

Link 4A Library Search using -L and -I only

Young W. Lim

2024-07-22 Mon

- 1 Based on
- 2 Search libraries using `-L` and `-l` only
 - TOC: Search libraries using `-L` and `-l` only
 - 1. Example source code and dependencies
 - 2. Making shared libraries
 - 3. Making an application
 - 4. Running an application

"Study of ELF loading and relocs", 1999

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

I, the copyright holder of this work, hereby publish it under the following licenses: GNU head Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled GNU Free Documentation License.

CC BY SA This file is licensed under the Creative Commons Attribution ShareAlike 3.0 Unported License. In short: you are free to share and make derivative works of the file under the conditions that you appropriately attribute it, and that you distribute it only under a license compatible with this one.

Compiling 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: Search libraries using `-L` and `-l` only

- 1 Example source code and dependencies
- 2 Making shared libraries
- 3 Making an application
- 4 Running an application

TOC: 1. Example source code and dependencies

- Example source codes
- Function dependencies
- Direct and nested dependencies of a binary
- Example summary using `-L` and `-l`

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-1>

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

<code>main()</code>	<code>→</code>	<code>foobar()</code>
<code>foobar()</code>	<code>→</code>	<code>foo(), bar()</code>

<code>main()</code>	in	<code>prog</code>
<code>foobar()</code>	in	<code>libfoobar.so</code>
<code>foo()</code>	in	<code>libfoo.so</code>
<code>bar()</code>	in	<code>libbar.so</code>

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

Direct and nested dependencies of a binary

binary	direct dependencies	nested dependencies
<code>libfoobar.so</code>	<code>→ libfoo.so,</code> <code>→ libbar.so</code>	
<code>prog</code>	<code>→ libfoobar.so</code>	<code>→ libfoo.so,</code> <code>→ libbar.so</code>

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 `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>

TOC: 2. Making shared libraries

- Making `libfoo.so`, `libbar.so`
- Using `-L`
- Making `libfoobar.so`

Making libfoo.so and libbar.so (1) summary

- 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
```

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

Making libfoo.so and libbar.so (2) no dependencies

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

- neither foo() nor bar() does depend on other user functions
- no need to specify *direct* dependencies
thus, `-l` was not used
- therefore, no **NEEDED** entries in the `.dynamic` section
for *direct* dependencies that are specified by a user

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

Making libfoo.so and libbar.so (3) NEEDED entries

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

- `libc.so.6` was not explicitly specified by a user via `-l`

```
$ readelf -d libfoo.so | grep NEEDED
Tag                Type                Name/Value
0x0000000000000001 (NEEDED)    Shared library: [libc.so.6]
```

```
$ readelf -d libbar.so | grep NEEDED
Tag                Type                Name/Value
0x0000000000000001 (NEEDED)    Shared library: [libc.so.6]
```

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

Using -L

- the `-L` option (`-Ldir`) tells the linker (`ld`) to search `dir` for libraries to resolve dependencies that are specified by the `-l` option
- the linker (`ld`) searches the `-L` directories, in their command line order;
 - eg. when multiple `-L` options are used like `-Ldir1 -Ldir2`
`dir1` is searched first, then `dir2`
- then it searches its configured default directories, in their configured order.

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

Making libfoobar.so (1) summary

- Make a third shared library, `libfoobar.so` that depends on the first two (`libfoo.so`, `libbar.so`)

```
$ gcc -c -Wall -fPIC foobar.c
```

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

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

Making libfoobar.so (2) NEEDED entries

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

- *direct dependencies* were specified by `-lfoo -lbar`
- these *dependencies* were recorded as the **NEEDED** entries in the `.dynamic` section of `libfoobar.so`

```
$ readelf -d libfoobar.so | grep NEEDED
Tag                Type                Name/Value
0x0000000000000001 (NEEDED)    Shared library: [libfoo.so] <---
0x0000000000000001 (NEEDED)    Shared library: [libbar.so] <---
...                  ...                  ...
```

<https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-1>

Making libfoobar.so (3) if -L. is omitted

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

- if `-lfoo` and `-lbar` are specified without `-L.` being specified,
 - *direct dependencies* (`libfoo.so` and `libbar.so`) were specified
 - but where to find the necessary libraries (the current directory) was not specified

<https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-1>

Making libfoobar.so (4) error messages

```
$ 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
```

- if `-L.` is not specified, error messages is displayed
- saying that the **direct dependency** libraries (`libfoo.so` and `-libbar.so`) could not be located
- the linker (`ld`) didn't know where to look to *resolve* `-lfoo` or `-lbar` thus were not able to *resolve* them

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

TOC: 3. Making an application

- Making an application prog that uses `libfooba.so`
 - Not specifying nested dependencies
 - Warning and error messages
 - Using `-L` and `-l` to make an application

Making an application prog that uses libfoobar.so (1)

- make a program `prog` that depends on `libfoobar.so`:

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

- `libfoo.so` and `libbar.so` are the *direct* dependencies of `libfoobar.so`, and thus the *nested* dependencies of `prog`
- only *direct* dependency is specified (`-lfoobar`) with the correct search path (`-L.`)
- *nested* dependencies are not specified (`-lfoo -lbar`) but `libfoo.so` and `libbar.so` can be found in the specified search path (`-L.`)

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

Making an application prog that uses libfoobar.so (2)

- make a program `prog` that depends on `libfoobar.so`:

```
$ gcc -c -Wall main.c
```

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

- only *direct* dependency was specified (`-lfoobar`) with the correct search path (`-L.`)
- *nested* dependencies were not specified (`-lfoo -lbar`) but can be handled by `-rpath-link=$(pwd)`
 - `libfoo.so` and `libbar.so` are the *direct* dependencies of `libfoobar.so`, and thus the *nested* dependencies of `prog`

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

Creating **NEEDED** entries

- make a program **prog** that depends on **libfoobar.so**:

```
$ gcc -c -Wall main.c
```

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

- in the **.dynamic** section of **prog**
 - *direct dependency* specified by **-lfoobar** was recorded as **NEEDED** entries
 - *nested dependency*, even though specified by **-lfoo -lbar**, are not recorded as **NEEDED** entries

```
$ readelf -d prog | grep NEEDED
```

Tag	Type	Name/Value
0x0000000000000001	(NEEDED)	Shared library: [libfoobar.so] <---
0x0000000000000001	(NEEDED)	Shared library: [libc.so.6]

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

NEEDED entries and nested dependencies (1)

- `libfoo.so`, `libbar.so` :
 - these are the *direct dependencies* of `libfoobar.so`
 - thus, these are the *nested dependencies* of `prog`
 - when `libfoobar.so` was made, its *direct dependencies* were specified with `-lfoo -lbar`
 - this allows the *direct dependencies* of `libfoobar.so` to be recorded as **NEEDED** entries in the `.dynamic` section of `libfoobar.so`

<https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-1>

Not specifying *nested* dependencies

- although `-lfoo` and `-lbar` are not specified,

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

- by looking into `NEEDED` entry of the `.dynamic` section of `libfoobar.so`,
- the linker (`ld`) detects the *nested dynamic dependencies* but they were not specified with `-lfoo -lbar`
warning : not found libfoo.so, not found libbar.so
- the linker (`ld`) did not resolve the *nested* dependencies because they were not specified
error: undefined reference to foo, undefined reference to bar

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

Warning and error messages

- make a program `prog` that depends on `libfoobar.so`:
 - the *nested* dependencies are not specified (`-lfoo -lbar`) though with the correct search path (`-L.`)
 - not found `libfoo.so` ← `-lfoo` not specified
 - not found `libbar.so` ← `-lbar` not specified
 - undefined reference to `bar` ← `-lbar` not resolved
 - undefined reference to `foo` ← `-lfoo` not resolved

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

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

Using only `-L` and `-l` to make an application

- to resolve the *nested dependencies*, we will consider the following ways
 - 1 `-L` and `-l`
 - 2 `-rpath-link`
 - 3 `-rpath`
- let us first ignore the gcc compiler's advice
try using `-rpath` or `-rpath-link`
- to handle *nested dependencies*, try first using `-L` and `-l`
 - search path for *nested dependencies* : `-L.`
(the same directory specified for `libfoobar.so`)
 - *nested dependencies* : `-lfoo -lbar`

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

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

TOC: 4. Running an application

- Need to specify runtime search paths
- More experiment with nested dependencies
- Specifying the runtime shared library paths
- Using `LD_LIBRARY_PATH` to run an application

Need to specify runtime search paths

- now, the application `prog` can be made, but cannot be made to run:

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

```
$ ./prog
```

```
./prog: error while loading shared libraries: libfoobar.so:\ncannot open shared object file: No such file or directory
```

- at the `runtime`, the loader (`ld.so`) could not find `libfoobar.so` nor `libfoo.so` nor `libbar.so`
- need to specify the `runtime` search paths

<https://stackoverflow.com/questions/49138195/whats-the-difference-between-rpath-1>

More experiment with nested dependencies

- before specifying runtime search paths, let's experiment more with *nested dependencies*
- move `libfoo.so` and `libbar.so` libraries to `lib2`

```
$ mkdir lib2
$ mv libfoo.so libbar.so lib2
```

- then, make `prog` as before

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

- the *nested dependencies* were specified (`-lfoo -lbar`)
- but the linker (`ld`) could not find `libfoo.so` and `libbar.so` at the specified directory (`-L.`)

```
/usr/bin/ld: cannot find -lfoo
/usr/bin/ld: cannot find -lbar
collect2: error: ld returned 1 exit status
```

- the correct search path `-Llib2` must also be specified

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

Specifying the runtime shared library paths

- now move `libfoo.so`, `libbar.so` back to the current directory . and make `prog` again

```
mv lib2/libfoo.so lib2/libbar.so .  
$ gcc -o prog main.o -L. -lfoobar -lfoo -lbar
```

- the `-L` option is used to tell the linker (`ld`) where to *find the libraries* (shared objects) at the compile, and link time
- lots of ways to tell the runtime linker (dynamic loader `ld.so`) where to *find the libraries* (shared objects) at the runtime
 - `-R`
 - `LD_LIBRARY_PATH`
 - `LD_RUN_PATH`

<https://stackoverflow.com/questions/31455979/how-to-specify-libraries-paths-in-gcc>

Using LD_LIBRARY_PATH to run an application

- `prog` is made by using `-L` and `-l` only
not by using `-rpath` nor `-rpath-link`

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

- `prog` is made run by us `LD_LIBRARY_PATH`

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

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

```
export LD_LIBRARY_PATH=.
```

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

NEEDED entries of each binary

binary	dependencies	entry	section
<code>prog</code>	<code>libfoobar.so</code>	NEEDED	<code>.dynamic</code>
<code>libfoobar.so</code>	<code>libfoo.so,</code> <code>libbar.so</code>	NEEDED	<code>.dynamic</code>

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