180100013 – Anish Deshpande
Lab 1: Introduction

<u>Q1:</u>
a) processor: The term here refers to the processor id (0,1,2,3 in my case) , for which more specifications are listed in proc/cpuinfo. A 'processor' is an electronic circuit in the computer that carries out instructions to perform arithmetic, logical, control and input/output operations. A core is an execution unit within a CPU that receives and executes instructions. In *proc*/cpuinfo, there is a term 'core id' (either 0 or 1, for me) and 'cpu cores' (2 for me).

b) My machine has 2 cores
c) My machine has 4 processors.
d) Each processor has a slightly different frequency, namely 800.030, 800.020, 753.878 and 639.951 Mhz.
e) The amount of physical memory, i.e, RAM, is 6GB, as inferred form the 39 bit length of the physical addresses, in the information shown in *proc/cpuinfo* or the 5974552 kB mentioned in *proc*/ meminfo.
f) 1947600 kB (~2GB) is free, according to *proc/meminfo*
g) 12176 forks (processes created) since boot. I used '**vmstat -f**'
h) ctxt 18965635 gives the number of context switches performed, in *proc/stat*

<u>Q2:</u>
a) '12339' is the PID of the process running the 'cpu' command.
b) It shows slightly varying CPU usage, between 99% and 100%. (This figure is relative, and it means that it is the only process really utilising the CPU at the moment) It uses little memory, as the figure shown is 0.0%(RES/total*100) mem used. (4372KiB virtual, 792 KiB shared memory used).
c) This is a running process, as confirmed by typing 'z' in the running top command, which highlights all running processes in red, as well as the 'R' state flag.

<u>Q3:</u>
a) 3813 is the PID of the cpu-print process
b) Using the pstree command with flags -s, -p and argument 3813 (PID of the cpu-print process) we see: systemd(1)─systemd(1301)─gnome-terminal(2836)─bash(3046)─cpu-print(3813)
c) FD 0 : *dev/*pts/1
FD 1: *tmp/tmp*.txt
FD 2: dev/pts/1
The input and error file descriptors for the process (./cpu-print) point to the standard special location *dev/*pts/1. This is the location from where the shell reads the terminal input and in general, prints the terminal output. However, instead of pointing to *dev*/pts/1, the output file descriptor points to tmp/tmp.txt . As output is directed to the location specified by the '1' FD pointer, the shell has implemented I/O redirection and not printed to the terminal, but written to a file.
d) ./cpu-print: 5934
grep: 5935.
The way the shell implements pipes is by manipulating the file descriptors. For the process before the pipe (cpu-print), the output file descriptor has been set to a unique pipe 'pipe:[131047]' and the input and error fds point to the normal *dev*/pts/1 location.
For the process after the pipe (grep) the fds for output and and error are *dev*/pts/1 but the input descriptor 0 is set to point to the same unique pipe: 'pipe:[131047]'. And hence, there is a stream set up between the output of one process and the input of the other.
e) cd and history are built in executables in the Linux kernel (shell builtins) while **ls and ps** are implemented by the bash code. We use $ type /command/ to find it out for each command.

This is seen when running these commands from different terminals (history gives the same output, but ps gives a different output per terminal).
Which ls, which ps give /*bin*/ls or /bin/ps
but which cd and which history give nothing.
<u>Q4:</u>

./memory1:
PID: 6393
Virtual memory used = VIRT = 8292 KiB
Physical memory used = RES = 856 KiB

./memory2
PID: 6499
Virtual memory used = VIRT = 8292 KiB
Physical memory used = RES = 3184 KiB

Both processes have the same virtual memory uses, but different physical memory uses.
In both programs, we have an array of length 10^6, to store integers whose size is 4 bytes. It is not practical to actually allocate this much memory. Here, virtualisation comes into play, with each process thinking that it has all the memory required to store such an array with 4-byte integers. 4bytes*1000000.
In memory1.c the array isn't being accessed and no initialisations are taking place, hence no allocation of physical memory will actually happen.
But in memory2.c, we are actually assigning values and storing them, accessing the locations pointed to by the array and hence while the process is running, it will use the RAM/physical memory to store the assigned values. Hence there is a differrence in physical memory use, beyond a token amount required for the programs.

<u>Q5:</u>
Using sudo iotop -p <PID>
<u>disk.c:</u>
PID:23032
DISK read ~33M/s (=actual/total disk reads) DISK write 0B/s
IO % = ~70%
<u>disk1.c:</u>
PID: 24116
Both the disk read and disk write are 0 B/s, though there is a non-zero value initially (for a split second)
And IO% is unavailable due to non use.

The reason for this difference is that in disk.c, a random file is being read each time. There are 5000 files (5GB total), and not all of them can be stored in the cache for immediate access. And hence, due to the law of averages/probability, every file will be removed from the dusk cache buffer and read from the disk at some point, and a constant flow of data from the disk is established.
In disk1.c, however, we are only reading file foo0.pdf. Hence, after an initial read from the disk, the process reads from the disk buffer cache, and hence its IO disk utilisation drops to 0.