Shared Libraries

Archived: 2026-04-06 03:32:36 UTC

Shared libraries are libraries that are loaded by programs when they start. When a shared library is installed properly, all programs that start afterwards automatically use the new shared library. It's actually much more flexible and sophisticated than this, because the approach used by Linux permits you to:

- update libraries and still support programs that want to use older, non-backward-compatible versions of those libraries;
- override specific libraries or even specific functions in a library when executing a particular program.
- do all this while programs are running using existing libraries.

3.1. Conventions

For shared libraries to support all of these desired properties, a number of conventions and guidelines must be followed. You need to understand the difference between a library's names, in particular its ``soname'' and ``real name'' (and how they interact). You also need to understand where they should be placed in the filesystem.

3.1.1. Shared Library Names

Every shared library has a special name called the ``soname''. The soname has the prefix ``lib'', the name of the library, the phrase ``.so'', followed by a period and a version number that is incremented whenever the interface changes (as a special exception, the lowest-level C libraries don't start with ``lib''). A fully-qualified soname includes as a prefix the directory it's in; on a working system a fully-qualified soname is simply a symbolic link to the shared library's ``real name''.

Every shared library also has a ``real name'', which is the filename containing the actual library code. The real name adds to the soname a period, a minor number, another period, and the release number. The last period and release number are optional. The minor number and release number support configuration control by letting you know exactly what version(s) of the library are installed. Note that these numbers might not be the same as the numbers used to describe the library in documentation, although that does make things easier.

In addition, there's the name that the compiler uses when requesting a library, (I'll call it the ``linker name''), which is simply the soname without any version number.

The key to managing shared libraries is the separation of these names. Programs, when they internally list the shared libraries they need, should only list the soname they need. Conversely, when you create a shared library, you only create the library with a specific filename (with more detailed version information). When you install a new version of a library, you install it in one of a few special directories and then run the program ldconfig(8). ldconfig examines the existing files and creates the sonames as symbolic links to the real names, as well as setting up the cache file /etc/ld.so.cache (described in a moment).

ldconfig doesn't set up the linker names; typically this is done during library installation, and the linker name is simply created as a symbolic link to the ``latest'' soname or the latest real name. I would recommend having the linker name be a symbolic link to the soname, since in most cases if you update the library you'd like to automatically use it when linking. I asked H. J. Lu why ldconfig doesn't automatically set up the linker names. His explanation was basically that you might want to run code using the latest version of a library, but might instead want *development* to link against an old (possibly incompatible) library. Therefore, ldconfig makes no assumptions about what you want programs to link to, so installers must specifically modify symbolic links to update what the linker will use for a library.

Thus, `/usr/lib/libreadline.so.3` is a fully-qualified soname, which ldconfig would set to be a symbolic link to some realname like `/usr/lib/libreadline.so.3.0`. There should also be a linker name, `/usr/lib/libreadline.so` which could be a symbolic link referring to `/usr/lib/libreadline.so.3`.

3.1.2. Filesystem Placement

Shared libraries must be placed somewhere in the filesystem. Most open source software tends to follow the GNU standards; for more information see the info file documentation at info:standards#Directory_Variables. The GNU standards recommend installing by default all libraries in /usr/local/lib when distributing source code (and all commands should go into /usr/local/bin). They also define the convention for overriding these defaults and for invoking the installation routines.

The Filesystem Hierarchy Standard (FHS) discusses what should go where in a distribution (see http://www.pathname.com/fhs). According to the FHS, most libraries should be installed in /usr/lib, but libraries required for startup should be in /lib and libraries that are not part of the system should be in /usr/local/lib.

There isn't really a conflict between these two documents; the GNU standards recommend the default for developers of source code, while the FHS recommends the default for distributors (who selectively override the source code defaults, usually via the system's package management system). In practice this works nicely: the ``latest'' (possibly buggy!) source code that you download automatically installs itself in the ``local'' directory (`/usr/local`), and once that code has matured the package managers can trivially override the default to place the code in the standard place for distributions. Note that if your library calls programs that can only be called via libraries, you should place those programs in /usr/local/libexec (which becomes /usr/libexec in a distribution). One complication is that Red Hat-derived systems don't include /usr/local/lib by default in their search for libraries; see the discussion below about /etc/ld.so.conf. Other standard library locations include /usr/X11R6/lib for X-windows. Note that /lib/security is used for PAM modules, but those are usually loaded as DL libraries (also discussed below).

3.2. How Libraries are Used

On GNU glibc-based systems, including all Linux systems, starting up an ELF binary executable automatically causes the program loader to be loaded and run. On Linux systems, this loader is named /lib/ld-linux.so.X (where X is a version number). This loader, in turn, finds and loads all other shared libraries used by the program.

The list of directories to be searched is stored in the file /etc/ld.so.conf. Many Red Hat-derived distributions don't normally include /usr/local/lib in the file /etc/ld.so.conf. I consider this a bug, and adding /usr/local/lib to

/etc/ld.so.conf is a common ``fix" required to run many programs on Red Hat-derived systems.

If you want to just override a few functions in a library, but keep the rest of the library, you can enter the names of overriding libraries (.o files) in /etc/ld.so.preload; these ``preloading" libraries will take precedence over the standard set. This preloading file is typically used for emergency patches; a distribution usually won't include such a file when delivered.

Searching all of these directories at program start-up would be grossly inefficient, so a caching arrangement is actually used. The program ldconfig(8) by default reads in the file /etc/ld.so.conf, sets up the appropriate symbolic links in the dynamic link directories (so they'll follow the standard conventions), and then writes a cache to /etc/ld.so.cache that's then used by other programs. This greatly speeds up access to libraries. The implication is that ldconfig must be run whenever a DLL is added, when a DLL is removed, or when the set of DLL directories changes; running ldconfig is often one of the steps performed by package managers when installing a library. On start-up, then, the dynamic loader actually uses the file /etc/ld.so.cache and then loads the libraries it needs.

By the way, FreeBSD uses slightly different filenames for this cache. In FreeBSD, the ELF cache is /var/run/ld-elf.so.hints and the a.out cache is /var/run/ld.so.hints. These are still updated by ldconfig(8), so this difference in location should only matter in a few exotic situations.

3.3. Environment Variables

Various environment variables can control this process, and there are environment variables that permit you to override this process.

3.3.1. LD_LIBRARY_PATH

You can temporarily substitute a different library for this particular execution. In Linux, the environment variable LD_LIBRARY_PATH is a colon-separated set of directories where libraries should be searched for first, before the standard set of directories; this is useful when debugging a new library or using a nonstandard library for special purposes. The environment variable LD_PRELOAD lists shared libraries with functions that override the standard set, just as /etc/ld.so.preload does. These are implemented by the loader /lib/ld-linux.so. I should note that, while LD_LIBRARY_PATH works on many Unix-like systems, it doesn't work on all; for example, this functionality is available on HP-UX but as the environment variable SHLIB_PATH, and on AIX this functionality is through the variable LIBPATH (with the same syntax, a colon-separated list).

LD_LIBRARY_PATH is handy for development and testing, but shouldn't be modified by an installation process for normal use by normal users; see ``Why LD_LIBRARY_PATH is Bad" at http://www.visi.com/~barr/ldpath.html for an explanation of why. But it's still useful for development or testing, and for working around problems that can't be worked around otherwise. If you don't want to set the LD_LIBRARY_PATH environment variable, on Linux you can even invoke the program loader directly and pass it arguments. For example, the following will use the given PATH instead of the content of the environment variable LD_LIBRARY_PATH, and run the given executable:

```
/lib/ld-linux.so.2 --library-path PATH EXECUTABLE
```

Just executing ld-linux.so without arguments will give you more help on using this, but again, don't use this for normal use - these are all intended for debugging.

3.3.2. LD_DEBUG

Another useful environment variable in the GNU C loader is LD_DEBUG. This triggers the dl* functions so that they give quite verbose information on what they are doing. For example:

```
export LD_DEBUG=files
command_to_run
```

displays the processing of files and libraries when handling libraries, telling you what dependencies are detected and which SOs are loaded in what order. Setting LD_DEBUG to ``bindings'' displays information about symbol binding, setting it to ``libs'' displays the library search paths, and setting ti to ``versions'' displays the version depdendencies.

Setting LD_DEBUG to ``help'' and then trying to run a program will list the possible options. Again, LD_DEBUG isn't intended for normal use, but it can be handy when debugging and testing.

3.3.3. Other Environment Variables

There are actually a number of other environment variables that control the loading process; their names begin with LD_ or RTLD_. Most of the others are for low-level debugging of the loader process or for implementing specialized capabilities. Most of them aren't well-documented; if you need to know about them, the best way to learn about them is to read the source code of the loader (part of gcc).

Permitting user control over dynamically linked libraries would be disastrous for setuid/setgid programs if special measures weren't taken. Therefore, in the GNU loader (which loads the rest of the program on program start-up), if the program is setuid or setgid these variables (and other similar variables) are ignored or greatly limited in what they can do. The loader determines if a program is setuid or setgid by checking the program's credentials; if the uid and euid differ, or the gid and the egid differ, the loader presumes the program is setuid/setgid (or descended from one) and therefore greatly limits its abilities to control linking. If you read the GNU glibc library source code, you can see this; see especially the files elf/rtld.c and sysdeps/generic/dl-sysdep.c. This means that if you cause the uid and gid to equal the euid and egid, and then call a program, these variables will have full effect. Other Unix-like systems handle the situation differently but for the same reason: a setuid/setgid program should not be unduly affected by the environment variables set.

3.4. Creating a Shared Library

Creating a shared library is easy. First, create the object files that will go into the shared library using the gcc -fPIC or -fpic flag. The -fPIC and -fpic options enable ``position independent code'' generation, a requirement for shared libraries; see below for the differences. You pass the soname using the -Wl gcc option. The -Wl option passes options along to the linker (in this case the -soname linker option) - the commas after -Wl are not a typo, and you must not include unescaped whitespace in the option. Then create the shared library using this format:

```
gcc -shared -Wl,-soname,your_soname \
    -o library_name file_list library_list
```

Here's an example, which creates two object files (a.o and b.o) and then creates a shared library that contains both of them. Note that this compilation includes debugging information (-g) and will generate warnings (-Wall), which aren't required for shared libraries but are recommended. The compilation generates object files (using -c), and includes the required -fPIC option:

```
gcc -fPIC -g -c -Wall a.c
gcc -fPIC -g -c -Wall b.c
gcc -shared -Wl,-soname,libmystuff.so.1 \
    -o libmystuff.so.1.0.1 a.o b.o -lc
```

Here are a few points worth noting:

- Don't strip the resulting library, and don't use the compiler option -fomit-frame-pointer unless you really have to. The resulting library will work, but these actions make debuggers mostly useless.
- Use -fPIC or -fpic to generate code. Whether to use -fPIC or -fpic to generate code is target-dependent. The -fPIC choice always works, but may produce larger code than -fpic (mnenomic to remember this is that PIC is in a larger case, so it may produce larger amounts of code). Using -fpic option usually generates smaller and faster code, but will have platform-dependent limitations, such as the number of globally visible symbols or the size of the code. The linker will tell you whether it fits when you create the shared library. When in doubt, I choose -fPIC, because it always works.
- In some cases, the call to gcc to create the object file will also need to include the option ``-Wl,-export-dynamic". Normally, the dynamic symbol table contains only symbols which are used by a dynamic object. This option (when creating an ELF file) adds all symbols to the dynamic symbol table (see ld(1) for more information). You need to use this option when there are 'reverse dependencies', i.e., a DL library has unresolved symbols that by convention must be defined in the programs that intend to load these libraries. For ``reverse dependencies" to work, the master program must make its symbols dynamically available. Note that you could say ``-rdynamic" instead of ``-Wl,export-dynamic" if you only work with Linux systems, but according to the ELF documentation the ``-rdynamic" flag doesn't always work for gcc on non-Linux systems.

During development, there's the potential problem of modifying a library that's also used by many other programs -- and you don't want the other programs to use the ``developmental"library, only a particular application that you're testing against it. One link option you might use is ld's ``rpath" option, which specifies the runtime library search path of that particular program being compiled. From gcc, you can invoke the rpath option by specifying it this way:

```
-Wl,-rpath,$(DEFAULT_LIB_INSTALL_PATH)
```

If you use this option when building the library client program, you don't need to bother with LD_LIBRARY_PATH (described next) other than to ensure it's not conflicting, or using other techniques to hide the library.

3.5. Installing and Using a Shared Library

Once you've created a shared library, you'll want to install it. The simple approach is simply to copy the library into one of the standard directories (e.g., /usr/lib) and run ldconfig(8).

First, you'll need to create the shared libraries somewhere. Then, you'll need to set up the necessary symbolic links, in particular a link from a soname to the real name (as well as from a versionless soname, that is, a soname that ends in ``.so'' for users who don't specify a version at all). The simplest approach is to run:

```
ldconfig -n directory_with_shared_libraries
```

Finally, when you compile your programs, you'll need to tell the linker about any static and shared libraries that you're using. Use the -l and -L options for this.

If you can't or don't want to install a library in a standard place (e.g., you don't have the right to modify /usr/lib), then you'll need to change your approach. In that case, you'll need to install it somewhere, and then give your program enough information so the program can find the library... and there are several ways to do that. You can use gcc's -L flag in simple cases. You can use the ``rpath'' approach (described above), particularly if you only have a specific program to use the library being placed in a ``non-standard'' place. You can also use environment variables to control things. In particular, you can set LD_LIBRARY_PATH, which is a colon-separated list of directories in which to search for shared libraries before the usual places. If you're using bash, you could invoke my_program this way using:

```
LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH  my_program
```

If you want to override just a few selected functions, you can do this by creating an overriding object file and setting LD_PRELOAD; the functions in this object file will override just those functions (leaving others as they were).

Usually you can update libraries without concern; if there was an API change, the library creator is supposed to change the soname. That way, multiple libraries can be on a single system, and the right one is selected for each program. However, if a program breaks on an update to a library that kept the same soname, you can force it to use the older library version by copying the old library back somewhere, renaming the program (say to the old name plus ``.orig''), and then create a small ``wrapper'' script that resets the library to use and calls the real (renamed) program. You could place the old library in its own special area, if you like, though the numbering conventions do permit multiple versions to live in the same directory. The wrapper script could look something like this:

```
#!/bin/sh
export LD_LIBRARY_PATH=/usr/local/my_lib:$LD_LIBRARY_PATH
exec /usr/bin/my_program.orig $*
```

Please don't depend on this when you write your own programs; try to make sure that your libraries are either backwards-compatible or that you've incremented the version number in the soname every time you make an incompatible change. This is just an ``emergency'' approach to deal with worst-case problems.

You can see the list of the shared libraries used by a program using ldd(1). So, for example, you can see the shared libraries used by ls by typing:

```
ldd /bin/ls
```

Generally you'll see a list of the sonames being depended on, along with the directory that those names resolve to. In practically all cases you'll have at least two dependencies:

- /lib/ld-linux.so.N (where N is 1 or more, usually at least 2). This is the library that loads all other libraries.
- libc.so.N (where N is 6 or more). This is the C library. Even other languages tend to use the C library (at least to implement their own libraries), so most programs at least include this one.

Beware: do *not* run ldd on a program you don't trust. As is clearly stated in the ldd(1) manual, ldd works by (in certain cases) by setting a special environment variable (for ELF objects, LD_TRACE_LOADED_OBJECTS) and then executing the program. It may be possible for an untrusted program to force the ldd user to run arbitrary code (instead of simply showing the ldd information). So, for safety's sake, don't use ldd on programs you don't trust to execute.

3.6. Incompatible Libraries

When a new version of a library is binary-incompatible with the old one the soname needs to change. In C, there are four basic reasons that a library would cease to be binary compatible:

1. The behavior of a function changes so that it no longer meets its original specification,
2. Exported data items change (exception: adding optional items to the ends of structures is okay, as long as those structures are only allocated within the library).
3. An exported function is removed.
4. The interface of an exported function changes.

If you can avoid these reasons, you can keep your libraries binary-compatible. Said another way, you can keep your Application Binary Interface (ABI) compatible if you avoid such changes. For example, you might want to add new functions but not delete the old ones. You can add items to structures but only if you can make sure that old programs won't be sensitive to such changes by adding items only to the end of the structure, only allowing the

library (and not the application) to allocate the structure, making the extra items optional (or having the library fill them in), and so on. Watch out - you probably can't expand structures if users are using them in arrays.

For C++ (and other languages supporting compiled-in templates and/or compiled dispatched methods), the situation is trickier. All of the above issues apply, plus many more issues. The reason is that some information is implemented ``under the covers'' in the compiled code, resulting in dependencies that may not be obvious if you don't know how C++ is typically implemented. Strictly speaking, they aren't ``new'' issues, it's just that compiled C++ code invokes them in ways that may be surprising to you. The following is a (probably incomplete) list of things that you cannot do in C++ and retain binary compatibility, as reported by Troll Tech's Technical FAQ:

1. add reimplementations of virtual functions (unless it it safe for older binaries to call the original implementation), because the compiler evaluates SuperClass::virtualFunction() calls at compile-time (not link-time).
2. add or remove virtual member functions, because this would change the size and layout of the vtbl of every subclass.
3. change the type of any data members or move any data members that can be accessed via inline member functions.
4. change the class hierarchy, except to add new leaves.
5. add or remove private data members, because this would change the size and layout of every subclass.
6. remove public or protected member functions unless they are inline.
7. make a public or protected member function inline.
8. change what an inline function does, unless the old version continues working.
9. change the access rights (i.e. public, protected or private) of a member function in a portable program, because some compilers mangle the access rights into the function name.

Given this lengthy list, developers of C++ libraries in particular must plan for more than occasional updates that break binary compatibility. Fortunately, on Unix-like systems (including Linux) you can have multiple versions of a library loaded at the same time, so while there is some disk space loss, users can still run ``old'' programs needing old libraries.
