Lectures on Operating Systems (Mythili Vutukuru, IIT Bombay)

Lab: Introduction to Linux Tools

In this lab, we will run a few simple commands on the Linux shell to understand the basics of operating systems. In each of the exercises 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`.

Before you begin

- Before you begin this course, you must be comfortable with writing and compiling code on Linux. You must also be comfortable with basic operations like creating / editing / copying / moving / viewing files using Linux commands, running commands from the terminal, and using command-line techniques like redirection or pipes or searching for a string in the output. If you are not familiar with Linux, it may be a good idea to familiarize yourself with the most common Linux commands before proceeding further. You will find many helpful tutorials online, like this one:

  `http://www.ee.surrey.ac.uk/Teaching/Unix/`

- Familiarize yourself with the common tools on Linux like `top` and `ps` that are used to monitor processes running in the system. Understand the most useful and common commandline options to use with these commands. Understand the various fields in the output produced by these commands.

- Understand the `/proc` filesystem in Linux. 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,` and `maps`.

- Understand what are system calls, and which system calls are supported by your OS.

Exercises

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

   (a) Run command `more /proc/cpuinfo` and explain the following terms: `processor` and `cores`. Use the command `lscpu` to verify your definitions. You may want to understand the concept of CPU hyperthreading at a high level before attempting this question.

   (b) How many `cores` does your machine have? 8

   (c) How many `processors` does your machine have? 16

   (d) What is the frequency of each processor? 3193.924

   (e) What is the architecture of your CPU?

   (f) How much physical memory does your system have ? more /proc/meminfo

   (g) How much of this memory is free ?

   (h) What is total number of number of forks and context switches since the system booted up?
   more /proc/stat - ctxt gives context switches and processes gives forks

2. In this question, 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.

   (a) What is the PID of the process running the `cpu` command? → 3582

   (b) How much CPU and memory does this process consume? → 100% cpu 0% mem

   (c) What is the current state of the process? For example, is it running or in a blocked state or a zombie state? R → running

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.

   (a) Compile the program `cpu-print.c` given to you and execute it in the `bash` or any other shell of your choice as follows.

```
$ 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) are 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) When you type in a command into the shell, the shell does one of two things. For some commands, executables that perform that functionality already come with your Linux kernel installation. For such commands, the shell simply invokes the executable like it runs the executables of your own programs. For other commands where the executable does not exist, the shell implements the command itself within its code. Consider the following commands that you can type in the bash shell: `cd`, `ls`, `history`, `ps`. Which of these commands already exist as executables in the Linux kernel directory tree that are then simply executed by the bash shell, and which are implemented by the bash code itself?

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.

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.

6. User programs make several system calls (which are sort of like function calls to the OS, but more complicated) to invoke OS functionality during their execution. The `strace` tool lets you trace the system calls invoked by a running executable.

   Consider any executable in the examples above, or any Linux command like `ls` and run it with the strace command as follows

   ```
   strace <executable>
   ```

   This command shows the list of all system calls made by the executable during its executation. How many system calls in total does the program make? Look up online to find the list of system calls supported by your operating system, and see if you can locate any of them in the output of the `strace` command.

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 roll number (say, 12345678).
- Tar and gzip the directory using the command `tar -zcvf 12345678.tar.gz 12345678` to produce a single compressed file of your submission directory. Submit this tar gzipped file on Moodle.