



Working with Shared Libraries | Set 2

[Read](#)[Discuss](#)[Courses](#)

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, and the 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 => (0x00007fff2e3fe000)
        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 a dynamic linker. Since **printf** is a standard function the compiler implicitly invokes 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*.

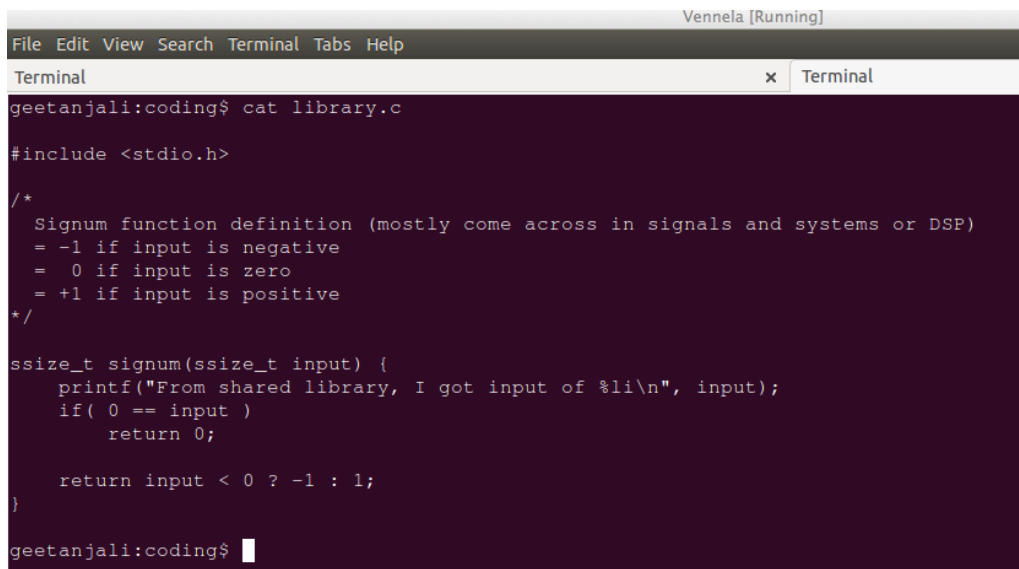
The file VDSO is a fast implementation of the system call interface and some other stuff, it is not our focus (on some older systems you may see the different file name in a line of *.vso.*). Ignore this file. We have an interest in the other two files.

The file **libc.so.6** is the 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 the bash terminal, typically the bash forks itself and replaces its address space with an image of the 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 the 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 a simple shared library on Linux. Create a file **library.c** with the following content.

A screenshot of a terminal window titled 'Vennela [Running]'. The terminal shows the command 'cat library.c' being executed. The output is the content of the file library.c, which defines a function 'signum' that takes an 'ssize_t' input and returns -1 for negative, 0 for zero, and 1 for positive values. The terminal prompt is 'geetanjali:coding\$'.

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

[Aptitude](#) [Engineering Mathematics](#) [Discrete Mathematics](#) [Operating System](#) [DBMS](#) [Computer Networks](#)

Annonce supprimée. [Détails](#)

`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 (%ld) = %ld\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 (`signum` function in our case). The option `-L` is a 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 **“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 the 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 the 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
```

```
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 (-10) = -1
```

Note: The path `/home/geetanjali/coding/` is a working directory path on my machine. You need to use your working directory path wherever 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 a workbook like an article. You won't gain much unless you practice and do some research.

1. Create a similar example and write your own function in the shared library. Invoke the function in another application.
2. Is (Are) there any other tool(s) that 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 if you want to share more information about the topic discussed above.

Whether you're preparing for your first job interview or aiming to upskill in this ever-evolving tech landscape, [GeeksforGeeks Courses](#) are your key to success. We provide top-quality content at affordable prices, all geared towards accelerating your growth in a time-bound manner. Join the millions