Grégor JOUET | gregor.jouet@devinci.fr

1 Requirements

1. You must have a working Ubuntu (or other Linux based) system installed **PRIOR** to this class. The system must be **BARE METAL** (no VM) and **OPERATIONAL**. Failure to do so will prevent you from doing this assignment.
2. It is recommended, although not required, to work on this assignment in advance and to use the actual assignment day to ask questions and engage in discussions on the topic.

> Is there anything that we associate more closely with intelligence than curiosity? Every intelligent species on Earth is attracted by the unknown. Our mythologies are full of riddles and mysteries and divine knowledge. Even the word apocalypse... Even the word apocalypse means revelation. It seems like our ancestors always imagined that even at the very end we would solve one last mystery.
>
> (Alexandra Drennan)

Contents

2 Foreword

In the *Operating System & Administration* lesson, you will discover the GNU/Linux based operating system Ubuntu. You will learn and understand the characteristics of these operating systems and the standard way they operate. Future lessons will heavily build on this class to use various tools, it is primordial that you understand thoroughly the content of this class. Should a question arise out of the class, I strongly advise you to send an email or talk with me in person regarding the misunderstanding.
The Operating Systems are a fundamental building block of computer science and in this day and age, of society as a whole. Their primary objective is to create a bridge between the applications and the hardware used. The fundamental concept of A.P.I [1] is essential to operating systems. They can take various forms and be used in a lot of ways. Although there are many operating systems, some concepts are fundamental and present across all existing O.S. This workshop focus on these concepts as well as practical considerations on how to operate a GNU/Linux[1] system.
Throughout this workshop and after, use the kernel archive [10], the man pages [7] and the Modern Operating System [9] book to get information on particular APIs, or functions.

3 Processes

Processes are the cornerstone of the operating systems. They are a way to separate resources between users and applications and provide a layer of isolation from the system used for security and interoperability.

3.1 Running a process

Running a process is a common task to do on a machine. Effectively, every time you run an executable, at least one process is launched. One of the OS purpose is to manage the process, attribute them memory and computing power. The process lifecycle diagram below show how the state of a process evolves in time. *ready* state means the process is paused and waiting for a processor to resume its execution, the *running* state means the process is currently being executed.

Figure 1: A Process lifecycle

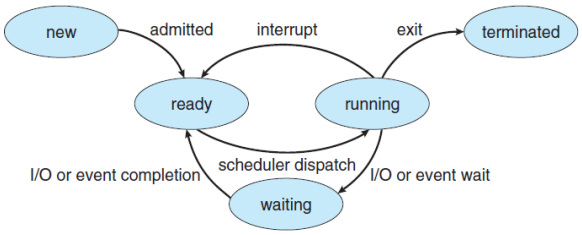

Figure 1 show the lifecycle of a process and its different states. Each process has its own identifier, a **PID**[2] which is an integer.

1. Find the **PID** of the bash instance you are using.
2. Print all the running processes on your system using the **ps** command.
3. What is the *TIME* column in the **ps aux** command output ?

For each process to be independent and not interfering with one another, their memories are kept separated. Each process only sees its own personal memory space.

1. What are the downsides of having complete memory isolation ? How do you think this is implemented in practice when multiple processes require the same library ?
2. What part of an OS makes sure that a program cannot interfere with another's program memory ?
3. Regarding compilation, how is memory isolation more practical ?

3.2 Processes Environment

When executing a program, a process also has configuration elements that are independent of the program being executed. These invariants are often used to pass configuration parameters to a program.

3.2.1 Environment Variable

Env. Vars. are the most direct way of doing program configuration for a process. They can be used to pass config file path, passwords or tokens, URL etc... They are widely used because they are accessible in any process written in any language.

1. List environment variables in bash
2. Add/ modify env variable in your bash with two different methods
3. Run a process with an environment variable *TEST* set to 'hello'
4. Write or find a program to print this variable to the console
5. What is the *PWD* variable used for?
6. What is the *PATH* variable used for?

[1] Although GNU/Linux is to be used to refer to the OS, I use the term *linux* out of simplicity and style in this document. I will let you research what is the difference between these two terms
[2] Process Identifier

3.2.2 Exit code

Exit codes are integers given by a program when exiting. They are used to indicate the exit status of a program. By convention an exit code above 0 indicates there was an error.

1. Find a way to print the exit code of a program in your terminal
2. Open a python shell and make it exit with the exit code 42
3. Run different commands with invalid parameters, what are the exit codes?
4. In bash, make a program that writes somethig only if a program has a 0 exit code
5. In the same fashion, write a program that writes something if a program has an exit code ¿ 0.

3.2.3 Users

To manage access to process resources and files, each process is run as a user. This can be observed in htop on the *USER* column.

1. Print the current user in your terminal
2. Write a bash script that prints the current user and the user id[3]
3. Execute the script as root. What is its user id?

We will see in Section 4 how the user and interact with the filesystem.

3.2.4 System Limits

Because the computer is not a Turing Machine [11], which has an infinite capacity, it has limits. Limits are set at the kernel level and can be seen and changed with the **limit** and **sysctl** commands.

1. What are the different limits?
2. What does the maxproc limit refers to?
3. Increase the max number of process on your machine
4. Find an example where increasing these limits is required

3.3 Signals

Signals are an essential part of an OS. They help manage the state of a process and are a well implemented interface for basic tasks and error handling. Process can receive and emit signals, they are then processed by *signal handlers* in the programs.

Figure 2: The different signals available

Signal	Value	Action	Comment
SIGHUP	1	Term	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGABRT	6	Core	Abort signal from abort(3)
SIGFPE	8	Core	Floating point exception
SIGKILL	9	Term	Kill signal
SIGSEGV	11	Core	Invalid memory reference
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers
SIGALRM	14	Term	Timer signal from alarm(2)
SIGTERM	15	Term	Termination signal
SIGUSR1	30,10,16	Term	User-defined signal 1
SIGUSR2	31,12,17	Term	User-defined signal 2
SIGCHLD	20,17,18	Ign	Child stopped or terminated
SIGCONT	19,18,25	Cont	Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process
SIGTSTP	18,20,24	Stop	Stop typed at terminal
SIGTTIN	21,21,26	Stop	Terminal input for background process
SIGTTOU	22,22,27	Stop	Terminal output for background process

The signals **SIGKILL** and **SIGSTOP** cannot be caught, blocked, or ignored.

Figure 2 show the different signals that can be sent to a process as well as a short description. You can find such info in the linux doc [10].

1. What does the **kill** command do?
2. Open a python console and find its pid, from within the console
3. Send a SIGTERM signal to a python console you have running
4. Whats the differece between a SIGKILL and a SIGTERM?
5. What is a SIGINT?
6. What is a SIGSEGV?

3.4 Conclusion

With these basic exercices you can now naviagte the terminal and the system and see how resources are divided amongst processes. There are several subjects that are left to your curiosity, namely *Capabilities* and *Process Scheduling techniques*

4 Files

A fundamental concept of Operating Systems are files. We will see how they operate in linux, how to manage them and their permissions. Regarding linux, the core concept behind its architecture is to consider everthing to be a file. We will see the implications of this desingn choice in this section and will use it to build on going forward.

[3]The user id is often just called UID, but different kinds exist, check the documentation [10]

4.1 File Management

In the linux filesystem, files are ordered hierarchically in the form of a tree. The base folder is the *root* of the filesystem, refered to as '/' and each folder is linked to this root. Unlike windows there is not C:\to refer to a specific drive, rather each drives can be placed in the filesystem, we will see how in the section 4.5. Because you will be using the filesystem daily, a good understanding of it is necessary before going forward.

4.2 Introduction

Let's just start by looking around and searching the use of different elements

1. What are the diffent folders immediatly at the root?
2. What can be found in the */dev* directory? And in */bin* and *proc*?
3. What is the size of /proc?

4.3 Permissions

Files permissions are an essential element of resource segmentation, files on disks are supported by a filesystem [6], the kernel provides a general api to access files and manages access list to those files are directories. Permissions are represented by an octal number and are of 3 different kinds, read permission (r), write permission (w) and execution permission (x). This can be seen in the *ls -ls* command

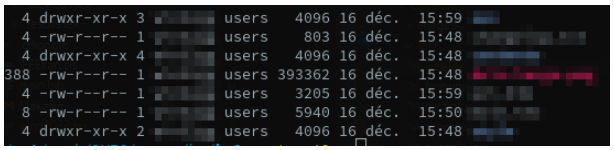

Figure 3: File infos given by *ls -ls*

The figure 3 show the different permissions on different files and directories.

1. What is the *r–r–rw–* kind of string in the output of the *ls -ls* command? What information does it carry?
2. What is the **chmod** command used for? What does the 777 octal code corresponds to?[4]
3. What is the **chown** command used for?
4. Create a directory that you do not own with multiple files iundide. What is the required permission to wrinte in them? What is the required permission to delete a file in that directory?
5. What does an *execution* permission do on a folder?
6. What can be a use case of groups? Make the previous directory belong to the *users* gorup.

[4]Use https://chmod-calculator.com/ to help you.

4.4 Virtual Filesystem architecture

The Virtual Filesystem[5] architecture is normalized accross linux systems to maintain stability in applications. The reference is the Filesystem Hierarchy Standard [5].

1. Go to the root of the VFS and explain what each folder is for.
2. What is the difference between */bin* and */sbin*
3. Why is there not *Program Files* folder in Linux? Explain the different paradigms
4. Using the **file** command look at the different files: */boot/vmlinuz* and */bin/bash*
5. Using the **ln** command create a sybolic link to */dev* in you home directory

4.4.1 About /proc

Some complementary questions on the */proc* filesystem.

1. What are the folder with only a number in the */proc* directory?
2. What is the */proc/cmdline* file?
3. What is the */proc/interrups* file?
4. What is the */proc/meminfo* file?
5. What is the */proc/sys/net/ipv6/conf/all/disable_ipv6* file?
6. What would happen if you wrote a 1 in it?
7. Where would that 1 be stored? On disk?

4.5 Mount points

Mount points are an ambiguous topic and is difficult to understand at first, but is essential to any running system. Mountpoints are folders where filesystems are *mounted* and can be accessed. The different mountpoints on your system can be seen with the **mount** command.

1. Using the mount command, inspect the different mount points on your system. Find the mount point for */dev* and */proc*.
2. Look for the mount point of the root of the VFS. How do you explain this mount point?
3. Using the **mount** command, mount a dev filesystem in your home folder. What is the difference with a symbolic link as seen earlier?
4. Mount your root filesystem a second time in a test folder in your home directory

[5]Often shorted to VFS

5 Process Memory Management

5.1 Memory Mapping

Before we can actually use the memory of a process with C, we must see how the memory used by a process is split up. Memory is the basing element that all process deal with and is fundamental to all program. To explore a process memory mapping we will use x86 assembly as it is the most widely used.

5.2 Hardware and Memory

```
;to build an executable:
; $ nasm -f elf64 -o hello.o hello.s
; $ ld -o hello hello.o
; $ ./hello
global _start

section .text

_start:
  mov rax, 1 ; write(
  mov rdi, 1 ; STDOUT_FILENO,
  mov rsi, msg ; "Hello, world!\n",
  mov rdx, msglen ; sizeof("Hello, world!\n")
  syscall ; );

  mov rax, 60 ; exit(
  mov rdi, 0 ; EXIT_SUCCESS
  syscall ; );

section .rodata
  msg: db "Hello, world!", 10
  msglen: equ $ - msg
```

Figure 4: Hello World in x86 Assembly

The memory management of the computer is closely tied to the way the processer works. It can run like a normal chip with its own memory, which is called *raw* mode or it can be used to separate process in different memory sections, which is the *protected* mode [14]. In protected mode the processor does the isolation per process and executes instructions, like the ones in the Figure 4[6]. The different instructions you see (mov, int) are what the processor executes. The *int* instruction hints at Part 6 on the syscalls. The *mov* instruction is used to move data accross memory [2, 13]

1. What does the *mov* instruction do?
2. What is the x86 language and what is it used for?
3. Run the code in Figure 4 on your computer. What happens?
4. What are the rax,rdi,rsi,rdx references? What are they used for?
5. Why is there no function declaration in this code?

[6]You can find the original code at https://jameshfisher.com/2018/03/10/linux-assembly-hello-world/

5.3 Process Memory Layout

Each process has its own personal memory, which is tied to the system memory via a *paging* system. Process memory is called *virtual memory* while real memory is called *physical memory*. The different sections of a process memory can be accessed via the */proc* interface. Figure 5 shows the way a process ahs a contiguous memory space that is tied to the physical memory. The kernel handles this mechanism.

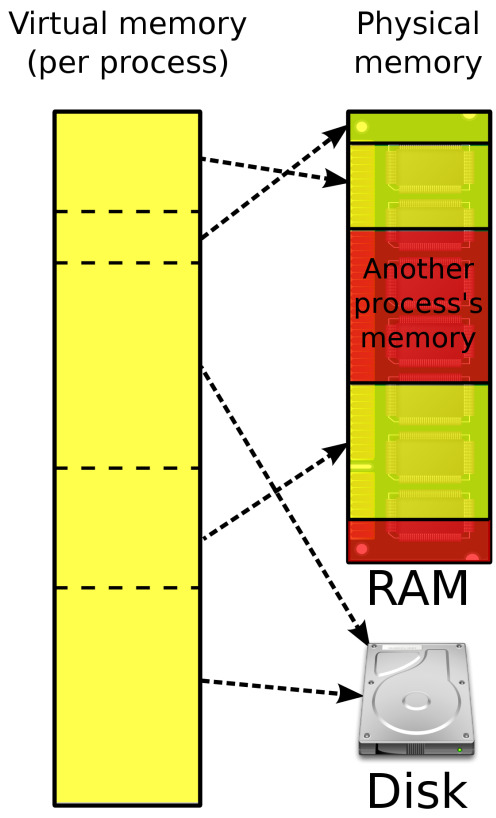

Figure 5: Process Virtual memory

1. Start a python interpreter and get its PID
2. Look at the file */proc/PID/maps* what does this file contains?
3. Look for the *[heap]* memory mapping, what is it?
4. Look for the *[stack]* memory mapping, what is it?
5. What are the other mappings you see?
6. (to go further) Look for the *[vdso]* memory mapping, what is it? [3]

5.4 Dynamic Linking

We just saw that a process had different memory regions called mappings, some of these regions are files that are directly mapped in memory. Before we use the concept of *Dynamic Linking* in the next lesson, we wiil explore the ways dynamic mlinking works on executable and introduce a few tools. Dynamic Linking is th emost common way of linking executables on modern systems and is omnipresent is every software used. [4]

1. Run the **file** on */bin/bash* what kind of file is it? What is ELF? [12]
2. Use the **readelf** on */bin/bash* explain the *ABI* field.
3. What does the **ldd** command do? Run this command on your python3 executable
4. What do you notice about the outputs of **ldd** on python3 and bash?
5. What do you notice about the output of **ldd** and the */proc/PID/maps* file?

5.5 Going Further

1. What does the *LD_PRELOAD* environment variable do?
2. What does the *LD_LIBRARY_PATH* environment variable do?
3. Clone and build the *stderred* repo: `https://github.com/sickill/stderred`
4. Use *LD_PRELOAD* to start a python3 instance with this library.
5. What do you notice? Did the executable change?

6 System Calls

System calls are a way for a process to request the kernel to do something or to provide information. Giving direct access to hardware to a process would go against the isolation we want and could put the system at risk. The kernel is the part of the system that handles communication with the hardware and thus, process need a way to communicate with it.

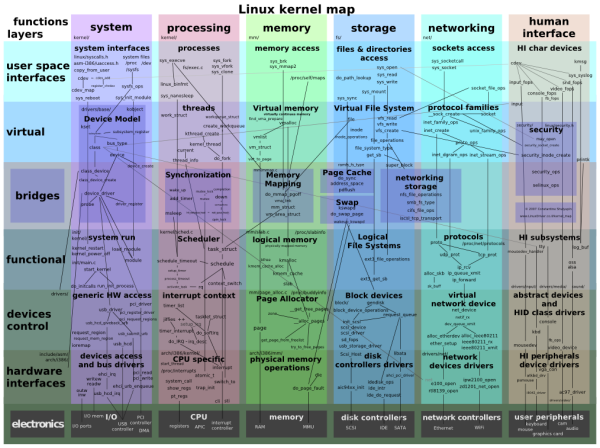

Figure 6: Linux Kernel stack

Because the kernel stack is coplicated and can interact with a lot of different hardware as you can see on figure 6, it is important to have a central access point for the hardware. Again the notion of API by which we mask the complexity of the kernel and expose only a stable and consistent set of functions.

6.1 An abstraction of the kernel

Practically speaking, syscall are functions that can be called just like any other. They are doccumented in the section 2 of the linux manual [7]. At compilation however, the syscalls are called through an interrupt [7] as you saw in the Figure 4.

6.2 Seeing the Syscalls

Making a syscall wil be done in the C class, for now let's see some tools to explore Syscalls.

1. Make sure the **strace** command is installed
2. Run the **strace** command with the executable you built earlier in Figure 4
3. Explain the output of the command.
4. (hard) Run the **strace** command on *ls* and explain the differences in outputs.
5. What does the *stat* system call do?

6.2.1 File Descriptors

Files descriptors are integers representing an open stream on which a program can read or write. The backend of file descriptors can be files, network connections, local memory etc...

1. What are file descriptors 0, 1 and 2?
2. On which file descriptor do you write when you use a *print* in python?
3. Visualise the file descriptor used by the **cat** command using **strace**

[7] 0x80 in older systems and with *syscall* in more recent architectures

7 To Go further

A collection of harder problems to go further. You may want to use the different exercices seen before to buidl up to this point. The exercices are designed to to be more difficult.

7.1 Loop Devices and filesystems

1. Use the **fallocate** command to create an empty file of 500Mb.
2. Use the **fdisk** command to create a partition on this file. On which file is this command usually used?
3. Use the **losetup** command and loopback mount points to mount this file as a drive.
4. Create a *ext4* filesystem on this partition and mount it in na folder.
5. Unmount all and use the **cryptsetup** command to encrypt the filesystem.

7.2 Cron Job

1. What is a *CronJob*?
2. Modify the */etc/crontab* file to execute a purge of the system logs every three months
3. Modify the crontab file to execute a program on restart

7.3 Systemd Services

1. What are systemd services? [8]
2. Show the status of your system using the **systemctl** command
3. Restart your NetworkManager using **systemd**
4. Create a new service and put it in the appropriate folder
5. Make this service run on startup

7.4 Special Files

7.4.1 Sticky Bit

1. What do you notice about the permissions of the file */bin/passwd*?
2. This permission is the Sticky Bit[8]
3. What could be a potential problem if you could add a sticky bit to a program belonging to another user?
4. Give another example of a program with sticky bit set
5. (harder) Test out this feature by copying localy the *id* command and and adding a sticky bit to it.

7.4.2 FIFO

1. What is the command **mkfifo** used for?
2. Create a file using this command
3. What do you notice about this file? Use the **file** command on it
4. Write a read at the same time from this file. Explain this behaviour.
5. Where are the contents written to this file stored?

7.4.3 Tmpfs

1. Use the mount command to mount a filesystem of type *tmpfs* on a directory.
2. Banchmark the write speed in this directory and compare it to a fs on disk.
3. Where is this mountpoint connected?
4. Inspect the */run* mountpoint. What kind of filesystem is it?

[8] See https://www.redhat.com/sysadmin/suid-sgid-sticky-bit

References

[1] *API*. en. Page Version ID: 1034884558. July 2021. URL: https://en.wikipedia.org/w/index.php?title=API&oldid=1034884558 (visited on 07/23/2021).

[2] Bisqwit. *x86 mov insns & short history of the most popular CPU architecture*. URL: https://www.youtube.com/watch?v=g9_FYRAfyqQ.

[3] Jonathan Corbet. "On vsyscalls and the vDSO". In: (2011). URL: https://lwn.net/Articles/446528/.

[4] *Dynamic Linking*. URL: https://www.qnx.com/developers/docs/6.5.0SP1.update/com.qnx.doc.neutrino_sys_arch/dll.html.

[5] *Filesystem Hierarchy Standard*. URL: https://refspecs.linuxfoundation.org/fhs.shtml.

[6] *Linux Filesystems*. URL: https://dri.freedesktop.org/docs/drm/filesystems/index.html.

[7] *Linux man pages*. URL: https://linux.die.net/man/.

[8] *Systemd Services*. URL: https://www.freedesktop.org/software/systemd/man/systemd.service.html.

[9] Andrew S. Tanenbaum. *Modern operating systems*. en. Fourth edition. Boston: Pearson, 2015. ISBN: 978-0-13-359162-0. URL: http://index-of.es/Varios-2/Modern%20Operating%20Systems%204th%20Edition.pdf.

[10] *The Linux Kernel Archives*. URL: https://www.kernel.org/.

[11] A. M. Turing. "Computing Machinery and Intelligence". In: *Mind* 59.236 (1950), pp. 433–460. ISSN: 00264423, 14602113. URL: http://www.jstor.org/stable/2251299.

[12] *Understanding the ELF File Format*. URL: https://linuxhint.com/understanding_elf_file_format/.

[13] *x86 Assembly Guide*. University of Virginia Computer Science. URL: https://www.cs.virginia.edu/~evans/cs216/guides/x86.html.

[14] *x86 Protected Mode*. URL: https://en.wikibooks.org/wiki/X86_Assembly/Protected_Mode.