Link 5. Search Libararies (II) Using RPATH

Young W. Lim

2023-04-26 Fri

Outline

1 Based on

2 Search libraries (II)

- `-rpath-link`
- `-rpath`
- `LD_RUN_PATH`
- BFD linkers
- Gold linkers

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`

-rpath-link

`-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.
  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 difference between `-rpath` and `-rpath-link`
  - directories specified by `-rpath` options
    are included in the executable
    and used at runtime,
- the `-rpath-link` option is
  only effective at link time

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

-rpath-link man page (5)

- 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

(1) informs the linker

- 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 to use `-lfoo -lbar`

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

(2) dynamic depencieds in `.dynamic` section

- the dynamic dependencies is defined in the `.dynamic` section of `libfoobar.so`
  - (`NEEDED` shared library file names)
  - 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

(3) the loader at rumtime

- But does `-rpath-link=dir` give us a executable `prog`? – No.

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

- 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

(4) effective at link time

- `-rpath-link=dir` gives the linker (`ld`) the information that the loader (`ld.so`) would need to resolve some of the dynamic dependencies of `prog` at runtime
  - directories specified by `-rpath` options are *included* in the executable and *used* at runtime,
  - the `-rpath-link` option is only *effective* at link time

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

(5) no need using `-l` options

- 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 spelling out all the recursive dynamic dependencies of the linkage by using `-l` options

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

-rpath-link (5)

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

    (DT_RPATH / DT_RUNPATH entry in .dynamic section)

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

-rpath-link (6-1)

- by using -rpath=$(pwd), prog contains the information that $(pwd) is a runtime search path for shared libraries that it depends on

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

- 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/scrap]
                              ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
 ...
 ...
```

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

-rpath-link (6-2)

- That search path will be tried
  (RPATH) /home/imk/develop/so/scrap
  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

-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.
  - dynamic linking is a sort of "lazy" linking
    of required shared libraries
    not during the stage of compiling
    but the later stage of running an executable.
- the rpath can be *stored there*
  at link time by the linker

https://en.wikipedia.org/wiki/Rpath#+end_src
https://nehckl0.medium.com/creating-relocatable-linux-executables-by-setting-rpat

-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
- tools such as `chrpath` and `patchelf` can *create* or *modify* the entry `DT_RPATH` later.

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

rpath and runpath (1)

- rpath and runpath are the most complex items in runtime search path
- the rpath and runpath of an executable or shared library are *optional entries* in the `.dynamic` section
- they are both a list of directories to search for

Name	Value	d_un	Executable	Shared Object
DT_RPATH*	15	d_val	optional	ignored
DT_RUNPATH	29	d_val	optional	optional

https://refspecs.linuxbase.org/elf/gabi4+/ch5.dynamic.html

rpath and runpath (2)

- The only difference between rpath and runpath is the order they are searched in.
- Specifically, their relation to `LD_LIBRARY_PATH`
  - rpath is searched in before `LD_LIBRARY_PATH`
  - runpath is searched in after `LD_LIBRARY_PATH`

  1. search rpath
  2. search `LD_LIBRARY_PATH`
  3. search runpath

- rpath cannot be changed dynamically
- runpath can be changed dynamically with environment variables

`https://refspecs.linuxbase.org/elf/gabi4+/ch5.dynamic.html`

rpath and runpath (3)

- 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`

Displaying RPATH / RUNPATH

- `readelf -d <binary_name> | grep 'R.*PATH'`
  - displays the RPATH or RUNPATH of a binary file.
  - In gcc, for instance, one could specify RPATH by `-Wl,-rpath,/custom/rpath/`

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

`--inhibit-rpath LIST`

- the option `--inhibit-rpath LIST` of the dynamic linker instructs it to ignore `DT_RPATH` and `DT_RUNPATH` attributes of the object names in `LIST`.
- to specify a `main` program in the `LIST`, give empty string

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

LD_PRELOAD environment variable

- libraries specified by the environment variable `LD_PRELOAD` and then those listed in `/etc/ld.so.preload` are loaded before the search begins.
- a preload can thus be used to replace some (or all) of the requested library's normal functionalities, or it can simply be used to supply a library that would otherwise not be found.
- static libraries are searched and linked into the ELF file at link time and are not searched at run time.

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

--enable-new-dtags (1)

- The GNU Linker (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.

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

--enable-new-dtags (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

Dynamic section

- If an object file participates in dynamic linking, its program header table will have an element of type `PT_DYNAMIC`.
- this segment contains the `.dynamic` section
- a special symbol, `_DYNAMIC`, labels the section, which contains an array of the following structures

https://refspecs.linuxbase.org/elf/gabi4+/ch5.dynamic.html

Dynamic structure

```
typedef struct {
  Elf32_Sword     d_tag;
  union {
    Elf32_Word    d_val;
    Elf32_Addr    d_ptr;
  } d_un;
} Elf32_Dyn;

extern Elf32_Dyn  _DYNAMIC[];
```

```
typedef struct {
  Elf64_Sxword    d_tag;
   union {
    Elf64_Xword   d_val;
    Elf64_Addr    d_ptr;
   } d_un;
    } Elf64_Dyn;

     extern Elf64_Dyn  _DYNAMIC[];
```

- d_tag controls the interpretation of d_un (*union*)
- d_val these objects represent integer values
  with various interpretations.
- d_ptr these objects represent program virtual addresses

https://refspecs.linuxbase.org/elf/gabi4+/ch5.dynamic.html

Dynamic array tags d_tag of the .dynamic section (1)

Name	Value	d_un	Executable	Shared Object
DT_NULL	0	ignored	mandatory	mandatory
DT_NEEDED	1	d_val	optional	optional
DT_PLTRELSZ	2	d_val	optional	optional
DT_PLTGOT	3	d_ptr	optional	optional
DT_HASH	4	d_ptr	mandatory	mandatory
DT_STRTAB	5	d_ptr	mandatory	mandatory
DT_SYMTAB	6	d_ptr	mandatory	mandatory
DT_RELA	7	d_ptr	mandatory	optional
DT_RELASZ	8	d_val	mandatory	optional
DT_RELAENT	9	d_val	mandatory	optional
DT_STRSZ	10	d_val	mandatory	mandatory
DT_SYMENT	11	d_val	mandatory	mandatory
DT_INIT	12	d_ptr	optional	optional
DT_FINI	13	d_ptr	optional	optional
DT_SONAME	14	d_val	ignored	optional
DT_RPATH*.......	15......	d_val.....	optional.......	ignored
DT_SYMBOLIC*	16	ignored	ignored	optional
DT_REL	17	d_ptr	mandatory	optional
DT_RELSZ	18	d_val	mandatory	optional
DT_RELENT	19	d_val	mandatory	optional

https://refspecs.linuxbase.org/elf/gabi4+/ch5.dynamic.html

Dynamic array tags d_tag of the .dynamic section (2)

Name	Value	d_un	Executable	Shared Object
DT_PLTREL	20	d_val	optional	optional
DT_DEBUG	21	d_ptr	optional	ignored
DT_TEXTREL*	22	ignored	optional	optional
DT_JMPREL	23	d_ptr	optional	optional
DT_BIND_NOW*	24	ignored	optional	optional
DT_INIT_ARRAY	25	d_ptr	optional	optional
DT_FINI_ARRAY	26	d_ptr	optional	optional
DT_INIT_ARRAYSZ	27	d_val	optional	optional
DT_FINI_ARRAYSZ	28	d_val	optional	optional
DT_RUNPATH	29	d_val	optional	optional
DT_FLAGS	30	d_val	optional	optional
DT_ENCODING	32	unspecified	unspecified	unspecified
DT_PREINIT_ARRAY	32	d_ptr	optional	ignored
DT_PREINIT_ARRAYSZ	33	d_val	optional	ignored
DT_LOOS	0x6000000D	unspecified	unspecified	unspecified
DT_HIOS	0x6ffff000	unspecified	unspecified	unspecified
DT_LOPROC	0x70000000	unspecified	unspecified	unspecified
DT_HIPROC	0x7fffffff	unspecified	unspecified	unspecified

https://refspecs.linuxbase.org/elf/gabi4+/ch5.dynamic.html

RPATH example

- an example of readelf output with RUNPATH and $ORIGIN:

```
Dynamic section at offset 0x210268 contains 30 entries:

Tag                Type         Name/Value

(d_tag)           (DT_RUNPATH) (d_val)

0x000000000000001d (RUNPATH)    Shared library: [$ORIGIN]
```

https://nehckl0.medium.com/creating-relocatable-linux-executables-by-setting-rpat

DT_RPATH

- DT_RPATH element holds the string table offset of a null-terminated search library search path string
- the offset is an index into the table recorded in the DT_STRTAB entry.
- this entry is at level 2.
- its use has been superseded by DT_RUNPATH

https://nehckl0.medium.com/creating-relocatable-linux-executables-by-setting-rpatl

DT_RUNPATH

- DT_RUNPATH element holds the string table offset of a null-terminated library search path string
- the offset is an index into the table recorded in the DT_STRTAB entry.

https://nehckl0.medium.com/creating-relocatable-linux-executables-by-setting-rpatl

$ORIGIN (1)

- paths in `rpath` and `runpath` can be
  1. absolute (e.g., `/path/to/my/libs/`)
  1. relative to the current working directory (e.g., `.`)
  1. relative to the executable
     by using the `$ORIGIN` variable
     in the `rpath` definition:

`https://amir.rachum.com/shared-libraries/`

`$ORIGIN` (2)

- when the dynamic linker loads an object that uses `$ORIGIN`, it must calculate the pathname of the directory containing the object
- the pathname will contain
  - no symbolic links
  - no use of `.` or `..` components.

`https://nehckl0.medium.com/creating-relocatable-linux-executables-by-setting-rpat`

$ORIGIN (3)

- within a string provided by dynamic array entries with the DT_NEEDED or DT_RUNPATH tags and in pathnames passed as parameters to the dlopen() routine, a dollar sign ($) introduces a substitution sequence.
- substituion sequence consists of the $ sign immediately followed by
  - either the longest name sequence
  - or a name contained within { and }

https://nehckl0.medium.com/creating-relocatable-linux-executables-by-setting-rpatl

$ORIGIN (4)

- If the name is ORIGIN,
  then the dynamic linker replaces
  the substitution sequence with
  the absolute pathname of the directory
  containing the object which
  the substitution sequence originated.
- Otherwise (when the name is not ORIGIN)
  the behavior of the dynamic linker is unspecified

https://refspecs.linuxbase.org/elf/gabi4+/ch5.dynamic.html#shobj_dependencies

how to *check* the value of RPATH / RUNPATH

- `$ objdump -x path/to/executable | grep RPATH`
- `$ readelf -d path/to/executable | head -20`
- `$ chrpath -l path/to/executable`

https://nehckl0.medium.com/creating-relocatable-linux-executables-by-setting-rpatl

how to *set* the value of RPATH / RUNPATH (1)

- during compilation time, use `configure -rpath=`

```
$ ./configure LDFLAGS=-Wl,-rpath=$ORIGIN/lib_path
```

  - this will tell the linker
    to build and run the executable
    under the *specified* library path,
    usually used to override the *default* library paths.

https://nehckl0.medium.com/creating-relocatable-linux-executables-by-setting-rpat

how to *set* the value of RPATH / RUNPATH (2)

- after compilation before execution

```
$ chrpath -r ‘‘\$\ORIGIN/lib_path’’ <executable>
```

  - this command could fail
    if no rpath was set previously for the executable.

https://nehckl0.medium.com/creating-relocatable-linux-executables-by-setting-rpatl

how to *set* the value of RPATH / RUNPATH (3)

- try below command with `patchelf` utility,
  which won't complain about an unset rpath,
  and will get `RUNPATH` set to achieve similar target.

```
$ patchelf --set-rpath '$ORIGIN/lib_path' <executable>
```

https://nehckl0.medium.com/creating-relocatable-linux-executables-by-setting-rpat

objdump -x

- objdump -x
  - display all available *header* information,
    including the symbol table and relocation entries
  - Using -x is equivalent to specifying all of
    - -a archive header information
    - -f file headers, summary from the overall header
    - -h section header
    - -p private headers, specific to the object file format
    - -r relocation entries
    - -t symbol table entries

objdump man page

readelf -d

- `readelf -d`
    - displays the contents of the file's dynamic section, if it has one.

readelf man page

Configure the software (1)

- The configure script is responsible for getting ready to build the software on your specific system.
- It makes sure all of the dependencies for the rest of the build and install process are available, and finds out whatever it needs to know to use those dependencies

https://thoughtbot.com/blog/the-magic-behind-configure-make-make-install

Configure the software (2)

- Unix programs are often written in C, so we'll usually need a C compiler to build them.
- in these cases the configure script will establish that your system does indeed have a C *compiler*, and find out what it's *called* and where to *find* it.

`https://thoughtbot.com/blog/the-magic-behind-configure-make-make-install`

Build the software

- Once configure has done its job,
  we can invoke `make` to build the software.
- this runs a series of tasks defined in a Makefile
  to build the finished program from its source code.
- The tarball you download usually
  doesn't include a finished Makefile.
- Instead it comes with a *template* called Makefile.in and
  the configure script produces a *customised* Makefile
  specific to your system.

`https://thoughtbot.com/blog/the-magic-behind-configure-make-make-install`

Install the software (1)

- when the software is built and ready to run,
  the files can be copied to their final destinations
- The `make install` command will copy
  - the built program, and
  - its libraries and
  - documentation,

  to the correct locations.

https://thoughtbot.com/blog/the-magic-behind-configure-make-make-install

Install the software (2)

- the program's binary will be copied to a directory on your PATH,
- the program's manual page will be copied to a directory on your MANPATH, and
- any other files it depends on will be safely stored in the appropriate place.

https://thoughtbot.com/blog/the-magic-behind-configure-make-make-install

Install the software (3)

- since the *install step* is also defined in the Makefile, where the software is installed can change based on options passed to the configure script, or things the configure script discovered about your system.
- depending on where the software is being installed, you might need escalated permissions for this step so you can copy files to system directories.
- Using `sudo` will often do the trick.

https://thoughtbot.com/blog/the-magic-behind-configure-make-make-install

Configure script

- a shell script (generally written by GNU Autoconf) that goes up and looks for software and even tries various things to see what works.
- it then takes its *instructions* from Makefile.in and *builds* Makefile (and possibly some other files) that work on the current system.

`https://tldp.org/LDP/LG/current/smith.html`

Configure, make, makeinstall

- You run configure, type `./configure`
  this builds a new Makefile
- Type `make`
  this *builds* the program.
  look for the first target in Makefile and
  do what the instructions said.
  The expected end result would be to build an executable program
- Now, as root, type `make install`
  this again invokes `make`,
  finds the target install in Makefile and
  copies files to the directories to install the program.

https://tldp.org/LDP/LG/current/smith.html

patchelf (1)

- PatchELF is a simple utility for modifying existing ELF executables and libraries.
    - can change the dynamic loader ("ELF interpreter") of executables
    - can change the `RPATH` of executables and libraries.

`https://nehckl0.medium.com/creating-relocatable-linux-executables-by-setting-rpat`

patchelf (2)

- patchelf
  - --set-rpath RUNPATH
    Change the DT_RUNPATH of the executable or library to RUNPATH
  - --add-rpath RUNPATH
    Add RUNPATH to the existing DT_RUNPATH of the executable or library.
  - --remove-rpath
    Removes the DT_RPATH or DT_RUNPATH entry
    of the executable or library.

https://nehckl0.medium.com/creating-relocatable-linux-executables-by-setting-rpat

patchelf (3)

- patchelf
    - --shrink-rpath

      Remove from the DT_RUNPATH or DT_RPATH all directories that do not contain a library referenced by DT_NEEDED fields of the executable or library.

      For instance, if an executable
      references one library libfoo.so,
      has an RPATH "/lib:/usr/lib:/foo/lib", and
      libfoo.so can only be found in /foo/lib,
      then the new RPATH will be "/foo/lib".

https://nehckl0.medium.com/creating-relocatable-linux-executables-by-setting-rpat

patchelf (4)

- patchelf
  - --allowed-rpath-prefixes PREFIXES
    Combined with the "--shrink-rpath" option,
    this can be used for further rpath tuning.
    for instance, if an executable has
    an RPATH "/tmp/build-foo/.libs:/foo/lib",
    it is probably desirable to keep the "/foo/lib" reference
    instead of the "/tmp" entry.

https://nehckl0.medium.com/creating-relocatable-linux-executables-by-setting-rpat

patchelf (5)

- patchelf
  - --print-rpath
    Prints the DT_RUNPATH or DT_RPATH for an executable or library.
  - --force-rpath
    Forces the use of the obsolete DT_RPATH in the file
    instead of DT_RUNPATH.
    By default DT_RPATH is converted to DT_RUNPATH

https://nehckl0.medium.com/creating-relocatable-linux-executables-by-setting-rpat

-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`

LD_RUN_PATH

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_P

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 $\rightarrow$ /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

Executable File (1)

- executable files of various formats
  can be directly executed by the CPU
  once loaded by a suitable executable loader,
  rather than being interpreted by other software

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

Executable File (2)

- typical executables contain

• binary application code
• headers and tables with relocation and fixup information
• various kinds of meta data

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

Executable File Formats

- the examples executable file formats

PE	on Microsoft Windows
ELF	on Linux and most other versions of Unix
Mach-O	on macOS and iOS
MZ	on DOS

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

BFD (Binary File Descriptor) (1)

- BFD is a package which allows applications to use the *same routines* to operate on object files whatever the object file format.
- BFD consists of two parts:
  - the front end - common for various object file formats
  - the back ends - one for each object file format
  - a *new* object file format can be supported simply by creating a *new* BFD back end and adding it to the library

https://ftp.gnu.org/old-gnu/Manuals/bfd-2.9.1/html_mono/bfd.html#SEC1

BFD (Binary File Descriptor) (2)

- the front end of BFD provides the interface to the user.
  - manages memory and various canonical data structures
  - decides which back end to use
    and when to call back end routines.
- the back ends provide BFD its view of the real world.
  - provides a set of calls which the BFD front end
    can use to maintain its canonical form
  - may keep around information for their own use,
    for greater efficiency.

`https://ftp.gnu.org/old-gnu/Manuals/bfd-2.9.1/html_mono/bfd.html#SEC1`

BFD (Binary File Descriptor) (3)

- to use the BFD library,
  - include `bfd.h`
  - link with `libbfd.a`
- BFD provides a common interface
  to the parts of an object file
  for a *calling application*
- when an application sucessfully opens
  a target file (object, archive, or whatever),
  a pointer to an internal structure is returned

`https://ftp.gnu.org/old-gnu/Manuals/bfd-2.9.1/html_mono/bfd.html#SEC1`

BFD (Binary File Descriptor) (4)

- this returned pointer points to
  a structure called bfd, described in `bfd.h`
- our convention is to call this pointer, a BFD,
  and instances of it within code, abfd.
- all operations on the target object file are applied
  as methods to the BFD
- the mapping is defined within `bfd.h` in a set of macros,
  all beginning with `bfd_` to reduce namespace pollution

`https://ftp.gnu.org/old-gnu/Manuals/bfd-2.9.1/html_mono/bfd.html#SEC1`

BFD Libraries (1)

- BFD libraries : the GNU Project's *main mechanism* for the portable manipulation of *object files*
  - as of 2003, it supports approximately 50 file formats for some 25 instruction set architectures.
- BFD libraries's main clients

gas	GNU Assembler
gld	GNU Linker
binutil	other GNU Binary Utilities tools
gdb	the GNU Debugger

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

BFD Libraries (2)

- the frequent need to tinker with the API
  to accommodate new systems' capabilities
  has tended to limit its use
- as a result, BFD is not distributed separately,
  but is always included with releases of binutils and GDB
- Nevertheless, BFD is a critical component
  in the use of GNU tools
  for embedded systems development

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

BFD linker (1)

- `ld` combines a number of *object* and *archive files*, relocates their data and ties up symbol references
- Usually the *last step* in compiling a program is to *run* `ld`
- `ld` accepts Linker Command Language files written in a superset of AT&T's Link Editor Command Language syntax, to provide *explicit* and *total* control over the linking process.

`https://manpages.debian.org/testing/binutils-common/ld.bfd.1.en.html`

BFD linker (2)

- the general purpose BFD libraries allows `ld`
  - to read, combine, and write
    object files in many different formats
    - for example, `COFF` or `a.out`
  - to link different formats together
    to produce *any available kind* of object file
  - to read the structured data out of a core dump

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

BFD linker (3)

- flexibile
- providing diagnostic information
  - many linkers abandon execution immediately upon encountering an error;
  - whenever possible, BFD `ld` continues executing, allowing you to identify other errors (or, in some cases, to get an output file in spite of the error).

https://manpages.debian.org/testing/binutils-common/ld.bfd.1.en.html

gold linker (1)

- gold is a linker for ELF files.
  - became an official GNU package
    was added to binutils in March 2008 and
    first released in binutils version 2.19.
  - gold was developed by Ian Lance Taylor
    and a small team at Google
  - to make a linker that is faster than the GNU linker (BFD `ld`),
    especially for large applications coded in C++.

https://en.wikipedia.org/wiki/Gold_(linker)

gold linker (2)

- Unlike the GNU linker,
  gold does not use the BFD library
  - *limits* the object file formats to ELF only
  - a *cleaner* and *faster* implementation may be possible
    without an additional abstraction layer
- BFD library was removed
  to create a new linker *from scratch*
  rather than incrementally improve the GNU linker
  - *fixes* some *bugs* in old `ld`
    that break ELF files in various minor ways.

`https://en.wikipedia.org/wiki/Gold_(linker)`

gold linker (3)

- To specify gold in a makefile,
  one sets the `LD` or `LD` environmental variable to `ld.gold`.
- to specify gold through a compiler option,
  one can use the gcc option `-fuse-ld=gold`

`https://en.wikipedia.org/wiki/Gold_(linker)`

GNU linker options

- to use, instead of the default linker

`-fuse-ld=bfd`	use the bfd linker
`-fuse-ld=gold`	use the gold linker
`-fuse-ld=lld`	use the LLVM lld linker
`-fuse-ld=mold`	use the Modern Linker (mold)

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

C Makefile variables

- commonly used variables in makefiles

CC	C compiler
LD	link editor / load
CPP	C preprocessor
CXX	a C++ compiler
AS	an assembly language compiler
AR	an archive-maintaining program

`https://stackoverflow.com/questions/8862450/in-makefiles-what-do-cc-and-ld-stand-`

LLVM (1)

- The LLVM Project is a collection of modular and reusable compiler and toolchain technologies.
    - the name "LLVM" itself is not an acronym;
    - it is the full name of the project.
    - despite its name *Low Level Virtual Machine*, LLVM has little to do with traditional virtual machines.
        - the LLVM project has grown beyond its initial scope as it is no longer focused on traditional virtual machines.

`https://llvm.org/`

LLVM (2)

- On the front end,
  the LLVM compiler infrastructure uses clang
  - a compiler for programming languages C, C++ and CUDA
  - to turn source code into an interim format
- On the back end
  - LLVM clang code generator turns the interim format
    into final machine code

`https://www.heavy.ai/technical-glossary/llvm`

LLVM (3)

- The compiler has five basic phases
  - Parsing : Groups the words and tokens from the lexical analysis into a form that makes sense.
  - Lexical Analysis : Converts program text into words and tokens (everything apart from words, such as spaces and semicolons)
  - Semantic Analyser : Identifies the types and logics of the programs.
  - Optimization : Cleans the code for better run-time performance and addresses memory-related issues.
  - Code Generation : Turns code into a binary file that is executable.

`https://www.heavy.ai/technical-glossary/llvm`

LLVM Clang

- Clang is an "LLVM native" C/C++/Objective-C compiler, which aims to deliver amazingly fast compiles, extremely useful error and warning messages and to provide a platform for building great source level tools.
- The Clang Static Analyzer and clang-tidy are tools that automatically find bugs in your code, and are great examples of the sort of tools that can be built using the Clang frontend as a library to parse C/C++ code.

`https://www.heavy.ai/technical-glossary/llvm`