Contents

Todo list

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 funct

kernel functions and concept

- *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 `prinf()` which in turns call `write()`

1.2.3 Internal mechanism for system call

- The functional block diagram is shown in Figure 1.1.

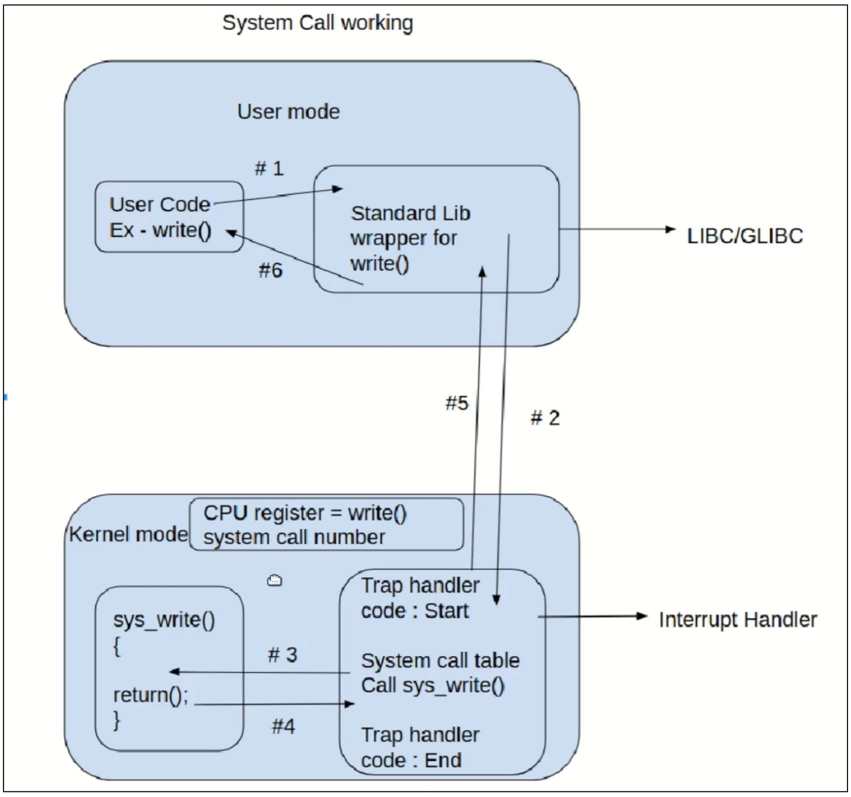


Figure 1.1: Main Components

- Every system call has a unique number associated with it
- When we use `write()` function in our code, it will no 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

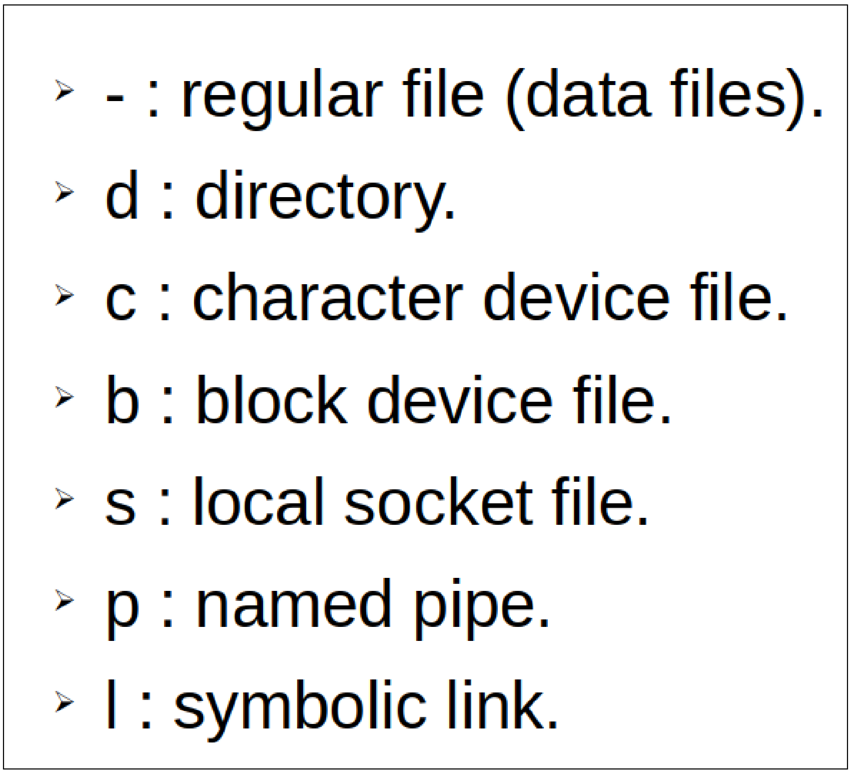
Now we move for file manipulation in system programming.
Files are important in linux system programming, because everything is a treated as file in linux (chapter in 1 in [1]).

Under the hood info:

- Once we open a file, a metadata is attached to this open state is called file descriptor, known as `ftd`, 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 defintions 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,$\cdots$.

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

ualizing
cesses

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)

cess Intro

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=PLBlnK6fEyqRgKl0MbI6kbI5ffNt7BF
  - Neso Academy for operating systems
  - Can serve as theory and complementary explanation for my books reading
- https://www.youtube.com/watch?v=cex9XrZCU14&list=PLfqABt5AS4FkW5mOn2Tn9ZZLLDwA3k
  - 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.

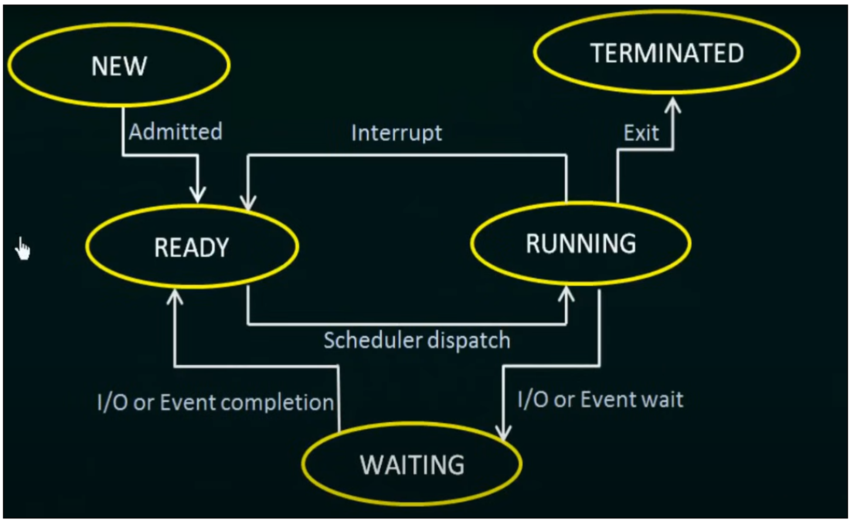


Figure 3.1: Different state of a process

- 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 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 3.2, 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

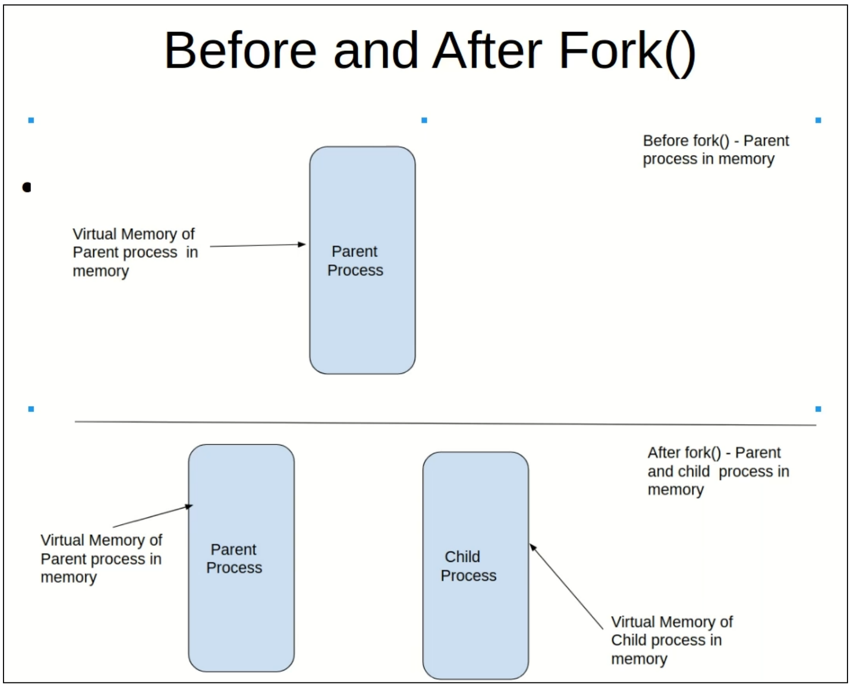

Before and After Fork()
Before fork() - Parent process in memory
Virtual Memory of Parent process in memory
Parent Process
After fork() - Parent and child process in memory
Virtual Memory of Parent process in memory
Parent Process
Child Process
Virtual Memory of Child process in memory

Figure 3.2: fork system call: before and after

3.5.1 Copy on- write

We can see that in Figure 3.2 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 note 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 chlid) 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].

3.6 Waiting a process

- 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

eo

Video course: video 38, at 20:17.

3.7 Memory Layout

Now we discuss memory layout for the process.
Memory a of 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* ↔ 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

3.7.1 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*

3.8 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

Chapter 4

Virtual Memory

4.1 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

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.