The Dynamic Linker

Michael Kerrisk, man7.org © 2024

January 2024

mtk@man7.org

Outline

4 The Dynamic Linker 4-1
4.1 The dynamic linker 4-3
4.2 Rpath: specifying library search paths in an object 4-5
4.3 Dynamic string tokens 4-12
4.4 Finding shared libraries at run time 4-18
4.5 Symbol resolution and symbol interposition 4-24
4.6 Link-map lists (namespaces) 4-35
4.7 The global look-up scope 4-37
4.8 Debugging the operation of the dynamic linker 4-40
The dynamic linker

- **Dynamic linker** (DL) == run-time linker == **loader**
  - Loads shared libraries needed by program and performs symbol relocations
  - Is itself a shared library, but special:
    - Loaded (by kernel) early in execution of a program
    - Is statically linked (thus, it has no dependencies itself)
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:
  - \texttt{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 (\texttt{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

[TLPI §41.10]
Defining an rpath list when linking

- To embed an rpath list in an executable, use the `-rpath` linker option
  - 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'
  RUNPATH       /home/mtk/tlpi/code/shlibs/demo
$ ./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` entry in `.dynamic` ELF section
- 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 entry in .dynamic ELF section
    - 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 (e.g., Ubuntu, Fedora) 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!
  - The DT_RPATH of object A can be used to find libraries needed by objects in dependency tree of A
  - See example in shlibs/rpath_dt_rpath
Dynamic string tokens

- DL understands certain special strings in rpath list
  - **Dynamic string tokens**
  - Written as $\text{Name}$ or ${\text{Name}}$

- DL also understands these names in some other contexts
  - `LD_LIBRARY_PATH`, `LD_PRELOAD`, `LD_AUDIT`
  - `DT_NEEDED` (i.e., in dependency lists)
    - See example in `shlibs/dt_needed_dst`
  - `dlopen()`
  - See `ld.so(8)`
Dynamic string tokens

- **$ORIGIN**: expands to directory containing program or library
  - Write turn-key applications!
  - Installer unpacks tarball containing application with library in (say) a subdirectory; application can be linked with:
    ```
    cc -Wl,-rpath,'$ORIGIN/lib'
    ```
  - Use quotes to prevent interpretation of $ by shell!
  - Example: `shlibs/shlib_origin_dst`

---

Dynamic string tokens

- **$ORIGIN** is generally **ignored in privileged programs**
  - Set-UID / set-GID / file capabilities
  - Prevents security vulnerabilities based on creation of hard links to privileged programs
  - Exception: $ORIGIN expansion that leads to path in trusted directory (e.g., `/lib64`) is permitted
    - E.g., allows binary in `/bin` with rpath such as $ORIGIN/../$LIB/sub
  - See comments in glibc’s `elf/dl-load.c` and https://amir.rachum.com/shared-libraries/
Dynamic string tokens

Other dynamic string tokens:

- **$LIB**: expands to `lib` or `lib64`, depending on architecture
  - E.g., useful on multi-arch platforms to build/supply 32-bit or 64-bit library, as appropriate

- **$PLATFORM**: expands to string corresponding to processor type (e.g., `x86_64`, `i386`, `i686`, `aarch64`, `aarch64_be`)
  - Rpath entry can include arch-specific directory component
    - E.g., on IA-32, could provide different optimized library implementations for `i386` vs `i686`
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`
Exercises

1. 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]

2. Verify that you can successfully run the executable without the use of `LD_LIBRARY_PATH`.
   - If you find that you can’t run the executable successfully, you may be able to debug the problem by inspecting the rpath of the executable:

   ```bash
   objdump -p mysleep_main | grep 'R[UN]*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?

[Exercise continues on next slide]
Exercises

- 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?

- Suppose you make the executable set-UID-`root` as follows:

  ```bash
  sudo chown root mysleep_main
  sudo chmod u+s mysleep_main
  ```

  What happens when you now try 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:

```
$ cd shlibs/sym_res_demo
$ 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?
    - ⇒ 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

- \texttt{-Bsymbolic} linker option causes references to global symbols within shared library to resolve to library's own symbols

\begin{verbatim}
$ cd shlibs/sym_res_demo
$ 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
\end{verbatim}

- Adds ELF DF_SYMBOLIC flag in .dynamic section of object
- Or DT_SYMBOLIC tag in older binaries
- To see if object was built with this option, use either of:

\begin{verbatim}
objdump -p libfoo.so | grep SYMBOLIC
readelf -d libfoo.so | grep SYMBOLIC
\end{verbatim}

- DF_SYMBOLIC flag in a library affects only the library itself (not dependencies of the library)
- More extensive example: \texttt{shlibs/demo_Bsymbolic}

Forcing global symbol references to resolve inside library

- \textbf{⚠️} Problem: \texttt{-Bsymbolic} affects \textit{all} symbols in shared library! 😊
  - And there are other problems...
- Preferable to control “local reference binds to local definition” behavior on a per-symbol basis
  - Other techniques (described later) allow this 😊
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`

- `libx2.so` and `liby1.so` both define public function `abc()`

- When `abc()` is called from inside `libz1.so`, which instance of `abc()` is invoked?
  - Call to `abc()` resolves to definition in `liby1.so`
Symbol resolution and library load order

The main program:

```
......main_prog.......  
/ |  \                      
lib1.so  liby1.so  libz1.so
  |    |    |                     
  abc(){...}    |    |  call abc()   
  |    |                        
lib2.so  liby2.so  libz2.so
  abc(){...}  xyz(){...}    |
  |                |    |                     
  xyz(){...}          |    |
                  libz3.so         |
                          xyz(){...}
```

- 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

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`
The set of all objects that have been loaded by application is recorded in a **link-map list** (AKA “namespace”)

- Doubly linked **list that is arranged in library load order**
  - Main program is at front of link map
- See definition of **struct link_map** in `<link.h>`
- **dl_iterate_phdr(3)** can be used to iterate through list
  - Example program: `shlibs/dl_iterate_phdr`
- See also **dlinfo(3)**, which obtains info about a dynamically loaded object
The global look-up scope

- In most cases, symbol resolution is performed via an ordered search of objects listed in the global look-up scope (GLS).
- GLS is a list of following objects (in this order):
  - The main program
  - All dependencies of main, loaded in breadth-first order
  - Libraries opened with `dlopen(RTLD_GLOBAL)`
    - And dependencies, added in breadth-first order
- An object appears only once in the GLS
  - E.g., `dlopen()` of a library already in GLS won’t add library a second time
- Order of objects in GLS is similar to link-map list order
  - But GLS does not include libraries opened with `dlopen(RTLD_LOCAL)`
Other look-up scopes

- In certain cases, symbol look-ups may search other scopes
  - E.g., “local” scope and “self” scope
  - See the discussion of Look-up scopes (later)
- An object’s **look-up scope(s)** == set of all scopes that might be searched when performing relocations for the object
The LD_DEBUG environment variable

- **LD_DEBUG** can be used to trace operation of dynamic linker
  - **LD_DEBUG=**"value" prog
    - *value* is one or more words separate by space/comma/colon
  - Ignored (for security reasons) in privileged programs
  - To send trace output to file (instead of stderr), use **LD_DEBUG_OUTPUT=path**
  - 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>
The LD_DEBUG environment variable

- **libs**: show locations where each library is searched for
- **files**: emit message as each library is opened
- **reloc**: emit message at start of relocation processing for each object
- **symbols**: for each symbol relocation, show which library symbol tables are searched
- **bindings**: for each symbol relocation, show object containing definition to which symbol binds
  - Corresponds to final entry shown by `symbols` (unless symbol is undefined)
- **versions**: display version dependency checks that are performed for each object
  - Relates to symbol-versioned libraries

All of preceding also cause DL to display messages when
- Each object’s constructors and destructors are executed
- On transfer of control to `main()`

- **scopes**: display search scopes for symbol relocation (objects that will be searched during relocation for this object)
  - See the discussion of Look-up scopes (later)
- **unused**: used to implement “`ldd -u`” (in conjunction with setting `LD_TRACE_LOADED_OBJECTS=1`
LD_DEBUG example

(Abridged) example of output from LD_DEBUG:

$ LD_DEBUG="reloc symbols bindings" ./prog
...
32150: relocation processing: ./prog
...
32150: symbol=x; lookup in file=./prog [0]
32150: symbol=x; lookup in file=./libdemo.so.1 [0]
32150: binding file ./prog [0] to ./libdemo.so.1 [0]: normal symbol `x'

- "relocation processing" message from reloc
  - One message per library
- "symbol...lookup in file" messages from symbols
  - One group of messages for each symbol relocation
- "binding file...symbol" message from bindings
  - One message for each relocated symbol, showing origin of reference, object containing defn, and symbol name
- Number at start of each line is PID of process

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 4-34). (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, and which locations are searched for each library.

$ LD_DEBUG=libs LD_LIBRARY_PATH=. ./main 2>&1 | less

[Exercise continues on the next slide]
Exercises

3 Run the program and use the dynamic linker’s debug output to show which libraries are searched and what definitions are finally bound for the calls to `abc()` and `xyz()`.

```
$ LD_DEBUG="symbols bindings" LD_LIBRARY_PATH=. ./main 2>&1 | less
```

4 The order in which the dependencies of `main` appear in the global look-up scope is determined by the order that the libraries are specified in the link command used to build `main`. Verify this as follows:

- **Modify the last target** in 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 symbol binding for the call to `xyz()` and the differences in the link-map order that is displayed by `dl_iterate_phdr()`.

Notes