Building and Using Shared Libraries on Linux

Shared Libraries: The Dynamic Linker

Michael Kerrisk, man7.org © 2023

February 2023

mtk@man7.org

Outline

4 The Dynamic Linker 4-1
4.1 Rpath: specifying library search paths in an object 4-3
4.2 Dynamic string tokens 4-10
4.3 Finding shared libraries at run time 4-14
4.4 Symbol resolution, interposition, and link maps 4-19
4.5 Debugging the operation of the dynamic linker 4-29
Specifying library search paths in an object

- So far, we have two methods of informing the dynamic linker (DL) of location of a shared library:
  - **LD_LIBRARY_PATH**
  - Installing library in one of the standard directories
- Third method: during static linking, we can **insert a list of directories into the executable**
  - A “run-time library path (**rpath**) list”
  - At run time, DL will search listed directories to resolve dynamic dependencies
  - Useful if libraries will reside in locations that are fixed, but not in standard list
Defining an rpath list when linking

- To embed an rpath list in an executable, use the `–rpath` linker option (`cc –Wl,–rpath,path-to-lib-dir`)
  - Multiple `–rpath` options can be specified ⇒ ordered list
  - Alternatively, multiple directories can be specified as a colon-separated list in a single `–rpath` option

Example:

```
$ cc -g -Wall -Wl,–rpath,$PWD -o prog prog.c libdemo.so
$ objdump -p prog | grep 'R[UN]*PATH'
  RPATH /home/mtk/tlpi/code/shlibs
$ ./prog
  Called mod1-x1
  Called mod2-x2
```

- Embeds current working directory in rpath list
- `objdump` command allows us to inspect rpath list
- Executable now “tells” DL where to find shared library

An rpath improvement: DT_RUNPATH

There are **two types of rpath list:**

- Differ in precedence relative to `LD_LIBRARY_PATH`
- Original (and default) rpath list has higher precedence
  - `DT_RPATH` ELF entry
- Original rpath behavior was a **design error**
  - User should have full control when using `LD_LIBRARY_PATH`
An rpath improvement: DT_RUNPATH

- Newer rpath type has lower precedence
  - Gives user possibility to override rpath at runtime using LD_LIBRARY_PATH usually what we want)
  - DT_RUNPATH ELF entry
    - Supported in DL since 1999
  - Use: `cc -Wl,-rpath,some-dir-path -Wl,--enable-new-dtags`
    - Since binutils 2.24 (2013): inserts only DT_RUNPATH entry
    - Some distros (Ubuntu) default to -Wl,--enable-new-dtags
    - Before binutils 2.24, inserted DT_RUNPATH and DT_RPATH
      (to allow for old DLs that didn’t understand DT_RUNPATH)

- If both types of rpath list are embedded in an object, DT_RUNPATH has precedence (i.e., DT_RPATH is ignored)

Shared libraries can have rpath lists

- Shared libraries can themselves have dependencies
  - ⇒ can use -rpath linker option to embed rpath lists when building shared libraries
An object’s rpath list is private to the object

- Each object (the main program or a shared library) can have an rpath...
- An object’s (DT_RUNPATH) rpath is used for resolving only its own immediate dependencies
  - One object’s rpath doesn’t affect search for any other object’s dependencies
    - See example in shlibs/rpath_independent
- ⚠️ Old style rpath (DT_RPATH) behaves differently
  - One object’s DT_RPATH can be used for resolving dependencies of another object
Dynamic string tokens

- DL understands certain special strings in rpath list
  - Dynamic string tokens
  - Written as $NAME or ${NAME}
- $ORIGIN: expands to directory containing program or library
  - Write turn-key applications!
  - Installer unpacks tarball containing application with library in (say) a subdirectory; application can be linked with:
    
    ```
    cc -Wl,-rpath,'$ORIGIN/lib'
    ```

- Note use of quotes to prevent interpretation of $ by shell!
Dynamic string tokens

- $LIB$: expands to `lib` or `lib64`, depending on architecture
  - E.g., useful on multi-arch platforms to supply 32-bit or 64-bit library, as appropriate
- $PLATFORM$: expands to string corresponding to processor type (e.g., `x86_64`)
  - Rpath entry can include arch-specific directory component
- DL also understands these names in some other contexts
  - `LD_LIBRARY_PATH`, `LD_PRELOAD`, `LD_AUDIT`, and `dlopen()`; see `ld.so(8)`
Finding shared libraries at run time

When resolving dependencies in dynamic dependency list, DL deals with each dependency string as follows:

- If the string contains a slash ⇒ interpret dependency as a relative or absolute pathname
- Otherwise, search for shared library using these rules
  - If calling object has **DT_RPATH** list and does **not** have **DT_RUNPATH** list, search directories in **DT_RPATH** list
  - If **LD_LIBRARY_PATH** defined, search directories it specifies
    - For security reasons, **LD_LIBRARY_PATH** is ignored in “secure” mode (set-UID and set-GID programs, etc.)
  - If calling object has **DT_RUNPATH** list, search directories in that list
  - Check **/etc/ld.so.cache** for a corresponding entry
  - Search **/lib** and **/usr/lib** (in that order)
    - Or **/lib64** and **/usr/lib64**
The directory `shlibs/mysleep` contains two files:

- `mysleep.c`: implements a function, `mysleep(nsecs)`, which prints a message and calls `sleep()` to sleep for `nsecs` seconds.
- `mysleep_main.c`: takes one argument that is an integer string. The program calls `mysleep()` with the numeric value specified in the command-line argument.

Using these files, perform the following steps to create a shared library and executable in the same directory:

- Build a shared library from `mysleep.c`. (You do not need to create the library with a soname or to create the linker and soname symbolic links.)
- Compile and link `mysleep_main.c` against the shared library to produce an executable that embeds an rpath list with the run-time location of the shared library (specified as an absolute path, e.g., use the value of `$PWD`).

[Exercise continues on next slide]

Verify that you can successfully run the executable without the use of `LD_LIBRARY_PATH`.

Try moving (not copying!) both the executable and the shared library to a different directory. What now happens when you try to run the executable? Why?

Now employ an rpath list that uses the `$ORIGIN` string:

- Modify the previous example so that you create an executable with an rpath list containing the string `$ORIGIN/sub`. ⚠️ Remember to use single quotes around `$ORIGIN`!
- Copy the executable to some directory, and copy the shared library to a subdirectory, `sub`, under that directory. Verify that the program runs successfully.
- If you move both the executable and the directory `sub` (which still contains the shared library) to a different location, is it still possible to run the executable?
Run-time symbol resolution

- Suppose main program and shared library both define a function `xyz()`, and another function inside library calls `xyz()`

```
prog
xyz()
{
    printf("main-xyz\n");
}
main()
{
    func();
}

libfoo.so
xyz()
{
    printf("foo-xyz\n");
}
func()
{
    xyz();
}
```

- To which symbol does reference to `xyz()` resolve?
- The results may seem a little surprising:

```bash
$ cc -g -c -fPIC -Wall foo.c
$ cc -g -shared -o libfoo.so foo.o
$ cc -g -o prog prog.c libfoo.so
$ LD_LIBRARY_PATH=. ./prog
main-xyz
```

- Definition in main program overrides version in library!
Symbol interposition

- When a symbol definition inside an object is overridden by an outside definition, we say symbol has been interposed.
  - Interposition can occur for both functions and variables.
- Surprising, but good historical reason for this behavior.
- Shared libraries are designed to mirror traditional static library semantics:
  - Definition of global symbol in main program overrides version in library.
  - Global symbol appears in multiple libraries?
    - \( \Rightarrow \) reference is resolved to first definition when scanning libraries in left-to-right order as specified in static link command line.
- Interposition behavior made transition from static to shared libraries easier.

Interposition vs libraries as self-contained subsystems

- Symbol interposition semantics conflict with model of shared library as a self-contained subsystem:
  - Shared library can’t guarantee that reference to its own global symbols will bind to those symbols at run time.
  - Properties of shared library may change when it is aggregated into larger system.
- Can sometimes be desirable to force symbol references within a shared library to resolve to library’s own symbols.
  - I.e., prevent interposition by outside symbol definition.
Forcing global symbol references to resolve inside library

- **-Bsymbolic** linker option causes references to global symbols within shared library to resolve to library's own symbols

$ cc -g -c -fPIC -Wall foo.c
$ cc -g -shared -Wl,-Bsymbolic -o libfoo.so foo.o
$ cc -g -o prog prog.c libfoo.so
$ LD_LIBRARY_PATH=. ./prog

foo-xyz

- ELF **DT_SYMBOLIC** tag
- ❗️ Affects **all** symbols in shared library! 😊
- Other techniques can provide this behavior on a per-symbol basis 😊
  - (Described later)

Symbol resolution and library load order

```
......main_prog........
/    |    \
libx1.so  liby1.so  libz1.so
  |    abc(){...}    |  call abc()
  |    |    |
libx2.so  liby2.so  libz2.so
  abc(){...}  xyz(){...}  |
  xyz(){...}  libz3.so
                  xyz(){...}
```

- Main program has three dynamic dependencies
- Some libraries on which main has dependencies in turn have dependencies
  - **Note**: main program has no direct dependencies other than `libx1.so, liby1.so, and libz1.so`
    - Likewise, `libz1.so` has no direct dependency on `libz3.so`
Symbol resolution and library load order

For the given arrangement of libraries:

```
........main_prog........
/ |  |  
libx1.so liby1.so libz1.so
|  | abc(){}...
|  |
libx2.so liby2.so libz2.so
|   |
abc(){}...
|   |
xyz(){}...
```

- **libx1.so** and **liby1.so** both define public function `abc()`.
- **libx2.so** and **liby2.so** both define public function `xyz()`.
- When `abc()` is called from inside **libz1.so**, which instance of `abc()` is invoked?
  - Call to `abc()` resolves to definition in **liby1.so**

---

**Dependent libraries are added in **breadth-first order****

- Immediate dependencies of main program are loaded first.
- Then dependencies of those dependencies, and so on.
  - Libraries that are already loaded are skipped (but are reference counted).
- Symbols are resolved by searching libraries in load order.
Symbol resolution and library load order

A quiz...

- **libx2.so**, **liby2.so**, and **libz3.so** all define public function **xyz()**
- When **xyz()** is called from inside **libz1.so**, which instance of **xyz()** is invoked?
  - Call to **xyz()** resolves to definition in **libx2.so**

Link-map lists ("namespaces")

- Set of shared objects that have been loaded by application is recorded on a **link-map list** (AKA "namespace")
  - Doubly linked list that is arranged in library load order
  - See definition of **struct link_map** in **<link.h>**
  - **dl_iterate_phdr(3)** can be used to iterate through link map
    - (Manual page has an example program)
  - See also **dlinfo(3)**, which obtains info about a dynamically loaded object
The LD_DEBUG environment variable

- **LD_DEBUG** can be used to monitor operation of dynamic linker
- **LD_DEBUG="value" prog**
- To list **LD_DEBUG** options, without executing program:

  ```
  $ LD_DEBUG=help ./prog
  Valid options for the LD_DEBUG environment variable are:
  
<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>libs</td>
<td>display library search paths</td>
</tr>
<tr>
<td>reloc</td>
<td>display relocation processing</td>
</tr>
<tr>
<td>files</td>
<td>display progress for input file</td>
</tr>
<tr>
<td>symbols</td>
<td>display symbol table processing</td>
</tr>
<tr>
<td>bindings</td>
<td>display information about symbol binding</td>
</tr>
<tr>
<td>versions</td>
<td>display version dependencies</td>
</tr>
<tr>
<td>scopes</td>
<td>display scope information</td>
</tr>
<tr>
<td>all</td>
<td>all previous options combined</td>
</tr>
<tr>
<td>statistics</td>
<td>display relocation statistics</td>
</tr>
<tr>
<td>unused</td>
<td>determined unused DSOs</td>
</tr>
<tr>
<td>help</td>
<td>display this help message and exit</td>
</tr>
</tbody>
</table>
  
  To direct the debugging output into a file instead of standard output a filename can be specified using the LD_DEBUG_OUTPUT environment variable.

- TLPI §42.6
Exercises

The files in the directory `shlibs/sym_res_load_order` set up the scenario shown earlier under the heading *Symbol resolution and library load order* (slide 4-27). (You can inspect the source code used to build the various shared libraries to verify this.) The `main` program uses `dl_iterate_phdr()` to display the link-map order of the loaded shared objects.

1. Use `make(1)` to build the shared libraries and the main program, and use the following command to run the program in order to verify the link-map order and also to see which versions of `abc()` and `xyz()` are called from inside `libz1.so`:

   ```
   LD_LIBRARY_PATH=. ./main
   ```

2. Run the program using `LD_DEBUG=libs` and use the dynamic linker’s debug output to verify the order in which the shared libraries are loaded.

   [Exercise continues on the next slide]

Exercises

3. Run the program using `LD_DEBUG=symbols` and use the dynamic linker’s debug output to discover which definitions the calls to `abc()` and `xyz()` bind to.

4. The order in which the immediate dependencies of `main` are loaded into the link map is determined by the order that the libraries are specified in the link command that is used to build `main`. Verify this as follows:
   - Modify the `Makefile` to rearrange the order in which the libraries are specified in the command that builds `main` to be: `libz1.so liby1.so libx1.so`
   - Remove the executable using `make clean`.
   - Rebuild the executable using `make`.
   - Run the executable again, and note the difference in the link-map order that is displayed by `dl_iterate_phdr()`.
