Link 4. Search Libararies

Young W. Lim

2023-04-22 Sat

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

Compile time and run time (1)

1. the compile-time linking
   gcc and ld
1. run-time linker lookups
   generally ld.so (/lib64/ld-linux-x86-64.so)

https://stackoverflow.com/questions/1904990/what-is-the-difference-between-ld-libr

Compile time and run time (2)

- when you compile your program,
  the compiler checks syntax, and then the linker ensures that
  the symbols required for execution *exist*
  (i.e *variables*, *methods* etc)

- when you run your program,
  the run-time linker
  - actually fetches the shared libraries
  - loads in the shared symbols / code / etc.

https://stackoverflow.com/questions/1904990/what-is-the-difference-between-ld-libr

(1) Linking

- for linking, make sure you specify
  - object files (or source files) before libraries (`-lxxx` options)
  - `-L` option for a given library before the `-l` option
- the order of libraries can matter
  - libraries listed earlier can be referenced in those listed later
  - avoid circular references between libraries

```
$ gcc imagefilter.c -o imagefilter -I/home/savio/opencv-3.0.0/include/opencv \
>     -L/home/savio/opencv-3.0.0/cmake_binary_dir/lib \
>     -lopencv_imgcodecs -lopencv_imgproc -lopencv_highgui -lopencv_core
```

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

(2) Running

- both the compiler / linker and the runtime system need to be able to *find* the shared objects
- the `-L` option is used to tell the linker where to find the libraries (shared objects)
- lots of ways of telling the runtime (dynamic loader) where to find the libraries (shared objects)
  `-R`, `LD_LIBRARY_PATH`, `LD_RUN_PATH`

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

Specifying library paths (3) -R

- On some systems,
  a -R option can be added to the command line
  to specify where libraries (shared objects)
  may be found at runtime :
- not all systems support this option.

```
$ gcc imagefilter.c -o imagefilter -I/home/savio/opencv-3.0.0/include/opencv \
>     -L/home/savio/opencv-3.0.0/cmake_binary_dir/lib \
>     -R/home/savio/opencv-3.0.0/cmake_binary_dir/lib \
>     -lopencv_imgcodecs -lopencv_imgproc -lopencv_highgui -lopencv_core
```

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

Specifying library paths (4) -R

- the disadvantage of this -R option is
  that the location you specify is embedded in the binary.
  - If the libraries on the customers' machines
    is not in the same place, the library won't be found.
  - Consequently, a path under someone's home directory
    is only appropriate for that user on their machines
  - not general
    if the software is installed by default
    in, say, /opt/packagename/lib,
    then specifying that with -R is probably appropriate.

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

Specifying library paths (5) LD_LIBRARY_PATH

- Add the directory
  to LD_LIBRARY_PATH environment variable or its equivalent

```
LD_LIBRARY_PATH=/home/savio/opencv-3.0.0/cmake_binary_dir/lib\
:$LD_LIBRARY_PATH ./imagefilter
```

or:

```
export LD_LIBRARY_PATH=/home/savio/opencv-3.0.0/cmake_binary_dir/lib\
:$LD_LIBRARY_PATH ./imagefilter
```

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

Specifying library paths (6) LD_LIBRARY_PATH

- The first notation sets the environment variable
  just for as long as *the program is running*
  - useful if you need to compare
    the behaviour of two versions of a library, for example.

```
LD_LIBRARY_PATH=/home/savio/.../lib:$LD_LIBRARY_PATH ./imagefilter
```

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

Specifying library paths (7) LD_LIBRARY_PATH

- The second notation sets the environment variable for *the session*.
  - might include that in your `.profile` or equivalent so it applies to every session.

```
export LD_LIBRARY_PATH=/home/savio/.../lib:$LD_LIBRARY_PATH ./imagefilter
```

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

Specifying library paths (8) LD_RUN_PATH

- Some systems have an LD_RUN_PATH environment variable too.
  - some have 32-bit and 64-bit variants
  - fiddly for users and installers alike;
  - how do you *ensure* the environment variable is set for everyone that uses your code?
  - an environment-setting shell script that then runs the real program can help here.

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

Specifying library paths (9) LD_RUN_PATH

- add the directory to the configuration file
  that specifies the list of known directories
  for the dynamic loader to search.
- platform specific
  - file name, format, location (usually under /etc somewhere)
    and mechanism used to edit it.
  - the file might be /etc/ld.so.conf.
  - there might well be a program to edit the config file correctly.

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

Specifying library paths (10)

- install the libraries in a location
  that will be searched anyway
  (without reconfiguring the dynamic loader).
- this might be /usr/lib, or maybe /usr/local/lib
  or some other related directory.

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

Specifying library paths (11)

- The way my IDE handles the process is
  to put the `-L` tag up front and
  the `-l` tag at the end
- all of the `-l` tags need to come after your target
  so that the compiler knows which symbols need
  to be resolved before searching.

```
gcc -L/path/to/library -o target_here -lfirst -lsecond -lthird ...
```

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

-l and -L (1)

- linking is done by two different instances of *linker*
  - when you *compile* and *link* your program
    linker `/usr/bin/ld`
    - checks external references
    - builds your executable
      by adding external reference `libdemo.so`
  - when you *run* your program
    run-time linker `ld.so`
    (`/lib64/ld-linux-x86-64.so.2`)
    - loads all needed *shared objects*

`https://stackoverflow.com/questions/28230983/gcc-l-command-confusion`

-l and -L (2)

- assume
  - `libdemo.so` : a shared library file
  - `ld.so` : the final linker
- the reasons why `-L` *path* is not saved
  - `libdemo.so` is not necessarily located at the same path where it was compiled
    - you could copy your binary unto another host
    - that path was internal build path, etc
  - it may be unsafe to save `-L` *path*
    - `ld.so` ususally seeks over list of trusted paths where non-root users cannot write

`https://stackoverflow.com/questions/28230983/gcc-l-command-confusion`

-l and -L (3)

- since the executable file
  does not *contains* copies of the shared object files,
  it needs some way to *identify* the *necessary* shared library
  - during the link, only the name of the shared library
    is embedded in the executable
    but the specific location is not yet specified.
  - So the `-L. -ldemo` is really just
    to provide the name of the library file and the location

    `libdemo.so` and `.`

`https://stackoverflow.com/questions/28230983/gcc-l-command-confusion`

-l and -L (4)

- `-Ldir` adds directory `dir`
  to the list of directories
  to be searched for `-l`

- `-ldemo` is only to provide
  the name of the library file

`https://stackoverflow.com/questions/28230983/gcc-l-command-confusion`

-l and -L (5)

- `-L. -ldemo` is not required when using the `-rpath`
  - because in `-rpath dir` command,
    the name of the library `libdemo.so` is passed directly
  - otherwise specifying it with `-L. -ldemo` was necessary.
- The run-time library path is subsequently provided
  to specify the exact location at the time of execution

`https://stackoverflow.com/questions/28230983/gcc-l-command-confusion`

-l and -L (6)

- in some cases, saving -L is useful
  when software installed into /opt
- therefore RPATH was introduced

https://stackoverflow.com/questions/28230983/gcc-l-command-confusion

-l and -L (7)

- if `-rpath` is used, `-L` is not needed
- `rpath=dir` adds a directory to the runtime library search path
- used when linking an ELF executable with shared objects.
- all arguments are concatenated and
  passed to the runtime linker,
  which uses them to locate shared objects at runtime

`https://stackoverflow.com/questions/28230983/gcc-l-command-confusion`

LD_LIBRARY_PATH (1)

- the predefined environmental variable
- contains the paths
  which the linker should look into
- in order to link
  shared / dynamic libraries
- a colon separated list of paths
- which the dynamic loader
  should look for shared libraries

https://stackoverflow.com/questions/7148036/what-is-ld-library-path-and-how-to-use

LD_LIBRARY_PATH (2)

- the standard library paths
  /lib and /usr/lib
- the paths in LD_LIBRARY_PATH have
  higher priority than
  the standard library paths
  - the standard paths will still be searched,
    but *only after* the paths in LD_LIBRARY_PATH
    have been searched

https://stackoverflow.com/questions/7148036/what-is-ld-library-path-and-how-to-use

LD_LIBRARY_PATH (3)

- The best way to use LD_LIBRARY_PATH is to set it on the command line or script *immediately before* executing the program.
- this way the new LD_LIBRARY_PATH isolated from the rest of your system.
- Example:

```
$ export LD_LIBRARY_PATH="/list/of/library/paths:/another/path"
$ ./program
```

https://stackoverflow.com/questions/7148036/what-is-ld-library-path-and-how-to-use

LD_LIBRARY_PATH and -L (3)

- LD_LIBRARY_PATH has the side-effect of *altering*
  - the way gcc and ld behave
  - the way the the run-time linker behaves

  by modifying the search path.
- LD_LIBRARY_PATH *affects* this search path implicitly (sometimes not a good thing)

https://stackoverflow.com/questions/1904990/what-is-the-difference-between-ld-libr

LD_LIBRARY_PATH and -L (4)

- without using LD_LIBRARY_PATH on most Linux systems
  - to *add* the path that contains your shared libraries in /etc/ld.so.conf file
  - create a file in /etc/ld.so.conf.d/ with the path in it
  - run ldconfig (/sbin/ldconfig as root) to update the runtime linker bindings cache.

```
$ cat ld.so.conf
include /etc/ld.so.conf.d/*.conf

$ ls
fakeroot-x86_64-linux-gnu.conf  libc.conf
i386-linux-gnu.conf             x86_64-linux-gnu.conf
```

https://stackoverflow.com/questions/1904990/what-is-the-difference-between-ld-libr

LD_LIBRARY_PATH and -L (5)

```
$ cat fakeroot-x86_64-linux-gnu.conf
/usr/lib/x86_64-linux-gnu/libfakeroot

$ cat libc.conf
# libc default configuration
/usr/local/lib

$ cat i386-linux-gnu.conf
# Multiarch support
/usr/local/lib/i386-linux-gnu
/lib/i386-linux-gnu
/usr/lib/i386-linux-gnu
/usr/local/lib/i686-linux-gnu
/lib/i686-linux-gnu
/usr/lib/i686-linux-gnu

$ cat x86_64-linux-gnu.conf
# Multiarch support
/usr/local/lib/x86_64-linux-gnu
/lib/x86_64-linux-gnu
/usr/lib/x86_64-linux-gnu
```

https://stackoverflow.com/questions/1904990/what-is-the-difference-between-ld-libr

LD_LIBRARY_PATH and -L (6)

- when the program is executed, the run-time linker
  will look in those directories for libraries
  that your binary has been linked against.

  Example on Debian:

```
jewart@dorfl:~$ cat /etc/ld.so.conf.d/usrlocal.conf
/usr/local/lib
```

https://stackoverflow.com/questions/1904990/what-is-the-difference-between-ld-libr

LD_LIBRARY_PATH and -L (7)

- If you want to know what libraries
  the run-time linker knows about,
  you can use:

```
$ ldconfig -v

/usr/lib:
libbfd-2.18.0.20080103.so -> libbfd-2.18.0.20080103.so
libkdb5.so.4 -> libkdb5.so.4.0
libXext.so.6 -> libXext.so.6.4.0
```

https://stackoverflow.com/questions/1904990/what-is-the-difference-between-ld-libr

LD_LIBRARY_PATH and -L (8)

- And, if you want to know what libraries
  a binary is linked against,
  you can use `ldd` like such,
  which will tell you which library
  your runtime linker is going to choose:

```
$ ldd /bin/ls
linux-vdso.so.1 =>  (0x00007fffda1ff000)
librt.so.1 => /lib/librt.so.1 (0x00007f5d2149b000)
libselinux.so.1 => /lib/libselinux.so.1 (0x00007f5d2127f000)
libacl.so.1 => /lib/libacl.so.1 (0x00007f5d21077000)
libc.so.6 => /lib/libc.so.6 (0x00007f5d20d23000)
```

https://stackoverflow.com/questions/1904990/what-is-the-difference-between-ld-libr

-rpath (1)

- rpath designates the run-time search path *hard-coded* in an executable file or library
- dynamic linking loaders use the rpath to find required libraries.

https://en.wikipedia.org/wiki/Rpath#+end_src

-rpath (2)

- Specifically, it *encodes* a path to shared libraries
  into the header of an executable (or another shared library).
- this RPATH header value (so named in the ELF header standards)
  may either *override* or *supplement*
  the system default dynamic linking search paths.

`https://en.wikipedia.org/wiki/Rpath#+end_src`

-rpath (3)

- The rpath of an executable or shared library
  is an optional entry in the .dynamic section
  of the ELF executable or shared libraries,
  with the type DT_RPATH, called the DT_RPATH attribute.
- tt can be *stored there* at link time by the linker
- Tools such as chrpath and patchelf
  can create or modify the entry later.

https://en.wikipedia.org/wiki/Rpath#+end_src

ld.so (1)

- The dynamic linker of the GNU C Library
  searches for shared libraries
  in the following locations in order:[1]

  1. The (colon-separated) paths
     in the DT_RPATH dynamic section attribute of the binary
     if *present* and the DT_RUNPATH attribute does not *exist*

https://en.wikipedia.org/wiki/Rpath#+end_src

ld.so (2-1)

2. the (colon-separated) paths
   in the environment variable LD_LIBRARY_PATH,

   if the executable is a setuid / setgid binary,
   then LD_LIBRARY_PATH is ignored.

   LD_LIBRARY_PATH can be overridden
   by calling the dynamic linker with the option --library-path

```
/lib/ld-linux.so.2 --library-path $HOME/mylibs myprogram
```

https://en.wikipedia.org/wiki/Rpath#+end_src

ld.so (2-2)

3. The (colon-separated) paths in the DT_RUNPATH dynamic section attribute of the binary if present.

https://en.wikipedia.org/wiki/Rpath#+end_src

ld.so (3)

1. Lookup based on the ldconfig cache file (often located at /etc/ld.so.cache) which contains a compiled list of candidate libraries previously found in the augmented library path (set by /etc/ld.so.conf). If, however, the binary was linked with the -z nodefaultlib linker option, libraries in the default library paths are skipped.
2. In the trusted default path /lib, and then /usr/lib. If the binary was linked with the -z nodefaultlib linker option, this step is skipped.

https://en.wikipedia.org/wiki/Rpath#+end_src

ld.so (4)

- The GNU Linker (GNU ld) implements a feature which it calls "new-dtags", which can be used to insert an rpath that has lower precedence than the LD_LIBRARY_PATH environment variable. [2]
- If the new-dtags feature is enabled in the linker (–enable-new-dtags), GNU ld, besides setting the DT_RPATH attribute, also sets the DT_RUNPATH attribute to the same string. At run time, if the dynamic linker finds a DT_RUNPATH attribute, it ignores the value of the DT_RPATH attribute, with the effect that LD_LIBRARY_PATH is checked first and the paths in the DT_RUNPATH attribute are only searched afterwards.

https://en.wikipedia.org/wiki/Rpath#+end_src

ld.so (5)

- The ld dynamic linker does not search DT_RUNPATH locations for transitive dependencies, unlike DT_RPATH. [3]
- Instead of specifying the -rpath to the linker, the environment variable LD_RUN_PATH can be set to the same effect.

https://en.wikipedia.org/wiki/Rpath#+end_src

-rpath man page (1)

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

https://ftp.gnu.org/old-gnu/Manuals/ld-2.9.1/html_node/ld_3.html

-rpath man page (2)

- -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 filesystems.

https://ftp.gnu.org/old-gnu/Manuals/ld-2.9.1/html_node/ld_3.html

-rpath man page (3)

- -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.

https://ftp.gnu.org/old-gnu/Manuals/ld-2.9.1/html_node/ld_3.html

-rpath-link man page (1)

- `rpath-link DIR`
  - when using ELF or SunOS, one shared library may require *another*
  - this happens when an `ld -shared` link includes a shared library as one of the input files.
  - may specify a sequence of directory names
    - by specifying a list of names separated by colons, or
    - by appearing multiple times

`https://ftp.gnu.org/old-gnu/Manuals/ld-2.9.1/html_node/ld_3.html`

-rpath-link man page (2)

- `rpath-link DIR`
  - when the linker encounters such a dependency
    when doing a non-shared, non-relocateable link,
    it will automatically try to locate
    the required shared library and
    include it in the link, if it is not included explicitly.
  - in such a case, the `-rpath-link` option specifies
    the first set of directories to search.

`https://ftp.gnu.org/old-gnu/Manuals/ld-2.9.1/html_node/ld_3.html`

-rpath-link man page (3)

- the linker uses the following search paths
  to locate required shared libraries.
  1. Any directories specified by -rpath-link options.
  2. Any directories specified by -rpath options.
     The difference between -rpath and -rpath-link is
     that directories specified by -rpath options
     are included in the executable and used at runtime,
     whereas the -rpath-link option is
     only effective at link time
  3. On an ELF system,
     if the -rpath and -rpath-link options were not used,
     search the contents of the environment variable LD_RUN_PATH

https://ftp.gnu.org/old-gnu/Manuals/ld-2.9.1/html_node/ld_3.html

-rpath-link man page (4)

- the linker uses the following search paths
  to locate required shared libraries.
  1. On SunOS, if the -rpath option was not used,
     search any directories specified using -L options.
  2. For a native linker, the contents of
     the environment variable LD_LIBRARY_PATH
  3. The default directories, normally /lib and /usr/lib
- If the required shared library is not found,
  the linker will issue a warning and continue with the link.

https://ftp.gnu.org/old-gnu/Manuals/ld-2.9.1/html_node/ld_3.html

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

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 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 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 use 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 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

- the first method using -L and l,
  ignoring the advice

  ```
  try  using -rpath or -rpath-link
  ```

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

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

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
```

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

Using -Wl, rpath . (1)

- The man page makes it pretty clear. If you want
- in order to pass -rpath . to the linker
  considerING them as two arguments
  (-rpath and .) to the -Wl
  you can write
  - -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 aaa bbb ccc
- gcc -Wl,aaa,bbb,ccc

- ld -rpath .
- gcc -Wl,-rpath,.

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

Using -Wl,-rpath,. (3)

- Alternatively, you can specify repeat instances of `-Wl`
  `gcc -Wl,aaa -Wl,bbb -Wl,ccc`
  - there is no comma between aaa and the second `-Wl`

- `ld -rpath .`
- `gcc -Wl,-rpath,.`
- `gcc -Wl,-rpath -Wl,.`

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

Using -Wl,-rpath,. (4)

- can remove the space

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

- `ld -rpath .`
- `gcc -Wl,-rpath,.`
- `gcc -Wl,-rpath -Wl,.`
- `gcc -Wl,-rpath=.`

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

-rpath-link (1)

- The `-rpath-link=dir` option tells the linker that
  when it encounters an input file
  that requests dynamic dependencies
  it should search `dir` to resolve them.
- `libfoobar.so` needs `libfoo.so` and `libbar.so`
  - if `rpath-link` is used,
    no need to specify dynamic dependencies
    no need to know what they are
    no need `-lfoo -lbar`

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

-rpath-link (2)

- the dynamic dependencies is defined
  in the dynamic section of `libfoobar.so`
  - therefore, just need to provide a directory
    where the required shared libraries can be found

```
$ 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 (3)

- But does -rpath-link=dir give us a executable prog?
  No. Same as story as before.

```
$ ./prog
./prog: error while loading shared libraries: libfoobar.so: cannot open shared
```

  - at runtime, libfoo.so, libbar.so, and libfoobar.so might not be where they were linked
  - but the loader might be able to locate them by other means:
    - through the ldconfig cache
    - by setting the LD_LIBRARY_PATH environment variable

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

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

-rpath-link (4)

- -rpath-link=dir gives the linker the information
  that the loader would need to resolve some of
  the dynamic dependencies of prog at runtime
  - assuming the dynamic dependencies *remained* true at runtime
  - but it doesn't write that information
    into the dynamic section of prog
  - it just lets the linkage succeed,
    without our needing to spell out
    all the recursive dynamic dependencies
    of the linkage with -l options.

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

-rpath-link (5)

- rpath=dir provides the linker with the same information as rpath-link=dir does
  and instructs the linker to bake that information into the dynamic section of the output file

```
$ 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-link (6)

- Now all good. Because this time, prog contains the information that $(pwd) is a runtime search path for shared libraries that it depends on, as we can see:

```
$ 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/
 ...                                     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
 ...
```

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

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

rpath and -rpath-link

LD_LIBRARY_PATH and LD_RUN_PATH (0)

LD_RUN_PATH	LD_LIBRARY_PATH
link time resolution	run time resolution
linker	dynamic loader

https://ftp.gnu.org/old-gnu/Manuals/ld-2.9.1/html_node/ld_3.html

LD_LIBRARY_PATH and LD_RUN_PATH (1)

LD_RUN_PATH is used for the *link time* resolution of libraries LD_LIBRARY_PATH is used for *run time* resolution of libraries.
LD_RUN_PATH is used by the *linker* to specify where to search libraries only at *run time* LD_LIBRARY_PATH is uded by the *dynamic loader* to specify where to search the libraries required to *execute* the binary (at the *run time* of the binary)
LD_RUN_PATH is the *runtime* library seach path LD_LIBRARY_PATH paths are *not* searched during *link time*

https://www.quora.com/What-is-the-difference-between-LD_LIBRARY_PATH-and-LD_RUN_PA

LD_LIBRARY_PATH and LD_RUN_PATH (2)

- LD_RUN_PATH variable is used by the linker (ld)
  the same way as -rpath argument to ld is used
- LD_RUN_PATH is used if -rpath is not specified
- However, if some binary is linked
  LD_RUN_PATH is not used and
  -rpath is specified on ld command line
  and you want to change the paths used
  to look for libraries at run time,
  LD_LIBRARY_PATH variable must be specified
  which is used by the dynamic linker (/lib/ld-linux.so.*)

https://bugzilla.redhat.com/show_bug.cgi?id=20218

LD_LIBRARY_PATH and LD_RUN_PATH (3)

- When you use the -l option,
  you must inform the dynamic linker about the directories
  of the dynamically linked libraries
  that are to be linked with your program at execution
- The environment variable LD_RUN_PATH
  lets you do this at link time
- to set LD_RUN_PATH, list the colon separated
  absolute pathnames of the directories
  in the order you want them searched

```
LD_RUN_PATH=/home/mylibs
export LD_RUN_PATH
```

http://osr507doc.sco.com/en/tools/ccs_linkedit_dynamic_dirsearch.html

LD_LIBRARY_PATH and LD_RUN_PATH (4)

- the command:
  `cc -static -fpic -o prog file1.c file2.c -L/home/mylibs -lfoo`
  directs the dynamic linker to search for `libfoo.so`
  in `/home/mylibs` when you execute your program `prog`
- the dynamic linker searches the standard place by default,
  after the directories you have assigned to `LD_RUN_PATH`
- Note that as far as the dynamic linker is concerned,
  the standard place for libraries is `/usr/lib`.
- Any executable versions of libraries
  supplied by the compilation system kept in `/usr/lib`

`http://osr507doc.sco.com/en/tools/ccs_linkedit_dynamic_dirsearch.html`

LD_LIBRARY_PATH and LD_RUN_PATH (5)

- The environment variable LD_LIBRARY_PATH lets you do the same thing at run time.
- Suppose you have moved libfoo.so to /home/sharedobs
  /home/mylibs → /home/sharedobs
- It is too late to change LD_RUN_PATH,
  at least without link editing your program again

```
LD_RUN_PATH=/home/sharedobs
export LD_RUN_PATH    (--> not woking)
```

- however, you can change LD_LIBRARY_PATH

```
LD_LIBRARY_PATH=/home/sharedobs
export LD_LIBRARY_PATH
```

http://osr507doc.sco.com/en/tools/ccs_linkedit_dynamic_dirsearch.html

LD_LIBRARY_PATH and LD_RUN_PATH (6)

- compile command

  ```
  cc -static -fpic -o prog file1.c file2.c -L/home/mylibs -lfoo
  ```

- now when you execute your program prog
- the dynamic linker
  searches for libfoo.so first in /home/mylibs
  and, not finding it there, in /home/sharedobs.

  ```
  LD_RUN_PATH=/home/mylibs
  LD_LIBRARY_PATH=/home/sharedobs
  ```

- the directory assigned to LD_RUN_PATH is searched
  before the directory assigned to LD_LIBRARY_PATH.

http://osr507doc.sco.com/en/tools/ccs_linkedit_dynamic_dirsearch.html

LD_LIBRARY_PATH and LD_RUN_PATH (7)

- because the pathname of `libfoo.so`
  is not hard-coded in `prog`,

  you can *direct* the dynamic linker
  to *search* a different directory
  when you execute your program. (`LD_LIBRARY_PATH`)

- You can move a dynamically linked library
  without breaking your application.

```
LD_RUN_PATH=/home/mylibs
LD_LIBRARY_PATH=/home/sharedobs
```

`http://osr507doc.sco.com/en/tools/ccs_linkedit_dynamic_dirsearch.html`

LD_LIBRARY_PATH and LD_RUN_PATH (8)

- You can set LD_LIBRARY_PATH
  without first having set LD_RUN_PATH
- once you have used LD_RUN_PATH for an application,
  the dynamic linker searches the specified directories
  whenever the application is executed

  unless you have relinked the application
  in a different environment
  - first LD_RUN_PATH, then LD_LIBRARY_PATH
  - LD_RUN_PATH overrides LD_LIBRARY_PATH

http://osr507doc.sco.com/en/tools/ccs_linkedit_dynamic_dirsearch.html

LD_LIBRARY_PATH and LD_RUN_PATH (9)

- can assign different directories to LD_LIBRARY_PATH whenever you execute the application.
- LD_LIBRARY_PATH directs the dynamic linker to search the assigned directories before it searches the standard place.
- directories, including those in the optional second list, are searched in the order listed.

http://osr507doc.sco.com/en/tools/ccs_linkedit_dynamic_dirsearch.html

LD_LIBRARY_PATH and LD_RUN_PATH (10)

- when linking a `set-user` or `set-group` program, the dynamic linker ignores any directories that are not built into the dynamic linker.
- Currently, the only built-in directory is `/usr/lib`

`http://osr507doc.sco.com/en/tools/ccs_linkedit_dynamic_dirsearch.html`

LD_LIBRARY_PATH and LD_RUN_PATH (11)

- can use the environment variable LD_LIBRARY_PATH
  which takes a colon(:) separated list of directories,
  to add to the link-editor's library search path.
- In its most general form, LD_LIBRARY_PATH
  takes two directory lists separated by a semicolon(;)
  - The first list is searched before
    the list(s) supplied on the command-line
  - the second list is searched after

https://docs.oracle.com/cd/E19455-01/816-0559/chapter2-48927/index.html

LD_LIBRARY_PATH and LD_RUN_PATH (12)

- Here is the combined effect of setting LD_LIBRARY_PATH and calling the link-editor with several -L occurrences:

```
$ LD_LIBRARY_PATH=dir1:dir2;dir3
$ export LD_LIBRARY_PATH
$ cc -o prog main.c -Lpath1 ... -Lpath2 ... -Lpathn -lfoo
```

  - the first path list dir1:dir2
  - the second path list dir3

- The effective search path will be

```
dir1:dir2:path1:path2... pathn:dir3:/usr/ccs/lib:/usr/lib.
```

https://docs.oracle.com/cd/E19455-01/816-0559/chapter2-48927/index.html

LD_LIBRARY_PATH and LD_RUN_PATH (13)

- If no semicolon(;) is specified
  as part of the LD_LIBRARY_PATH definition,
  the specified directory list is interpreted
  after any -L options (the second list)

```
$ LD_LIBRARY_PATH=dir1:dir2
$ export LD_LIBRARY_PATH
$ cc -o prog main.c -Lpath1 ... -Lpath2 ... -Lpathn -lfoo
```

- Here the effective search path will be

```
path1:path2... pathn:dir1:dir2:/usr/ccs/lib:/usr/lib.
```

https://docs.oracle.com/cd/E19455-01/816-0559/chapter2-48927/index.html

LD_LIBRARY_PATH and LD_RUN_PATH (14)

- This environment variable can also be used
  to augment the search path of the runtime linker
  (see "Directories Searched by the Runtime Linker" for more details).
- To prevent this environment variable from
  influencing the link-editor, use the `-i` option.

`https://docs.oracle.com/cd/E19455-01/816-0559/chapter2-48927/index.html`