CS 1217 Spring 2023

Assignment 2

Due - Tuesday, 14th Feb 11:59 pm

(60 Points)

Instructions

1. **Only one** person of a team (of 2) should submit the assignment. This should be the same person who had submitted assignment 1.
2. The answers should be submitted as a PDF file. The PDF file should have names of both the team members.
3. This assignment constitutes **2% of the total course grade**.
4. There is only one pdf that needs to be submitted for this assignment through **Google Classroom**. Please name the file as **<Team_Member_1_Name>_cs1217_assignment2.pdf**
5. Please refer to the class webpage for notes on late assignment submission policy
6. The template code required for this assignment is available on Github Classroom. Use this link to access the Github Classroom assignment.
    a. You should compile these codes such that the names of the executables are the same as that of the .c files. For example, to compile `cpu.c` to an executable called `cpu`, you can use the following command from the terminal : `gcc -o cpu cpu.c` This executable can then be run as `./cpu` from the shell command line.

Goal

The goal of this assignment is to get you familiar with the concepts of threads and processes, as well as some of the command-line tools available for carrying out various tasks.

Q1.

Understand Linux's `proc` file system. A web search or `man proc` should give you a lot of information. You should get familiar with the use of `man` pages - they provide a lot of documentation right at your fingertips, without having to use the browser. **(0 Points)**

Q2.

Learn how to use the basic shell commands, which we have talked about in class as well. `top` provides information about processes and their resource consumption. `ps` tells you information about all processes in the system. `ps` has several useful command-line arguments. It is worthwhile understanding and learning the most useful ones. For example, `ps -A` gives you information about **all** the processes in the system, irrespective of the user. What are the

command-line arguments that you'd have to provide to get only the processes that belong to a specified user, say `cs304`. **(5 Points)**

Q3.
Read Chapter 0 of the xv6 book, which focuses on the interfaces that the OS (in this case, xv6) provides. One of the most important, and widely used interfaces is the shell, or the terminal. In the days of GUI-less systems, shell was the primary interface that the user had with the system. Focus on how the `xv6` shell works. Think about how you'd implement a program that resembles a shell and implements the basic functionality that any modern shell exports. Is there some support from the operating system that you'd need to implement a shell? If so, what is that support? Write down the following points as an answer to the question (a) what are the core requirements of any shell? (b) What is the support/functionality from the OS that you'd require to implement a shell. (c) Provide the steps that you'd need to implement to have a basic, working shell program. **(10 Points)**

Q4.
We talked about in class how 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 that belongs to system calls etc. Compare (qualitatively) the programs `cpu` and `cpu-print` (download link is provided in bullet 6 of ***instructions*** above) in terms of the amount of time each spends in the user mode and kernel mode, using information from the `proc` file system. Another, relatively easier mechanism would be to look up the `time` command. For the example code that has been provided, which programs spend more time in `kernel` mode than in `user` mode, and vice versa? Read through their code and justify your observations. **(10 Points)**

Q5.
Open the terminal program. Can you find out ***which*** shell is the program running? (Hint: The terminal program maintains the name of the shell type it is running as a special variable. Find out which variable that is. Once you know the variable's name, figure out a way to print it from the command-line. Document the process). Find the pid of the shell. Write down the process tree starting from the first `init` process (pid = 1) to your bash shell, and describe how you obtained it. Read up the documentation of the `pstree` command and figure how that can be used for this part. **(5 Points)**

Q6.
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? **(10 Points)**

Q7.

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 (download link is provided in bullet 6 of ***instructions*** above -- `cpu`, `cpu-print`, `disk`, and `disk1`) separately, and identify what the bottleneck resource for each is (without reading the code).

For `disk` and `disk1` , you need to create 10,000 random files in a subfolder called **files**. Here is a `sample code` in python for creating ONE file, of size 2 MB. Modify it suitably for the purpose of this question. You need to create files with the names `foo0.txt`, `foo1.txt`, `foo2.txt`, … , `foo10000.txt`, Please remember this will consume a lot of space on your disk, so once you are done, delete them.

```
import os

file_name = "foo1.txt"
file_size = 2*1024*1024 #size in Bytes

with open(file_name, "wb") as f:
      f.write(os.urandom(file_size))
```

For example, you may monitor the utilization 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. You should try tools like **top, htop,** and **vmstat**.
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. **(20 Points)**