CS333: Operating Systems Lab

Lab 1: Introduction to Operating Systems

Goal

The goal of this lab is to give you a basic introduction to the OS, shell, and processes in general. You will run a few simple programs (as processes on your Linux machine), and monitor their behavior using the `proc` file system and other related tools.

Before you start

- Understand the `proc` file system of linux. A web search or `man proc` should give you a lot of information. In particular, understand the `stat` and `status` files of a process, and the system-wide files of the proc directory.
- Learn how to use the following basic commands. `top` tells you information about processes and their resource consumption. `ps` tells you information about all processes in the system. `ps` has several useful commandline arguments. It is worthwhile understanding and learning the most useful ones. For example, `ps -A` gives you information about all processes in the system. Also, learn to use any disk monitoring tool. For example, `iostat` gives you information about disk utilization. Again, understanding various useful commandline arguments (`-c -x -m` etc.) will be quite handy.
- Read through Chapter 0 in the xv6 textbook, especially the description about how the xv6 shell works.
- Four simple C programs are provided to you with this lab: `cpu.c`, `cpu-print.c`, `disk.c`, and `disk1.c`. You may want to compile and keep all the executables handy for solving the exercises. For example, you can create the `cpu` executable as follows.

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

- The disk programs read a large number of files, and you must create and setup these files before you run the program. Create a subfolder `files` in the working directory of the `disk` / `disk1` programs. Create 10,000 files of size 2MB each in the folder, with names `foo0.txt`, `foo1.txt`, and so on until `foo9999.txt`. This repository of 10,000 files of 2MB each will be used in several take-home labs subsequently, so you may want to keep them around on your machine. Feel free to write 2MB of random data into each of these files.

- Read up briefly about disk I/O and buffer caches. When files are read from a hard disk, they are cached in memory (as we will see later when we study file systems). Of course, this cache is limited in size. So, when you read and write a large number of files in a short span of time, most files will have to be accessed from the disk, as the cache can only hold a small subset of these files. However, if you are reading only one small file multiple times, for example, then that file will most likely be served from the cache after its first access.

Exercises

Do the following exercises, and record your observations in your report. Note: run the programs one after the other (and not simulatenously) when doing the exercises, in order to cleanly record the observations from each program separately.

1. Collect the following basic information about your machine using `proc`. How many CPU cores does the machine have? How much memory, and what fraction of it is free? How many context switches has the system performed since bootup? How many processes has it forked since bootup?

2. 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 four programs (`cpu`, `cpu-print`, `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. 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.

3. Recall that every process runs in one of two modes at any time: *user mode* and *kernel mode*. It runs in user mode when it is executing instructions / code from the user. It executes in kernel mode when running code corresponding to system calls etc.

   Compare (qualitatively) the programs `cpu` and `cpu-print` in terms of the amount of time each spends in the user mode and kernel mode, using information from the `proc` file system. For examples, which programs spend more time in kernel mode than in user mode, and vice versa? Read through their code and justify your observations.

4. Recall that a running process can be interrupted for several reasons. When a process must stop running and give up the processor, it's CPU state and registers are stored, and the state of another process is loaded. A process is said to have experienced a *context switch* when this happens. Context switches are of two types: voluntary and involuntary. A process can voluntarily decide to give up the CPU and wait for some event, e.g., disk I/O. A process can be made to give up its CPU forcibly, e.g., when it has run on a processor for too long, and must give a chance to other processes sharing the CPU. The former is called a voluntary context switch, and the latter is called an involuntary context switch.

Compare the programs `cpu` and `disk` in terms of the number of voluntary and involuntary context switches. Which program has mostly voluntary context switches, and which has mostly involuntary context switches? Read through their code and justify your observations.

5. Open a bash shell. Find its `pid`. Write down the process tree starting from the first `init` process (`pid = 1`) to your bash shell, and describe how you obtained it. You may want to use the `pstree` command.

6. 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 exec'ed by the bash shell, and which are implemented by the bash code itself?

7. Run the following command in bash.

   ```
   $./cpu-print > /tmp/tmp.txt &
   ```

   Find out the pid of the new process spawned to run this command. Go to the `proc` folder of this process, and describe where its I/O file descriptors 0, 1, 2 are pointing to. Can you describe how I/O redirection is being implemented by bash?

8. Run the following command with `cpu-print`.

   ```
   $./cpu-print | grep hello
   ```

   Once again, identify which processes are spawned by bash, look at the file descriptor information in their `proc` folders, and use it to explain how pipes work in bash.

Submission and Grading

You may solve this assignment in groups of one or two students. You must submit a tar gzipped file, whose filename is a string of the roll numbers of your group members separated by an underscore. For example, `rollnumber1_rollnumber2.tgz`. The tar file should contain the following:

- `report.pdf`, which contains your answers to the exercises above. Be clear and concise in your writing.

Evaluation of this lab will be based on reading your answers to the exercises in your report.