Computer Science Practice and Experience: Operating Systems (Comp Sci 3SH3), Term 2, Winter 2022
Prof. Neerja Mhaskar

Assignment 1 [40 points]

Due by 11:59pm on February 10th, 2022

- **No late assignment accepted.**
- It is advisable to start your assignment early.
- Make sure to submit a version of your assignment ahead of time to avoid last minute uploading issues.
- Note that students/groups copying each other's solution will get a zero.
- The assignment should be submitted on Avenue under Assessments -> Assignments -> Assignment 1 -> [Assignment Group #] folder.
- In your C programs, you should follow good programming style, which includes providing instructive comments and well-indented code.

[10 points] Question 1: This question involves designing a kernel module.

The Linux kernel keeps track of the global variable `jiffies`, which maintains the number of timer interrupts that have occurred since the system was booted. The `jiffies` variable is declared in the file `<linux/jiffies.h>`.

Design a kernel module that creates a `/proc` file named `/proc/jiffies` that reports the current value of `jiffies` when the `/proc/jiffies` file is read, such as with the command

```
cat /proc/jiffies
```

Be sure to remove `/proc/jiffies` when the module is removed.

This question should be completed using the Linux virtual machine you installed as part of Lab1.

Deliverables:

1. **jiffies.c** - You are to provide your solution as a single C program named `jiffies.c` that contains the entire solution for question 1.

   It is important for you to name your file jiffies.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, the current value of `jiffies`, and final that the kernel module was removed.

[20 points] Question 2: UNIX Shell and History Feature

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 Chapter 3). A C program that provides the general operations of a command-line shell can be seen below.

```
#include <stdio.h>
#include <unistd.h>

#define MAX_LINE 80 /* The maximum length command */

int main(void)
{
char *args[MAX_LINE/2 + 1]; /* command line arguments */
int should_run = 1; /* flag to determine when to exit program */

while (should_run) {
printf("osh>");
fflush(stdout);

/**
* After reading user input, the steps are:
* (1) fork a child process using fork()
* (2) the child process will invoke execvp()
* (3) parent will invoke wait() unless command included &
*/
}
return 0;
}
```

Figure 3.32 Outline of simple shell.

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] Creating the child process and executing the command in the child

Your shell interface needs to handle the following two cases.

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

In this case, the parent process first reads 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 Lab 3.

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] Modifying the shell to allow a history feature

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

[10 points] Question 3—Dining Philosophers Problem

In the Lecture Slides on Chapters 6 and 7 an outline of a solution to the dining-philosophers problem using monitors is given. For this question you will be implementing a solution to the dining-philosophers problem using `POSIX` mutex locks and `POSIX` condition variables. Your solution should be based on the algorithm illustrated in Figure 7.7 of the textbook and given at the end of the question for your reference.

Your implementation will require creating **five** philosophers, each identified by a number 0 . . 4. Each philosopher will run as a separate thread. Create threads using Pthreads as discussed in the Lecture slides on Chapter 4 and Practice Lab on Threads. Philosophers alternate between thinking and eating. To simulate both activities, have each thread sleep for a random period between one and three seconds.

When a philosopher wishes to eat, she invokes the function

`pickup_forks(int philosopher_number)`

where `philosopher_number` identifies the number of the philosopher wishing to eat. When a philosopher finishes eating, she invokes

`return_forks(int philosopher_number)`

Deliverables:

1. **q3.c** - You are to provide a C file named q3.c that contains the entire solution for question 3.
2. **q3output3.txt** - You are also to provide a text file called output3.txt that contains two test cases (of your choice) and your solution's output for these test cases.

```
monitor DiningPhilosophers
{
  enum {THINKING, HUNGRY, EATING} state[5];
  condition self[5];

  void pickup(int i) {
    state[i] = HUNGRY;
    test(i);
    if (state[i] != EATING)
      self[i].wait();
  }

  void putdown(int i) {
    state[i] = THINKING;
    test((i + 4) % 5);
    test((i + 1) % 5);
  }

  void test(int i) {
    if ((state[(i + 4) % 5] != EATING) &&
     (state[i] == HUNGRY) &&
     (state[(i + 1) % 5] != EATING)) {
       state[i] = EATING;
       self[i].signal();
    }
  }

  initialization_code() {
    for (int i = 0; i < 5; i++)
      state[i] = THINKING;
   }
}
```

Figure 7.7 A monitor solution to the dining-philosophers problem.