OS Lab 1: Introduction

Shubham Gupta

1.
   a. processor – Provides each processor on the system with an identifying number. Processor contains the “cores”.

      Core – the basic computation unit of the CPU, can run a single program context or multiple ones if it supports it

   b. 4 cores
   c. 8 processors
   d. 800 MHz for each processor; command -> more /proc/cpuinfo | grep "cpu MHz"
   e. MemTotal: 12032952 kB command -> more /proc/meminfo
   f. MemFree: 1134784 kB command -> more /proc/meminfo
   g. (number of forks) processes 219143 command -> more /proc/stat
   h. (number of context switches) ctxt 84313576 command -> more /proc/stat

2.
   a. 32743 is PID
   b. %CPU - 100.0, %MEM(memory) - 0.0
   c. R state i.e running

3.
   a. 1279 is PID. command -> ps -e | grep cpu-print, output -> 1279 pts/1 00:00:09 cpu-print
   b. systemd(1)─systemd(2022)─gnome-terminal-(31961)─bash(28211)─cpu-print(1279) These are the PIDs (inside the parentheses) in hierarchy, systemd being the init, and its ancestors going in order from left to right. Command -> pstree -s -g 1279
      PID of parent of cpu-print is 28211
   c. Commands and outputs ->
      $ ./cpu-print > /tmp/tmp.txt &
      [1] 3181
      $ readlink /proc/3181/fd/0 /dev/pts/1
      $ readlink /proc/3181/fd/1 /tmp/tmp.txt
      $ readlink /proc/3181/fd/2
      /dev/pts/1
      Here, /dev/pts/1 is basically std io. In this process, the shell takes commands from there as input, and also the error file i.e 2 is also std io, so any error will be printed there. Now, the output file, i.e 1 is kept as the /tmp/tmp.txt file where we essentially want the output. So the ‘>’ redirection command has simply set the output file for the process to /tmp/tmp.txt which would otherwise be the std io.
   d. Commands and outputs ->
      $ ./cpu-print | grep hello &
      [1] 16573
      $ ps

```
  PID TTY          TIME CMD
16550 pts/0    00:00:00 bash
16572 pts/0    00:00:01 cpu-print
16573 pts/0    00:00:00 grep
16577 pts/0    00:00:00 ps $
readlink /proc/16573/fd/0
pipe:[1356238]
$ readlink /proc/16573/fd/1 /dev/pts/1
$ readlink /proc/16573/fd/2 /dev/pts/1
$ readlink /proc/16572/fd/0 /dev/pts/1
$ readlink /proc/16572/fd/1 pipe:[1356238]
$ readlink /proc/16572/fd/2
/dev/pts/1
```

Here we have 2 processes, first is "./cpu-print" , second is "grep hello" . The pipe sets input for the "grep" command as the "pipe:[1356238]" and the output of "./cpu-print" to also "pipe:[1356238]", all other file descriptors remain stdio. So basically, output of first process goes to a temporary place i.e "pipe:[1356238]" and that itself serves as input to the 2nd process.

e. ls, ps are execed by bash.
history, cd are implemented by bash code.
Reasoning → in the directory "/bin" are some executable files. These include ls, ps but not history and cd. Hence history and cd are being *implemented* by bash. For ls, ps, bash directly uses the executable file and hence are *execed* by bash.

4.

memory1.c -> VSZ=8292 kB , RSS=712 kB VSZ= total size of virtual memory memory2.c -> VSZ=8292 kB , RSS=3300 kB RSS= physically resident memory size

Command used = ps -p <PID> -o pid,stat,vsz,rss,comm

RSS is the memory actually used by a process in RAM. Since memory2.c is actually accessing the array, it has a higher RSS than memory1.c which is just declaring the array and not doing anything else.

VSZ (virtual size) is the memory that a process "believes" it has. This majorly corresponds to the array declaration which has been done in both the programs. Both processes have the same VSZ as they have declared an "int" array of the same size.

5.

For the disk.c process, kB_read/s of iostat is 1277.77. For the disk1.c process, it is 752.80.

Here kB_read/s corresponds to the disk utilization in terms of the amount of data being read from the disk per second. From the kB_read/s, it is clear that disk.c involved more disk utilization. Reason for this is that disk.c reads a different file every time, due to the index of foo{i}.pdf being generated by the rand() function. Hence every time it reads the file from disk. In the disk1.c file. The same file i.e foo0.pdf is being read repeatedly. Now after running once, the file gets stored in the cache and the file is not needed to be read from the disk every time. Hence the disk utilization is only done while reading the file once, while in the disk.c it is being done every time (at least 5000 times, i.e number of different pdfs available)