

Static and Dynamic Libraries | Set 1

When a C program is compiled, the compiler generates object code. After generating the object code, the compiler also invokes linker. One of the main tasks for linker is to make code of library functions (eg `printf()`, `scanf()`, `sqrt()`, ..etc) available to your program. A linker can accomplish this task in two ways, by copying the code of library function to your object code, or by making some arrangements so that the complete code of library functions is not copied, but made available at run-time.

Static Linking and Static Libraries is the result of the linker making copy of all used library functions to the executable file. Static Linking creates larger binary files, and need more space on disk and main memory. Examples of static libraries (libraries which are statically linked) are, ***.a*** files in Linux and ***.lib*** files in Windows.

Steps to create a static library Let us create and use a Static Library in UNIX or UNIX like OS.

1. Create a C file that contains functions in your library.

```
/* Filename: lib_mylib.c */
#include <stdio.h>
void fun(void) {
    printf("fun() called from a static library");
}
```

We have created only one file for simplicity. We can also create multiple files in a library.

2. Create a header file for the library

```
/* Filename: lib_mylib.h */
void fun(void);
```

3. Compile library files.

```
gcc -c lib_mylib.c -o lib_mylib.o
```

4. Create static library. This step is to bundle multiple object files in one static library (see **ar** for details). The output of this step is static library.

```
ar rcs lib_mylib.a lib_mylib.o
```

5. Now our static library is ready to use. At this point we could just copy lib **_mylib.a** somewhere else to use it. For demo purposes, let us keep the library in the current directory.

Let us create a driver program that uses above created static library.

1. Create a C file with main function

```
/* filename: driver.c */
#include "lib_mylib.h"
void main() {
    fun();
}
```

2. Compile the driver program.

```
gcc -c driver.c -o driver.o
```

3. Link the compiled driver program to the static library. Note that `-L.` is used to tell that the static library is in current folder (See [this](#) for details of `-L` and `-l` options).

```
gcc -o driver driver.o -L. -l_mylib
```

4. Run the driver program

```
./driver  
fun() called from a static library
```

Following are some important points about static libraries.

1. For a static library, the actual code is extracted from the library by the linker and used to build the final executable at the point you compile/build your application.
2. Each process gets its own copy of the code and data. Whereas in case of dynamic libraries it is only code shared, data is specific to each process. For static libraries memory footprints are larger. For example, if all the window system tools were statically linked, several tens of megabytes of RAM would be wasted for a typical user, and the user would be slowed down by a lot of paging.
3. Since library code is connected at compile time, the final executable has no dependencies on the library at run time i.e. no additional run-time loading costs, it means that you don't need to carry along a copy of the library that is being used and you have everything under your control and there is no dependency.
4. In static libraries, once everything is bundled into your application, you don't have to worry that the client will have the right library (and version) available on their system.
5. One drawback of static libraries is, for any change(up-gradation) in the static libraries, you have to recompile the main program every time.
6. One major advantage of static libraries being preferred even now "is speed". There will be no dynamic querying of symbols in static libraries. Many production line software use static libraries even today.

Dynamic linking and Dynamic Libraries Dynamic Linking doesn't require the code to be copied, it is done by just placing name of the library in the binary file. The actual linking happens when the program is run, when both the binary file and the library are in memory. Examples of Dynamic libraries (libraries which are linked at run-time) are, `*.so*` in Linux and `*.dll*` in Windows.

We will soon be covering more points on Dynamic Libraries and steps to create them.

This article is compiled by **Abhijit Saha** and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GeeksforGeeks has prepared a complete interview preparation course with premium videos, theory, practice problems, TA support and many more features. Please refer [Placement 100](#) for details

Working with Shared Libraries | Set 1

This article is not for those algo geeks. If you are interested in systems related stuff, just read on...

Shared libraries are useful in sharing code which is common across many applications. For example, it is more economic to pack all the code related to TCP/IP implementation in a shared library. However, data can't be shared as every application needs its own set of data. Applications like, browser, ftp, telnet, etc... make use of the shared 'network' library to elevate specific functionality.

Every operating system has its own representation and tool-set to create shared libraries. More or less the concepts are same. On Windows every object file (*.obj, *.dll, *.ocx, *.sys, *.exe etc...) follow a format called Portable Executable. Even shared libraries (called as Dynamic Linked Libraries or DLL in short) are also represented in PE format. The tool-set that is used to create these libraries need to understand the binary format. Linux variants follow a format called Executable and Linkable Format (ELF). The ELF files are position independent (PIC) format. Shared libraries in Linux are referred as shared objects (generally with extension *.so). These are similar to DLLs in Windows platform. Even shared object files follow the ELF binary format.

Remember, the file extensions (*.dll, *.so, *.a, *.lib, etc...) are just for programmer convenience. They don't have any significance. All these are binary files. You can name them as you wish. Yet ensure you provide absolute paths in building applications.

In general, when we compile an application the steps are simple. Compile, Link and Load. However, it is not simple. These steps are more versatile on modern operating systems.

When you link your application against static library, the code is part of your application. There is no dependency. Even though it causes the application size to increase, it has its own advantages. The primary one is speed as there will be no symbol (a program entity) resolution at runtime. Since every piece of code part of the binary image, such applications are independent of version mismatch issues. However, the cost is on fixing an issue in library code. If there is any bug in library code, entire application need to be recompiled and shipped to the client. In case of dynamic libraries, fixing or upgrading the libraries is easy. You just need to ship the updated shared libraries. The application need not to recompile, it only need to re-run. You can design a mechanism where we don't need to restart the application.

When we link an application against a shared library, the linker leaves some stubs (unresolved symbols) to be filled at application loading time. These stubs need to be filled by a tool called, *dynamic linker* at run time or at application loading time. Again loading of a library is of two types, static loading and dynamic loading. Don't confuse between ***static loading*** vs ***static linking*** and ***dynamic loading*** vs ***dynamic linking***.

For example, you have built an application that depends on *libstdc++.so* which is a shared object (dynamic library). How does the application become aware of required shared libraries? (If you are interested, explore the tools *tdump* from Borland tool set, *objdump* or *nm* or *readelf* tools on Linux).

Static loading:

- In static loading, all of those dependent shared libraries are loaded into memory even before the application starts execution. If loading of any shared library fails, the application won't run.
- A dynamic loader examines application's dependency on shared libraries. If these libraries are already loaded into the memory, the library address space is mapped to application virtual address space (VAS) and the dynamic linker does relocation of unresolved symbols.
- If these libraries are not loaded into memory (perhaps your application might be first to invoke the shared library), the loader searches in standard library paths and loads them into memory, then maps and resolves symbols. Again loading is big process, if you are interested write your own loader :).
- While resolving the symbols, if the dynamic linker not able to find any symbol (may be due to older version of shared library), the application can't be started.

Dynamic Loading:

- As the name indicates, dynamic loading is about loading of library on demand.
- For example, if you want a small functionality from a shared library. Why should it be loaded at the application load time and sit in the memory? You can invoke loading of these shared libraries dynamically when you need their functionality. This is called dynamic loading. In this case, the programmer aware of situation 'when should the library be loaded'. The tool-set and relevant kernel provides API to support dynamic loading, and querying of symbols in the shared library.

More details in later articles.

Note: If you come across terms like loadable modules or equivalent terms, don't mix them with shared libraries. They are different from shared libraries. The kernels provide framework to support loadable modules.

Working with Shared Libraries | Set 2

Exercise:

1. Assuming you have understood the concepts, How do you design an application (e.g. Banking) which can upgrade to new shared libraries without re-running the application.

— **Venki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

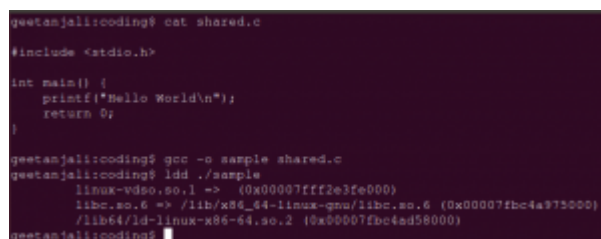
GeeksforGeeks has prepared a complete interview preparation course with premium videos, theory, practice problems, TA support and many more features. Please refer **Placement 100** for details

Working with Shared Libraries | Set 2

We have covered basic information about shared libraries in the **previous post**. In the current article we will learn how to create shared libraries on Linux.

Prior to that we need to understand how a program is loaded into memory, various (basic) steps involved in the process.

Let us see a typical "Hello World" program in C. Simple Hello World program screen image is given below.



```
geetanjali:coding$ cat shared.c
#include <stdio.h>

int main() {
    printf("Hello World\n");
    return 0;
}

geetanjali:coding$ gcc -o sample shared.c
geetanjali:coding$ ldd ./sample
        linux-vdso.so.1 => (0x00007ffff2e3fe000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fbc4a975000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fbc4ad58000)
geetanjali:coding$
```

We were compiling our code using the command "**gcc -o sample shared.c**". When we compile our code, the compiler won't resolve implementation of the function **printf()**. It only verifies the syntactical checking. The tool chain leaves a stub in our application which will be filled by dynamic linker. Since printf is standard function the compiler implicitly invoking its shared library. More details down.

We are using `ldd` to list dependencies of our program binary image. In the screen image, we can see our sample program depends on three binary files namely, `linux-vdso.so.1`, `libc.so.6` and `/lib64/ld-linux-x86-64.so.2`.

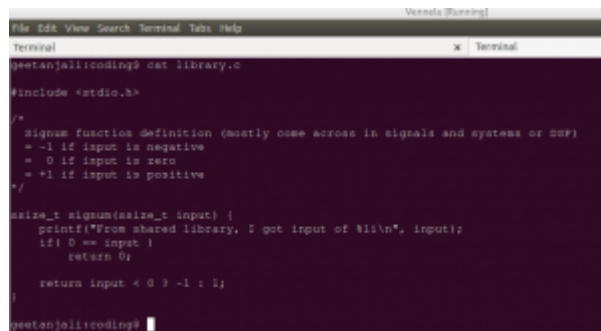
The file VDSO is fast implementation of system call interface and some other stuff, it is not our focus (on some older systems you may see different file name in lieu of `.vso`). Ignore this file. We have interest in the other two files.

The file **libc.so.6** is C implementation of various standard functions. It is the file where we see `printf` definition needed for our *Hello World*. It is the shared library needed to be loaded into memory to run our Hello World program.

The third file `/lib64/ld-linux-x86-64.so.2` is in fact an executable that runs when an application is invoked. When we invoke the program on bash terminal, typically the bash forks itself and replaces its address space with image of program to run (so called fork-exec pair). The kernel verifies whether the `libc.so.6` resides in the memory. If not, it will load the file into memory and does the relocation of `libc.so.6` symbols. It then invokes the dynamic linker (`/lib64/ld-linux-x86-64.so.2`) to resolve unresolved symbols of application code (`printf` in the present case). Then the control transfers to our program *main*. (I have intentionally omitted many details in the process, our focus is to understand basic details).

Creating our own shared library:

Let us work with simple shared library on Linux. Create a file **library.c** with the following content.



```
geetanjali:coding$ cat library.c
#include <stdio.h>

/*
 * Signum function definition (mostly come across in signals and systems or OSF)
 * = -1 if input is negative
 * = 0 if input is zero
 * = +1 if input is positive
 */
ssize_t signum(ssize_t input) {
    printf("From shared library, I got input of %i\n", input);
    if( 0 == input )
        return 0;

    return input < 0 ? -1 : 1;
}

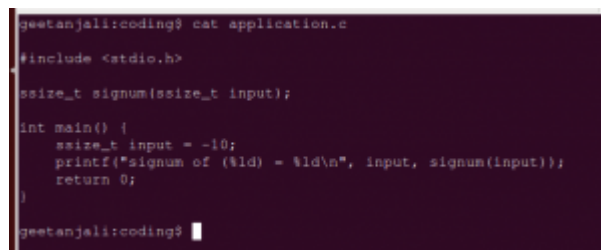
geetanjali:coding$
```

The file `library.c` defines a function **signum** which will be used by our application code. Compile the file `library.c` file using the following command.

gcc -shared -fPIC -o liblibrary.so library.c

The flag `-shared` instructs the compiler that we are building a shared library. The flag `-fPIC` is to generate position independent code (ignore for now). The command generates a shared library `liblibrary.so` in the current working directory. We have our shared object file (shared library name in Linux) ready to use.

Create another file **application.c** with the following content.



```
geetanjali:coding$ cat application.c
#include <stdio.h>

ssize_t signum(ssize_t input);

int main() {
    ssize_t input = -10;
    printf("signum of (%i) = %i\n", input, signum(input));
    return 0;
}

geetanjali:coding$
```

In the file **application.c** we are invoking the function `signum` which was defined in a shared library. Compile the `application.c` file using the following command.

gcc application.c -L /home/geetanjali/coding/ -llibrary -o sample

The flag `-llibrary` instructs the compiler to look for symbol definitions that are not available in the current code (signature function in our case). The option `-L` is hint to the compiler to look in the directory followed by the option for any shared libraries (during link time only). The command generates an executable named as “**sample**”.

If you invoke the executable, the dynamic linker will not be able to find the required shared library. By default it won't look into current working directory. You have to explicitly instruct the tool chain to provide proper paths. The dynamic linker searches standard paths available in the `LD_LIBRARY_PATH` and also searches in system cache (for details explore the command `*ldconfig*`). We have to add our working directory to the `LD_LIBRARY_PATH` environment variable. The following command does the same.

```
export LD_LIBRARY_PATH=/home/geetanjali/coding/:$LD_LIBRARY_PATH
```

You can now invoke our executable as shown.

```
./sample
```

Sample output on my system is shown below.



```
geetanjali@coding:~$ export LD_LIBRARY_PATH=/home/geetanjali/coding/:$LD_LIBRARY_PATH
geetanjali@coding:~$ ./sample
From shared library, I got input of 10
Signum of 1-10 = -1
```

Note: The path `/home/geetanjali/coding/` is working directory path on my machine. You need to use your working directory path where ever it is being used in the above commands.

Stay tuned, we haven't even explored 1/3rd of shared library concepts. More advanced concepts in the later articles.

Exercise:

It is workbook like article. You won't gain much unless you practice and do some research.

\1. Create similar example and write your won function in the shared library. Invoke the function in another application.

\2. Is (Are) there any other tool(s) which can list dependent libraries?

\3. What is position independent code (PIC)?

\4. What is system cache in the current context? How does the directory `/etc/ld.so.conf.d/*` related in the current context?

— **Venki** . Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GeeksforGeeks has prepared a complete interview preparation course with premium videos, theory, practice problems, TA support and many more features. Please refer **Placement 100** for details