Link 6. Library Search Examples

Young W. Lim

2023-10-23 Mon

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

Example source codes of foo(), bar(), foobar()

1. foo.c

```
#include <stdio.h>

void foo(void)
{
    puts(__func__);
    // puts("foo");
}
```

2. bar.c

```
#include <stdio.h>

void bar(void)
{
    puts(__func__);
    // puts("bar");
}
```

3. foobar.c

```
extern void foo(void);
extern void bar(void);

void foobar(void)
{
    foo();
    bar();
}
```

4. main.c

```
extern void foobar(void);

int main(void)
{
    foobar();
    return 0;
}
```

https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-li

Function dependencies of foo(), bar(), foobar()

main()	→	foobar()
foobar()	→	foo(), bar()

foobar()	in libfoobar.so
foo()	in libfoo.so
bar()	in libbar.so

https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-li

-L and -l examples

Example summary using -L and -l

1. Make two shared libraries, `libfoo.so` and `libbar.so`:

```
$ gcc -c -Wall -fPIC foo.c bar.c
$ gcc -shared -o libfoo.so foo.o
$ gcc -shared -o libbar.so bar.o
```

2. Make a third shared library, `libfoobar.so`

```
$ gcc -c -Wall -fPIC foobar.c
$ gcc -shared -o libfoobar.so foobar.o -L. -lfoo -lbar
```

3. Make a program that depends on `libfoobar.so`:

```
$ gcc -c -Wall main.c
$ gcc -o prog main.o -L. -lfoobar -lfoo -lbar
```

4. Execute using `LD_LIBRARY_PATH`

```
$ export LD_LIBRARY_PATH=.
$ ./prog
foo
bar
```

https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-li

Making libfoo.so, libbar.so, libfoobar.so

1. Make two shared libraries, libfoo.so and libbar.so:

```
$ gcc -c -Wall -fPIC foo.c bar.c
$ gcc -shared -o libfoo.so foo.o
$ gcc -shared -o libbar.so bar.o
```

2. Make a third shared library, libfoobar.so that depends on the first two;

```
$ gcc -c -Wall -fPIC foobar.c
$ gcc -shared -o libfoobar.so foobar.o -lfoo -lbar
/usr/bin/ld: cannot find -lfoo
/usr/bin/ld: cannot find -lbar
collect2: error: ld returned 1 exit status
```

   - The linker (ld) doesn't know where to look to resolve -lfoo or -lbar

https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-li

Using -L. -lfoo -lbar to make libfoobar.so

- The -L. informs where to look to resolve -lfoo and -lbar

```
$ gcc -shared -o libfoobar.so foobar.o -L. -lfoo -lbar
```

- The -L option (-Ldir) tells the linker
  that dir is one of the directories to search for libraries
  that resolve the -l option (-lfile) it is given.
  - the linker (ld) searches the -L directories first,
    in their command line order;
  - then it searches its configured default directories,
    in their configured order.

https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-li

Making an application that uses libfooba.so

- assume a program that depends on libfoobar.so:

```
$ gcc -c -Wall main.c
$ gcc -o prog main.o -L. -lfoobar
/usr/bin/ld: warning: libfoo.so, needed by ./libfoobar.so, not found
(try using -rpath or -rpath-link)
/usr/bin/ld: warning: libbar.so, needed by ./libfoobar.so, not found
(try using -rpath or -rpath-link)
./libfoobar.so: undefined reference to ‘bar’
./libfoobar.so: undefined reference to ‘foo’
collect2: error: ld returned 1 exit status
```

  - the linker (ld) detects the dynamic dependencies requested by libfoobar.so but can't satisfy them.
    - bar() in libbar.so
    - foo() in libfoo.so

https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-li

Using -L. -lfoobar -lfoo -lbar to make an application

- ignoring the gcc compiler's advice
  try using -rpath or -rpath-link
  use -L and -l only

```
$ gcc -o prog main.o -L. -lfoobar -lfoo -lbar
```

- -lfoo -lbar must also be specified together with -lfoobar
  for the dependencies of libfoobar.so
- the prog application can be made, but:

```
$ ./prog
./prog: error while loading shared libraries: libfoobar.so:\
cannot open shared object file: No such file or directory
```

  - at runtime, the loader (ld.so) can't find libfoobar.so
    nor libfoo.so and libbar.so

https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-li

Specifying the runtime shared library paths

- the -L option is used to tell the linker (ld)
  where to *find the libraries* (shared objects)
  in the compile, link time

- lots of ways of telling the runtime (dynamic loader ld.so)
  where to *find the libraries* (shared objects)
  in the runtime
  - -R
  - LD_LIBRARY_PATH
  - LD_RUN_PATH

https://stackoverflow.com/questions/31455979/how-to-specify-libraries-paths-in-gc

Using LD_LIBRARY_PATH to run an application

- without using -rpath or -rpath-link
  but using -L and -l only

```
$ gcc -o prog main.o -L. -lfoobar -lfoo -lbar
```

to make the prog application run :

```
$ export LD_LIBRARY_PATH=.
$ ./prog
foo
bar
```

  - at runtime,
    LD_LIBRARY_PATH enables the loader (ld.so)
    to find libfoobar.so, libfoo.so, and libbar.so
    in the current directory .

https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-li

-rpath-link examples

Example summary using -rpath-link

1. Make two shared libraries, libfoo.so and libbar.so:

```
$ gcc -c -Wall -fPIC foo.c bar.c
$ gcc -shared -o libfoo.so foo.o
$ gcc -shared -o libbar.so bar.o
```

2. Make a third shared library, libfoobar.so
   that depends on the first two;

```
$ gcc -c -Wall -fPIC foobar.c
$ gcc -shared -o libfoobar.so foobar.o -L. -lfoo -lbar
```

3. Make an application, prog that depends on libfoobar.so

```
$ gcc -c -Wall main.c
$ gcc -o prog main.o -L. -lfoobar -Wl,-rpath-link=$(pwd)
```

4. Make prog run

```
$ export LD_LIBRARY_PATH=.
$ ./prog
```

https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-li

-rpath-link example (1)

- the -rpath-link=dir option tells the linker that

  when an input file requests dynamic dependencies
  it should *search* directory dir to resolve them.

  - only the *direct* dependency libfoobar.so needs
    to be specified with -lfoobar

  - no need to specify the *nested* dependencies
    libfoo.so and libbar.so with -lfoo -lbar
    no need to know what they are

```
$ gcc -o prog main.o -L. -lfoobar -Wl,-rpath-link=$(pwd)
```

https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-li

-rpath-link example (2)

- with -rpath-link, we can do:

```
$ gcc -o prog main.o -L. -lfoobar -Wl,-rpath-link=$(pwd)
```

  - this linkage succeeds only, but not the execution
  - only the *direct* dependency needs to be specified
  - with -lfoobar, without -lfoo, -lbar (unnecessary)
  - *nested* dependencies inherit the search path of the current binary
  - in order to link successfully, $(pwd) is searched for libfoo.so, libbar.so, and libfoobar.so
  - $(pwd) means "Print Working Directory" and just displays the current path

https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-li

-rpath-link example (3)

- these *nested* dependencies of `libfoobar.so` (`libfoo.so`, `libbar.so`) is already written in its `.dynamic` section
- `-rpath-link=dir` can utilize these *nested* dependencies therefore, need not specify `-lfoo -lbar`

```
$ gcc -o prog main.o -L. -lfoobar -Wl,-rpath-link=$(pwd)

$ readelf -d libfoobar.so

Dynamic section at offset 0xdf8 contains 26 entries:
  Tag        Type                         Name/Value
 0x0000000000000001 (NEEDED)             Shared library: [libfoo.so]
 0x0000000000000001 (NEEDED)             Shared library: [libbar.so]
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
 ...
 ...
```

https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-li

-rpath-link example (4)

- therefore, for a successful linkage,
  just specify the *directory*
  where they can be found, whatever they are.
  - the dependencies of the current binary
    `prog` depends on `libfoobar.so` only
- `-rpath-link=dir`
  is used to specify such a *directory*
  - $(pwd) is the directory where `libfoobar.so` resides

```
$ gcc -o prog main.o -L. -lfoobar -Wl,-rpath-link=$(pwd)
```

https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-li

-rpath-link example (5)

- -rpath-link=dir
  - does not guarantee us a *runnable prog* but only a *successful linkage*
  - neither RUNPATH nor RPATH is set

```
$ gcc -o prog main.o -L. -lfoobar -Wl,-rpath-link=$(pwd)
$ ./prog
./prog: error while loading shared libraries: libfoobar.so\
: cannot open shared object file: No such file or directory
```

https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-li

-rpath-link example (6)

- -rpath-link=dir gives the linker the information
  - that the loader *would* need to resolve some of the dynamic dependencies of prog at runtime
  - only the *direct* dependency (libfoobar.so) is written in the .dynamic section of prog
  - assuming that the directory information remained unchanged at runtime
  - therefore, in order to execute prog, runtime search path must be specified

```
$ gcc -o prog main.o -L. -lfoobar -Wl,-rpath-link=$(pwd)
$ ./prog
./prog: error while loading shared libraries: libfoobar.so\
: cannot open shared object file: No such file or directory
```

https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-li

-rpath-link example (7)

- the dependency (libfoobar.so) of prog
  is recorded in the .dynamic section of prog
- the dependencies (libfoo.so, libbar.so) of libfoobar.so
  are recorded in the .dynamic section of libfoobar.so
- rpath-link=dir just lets the linkage succeed,
  without specifying all the *nested* dynamic dependencies
  of the linkage with -l options
- -lfoo -bar need not be specified

```
$ gcc -o prog main.o -L. -lfoobar -Wl,-rpath-link=$(pwd)

$ ./prog
./prog: error while loading shared libraries: libfoobar.so\
cannot open shared object file: No such file or directory
```

https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-li

-rpath-link example (8)

- at runtime, all the dependencies
  libfoo.so, libbar.so and libfoobar.so
  might not be where they are now
  $(pwd) the current directory

- but the loader might be able to locate them by other means:
  - through the ldconfig cache or
  - a setting of the LD_LIBRARY_PATH environment variable, e.g:

```
$ export LD_LIBRARY_PATH=.; ./prog
foo
bar
```

https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-li

bfd ld and -rpath-link

- The --rpath-link option is used by bfd ld
  to add to the search path used
  for finding DT_NEEDED shared libraries
  when doing link-time symbol resolution
- It's basically telling the linker
  what to use as the runtime search path
  when attempting to mimic
  what the dynamic linker would do
  when resolving symbols
  - as set by --rpath options or
    the LD_LIBRARY_PATH environment variable).

https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-li

Gold ld and -rpath-link

- Gold linker does not follow DT_NEEDED entries
  when resolving symbols in shared libraries,
- so the --rpath-link option is ignored
- this was a deliberate design decision;
  indirect dependencies do not need to be present
  or in their runtime locations during the link process.

https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-li

-rpath examples

Example summary using -rpath

1. Make two shared libraries, libfoo.so and libbar.so:

```
$ gcc -c -Wall -fPIC foo.c bar.c
$ gcc -shared -o libfoo.so foo.o
$ gcc -shared -o libbar.so bar.o
```

2. Make a third shared library, libfoobar.so
   that depends on the first two;

```
$ gcc -c -Wall -fPIC foobar.c
$ gcc -shared -o libfoobar.so foobar.o -L. -lfoo -lbar
```

3. Make an application, prog that depends on libfoobar.so

```
$ gcc -c -Wall main.c
$ gcc -o prog main.o -L. -lfoobar -Wl,-rpath=$(pwd)
```

4. Make prog run

```
$ ./prog
```

https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-li

-rpath example (1)

- rpath=dir
  - provides the linker with the same information as rpath-link=dir does
  - instructs the linker to bake that information into the RPATH entry of the .dynamic section of the output file (here, prog) (run time search path)

```
# to show that this environment variable is not used
$ export LD_LIBRARY_PATH=

$ gcc -o prog main.o -L. -lfoobar -Wl,-rpath=$(pwd)
$ ./prog
foo
bar
```

https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-li

-rpath example (2)

- because now, prog contains the information

```
$(pwd) -->  /home/imk/develop/so/scrap
```

which is a runtime search path for shared libraries
that porg depends on

```
$ readelf -d prog

Dynamic section at offset 0xe08 contains 26 entries:
  Tag        Type                         Name/Value
 0x0000000000000001 (NEEDED)             Shared library: [libfoobar.so]
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
 0x000000000000000f (RPATH)              Library rpath: [/home/imk/develop/so/scrap]
 ...                                     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
 ...
```

https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-li

-rpath example (3)

- the search path of the RPATH will be tried
  - after the directories listed in LD_LIBRARY_PATH,
    if any are set,
  - before the system defaults -
    the ldconfig-ed directories,
    plus /lib and /usr/lib

1. LD_LIBRARY_APTH directoris
2. RPATH directories
3. ldconfig-ed directories

https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-li

-rpath example (4)

- only 'older' versions of gcc have RPATH by default.
- in 'modern' versions of gcc will use RUNPATH by default
  - RUNPATH does not make dependencies inherit the search path
  - it only applies to the search path of the *current binary* (no recursive application)

older gcc	RPATH	dependencies inherit the search path
modern gcc	RUNPATH	search path of the current binary

https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-li

-rpath example (5)

- In this example:
  libfoobar.so will be found at runtime,
  but libfoo.so and libbar.so won't
- -Wl,-rpath=$(pwd) must be specified for libfoobar.so
  - because RUNPATH is used by default (using *modern* gcc)
    which applies to the search path of the current binary only

```
$ export LD_LIBRARY_PATH=

$ gcc -shared -o libfoobar.so foobar.o -L. -lfoo -lbar -Wl,-rpath=$(pwd)
$ gcc -o prog main.o -L. -lfoobar -Wl,-rpath=$(pwd)
$ ./prog
foo
bar
```

https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-li

-rpath example (6)

- if -Wl,--disable-new-dtags is specified
  RPATH is used instead of RUNPATH
  (using older gcc)
  - makes dependencies inherit the search path

```
$ export LD_LIBRARY_PATH=

$ gcc -shared -o libfoobar.so foobar.o -L. -lfoo -lbar
$ gcc -o prog main.o -L. -lfoobar -Wl,-rpath=$(pwd) -Wl,--disable-new-dtags
$ ./prog
foo
bar
```

- then the example is still valid

https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-li

-rpath for shared libraries examples

Example summary using -rpath

1. Make two shared libraries, libfoo.so and libbar.so:

```
$ gcc -c -Wall -fPIC foo.c bar.c
$ gcc -shared -o libfoo.so foo.o
$ gcc -shared -o libbar.so bar.o
```

2. Make a third shared library, libfoobar.so
   that depends on the first two;

```
$ gcc -c -Wall -fPIC foobar.c
$ gcc -shared -o libfoobar.so foobar.o -L. -lfoo -lbar -Wl,-rpath=$(pwd)
```

3. Make an application, prog that depends on libfoobar.so

```
$ gcc -c -Wall main.c
$ gcc -o prog main.o -L. -lfoobar -Wl,-rpath=$(pwd)
```

4. Make prog run

```
$ ./prog
```

https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-li

-rpath for shared libraries (1)

- can use -Wl,-rpath with ELF shared objects
  just like with ELF executables

```
$ gcc -c -Wall -fPIC foo.c bar.c
$ gcc -shared -o libfoo.so foo.o
$ gcc -shared -o libbar.so bar.o

$ gcc -c -Wall -fPIC foobar.c
$ gcc -shared -o libfoobar.so foobar.o -L. -lfoo -lbar -Wl,-rpath=$(pwd)

$ gcc -c -Wall main.c
$ gcc -o prog main.o -L. -lfoobar -Wl,-rpath-link=$(pwd)
```

https://unix.stackexchange.com/questions/571861/is-there-an-rpath-for-dynamic-linl

-rpath for shared libraries (2)

- check what libraries are needed by the main ELF executable:

```
$ readelf -d prog

Dynamic section at offset 0xe08 contains 26 entries:
  Tag        Type                         Name/Value
 0x0000000000000001 (NEEDED)             Shared library: [libfoobar.so]
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
 0x000000000000000f (RPATH)              Library rpath: [/home/imk/develop/so/scrap]
 ...                                     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
 ...
```

https://unix.stackexchange.com/questions/571861/is-there-an-rpath-for-dynamic-linl

-rpath for shared libraries (3)

- what libraries are needed by `libfoobar.so` shared object could be:

```
$ readelf -d ./libtwo.so

Dynamic section at offset 0xe38 contains 22 entries:
  Tag        Type                         Name/Value
   0x0000000000000001 (NEEDED)    Shared library: [libfoo.so]
   0x0000000000000001 (NEEDED)    Shared library: [libbar.so]
   0x0000000000000001 (NEEDED)    Shared library: [libc.so.6]
   0x000000000000000f (RPATH)     Library rpath: [/home/imk/develop/so/scrap]
  (...)
```

https://unix.stackexchange.com/questions/571861/is-there-an-rpath-for-dynamic-linl

-rpath for shared libraries (4)

- both prog executable and libfoobar.so shared object
  need some shared objects
  and have RPATH set by -Wl,-rpath=$(pwd)
- to run prog doses not need
  to set LD_LIBRARY_PATH to run ./prog:

```
$ ./main
foo
bar
```

https://unix.stackexchange.com/questions/571861/is-there-an-rpath-for-dynamic-linl

-rpath for shared libraries (5)

- prog will always look for its libfoobar.so dependency in $(pwd) that is a current directory and
- libtwo.so in turn will always look for its libfoo.so and libbar.so dependencies in $(pwd) directory

https://unix.stackexchange.com/questions/571861/is-there-an-rpath-for-dynamic-linl

Examples of -Wl,-rpath,.

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