**Operating System Lab#1**

**Objective:**

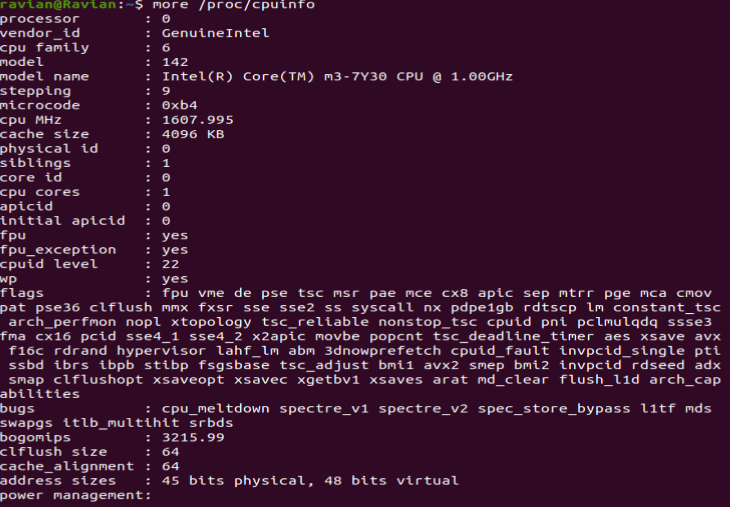
Understanding the basic functions of Linux commands by running the basic and simple commands on Linux shell and by observing the outputs of these basic commands.

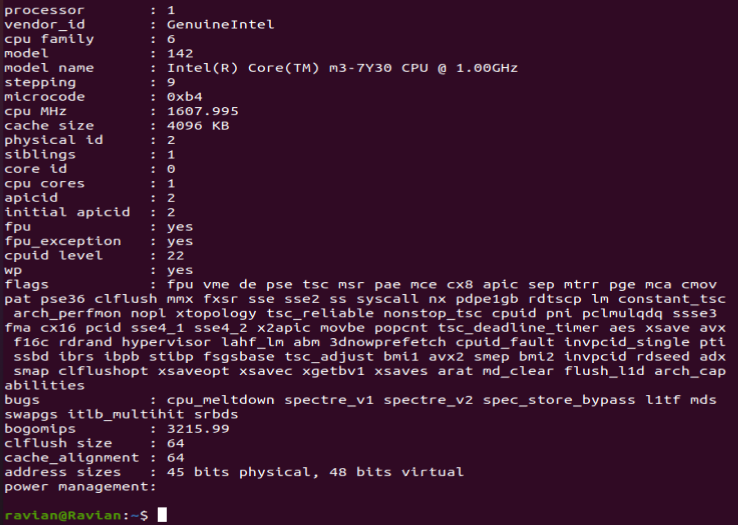
**Task-1:**

In this task, we will understand the hardware configuration of your working machine using the /proc filesystem.

1. Run command more /proc/cpuinfo and explain the following terms: processor and cores

**ANS:** Processor is a unit that takes the instructions to perform some specific operations and core is a unit in processor that execute the operations.





1. How many cores does your machine have?

**ANS:** By seeing this my machine have **two cores** one for each processor.

1. How many Processors does your machine have?

**ANS:** By seeing this my machine have **two processors** which is shown by

“0” and “1”.

1. What is the frequency of each processor?

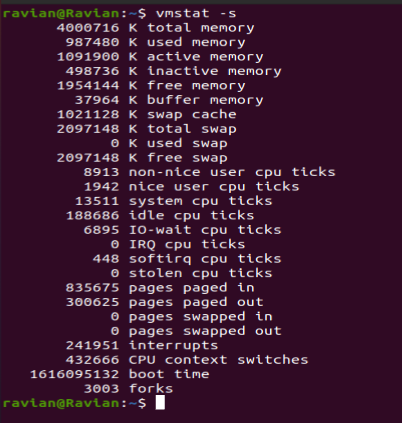
**ANS:** The frequency of my processor is **1.6GHz**.

1. How much physical memory does your system have?

**ANS:** Below figure show that the total memory of my system is **4GB**.

1. How much of this memory is free ?

**ANS:** The free memory from **4GB is 1.95GB**.



1. What is total number of number of forks since the boot in the system ?

**ANS:** The total number of forks is **3003**.

1. How many context switches has the system performed since bootup ?

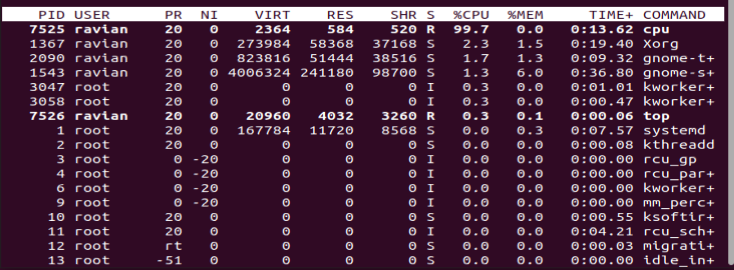
**ANS:** Total context switches that system performed is **432666**.

**Task-2:**

In this task, we will understand how to monitor the status of a running process using the top command. Compile the program cpu.c given to you and execute it in the bash or any other shell of your choice as follows.  
$ gcc cpu.c -o cpu  
$ ./cpu  
This program runs in an infinite loop without terminating. Now open another terminal, run the top command and answer the following questions about the cpu process.



1. What is the PID of the process running the cpu command?



**ANS:** The PID of the process running the CPU command is **7525**.

1. How much CPU and memory does this process consume?

**ANS:** That process consume **99.7% CPU** and **0.0% memory**.

1. What is the current state of the process? For example, is it running or in a blocked state or a zombie state?

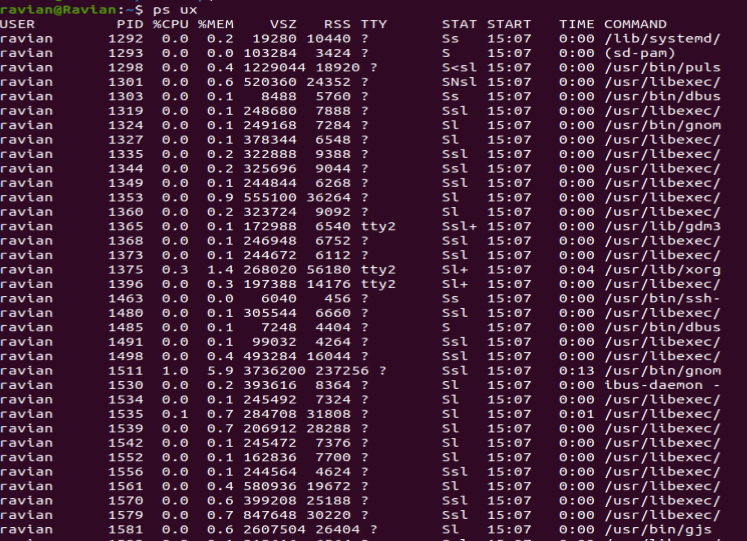
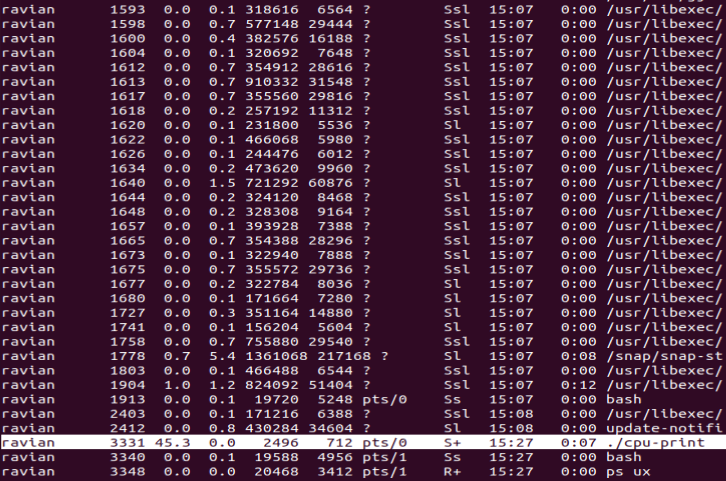
**ANS:** In figure **“S”** is tell us about status so it give **“R”** mean the current state of the process is **running**.

**Task-3:**

In this question, we will understand how the Linux shell (e.g., the bash shell) runs user commands by spawning new child processes to execute the various commands.

1. Compile the program cpu-print.c given to you and execute it in the bash or any other shell of your choice.

This program runs in an infinite loop printing output to the screen. Now, open another terminal and use the ps command with suitable options to find out the pid of the process spawned by the shell to run the cpu-print executable. You may want to explore the ps command thoroughly to understand the various output fields it shows.



**ANS:** The **PID** of the process spawned by shell to run cpu-print is **3331**.

The **ps** command give us the username who run the process, the PID of the process, its **%CPU and %MEM information**, time at which the **./cpu-print** process is started, provide status of the process, tell us how much memory that process have for its execution **(VSZ)** and how much that process currently using memory **(RSS)** and tell about terminal that executed the command **(TTY)**.

1. Find the PID of the parent of the cpu-print process, i.e., the shell process. Next, find the PIDs of all the ancestors, going back at least 5 generations (or until you reach the init process).

**ANS:** The PID of the ./cpu-print parent process is **1913** the bash command.

The ancestor PID’s going back in 5 generations are:

**1913 -> 1604 -> 1463 -> 1292 -> 1(init)**

1. We will now understand how the shell performs output redirection. Run the following command.  
   ./cpu-print > /tmp/tmp.txt  
   Look at the proc file system information of the newly spawned process. Pay particular attention to where its file descriptors 0, 1, and 2 (standard input, output, and error) are pointing to. Using this information, can you describe how I/O redirection is being implemented by the shell?

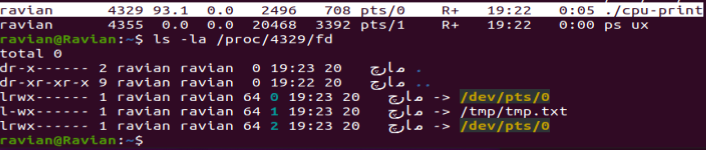


**ANS:** By running the command we find the PID of the process which is **3707**. After that, by using the **lsof -p 3707** command we see their file descriptors (FD). It shown the description of redirection as:

Standard input: <

Standard output: >

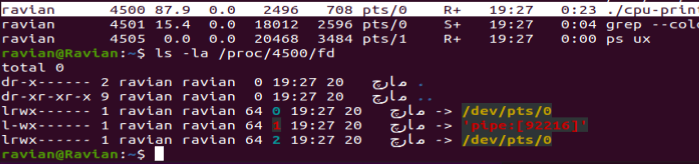
Standard error: 2>



1. Next, we will understand how the shell implements pipes. Run the following command.  
   ./cpu-print | grep hello   
   Once again, identify the newly spawned processes, and find out where their standard input/output/error file descriptors are pointing to. Use this information to explain how pipes are implemented by the shell.



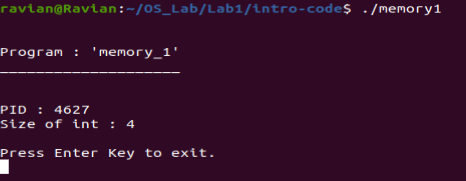
**ANS:** **Pipes** provide a unidirectional inter-process communication channel. A **pipe** has a read end and a write end. The parent process calls **pipe**() to obtain connected FDs, one child writes to one FD and another reads the same data from the other FD.

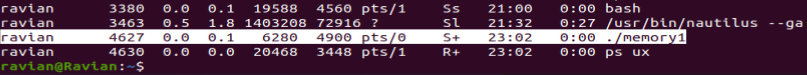


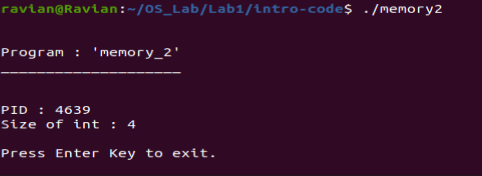
**Task-4:**

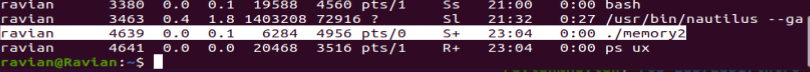
Consider the two programs memory1.c and memory2.c given to you. Compile and run them one after the other. Both programs allocate a large array in memory. One of them accesses the array and the other doesn’t. Both programs pause before exiting to let you inspect their memory usage. You can inspect the memory used by a process with the ps command. In particular, the output will tell you what the total size of the “virtual” memory of the process is, and how much of this is actually physically resident in memory. You will learn later that the virtual memory of  
the process is the memory the process thinks it has, while the OS only allocates a subset of this memory physically in RAM.  
Compare the virtual and physical memory usage of both programs, and explain your observations. You can also inspect the code to understand your observations.

**ANS:** To perform this task first the **memory1.c** and then **memory2.c** is compile and at the end their virtual and physical memory comparison is shown.

**Memory1.c:**



**Memory2.c:**



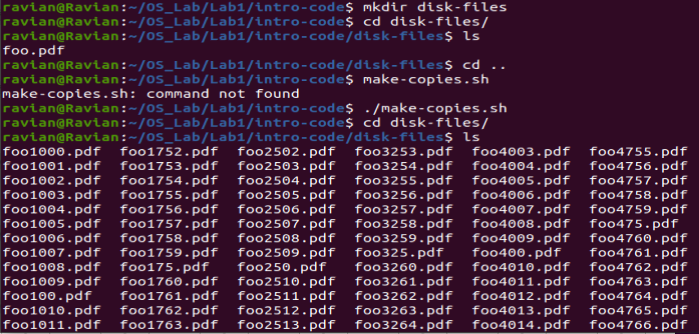
**Virtual and Physical memory comparison:**

By seeing the **memory1.c** file the **virtual memory (VSZ)** that is allocated for the execution of **memory1.c** process is **6280KB** while the **physically (RSS)** or actual memory that is assigned to this process is **4900KB**. Similarly, for **memory2.c** the virtually memory that is allocate for this process is **6284KB** while actually that process is assigned by **4956KB memory**. The difference is that **virtual (VSZ) memory** is the imagination memory that is assign to the process while the **physical (RSS) memory** is the actual or real memory that is available for some process.

**Task-5:**

In this question, you will compile and run the programs disk.c and disk1.c given to you. These programs read a large number of files from disk, and you must first create these files as follows. Create a folder disk-files and place the file foo.pdf in that folder. Then use the script make-copies.sh to make 5000 copies of the same file in that folder, with different filenames. The disk programs will read these files. Now, run the disk programs one after the other. For each program, measure the utilization of the disk while the program is running. Report and explain your observations. You will find a tool like iostat useful for measuring disk utilization.  
Also read through the code of the programs to help explain your observations.  
Note that for this exercise to work correctly, you must be reading from a directory on the local disk. If your disk-files directory is not on a local disk (but, say, mounted via NFS), then you must alter the location of the files in the code provided to you to enable reading from a local disk. Also, modern operating systems store recently read files in a cache in memory (called disk buffer cache)  
for faster access to the same files in the future. In order to ensure that you are making observations while actually reading from disk, you must clear your disk buffer cache between multiple runs of disk.c. If you do not clear the disk buffer cache between successive runs of disk.c, you will be reading the files not from disk but from memory. Look up online for commands on how to clear your disk buffer cache, and note that you will need superuser permissions to execute these commands.

**ANS:** First we make **disk-files** folder and then make **5000 copies** of **foo.pdf.**



**Clearing buffer cache memory:**

By using one of these commands we clear the buffer cache of our disk.

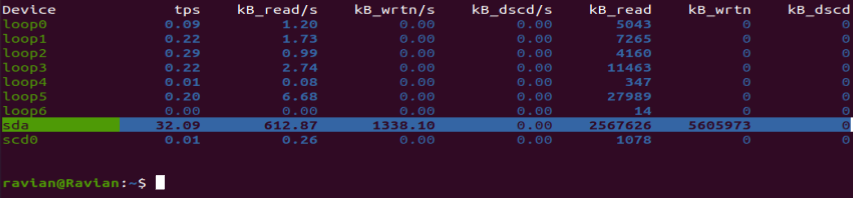
**sysctl -a vm.drop\_caches=1**

**sysctl -a vm.drop\_caches=2**

**sysctl -a vm.drop\_caches=3**

**Compiling of both files:**



**Utilization of disk while running disk.c:**

**Utilization of disk while running disk1.c:**

