Link 5A Library Search using RPATH / RUNPATH

Young W. Lim

2024-09-10 Tue

Outline

1. Based on
2. 1. Background
   - TOC: Background of RPATH / RUNPATH
   - 1. What is RPATH / RUNPATH
   - 2. Using -rpath-link and -rpath
   - 3. Specifying linker option using -W,option
3. 2. Example code
   - 1. Source code and dependencies
   - 2. Single and Multiple Directory Examples
   - 3. Four Methods
   - 4. Summary
4. 3. Single directory case
   - TOC: 2. Example
   - 2. Four methods summary
   - 2. Example using -rpath-link
   - 3. Example using -rpath
5. 4. Multiple directory case

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: Background

1. Using `-rpaht-link` and `-rpath`
2. Using `-Wl,-rpath,.`

TOC: What is `RPATH / RUNPATH`

- What is `RPATH / RUNPATH`
- Dependency related entries of the `.dynamic` section

What is RPATH / RUNPATH (1)

- if an executable foo links to the shared library bar, the library bar has to be *found* and *loaded* when the executable foo is *executed*.
- this searching and loading the shared library is done by the linker, ld.so.
- the linker searches a set of directories for the library bar, i.e., libbar.so

https://gitlab.kitware.com/cmake/community/-/wikis/doc/cmake/RPATH-handling

What is RPATH / RUNPATH (2)

- The linker will search the shared libraries in the following directories in the given order:
  1. RPATH (deprecated)
  2. LD_LIBRARY_PATH
  3. RUNPATH (only direct dependency paths are searched)
  4. /etc/ld.so.conf

https://gitlab.kitware.com/cmake/community/-/wikis/doc/cmake/RPATH-handling

What is RPATH / RUNPATH (3)

1. RPATH
   - a list of directories which is linked into the executable
   - ignored if RUNPATH is present (RPATH is deprecated)
2. LD_LIBRARY_PATH
   - an environment variable which holds a list of directories
3. RUNPATH
   - same as RPATH, but searched after LD_LIBRARY_PATH, supported only on most current Linux systems
4. /etc/ld.so.conf
   - configuration file for ld.so which lists
     additional library directories (builtin directories)
     basically /lib and /usr/lib

https://gitlab.kitware.com/cmake/community/-/wikis/doc/cmake/RPATH-handling

What is RPATH / RUNPATH (4)

- different reasons for needs for other directories to be searched than the builtin ones
  1. a user may install a library *privately* into his *home directory*, e.g. `~/lib/`
  2. there may be different *versions* of the same library installed, e.g. `/opt/kde3/lib/libkdecore.so` and `/opt/kde4/lib/libkdecore.so`

`https://gitlab.kitware.com/cmake/community/-/wikis/doc/cmake/RPATH-handling`

What is RPATH / RUNPATH (5)

1. a user may install a library *privately*
   into his *home directory*, e.g. ~/lib/

   - in this case, LD_LIBRARY_PATH can be set
     export LD_LIBRARY_PATH=$HOME/lib:$LD_LIBRARY_PATH

https://gitlab.kitware.com/cmake/community/-/wikis/doc/cmake/RPATH-handling

What is RPATH / RUNPATH (6)

2. there may be different *versions* of the same library installed, e.g. /opt/kde3/lib/libkdecore.so and /opt/kde4/lib/libkdecore.so

- cases for some programs /opt/kde3/lib has to be searched and for other applications /opt/kde4/lib has to be searched, but never both directories
- the only way to have an executable-dependent library search path is by using RPATH (deprecated) or RUNPATH (not always supported)

https://gitlab.kitware.com/cmake/community/-/wikis/doc/cmake/RPATH-handling

Dependency related entries of the .dynamic section

DT_NEEDED	• created by -L -l options of gcc compiler • specifies direct dependencies • can be used to find nested dependencies
DT_RPATH / DT_RUNPATH	• created by -rpath option of ld linker • specifies runtime search path • DT_RPATH is deprecated → searches direct and nested dependency paths • DT_RUNPATH is not supported by all systems → searches only direct dependency paths

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

RPATH v.s. RUNPATH (1)

- in the `.dynamic` section of a binary (*executable* or *shared library*)
  - the `RPATH` entry is used by default
    in the *older* versions of gcc
    - `RPATH` allows nested dependencies
      to inherit the specified search path
  - the `RUNPATH` entry is used by default
    in *modern* versions of gcc
    - `RUNPATH` applies the search path
      only to the direct dependencies
      of the *current binary* (no recursive application)

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

RPATH v.s. RUNPATH (2)

older gcc	RPATH	all dependencies (direct, nested) utilize the specified path
modern gcc	RUNPATH	only direct dependencies utilize the specified path

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

TOC: What is RPATH / RUNPATH

- Handling *direct* and *nested* dependencies
- Specifying -L and -l handles *direct* dependencies
- Specifying -rpath-link handles *nested* dependencies
- -rpath-link v.s. -rpath
- -rpath-link does not create RUNPATH / RPATH entries
- -rpath creates RUNPATH / RPATH entries
- -rpath-link in bfd and gold linkers
- bfd ld and -rpath-link
- gold ld and -rpath-link

Handling *direct* and *nested* dependencies

- *direct* dependecy must be handled
  by specifying `-L` and `-l`
- *nested* dependecies can be handled
  by specifying `-rpath-link` or `-rpath`

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

Specifying -L and -l handles *direct* dependencies

- the *direct* dependencies of the current binary must be handled by -L and -l
  - specifying -L and -l creates NEEDED entries in .dynamic section of the current binary
  - by specifying -rpath-link or -rpath
    - the NEEDED entries are not created, but
    - the NEEDED entries of each binary can be utilized to find the *nested* dependencies of a given binary

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

Specifying -rpath-link handles *nested* dependencies

- the -rpath-link=dir option tells the linker (ld) that when *dynamic nested* dependencies are requested, directory dir is searched to *resolve* them.

- only for a successful linkage, -rpath-link specifies the *directories* where the *nested* dependencies of the current binary can be found

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

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

-rpath-link v.s. -rpath

- -rpath-link=dir
  - provides the linker with runtime search path information
  - but does not instruct the linker to write that information into RUNPATH or RPATH entries in the .dynamic section
- -rpath=dir
  - also provides the linker with runtime search path information
  - and instructs the linker to write that information into RUNPATH or RPATH entries in the .dynamic section

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

-rpath-link makes only a successful linkage

- -rpath-link=dir
  - does not guarantee us a *runnable prog* but only a *successful linkage*

```
$ 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://unix.stackexchange.com/questions/22926/where-do-executables-look-for-share
https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-li

-rpath-link does not create RUNPATH / RPATH entries

- there are many other ways to specify the runtime search path
- `-rpath-link=dir` does not give any information of runtime search path
  - does not creates RUNPATH
  - does not creates RPATH
  - therefore, for a *successful execution*, explicit specification of runtime search path may be needed.

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

-rpath does create RUNPATH / RPATH entries

- -rpath=dir
  - creates RUNPATH or RPATH entries
    in the .dynamic section
    to specify runtime search path
    - RUNPATH (for modern gcc)
    - RPATH (for older gcc)
  - guarantees us a *runnable prog*
  - no need to specify runtime search path explicitly

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

-rpath-link in bfd and gold linkers

	bfd ld	gold ld
-rpath-link	(O)	(X) ignored
DT_NEEDED	(0)	(X) not used

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

bfd ld and -rpath-link (1)

- The --rpath-link option is used by bfd ld
  to add to the search path used for
  finding DT_NEEDED shared libraries

  (direct dependencies of a given binary)

  when doing link-time symbol resolution

    - by following DT_NEEDED entries recursively
      indirect (nested) dependencies can be found

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

bfd ld and -rpath-link (2)

- 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 the runtime search path set
  by --rpath options or
  the LD_LIBRARY_PATH environment variable

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

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 when gold linker is used
- this was a deliberate design decision; indirect (nested) 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

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

TOC: 1. Example source code and dependencies

- Example source codes of `foo()`, `bar()`, `foobar()`
- Function dependencies of `foo()`, `bar()`, `foobar()`

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-l

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

main()	→	foobar()	→	foo()
prog		libfoobar.so		libfoo.so
			→	bar()
				libbar.so

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

direct and nested dependencies of a binary

binary	direct dependencies	nested dependencies
libfoobar.so	$\rightarrow$ libfoo.so, $\rightarrow$ libbar.so	
prog	$\rightarrow$ libfoobar.so	$\rightarrow$ libfoo.so, $\rightarrow$ libbar.so

Single Directories for example binaries

binaries	compile time	run time A	run time B
`libfoo.so`	.	.	.
`libbar.so`	.	.	.
`libfoobar.so`	.	.	.
`prog`	.	.	.

- all binaries are in the current directory .

Multiple Directories for example binaries (1)

binaries	compile time	run time A	run time B
libfoo.so	./lib2	./lib2	./librun
libbar.so	./lib2	./lib2	./librun
libfoobar.so	./lib	./librun	./librun
prog	.	.	.

Multiple Directories for example binaries (2)

directories	compile time	run time A	run time B
./lib2	libfoo.so libbar.so	libfoo.so libbar.so	
./lib	libfoobar.so		
./librun		libfoobar.so	libfoo.so libbar.so libfoobar.so
.	prog	prog	prog

Four methods

- Method 1. using -L and -l
- Method 2. using -rpath-link
- Method 3. using -rpath (like using -rpath-link)
- Method 4. using -rpath (using RUNPATH)

	for direct dependencies	for nested dependencies
Method 1	-L *d_direct* -l *direct*	-L *d_nest* -l *nest*
Method 2	-L *d_direct* -l *direct*	-rpath-link *d_nest*
Method 3	-L *d_direct* -l *direct*	-rpath *d_nest*
Method 4	-L *d_direct* -l *direct*	-rpath *d_direct*

TOC: 5. Summary

Specifying dependencies and search paths (1)

	dependencies	link time search paths	runtime search paths
-l	○		
-L		○	
-rpath-link		○	
-rpath		○	○

Specifying dependencies and search paths (2)

	for direct dependencies	for nested dependencies
Method 1	-L *d_direct* -l *direct*	-L *d_nest* -l *nest*
Method 2	-L *d_direct* -l *direct*	-rpath-link *d_nest*
Method 3	-L *d_direct* -l *direct*	-rpath *d_nest*
Method 4	-L *d_direct* -l *direct*	-rpath *d_direct*

Specifying dependencies and search paths (3)

Method 1	-L *d_direct* -l *direct* -L *d_nest* -l *nest*
Method 2	-L *d_direct* -l *direct* -rpath-link *d_nest*
Method 3	-L *d_direct* -l *direct* -rpath *d_nest*

need to specify *runtime* search paths, e.g.,
export LD_LIBRARY_PATH=dir1:dir2

Method 4	-L *d_direct* -l *direct* -rpath *d_direct*

no need to specify *runtime* search paths
-rpath enables each binary to *record*
its *direct* search paths in the RUNPATH entry
of its .dynamic section

NEEDED entries of each binary

binary	dependencies	entry	section
prog	libfoobar.so	NEEDED	.dynamic
libfoobar.so	libfoo.so, libbar.so	NEEDED	.dynamic

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

TOC: 2. Example

1. Example source code and dependencies
2. `-rpath-link` examples
3. `-rpath` examples

Ex1 M1 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 prog 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-l

Ex1 M2 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

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

3. Make prog that depends on libfoobar.so

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

4. Execute using LD_LIBRARY_PATH

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

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

Ex1 M4 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

```
# to show that this environment variable is not used
export LD_LIBRARY_PATH=    # clear the env variable
$ ./prog
```

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

NEEDED entries and nested dependencies (2)

- -rpath-link=dir
  - the *nested* dependencies of prog can be found through the NEEDED entries in the .dynamic section of the *direct* dependency of prog
    - when prog was made, its *direct* dependency were specified with -lfoobar
    - the *direct* dependencies of libfoobar.so can be found by looking the NEEDED entries in the .dyanmic section of libfoobar.so
  - the directory dir will be searched for these *nested* dependencies of prog

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

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

Using LD_LIBRARY_PATH to specify a runtime search path

- but the loader might be able to locate them
  - 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-l

-rpath-link example (6)

- -rpath-link=dir
  - gives the linker (ld) the directory information that the loader (ld.so) *would* need to resolve some of the dynamic dependencies of prog at runtime
    - assuming that the directory information remained true at runtime
  - but does not write that directory information into the .dynamic section of prog
    - only the *direct* dependency (libfoobar.so) is written in the .dynamic section of prog

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

-rpath example (4)

- prog contains the runtime search path information for shared libraries that porg depends on

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

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


$ 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 (RUNPATH)            Library rpath: [/home/imk/develop/so/scrap]
 ...                                     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
 ...
```

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

-rpath example (5)

- libfoobar.so (direct dependency) will be found at runtime, but libfoo.so and libbar.so (nested dependencies) won't,
  - because libfoobar.so does not inherit RUNPATH information of prog
- -rpath=$(pwd) must be specified also for libfoobar.so to write *runtime search path* information into RUNPATH entry of the .dynamic section of libfoobar.so

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

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

-rpath example (6)

- check what libraries are needed by libfoobar.so could be:

```
$ readelf -d ./libfoobar.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-lin

-rpath example (7)

- prog executable depends on libfoobar.so shared object
  RUNPATH entry of .dynamic section of prog set by

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

- libfoobar.so shared object depends on
  libfoo.so and libbar.so shared objects
  RUNPATH entry of .dynamic section of libfoobar.so set by

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

- to run prog doses not need to set LD_LIBRARY_PATH

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

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

-rpath example (8*)

- RPATH is searched in before LD_LIBRARY_PATH
- RUNPATH is searched in after LD_LIBRARY_PATH

1. search RPATH (older versions of gcc)
2. search LD_LIBRARY_PATH
3. search RUNPATH (modern versions of gcc)
4. search ldconfig-ed directories

https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-li
https://refspecs.linuxbase.org/elf/gabi4+/ch5.dynamic.html

-rpath example (9*)

- if `-Wl,--disable-new-dtags` is specified
  RPATH is used as if 'older versions' of gcc were used,
  instead of RUNPATH
  - makes *nested* dependencies inherit the specified search path
  - thus, `-rpath=$(pwd)` need not be specified for `libfoobar.so`

```
$ 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
```

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

TOC: 3. More Examples

1. More Exammples

TOC: 5. More Examples

Specifying dependencies and search paths (1)

binaries	*d_direct*	*direct*	*d_nest*	*nest*
libfoobar.so	lib2	foo bar		-
prog	lib	foobar	lib2	foo bar

d_direct	directories for direct dependencies	*direct*	direct dependencies
d_nest	directories for nested dependencies	*nest*	nested dependencies

Specifying dependencies and search paths (2)

- for libfoobar.so

	for direct dependencies	for nested dependencies
Method 1.	-Llib2 -lfoo -lbar	
Method 2.	-Llib2 -lfoo -lbar	
Method 3.	-Llib2 -lfoo -lbar	
Method 4.	-Llib2 -lfoo -lbar	-Wl,-rpath=lib:librun

- for prog

	for direct dependencies	for nested dependencies
Method 1.	-Llib -lfoobar	-Llib2 -lfoo -lbar
Method 2.	-Llib -lfoobar	-Wl,-rpath-link=lib2
Method 3.	-Llib -lfoobar	-Wl,-rpath=lib2
Method 4.	-Llib -lfoobar	-Wl,-rpath=lib:librun

Using -rpath-link=dir for dependencies

- when `rpath-link` or `rpath` is used
  - specify only *direct* dependencies using `-l` and their search paths with `-L`
  - no need to specify *nested* dependencies
    - *nested* dependencies can be found by the `NEEDED` entry in the `.dynamic` section of a given *direct* dependency
  - `-lfoobar` necessary
  - `-lfoo -lbar` unnecessary

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

  - the *direct* dependency of `prog` : `libfoobar.so`
  - the *nested* dependencies of `prog` : `libfoo.so`, `libbar.so` (the *direct* dependencies of `libfoobar.so`)

Using `-rpath-link=dir` for link time search paths

- when `-rpath-link=dir` is used
  - since *nested* dependencies do *inherit* the search path
  - specify all the search paths for *direct* and *nested* dependencies using `rpath-link=dir1:dir2` or multiple `rpath-link` options
  - only for a successful linkage, not for a successful execution
  - in this example, to link successfully, `$(pwd)` is searched
    - for `libfoobar.so` (the *direct* dependency)
    - for `libfoo.so` and `libbar.so` (the *nested* dependencies)

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

Using -rpath=dir for dependencies

- when rpath-link or rpath is used
  - specify only *direct* dependencies using -l
    and their search paths with -L
  - no need to specify *nested* dependencies
    - *nested* dependencies can be found
      by the NEEDED entry in the .dynamic section
      of a given *direct* dependency
  - -lfoobar necessary
  - -lfoo -lbar unnecessary

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

  - the *direct* dependency of prog : libfoobar.so
  - the *nested* dependencies of prog : libfoo.so, libbar.so
    (the *direct* dependencies of libfoobar.so)

Using -rpath=dir for link time search paths

- when `-rpath` is used, there are two approaches for specifying the *link time* search paths

  1. specify *all* the search paths for *direct* and *nested* dependencies of a given binary using `-rpath`
     - for a successful linkage only, not for a successful execution
     - since *nested* dependencies *inherit* the search path
     - as long as specifying *link time* search paths are concerned, the `rpath` option is the same as the `rpath-link` option
  2. let each binary be specified with search paths using `-rpath` for its *direct* dependencies only
     - those paths are recorded as *runtime* search paths in the `RUNPATH` entry of `.dynamic` section of a binary

Using -rpath=dir for run time search paths

- -rpath=dir
  - the ld *searches* directory dir to *resolve* references
  - the ld.so *searches* directory dir to *load* shared libraries

  - to load shared libraries, *nested* dependencies may not *inherit* the search path

  - for modern versions of gcc that use RUNPATH instead RPATH do not allow the search path to be *inherited*
    - thus, each binary should be specified with search paths for its *direct* dependencies, using -rpath
    - that those paths may be recorded as *runtime* search path in the RUNPATH entry of .dynamic section of the binary

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

Example2 summary using -L and -l

1. Make `libfoo.so` and `libbar.so` in `./lib2`

```
$ gcc -c -Wall -fPIC foo.c bar.c
$ gcc -shared -o libfoo.so foo.o
$ gcc -shared -o libbar.so bar.o
$ mv libfoo.so libbar.so lib2
```

2. Make `libfoobar.so` in `./lib`

```
$ gcc -c -Wall -fPIC foobar.c
$ gcc -shared -o libfoobar.so foobar.o -Llib2 -lfoo -lbar
$ mv libfoobar.so lib
```

3. Make `prog` in `.`

```
$ gcc -c -Wall main.c
$ gcc -o prog main.o -Llib -lfoobar -Llib2 -lfoo -lbar
```

4. Execute using LD_LIBRARY_PATH (libraries in `librun`, `lib2`)

```
$ mv lib/libfoobar.so librun
$ export LD_LIBRARY_PATH=librun:lib2
$ ./prog
```

Example2 summary using -rpath-link

1. Make libfoo.so and libbar.so in ./lib2

```
gcc -c -Wall -fPIC foo.c bar.c
gcc -shared -o libfoo.so foo.o
gcc -shared -o libbar.so bar.o
mv libfoo.so libbar.so lib2
```

2. Make y, libfoobar.so in ./lib

```
gcc -c -Wall -fPIC foobar.c
gcc -shared -o libfoobar.so foobar.o -Llib2 -lfoo -lbar
mv libfoobar.so lib
```

3. Make prog in .

```
gcc -c -Wall main.c
gcc -o prog main.o -Llib -lfoobar -Wl,-rpath-link=lib2
```

4. Execute using LD_LIBRARY_PATH (libraries in librun, lib2)

```
mv lib/libfoobar.so librun
export LD_LIBRARY_PATH=librun:lib
./prog
```

Example2 summary using -rpath (like using -rpath-link)

1. Make libfoo.so and libbar.so in ./lib2

```
gcc -c -Wall -fPIC foo.c bar.c
gcc -shared -o libfoo.so foo.o
gcc -shared -o libbar.so bar.o
mv libfoo.so libbar.so lib2
```

2. Make libfoobar.so in ./lib

```
gcc -c -Wall -fPIC foobar.c
gcc -shared -o libfoobar.so foobar.o -Llib2 -lfoo -lbar
mv libfoobar.so lib
```

3. Make prog in .

```
gcc -c -Wall main.c
gcc -o prog main.o -Llib -lfoobar -Wl,-rpath=lib2
```

4. Execute using LD_LIBRARY_PATH (libraries in librun, lib2)

```
mv lib/libfoobar.so librun
export LD_LIBRARY_PATH=librun:lib
./prog
```

Example2 summary using -rpath (using RUNPATH)

1. Make libfoo.so and libbar.so in ./lib2

```
gcc -c -Wall -fPIC foo.c bar.c
gcc -shared -o libfoo.so foo.o
gcc -shared -o libbar.so bar.o
mv libfoo.so libbar.so lib2
```

2. Make libfoobar.so in ./lib

```
gcc -c -Wall -fPIC foobar.c
gcc -shared -o libfoobar.so foobar.o -Llib2 -lfoo -lbar -Wl,-rpath=lib:librun
mv libfoobar.so lib
```

3. Make prog in .

```
gcc -c -Wall main.c
gcc -o prog main.o -Llib -lfoobar -Wl,-rpath=lib2:librun
```

4. Execute without LD_LIBRARY_PATH (now all libraries in librun)

```
mv lib/libfoobar.so lib2/libfoo.so lib2/libbar.so librun
export  LD_LIBRARY_PATH=
./prog
```