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.

Before you begin

- Before you begin this course, you must be comfortable with writing and compiling code on Linux. Learn how to compile and run C programs, both from graphical IDEs, as well as using a terminal on a Linux machine. Type up your programs in the editor of your choice. Compile and run your code from the terminal as follows.

```
$gcc helloworld.c
$ ./a.out
$gcc helloworld.c -o hello
$./hello
```

- You must also be comfortable with basic operations like creating, editing, copying, moving, viewing files using Linux commands, 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/`

- Learn how to tar and untar files. For example, you can untar the tar-gzip file provided with the lab using the following command:
  `tar -zxvf intro-code.tgz`

- Familiarize yourself with the commands `top`, `ps`, and `pstree` that are used to monitor processes running in the system. Learn how to use the man pages for these tools, e.g., by typing `man ps` in a terminal. You can press 'h' when inside a man page to get useful keyboard shortcuts to help you effectively navigate the page.

- 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 at `man proc`. Understand the system-wide proc files such as `meminfo`, `cpuinfo` and `stat`, as well as per-process files like `status`.

Exercises

1. We will begin with Linux shell commands used to examine the processes running in a system.

   (a) List all processes running in the system. Explore the different styles of displaying the output, e.g., BSD syntax vs. standard (ps) syntax.

   (b) Print a process tree for all processes running in the system.

   (c) List the pid, ppid, state, command, for all process running in the system.

   (d) List the pid, ppid, and name of the command of process with pid 123 (or any PID).

   (e) Print the process tree of a process with pid 123 using ascii characters.

2. Answer the following questions by looking at the files in the proc filesystem, and the outputs of the `lscpu`, `uname`, `uptime` commands on your system.

   (a) How many CPUs / cores / processors does your machine have? What is the frequency of each processor? What is the architecture of your CPU?

   (b) How much physical memory does your system have? How much of this memory is free?

   (c) For how long has your system been running? What is total number of context switches since the system booted up?

   (d) What is name of your operating system? What is the kernel version?

3. 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 shell.

   ```
   $ 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?

   (b) How much CPU and memory does this process consume?

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

4. 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.

(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) Using the `pstree` command, print the process tree of the shell process that is running your `cpu-print` program.

(d) Now, stop the execution of the `cpu-print` executable. Run a long background command, e.g., `sleep 10000 &` on your shell. Restart the long `cpu-print` process in the foreground again. Visualize the process tree of your shell again.

5. 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 to run the command. 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`, `ps`, `sleep`, `history`. Which of these commands already exist as executables in the Linux kernel directory tree, and which are implemented by the bash code itself? If the executable already exists, what is the pathname of the executable file? You may use the `which` command to help you.

6. 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.

7. 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.

8. 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 execution. 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.