Link 3B. Shared Libraries

Young W. Lim

2024-11-13 Wed

Outline

1. Based on

2. Shared libraries Overview
   - Including libraries in an executable
   - Shared library background
   - Specifying linker option using `-W,option`

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`

Code sharing (1)

- the operating system code is considered read only, and separated from data.
- if programs can not modify code and have large amounts of common code, instead of having *multiple copies* of common code for each executable it would be better to share a *single copy* between many executables.

https://bottomupcs.sourceforge.net/csbu/c3673.htm

Code sharing (2)

- With virtual memory this can be easily done.
  - The physical pages of memory,
    which the library code is loaded into,
    can be easily referenced
    by any number of virtual pages
    in any number of address spaces (process ID)
  - only have a single physical copy
    of the library code in system memory
  - every process can have access to that library code
    at any virtual address it likes.

https://bottomupcs.sourceforge.net/csbu/c3673.htm

Code sharing (3)

- sharing a library :
  a single copy of a library code loaded in the memory
  is shared by *multiple* executables

- each executable contains only a reference to a library

https://bottomupcs.sourceforge.net/csbu/c3673.htm

Code sharing (4)

- when the program is loaded for execution, it is up to the system
  - to check if some other program has already loaded the library code `foo` into memory,
  - if so, share it by mapping pages into the executable for the physical memory where the library `foo` has been loaded
  - otherwise load the library `foo` into memory for the executable

https://bottomupcs.sourceforge.net/csbu/c3673.htm

Code sharing (5)

- This process is called dynamic linking
  because it does part of the linking process *"on the fly"*
  as programs are executed in the system.
  - sharing the library code (already loaded by another executable)
  - loading the library code (never been loaded)

`https://bottomupcs.sourceforge.net/csbu/c3673.htm`

Compilation (1)

- when you compile your program
  that uses a dynamic library,

  object files contains only
  references to the library functions
  just as for any other external reference.

- need to include the header for the library functions
  to inform the compiler about the function prototype
  the specific types of the functions you are calling.

https://bottomupcs.sourceforge.net/csbu/c3673.htm

Compilation (2)

- the compiler *only* needs to know the types associated with a function
  - the prototype of a function
  - such as, it takes an `int` and returns a `char *`
- so that it can correctly allocate space for the function call.
  - the stack frame for the function call

https://bottomupcs.sourceforge.net/csbu/c3673.htm

Linking (1)

- the traditional linker still has a role to play in creating the executable.
  - the traditional linker records references to the library functions in the executable
  - the dynamic section of the executable requires a `NEEDED` entry for each shared library that the executable depends on.
- the dynamic linker can determine which shared libraries will satisfy the dependencies at runtime

https://bottomupcs.sourceforge.net/csbu/c3673.htm

Linking (2)

- can inspect NEEDED fields with the readelf command
- Specifying Dynamic Libraries

```
$ readelf --dynamic /bin/ls

Dynamic segment at offset 0x22f78 contains 27 entries:
  Tag        Type                         Name/Value
 0x0000000000000001 (NEEDED)             Shared library: [librt.so.1]
 0x0000000000000001 (NEEDED)             Shared library: [libacl.so.1]
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6.1]
 0x000000000000000c (INIT)               0x4000000000001e30
 ... snip ...
```

- here, three libraries are specified
  - the most commonly shared library is libc.
  - the other libraries are libacl and librt

https://bottomupcs.sourceforge.net/csbu/c3673.htm

Linking (3)

- ldd "walks" the dependencies of libraries for you;
  that is if a library depends on another library,
  it will show it to you.
- Looking at dynamic libraries

```
$ ldd /bin/ls
    librt.so.1 => /lib/tls/librt.so.1 (0x2000000000058000)
    libacl.so.1 => /lib/libacl.so.1 (0x2000000000078000)
    libc.so.6.1 => /lib/tls/libc.so.6.1 (0x2000000000098000)
    libpthread.so.0 => /lib/tls/libpthread.so.0 (0x20000000002e0000)
    /lib/ld-linux-ia64.so.2 => /lib/ld-linux-ia64.so.2 (0x2000000000000000)
    libattr.so.1 => /lib/libattr.so.1 (0x2000000000310000)
librt -> libacl -> libc -> libpthread -> ld-linux-ia64 -> libattr~
```

https://bottomupcs.sourceforge.net/csbu/c3673.htm

Linking (4)

- Looking at dynamic libraries

```
librt.so.1 => /lib/tls/librt.so.1 (0x2000000000058000)
libacl.so.1 => /lib/libacl.so.1 (0x2000000000078000)

$ readelf --dynamic /lib/librt.so.1

Dynamic segment at offset 0xd600 contains 30 entries:
  Tag        Type                         Name/Value
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6.1]
 0x0000000000000001 (NEEDED)             Shared library: [libpthread.so.0]
 ... snip ...
```

- libpthread has been required from `librt`

https://bottomupcs.sourceforge.net/csbu/c3673.htm

Using `ldd` command (1)

- the `ldd` command provides a way to view the shared libraries that a program is dynamically linked against.
- a tool that helps developers
  - understand the dependencies of their programs and
  - optimize their performance.

https://ioflood.com/blog/ldd-linux-command/

Using ldd command (2)

-v	Provides detailed information.	ldd -v /usr/bin/grep
-u	Shows unused direct dependencies.	ldd -u /usr/bin/grep
-r	Shows relocation processing.	ldd -r /usr/bin/grep
-d	Shows missing function dependencies.	ldd -d /usr/bin/grep
-e	Sets the environment variable.	ldd -e LD_LIBRARY_PATH=/lib
-f	Specifies the format.	ldd -f '%p %o' /usr/bin/grep
-n	Avoids displaying the version number.	ldd -n /usr/bin/grep
-N	Specifies the version.	ldd -N2 /usr/bin/grep
-q	Quiet mode. Only display errors.	ldd -q /usr/bin/grep
-h	Displays help.	ldd -h

https://ioflood.com/blog/ldd-linux-command/

Using ldd command (3)

```
[root@wdctc1281 bin]#  ldd node
        linux-vdso.so.1 =>  (0x00007fffd33f2000)
        libdl.so.2 => /lib64/libdl.so.2 (0x00007f70f7855000)
        librt.so.1 => /lib64/librt.so.1 (0x00007f70f764d000)
        libstdc++.so.6 => /lib64/libstdc++.so.6 (0x00007f70f7345000)
        libm.so.6 => /lib64/libm.so.6 (0x00007f70f7043000)
        libgcc_s.so.1 => /lib64/libgcc_s.so.1 (0x00007f70f6e2d000)
        libpthread.so.0 => /lib64/libpthread.so.0 (0x00007f70f6c10000)
        libc.so.6 => /lib64/libc.so.6 (0x00007f70f684f000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f70f7a61000)
```

What does the first line and last line mean?
They don't look like the normal

```
xxxx.so => /lib64/xxxxx.so (0xxxxxxxxxxxxxxxxxxxxx)
```

https://stackoverflow.com/questions/34428037/how-to-interpret-the-output-of-the-lo

Using ldd command (4)

- the first line is the VDSO.

  `linux-vdso.so.1 =>  (0x00007fffd33f2000)`

  this is described in depth in the vdso(7) manpage.
  basically it's a shared library that's embedded in your kernel
  and automatically loaded whenever a new process is exec-ed.
- that's why there's no filesystem path on the right side
  there is none!
- the file only exists in the kernel memory
  (well, not 100% precise, but see the man page for more info).

https://stackoverflow.com/questions/34428037/how-to-interpret-the-output-of-the-lo

Using `ldd` command (5)

- the last line is the ELF interpreter.
  `/lib64/ld-linux-x86-64.so.2 (0x00007f70f7a61000)`
  - the ELF interpreter is described in depth in the `ld.so` manpage.
  - it has a full path because your program `node` has the full path hardcoded in it.
    `/lib64/ld-linux-x86-64.so.2`
  - it doesn't have an entry on the right side
    `=> /lib64/xxxxx.so`
  - because it's not linked against (stand alone) thus no search was performed.

https://stackoverflow.com/questions/34428037/how-to-interpret-the-output-of-the-lo

Using ldd command (6)

- you can check this by running:

```
$ readelf -l node | grep interpreter
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]

$ scanelf -i node
ET_EXEC /lib64/ld-linux-x86-64.so.2 node
```

  - scanelf is a user-space utility to quickly scan given ELFs, directories, or common system paths for different information.
  - this may include ELF types, their PaX markings, TEXTRELs, etc. . .
  - to print INTERP information
    `scanelf -i, --interp`

https://stackoverflow.com/questions/34428037/how-to-interpret-the-output-of-the-l

Using ldd command (7)

- all the other lines are libraries you've linked against.

```
libdl.so.2 => /lib64/libdl.so.2 (0x00007f70f7855000)
librt.so.1 => /lib64/librt.so.1 (0x00007f70f764d000)
libstdc++.so.6 => /lib64/libstdc++.so.6 (0x00007f70f7345000)
libm.so.6 => /lib64/libm.so.6 (0x00007f70f7043000)
libgcc_s.so.1 => /lib64/libgcc_s.so.1 (0x00007f70f6e2d000)
libpthread.so.0 => /lib64/libpthread.so.0 (0x00007f70f6c10000)
libc.so.6 => /lib64/libc.so.6 (0x00007f70f684f000)
```

- you can see those by looking at `DT_NEEDED` tags when you run `readelf -d` on the file.

https://stackoverflow.com/questions/34428037/how-to-interpret-the-output-of-the-l

Using ldd command (8)

- since those files lack full paths,
  the ld.so needs to perform a dynamic path search for it.
- that's actually what the lines are telling you:
  `libdl.so.2 => /lib64/libdl.so.2 (0x00007f70f7855000)`
    - libdl.so.2 is needed, so when ld.so searched for it,
      ld.so found it at /lib64/libdl.so.2
    - and was loaded into memory at address 0x00007f70f7855000

https://stackoverflow.com/questions/34428037/how-to-interpret-the-output-of-the-lo

vDSO (1)

- vDSO (virtual dynamic shared object) is a kernel mechanism for exporting a carefully selected set of kernel space routines to user space applications
  - applications can call these kernel space routines in-process, without the performance overhead of a mode switch from user mode to kernel mode
  - the performance overhead is inherent when calling these same kernel space routines by means of the system call interface.

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

vDSO (2)

- vDSO uses standard ELF format for linking and loading
- vDSO is a memory area allocated in user space
  - *exposes* some kernel functionalities
  - dynamically allocated to offer improved safety through address space layout randomization
  - supports more than *four* system calls
- Some C standard libraries, like glibc, may provide vDSO links so that if the kernel does not support vDSO, a traditional `syscall` is made.

`https://en.wikipedia.org/wiki/VDSO`

vDSO (3)

- vDSO helps to
  - reduce the calling overhead on simple kernel routines
  - the best method of performing a system call on IA-32
    - such exported routines can provide proper DWARF (Debug With Attributed Record Format) debugging information.
- implementation generally implies
  hooks in the dynamic linker to find the vDSOs

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

Creating shared libary exampe (1)

Firstly I test with all dynamic: gcc -shared libtest.c -o libtest.so gcc -c main.c -o main.o gcc main.o -o test -L. -ltest
It's working (compile and execute)
Secondly I test what I want (dynamic lib and static libc) : gcc -shared libtest.c -o libtest.so gcc -c main.c -o main.o gcc main.o -o test libtest.so /usr/lib/libc.a
It's compiling, but at execution, it segfault! A strace show that it's trying to access libc.so!!!
Finally I've tried to compile a simple progam with no reference to dynamic lib gcc -static main.c –> compile ok, run ok gcc main.c /usr/lib/libc.a –> compile ok, run : segmentation fault (a strace show that it's access to libc.so)

https://stackoverflow.com/questions/2176181/how-to-linking-with-dynamic-lib-so-and

Creating shared libary exampe (2)

$ cat libtest.c #include <stdio.h> void foo() { printf("%d", 42); } $ cat main.c #include <stdio.h> extern void foo(); int main() { puts("The answer is:"); foo(); } $ export LD_LIBRARY_PATH=$PWD $ gcc -shared libtest.c -o libtest.so && gcc -c main.c -o main.o && gcc main.o -o test -L. -ltest && ./test The answer is: 42 $ gcc -shared libtest.c -o libtest.so && gcc -c main.c -o main.o && gcc main.o -o test libtest.so /usr/lib/libc.a && ./test The answer is: 42

https://stackoverflow.com/questions/2176181/how-to-linking-with-dynamic-lib-so-and

Creating shared libary exampe (3)

However, you have to realise that the shared library you've build depends on the shared libc. So, it's natural that it's trying to open it at runtime.
$ ldd ./libtest.so linux-gate.so.1 => (0xb80c7000) libc.so.6 => /lib/i686/cmov/libc.so.6 (0xb7f4f000) /lib/ld-linux.so.2 (0xb80c8000)
One way to achieve what you want is to use: -static-libgcc -Wl,-Bstatic -lc.

`https://stackoverflow.com/questions/2176181/how-to-linking-with-dynamic-lib-so-and`

Creating shared libary

- *create* a file `library.c`
- *compile* the `library.c` file

  ```
  gcc -shared -fPIC -o liblibrary.so library.c
  ```

  - `-shared` instructs the compiler
    that we are building a shared library
  - `-fPIC` is to generate position-independent code
  - generates a shared library `liblibrary.so`
    in the current working directory.
  - We have our shared object file
    (shared library name in Linux)
    ready to use.

https://www.geeksforgeeks.org/working-with-shared-libraries-set-2/

Using shared libary (1)

- *create* a file `application.c`
- *compile* the `application.c` file

```
gcc application.c -L /home/coding/ -library -o sample
```

  - `-library` instructs the compiler
    to look for symbol definitions
    that are not available in the current code
  - the option `-L` is a hint to the compiler
    to look in the directory followed by the option
    for any shared libraries (during link-time only).
  - generates an executable named `sample`

https://www.geeksforgeeks.org/working-with-shared-libraries-set-2/

Using shared libary (2)

- By default, it will not look into the current working directory
- You have to explicitly instruct the tool chain to provide proper paths
- otherwise, when you invoke the executable, the dynamic linker will not be able to find the required shared library

https://www.geeksforgeeks.org/working-with-shared-libraries-set-2/

Using shared libary (3)

- The dynamic linker searches standard paths
  available in the LD_LIBRARY_PATH
  and also searches in the system cache
- We have to add our *working directory*
  to the LD_LIBRARY_PATH environment variable

  ```
  export LD_LIBRARY_PATH=/home/work/:$LD_LIBRARY_PATH
  ```
- You can now invoke our executable as shown.
  ./sample

https://www.geeksforgeeks.org/working-with-shared-libraries-set-2/

gcc -fpic (1)

- `-fpic`
  - generate position-independent code (PIC) suitable for use in a shared library, if supported for the target machine.
  - Such code accesses all constant addresses through a global offset table (GOT).
  - The dynamic loader resolves the GOT entries when the program starts
    - the dynamic loader is not part of GCC; it is part of the operating system

`man gcc`

gcc -fpic (2)

- `-fpic`
  - If the GOT size for the linked executable *exceeds* a machine-specific maximum size, you get an error message from the linker indicating that `-fpic` does not work; in that case, recompile with `-fPIC` instead.
    - These maximums are 8k on the SPARC, 28k on AArch64 and 32k on the m68k and RS/6000.
    - The x86 has no such limit

`man gcc`

gcc -fPIC

- `-fPIC`
  - if supported for the target machine,
    emit position-independent code,
    suitable for dynamic linking and
    avoiding any limit on the size of the global offset table
    - This option makes a difference on
      AArch64, m68k, PowerPC and SPARC (not on x86)
  - Position-independent code requires special support,
    and therefore works only on certain machines.
  - When this flag is set, the macros `__pic__` and `__PIC__`
    are defined to 2.

`man gcc`

`gcc -shared` (1)

- `-shared`
  - produce a shared object
    which can then be *linked* with other objects
    to form an executable
  - not all systems support this option.
  - For predictable results, you must also specify
    the same set of options used for *compilation*
    (`-fpic`, `-fPIC`, or model suboptions)
    when you specify this linker option

`man gcc`

gcc -shared (2)

- Shared libraries and executables use the same format: they are both loadable images. However,
  - shared libraries are usually position-independent, executables are often not
    - This affects code generation: for position-independent you have to load globals or jump to functions using relative addresses
  - executables have an entry point which is where execution starts.
    - this is usually not `main()`, because `main()` is a function, and functions return, but execution should never return from the entry point

https://stackoverflow.com/questions/25084855/how-does-gcc-shared-option-affect-the

gcc -shared (3)

- parameters for collect2 without -shared:

```
-dynamic-linker
/lib64/ld-linux-x86-64.so.2
/usr/lib/gcc/x86_64-linux-gnu/4.7/../../../x86_64-linux-gnu/crt1.o
/usr/lib/gcc/x86_64-linux-gnu/4.7/crtbegin.o
/usr/lib/gcc/x86_64-linux-gnu/4.7/crtend.o
```

- parameters for collect2 with -shared:

```
-shared
/usr/lib/gcc/x86_64-linux-gnu/4.7/crtbeginS.o
/usr/lib/gcc/x86_64-linux-gnu/4.7/crtendS.o
```

https://stackoverflow.com/questions/25084855/how-does-gcc-shared-option-affect-the

gcc -shared (4)

- you still have to use -fpic or -fPIC, when -shared is used
- it looks like code generation is not affected:
- crt1.o (the C runtime) is only included
  when linking the executable,
  and thus when -shared is not used
  or when Wl,-shared is used

https://stackoverflow.com/questions/25084855/how-does-gcc-shared-option-affect-the

gcc -shared (5)

- Using nm crt1.o

```
$ nm /usr/lib/x86_64-linux-gnu/crt1.o
0000000000000000 R _IO_stdin_used
0000000000000000 D __data_start
                 U __libc_csu_fini
                 U __libc_csu_init
                 U __libc_start_main
0000000000000000 T _start
0000000000000000 W data_start
                 U main
```

- seems to define something to do with stdin, as well as _start (which is the entry point),
- has an undefined reference to main

https://stackoverflow.com/questions/25084855/how-does-gcc-shared-option-affect-the

gcc -shared (6)

- passing -shared to gcc (gcc -shared)
  - gcc -shared -Wl,-soname,libtest.so -o libtest.so *.o
  - may enable or disable other flags at link time.
    different *~crt*~* files might be involved.
  - To get more information, grep for -shared
    in GCC's gcc/config/ directory and subdirectories.
- passing -shared to ld (gcc -Wl,-shared).
  - gcc -Wl,-shared -Wl,-soname,libtest.so -o libtest.so *.o
  - on i386 FreeBSD, gcc -shared will link in object file crtendS.o,
    while without -shared, it will link in crtend.o instead.

https://stackoverflow.com/questions/4623915/difference-between-shared-and-wl-share

`ld -R filename`

- `-R filename`
  `--just-symbols=filename`
  - read *symbol names* and their *addresses* from filename
  - but do not *relocate* it or *include* it in the output.
    - this allows your output file to refer symbolically to absolute locations of memory defined in other programs.
  - may use this option more than once
  - for compatibility with other ELF linkers, if the `-R` option is followed by a directory name, rather than a file name, it is treated as the `-rpath` option.

`man ld`

`ld -rpath=dir` (1)

- `-rpath=dir`
  - add a directory to the runtime library search path
  - used when linking an ELF executable with shared objects
  - all `-rpath` arguments are *concatenated* and *passed* to the runtime linker, which uses them to *locate* shared objects at runtime

`man ld`

ld -rpath=dir (2)

- -rpath=dir
  - also used when *locating* shared objects which are needed by shared objects explicitly included in the link;
    - see the description of the `-rpath-link` option.
    - Searching `-rpath` in this way is only supported by native linkers and cross linkers which have been configured with the `--with-sysroot` option.

man ld

ld -rpath=dir (3)

- -rpath=dir
  - if -rpath is not used when *linking* an ELF executable, the contents of the environment variable LD_RUN_PATH will be *used* if it is defined.
  - If a -rpath option is used, the runtime search path will be formed *exclusively* using the -rpath options, *ignoring* the -L options.
  - This can be useful when using gcc, which adds many -L options which may be on NFS mounted file systems.

man ld

ld -rpath=dir (4)

- -rpath=dir
  - for compatibility with other ELF linkers, if the -R option is followed by a directory name, *rather than* a file name, it is treated as the -rpath option.
  - the -rpath option may also be used on SunOS.
  - By default, on SunOS, the linker will form a runtime search path out of all the -L options it is given.

man ld

TOC: 5. `-Wl,-rpath,.` examples

Using -Wl,option

- Pass *option* as an option to the linker.
- If *option* contains commas,
  it is split into multiple options at the commas.
- You can use this syntax to pass an argument to the option.
- For example, `-Wl,-Map,output.map` passes
  `-Map output.map` to the linker.
- When using the GNU linker, you can also get the same effect with
  `-Wl,-Map=output.map`

https://gcc.gnu.org/onlinedocs/gcc/Link-Options.html

Using `-Wl, rpath .` (1)

- in order to pass `-rpath .` to the linker, consider them as two arguments (`-rpath` and `.`) to the `-Wl`
- you can write (`-Wl,arg1,arg2`) or (`-Wl,arg1, -Wl,arg2`)
  - `-Wl,-rpath,.`
  - `-Wl,-rpath -Wl,.`

https://stackoverflow.com/questions/6562403/i-dont-understand-wl-rpath-wl

Using -Wl,-rpath,. (2)

- the -Wl,xxx option for gcc passes
  a comma-separated list of tokens
  as a space-separated list of arguments
  to the linker (ld)
- to pass ld aaa bbb ccc (space separated)
  gcc -Wl,aaa,bbb,ccc (comma separated)
- to pass ld -rpath . (space separated)
  gcc -Wl,-rpath,. (comma separated)

https://stackoverflow.com/questions/6562403/i-dont-understand-wl-rpath-wl

Using -Wl,-rpath,. (3)

- alternatively, repeat instances of -Wl can be specified
- to pass `ld aaa bbb ccc` (space separated)
  `gcc -Wl,aaa -Wl,bbb -Wl,ccc` (repeated instances)
  - there is no comma between -Wl,aaa and the second -Wl,bbb but there is space
- thus, to pass `ld -rpath .`
  - `gcc -Wl,-rpath,.` (comma separated)
  - `gcc -Wl,-rpath -Wl,.` (repeated instances)

https://stackoverflow.com/questions/6562403/i-dont-understand-wl-rpath-wl

Using -Wl,-rpath,. (4)

- can remove the comma by using =

  gcc -Wl,-rpath=.
  - arguably more readable than adding extra commas
  - exactly what gets passed to ld

- thus, to pass ld -rpath .
  - gcc -Wl,-rpath,. (comma separated)
  - gcc -Wl,-rpath -Wl,. (repeated instances)
  - gcc -Wl,-rpath=. (using = instead of ,)

https://stackoverflow.com/questions/6562403/i-dont-understand-wl-rpath-wl

Using -Wl,-rpath,. (5)

- You may need to specify the -L option as well

  `-Wl,-rpath,/path/to/foo -L/path/to/foo -lbaz`

  or you may end up with an error like
  `ld: cannot find -lbaz`

https://stackoverflow.com/questions/6562403/i-dont-understand-wl-rpath-wl