Computer Science Practice and Experience: Operating Systems (Comp Sci 3SH3), Term 2, Winter 2018
Dr. Neerja Mhaskar

Assignment-I [35 points]

Due at 11:59pm on February 10th, 2018

- **No late assignments accepted.**
- Make sure to submit a version of your assignment ahead of time to avoid last minute uploading issues.
- Note that groups copying each other's solution will get a zero, and reported to the department.
- Only one copy of your assignment should be submitted under your group using Avenue.
- In your C programs, you should follow good programming style, which includes providing instructive comments and well-indented code.
- You submission should include a readme file. Your readme file should contain the names, studentIDs and MACIDs of each student in the group. It should also clearly state the work done by each student in the group.
- You need to ensure that all your solutions work correctly on the Linux virtual machine.

[10 points] Question 1: Linux Kernel Module for Iterating over Tasks with a Depth-First Search Tree.

Beginning from the `init_task`, design a kernel module that iterates over all tasks in the system using a DFS tree. Output the name, state, and `pid` of each task. Perform this iteration in the kernel entry module so that its output appears in the kernel log buffer. If you output all tasks in the system, you may see many more tasks than the ones that appear with the `ps -ael` command. This is because some threads appear as children but do not show up as ordinary processes. Therefore, to check the output of the DFS tree, use the command

```
ps -eLf
```

This command lists all tasks—including threads—in the system. To verify that you have indeed performed an appropriate DFS iteration, you will have to examine the relationships among the various tasks output by the `ps` command.

This question should be completed using the Linux virtual machine you installed as part of Lab1a.

Notes:

1. Be sure to review the Lab2a and Lab2b material, which deals with creating Linux kernel modules and task_struct data structures in Linux.
2. Please see the textbook (page 160, "Part II—Iterating over Tasks with a Depth-First Search Tree") for helpful hints and further explanation.

Deliverables:

1. **q1.c** - You are to provide your solution as a single C program named q1.c that contains your solution for this question.
   It is important for you to name your file q1.c as the TA grading this question has a makefile using this name to test your code.

2. **q1output.txt** - You are also to provide the output of **`dmesg`** command in a text file called **`q1output.txt`**. The output should show that the kernel module was loaded into the kernel, and the name, state, and `pid` of all iterated tasks in the system using DFS tree.

[15 points] Question 2: <u>UNIX Shell and History Feature</u>

This question consists of designing a C program to serve as a shell interface that accepts user commands and then executes each command in a separate process.

A shell interface gives the user a prompt, after which the next command is entered. The example below illustrates the prompt **`osh>`** and the user's next command: **`cat prog.c.`** The UNIX/Linux **`cat`** command displays the contents of the file **`prog.c`** on the terminal using the UNIX/Linux cat command and your program needs to do the same.

```
osh> cat prog.c
```

The above can be achieved by running your shell interface as a parent process. Every time a command is entered, you create a child process by using **`fork(),`** which then executes the user's command using one of the system calls in the exec() family (as described in Section 3.3.1). A C program that provides the general operations of a command-line shell can be seen in Fig 3.36 (page 158 of the textbook). The **`main()`** function presents the prompt `osh->` and outlines the steps to be taken after input from the user has been read. The `main()` function continually loops as long as `should_run` equals 1; when the user enters exit at the prompt, your program will set `should_run` to 0 and terminate.

This question is organized into two parts:

[10 points] <u>Creating the child process and executing the command in the child</u>

Your shell interface need to handle the following two cases.

1. **Parent waits while the child process executes.**

In this case, the parent process first read what the user enters on the command line (in this case, **`cat prog.c`**), and then creates a separate child process that executes the command. Unless otherwise specified, the parent process waits for the child to exit before continuing. This is similar in functionality to the new process creation illustrated in Figure 3.10.

2. **Parent executes in the background or concurrently while the child process executes (similar to UNIX/Linux)**

   To distinguish this case from the first one, add an ampersand (&) at the end of the command. Thus, if we rewrite the above command as **`osh> cat prog.c &`** the parent and child processes will run concurrently.

[10 points**]** **<u>Modifying the shell to allow a history feature</u>**

In this part your shell interface program should provide a ***history*** feature that allows the user to access the most recently entered commands. The user will be able to access up to 5 commands by using the feature. The commands will be consecutively numbered starting at 1, and the numbering will continue past 5. For example, if the user has entered 35 commands, the 5 most recent commands will be numbered 31 to 35. The user will be able to list the command history by entering the command

`osh> history`

As an example, assume that the history consists of the commands (from most to least recent): `ls -l, top, ps, who, date`. The command history should output:

```
5 ls -l
4 top
3 ps
2 who
1 date
```

Your program should support two techniques for retrieving commands from the command history:

1. When the user enters **!!**, the most recent command in the history is executed. In the example above, if the user entered the command:

   `Osh> !!`

   The '`ls` **`-l`**' command should be executed and echoed on user's screen. The command should also be placed in the history buffer as the next command.

2. When the user enters a single **!** followed by an integer *N*, the *Nth* command in the history is executed. In the example above, if the user entered the command:

   Osh> ! 3

   The '**ps**' command should be executed and echoed on the user's screen. The command should also be placed in the history buffer as the next command.

Error handling:

The program should also manage basic error handling. For example, if there are no commands in the history, entering !! should result in a message "No commands in history." Also, if there is no command corresponding to the number entered with the single !, the program should output "No such command in history."

Deliverables:

1. **q2.c** - You are to provide your solution as a single C program named q2.c that contains your solution for this question.

[10 points] Question 3: <u>Thread Synchronization</u>

Write a multithreaded C program that prints numbers from 1 to 100. In particular, your main program should create 2 separate threads (Thread 1 and Thread 2) using `pthread_Create()` function. Thread 1 should print odd numbers on the console with exactly one number on each line, for example:
1
3
5
7
and so on..

Thread 2 should print even numbers on the console, again with 1 number on each line

Example:
2
4
6
8
and so on.

Your main program waits for both threads to finish printing numbers. After both threads finish executing, the control is returned to main which simply prints the following line and exits:

"Finished printing numbers 1- 100."

Observe that when your solution uses only threads to print numbers from 1 - 100, the numbers are not necessarily in an order. To ensure that your output is in order, you need to use thread synchronization techniques. In particular, your solution should use mutex locks and semaphores to ensure the output is in order.

Notes:

1. Since the numbers begin from 1, Thread 1 needs to run first. After Thread 1 prints an odd number, i.e., 1 it should inform Thread 2 to print and wait till Thread 2 prints a number. After Thread 2 prints a number it informs Thread 1 to print and waits till Thread 1 prints a number. Thus the two threads alternate to print numbers on the console in order.
2. See practice labs 3a/3b to create threads using Pthreads API.
3. See practice labs 4a/5a to use mutex locks provided by the Pthreads API
4. See practice labs 5b/6a to use semaphores provided by the POSIX SEM extension.

Deliverables:

1. q3.c - You are to provide your solution as a single C program named q3.c that contains your solution for this question.