

Contents

1	Introduction	7
1.1	System programming concepts	7
1.2	System Call	9
1.2.1	Library	9
1.2.2	Getting into system call	9
1.2.3	Internal mechanism for system call	10
2	File I/O	11
2.1	Introduction	11
2.2	File types	11
3	Processes	13
3.1	References	13
3.2	Introduction	13
3.2.1	Threads	13
3.2.2	Visualizing processes	14
3.3	Process ID	14
3.4	Process states	15
3.5	Process Control Block	16
3.6	Process creation	17
3.6.1	Copy on- write	17
3.7	Waiting a process	19
3.7.1	Functions	19
3.8	Abnormal Processes	20
3.8.1	Zombie process	20
3.9	Memory Layout	21
3.9.1	Code Example	21
3.10	Exec Family	22
3.11	Notes and summary	23
3.12	TODO for Processes	24
4	Signals	25
4.1	Intro	25
4.2	Examples	26
5	Virtual Memory	27
5.1	The big picture	27

6	Threads	29
6.1	Introduction	29
6.2	Race Condition	29
6.2.1	Race Condition Question	29
6.2.2	Mutex	30
6.3	General Points	30
7	Libraries in C/C++	31
7.1	Introduction	31
7.2	Some Tools	31
7.3	Static Libraries	31
7.3.1	Creating a static library	32
7.3.2	Pros and Cons	32
7.4	Shared Library	32
7.4.1	Creating a shared library	32
7.4.2	Compiling C code with a shared library	33
7.4.3	Running the program	33

Todo list

Linux layer	7
kernel mode	8
kernel function	8
Visualizing processes	14
Process Intro	14
wating process motivation	19
Video	19
Naming	20
zombie process	20
Zombie process concept	20
Code Example	21
Alarm signal	26
Memory Management	27
intro to thread	29
Race Conditions points	29
Race Condition Questions	29

Abstract

The goal of this reader is to summarize my learning in system programming in `C`.

Chapter 1

Introduction

1.1 System programming concepts

- purpose for user mode: user application software must prevent to access the hardware directly, because it could damage the hardware.

Hence the kernel provide a layer between the user application and the hardware.

- If a user program needs to access hardware, it will be done via the kernel mode
 - In this mode, we have a set of calls and programs to access the hardware
 - Once the required information is fetched from the hardware, it will be passed to user application

- What is a kernel?

The kernel is the core of an OS, which handles resources.

It resides inside the memory, and it is the 1st thing that comes at the start of a bootloader

- Function of a kernel
 - file management
 - memory management: making the memory available to different processes via the virtual memory management
 - Interprocess communication: how different processes communicate with each other
 - Threads (so multitasking system)

- Linux as layer view

- Linux hide hardware from direct access by user application, because if so it can damage the hardware
- Linux layer: to insert a picture later

Linux layer

- When accessing hardware, the OS turns into kernel mode

- *kernel mode: to see if this is called also privilege mode*

kernel mode

kernel functions and concept

kernel funct

- *to define and write later different new concept used, like what is a process, a thread, concepts about Linux OS, ...*
- *Add references about system programming later*

1.2 System Call

1.2.1 Library

Before diving into system call, let's fix the terminology first.

In a normal C program, we need to include *header files* to use some function, such as `stdio.h` to use `printf()`. This type of library is called *C standard library*.

1.2.2 Getting into system call

Now in system programming, we have some libraries which are designed to operate in a system call.

So what is a system call?

System calls (often shortened to *syscalls*) are function invocations made from user space—your text editor, favorite game, and so on—into the kernel (the core internals of the system) in order to request some service or resource from the operating system (chapter 1 in [1])

- The programs run in a kernel mode during system call execution
- There are 2 ways to perform system call:
 - Or using directly the kernel like the function `write()`
 - Through a library function.
Example is `printf()` which in turns call `write()`

1.2.3 Internal mechanism for system call

- The functional block diagram is shown in [Figure 1.1](#).

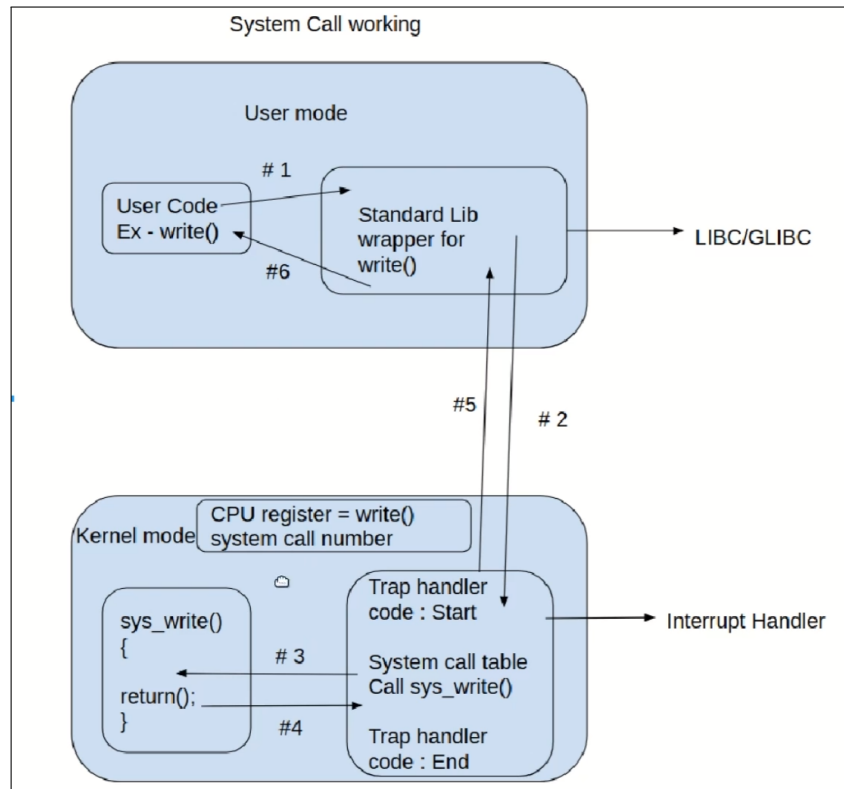


Figure 1.1: Main Components

- Every system call has a unique number associated with it
- When we use `write()` function in our code, it will not directly invoke the definition (implementation), rather it will call a *wrapper*
- This wrapper will raise an interrupt specific to hardware, and will compare the number associated to this function, in order to search to system call actual definition
- Finally we will get to the actual function and get back the outputs
- Then we turn our way up to the user space again with the correct output.

Chapter 2

File I/O

2.1 Introduction

Now we move for file manipulation in system programming.

Files are important in linux system programming, because everything is a treated as file in linux (chapter in 1 in [\[1\]](#)).

Under the hood info:

- Once we open a file, a metadata is attached to this open state is called file descriptor, known as `fd`, which is handled as `int` C type.

2.2 File types

There exist many file types in system programming, as shown in [Figure 2.1](#).

- - : regular file (data files).
- d : directory.
- c : character device file.
- b : block device file.
- s : local socket file.
- p : named pipe.
- l : symbolic link.

Figure 2.1: File types

Chapter 3

Processes

3.1 References

- chapter 5 in [1]
- chapter 2 in [2]

3.2 Introduction

In this section I will present multiple definitions in order to have a complete picture about processes.

1st definition:

Processes are next to files, the 2nd important abstraction unit in Unix (and OS in general) system.

In simple terms, we can define a process a program being executed such as C program. But a process is also much bigger than that: it contains kernel resources, virtualized memory, many threads,...

2nd definition of a processe:

Imagine we have some program written in some high level language such as C or java. Computers doesn't understand high level languages, this why we have compilers that compile our code, and transfrome it into an binary executable.

However, this binary executable is not enough to run the program, it needs to be ***loaded into the memory***, and allocate some resources. The OS is the brain which will do so. Once the program start running, *at this moment*, we can call it a process.

3.2.1 Threads

Now we move to the 2nd concepts which is thread. Thread is the basic unit wihtin a process. A process can have from 1 thread to many threads.

Programs, Process, Threads It should be noted that a program (for example a web browser) can have multiple process, and same between processes and threads.

3.2.2 Visualizing processes

Visualizing processes:

- *To insert the commands in Linux with some iamges*

Some concepts:

- a process is an exected program (such as a build C program)

Process Intro

- *To read later from books about processes, and the youtube videos I saw before*
- *To see the youtube playlist about processes*

youtube playlist

- <https://www.youtube.com/watch?v=OrM7nZcxXZU&list=PLBlNk6fEyqRgKl0MbI6kbI5ffNt7BF8>
 - Neso Academy for operating systems
 - Can serve as theory and complementary explanation for my books reading
- <https://www.youtube.com/watch?v=cex9XrZCU14&list=PLfqABt5AS4FkW5mOn2Tn9ZZLLDwA3k7>
 - nice playliste for examples in C
 - Contains alos a playlist for threads

3.3 Process ID

3.4 Process states

Now we have learned that a process is a program that is being executed (running). While this program is running, its *state is beign changed*, that is it passes from 1 state to another.

How we define the state of a program ? by the activity it is currently doing.

In [Figure 3.1](#) we have the different state, and also how they are running.

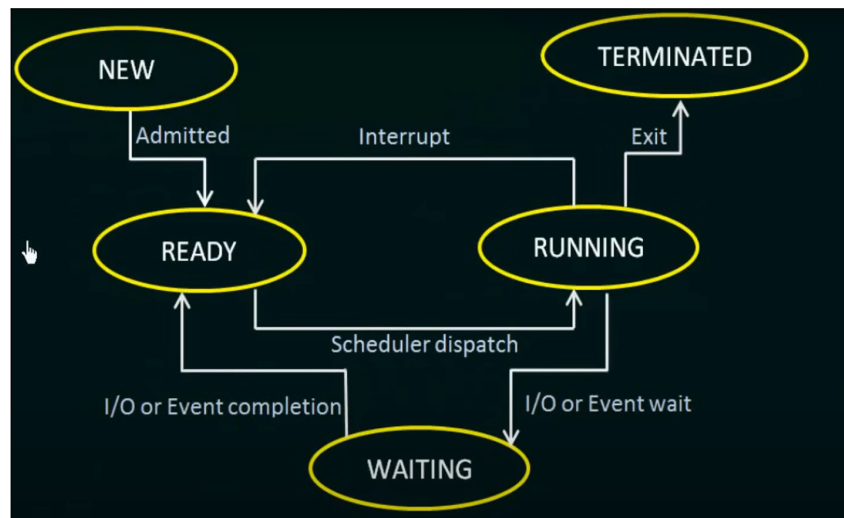


Figure 3.1: Different state of a process

- New: is the creation of the processe.
- Once the processe is being created, it enters into a ready staty to be assigned into a processor in order to run.
- Now the process is assigned, it enter the running state, that is being executed and doing its job
- After this running state, we have 3 cases:
 1. terminate: this is the normal case, that is the process has not been interrupted in some way
 2. the process get back to the ready state due to some interrupt signal , like some other process with higer priority need to be run.
 3. Finally, the current process maybe need some I/O or some signals. In that case, the process enter a waiting state in order to have these informations

3.5 Process Control Block

Each process is represented by the OS by a process control block (PCB). An example is shown in [Figure 3.2](#).

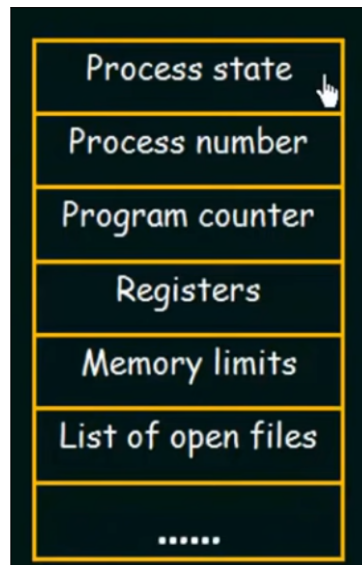


Figure 3.2: Process control block example

- program counter: pointer which tells us the address of the next instruction (instruction of the code) that will be executed
- Registers (CPU registers): tells us which registers are being used by this process (like stack pointers, general purpose registers, ...).
- Memory management information: section of the memory attached to this particular process
- I/O status info: which I/O will be used by the process

3.6 Process creation

Important points of the video: The following points are about the `fork()` system call used to create processes in linux

- once `fork()` is used to create the process, the child process will have the same resources as the parent process.
- The special thing about `fork()` is it returns value in 2 places:
 - In the parent process, the process ID `pid_t`
 - In the child process, it returns -1 or error
- The child process execute the same program as the parent process
- Each process can now modify their variables without affecting the other, and store in its correspondent memory segment (heap, stack, ...)

In [Figure 3.3](#), we have the state of before and after.

In other words, once we use the `fork()` in some C source file, starting from this line we have 2 branches (2 codes), each with a separate memory

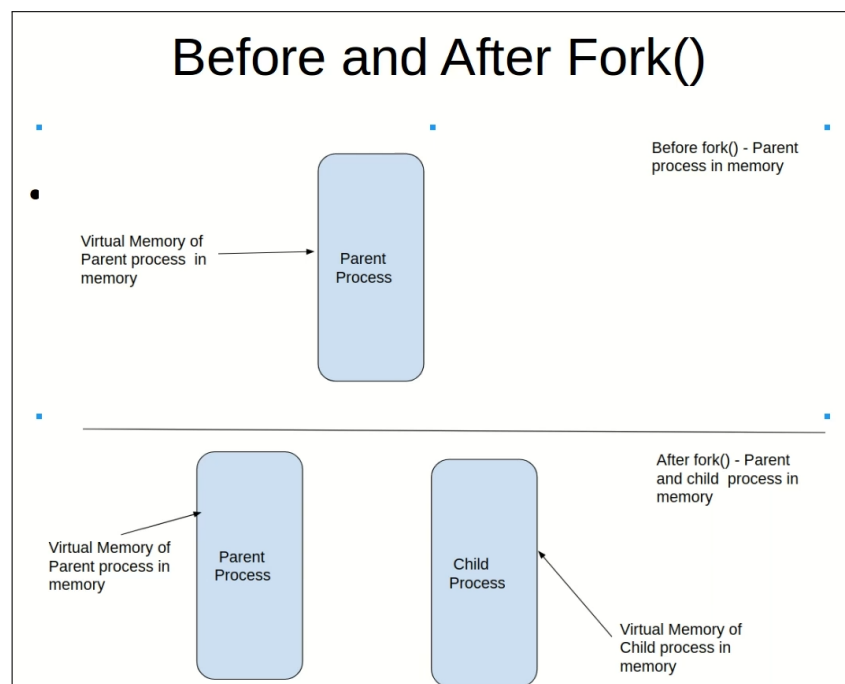


Figure 3.3: fork system call: before and after

3.6.1 Copy on- write

We can see that in [Figure 3.3](#) after `fork()` is being called, that we have 2 memory section: once for each process. So we can say that the OS make a copy of the memory. This was the old style employed by Unix system.

Now in modern Unix system such as Linux, copy is not made directly (to avoid unnecessary copy), and the 2 processes share the same memory. It is only when one of the processes need to alter or change some resources, copy will be made.

In other words, if one of the processes (parent or child) require some ***modification of the data***, only then a separate memory will be duplicated.

Hence the name copy on write.

For further information about the mechanism, see chapter 5 in [1].

3.7 Waiting a process

Video 38 of the course. I reached 29.37.

wating process motivation:

wating proc
motivation

- *To write a better motivation for this section after reading from the references [1], [3]*

Explanation:

- *To repharase it later*
- usually when we have 2 processes (a parent and a child), the parent need to wait for the child in order to terminate, and not the other way around
 - In other words, the child wait for the parent
 - In that case, the child will be assigned some parent process
 - This can cause a zombie process and we can have a memory leak
- Sometimes a process need to wait for the child process in order to terminate
- Each process which terminate has an exit status
- terminating a process is done via the `void exit(int status)`
 - The important thing about this function is that it does not return anything to see if the a certain process is finished or not
 - We need to see the `status` variable to know if the status has finished or not

Video

Video course: video 38, at 20:17.

3.7.1 Functions

- We have 2 functions to wait a process
- Header: `#include <sys/wait.h>`
- `int wait(int* status)` and

3.8 Abnormal Processes

Naming:

- *To see if I need to rename this section later.*

In this section we will see 3 special processes: orphan, zombie and sleeping process.

- If we have a parent and child process, and the parent process finish before the child (like it has exit), the child process will become an orphan process.

3.8.1 Zombie process

This section is from chapter 1 in [1].

- When a process terminates, it is not immediately removed from the system.
- Instead, the kernel keeps parts of the process resident in memory to allow the process's parent to inquire about its status upon terminating.
- This inquiry is known as waiting on the terminated process. Once the parent process has waited on its terminated child, the child is fully destroyed.
- A process that has terminated, but has not yet been waited upon, is called a zombie process.

zombie process :

- *To understand this sentence even more.*

Zombie process concept:

- *To understand later the concept of zombie*
- *From what I understand from the video, if a child process exit before the parent process and its entry table remains, the child process is known as a zombie process*
- *Using wait() or waitpid() in the parent process resolve the issue*
 - *To dig more why this is the solution*

3.9 Memory Layout

Now we discuss memory layout for the process.

Memory of a process is composed of different *segments* as enumerated below

1. Text: code reside here
 - that is the code of the program is being executed
 - this segment is a *read only* \leftrightarrow can't be altered by any pointer
2. Data: data variables during compile time
 - It is composed between uninitialized and initialized data
3. stack: for local variables and functions
 - the composition inside the stack is called *frame*
 - each frame contains 1 functions and its related variables
4. heap: dynamic variables

3.9.1 Code Example

Code Exam

Code Example:

- To do later some examples using some example programs
- To see video number 26 from the course, and examples from the books also

3.10 Excec Family

Till now we have seen the `fork()` function, which duplicates a program, *but the heap and the stack remains the same*.

Another form of creating process is through the `exec` family of calls. The difference here is:

- In this type of function, the program will load and execute some new process, and hence it will *replace the previous contents of the address space, and begin execution the new program* (see chapter 5 in [1])

An example is shown in Figure 3.4.

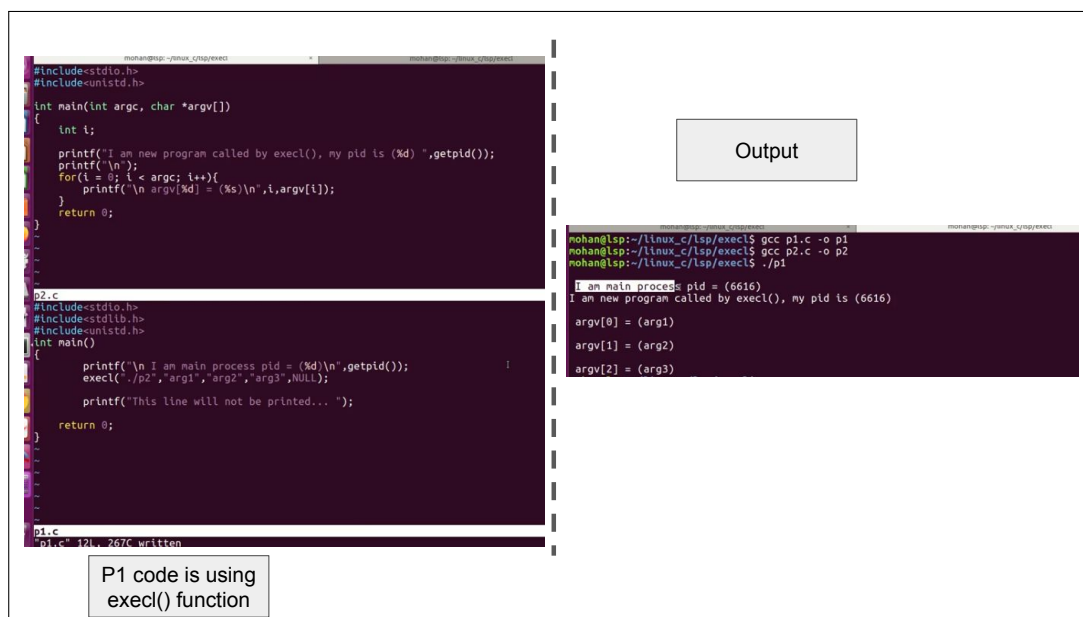


Figure 3.4: Excec() example

- `p1` is calling `p2` using `execl()`
- The arguments to `execl()` are the binary (compiled code)
- Note that is because we have a replacement of memory, the `printf()` for `p1.c` is never executed
- Even we are running `p2.c` from `p1.c`, but we have the same process id in both

3.11 Notes and summary

- About the memory layout: a nice example about text segment and how it is read only, the code `char* buff = "welcome"`.
 - `welcome` variable here is store in the text segment, so we can't change it
 - If we try to do `buff[0] = '\n'`, it will result in a segmentation fault

3.12 TODO for Processes

This section contains general ideas for some points to do later for processes.

- Write a better introduction later for the beginning of this chapter
- There is the concept of using `fork()` and `exec()` together in the same code
 - This is the in the video 44 of the course
 - But need to search better in the books for some practical aspect also, and not only theory
- To see also the idea of files handling in a `fork()` call
 - What happens to these resources before and after the `fork()`
 - For this I need to review the low level things about file handling (file as a byte, offsetting a file, ...)

Chapter 4

Signals

4.1 Intro

Section from course: number 11

- Signals are interrupts that provide mechanism to deal with some unexpected events
- The signals comes from the kernel side
- Every signal has a signal handler
 - The signal handler is a juste function

Also some points from my reading in [3] (chapter 4, section 6.2):

- Signals are ways to notify the process for some particular events
- 2 types of signals:
 1. synchronously: that is generated within the same process.
Example can be a program that have done a division by 0, or accessing some illegal memory side
 2. asynchronously: the signal comes from outside the process, such pressing `Ctrl C` to terminate a program (force it to terminate)
- Once the signal is generated (either synchronous or asynchronous), ***it must be handled by a handler.***
2 possible way to handle the signal:
 - A default signal handler
 - A user signal handler
- Default signal handler: this is handled by the kernel, and we have 2 possibilities:
 - can be ignored
 - catch and handle the signal \leftrightarrow
- If a signal doesn't have a handler, it has a ***default action to be taken***

4.2 Examples

Alarm signal

See example in `signal` project in `System Programming` workspace.

Alarm signal:

- *To redo the alarm signal example because I didn't understand very clearly how the flow of the code is*

Chapter 5

Virtual Memory

5.1 The big picture

The virtual memory concept allow the process to access more address space then the actual physical RAM.

Concept to be reviewed later:

- *Concept of an address space*
- *It's relation to the processes*

Memory Management: *I will redo this chapter later after the process chapter, and when the book arrive.*

Memory Ma
agement

Chapter 6

Threads

6.1 Introduction

intro to thread

intro to thre

- *To write later the intro about the thread*
- *Difference between thread and processes*
- *Read from books*

Some points from the video lectures:

- Different threads share the virtual memory with the process that created them
 - Unlike processes, which can't share memory among each other
- In Linux, the most famous thread interface is POSIX: abbreviation for portable operating system interfacance
- The advantage of having many threads is to have many threads is to divid a task into smaller one, and each handle by a thread.

6.2 Race Condition

In a high level, race conditions happen when a 2 threads try to access the same variable, and one of them has not finished yet.

Race Conditions points:

Race Condi-
tions points

- *To revisit later the video number 3 in CodeVolt playlist*
- *To see what later to other points we can add from other videos or resources*

6.2.1 Race Condition Question

Race Condition Questions:

Race Condi-
tion Questio

- Are threads happening in a parallel execution always, or they wait each other (overlapping manner) ?

6.2.2 Mutex

Mutex is a way to prevent race condition.

As a definition, a mutex is a lock about a section of code.

Some important points (taken from the CodVault playlist):

- The idea is if we have 2 threads, by implementing the lock, there is no way we have a thread is reading, and the other thread will be executing
 - In other words, the 1st thread will finish its work

6.3 General Points

- When we have 1 threads created, the thread and the main process share same process ID via the `getpid()`.
- When passing arguments to threads
 - this means we are passing input to the routine function
 - This is done via `pthread_create()`
 - Don't forget that we need to cast the `void*` pointer to the right type we are passing
- Whenever we create a thread, we need to wait this thread to be finish its work
 - This is done via `pthread_join()`
 - The same is true if we have multiple threads.
- If we need to ***return some result*** (so *an output*) from the thread
 - We do it via `pthread_join()`, using its 2nd argument

Chapter 7

Libraries in C/C++

7.1 Introduction

This chapter deal with external libraries in C/C++.

The libraries can be linked and loaded also via 2 options:

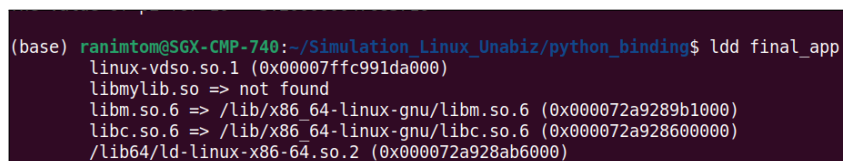
1. Static library and loaded statically
2. Dynamic library (or shared object in Linux `.so`), which linked dynmaically.

It is important to note that libraries are *precompiled files*, and we want to use them in our applications. The purpose of this precompilation is to save time and don't compile them each time we modify our code.

In other words, this means we compile only our source code, and the libraries remains intact.

7.2 Some Tools

- `ldd some_output` command used to print out all the shared object dependency related to our executable C code, named `some_output`.
- An example is shown in [Figure 7.1](#), where I genrate some executable named `final_app`



```
(base) ranimtom@SGX-CMP-740:~/Simulation Linux_Unabiz/python_binding$ ldd final_app
linux-vdso.so.1 (0x00007ffc991da000)
libmylib.so => not found
libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x000072a9289b1000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x000072a928600000)
/lib64/ld-linux-x86-64.so.2 (0x000072a928ab6000)
```

Figure 7.1: `ldd` command example

7.3 Static Libraries

- Static libraries are binary files that are compiled using a compiler like `gcc`
- They result in a files with `.o` extensions, which stands for object files

- The important point about these libraries that they are ***architecture dependent***, so if they run on my machine (Linux), doesn't mean necessarily that they run on other machine
- The final executable program will be big in size
 - We can use the command `ls -lh` to display the size of each file in Kbytes

7.3.1 Creating a static library

- Suppose we have 2 object files that we compile using `gcc`. They are `mymath.o` and `mymath2.o`
- To create the library, we need to bundle these 2 `.o` files. We use `ar` tool for this purpose.
- The command will be:
`ar rcs mymath.a mymath.o mymath2.o`, where:
 - `mymath.a` is the name of the library, with `.a` extension
 - `rsc` is for

7.3.2 Pros and Cons

- Static libraries are easy to deal with, since everything is bundled together, this simplifies distribution to other
- but the size on disk and at run time is big
- Also the static libraries are not cross compiled \leftrightarrow this means they can be shared by 1 program only

7.4 Shared Library

- shared library (or dynamic library) are another type, with `.so` extension for Linux (and `.dll` for window)

7.4.1 Creating a shared library

- `gcc -fPIC -shared .src/mymath.c -o libmymath.so`
 - This command instruct `gcc` to create a shared library (hence the flag `-shared`) named `libmymath.so`
 - `lib` prefix is necessary for any name we choose, because when compiling our code (see 7.4.2), the linker assumes always the library name starts with `lib`.
 - Recall that `-o` stands for output

7.4.2 Compiling C code with a shared library

- Now once we create the shared library, we need to compile our C code with this shared library
- As any library, we need to link `.so` to our code using the linker (so using `-l` flag)
- Now this step is a little tricky
 - Although our created shared library name is `libmymath.so`, in the command we use only `mymath`, where we omit both `lib` (prefix) and the `.so` extension
 - The linker expects these prefixes/postfixes so no need to add them in the command
- The command is:

```
gcc ./src/*.c -o prog -lmymath L./
```

 - `.L/` is to indicate the libraries are in the current directories

7.4.3 Running the program

We use the following command using the environment variable:

- `LD_LIBRARY_PATH="./" ./final_app`

Bibliography

- [1] R. Love, *Linux System Programming*. O'Reilly Media, Incorporated.
- [2] A. Tanenbaum and H. Bos, *Modern Operating Systems, Global Edition*. Pearson Education, 2015.
- [3] A. Silberschatz, P. Galvin, and G. Gagne, *Operating System Concepts, 10e Abridged Print Companion*. Wiley, 2018.