ELF1 7B Loading Background - ELF Study 1999

Young W. Lim

2020-11-03 Tue

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: 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-a`

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-a

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-ar

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 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-ar

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-ar

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-an

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-an

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-ar

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 standar 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-a`

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-a

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-ar

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: Load addreses

- Memory Map
- Library load addreses

TOC: Memory Map

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

Load address

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

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

i386 load addresses 1999 (increasing from the top)

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

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

i386 load addresses 1999 (increasing from the bottom)

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

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

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

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

mmap (1)

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

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

mmap (2)

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

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

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

sys_brk (1)

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

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

sys_brk (2)

- ```
  #include <unistd.h>

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

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

TOC: Library load addresses

Library load addresses (1)

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

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

Library load addresses (2)

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

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

Shared library address

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

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

Dynamic loader names

- dynamic loader
- dynamic linker
- runtime linker
- interpreter
- `ld-linux.so.2`
- `ld-linux.so`
- `ld.so`

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

load address example (1)

- 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 it `argv[1]` and loads the `foo_app` at its preferred location (`0x0800.0000`)
- other libraries are loaded up a the mmap area.

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

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

library built with -fPIC

- 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-ra

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 cannotmodify themselves while running.
  it is necessary to write libraries.
- re-entrant code is useful for shared libraries
- Some functions in a library may be reentrant, whereas
  others in the same library are non-reentrant.
- A library is reentrant if and only if
  all of the functions in it are reentrant.

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