Dissecting shared libraries

Get to know your shared library

Peter Seebach January 11, 2005

Shared libraries use version numbers to allow for upgrades to the libraries used by applications while preserving compatibility for older applications. This article reviews what's really going on under the book jacket and why there are so many symbolic links in /usr/lib on a normal Linux™ system.

Shared libraries are a fundamental component for the efficient use of space and resources on a modern UNIX® system. The C library on a SUSE 9.1 system is made up of about 1.3 MB. A copy of that library for every program in /usr/bin (and I have 2,569!) would take up a couple of gigabytes of space.

Of course this number is inflated -- statically linked programs would incorporate only those parts of the library that they use. Nonetheless, the amount of space tied up by all those duplicate copies of `printf()` would give the system a very bloated feel.

Shared libraries can save memory, not just disk space. The kernel can keep a single copy of a shared library in memory, sharing it among multiple applications. So, not only do we only have one copy of `printf()` on the disk, we only have one in memory. That has a pretty noticeable effect on performance.

In this article, we'll review the underlying technology used for shared libraries and the way in which shared library versioning helps prevent the compatibility nightmares that naive shared library implementations have had in the past. First, a look at how shared libraries work.

How shared libraries work

The concept is easy enough to understand. You have a library; you share the library. But what actually happens when your program tries to call `printf()` -- the real way this works -- is a bit more complex.

It is a simpler process in a static linking system than in a dynamically linked system. In a static linked system, the generated code possesses a reference to a function. The linker replaces that reference with the actual address at which it had loaded the function, so that the resulting binary

code has the right address in place. Then, when the code is run, it simply jumps to the relevant address. This is a simple task to administer because it lets you to link in only those objects that are actually referred to at some point in the program.

But most shared libraries are dynamically linked. That has several further implications. One is that you can't predict in advance at which address a function will really be when it's called! (There have also been statically linked shared library schemes, such as the one in BSD/OS, but they are beyond the scope of this article.)

The dynamic linker can do a fair amount of work for each function linked, so most linkers are lazy. They only actually finish that work when the function is called. With more than a thousand externally visible symbols in the C library and nearly three thousand more local ones, this idea could save a noticeable amount of time.

The magic trick here that makes it work is a chunk of data called a *Procedure Linkage Table* (PLT), a table in the program that lists every function that a program calls. When the program is started, the PLT contains code for each function to query the runtime linker for the address at which it has loaded a function. It then fills in that entry in the table and jumps there. As each function is called, its entry in the PLT is simplified into a direct jump to the loaded function.

However, it's important to notice that this still leaves an extra layer of indirection -- each function call is resolved through a jump into a table.

Compatibility's not just for relationships

This means that the library you end up being linked to had better be compatible with the code that's calling it. With a statically linked executable, there is some guarantee that nothing will change on you. With dynamic linking, you don't have that guarantee.

What happens if a new version of the library comes out? Especially, what happens if the new version changes the calling sequence for a given function?

Version numbers to the rescue -- a shared library will have a version. When a program is linked against a library, it has the version number it's designed for stored in it. The dynamic linker can check for a matching version number. If the library has changed, the version number won't match, and the program won't be linked to the newer version of library.

One of the potential advantages of dynamic linking, however, is in fixing bugs. It'd be nice if you could fix a bug in the library and not have to recompile a thousand programs to take advantage of that fix. So sometimes, you want to link to a newer version.

Unfortunately, that creates some cases where you want to link to the newer version and some cases where you'd rather stick with an older version. There is a solution, though -- two kinds of version numbers:

- A major number indicates a potential incompatibility between library versions.
- A minor number indicates only bug fixes.

So under most circumstances, it is safe to load a library with the same major number and a higher minor number; consider it an unsafe practice to load a library with a higher major number.

To prevent users (and programmers) from needing to track library numbers and updates, the system comes with a large number of symbolic links. In general, the pattern is that

`libexample.so`

will be a link to

`libexample.so.N`

in which *N* is the highest *major* version number found on the system.

For every major version number supported,

`libexample.so.N`

will be a link in turn to

`libexample.so.N.M`

in which *M* is the largest *minor* version number.

Thus, if you specify `-lexample` to the linker, it looks for `libexample.so` which is a symbolic link to a symbolic link to the most recent version. On the other hand, when an existing program is loaded, it will try to load `libexample.so.N` in which *N* is the version to which it was originally linked. Everyone wins!

To debug, first you must know how to compile

To debug problems with shared libraries, it's useful to know a little more about how they're compiled.

In a traditional static library, the code generated is usually bound together into a library file with a name ending in `.a` and then it's passed to the linker. In a dynamic library, the library file's name generally ends in `.so`. The file structures are somewhat different.

A normal static library is in a format created by the `ar` utility, which is basically a very simple-minded archive program, similar to `tar` but simpler. In contrast, shared libraries are generally stored in more complicated file formats.

On modern Linux systems, this generally means the ELF binary format (Executable and Linkable Format). In ELF, each file is made up of one ELF header followed by zero or some segments and zero or some sections. The *segments* contain information necessary for runtime execution of the file, while *sections* contain important data for linking and relocation. Each byte in the entire file is taken by no more than one section at a time, but there can be orphan bytes that are not covered by a section. Normally in a UNIX executable, one or more sections are enclosed in one segment.

The ELF format has specifications for applications and libraries. The library format is a lot more complicated than just a simple archive of object modules, though.

The linker sorts through references to symbols, making notes about in which libraries they were found. Symbols from static libraries are added to the final executable; symbols from shared libraries are put into the PLT, and references to the PLT are created. Once those tasks are done, the resulting executable has a list of symbols it plans to look up from libraries it will load at runtime.

At runtime, the application loads the dynamic linker. In fact, the dynamic linker itself uses the same kind of versioning as the shared libraries. On SUSE Linux 9.1, for instance, the file `/lib/ld-linux.so.2` is a symbolic link to `/lib/ld-linux.so.2.3.3`. On the other hand, a program looking for `/lib/ld-linux.so.1` won't try to use the new version.

The dynamic linker then gets to do all the fun work. It looks to see which libraries (and which versions) a program was originally linked to and then loads them. Loading a library consists of:

- Finding it (and it may be in any of several directories on a system)
- Mapping it into the program's address space
- Allocating blocks of zero-filled memory the library may need
- Attaching the library's symbol table

Debugging this process can be difficult. There are a few kinds of problems you can encounter. For example, if the dynamic linker can't find a given library, it will abort loading the program. If it finds all the libraries it wants but can't find a symbol, it can abort for that too (but it may not act until the actual attempt to reference that symbol occurs) -- this is rare case though because normally, if the symbol isn't there, it will be noticed during the initial link.

Modifying the dynamic linker search path

When linking a program, you can specify additional paths to search at runtime. In `gcc` the syntax is `-Wl,-R/path`. If the program is already linked, you can also change this behavior by setting the environment variable `LD_LIBRARY_PATH`. Usually this is needed only if your application wants to search paths that aren't part of the system-wide default, a rare case for most Linux systems. In theory, the Mozilla people could have distributed a binary compiled with that path set, but they preferred to distribute a wrapper script that sets the library path appropriately before launching the executable.

Setting the library path can provide a workaround in the rare case where two applications require incompatible versions of a library. A wrapper script can be used to have one application search in a directory using the special version of the library it requires. Hardly an elegant solution, but in some cases it's the best you can do.

If you have a compelling reason to add a path to many programs, you can also change the system's default search path. The dynamic linker is controlled through `/etc/ld.so.conf`, which contains a list of directories to search by default. Any paths specified in `LD_LIBRARY_PATH` will be searched before the paths listed in `ld.so.conf`, so users can override these settings.

Most users have no reason to change the system default library search paths; generally the environment variable is a better match for likely reasons to change the search path, such as linking with libraries in a toolkit or testing programs against a newer version of a library.

Using `ldd`

One useful tool for debugging shared library problems is `ldd`. The name derives from *list dynamic dependencies*. This program looks at a given executable or shared library and figures out what shared libraries it needs to load and which versions would be used. The output looks like this:

Listing 1. Dependencies of /bin/sh

```
$ ldd /bin/sh
        linux-gate.so.1 =>  (0xffffe000)
        libreadline.so.4 => /lib/libreadline.so.4 (0x40036000)
        libhistory.so.4 => /lib/libhistory.so.4 (0x40062000)
        libncurses.so.5 => /lib/libncurses.so.5 (0x40069000)
        libdl.so.2 => /lib/libdl.so.2 (0x400af000)
        libc.so.6 => /lib/tls/libc.so.6 (0x400b2000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

It can be a little surprising to find out how many libraries a "simple" program uses. It's probably the case that `libhistory` is the one calling for `libncurses`. To find out, we can just run another `ldd` command:

Listing 2. Dependencies of libhistory

```
$ ldd /lib/libhistory.so.4
        linux-gate.so.1 =>  (0xffffe000)
        libncurses.so.5 => /lib/libncurses.so.5 (0x40026000)
        libc.so.6 => /lib/tls/libc.so.6 (0x4006b000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x80000000)
```

In some cases, an application may need extra library paths specified. For instance, the first few lines of an attempt to run `ldd` on the Mozilla binary came out like this:

Listing 3. Result of ldd for items not in search path

```
$ ldd /opt/mozilla/lib/mozilla-bin
  linux-gate.so.1 =>  (0xffffe000)
  libmozjs.so => not found
  libplds4.so => not found
  libplc4.so => not found
  libnspr4.so => not found
  libpthread.so.0 => /lib/tls/libpthread.so.0 (0x40037000)
```

Why aren't these libraries found? Because they're not in the usual search path for libraries. In fact, they're found in `/opt/mozilla/lib`, so one solution would be to add that directory to `LD_LIBRARY_PATH`.

Another option is to set the path to `.` and run `ldd` from that directory, although this is a little more dangerous -- putting the current directory in your library path is just as potentially treacherous as putting it in your executable path.

In this case, it's pretty clear that adding the directory these are in to the system-wide search path would be a bad idea. Nothing but Mozilla needs these libraries.

Linking Mozilla

And speaking of Mozilla, in case you were thinking that you'd never see more than a few lines of libraries, here's a somewhat more typical large application. Now you can see why Mozilla takes so long to launch!

Listing 4. Dependencies of mozilla-bin

```
linux-gate.so.1 =>  (0xffffe000)
libmozjs.so => ./libmozjs.so (0x40018000)
libplds4.so => ./libplds4.so (0x40099000)
libplc4.so => ./libplc4.so (0x4009d000)
libnspr4.so => ./libnspr4.so (0x400a2000)
libpthread.so.0 => /lib/tls/libpthread.so.0 (0x400f5000)
libdl.so.2 => /lib/libdl.so.2 (0x40105000)
libgtk-x11-2.0.so.0 => /opt/gnome/lib/libgtk-x11-2.0.so.0 (0x40108000)
libgdk-x11-2.0.so.0 => /opt/gnome/lib/libgdk-x11-2.0.so.0 (0x40358000)
libatk-1.0.so.0 => /opt/gnome/lib/libatk-1.0.so.0 (0x403c5000)
libgdk_pixbuf-2.0.so.0 => /opt/gnome/lib/libgdk_pixbuf-2.0.so.0 (0x403df000)
libpangoxft-1.0.so.0 => /opt/gnome/lib/libpangoxft-1.0.so.0 (0x403f1000)
libpangox-1.0.so.0 => /opt/gnome/lib/libpangox-1.0.so.0 (0x40412000)
libpango-1.0.so.0 => /opt/gnome/lib/libpango-1.0.so.0 (0x4041f000)
libgobject-2.0.so.0 => /opt/gnome/lib/libgobject-2.0.so.0 (0x40451000)
libgmodule-2.0.so.0 => /opt/gnome/lib/libgmodule-2.0.so.0 (0x40487000)
libglib-2.0.so.0 => /opt/gnome/lib/libglib-2.0.so.0 (0x4048b000)
libm.so.6 => /lib/tls/libm.so.6 (0x404f7000)
libstdc++.so.5 => /usr/lib/libstdc++.so.5 (0x40519000)
libgcc_s.so.1 => /lib/libgcc_s.so.1 (0x405d5000)
libc.so.6 => /lib/tls/libc.so.6 (0x405dd000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
libX11.so.6 => /usr/X11R6/lib/libX11.so.6 (0x406f3000)
libXrandr.so.2 => /usr/X11R6/lib/libXrandr.so.2 (0x407ef000)
libXi.so.6 => /usr/X11R6/lib/libXi.so.6 (0x407f3000)
libXext.so.6 => /usr/X11R6/lib/libXext.so.6 (0x407fb000)
libXft.so.2 => /usr/X11R6/lib/libXft.so.2 (0x4080a000)
libXrender.so.1 => /usr/X11R6/lib/libXrender.so.1 (0x4081e000)
libfontconfig.so.1 => /usr/lib/libfontconfig.so.1 (0x40826000)
libfreetype.so.6 => /usr/lib/libfreetype.so.6 (0x40850000)
libexpat.so.0 => /usr/lib/libexpat.so.0 (0x408b9000)
```

Learning more about shared libraries

Users interested in learning more about dynamic linking on Linux have a broad field of options. The GNU compiler and linker tool chain documentation is excellent, although the guts of it are stored in the `info` format and not mentioned in the standard man pages.

The manual page for `ld.so` contains a fairly comprehensive list of variables that modify the behavior of the dynamic linker, as well as explanations of the different versions of the dynamic linker that have been used in the past.

Most Linux documentation assumes that all shared libraries are dynamically linked because on Linux systems, they generally are. The work needed to make statically linked shared libraries is substantial and most users don't gain any benefit from it, although the performance difference is noticeable on systems that support the feature.

If you're using a pre-packaged system off the shelf, you probably won't run into very many shared library versions -- the system probably just ships with the ones it was linked against. On the other hand, if you do a lot of updates and source builds, you can end up with many versions of a shared library since old versions get left around "just in case."

As always, if you want to know more, experiment. Remember that nearly everything on a system refers back to those same few shared libraries, so if you break one of the system's core shared libraries, you're going to get to play with some kind of system recovery tool.

Related topics

- *Linkers and Loaders* by John Levine (Morgan Kauffman, October 1999) is an authoritative source devoted to compile-time and run-time processes. (Some manuscript chapters are available online.)
- Read this communique if you ever wondered why a versioning scheme for shared libraries is important.
- Writing DLLs for Linux apps (developerWorks, October 2001) demonstrates how dynamically linked libraries are often a great way to add functionality without writing a whole new Linux application.
- Shared objects for the object disoriented! (developerWorks, April 2001) explains how to write dynamically loadable libraries and suggests tools to use in the process.