to **physical addresses**

<https://www.student.cs.uwaterloo.ca/~cs350/F07/notes/mem.pdf>

Virtual address and physical address (4)

- size of each section except stack is specified in ELF file
- sections which are initialized from the ELF file
 - code (i.e., `.text`)
 - read-only data
 - initialized data segments
- other remaining sections are initially zero-filled
- sections have their own specified alignment
- segments are page aligned
- 3 segments = (`.text` + `.rodata`), (`.data` + `.sbss` + `.bss`), (stack)
- not all programs contain this many segments and sections

<https://www.student.cs.uwaterloo.ca/~cs350/F07/notes/mem.pdf>

Single address space (1)

- simple systems
- sharing the same memory space
 - memory and peripherals
 - all processes and OS
- no memory protection

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

Single address space (2)

- CPUs with single address space
 - 8086 - 80286
 - ARM Cortex-M
 - 8 / 16-bit PIC
 - AVR
 - most 8- and 16-bit systems

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

Single address space (3)

- portable c programs expect flat memory
 - multiple memory access methods limit portability
- management is tricky
 - need to know / detect total RAM
 - need to keep processes separated
- no protection

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

Virtual memory (1)

- a system that uses an address mapping
- maps virtual address space to physical address space
 - to physical RAM
 - to hardware devices
 - PCI devices
 - GPU RAM
 - On-SOC IP blocks

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

Virtual memory (2)

- Advantages

- each process can have a different memory mapping
 - one process' RAM is invisible to other processes
 - built in memory protection
 - kernel RAM is invisible to user space processes
- memory can be moved
- memory can be swapped to disk

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

Virtual memory (3)

- Advantages (continued)
 - hardware device memory can be mapped into process' address space
requires the kernel to perform the mapping
 - physical RAM can be mapped into multiple processes at once
shared memory
 - memory regions can have access permissions
read / write / execute

https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf

Memory-mapped I/O (1)

- On modern operating systems, it is possible to **mmap** a file to a region of memory then, the file can be accessed just like an array
- This is more efficient than **read** or **write**, as only the regions of the file that a program actually accesses are loaded.

https://www.gnu.org/software/libc/manual/html_node/Memory_002dmapped-I_002f0.html

Memory-mapped I/O (2)

- accesses to not-yet-loaded parts of the mmapped region are handled in the same way as *swapped out pages*.
- since mmapped pages can be *stored back* to their file when physical memory is low, it is possible to **mmap** files orders of magnitude larger than both the physical memory and swap space

https://www.gnu.org/software/libc/manual/html_node/Memory_002dmapped-I_002f0.html

Memory-mapped I/O (3)

- The only limit is address space.
- the theoretical limit is 4GB on a 32-bit machine -
- the actual limit will be smaller since some areas will be reserved for other purposes.
- If the LFS (Large File Storage) interface is used
 - the file size on 32-bit systems is not limited to 2GB
 - offsets are signed which reduces the addressable area of 4GB by half
 - the full 64-bit are available.

https://www.gnu.org/software/libc/manual/html_node/Memory_002dmapped-I_002f0.html

Memory-mapped I/O (4)

- `mmap` 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 and 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>

Memory-mapped I/O (5)

- after the memory is no longer needed, it is important to `munmap` the pointers to it.
- protection information can be managed using `mprotect`
- special treatment can be enforced using `madvise`

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

Memory-mapped I/O (6)

- **demand paging** is a method of virtual memory management (as opposed to anticipatory paging)
- the os copies a disk page into physical memory only if an attempt is made to **access** it and that page is not already in memory (**page fault**)

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

Memory-mapped I/O (7)

- it follows that a process begins execution with none of its pages in physical memory, and many **page faults** will occur until most of a process's **working set** of pages are located in physical memory.
- this is an example of a **lazy loading** technique.

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

Memory-mapped I/O (8)

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset)
```

- `mmap()` 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 (which must be greater than 0).

<https://man7.org/linux/man-pages/man2/mmap.2.html>

Memory-mapped I/O (9)

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset)
```

- If `addr` is `NULL`, then the kernel chooses the (page-aligned) address at which to create the mapping; this is the most portable method of creating a new mapping.
- If `addr` is not `NULL`, then the kernel takes it as a *hint* about where to place the mapping;

<https://man7.org/linux/man-pages/man2/mmap.2.html>

Memory-mapped I/O (10)

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset)
```

- on Linux, the **kernel** will pick a nearby page boundary but always above or equal to the value specified by `/proc/sys/vm/mmap_min_addr` and attempt to create the mapping there.
- If another mapping already exists there, the kernel picks a new address that may or may not depend on the *hint*
- The address of the new mapping is returned as the result of the call.

<https://man7.org/linux/man-pages/man2/mmap.2.html>

Memory-mapped I/O (11)

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset)
```

- The contents of a **file mapping** (as opposed to an anonymous mapping), are initialized using `length` bytes starting at `offset` in the file (or other object) referred to by the file descriptor `fd`
- `offset` must be a multiple of the page size as returned by `sysconf(_SC_PAGE_SIZE)`.

<https://man7.org/linux/man-pages/man2/mmap.2.html>

Memory-mapped I/O (12)

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset)
```

- After the `mmap()` call has returned, the file descriptor, `fd`, can be closed immediately without invalidating the mapping.

<https://man7.org/linux/man-pages/man2/mmap.2.html>

Memory-mapped I/O (13)

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset)
```

- The `prot` argument describes the desired memory protection of the mapping and must not conflict with the open mode of the file
- It is either `PROT_NONE` or the bitwise OR of one or more of the following flags:
 - `PROT_EXEC` Pages may be executed.
 - `PROT_READ` Pages may be read.
 - `PROT_WRITE` Pages may be written.
 - `PROT_NONE` Pages may not be accessed.

<https://man7.org/linux/man-pages/man2/mmap.2.html>