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()

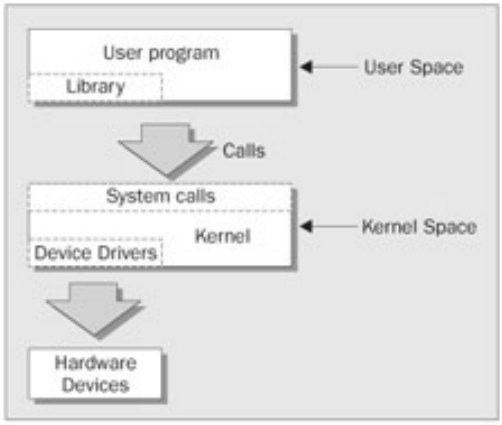
The result is [3]

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

File Input Ouput Operations

System Calls :

The files and devices can be accessed using a small functions, These functions are known as system calls. They are provided by Linux directly and they are the interface to the operating system itself.



Low−level File Access

Each running program, called a process, has associated with it a number of file descriptors. These are small integers that can be used to access open files or devices. How many of these are available will vary depending on how the Linux system has been configured. When a program starts, it usually has three of these descriptors already opened. These are:

0	Standard Input
1	Standard Output
2	Standard Error

Other file descriptors are associated with files and devices by using the open system call

open

To create a new file descriptor the open system call is used.

```
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
int open(const char *path, int oflags);
int open(const char *path, int oflags, mode_t mode);
```

In simple terms, open establishes an access path to a file or device. If successful, it returns a file descriptor that can be used in read, write and other system calls.

Open returns the new file descriptor (always a non−negative integer) if successful, or −1 if it fails, open also sets the global variable errno to indicate the reason for the failure.

The file descriptor is unique and isn't shared by any other processes that may be running. If two programs have a file open at the same time, they maintain distinct file descriptors. If they both write to the file, they will continue to write where they left off. Their data isn't interleaved, but one will overwrite the other. Each keeps its own idea of how far into the file (the offset) it has read or written.

The name of the file or device to be opened is passed as a parameter, path, and the oflags parameter is used to specify actions to be taken on opening the file. The oflags are specified as a bitwise OR of a mandatory file access mode and other optional modes. The open call must specify one of the following file access modes:

Mode	**Description**
O_RDONLY	Open for read−only
O_WRONLY	Open for write−only
O_RDWR	Open for reading and writing

The call may also include a combination (bitwise OR) of the following optional modes in the oflags

parameter:

O_APPEND	Place written data at the end of the file.
O_TRUNC	Set the length of the file to zero, discarding existing contents.
O_CREAT	Creates the file, if necessary, with permissions given in mode.
O_EXCL	Used with O_CREAT, ensures that the caller creates the file. The open is atomic, i.e. it's performed with just one function call. This protects against two programs creating the file at the same time. If the file already exists, open will fail.

Other possible values for oflags are documented in the open manual page, found in section 2 of the manual (use man 2 open).

Initial Permissions:

When a file is created using the O_CREAT flag with open, three parameter form is used. mode, the third parameter, is made from a bitwise OR of the flags defined in the header file sys/stat.h. These are:

- **S_IRUSR** - Read permission, owner.
- **S_IWUSR** - Write permission, owner.
- **S_IXUSR** - Execute permission, owner.
- **S_IRGRP** - Read permission, group.
- **S_IWGRP** - Write permission, group.
- **S_IXGRP** - Execute permission, group.

• **S_IROTH** - Read permission, others.

• **S_IWOTH** - Write permission, others.

• **S_IXOTH** - Execute permission, others.

Write:

```
#include <unistd.h>
size_t write(int fildes, const void *buf, size_t nbytes);
```

The write system call arranges for the first nbytes bytes from buf to be written to the file associated with the file descriptor fildes. It returns the number of bytes actually written. This may be less than nbytes if there has been an error in the file descriptor. If the function returns 0, it means no data was written, if −1, there has been an error in the write call and the error will be specified in the errno global variable.

Read:

```
#include <unistd.h>
size_t read(int fildes, void *buf, size_t nbytes);
```

The read system call reads up to nbytes bytes of data from the file associated with the file descriptor fildes and places them in the data area buf. It returns the number of data bytes actually read, which may be less than the number requested.

If a read call returns 0, it had nothing to read; it reached the end of the file. Again, an error on the call will cause it to return −1.

Close:

```
#include <unistd.h>
int close(int fildes);
```

close is used to terminate the association between a file descriptor, fildes, and its file. The file descriptor becomes available for reuse. It returns 0 if successful and −1 on error. Note that it can be important to check the return result from close.

Ioctl:

```
#include <unistd.h>
int ioctl(int fildes, int cmd, ...);
```

ioctl provides an interface for controlling the behavior of devices, their descriptors and configuring underlying services. Terminals, file descriptors, sockets, even tape drives may have ioctl calls defined for them and you need to refer to the specific device's man page for details.

ioctl performs the function indicated by cmd on the object referenced by the descriptor fildes. It may take an optional third argument depending on the functions supported by a particular device.

Processes

Duplicating a Process Image:

A new process is created by calling fork . This system call duplicates the current process, creating a new entry in the process table with many of the same attributes as the current process. The new process is almost identical to the original, executing the same code but with its own data space, environment, and file descriptors. The call to fork in the parent returns the PID of the new child process. The new process continues to execute just like the original, with the exception that in the child process the call to fork returns 0 . This allows both the parent and child to determine which is which.

If fork fails, it returns -1 . This is commonly due to a limit on the number of child processes that a parent may have (CHILD_MAX), in which case errno will be set to EAGAIN . If there is not enough space for an entry in the process table, or not enough virtual memory, the errno variable will be set to ENOMEM .

#include <sys/types.h>

#include <unistd.h>

pid_t fork(void);

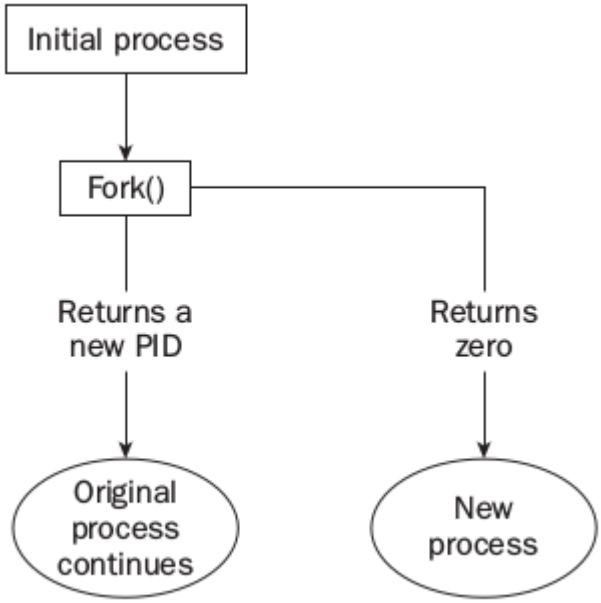


Fig: Creation of Process with fork

A typical code fragment using fork is

```
pid_t new_pid;

new_pid = fork();

switch(new_pid) {
case -1 :      /* Error */
    break;
case 0 :       /* We are child */
    break;
default :      /* We are parent */
    break;
}
```

Waiting for a Process:

When a child process is started with fork , it takes on a life of its own and runs independently. Sometimes,If the parent finishes ahead of the child and you get some messy output as the child

*conti*nues to run. The child is adapted by other process and it contintues its execution. You can arrange for the parent process to wait until the child finishes before continuing by calling wait .

#include <sys/types.h>

#include <sys/wait.h>

pid_t wait(int *stat_loc);

The wait system call causes a parent process to pause until one of its child processes is stopped. The call returns the PID of the child process.This will normally be a child process that has terminated. The status information allows the parent process to determine the exit status of the child process, that is, the value returned from main or passed to exit . If stat_loc is not a null pointer, the status information will be written to the location to which it points.

You can interpret the status information using macros defined in sys/wait.h , shown in the following table.

Macro	Definition
`WIFEXITED(stat_val)`	Nonzero if the child is terminated normally.
`WEXITSTATUS(stat_val)`	If `WIFEXITED` is nonzero, this returns child exit code.
`WIFSIGNALED(stat_val)`	Nonzero if the child is terminated on an uncaught signal.
`WTERMSIG(stat_val)`	If `WIFSIGNALED` is nonzero, this returns a signal number.
`WIFSTOPPED(stat_val)`	Nonzero if the child has stopped.
`WSTOPSIG(stat_val)`	If `WIFSTOPPED` is nonzero, this returns a signal number.

There’s another system call that you can use to wait for child processes. It’s called waitpid , and you can

use it to wait for a specific process to terminate.

#include <sys/types.h>

#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *stat_loc, int options);

The pid argument specifies the PID of a particular child process to wait for. If it's -1 , waitpid will return information for any child process. Like wait , it will write status information to the location pointed to by stat_loc , if that is not a null pointer. The options argument allows you to modify the behavior of waitpid . The most useful option is WNOHANG , which prevents the call to waitpid from suspending execution of the caller. You can use it to find out whether any child processes have terminated and, if not, to continue. Other options are the same as for wait .

So, if you wanted to have a parent process regularly check whether a specific child process has terminated, you could use the call waitpid(child_pid, (int *) 0, WNOHANG);

This will return zero if the child has not terminated or stopped, or child_pid if it has. waitpid will return -1 on error and set errno . This can happen if there are no child processes (errno set to ECHILD),

if the call is interrupted by a signal (EINTR), or if the option argument is invalid (EINVAL).

Zombie Processes

Using fork to create processes can be very useful, but you must keep track of child processes. When a child process terminates, an association with its parent survives until the parent in turn either terminates normally or calls wait . The child process entry in the process table is therefore not freed up immediately. Although no longer active, the child process is still in the system because its exit code needs to be stored in case the parent subsequently calls wait . It becomes what is known as defunct, or a zombie process.

If the parent then terminates abnormally, the child process automatically gets the process with PID 1 (init) as parent. The child process is now a zombie that is no longer running but has been inherited by init because of the abnormal termination of the parent process. The zombie will remain in the process table until collected by the init process. The bigger the table, the slower this procedure. You need to avoid zombie processes, because they consume resources until init cleans them up.

Replacing a Process Image

There is a whole family of related functions grouped under the exec heading. They differ in the way that they start processes and present program arguments. An exec function replaces the current process with a new process specified by the path or file argument. You can use exec functions to "hand off" execution of your program to another.These functions belong to two types. execl , execlp , and execle take a variable number of arguments ending with a null pointer. execv and execvp have as their second argument an array of strings.

The functions with names suffixed with a p differ in that they will search the PATH environment variable to find the new program executable file. If the executable isn't on the path, an absolute filename, including directories, will need to be passed to the function as a parameter.

The global variable environ is available to pass a value for the new program environment. Alternatively,an additional argument to the functions execle and execve is available for passing an array of strings to be used as the new program environment.

If you want to use an exec function to start the ps program, you can choose from among the six exec family functions, as shown in the calls in the code fragment that follows:

```
#include <unistd.h>

/* Example of an argument list */
/* Note that we need a program name for argv[0] */
char *const ps_argv[] =
    {"ps", "ax", 0};

/* Example environment, not terribly useful */
char *const ps_envp[] =
    {"PATH=/bin:/usr/bin", "TERM=console", 0};

/* Possible calls to exec functions */
execl("/bin/ps", "ps", "ax", 0);            /* assumes ps is in /bin */
execlp("ps", "ps", "ax", 0);                /* assumes /bin is in PATH */
execle("/bin/ps", "ps", "ax", 0, ps_envp);  /* passes own environment */

execv("/bin/ps", ps_argv);
execvp("ps", ps_argv);
execve("/bin/ps", ps_argv, ps_envp);
```