ELF1 7A Linking Background - ELF Study 1999

Young W. Lim

2020-07-29 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

- Static vs. dynamic linking
- Static vs. dynamic binaries
- Build-time, load-time, run-time 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-dynamica

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-dynamica

TOC: Static vs. dynamic binaries

Static binary (1) copying

- dynamic binaries load code from external binaries (.so file) at run time
- in static binairies, library code (.a libraries) is copied inside the binary at build time
- advantages of dynamic binaries are
  - libraries can be reused between different running applications.
  - so they need less memory
  - libraries can be changed later on without recompiling as long as the ABI (Application binary interface) of the library doesn't change.

https://www.reddit.com/r/linux/comments/6pkzf5/static_and_dynamic_binaries/

Static binary (2) non PIC

- code in a static library need not be PIC
  - position dependent code can jump and call directly without needing any intermediate steps
  - the linker adjusts the instructions/data for direct cross-references
- this performance benefit shows up only in high-performance code; e.g., you might get a teenthy boost from a video encoder if you statically compile & link it.
- if the program does any blocking I/O it won't matter at all.

https://www.reddit.com/r/linux/comments/6pkzf5/static_and_dynamic_binaries/

Static binary (3) link time optimization

- modern linkers are able to do link-time optimization (LTO)
  - the compiler does its job and emits some GIMPLE bytecode along with the usual machine code in the object file.
  - the bytecode gives a gist of
    - what functions/variables are where,
    - what everything does,
    - how to piece it together,
  - and possibly some analysis that's been done as well, since that's a side-effect of optimization.

https://www.reddit.com/r/linux/comments/6pkzf5/static_and_dynamic_binaries/

Static binary (4) performance

- The linker can use optimization analysis
  to knit together code from several object files
  when it's producing the final executable,
- without LTO, the linker basically mashes pieces of object files together as indivisible blocks (+noise).
- So in theory, you might be able to do significantly better with static compilation as long as
  - sufficiently many of your object files are produced by gcc -flto
  - linker supports LTO
  - burning on inter-object calls/accesses in loops.

https://www.reddit.com/r/linux/comments/6pkzf5/static_and_dynamic_binaries/

Dynamic binary (1) a program and its dependencies

- the shared libraries must be able to be loaded anywhere in the process' virtual address space and must be relocated
- the kernel does only map the program file in memory the dynamic linker (the interpreter) must
  - locate and map all dependencies as well as shared object specified in LD_PRELOAD
  - relocate the files

https://www.gabriel.urdhr.fr/2015/01/22/elf-linking/#base-address

Dynamic binary (2) kernel

- the kernels initialises the process:
- it maps in the virtual address space
  - the main program,
  - the interpreter (dynamic linker) segments
  - the vDSO (virtual Dynamic Shared Object)
- it sets up the stack (passing the arguments, environment)
- calls the dynamic linker entry point

https://www.gabriel.urdhr.fr/2015/01/22/elf-linking/#base-address

Dynamic binary (3) vDSO

- the vDSO (virtual Dynamic Shared Object)
  - a Linux kernel mechanism for exporting a carefully selected set of kernel space routines to user space applications
  - to overcome the performance penalty of a mode switch from user to kernel mode that is inherent when system call interface is used

https://www.gabriel.urdhr.fr/2015/01/22/elf-linking/#base-address

Dynamic binary (4) dynamic linker

- the dynamic linker loads the different ELF objects and binds them together
  - relocates itself
  - finds and loads the necessary libraries
  - does the relocations (which binds the ELF objects)
  - calls the initialisation functions of the shared objects
    - those functions are specified in the DT_INIT and DT_INIT_ARRAY entries of the ELF objects.
  - calls the main program entry point
    - found in the AT_ENTRY entry of the auxiliary vector: it has been initialised by the kernel from the e_entry ELF header field.
- the executable then initialises itself.

https://www.gabriel.urdhr.fr/2015/01/22/elf-linking/#base-address

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

- Build-time, load-time, run-time
- Build-time vs. load-time linking
- (1) build-time linking for staic executables / libraries
- (2) build-time linking for dynamic executables / libraries
- (3) load-time linking for dynamic executables / libraries
- Load-time vs. run-time dynamic linking
- Run-time dynamic linking
- Build-time linker `ld`
- Run-time linker `ld.so`
- Linker at the build time
- Kernel at the load time
- Dynamic loader at the load time

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 executables	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

- 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: 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-

initializing and terminating routines

- Dynamic objects may supply code for
  - runtime initialization processing
    - this code is executed once
      each time the dynamic object is loaded in a process
  - runtime termination processing
    - this is executed once
      each time the dynamic object is unloaded from a process or
      at process termination

`https://docs.oracle.com/cd/E19253-01/817-1984/chapter2-48195/index.html`

related sections for initializing and terminating routines

1. an array of function pointers
   - `.preinit_array` section : arrays of runtime pre-initialization
   - `.init_array` section : arrays of initialization
   - `.fini_array` section : arrays of termination
2. a single code block.
   - `.init` section : a runtime initialization code block
   - `.fini` section : a runtime termination code block

- Each of these section types is built from a concatenation of like sections from the input relocatable objects.

https://docs.oracle.com/cd/E19253-01/817-1984/chapter2-48195/index.html

Making an array of function pointers

- the link editor identifies these arrays with the `.dynamic` tags
  - `DT_PREINIT_ARRAY` and `DT_PREINIT_ARRAYSZ`
  - `DT_INIT_ARRAY` and `DT_INIT_ARRAYSZ`
  - `DT_FINI_ARRAY` and `DT_FINI_ARRAYSZ`

  so that they may be called by the runtime linker
- functions that are assigned to these arrays
  must be provided from the object that is being built.

https://docs.oracle.com/cd/E19253-01/817-1984/chapter2-48195/index.html

Making a single code block

- the compiler drivers typically supply `.init` and `.fini` sections with files they add to the beginning and end of your input file list.
- these compiler provided files have the effect of encapsulating the `.init` and `.fini` code from your relocatable objects into individual functions
  - the function identified by the symbol `_init`
  - the function identified by the symbol `_fini`
- the link-editor identifies these symbols with the `.dynamic` tags
  - `DT_INIT`
  - `DT_FINI`

  so that they may be called by the runtime linker

https://docs.oracle.com/cd/E19253-01/817-1984/chapter2-48195/index.html

TOC: Load-time linking for dynamic executables / libraries

- 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`

(10) managing shared libraries

- 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

e_entry

- 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.
- File header
  - The ELF header defines whether to use 32-bit or 64-bit addresses. The header contains three fields that are affected by this setting and offset other fields that follow them. 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

finding main function's entry point

As far as I know, 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: in the ELF format, the start of the program is specified by the e_entry field of the ELF executable 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.
In some cases, implementation-specific heuristics (i.e., the specific knowledge of the internals of the C runtime) could be used to determine the location of main in a stripped executable. However, I am not aware of a portable way to do so.

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