Shared libraries with GCC on Linux

By anduril462

Libraries are an indispensable tool for any programmer. They are pre-existing code that is compiled and ready for you to use. They often provide generic functionality, like linked lists or binary trees that can hold any data, or specific functionality like an interface to a database server such as MySQL.

Most larger software projects will contain several components, some of which you may find use for later on in some other project, or that you just want to separate out for organizational purposes. When you have a reusable or logically distinct set of functions, it is helpful to build a library from it so that you do not have to copy the source code into your current project and recompile it all the time - and so you can keep different modules of your program disjoint and change one without affecting others. Once it is been written and tested, you can safely reuse it over and over again, saving the time and hassle of building it into your project every time.

Building static libraries is fairly simple, and since we rarely get questions on them, I will not cover them. I will stick with shared libraries, which seem to be more confusing for most people.

Before we get started, it might help to get a quick rundown of everything that happens from source code to running program:

1. C Preprocessor: This stage processes all the preprocessor directives. Basically, any line that starts with a #, such as #define and #include.
2. Compilation Proper: Once the source file has been preprocessed, the result is then compiled. Since many people refer to the entire build process as compilation, this stage is often referred to as compilation proper. This stage turns a .c file into an .o (object) file.
3. Linking: Here is where all of the object files and any libraries are linked together to make your final program. Note that for static libraries, the actual library is placed in your final program, while for shared libraries, only a reference to the library is placed inside. Now you have a complete program that is ready to run. You launch it from the shell, and the program is handed off to the loader.
4. Loading: This stage happens when your program starts up. Your program is scanned for references to shared libraries. Any references found are resolved and the libraries are mapped into your program.

Steps 3 and 4 are where the magic (and confusion) happens with shared libraries.

Now, on to our (very simple) example.

foo.h:

```
#ifndef foo_h__
#define foo_h__

extern void foo(void);

#endif  // foo_h__
```

##foo.c:

```
#include <stdio.h>


void foo(void)
{
    puts("Hello, I am a shared library");
}
```

##main.c:

```
#include <stdio.h>
#include "foo.h"

int main(void)
{
    puts("This is a shared library test...");
    foo();
    return 0;
}
```

foo.h defines the interface to our library, a single function, foo(). foo.c contains the implementation of that function, and main.c is a driver program that uses our library.

For the purposes of this example, everything will happen in /home/username/foo

Step 1: Compiling with Position Independent Code

We need to compile our library source code into position-independent code (PIC):[1]

```
$ gcc -c -Wall -Werror -fpic foo.c
```

Step 2: Creating a shared library from an object file

Now we need to actually turn this object file into a shared library. We will call it libfoo.so:

```
gcc -shared -o libfoo.so foo.o
```

Step 3: Linking with a shared library

As you can see, that was actually pretty easy. We have a shared library. Let us compile our main.c and link it with libfoo. We will call our final program test.

Telling GCC where to find the shared library

Uh-oh! The linker does not know where to find libfoo. GCC has a list of places it looks by default, but our directory is not in that list.[2] We need to tell GCC where to find libfoo.so. We will do that with the -L option. In this example, we will use the current directory, /home/username/foo:

```
$ gcc -L/home/username/foo -Wall -o test main.c -lfoo
```

Step 4: Making the library available at runtime

Good, no errors. Now let us run our program:

```
$ ./test
./test: error while loading shared libraries: libfoo.so: cannot open shared object file: No such file or directory
```

Oh no! The loader cannot find the shared library.[3] We did not install it in a standard location, so we need to give the loader a little help. We have a couple of options: we can use the environment variable LD_LIBRARY_PATH for this, or rpath. Let us take a look first at LD_LIBRARY_PATH:

Using LD_LIBRARY_PATH

```
$ echo $LD_LIBRARY_PATH
```

There is nothing in there. Let us fix that by prepending our working directory to the existing LD_LIBRARY_PATH:

```
$ LD_LIBRARY_PATH=/home/username/foo:$LD_LIBRARY_PATH
$ ./test
./test: error while loading shared libraries: libfoo.so: cannot open shared object file: No such file or directory
```

What happened? Our directory is in LD_LIBRARY_PATH, but we did not export it. In Linux, if you do not export the changes to an environment variable, they will not be inherited by the child processes. The loader and our test program did not inherit the changes we made. Thankfully, the fix is easy:

```
$ export LD_LIBRARY_PATH=/home/username/foo:$LD_LIBRARY_PATH
$ ./test
This is a shared library test...
Hello, I am a shared library
```

Good, it worked! LD_LIBRARY_PATH is great for quick tests and for systems on which you do not have admin privileges. As a downside, however, exporting the LD_LIBRARY_PATH variable means it may cause problems with other programs you run that also rely on LD_LIBRARY_PATH if you do not reset it to its previous state when you are done.

Using rpath

Now let s try rpath (first we will clear LD_LIBRARY_PATH to ensure it is rpath that is finding our library). Rpath, or the run path, is a way of embedding the location of shared libraries in the executable itself, instead of relying on default locations or environment variables. We do this during the linking stage. Notice the lengthy â€œ-Wl,-rpath=/home/username/fooâ€□ option. The -Wl portion sends comma-separated options to the linker, so we tell it to send the -rpath option to the linker with our working directory.

```
$ unset LD_LIBRARY_PATH
$ gcc -L/home/username/foo -Wl,-rpath=/home/username/foo -Wall -o test main.c -lfoo
$ ./test
This is a shared library test...
Hello, I am a shared library
```

Excellent, it worked. The rpath method is great because each program gets to list its shared library locations independently, so there are no issues with different programs looking in the wrong paths like there were for LD_LIBRARY_PATH.

rpath vs. LD_LIBRARY_PATH

There are a few downsides to rpath, however. First, it requires that shared libraries be installed in a fixed location so that all users of your program will have access to those libraries in those locations. That means less flexibility in system configuration. Second, if that library refers to a NFS mount or other network drive, you may experience undesirable delays - or worse - on program startup.

Using ldconfig to modify ld.so

What if we want to install our library so everybody on the system can use it? For that, you will need admin privileges. You will need this for two reasons: first, to put the library in a standard location, probably /usr/lib or /usr/local/lib, which normal users do not have write access to. Second, you will need to modify the ld.so config file and cache. As root, do the following:

```
$ cp /home/username/foo/libfoo.so /usr/lib
$ chmod 0755 /usr/lib/libfoo.so
```

Now the file is in a standard location, with correct permissions, readable by everybody. We need to tell the loader it is available for use, so let us update the cache:

```
$ ldconfig
```

Now our library is installed. Before we test it, we have to clean up a few things:

Clear our LD_LIBRARY_PATH once more, just in case:

```
$ unset LD_LIBRARY_PATH
```

Re-link our executable. Notice we do not need the -L option since our library is stored in a default location and we are not using the rpath option:

```
$ gcc -Wall -o test main.c -lfoo
```

Let us make sure we are using the /usr/lib instance of our library using ldd:

```
$ ldd test | grep foo
libfoo.so => /usr/lib/libfoo.so (0x00a42000)
```

Good, now let us run it:

```
$ ./test
This is a shared library test...
Hello, I am a shared library
```

That about wraps it up. We have covered how to build a shared library, how to link with it, and how to resolve the most common loader issues with shared libraries - as well as the positives and negatives of different approaches.

1. It looks in the DT_RPATH section of the executable, unless there is a DT_RUNPATH section.
2. It looks in LD_LIBRARY_PATH. This is skipped if the executable is setuid/setgid for security reasons.
3. It looks in the DT_RUNPATH section of the executable unless the setuid/setgid bits are set (for security reasons).
4. It looks in the cache file /etc/ld/so/cache (disabled with the -z nodeflib linker option).
5. It looks in the default directories /lib then /usr/lib (disabled with the -z nodeflib linker option).

1. What is position independent code? PIC is code that works no matter where in memory it is placed. Because several different programs can all use one instance of your shared library, the library cannot store things at fixed addresses, since the location of that library in memory will vary from program to program. ↩
2. GCC first searches for libraries in /usr/local/lib, then in /usr/lib. Following that, it searches for libraries in the directories specified by the -L parameter, in the order specified on the command line. ↩
3. The default GNU loader, ld.so, looks for libraries in the following order: ↩