Operating Systems (SFWRENG 3SH3), Term 2, Winter 2023
Prof. Neerja Mhaskar

Assignment 1 [40 points]

Due by 11:59pm on February 15th, 2023

- **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 I -> [Group #] folder.
- In your C programs, you should follow good programming style, which includes providing instructive comments and well-indented code. If this is not followed, marks will be deducted.
- **If working in a group of two, a `Readme` file containing information on who did what should be provided.**

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

Design a kernel module that creates a `/proc` file named `/proc/seconds` that reports the number of elapsed seconds since the kernel module was loaded. This will involve using the value of jiffies as well as the HZ rate. When a user enters the command

```
cat /proc/seconds
```

your kernel module will report the number of seconds that have elapsed since the kernel module was first loaded. Be sure to remove `/proc/seconds` when the module is removed.

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

Some useful information: The `/proc` file system is a "pseudo" file system that exists only in kernel memory and is used primarily for querying various kernel and per-process statistics. Furthermore, 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>`.

Deliverables:

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

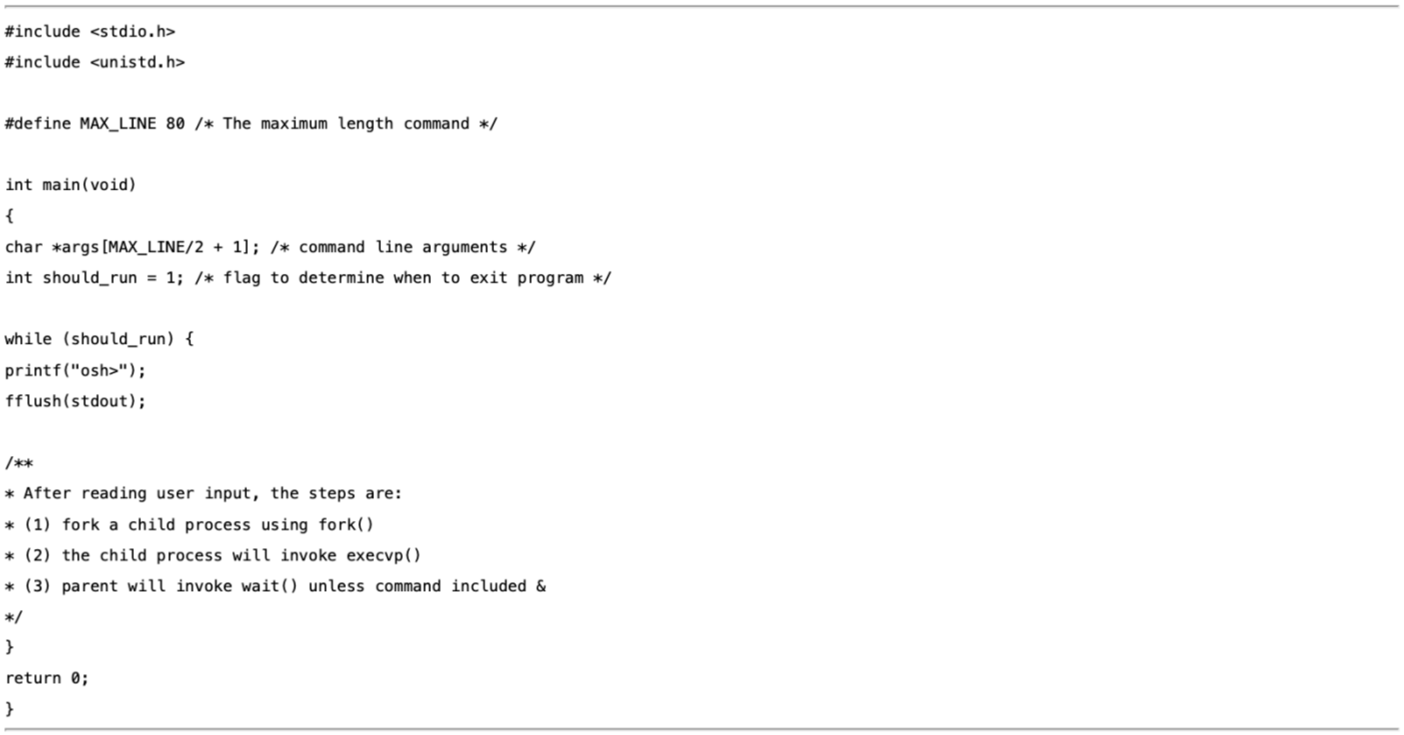
It is important that name your C file seconds.c as the TA grading this question has a Makefile using this name to test your code.

[15 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. 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:

[5 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 Practice 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 **with arguments**. The user will be able to access up to 5 commands (with its arguments) 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 with arguments (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 the following technique for retrieving command from the command history: When the user enters **!!**, the most recent command (with its arguments) 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.

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."

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. Please make sure to name your solution with the correct file name for grading purposes.

[10 points] Question 3—The Sleeping Teaching Assistant.

A university computer science department has a teaching assistant (TA) who helps undergraduate students with their programming assignments during regular office hours. The TA's office is rather small and has room for only one desk with a chair and computer. There are three chairs in the hallway outside the office where students can sit and wait if the TA is currently helping another student. When there are no students who need help during office hours, the TA sits at the desk and takes a nap. If a student arrives during office hours and finds the TA sleeping, the student must awaken the TA to ask for help. If a student arrives and finds the TA currently helping another student, the student sits on one of the chairs in the hallway and waits. If no chairs are available, the student will come back at a later time.

Using POSIX threads, mutex locks, and semaphores, implement a solution that coordinates the activities of the TA and the students. Details for this assignment are provided below.

Using `Pthreads`, begin by creating n students where each student will run as a separate thread. The TA will run as a separate thread as well. Student threads will alternate between programming for a period of time and seeking help from the TA. If the TA is available, they will obtain help. Otherwise, they will either sit in a chair in the hallway or, if no chairs are available, will resume programming and will seek help at a later time. If a student arrives and notices that the TA is sleeping, the student must notify the TA using a semaphore. When the TA finishes helping a student, the TA must check to see if there are students waiting for help in the hallway. If so, the TA must help each of these students in turn. If no students are present, the TA may return to napping.

Perhaps the best option for simulating students programming—as well as the TA providing help to a student—is to have the appropriate threads sleep for a random period of time.

Refer to the practice labs on the use of POSIX mutex locks and semaphores.

Deliverables:

1. **q3.c** - You are to provide a C file named q3.c that contains the entire solution for question 3. Please make sure to name your solution with the correct file name for grading purposes.