sheet

In this sheet, we will run a few simple commands on the Linux shell to understand the basics of operating systems. In each question below, you will be expected to run a command, observe some output and report it. Please note down the answers to all questions in a report. In addition to the final answer, you must also state the commands or tools you used to arrive at your answer.

The following helper files are provided to you: `cpu.c`, `cpu-print.c`, `disk.c`, `disk1.c`, `foo.pdf`, `make-copies.sh`, `memory1.c`, and `memory2.c`.

1. The `/proc` file system is a mechanism provided by Linux for the kernel to report information about the system and processes to users. The `/proc` file system is nicely documented in the proc man page. You can access this document by running the command `man proc` on a Linux system. Understand the system-wide proc files such as `meminfo` and `cpuinfo`, and per-process files such as `status, stat, limits, maps` and so on. Now answer the following questions using the `/proc` filesystem.

   (a) Run command `more /proc/cpuinfo` and explain the following terms: `processor` and `cores`. [Hint: Use `lscpu` to verify your definitions.]

   (b) How many `cores` does your machine have?

   (c) How many `processors` does your machine have?

   (d) What is the frequency of each processor?

   (e) How much physical memory does your system have ?

   (f) How much of this memory is free ?

   (g) What is total number of number of forks since the boot in the system ?

   (h) How many context switches has the system performed since bootup ?

2. 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.

   (a) Compile the program `cpu-print.c` given to you and execute it in the `bash`.

   ```
   $ gcc cpu-print.c -o cpu-print
   $ ./cpu-print
   ```

   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.

   (b) 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).

   (c) 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) and pointing to. Using this information, can you describe how I/O redirection is being implemented by the shell?

   (d) 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.

   (e) Consider the following commands that you can type in the bash shell: `cd`, `ls`, `history`, `ps`. Which of these are system programs that are simply execed by the bash shell, and which are implemented by the bash code itself?

3. 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. Compare the virtual and physical memory usage of both programs, and explain your observations. You can also inspect the code to understand your observations.

4. 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, if you have the permissions for it, you must clear your disk buffer cache between multiple runs of `disk.c`, in order for this lab to work correctly.

5. Continue with your setup of 5000 files on disk for this question as well (you may delete the files after this, if you find that they are eating up too much space on disk). For this question, you will need to compile and run the three programs `cpu.c`, `disk1.c`, and `disk.c` one after the other.

   Every process consumes some resources (CPU, memory, disk or network bandwidth, and so on). When a process runs as fast as it can, one of these resources is fully utilized, limiting the maximum rate at which the process can make progress. Such a resource is called the bottleneck resource of a process. A process can be bottlenecked by different resources at different points of time, depending on the type of work it is doing.

   Run each of the three programs `(cpu, disk, and disk1)` separately, and identify what the bottleneck resource for each is (without reading the code). For example, you may monitor the utilizations of various resources and see which ones are at the peak. Tools like `top` help you monitor the CPU utilization, while a tool like `iostat` will help you measure disk utilization. Next, read through the code and justify how the bottleneck you identified is consistent with what the code does.

   For each of the programs, you must write down three things: the bottleneck resource, the reasoning that went into identifying the bottleneck, (e.g., the commands you ran, and the outputs you got), and a justification of the bottleneck from reading the code.

Submission instructions

- You must submit a text/pdf file containing answers to all the questions above in order.
- Place this file and any other files you wish to submit in your submission directory, with the directory name being your name.
- Tar and gzip the directory using the command ***tar -zcvf name.tar.gz name*** to produce a single compressed file of your submission directory. Submit this tar gzipped file.