CS 333: Operating Systems Lab
Lab1 : Introduction to Operating systems

140050037 - Nikhil Kumar Lakumarapu
140050048 - Uday Kusupati

1.

4 cores[/proc/cpuinfo]
Total Memory: 8142220 kB
Free memory: 5267604 kB(Fraction:64.69%)[/proc/meminfo]
No.of context switches: 9494040 [/proc/stat ctxt]
No.of forks(processes forked) since boot: 6921[/proc/stat processes]

2.

We went through the utilization of the resources using the following commands
- top for %cpu
- top for %mem
- iostat -d -x *interval* for %disk(interval of order of a sec)
- sar -n DEV for % %network util
Values for the files as follows respectively
cpu.c: 100.0% 0.0% 2.7%(cpu,mem,disk)

Clearly cpu is the bottleneck resource. It can be justified as the code doesn't have any blocking calls, just does arithmetic operations infinitely using cpu and thereby pushes the cpu to its maximum.

cpu-print.c : 5.0% 0.0% 24%

Here, we redirected the output to a file, which resulted in making the cpu bottleneck resource. Thereby, we can say that the process writing the output to the terminal is working at its maximum and cannot run faster thereby acting as a bottleneck. This is because writing to a file is faster as the data can be buffered, whereas to terminal it needs to write a lot more times due to absence of buffer. While writing to file, writing is faster and then cpu reaches its limit and acts as bottleneck. In this way we figured that writing to terminal is the bottleneck. It can be justified by reading the code as it doesn't have blocking calls and no access to disk, just printing the time, thereby supporting our observation

disk.c : 4.75% 0.0% 98.3%

The disk utilisation and the disk read rate were reaching their maximas and observed using the above syntax. The code suggests the same as we are fetching a random file to read every single time in a loop from the disk which is slow and the disk works at its maximum to fetch the file.

disk1.c : 100.0% 0.0% 3.7%

This time we access a single file every single time thereby simply using the file from the cache which is pretty much fast and therefore the cpu acts as a bottleneck again.

3. cpu.c

User mode and kernel mode number of jiffies(processor times) are 2477 , 11 respectively. So in this case, the process runs majorly in the user mode.

cpu-print.c

User mode and kernel mode number of jiffies(processor times) are 42, 229 respectively.. So in this case, the process runs majorly in the kernel mode

This information is found from the stat file in the /proc folder

The time spent is like this because in cpu.c, there are no system calls like printf() which make the OS go to the kernel mode, so the process spends most of the time in the user mode. Where as cpu-print.c, there is a printf() which is a system call, so forces the process into kernel mode

4. cpu.c has mostly involuntary context switches
disk.c has mostly voluntary context switches
Justification:
cpu.c has no blocking calls and uses cpu continuously and doesn't have a voluntary end due to infinite loop. So the only way it is context switched is through the timer/scheduler code of the kernel which makes a involuntary context switch.
disk.c has a blocking call of writing to a random file in every iteration, so it explains that it voluntarily context switches when the call to access the file happens many times rather compared to the involuntary switches due to the scheduler

5.
This can be found using i) pstree -p -s <pid> or ii)pstree -H PID(where PID is id of bash process found using ps) . This is
for "Terminator" :

init(1)———lightdm(1907)———lightdm(22742)———init(22940)———/usr/bin/x-term(30768)———ba
sh(30780)

for "Terminal(gnome)" :

init(1)———lightdm(1896)———lightdm(2168)———init(2657)———gnome-terminal(4116)———bash(
5137)

6. cd and history are implemented by the bash code itself as they are bash shell builtins, as they should run within the process of bash for them to function properly. Whereas ps, ls are simply executed by the bash shell and their executables can be located through *type ps* or *type ls*. The built-ins can be looked up in *man bash*. Alternatively, simple bash scripts that infinitely run ls or such commands can be executed and the process list top can be checked for the presence of ls which appears. Or we can use ps -a to show all processes associated with terminal when running the above denoted bash script.

7. 0 points to stdin, 1 points to stdout, 2 points to stderr
So I/O redirection is implemented by bash, as follows. Here when we redirect the output to a file, then when see the file pointed to by 1(stdout), it gets written by the output and then gets redirected to the file tmp.txt from there.

8. To know the processes spawned by bash, type $ ps -a in the terminal, it shows cpu-print, and grep as the processes.
Check the fd folder of both the processes, $cat 1 of cpu-print shows the output of the cpu-print, and when you do $cat 0 in the grep's fd folder, we see the same thiong as $cat 1 of cpu-print. So the output file of the left operand of "pipe" gets redirected to the input file of right operand of pipe.