Shared objects for the object disoriented!

How to write dynamically loadable libraries

Ashish Bansal April 01, 2001

Ashish Bansal tells you how to write dynamically loadable libraries and suggests tools you want to use in the process. He reviews the compilation process and naming conventions, and then walks you through writing, compiling, and installing a shared library.

Relax, a shared object has nothing to do with object-oriented technology! What we're talking about are dynamically linked libraries on the Linux platform (analogous to DLLs on Windows). At different times in our coding lives, all of us have used some sort of library, be it for a simple function like `printf()` in C or for a complex function like `sort()` in the C++ generic function library. Libraries make everyday programming easier and let the developers focus on the task at hand. Imagine having to write printf() and file I/O functions for every piece of code you write!

Almost any software built relies heavily on the libraries that allow the software to do a variety of tasks -- from printing on screen to logging onto a network. Some of these libraries are provided by the system and some are written by third-party vendors or the users themselves. During the process of compilation these libraries are linked into the application. If an application were to use a lot of libraries, and all code were linked in, the size of the application would become prohibitive. In comes the "shared Library," which is not linked into the application source but loaded dynamically at the moment it's required by the application.

The compilation process

Before we dive in to building shared objects, let's understand the compilation process and what a shared object is. As an illustration, take the famous and by now historical example of the hello world code.

Listing 1: Hello.c; Shared objects defined

```
#include "stdio.h"
void main()
{
 printf("Hello World!");
}
```

To compile this program, the following command line could be used:

```
$ gcc -o hello hello.c
```

This creates an executable named *hello*. Now the compiler can go through the following steps:

1. Syntax checking: Check the syntax and grammar of the file.
2. Compilation: Compile the file to produce an object file for the code. Unresolved function names, like `printf()` in this case, are marked out in the object file produced. (See File formats below.)
3. Linking: Invoke a separate program called the linker. (In UNIX the program is `ld`.) The linker tries to resolve functions and variables by searching in the various libraries for the code. For example, the code for `printf()` resides in a file `libc.a` (or `libc.so`). If you need libraries other than the standard set, they must be specified.

The above command line to compile and link the code could be broken into the following two pieces:

```
>$ gcc -c hello.c
$ ld -lc -o hello hello.c
```

This is the compilation step (as specified by the `-c` option). The second step is the linking, which uses the `ld` program to produce the executable hello.

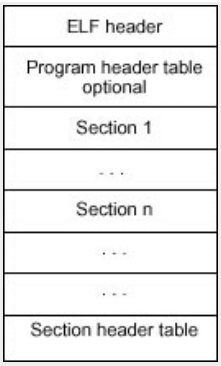
File formats

ELF stands for *Executable and Linking Format*. This format is used for object files on most UNIX platforms including Linux. There are basically three types of object files:

1. A relocatable file holds code and data suitable to be linked with other object fields to create an executable or shared object file, or another relocatable object.
2. An executable file holds a program that is ready to execute.
3. A shared object file holds code and data suitable to be linked to other shared objects or relocatable files. (The object file format is shown in the figure below.)

Every file has an ELF header that resides at the beginning of the file and acts like a roadmap to the rest of the file. Sections represent the smallest unit that can be processed in a file and hold the bulk of the information required for linking. A section header table contains information about the sections in the file. A program header table, if present, tells how to create a process image, which is used if the object file is an executable. The `exec` program uses the program header table to fork a process. Note that the position of the ELF header is constant in the file, although other parts may appear at different places than shown in the figure.

The ELF header is used by the `file` program, (see Utility programs and tools later in this article) to print out information about the file. (The *libelf* library package provides a programming interface to access information in the ELF headers.)



This type of linking is called *static linking*. Using static compilation, the code in the library combines with the application (which can have the downside of making the executable very large). Any worthwhile application uses hundreds of functions as libraries, many of which are supplied. Some of these libraries are standard, some third party, and others internal. Using static compilation, the size of the final executable becomes pretty large and all of it has to be loaded into the memory at runtime, so regardless of whether a function is being used or not, its code is in the memory.

It would be much better if there were a mechanism by which the libraries could be dynamically loaded into the memory as they are needed, which would reduce the memory footprint of the program and also break the application into smaller parts. It would also allow easy distribution, installation, and upgrading. It just so happens that such a mechanism does exist, namely our dynamically linked libraries (DLLs on Windows, and Shared Objects on Linux). Applications using them are called dynamic executables.

Naming conventions

Before we dive into shared objects, let's take a brief look at library naming conventions. Static libraries generally start with the letter *lib*, and have the extension *.a*. Shared objects have two different names: the *soname* and the *real name*. The *soname* consists of the prefix "lib", followed by the name of the library, a ".so" followed by another dot, and a number indicating the major version number. The soname may be fully qualified by prefixing path information. The real name is the actual file name containing the compiled code for the library. The real name adds a dot, a minor number, another dot, and the release number, to the soname. (The release number and the associated dot are optional.)

There is one other name defined by the *Program-Library How-To* we should look at, called the *linker name*, which may be used to refer to the soname without the version number information. Clients using this library refer to it using the linker name. Generally, this is a link to the soname. And the soname is a link to the real name.

As an example, take the soname `/usr/lib/libhello.so.1` This is a fully qualified soname and would link pointing to `/usr/lib/libhello.so.1.5`. The corresponding linker name would be `/usr/lib/libhello.so`. This does all seem like a lot of names to manage, but there are tools to help you manage them. (See `ldconfig` in Utility programs and tools later in this article.)

Now let's get to the meaty stuff and write a sample shared object. The soname of the library will be `libprint.so.1` and the real name will be `libprint.so.1.0` . Our library will have one function, `printstring(char*)`, which will print the word "String: " followed by whatever string was passed to it as an argument.

Writing a shared library

There are basically two files that have to be written for a usable library. The first is a header file, which declares all the functions exported by the library and will be included by the client in the code. The second is the definition of the functions to be compiled and placed as the shared object. For our example, the header file looks like this:

Listing 2: Libprint.h Code; Header file

```
/* file libprint.h - for example use! */
void printstring(char* str);
```

The code for the library is pretty basic and is shown in the next listing.

Listing 3: libprint.c Code

```
/* file libprint.c */
#include "stdio.h"
void printstring(char* str)
 printf("String: %s\n", str);
}
```

There are two special functions, `_init(void)` and `_fini(void)`, which are called automatically by the dynamic loader whenever a library is loaded. A default implementation is typically provided for these two functions, although you can bypass these and write your own. Let us add these two functions to our libprint.c code to print out diagnostic messages whenever they are called.

Listing 4: Code for _init() and _fini()

```
void _init()
{
 printf("Inside _init()\n");
}
void _fini()
{
 printf("Inside _fini()\n");
}
```

Now we combine this with the code in Listing 3. Pretty simple, huh? To write a custom library you just have to use the templates of `libprint.c` and `libprint.h` and then write the appropriate functions. Now let's go ahead and compile the library.

Compiling a shared library

The sequence of commands to compile the library are:

```
 $ gcc -fPIC -c libprint.c
 $ ld -shared -soname libprint.so.1 -o libprint.so.1.0 -lc libprint.o
```

Notice the `-fPIC` option in the gcc command line. This is essential to produce *Position-Independent Code*. Translated into English, this command means to "generate code that can be loaded any where in the process space of a process". It's also very important for a shared object. By using this option, the number of relocations that have to be performed are cut down to the very minimum. On loading a shared object that is used by an executable, some space has to be allocated for it. And the text and data sections have to be allocated some locations. If they are not built in a position-independent way, then a fair amount of relocations have to be done by the program loading the shared object, thus impacting performance adversely.

Now let's analyze the options passed to `ld`. The `-shared` option indicates that the output file is supposed to be a shared library. By specifying the `-soname name` option, we specify what the soname will be. The `-o name` specifies the real name of the shared object. It is important to specify the soname and the real name because these are used during installation of the library.

Installing and using shared libraries

Now that we've built our library, let's install it and make a small client program that uses it. A special program called `ldconfig` is used for installing shared libraries. Generally, shared libraries are installed in either `/usr/lib`, `lib` or `/usr/local/lib`. Once the library is made, it should be copied in one of these directories. Then we just run the `ldconfig` program.

ldconfig

```
 $ldconfig -v -n .
...:
 libprint.so.1 => ./libprint.so.1.0
```

Now we've created a symbolic link named `libprint.so.1` to `libprint.so.1.0`. The next step in installing is creating another link for the linker name, like this:

Link for the linker name

```
 $ ln -sf libprint.so.1 libprint.so
```

To copy stuff in the `/usr/lib`, `lib` or `/usr/local/lib` directories, super user permissions are required. When an application is run, these directories are automatically searched to resolve libraries. If super user permission is not there, the shared library can be installed in any directory, but another set has to performed before executables using this shared library can be run. An environment variable, `LD_LIBRARY_PATH`, has to be set to point to the path in which the shared library resides. As an example, if the shared library is in the same directory as the executable, you'd want to do this:

LD_LIBRARY_PATH

```
$ export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
```

Now we're all set to compile and execute the client that uses the shared library we've just built. Using shared libraries is pretty simple. :)

Listing 5: Client.c; a sample client that uses the function printstring() from the library

```
#include "libstring.h"
void main()
{
 printf("In Main!\n");
 printstring("In Main!");
}
```

This program can be compiled to an executable using the command line:

```
$ gcc -o client client.c -L. -lprint
```

This produces an executable called *client*, which assumes that the library resides in the same directory as the code (this is indicated by `-L. -lprint`). Upon execution, the following output should be produced:

```
  $ client
  Inside _init()
  In Main!
  String: In Main!
  Inside _fini()
  $
```

So now we have made a shared object, installed it, and used it! So let's take a look under the hood and see what happens when we try to execute *client*. On starting the program, the system recognizes that this program depends on dynamic libraries. So it calls a loader, `/lib/ld-linux.so.X` (where X is a version number), to load the required libraries. You can use `ldd` to determine which libraries an executable depends on.

ldd

```
 $ ldd client
 libprint.so.1 => ./libprint.so.1
 libc.so.6 => /lib/libc.so.6
 /lib/ld-linux.so.2=> /lib/ld-linux.so.2
```

This tells which libraries the file client relies on. These libraries will then be loaded by the loader, and the corresponding `_init()` sections will be called. The loader first looks for libraries in the path mentioned in the `LD_LIBRARY_PATH` environment variable and then turns to the standard paths mentioned in `/etc/ld.so.conf`. In case a library can't be found, an error is thrown. In normal circumstances, the loader loads the libraries and normal execution of the program resumes. All this is transparent and done behind the scenes.

Utility programs and tools

Now let's take a brief look at the file, nm and objdump tools, three very useful binary utilities.

The file program can be used to find out a file's type. The file being tested can be a text file (ex. libprint.c), an executable (ex. client) or a data (ex. /dev/hda5). File is very useful in finding out

which platform a particular file was compiled for and whether or not it's executable, among other things. It is invoked with the following command line:

File

```
$ file client
client: ELF 32-bit LSB executable, Intel 80386, version 1, dynamically linked
(uses shared libs), not stripped
```

nm lists all the symbols that exist in an object. (By object, we generally mean an object file or a library.) Passing an object through nm displays names of the functions used or exported by this object, the objects various sections, symbols, and the objects type. A symbol, for example, can be undefined or external; it may be global or some other identifier. Running `nm` on our `libprint.so` produces this output:

nm

```
$ nm libprint.so
00001490 A _DYNAMIC
00001480 A _GLOBAL_OFFSET_TABLE
00001510 A __bss_start
00001510 A  _edata
00001510 A _end
00000452 A _etext
00000400 T _fini
000003d8 T _init
000003d8 t gcc2_compiled
  U printf@@GLIBC_2.0
00000428 T printstring
```

Objdump displays information about object files, which can be specified on the command line. The options passed to objdump control what information gets displayed. It's a good utility to use for a detailed look at the internal of an object file.

Well, I hope you've had fun learning about shared objects. Stick around, there's lots more to come!

Related topics

- Review the File man page, which documents version 3.27 of the file command.
- The Objdump Man Page documents the objdump command.
- Check out the brief tutorial Building Shared objects on Sun Solaris.