ELF1 7A Linking Background - ELF Study 1999

Young W. Lim

2020-07-08 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`

TOC: Types of linking

TOC: Static and dynamic linking

- Binary executable files
- Statically linked files
- Dynamically linked files
- In-memory copy of an executable

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 files

- when you statically link a file into an executable,
  the contents of the files are included
  in the executable at link time.
- statically linked executable and library files
  never change
  (the last step in the compilation prcess)

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

Dynamically linked files

- 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.
- these referenced dynamically linked files are
  - not brought in the memory until you run the executable
  - loaded into memory by the dynamic linker at run time

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

In-memory copy of an executable

- dynamically linked files are only brought into the in-memory copy of the executable, not the executable file on the disk.
  - files on the disk are not modified
  - a shared library is shared across several processes
- dynamically loaded libraries can change at the next run time just by replacing the corresponding files on the disk.

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

TOC: Build-time, load-time, run-time linking

Build-time, load-time, run-time

compile step	link step	run step	run step
build-time	build-time	load-time	run-time

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

Build-time and. load-time linkers

build-time linking	build-time linking	load-time linking
static linking	static linking	dynamic linking
ld	ld	ld.so
for statically linked exectuables	for dynamically linked executables	for dynamically linked executables
or static libraries	or shared libraries	or shared libraries

https://unix.stackexchange.com/questions/449107/what-differences-and-relations-are

(1) build-time linking for static executables / libraries

- static linking, at build-time
  the build-time linker ld
  - resolves all the objects used in the program
    during the build,
  - merges the objects which are used, and
  - produces an executable binary
    which doesn't use external libraries;

https://unix.stackexchange.com/questions/449107/what-differences-and-relations-are

(2) build-time linking for dynamic executables / libraries

- static linking, at build-time:
  the build-time linker `ld`
  - resolves all objects used in the program, but
  - it only stores *references* to them;
  - instead of storing them in the executable (no merge)
  - 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

https://unix.stackexchange.com/questions/449107/what-differences-and-relations-are
https://stackoverflow.com/questions/52118756/is-ld-called-at-both-compile-time-and

(3) load-time linking for dynamic executables / libraries

- dynamic linking, at run-time (specifically load-time) :
  the run-time linker `ld.so`, or dynamic linker,
  - resolves all the references stored in the executable,
  - loading all the required libraries (shared objects) and
  - updating all the object references before running the program.

https://unix.stackexchange.com/questions/449107/what-differences-and-relations-are

Load-time vs. run-time dynamic linking

- load-time dynamic linking
  the OS handles unresolved symbols in the library
    - referenced by the executable (or another library)
    - resolved when the executable/library is loaded into memory

- run-time dynamic linking
  an API provided by the OS or through a library
    - can explicitly load a DLL or DSO when you need it
    - and then perform the symbol resolution

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

Run-time dynamic linking

- using libdl

dlopen()	gain access to an executable object file
dclose()	close a dlopen object
dlsym()	obtain the address of a symbol from a dlopen object
dlvsym()	Programming interface to dynamic linking loader.
dlerror()	get diagnostic information

http://www.yolinux.com/TUTORIALS/LibraryArchives-StaticAndDynamic.html

Build-time linker ld

- a static linker
  - links a program or library at compile (build) time
  - usually as the last step in the compilation process, creating a binary executable or a library.
- a static library
  - has the suffix name .a denoting archive
  - is created by the ar utility
- ld is a static linker (build-time linker)
- ld also plays a role in dynamic linking (build-time linker)
  - stores all object references in a dynamic executable

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

Run-time linker `ld.so`

- a dynamic linker
  - loads the dynamic libraries
    into the process' address space at run time.
  - libraries were dynamically linked at compile (build) time

- a dynamic library
  - so represents shared object
  - the suffix name of shared libraries
  - a library that may be dynamically linked into programs
  - one library is shared among several programs

- `ld.so` is a dynamic linker (run-time linker)

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

Linker at the build time

- compile, link, run
  - `ld` is not called at either compile or run time
  - only at the link step is `/usr/bin/ld` is invoked.
  - on Linux, `ld` is part of the binutils package.

- a link step is performed as a final step
  in producing an executable, or a shared library (build time)
  - this is called static linking, to differentiate this step
    from dynamic loading that will happen at run time
    (specifically load time)

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

kernel at the load time

- The kernel
  - loads executable into memory, and
  - checks whether runtime 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

dynamic loader at the load time

- the dynamic loader
  - examines the executable
    - which other libraries are required
    - whether correct versions can be found,
  - loads them into memory, and
  - performs symbol resolution
    between the main executable and the shared libraries
  - this is the runtime loading step,
    often also called dynamic linking
  - on Linux, dynamic loader is a 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

TOC: Dynamic loading

- 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-a

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 apporiate location in the processes logical adress space.

https://cs.stackexchange.com/questions/92484/difference-between-dynamic-loading-a

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-a

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-ar

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-ar

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-an

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-ar

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-an

TOC: Dynamic linking

- dynamic linking

Dynamic linking (1)

- suppose our 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-an

Dynamic linking (2)

- when a module needs to be called,
  that module is loaded into memory and
  a link between the calling module and called module
  is established by the stub
  which is a piece of code that is linked
  in linking time of the program.
- dynamic Linking mostly used with
  shared libraries which different users may use.

https://cs.stackexchange.com/questions/92484/difference-between-dynamic-loading-a

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, the compiler places a temporary small function, called a stub, that 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 process 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-a

Dynamic linking (4)

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

https://stackoverflow.com/questions/10052464/difference-between-dynamic-loading-ar

- Dynamic linking :
  system library or other routine
  is linked during runtime and
  it is supported by OS
- when compiling your executable you must
  specify the shared library your program uses,
  otherwise it won't even compile.
- When your program starts it's the system's job
  to open these libraries,
  which can be listed using the ldd command.

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

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 (and so on)
  - load all those libraries and make all references to functions point to the right places
- the "hello world" program requires the C library
  - the dynamic linker will load the 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-a

Dynamic linking (7)

- both dynamic loading and linking happen at runtime, 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-ar

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 subroutine libraries,
  which are shared by many programs.

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

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, Dtatic 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: Linking for dynamic executables / libraries

- Build-time linking for dynamic executables / libraries
- Load-time linking for dynamic executables / libraries

TOC: Build-time linking for dynamic executables / libraries

Unresolved symbols

- unresolved symbols in a dynamic execuble
  - should be resolved
- unresolved symbols in a shared library
  - remain valid

https://stackoverflow.com/questions/19736853/what-does-ld-do-when-linking-against-

Referenced libraries

- ld stores the needed library
  in a DT_NEEDED record of
  the _DYNAMIC object of the output file
  - When the application starts, the dynamic linker
    looks at the DT_NEEDED field to find the required libraries.
    This field contains the soname of the library,
    so the next step is for the dynamic linker
    to walk through all the libraries in its search path
    looking for it.

http://bottomupcs.sourceforge.net/csbu/x4012.htm
https://stackoverflow.com/questions/19736853/what-does-ld-do-when-linking-against-

Copy relocation and symbol table

- If the output is not position-independent and references *data* objects in the shared library,
  - generate a copy relocation
    to copy the original image of the object
    into the main program's data segment at load time,
  - create a proper symbol table entry
    so that references to the object in the shared library itself
    get resolved to the new copy in the main program,
    rather than the original copy in the library.

https://stackoverflow.com/questions/19736853/what-does-ld-do-when-linking-against-

PLT thunks

- generating PLT thunks
  for the destination of each function call in the output
  - remain unresolved at build-time

https://stackoverflow.com/questions/19736853/what-does-ld-do-when-linking-against-

Dynamic symbol table

- creating a dynamic symbol table,
  - the runtime linker `ld.so` can use dynamic symbol table to link the executable against the library at run-time
- To see details:

```
objdump -T myprog    (--dynamic-syms)
```

https://stackoverflow.com/questions/19736853/what-does-ld-do-when-linking-against-

Dynamic relocation table

- creating the dynamic relocation table
  to check which machine code locations need to be changed
  to point toynamically linked symbols.

- To see details:

  `objdump -R myprog    (--dynamic-reloc)`

https://stackoverflow.com/questions/19736853/what-does-ld-do-when-linking-against-

Converted relocation types

- that `ld` takes object files with various relocation types
  - representing anything the compiler or assembler can produce
- resolves most of them except a small number of relocation types
  - for static linking, unresolved relocations are not allowed
  - for dynamic linking, all the remaining relocations shall be converted into a limited set of relocation types shall be resolved by the dynamic linker at load time.

`https://stackoverflow.com/questions/19736853/what-does-ld-do-when-linking-against-`

- At the link time
- `ld-linux.so` vs. `ld.so`
- `glibc`
- `ld-linux.so`

(1) dynamic applications

- a dynamic applications (binary, executable)
  - consist of one or more dynamic objects
  - typically a dynamic executable and one or more shared object dependencies
- run time linker for dynamic objects

https://renenyffenegger.ch/notes/development/dynamic-loader
https://docs.oracle.com/cd/E19253-01/816-5165/ld.so.1-1/index.html

(2) search shared libraries

- to see the shared object libraries used by a given application use the `ldd` command
- shared library directories
  - `/lib`
  - `/usr/lib`.
- additional search directory
  - `/etc/ld.so.conf` can be used to configure the dynamic loader to search for other directories (eg. `/usr/local/lib` or `/opt/lib`)

https://renenyffenegger.ch/notes/development/dynamic-loader
https://docs.oracle.com/cd/E19253-01/816-5165/ld.so.1-1/index.html

(3) `ldd` print shared object dependencies

- `ldd` prints the shared objects (shared libraries) required by each program or shared object specified on the command line.
- An example of its use and output is the following:

```
$ ldd /bin/ls
   linux-vdso.so.1 (0x00007ffcc3563000)
   libselinux.so.1 => /lib64/libselinux.so.1 (0x00007f87e5459000)
   libcap.so.2 => /lib64/libcap.so.2 (0x00007f87e5254000)
   libc.so.6 => /lib64/libc.so.6 (0x00007f87e4e92000)
   libpcre.so.1 => /lib64/libpcre.so.1 (0x00007f87e4c22000)
   libdl.so.2 => /lib64/libdl.so.2 (0x00007f87e4a1e000)
   /lib64/ld-linux-x86-64.so.2 (0x00005574bf12e000)
   libattr.so.1 => /lib64/libattr.so.1 (0x00007f87e4817000)
   libpthread.so.0 => /lib64/libpthread.so.0 (0x00007f87e45fa000)
```

https://stackoverflow.com/questions/19736853/what-does-ld-do-when-linking-against-

(4) loading shared libraries

- most modern programs are dynamically linked
- when a dynamically linked application is loaded by the operating system kernel
- the dynamic loader must locate and load the dynamic libraries it needs for execution.

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

(5) interpreter

- As part of the *initialization* and *execution* of a dynamic application, an interpreter is called
  - to run the executable, an interpreter program is used
- 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

(6) 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	a.out	
ld-linux.so	ELF	
/lib/ld-linux.so.1		libc5
/lib/ld-linux.so.2		glibc2

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

(7) specifying an interpreter

- ELF allows executables to specify an interpreter,
  - the compiler and static linker set the interpreter of executables
  - the interpreter is set to be `/lib/ld-linux-ia64.so.2` which is the dynamic linker
- when the kernel loads the binary executable
  - it will check if the `PT_INTERP` field is present
  - if so load what it points to into memory and start it.

https://www.bottomupcs.com/dynamic_linker.xhtml

(8) dynamic linker name

- linux's dynamic loader / linker
  - `ld.so` for `a.out`
  - `ld-linux.so` for `ELF`
  - ld-linux.so.2 for `glibc`

  - `/lib/ld-linux.so.2`
  - `/lib/ld-linux-x86-64.so.2`

- finding the name of the dynamic loader with
  `readelf -l executable | grep interpreter`
  - `readelf -l` dsplays the information contained in the file's segment headers

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

(9) executing an interpreter

- indirect execution
  by running some dynamically linked program or shared object
  - the dynamic linker is specified
    in the `.interp` section of an ELF file (program)
  - no command-line options to the dynamic linker
- direct execution
  by the command-line
  - `/lib/ld-linux.so.*  [OPTIONS] [PROGRAM [ARGUMENTS]]`

`man ld-linux.so`

- The dynamic linker is the program that *manages* shared dynamic libraries *on behalf of* an executable.
    - load libraries into memory
    - modify the program at runtime (resolving relocation)
    - call the functions in the library

https://www.bottomupcs.com/dynamic_linker.xhtml

(11) relocations

- dynamically linked executables leave behind references that will be fixed at the runtime
  - eg. the address of a function in a shared library.
  - the references that are left behind are called relocations
- the essential part of the dynamic linker is fixing up these unresolved addresses at runtime,
  - these addresses can be known only when the executable and shared libraries are loaded in memory

https://www.bottomupcs.com/dynamic_linker.xhtml

(12) resolving relocations

- A relocation can simply be thought of as a *note* that a particular address will need to be fixed at the load time of the runtime
- before the code is ready to run all the relocations need to be resolved
  - fixing the addresses it refers to to point to the right place.

https://www.bottomupcs.com/dynamic_linker.xhtml

(13) base address

- the executable code is not shared, and each executable gets its own fresh address space
  - in an executable file, the code and data segments are given by a base address in virtual memory
  - the compiler knows exact location of the data section and can reference it directly
- shared libraries have no such guarantee.
  - the data section will be a specified as an offset from the base address
  - but exact location of the base address can only be known at runtime

https://www.bottomupcs.com/dynamic_linker.xhtml

(14) PIC

- all the shared libraries must be produced as position independent codes (PIC).
- note that the data section is still specified as a fixed offset from the code section;
- but to actually find the address of data the offset needs to be added to the load address

https://www.bottomupcs.com/dynamic_linker.xhtml

(15) SONAME

- the string written to the executable will actually be the SONAME of the library, e.g. `mylib.so.0`
- This will ensure that even when a newer and incompatible `mylib.so.1.42` is installed later, the executable will use the compatible ABI version 0 instead.
  - To see details:

    ```
    ldd myprog
    ```

https://stackoverflow.com/questions/19736853/what-does-ld-do-when-linking-against-

(16) Symbolic link

- Usually dynamic libraries are set up using symlinks only
  - `libfoo.so` is used by `ld`, and
  - `libfoo.so` points to `libfoo.so.1` or
    to whatever which is used by `ld.so`, and
  - `libfoo.so` is itself typically a symlink
    to the currently-installed version of the library,
    e.g. `libfoo.so.1.2.3`

https://unix.stackexchange.com/questions/449107/what-differences-and-relations-are

(17) libc

- libc implements both
  standard C functions like strcpy() and
  POSIX functions (which may be system calls) like getpid()
  Note that not all standard C functions are in libc
  - most math functions are in libm

https://stackoverflow.com/questions/11372872/what-is-the-role-of-libcglibc-in-our-

(18) System calls and thunks

- system calls is different from normal functions
  because they call to the kernel
  they can't be resolved by the linker
- architecture-specific assembly language thunks
  are used to call into the kernel
- `libc` provides those assembly language thunks

https://stackoverflow.com/questions/11372872/what-is-the-role-of-libcglibc-in-our-

(19) libc and glibc

- in Linux, it is the combination of the kernel and `libc` that provides the POSIX API
- `libc` is a single library file (both .so and .a versions are available) in most cases resides in `/usr/lib`
- the `glibc` (GNU `libc`) project provides more than just `libc` it also provides the `libm` and other core libraries like `libpthread`
- So `libc` is just one of the libraries provided by `glibc` and there are other alternate implementations of `libc` other than `glibc`

https://stackoverflow.com/questions/11372872/what-is-the-role-of-libcglibc-in-our-

(20) glibc

1. C library described in ANSI,c99,c11 standards.
   - includes macros, symbols, function implementations etc.
   - `printf()`, `malloc()` etc
2. POSIX standard library.
   - the "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

(21) libc.so

- libc.so is usually a linker script
  - pointing to
    - the 64-bit C library (dynamic or shared)
    - dynamic linker
  - used to link 64-bit executables at the build-time
  - provides instructions for ld
-
```
/* GNU ld script
   Use the shared library, but some functions are only in
   the static library, so try that secondarily.  */
OUTPUT_FORMAT(elf64-x86-64)
GROUP ( /lib/x86_64-linux-gnu/libc.so.6
        /usr/lib/x86_64-linux-gnu/libc_nonshared.a
   AS_NEEDED ( /lib/x86_64-linux-gnu/ld-linux-x86-64.so.2 ) )
```

https://unix.stackexchange.com/questions/449107/what-differences-and-relations-are

(22) Linker script

- In the GNU C library's case
  dynamically linked programs still need
  some symbols from the static library
  so a linker script is used instead
  so that the linker can try both
  (dynamic linking and static linking)

- the linker script also refers to the dynamic linker
  which will be used at the runtime
  (`/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2`)
  its name is embedded in executables in `.interp`

https://unix.stackexchange.com/questions/449107/what-differences-and-relations-are