Libraries

Libraries are collections of precompiled functions that have been written to be reusable. Typically, they consist of sets of related functions to perform a common task. Examples include libraries of screen-handling functions. Standard system libraries are usually stored in /lib and /usr/lib. The C compiler (or more exactly, the linker) needs to be told which libraries to search, because by default it searches only the standard C library.

The libraries usually exist in static and shared formats:

1. Static Libraries, have extension .a

2. Shared Libraries, have extension .so

1. Static Libraries

The simplest form of library is just a collection of object files kept together in a ready-to-use form. When a program needs to use a function stored in the library, it includes a header file that declares the function, The compiler and linker take care of combining the program code and the library into a single executable program. The –l option must be used to indicate which libraries other than the standard C runtime library are required.

Static libraries, also known as archives, conventionally have names that end with .a .

Examples are /usr/lib/libc.a and /usr/lib/libX11.a for the standard C library and the X11 library, respectively.

You can create and maintain your own static libraries very easily by using the ar (for archive) program and compiling functions separately with gcc -c . Try to keep functions in separate source files as much as possible.

Steps to create a static Library

1. First, create separate source files for each function. (For example fact.c mul.c)

2. Compile these functions individually to produce object files ready for inclusion into a library. Do this by invoking the C compiler with the -c option.

$ gcc -c fact.c

$ gcc -c mul.c

3. Create a header file for your library. This will declare the functions in your library and should be included by all applications that want to use your library.

4. Create and use a library. Use the ar program to create the archive and add your object files to it. The program is called ar because it creates archives, or collections, of individual files placed together in one large file. Note that you can also use ar to create archives of files of any type.

$ ar crv libfoo.a fact.o mul.o

5. The library is created and the two object files added. To use the library successfully,Now write a program that calls the functions.

Your library is now ready to use. You can add to the list of files to be used by the compiler to create your program like this:

$ gcc main.c -o main libfoo.a

You could also use the –l option to access the library, but because it is not in any of the standard places, you have to tell the compiler where to find it by using the –L option like this:

$ gcc main.c -o m –L. –lfoo

The –L. option tells the compiler to look in the current directory (.) for libraries. The –lfoo option tells the compiler to use a library called libfoo.a (or a shared library, libfoo.so , if one is present). To see which functions are included in an object file, library, or executable program, you can use the nm command. If you take a look at program and libfoo.a , you see that the library contains both fact and mul, When the program is created, it includes only functions from the library that it actually needs. Including the header file, which contains declarations for all of the functions in the library, doesn’t cause the entire library to be included in the final program.

2. Shared Libraries

Shared Libraries are the libraries that can be linked to any program at run-time. They provide a means to use code that can be loaded anywhere in the memory. Once loaded, the shared library code can be used by any number of programs. So, this way the size of programs(using shared library) and the memory footprint can be kept low as a lot of code is kept common in form of a shared library.

Shared libraries provide modularity to the development environment as the library code can be changed, modified and recompiled without having to re-compile the applications that use this library. For example, for any change in the pthread library code, no change is required in the programs using pthread shared library. A shared library can be accessed through different names :

Name used by linker (‘lib’ followed by the library name, followed by ‘.so’ . For example libpthread.so)

Fully qualified name or soname (‘lib’ followed by the library name, followed by ‘.so’, followed by ‘.’ and a version number. For example : libpthread.so.1)

Real name (‘lib’ followed by the library name, followed by ‘.so’, followed by ‘.’ and a version number, followed by a ‘.’ and a minor number, followed by a ‘.’ and a release number. Release number is optional. For example, libpthread.so.1.1)

A version number is changed for a shared library when the changes done in the code make the shared library incompatible with the previous version. For example, if a function is completely removed then a new version of the library is required.

A minor number is changed in case there is a modification in the code that does not make the shared library incompatible with the previous version being used. For example, a small bug fix won’t break the compatibility of the existing shared library so only a minor number is changed while version remains the same.

These naming conventions help multiple versions of same shared library to co-exist in a system. The programs linking with the shared library do not need to take care about the latest version of the shared library installed in the system. Once the latest version of the shared library is installed successfully, all the programs automatically start linking to the latest version.

The name used by linker is usually a symbolic link to the fully qualified soname which in turn is a symbolic link to the real name.

Placement in File System

There are mainly three standard locations in the filesystem where a library can be placed.

/lib

/usr/lib

/usr/local/lib

We will go by the Filesystem Hierarchy standards(FHS) here. According to the FHS standards, All the libraries which are loaded at start up and running in the root filesystem are kept in /lib. While the libraries that are used by system internally are stored at /usr/lib. These libraries are not meant to be directly used by users or shell scripts. There is a third location /usr/local/lib(though it is not defined in the latest version of FHS). If it exists, it contains all the libraries that are not part of standard distribution. These non-standard libraries are the one’s which you download and could be possibly buggy.

Using ldconfig

Once a shared library is created, copy the shared library to directory in which you want the library to reside (for example /usr/local/lib or /usr/lib). Now, run ldconfig command in this directory.

What does ldconfig do?

You remember that we discussed earlier that a linker name for shared library is a symbolic link to the fully qualified soname which in turn is a symbolic link to the real name. Well, this command does exactly the same.

When you run an ELF executable, by default the loader is run first. The loader itself is a shared object file /lib/ld-linux.so.X where 'X' is a version number. This loader in turn finds and loads all the shared libraries on which our program depends.

All the directories that are searched by the loader in order to find the libraries is stored in /etc/ld.so.conf. Searching all the directories specified in /etc/ld.so.conf file can be time consuming so every time ldconfig command is run, it sets up the required symbolic links and then creates a cache in file /etc/ld.so.cache where all the information required for executable is written. Reading information from cache is very less time consuming. The catch here is that ldconfig command needs to be run every-time a shared library is added or removed. So on start-up the program uses /etc/ld.so.cache to load the libraries it requires.

Using Non Standard Library Locations

When using non standard library locations. One of the following three steps could be carried out :

Add the path to /etc/ld.so.conf file. This file contains paths to all the directories in which the library is searched by the loader. This file could sometime contain a single line like :

include /etc/ld.so.conf.d/*.conf

In that case, just create a conf file in the same directory. You can directly add a directory to cache by using the following command :

ldconfig -n [non standard directory path containing shared library]

Note that this is a temporary change and will be lost once the system is rebooted. Update the environment variable LD_LIBRARY_PATH to point to your directory containing the shared

library. Loader will use the paths mentioned in this environment variable to resolve dependencies.

Note that on some Unix systems the name of the environment variable could differ.

Note: On a related topic, as we explained earlier, there are four main stages through which a source code passes in order to finally become an executable.

Example (How to Create a Shared Library)

Lets take a simple practical example to see how we can create and use shared libraries. The following is the piece of code (shared.c) that we want to put in a shared library :

```
#include "shared.h"

unsigned int add(unsigned int a, unsigned int b)

{

   printf("\n Inside add()\n");

   return (a+b);

}
```

shared.h looks like :

```
#include<stdio.h>

extern unsigned int add(unsigned int a, unsigned int b);
```

Lets first make shared.c as a shared library.

1. Run the following two commands to create a shared library :

```
$gcc -c -Wall -Werror -fPIC shared.c
```

$gcc -shared -o libshared.so shared.o

The first command compiles the code shared.c into position independent code which is required for a shared library.

The second command actually creates a shared library with name ‘libshared.so’.

2. Here is the code of the program that uses the shared library function ‘add()’

```
#include<stdio.h>

#include"shared.h"

int main(void)

{

   unsigned int a = 1;

   unsigned int b = 2;

   unsigned int result = 0;

   result = add(a,b);

   printf("\n The result is [%u]\n",result);

   return 0;

}
```

3. Next, run the following command :

```
gcc -L/home/himanshu/practice/ -Wall main.c -o main -lshared
```

This command compiles the main.c code and tells gcc to link the code with shared library libshared.so (by using flag -l) and also tells the location of shared file(by using flag -L).

4. Now, export the path where the newly created shared library is kept by using the following command :

export LD_LIBRARY_PATH=/home/himanshu/practice:$LD_LIBRARY_PATH

The above command exports the path to the environment variable ‘LD_LIBRARY_PATH’.

5. Now run the executable ‘main’ :

$ ./main

Inside add()

The result is [3]

So we see that shared library was loaded and the add function inside it was executed.