Learn Linux, 101: Manage shared libraries

Find and load the libraries a program needs

Ian Shields
Senior Programmer
IBM

31 August 2011
(First published 10 March 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 article to study for the LPI 101 exam for Linux system administrator certification, or just to learn for fun. [*Typographical errors noted by alert readers (see Comments at the end of this article) have been corrected, thanks! --Ed.*]

View more content in this series

About this series

This series of articles helps you learn Linux system administration tasks. You can also use the material in these articles to prepare for Linux Professional Institute Certification level 1 (LPIC-1) exams.

See our developerWorks roadmap for LPIC-1 for a description of and link to each article in this series. The roadmap is in progress and reflects the latest (April 2009) objectives for the LPIC-1 exams: as we complete articles, we add them to the roadmap. In the meantime, though, you can find earlier versions of similar material, supporting previous LPIC-1 objectives prior to April 2009, in our LPI certification exam prep tutorials.

Overview

In this article, 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

This article helps you prepare for Objective 102.3 in Topic 102 of the Linux Professional Institute's Junior Level Administration (LPIC-1) exam 101. The objective has a weight of 1.

Prerequisites

To get the most from the articles 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 article. Sometimes different versions of a program will format output differently, so your results may not always look

exactly like the listings and figures shown here. In particular, many of the examples in this article come from 64-bit systems. We have included some examples from 32-bit systems to illustrate significant differences.

Static and dynamic linking

Connect with Ian

Ian is one of our most popular and prolific authors. Browse all of Ian's articles on developerWorks. Check out Ian's profile and connect with him, other authors, and fellow readers in My developerWorks.

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 are much smaller programs; they are incomplete in the sense that they require functions from external *shared* libraries in order to run. Besides being smaller, dynamic linking permits a package to specify prerequisite libraries without needing to include the libraries in the package. The use of dynamic linking also allows many running programs to 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.

Develop skills on this topic

This content is part of a progressive knowledge path for advancing your skills. See Basics of Linux system administration: Setting up your system and software

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 12 64-bit system.

Listing 1. Sizes of sln and ln

```
[ian@echidna ~]$ ls -l /sbin/sln /bin/ln
-rwxr-xr-x. 1 root root  47384 2010-01-12 09:35 /bin/ln
-rwxr-xr-x. 1 root root 603680 2010-01-04 09:07 /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 12 64-bit system (echidna). For comparison, the output from an older Fedora 8 32-bit system (pinguino) is shown for /bin/ln.

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

```
[ian@echidna ~]$ #Fedora 12 64-bit
[ian@echidna ~]$ 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 =>  (0x00007fff644af000)
        libc.so.6 => /lib64/libc.so.6 (0x00000037eb800000)
        /lib64/ld-linux-x86-64.so.2 (0x00000037eb400000)

[ian@pinguino ~]$ # Fedora 8 32-bit
[ian@pinguino ~]$ ldd /bin/ln
        linux-gate.so.1 =>  (0x00110000)
        libc.so.6 => /lib/libc.so.6 (0x00a57000)
        /lib/ld-linux.so.2 (0x00a38000)
```

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 we will discuss in a moment. You may also see linux-gate.so.1 as in the Fedora 8 example.

libc.so.6
: has a pointer to /lib64/libc.so.6.

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

In Listing 3, we 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 12 64-bit system.

Listing 3. Library symbolic links

```
[ian@echidna ~]$ ls -l /lib64/libc.so.6 /lib64/ld-linux-x86-64.so.2
lrwxrwxrwx. 1 root root 12 2010-01-14 14:24 /lib64/ld-linux-x86-64.so.2 -> ld-2.11.1.so
lrwxrwxrwx. 1 root root 14 2010-01-14 14:24 /lib64/libc.so.6 -> libc-2.11.1.so
```

Linux Virtual Dynamic Shared Objects

In the early days of x86 processors, communication from user programs to supervisor services was performed 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 resides only in each program's address space. Older systems called 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* or (*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@echidna ~]$ /lib64/ld-linux-x86-64.so.2 --list /bin/ln
        linux-vdso.so.1 =>  (0x00007fffc9fff000)
        libc.so.6 => /lib64/libc.so.6 (0x00000037eb800000)
        /lib64/ld-linux-x86-64.so.2 (0x00000037eb400000)

[ian@pinguino ~]$ /lib/ld-linux.so.2 --list /bin/ln
        linux-gate.so.1 =>  (0x00110000)
        libc.so.6 => /lib/libc.so.6 (0x00a57000)
        /lib/ld-linux.so.2 (0x00a38000)
```

Note that the hex addresses may 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 12 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 may 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 may be different on your system.

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

```
[ian@echidna ~]$ cat /etc/ld.so.conf
include ld.so.conf.d/*.conf
[ian@echidna ~]$ ls /etc/ld.so.conf.d/*.conf
/etc/ld.so.conf.d/kernel-2.6.31.12-174.2.19.fc12.x86_64.conf
/etc/ld.so.conf.d/kernel-2.6.31.12-174.2.22.fc12.x86_64.conf
/etc/ld.so.conf.d/kernel-2.6.31.12-174.2.3.fc12.x86_64.conf
/etc/ld.so.conf.d/mysql-x86_64.conf
/etc/ld.so.conf.d/qt-x86_64.conf
/etc/ld.so.conf.d/tix-x86_64.conf
/etc/ld.so.conf.d/xulrunner-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 as well as 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@lyrebird ian]$ /sbin/ldconfig -p | less
1602 libs found in cache `/etc/ld.so.cache'
        libzip.so.1 (libc6,x86-64) => /usr/lib64/libzip.so.1
        libz.so.1 (libc6,x86-64) => /lib64/libz.so.1
        libz.so (libc6,x86-64) => /usr/lib64/libz.so
        libx86.so.1 (libc6,x86-64) => /usr/lib64/libx86.so.1
        libx11globalcomm.so.1 (libc6,x86-64) => /usr/lib64/libx11globalcomm.so.1
        libxul.so (libc6,x86-64) => /usr/lib64/xulrunner-1.9.1/libxul.so
        libxtables.so.2 (libc6,x86-64) => /usr/lib64/libxtables.so.2
        libxslt.so.1 (libc6,x86-64) => /usr/lib64/libxslt.so.1
        libxslt.so (libc6,x86-64) => /usr/lib64/libxslt.so
        libxpcom.so (libc6,x86-64) => /usr/lib64/xulrunner-1.9.1/libxpcom.so
        libxml2.so.2 (libc6,x86-64) => /usr/lib64/libxml2.so.2
        libxml2.so (libc6,x86-64) => /usr/lib64/libxml2.so
       ...
        libABRTdUtils.so.0 (libc6,x86-64) => /usr/lib64/libABRTdUtils.so.0
        libABRTUtils.so.0 (libc6,x86-64) => /usr/lib64/libABRTUtils.so.0
        ld-linux.so.2 (ELF) => /lib/ld-linux.so.2
        ld-linux-x86-64.so.2 (libc6,x86-64) => /lib64/ld-linux-x86-64.so.2
```

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 may also be needed by scripts that use product-specific shared libraries that may 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 the Resources below for additional details and links to other articles in this series.

Resources

Learn

- Use the developerWorks roadmap for LPIC-1 to find the developerWorks articles to help you study for LPIC-1 certification based on the April 2009 objectives.
- At the LPIC Program site, find detailed objectives, task lists, and sample questions for the three levels of the Linux Professional Institute's Linux system administration certification. In particular, see their April 2009 objectives for LPI exam 101 and LPI exam 102. Always refer to the LPIC Program site for the latest objectives.
- Review the entire LPI exam prep series on developerWorks to learn Linux fundamentals and prepare for system administrator certification based on earlier LPI exam objectives prior to April 2009.
- The Linux Documentation Project has a variety of useful documents, especially its HOWTOs.
- In the developerWorks Linux zone, find hundreds of articles, tutorials, discussion forums, and a wealth other resources for Linux developers and administrators.
- Stay current with developerWorks technical events and webcasts focused on a variety of IBM products and IT industry topics.
- Attend a free developerWorks Live! briefing to get up-to-speed quickly on IBM products and tools as well as IT industry trends.
- Watch developerWorks on-demand demos ranging from product installation and setup demos for beginners, to advanced functionality for experienced developers.
- Follow developerWorks on Twitter.

Get products and technologies

- Evaluate IBM products in the way that suits you best: Download a product trial, try a product online, use a product in a cloud environment, or spend a few hours in the SOA Sandbox learning how to implement Service Oriented Architecture efficiently.

Discuss

- Participate in the discussion forum for this content.
- Get involved in the My developerWorks community. Connect with other developerWorks users while exploring the developer-driven blogs, forums, groups, and wikis.

About the author

Ian Shields

Ian Shields works on a multitude of Linux projects for the developerWorks Linux zone. He is a Senior Programmer at IBM at the Research Triangle Park, NC. He joined IBM in Canberra, Australia, as a Systems Engineer in 1973, and has since worked on communications systems and pervasive computing in Montreal, Canada, and RTP, NC. He has several patents and has published several papers. His undergraduate degree is in pure mathematics and philosophy from the Australian National University. He has an M.S. and Ph.D. in computer science from North Carolina State University. Learn more about Ian in Ian's profile on developerWorks Community.