CS 347M Spring 2022

Programming assignment 1

This assignment has three parts, all pertaining to process management in operating systems.

Before you begin

- 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 `/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`.
- Familiarize yourself with the various process related system calls in Linux: `fork`, `exec`, `exit` and `wait`. The "man pages" in Linux are a good source of learning. You can access the man pages from the Linux terminal by typing `man fork`, `man 2 fork` and so on. You can also find several helpful links online (e.g., `http://manpages.ubuntu.com/manpages/trusty/man2/fork.2.html`). It is important to understand the different variants of these system calls. In particular, there are many different variants of the `exec` and `wait` system calls; you need to understand these to use the correct variant in your code.
- Download, install, and run the original xv6 OS code from `https://www.cse.iitb.ac.in/~mythili/os/labs/xv6-public.tgz`. You can use your regular desktop/laptop to run xv6; it runs on an x86 emulator called QEMU that emulates x86 hardware on your local machine. In the xv6 folder, run `make`, followed by `make qemu` or `make-qemu-nox`, to boot xv6 and open a shell.

Part A: Understanding Linux processes

This part has multiple exercises, and for each exercise, you will be expected to run some commands on a Linux shell and note down your observations. Please note down the answers to all questions in a PDF or text file and submit. In addition to the final answer, you must also state the commands or tools you used to arrive at your answer, and be able to demonstrate an understanding of these commands in the evaluation. The following helper files are provided to you for this part of the assignment: `cpu.c`, `cpu-print.c`, `fork4.c`.

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 find out how many CPU cores or processors your machine has. Note that if your CPU has hyperthreading, you will notice that a CPU core can appear as multiple "processors" due to hyperthreading. If there is no hyperthreading, the number of CPU cores will be equal to the number of processors.

   (b) What is the frequency of each CPU core/processor?

   (c) How much physical memory does your system have, and how much of this memory is free ?

   (d) What is total number of number of forks and context switches in the system since bootup?

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?

   (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?

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?

4. Consider the program `fork4.c` provided to you.

   (a) Compile and run the program. Explain the output produced.

   (b) The program currently does not have any wait statements. Write suitable wait statements in the program, reap the child processes, and print the PID of the processes as they are reaped to the screen. You must describe the changes made in your report and demonstrate the modified program in your evaluation. There is no need to submit your modified `fork4.c` code.

Part B: A simple shell

In this part of the assignment, we will build a shell to run simple Linux commands. Your shell must take in user input, fork a child using the `fork` system call, call `exec` from the child to execute the user command, reap the dead child using the `wait` system call, and come back to the user for the next command. Here are some more specifications for your shell.

- Your shell must execute *all* simple Linux commands like `ls`, `cat`, `echo` and `sleep`. These commands are readily available as executables on Linux, and your shell must simply invoke the existing executable. It is important to note that you must implement the shell functionality yourself, using the fork, exec, and wait system calls. You must not use library functions like `system` which implement shell commands by invoking the Linux shell—doing so defeats the purpose of this assignment!
- Your simple shell must use the string "$ " as the command prompt.
- Your shell should interactively accept inputs from the user and execute them one after the other, until the user types `exit` on the shell. Upon receiving the exit command, the shell should terminate cleanly.
- You can assume that the command to run and its arguments are separated by one or more spaces in the input, so that you can "tokenize" the input stream using spaces as the delimiters.
- You can assume that the Linux commands are invoked with simple command-line arguments, and without any special modes of execution like background execution, I/O redirection, or pipes. You need not parse any other special characters in the input stream. *Please do not worry about corner cases or overly complicated command inputs for now; just focus on getting the basics right.*
- An empty command (typing return) should simply cause the shell to display a prompt again without any error messages.
- Your shell must gracefully handle errors. For all incorrect commands or any other erroneous input, the shell itself should not crash. It must simply notify the error and move on to prompt the user for the next command. Note that it is not easy to identify if the user has provided incorrect options to the Linux command (unless you can check all possible options of all commands), so you need not worry about checking the arguments to the command, or whether the command exists or not. Your job is to simply invoke `exec` on any command that the user gives as input. If the Linux command execution fails due to incorrect arguments, an error message will be printed on screen by the executable, and your shell must move on to the next command. If the command itself does not exist, then the exec system call will fail, and you will need to print an error message on screen, and move on to the next command. In either case, errors must be suitably notified to the user, and you must move on to the next command.
- For all commands, you must take care to terminate and carefully reap any child process the shell may have spawned. When the forked child calls `exec` to execute a command, the child automatically terminates after the executable completes. However, if the `exec` system call did not succeed for some reason, the shell must ensure that the child is terminated suitably. When not running any command, there should only be the one main shell process running in your system, and no other children.

To test your code, run a few common Linux commands in your shell, and check that the output matches what you would get on a regular Linux shell. Test your code with correct as well as incorrect commands. Further, check that your shell is correctly reaping dead children, so that there are no extra zombie processes left behind in the system. You can use a long running command like `sleep 100` in your shell, and open another termonal to check the process-related information using the `ps` or `pstree` or `top` or some other such command. You must ensure that there is only one shell parent process, one child process when a command is running, and no other zombie children in the system.

A skeleton code `my_shell.c` is provided to get you started. This program reads input and tokenizes it for you. You must add code to this file to execute the commands found in the "tokens". You may assume that the input command has no more than 1024 characters, and no more than 64 tokens. Further, you may assume that each token is no longer than 64 characters. You can extend this program, or write another C/C++ program, to solve this assignment.

Part C: New system calls in xv6

In this part of the assignment, we will add new system calls to the xv6 OS, to understand the mechanism of system calls better. Before we add new system calls, you must understand the existing code for implementing system calls in xv6. For this purpose, pick any of the existing system calls in xv6 and trace their implementation throughout the xv6 code. Any given system call is implemented in multiple places in xv6 code, some of which are described below:

- `user.h` contains the system call definitions in xv6, which are exported to userspace programs.
- `usys.S` contains code to invoke the trap instruction for each system call.
- `syscall.h` contains a mapping from system call name to system call number. Every system call must have a number assigned here.
- `syscall.c` contains helper functions to parse system call arguments, and pointers to the actual system call implementations.
- `sysproc.c` contains the implementations of process related system calls.
- `defs.h` is a header file with function definitions in the kernel.
- `proc.h` contains the `struct proc` structure.
- `proc.c` contains implementations of various process related system calls, and the scheduler function. This file also contains the definition of `ptable`, so any code that must traverse the process list in xv6 must be written here.

Next, understand how arguments are passed to system calls and how return values are passed back to user code. Some system calls do not take any arguments and return just an integer value (e.g., `uptime` in `sysproc.c`). Some other system calls take in multiple arguments like strings and integers (e.g., `open` system call in `sysfile.c`), and return a simple integer value. The arguments to system calls are first stored on the userstack, and xv6 kernel helper functions fetch these arguments for use inside kernel code. Look through the implementing of existing system calls to understand how this happens.

Now, using this understanding, write the following new system calls in xv6.

1. Implement a system call, called `hello()`, which prints Hello to the console. You can use cprintf for printing in kernel mode.
2. Implement a system call, called `helloYou(`*`name`*`)`, which prints a string *name* to the console. You can use cprintf for printing in kernel mode.
3. Implement the system call `getNumProc()`, which returns the total number of active processes in the system (either in embryo, running, runnable, sleeping, or zombie states).
4. Implement the system call `showAncestry())`, which prints to screen the PIDs of all ancestors of the process. The first line of your output should be the PID of the parent, the next line should be the PID of the grandparent, and so on. You must do this until you reach the init process.

For all system calls that do not have an explicit return value mentioned above, you must return 0 on success and a negative value on failure. Note that the process table structure `ptable` is protected by a lock. You must acquire the lock before accessing this structure for reading or writing, and must release the lock after you are done. Please ensure good locking discipline to avoid bugs in your code.

Once you have written the system call code, you should write user programs that invoke the system call, so that you can test your implementation. Below are some guidelines to do so.

- Learn how to write your own user programs in xv6 by looking at existing examples. We have also provided a simple test program `testcase.c` as part of this lab. This test program is compiled by the updated `Makefile` which is provided to you. Copy these two files into your xv6 folder, compile and run xv6 again. You can now run this new testcase on the xv6 shell by typing `testcase` at the command prompt. This example should help you understand how to write new test cases in xv6, how to add them to the Makefile, and how to execute it from the xv6 shell.

- With this understanding, proceed to write other such user programs to test your new system call implemenations. We have provided you with `test-hello.c` and `test-helloyou.c` to test the first two system calls. We have not added these test cases to the Makefile because the corresponding system calls are not defined yet, and so the test cases will not compile right now. You must write test programs for the other system calls yourself. Write interesting test cases that showcase the correctness of your implementation, e.g., for testing your system call that displays the process tree, you may want to fork multiple times, and then invoke the system call, so that you have a long family tree to display.

- Once you implement the system calls, you must add these test programs to the xv6 Makefile, and then compile and run xv6. You can then run the commands corresponding to the testcase on the command prompt in order to execute and test your system call implementation. Remember that any test program you write should be included in the Makefile for it to be compiled and executed from the xv6 shell.

- Note that the xv6 OS itself does not have any text editor or compiler support, so you must write and compile the code in your host machine, and then run the executable in the xv6 QEMU emulator.

Submission instructions

- For part A, you must submit a text/pdf file containing answers to all the questions above in order. For part B, you must submit the shell code `my_shell.c` or `my_shell.cpp`. For part C, you must submit all xv6 files you modified to implement the assignment.
- 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.

We will grade your submission in a demo/viva session. You will be asked to run your code and show it to the TAs. You will also be asked to explain your code, and answer other questions about it. Therefore, please make sure that you write all the code you submit yourself, and also ensure that you fully understand the code you write. All submissions will be checked for plagiarism, and any copying detected will be dealt with strictly.