Building and Using Shared Libraries on Linux

The Dynamic Linker

Michael Kerrisk, man7.org © 2026

mtk@man7.org

January 2026

Outline

Rev: # d49d0b54be95

Outline

The dynamic linker

- **Dynamic linker** (DL) == run-time linker == **loader**
- Loads shared libraries needed by program
- Performs symbol relocations
  - By examining dynamic symbol tables (`.dynsym`) of all objects
- 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)

Outline

6 The Dynamic Linker 6-1

6.1 The dynamic linker 6-3

6.2 Rpath: specifying library search paths in an object 6-5

6.3 Dynamic string tokens 6-12

6.4 Finding shared libraries at run time 6-17

6.5 How programs get run 6-19

6.6 Exercises 6-23

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

[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/lsp/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 type of rpath list has higher precedence
  - DT_RPATH entry in .dynamic ELF section
  - This 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
    - Before *binutils* 2.24, inserted `DT_RUNPATH` **and** `DT_RPATH` (to allow for old DLs that didn't understand `DT_RUNPATH`)
    - Some distros (e.g., Ubuntu, Fedora) default to *–Wl,--enable-new-dtags*
- If both types of rpath list are embedded in an object, **`DT_RUNPATH` has precedence** (i.e., `DT_RPATH` is ignored)

man7.org

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

man7.org

An object's rpath list is private to the object

- Each object (`main` or a shared library) can have an rpath...
- An object's (`DT_RUNPATH`) rpath is used for resolving only its own immediate dependencies
  - E.g., suppose that:
    - `main` depends on `libX.so` and has rpath that specifies where to find `libX.so`
    - `libX.so` depends on `libY.so`, but has no rpath
    - Rpath of `main` isn't used to help find dependency of `libX.so`
    - See example in `shlibs/rpath_independent`
  - Old style rpath (`DT_RPATH`) behaves differently!
    - The `DT_RPATH` of object A can be used to find objects needed by libraries in dependency tree of A
    - See example in `shlibs/rpath_dt_rpath`

Outline

6 The Dynamic Linker 6-1

6.1 The dynamic linker 6-3

6.2 Rpath: specifying library search paths in an object 6-5

6.3 Dynamic string tokens 6-12

6.4 Finding shared libraries at run time 6-17

6.5 How programs get run 6-19

6.6 Exercises 6-23

Dynamic string tokens

- DL understands certain special strings in rpath list
  - **Dynamic string tokens**
  - Written as `$NAME` or `${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)*

man7.org

Dynamic string tokens

- `$ORIGIN`: expands to directory containing program or library
  - Allow us to write “turn-key” applications:
    - Installer unpacks tarball containing application with library in (say) a subdirectory
    - Application can be executed without installing library in “standard” location
  - Application can be linked with:

```
cc -Wl,-rpath,'$ORIGIN/lib'
```

    - ⚠ ⚠ Use quotes to prevent interpretation of `$` by shell!
  - Example: `shlibs/shlib_origin_dst`

man7.org

Dynamic string tokens

- $ORIGIN is generally **ignored in privileged programs**
  - Privileged = 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
  - On Debian/Ubuntu expands to (on x86 platforms): lib32 or lib/x86_64-linux-gnu
- $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

Outline

6 The Dynamic Linker 6-1

6.1 The dynamic linker 6-3

6.2 Rpath: specifying library search paths in an object 6-5

6.3 Dynamic string tokens 6-12

6.4 Finding shared libraries at run time 6-17

6.5 How programs get run 6-19

6.6 Exercises 6-23

Finding shared libraries at run time

When resolving dependencies in an object's 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
  1. If object has `DT_RPATH` list and does **not** have `DT_RUNPATH` list, search directories in `DT_RPATH` list
  2. If `LD_LIBRARY_PATH` defined, search directories it specifies
     - For security reasons, `LD_LIBRARY_PATH` is ignored in secure-execution mode (set-UID and set-GID programs, programs with capabilities)
  3. If object has `DT_RUNPATH` list, search directories in that list
  4. Check `/etc/ld.so.cache` for a corresponding entry
  5. Search `/lib` and `/usr/lib` (in that order)
     - Or `/lib64` and `/usr/lib64`

[TLPI §41.11]

Outline

Steps in execution of a program

1. A **process calls *execve()***, specifying ELF file to execute
2. Kernel's ELF program loader[†] **reinitializes process image** based on contents of ELF file
    - **Builds segments of process** based on program header table (PHT) and ELF sections
        - Kernel has only a rudimentary understanding of ELF format (It knows "just enough")
    - Constructs **auxiliary vector** (AV) at top of process address space
        - AV is a table of key-value pairs containing info that is useful primarily for DL
        - AV sits just past end of *environ* (see `proc/auxvec.c`)

Steps in execution of a program

3. If PHT contains `PT_INTERP` entry, **kernel loads interpreter** into process address space and passes control to it
   - Loading of interpreter proceeds similarly to loading program
   - `PT_INTERP` entry normally specifies the **dynamic linker**
4. Dynamic linker:
   - Examines `DT_NEEDED` entries in ELF image
   - Loads specified shared libraries
     - Iteratively, since libraries may in turn have `DT_NEEDED` entries
   - Performs relocations
     - DL has much more detailed understanding of ELF format
   - Passes control to program entry point
     - Entry point is obtained from auxiliary vector (`AT_ENTRY`)

Further information

- *How programs get run: ELF binaries*, D. Drysdale, 2015, `http://lwn.net/Articles/631631/`
- *A look at dynamic linking*, D. Allen, 2024, `https://lwn.net/Articles/961117/`
- *getauxval() and the auxiliary vector*, M. Kerrisk, 2012, `http://lwn.net/Articles/519085/`

Outline

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. (You may find it easiest the write a script to perform the necessary commands to build the shared library and executable; you can then modify that script in the next exercise.)
   - 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*).
   - 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:

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

Exercises

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

2. 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 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:

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

      Suppose you now try to run the executable. You should find that the library fails to load because `$ORIGIN` is ignored in set-UID programs.
        - If you *don't* encounter a failure, it may be because your filesystem is mounted with the `nosuid` option. You can check this as follows:

          ```
          findmnt -T <dir>.
          ```

This page intentionally blank