Learn Linux, 101: Manage shared libraries

Find and load the libraries a program needs

Ian Shields

August 18, 2015
(First published March 10, 2010)

Learn how to determine which shared libraries your Linux® executable programs depend on and how to load them. You can use the material in this tutorial to study for the LPI 101 exam for Linux system administrator certification, or just to learn for fun.

View more content in this series

Overview

In this tutorial, learn to find and load the shared libraries that your Linux programs need. Learn to:

- Determine which libraries a program needs
- Know how the system finds shared libraries
- Load shared libraries

Shared libraries

When you write a program, you rely on many pieces of code that someone else has already written to perform routine or specialized functions for you. These pieces of code are stored in shared libraries. To use them, you link them with your code, either when you build the program or when you run the program.

About this series

This series of tutorials helps you learn Linux system administration tasks. You can also use the material in these tutorials to prepare for the Linux Professional Institute's LPIC-1: Linux Server Professional Certification exams.

See "*Learn Linux, 101*: A roadmap for LPIC-1" for a description of and link to each tutorial in this series. The roadmap is in progress and reflects the version 4.0 objectives of the LPIC-1

exams as updated April 15th, 2015. As tutorials are completed, they will be added to the roadmap.

This tutorial helps you prepare for Objective 102.3 in Topic 102 of the Linux Server Professional (LPIC-1) exam 101. The objective has a weight of 1.

Prerequisites

To get the most from the tutorials in this series, you should have a basic knowledge of Linux and a working Linux system on which you can practice the commands covered in this tutorial. Sometimes different versions of a program format output differently, so your results might not always look exactly like the listings and figures shown here. In particular, many of the examples in this tutorial come from 64-bit systems. I have included some examples from 32-bit systems to illustrate significant differences.

Static and dynamic linking

Linux systems have two types of executable programs:

- **Statically linked executables**: Contain all the library functions that they need to execute; all library functions are linked into the executable. They are complete programs that do not depend on external libraries to run. One advantage of statically linked programs is that they work without your needing to install prerequisites.
- **Dynamically linked executables**: Much smaller programs; they are incomplete in the sense that they require functions from external shared libraries to run. Besides being smaller, dynamic linking permits a package to specify prerequisite libraries without needing to include the libraries in the package. By using dynamic linking, many running programs can share one copy of a library rather than occupying memory with many copies of the same code. For these reasons, most programs today use dynamic linking.

An interesting example on many Linux systems is the `ln` command (/bin/ln), which creates links between files, either *hard* links or *soft* (or *symbolic*) links. This command uses shared libraries. Shared libraries often involve symbolic links between a generic name for the library and a specific level of the library, so if the links are not present or broken for some reason, then the `ln` command itself might be inoperative, creating a circular problem. To protect against this possibility, some Linux systems include a statically linked version of the `ln` program as the `sln` program (/sbin/sln). Listing 1 illustrates the great difference in size between the dynamically linked `ln` and the statically linked `sln`. The example is from a Fedora 22 64-bit system.

Listing 1. Sizes of sln and ln

```
[ian@atticf20 ~]$ # Fedora 22 64-bit
[ian@atticf20 ~]$ ls -l /sbin/sln /bin/ln
-rwxr-xr-x. 1 root root  58656 May 14 04:56 /bin/ln
-rwxr-xr-x. 1 root root 762872 Feb 23 10:36 /sbin/sln
```

Which libraries are needed?

Though not part of the current LPI exam requirements for this topic, you should know that many Linux systems today run on hardware that supports both 32-bit and 64-bit executables. Many

libraries are thus compiled in 32-bit and 64-bit versions. The 64-bit versions are usually stored under the /lib64 tree in the filesystem, while the 32-bit versions live in the traditional /lib tree. You will probably find both /lib/libc-2.11.1.so and /lib64/libc-2.11.1.so on a typical 64-bit Linux system. These two libraries allow both 32-bit and 64-bit C programs to run on a 64-bit Linux system.

The `ldd` command

Apart from knowing that a statically linked program is likely to be large, how can you tell whether a program is statically linked? And if it is dynamically linked, how do you know what libraries it needs? The `ldd` command can answer both questions. If you are running a system such as Debian or Ubuntu, you probably don't have the sln executable, so you might also want to check the /sbin/ldconfig executable. Listing 2 shows the output of the `ldd` command for the ln and sln executables and also the ldconfig executable. The example is from a Fedora 22 64-bit system (atticf20). For comparison, the output from Ubuntu 14 32-bit system (attic-u14) is shown for /bin/ln.

Listing 2. Output of `ldd` for sln and ln

```
[ian@atticf20 ~]$ # Fedora 22 64-bit
[ian@atticf20 ~]$ ldd /sbin/sln /sbin/ldconfig /bin/ln
/sbin/sln:
 not a dynamic executable
/sbin/ldconfig:
 not a dynamic executable
/bin/ln:
 linux-vdso.so.1 (0x00007ffedd31e000)
 libc.so.6 => /lib64/libc.so.6 (0x00007f2d3bd5d000)
 /lib64/ld-linux-x86-64.so.2 (0x00007f2d3c11d000)

ian@attic-u14:~/data/lpic-1$ # Ubuntu 14 32-bit
ian@attic-u14:~/data/lpic-1$ ldd /bin/ln
 linux-gate.so.1 =>  (0xb779d000)
 libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb75d7000)
 /lib/ld-linux.so.2 (0xb77a0000)
```

Because `ldd` is actually concerned with dynamic linking, it tells us that both sln and ldconfig are statically linked by telling us that they are "not a dynamic executable," while it tells us the names of three shared libraries (linux-vdso.so.1, libc.so.6, and /lib64/ld-linux-x86-64.so.2) that the `ln` command needs. Note that *.so* indicates that these are *shared objects* or dynamic libraries. This output also illustrates three different types of information you are likely to see.

linux-vdso.so.1
: is the Linux *Virtual Dynamic Shared Object*, which I discuss in a moment. You can also see linux-gate.so.1 as in the Ubuntu 14 example.

libc.so.6
: has a pointer to /lib64/libc.so.6 or /lib/i386-linux-gnu/libc.so.6. You can also see this pointing to /lib/libc.so.6 on older 32-bit systems.

/lib64/ld-linux-x86-64.so.2
: is the absolute path to another library.

In Listing 3, I use the `ls -l` command to show that the last two libraries are, in turn, symbolic links to specific versions of the libraries. The example is from a Fedora 22 64-bit system. This allows library updates to be installed without the need to relink all the executables that use the library.

Listing 3. Library symbolic links

```
[ian@atticf20 ~]$ # Fedora 22 64-bit
[ian@atticf20 ~]$ ls -l /lib64/libc.so.6 /lib64/ld-linux-x86-64.so.2
lrwxrwxrwx. 1 root root 10 Feb 23 10:33 /lib64/ld-linux-x86-64.so.2 -> ld-2.21.so
lrwxrwxrwx. 1 root root 12 Feb 23 10:33 /lib64/libc.so.6 -> libc-2.21.so
```

Linux Virtual Dynamic Shared Objects

In the early days of x86 processors, communication from user programs to supervisor services occurred through a software interrupt. As processor speeds increased, this became a serious bottleneck. Starting with Pentium® II processors, Intel® introduced a *Fast System Call* facility to speed up system calls using the SYSENTER and SYSEXIT instructions instead of interrupts.

The library that you see as linux-vdso.so.1 is a virtual library, or Virtual Dynamic Shared Object, that is located only in each program's address space. Some systems call this linux-gate.so.1. This virtual library provides the necessary logic to allow user programs to access system functions through the fastest means available on the particular processor, either interrupt, or with most newer processors, fast system call.

Dynamic loading

From the preceding, you might be surprised to learn that /lib/ld-linux.so.2 and its 64-bit cousin, /lib64/ld-linux-x86-64.so.2, which both look like shared libraries, are actually executables in their own right. They are the code that is responsible for dynamic loading. They read the header information from the executable, which is in the Executable and Linking Format (ELF) format. From this information, they determine what libraries are required and which ones need to be loaded. They then perform dynamic linking to fix up all the address pointers in your executable and the loaded libraries so that the program will run.

The man page for ld-linux.so also describes ld.so, which performed similar functions for the earlier *a.out* binary format. Listing 4 illustrates using the `--list` option of the ld-linux.so cousins to show the same information for the `ln` command that Listing 2 showed with the `ldd` command.

Listing 4. Using ld-linux.so to display library requirements

```
[ian@atticf20 ~]$ # Fedora 22 64-bit
[ian@atticf20 ~]$ /lib64/ld-linux-x86-64.so.2 --list /bin/ln
 linux-vdso.so.1 (0x00007ffe725f6000)
 libc.so.6 => /lib64/libc.so.6 (0x00007f2179b5d000)
 /lib64/ld-linux-x86-64.so.2 (0x00007f2179f1d000)

ian@attic-u14:~$ # Ubuntu 14 32-bit
ian@attic-u14:~$ /lib/ld-linux.so.2 --list /bin/ln
 linux-gate.so.1 =>  (0xb77bc000)
 libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb75f6000)
 /lib/ld-linux.so.2 (0xb77bf000)
```

Note that the hex addresses might be different between the two listings. They are also likely to be different if you run `ldd` twice.

Dynamic library configuration

So how does the dynamic loader know where to look for executables? As with many things on Linux, there is a configuration file in /etc. In fact, there are two configuration files: /etc/ld.so.conf and /etc/ld.so.cache. Listing 5 shows the contents of /etc/ld.so.conf on a 64-bit Fedora 22 system. Note that /etc/ld.so.conf specifies that all the .conf files from the subdirectory ld.so.conf.d should be included. Older systems might have all entries in /etc/ld.so.conf and not include entries from the /etc/ld.so.conf.d directory. The actual contents of /etc/ld.so.conf or the /etc/ld.so.conf.d directory might be different on your system.

Listing 5. Content of /etc/ld.so.conf

```
[ian@atticf20 ~]$ # Fedora 22 64-bit
[ian@atticf20 ~]$ cat /etc/ld.so.conf
include ld.so.conf.d/*.conf
[ian@atticf20 ~]$ ls /etc/ld.so.conf.d/*.conf
/etc/ld.so.conf.d/atlas-x86_64.conf
/etc/ld.so.conf.d/bind99-x86_64.conf
/etc/ld.so.conf.d/kernel-4.0.4-301.fc22.x86_64.conf
/etc/ld.so.conf.d/kernel-4.0.4-303.fc22.x86_64.conf
/etc/ld.so.conf.d/kernel-4.0.6-300.fc22.x86_64.conf
/etc/ld.so.conf.d/libiscsi-x86_64.conf
/etc/ld.so.conf.d/llvm-x86_64.conf
/etc/ld.so.conf.d/mariadb-x86_64.conf
```

Program loading needs to be fast, so use the `ldconfig` command to process the ld.so.conf file and all the included files from ld.so.conf.d and libraries from the trusted directories, /lib and /usr/lib, and any others supplied on the command line. The `ldconfig` command creates the necessary links and cache to recently used shared libraries in /etc/ld.so.cache. The dynamic loader uses the cached information from ld.so.cache to locate files that are to be dynamically loaded and linked. If you change ld.so.conf (or add new included files to ld.so.conf.d), you must run the `ldconfig` command (as root) to rebuild your ld.so.cache file.

Normally, you use the ldconfig command without parameters to rebuild ld.so.cache. There are several other parameters you can specify to override this default behavior. As usual, try `man ldconfig` for more information. Listing 6 illustrates the use of the `-p` parameter to display the contents of ld.so.cache.

Listing 6. Using ldconfig to display ld.so.cache

```
[ian@atticf20 ~]$ # Fedora 22 64-bit
[ian@atticf20 ~]$ /sbin/ldconfig -p | less
1361 libs found in cache `/etc/ld.so.cache'
        p11-kit-trust.so (libc6,x86-64) => /lib64/p11-kit-trust.so
        libzeitgeist-2.0.so.0 (libc6,x86-64) => /lib64/libzeitgeist-2.0.so.0
        libzapojit-0.0.so.0 (libc6,x86-64) => /lib64/libzapojit-0.0.so.0
        libz.so.1 (libc6,x86-64) => /lib64/libz.so.1
        libyelp.so.0 (libc6,x86-64) => /lib64/libyelp.so.0
        libyaml-0.so.2 (libc6,x86-64) => /lib64/libyaml-0.so.2
        libyajl.so.2 (libc6,x86-64) => /lib64/libyajl.so.2
        libxtables.so.10 (libc6,x86-64) => /lib64/libxtables.so.10
        libxslt.so.1 (libc6,x86-64) => /lib64/libxslt.so.1
        libxshmfence.so.1 (libc6,x86-64) => /lib64/libxshmfence.so.1
        libxml2.so.2 (libc6,x86-64) => /lib64/libxml2.so.2
```

```
        libxmlrpc_util.so.3 (libc6,x86-64) => /lib64/libxmlrpc_util.so.3
        libxmlrpc_server_cgi.so.3 (libc6,x86-64) => /lib64/libxmlrpc_server_cgi.so.3
        libxmlrpc_server_abyss.so.3 (libc6,x86-64) => /lib64/libxmlrpc_server_abyss.so.3
        libxmlrpc_server.so.3 (libc6,x86-64) => /lib64/libxmlrpc_server.so.3
        libxmlrpc_client.so.3 (libc6,x86-64) => /lib64/libxmlrpc_client.so.3
        libxmlrpc_abyss.so.3 (libc6,x86-64) => /lib64/libxmlrpc_abyss.so.3
        libxmlrpc.so.3 (libc6,x86-64) => /lib64/libxmlrpc.so.3
        libxml-security-c.so.16 (libc6,x86-64) => /lib64/libxml-security-c.so.16
        libxlutil.so.4.3 (libc6,x86-64) => /lib64/libxlutil.so.4.3
        libxklavier.so.16 (libc6,x86-64) => /lib64/libxklavier.so.16
        libxkbfile.so.1 (libc6,x86-64) => /lib64/libxkbfile.so.1
:
```

Loading specific libraries

If you're running an older application that needs a specific older version of a shared library, or if you're developing a new shared library or version of a shared library, you might want to override the default search paths used by the loader. This might also be needed by scripts that use product-specific shared libraries that might be installed in the /opt tree.

Just as you can set the PATH variable to specify a search path for executables, you can set the LD_LIBRARY_PATH variable to a colon-separated list of directories that should be searched for shared libraries before the system ones specified in ld.so.cache. For example, you might use a command like:

```
export LD_LIBRARY_PATH=/usr/lib/oldstuff:/opt/IBM/AgentController/lib
```

See Related topics for additional details and links to other tutorials in this series.

This concludes our brief introduction to managing shared libraries on Linux.

Related topics

- Use the developerWorks roadmap for LPIC-1 to find the developerWorks tutorials to help you study for LPIC-1 certification based on the LPI Version 4.0 April 2015 objectives.
- At the Linux Professional Institute website, find detailed objectives, task lists, and sample questions for the certifications. In particular, see:
  - The LPIC-1: Linux Server Professional Certification program details
  - LPIC-1 exam 101 objectives
  - LPIC-1 exam 102 objectives

  Always refer to the Linux Professional Institute website for the latest objectives.
- The Linux Documentation Project has a variety of useful documents, especially its HOWTOs.