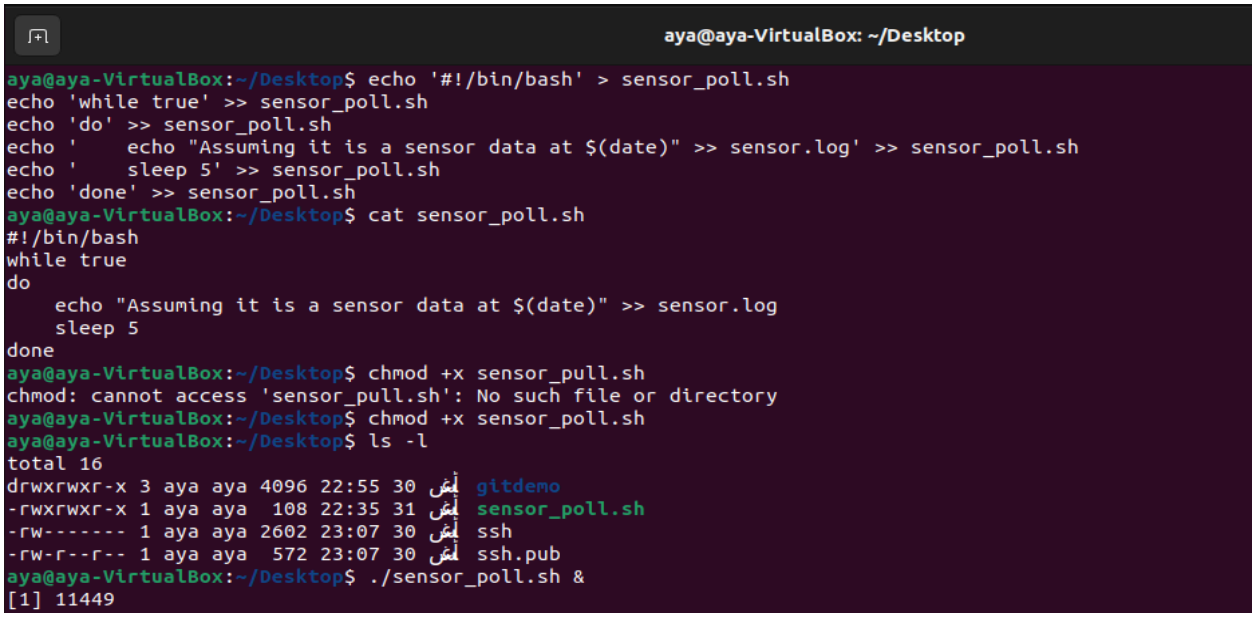
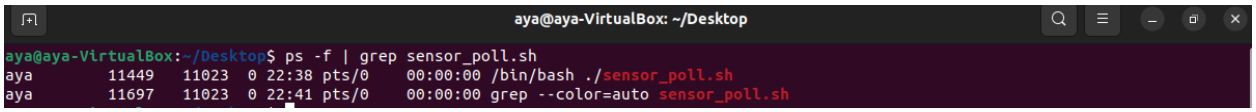
Day 4 – Phase 4: Process and Network Monitoring

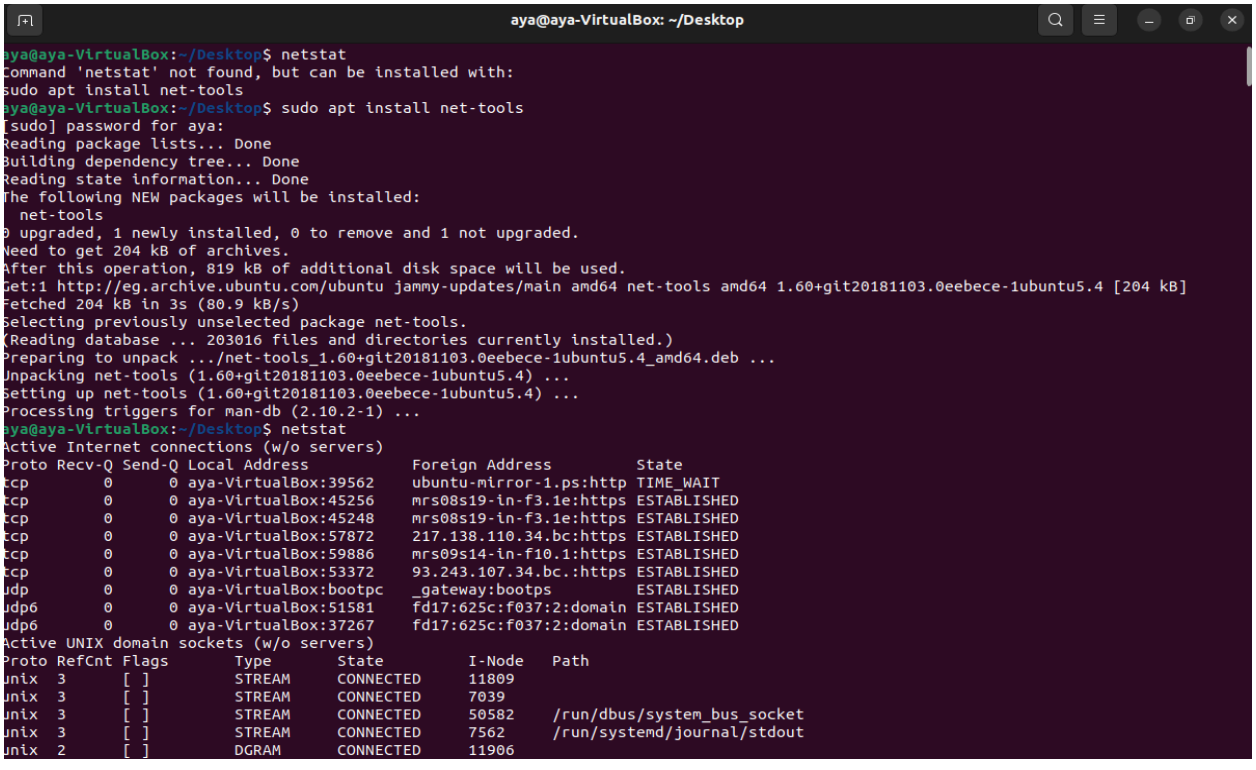
1) I have the “x” letter not working on the laptop, that's why I did not use the nano command; however, I used the echo one.



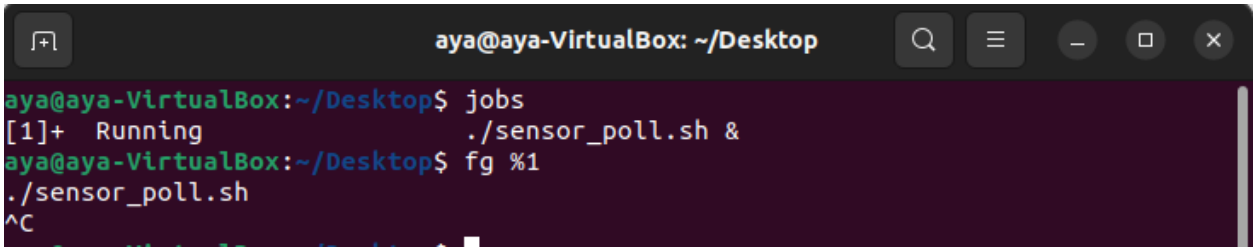
2)



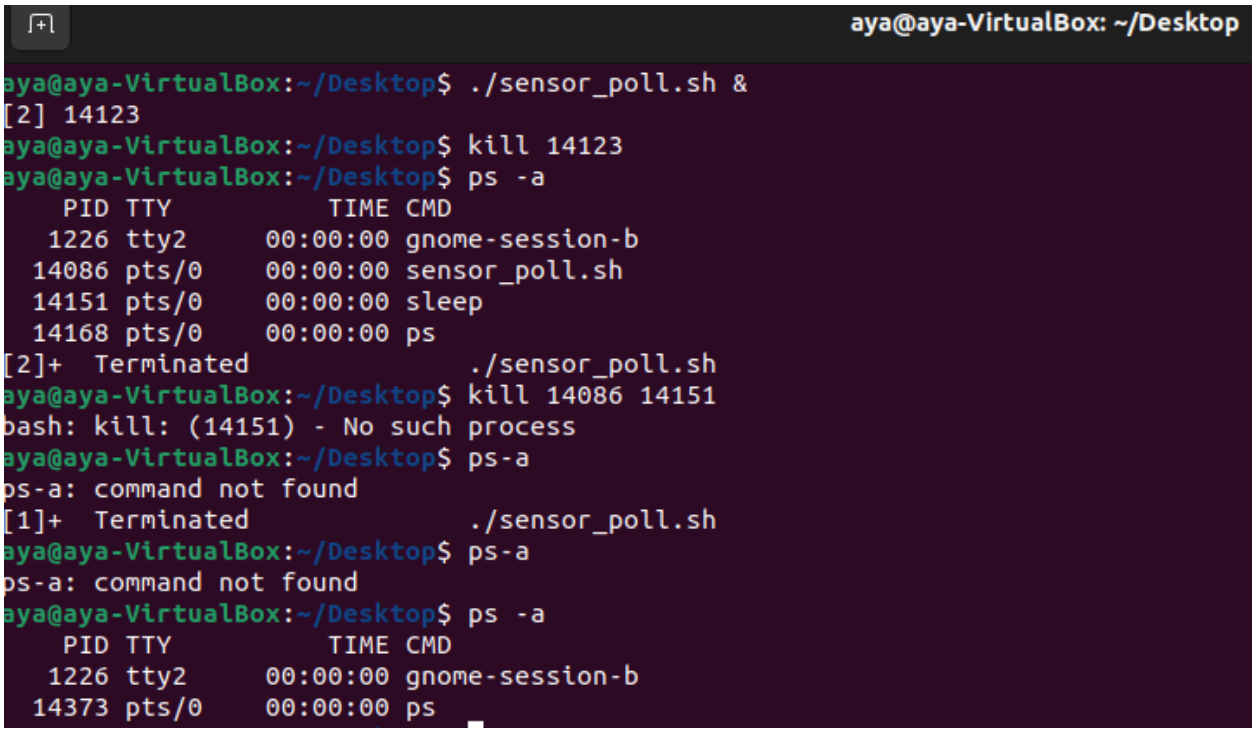
3)



4)


aya@aya-VirtualBox: ~/Desktop
aya@aya-VirtualBox:~/Desktop$ jobs
[1]+ Running   ./sensor_poll.sh &
aya@aya-VirtualBox:~/Desktop$ fg %1
./sensor_poll.sh
^C

5)

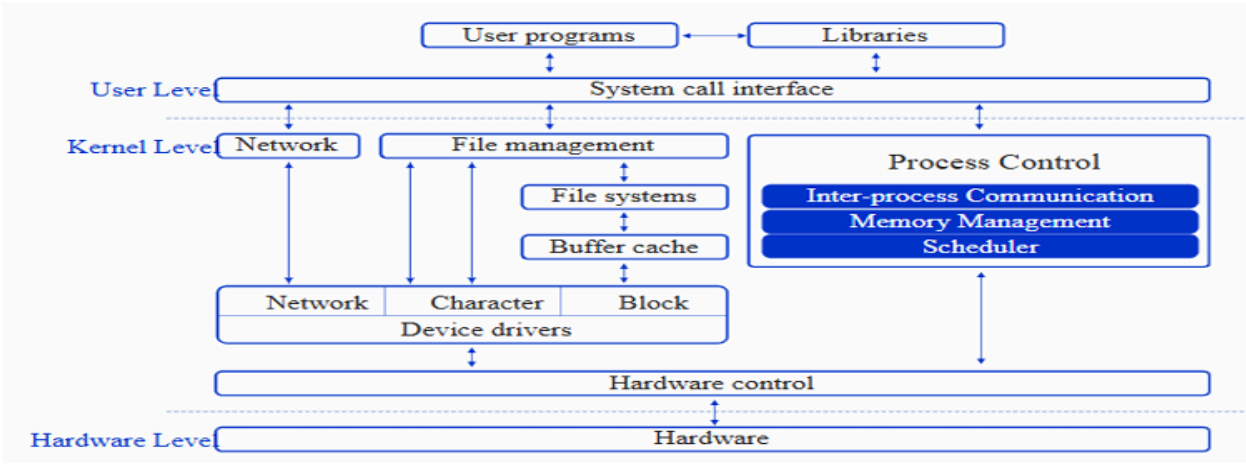

aya@aya-VirtualBox: ~/Desktop
aya@aya-VirtualBox:~/Desktop$ ./sensor_poll.sh &
[2] 14123
aya@aya-VirtualBox:~/Desktop$ kill 14123
aya@aya-VirtualBox:~/Desktop$ ps -a
PID TTY TIME CMD
1226 tty2 00:00:00 gnome-session-b
14086 pts/0 00:00:00 sensor_poll.sh
14151 pts/0 00:00:00 sleep
14168 pts/0 00:00:00 ps
[2]+ Terminated ./sensor_poll.sh
aya@aya-VirtualBox:~/Desktop$ kill 14086 14151
bash: kill: (14151) - No such process
aya@aya-VirtualBox:~/Desktop$ ps-a
ps-a: command not found
[1]+ Terminated ./sensor_poll.sh
aya@aya-VirtualBox:~/Desktop$ ps-a
ps-a: command not found
aya@aya-VirtualBox:~/Desktop$ ps -a
PID TTY TIME CMD
1226 tty2 00:00:00 gnome-session-b
14373 pts/0 00:00:00 ps

Open-Ended Questions

1) If we take the linux architecture as a reference, firstly for the user level (Command Entry), I write`ls` in the bash terminal which receives the , and input parses the command to identify its name and any arguments. Then, the shell processing where the bash checks if `ls` is a built-in command (it's not) and it searches for the `ls` executable in directories listed in the `$PATH` environment variable. For the System Call Interface Transition, the bash makes system calls to create a new process: `fork()` - creates a child process, -`execve()` - replaces the child process with the `ls` program in which those transitions (from User Level to Kernel Level) is done through the system calls (c functions). Regarding the Kernel Level (Process Manager), it creates a new process control block (PCB), allocates process ID (PID), and sets up memory space for the new process "process control subsystem". Then, the scheduler decides when the `ls` process gets CPU time, and the memory management allocates virtual memory pages for the process. Afterwards, we will be in the "File System Operations" layer in which the `ls` program needs to read directory contents, makes system calls like `opendir()`, `readdir()`, `stat()` so that the file management subsystem processes these calls and the file systems layer accesses the actual directory data. The Hardware Interaction, the communication between the kernel layer and the hardware is done through the usage of the application binary interface (ABI). the Output Generation, the `ls` formats the directory listing in user space, the program writes output using `write()` system calls to stdout with the Character Device Drivers handle terminal output and finally, the Text flows through the terminal emulator back to the screen. Eventually, there is Process Cleanup in which when the`ls` process completes and calls `exit(), the Process Control cleans up process resources, Memory Management frees allocated memory, Control returns to the parent bash process, and the Bash displays the command prompt again.

a) The entire process involves data flowing up and down through all the architecture layers, from the user input at the user level, through system calls to kernel space, potentially down to hardware for disk access, and back up through the stack to display results on the screen.

b)



2) The demeon: is a long-running background process that runs without the need for a terminal to show its output on such as systemd (system daemon “init”). It can be detected by “ps -ef | grep sshd”.

a) The zombie: it is when the parent does not care about the removal of the child resources after it dies (finishes its work) as if there is a child process running and finished so the resources do exist as the parent does not delete them until the kernel or the systemd (big parent) delete those resources. It can be detected by “ps aux | grep Z”.

b) The orphan: is a process running where the parent dies while the child process is still running. It can be detected by “ps -ef” to check the processes with PID=1.

3) In Linux, processes usually run independently; however, in many situations, they need to share data, coordinate tasks, or signal events and that’s why we need IPC. Without IPC, each process would be isolated, making it impossible to build modern systems where multiple processes must work together.

Main reasons we need IPC:

a) Data sharing: processes need to exchange information.

b) Synchronization: processes must not conflict (e.g., two processes writing to the same file).

c) Event notification: one process may need to notify another (e.g., server notifies client).

d) Resource sharing: coordinate access to shared resources (e.g., database, hardware).

e) Modularity: break a big program into multiple processes that communicate efficiently.

f) IPC Mechanisms in Linux: pipes which is a one-way communication channel between processes (example: ls | grep txt (the output of ls goes through a pipe to grep)).