Programming Assignment 2
Due: October 9, 2019 by 11:55pm

The assignment consists of two projects:

Project 1 —UNIX Shell and History Feature

This project 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. This project can be completed on any Linux or UNIX system (preferably OSC). 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 . (This command displays the file prog.c on the terminal using the UNIX cat command.)

osh> cat prog.c

One technique for implementing a shell interface is to have the parent process first read what the user enters on the command line (in this case, cat prog.c), and then create a separate child process that performs 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 (of the Text Book). However, UNIX shells typically also allow the child process to run in the background, or concurrently. To accomplish this, we 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.

The separate child process is created using the fork() system call, and the user's command is executed using one of the system calls in the exec() family (as described in Section 3.3.1 of the Text Book).

A C program that provides the general operations of a command-line shell is supplied in Figure 3.36 (of the Text Book). 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. The skeleton file is also present in OSS VM, Chapter 3, named simple-shell.c.

This project is organized into two parts:
(1) Creating the child process and executing the command in the child, and
(2) Modifying the shell to allow a history feature

```
#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) if command included &, parent will NOT invoke wait()
*/
  }
return 0;

}
```

Part I— Creating a Child Process

The first task is to modify the main() function in Figure 3.36 so that a child process is forked and executes the command specified by the user. This will require parsing what the user has entered into separate tokens and storing the tokens in an array of character strings (args in Figure 3.36). For example, if the user enters the command ***ps -ael*** at the osh> prompt, the values stored in the args array are:

```
args[0] = "ps"
args[1] = "-ael"
args[2] = NULL
```

This args array will be passed to the execvp() function, which has the following prototype: execvp(char *command, char *params[]);

Here, command represents the command to be performed and params stores the parameters to this command. For this project, the execvp() function should be invoked as execvp(args[0], args). Be sure to check whether the user included an & to determine whether or not the parent process is to wait for the child to exit.

Part II—Creating a History Feature

The next task is to modify the shell interface program so that it provides a history feature that allows the user to access the most recently entered commands. The user will be able to access up to 10 commands by using the feature. The commands will be consecutively numbered starting at 1, and the numbering will continue past 10. For example, if the user has entered 35 commands, the 10 most recent commands will be numbered 26 to 35.

The user will be able to list the command history by entering the command history at the prompt

osh> history

As an example, assume that the history consists of the commands (from most to least recent): ps, ls -l, top, cal, who, date

The command history will output:

ps
ls -l
top
cal
who
date

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

When the user enters **!!** , the most recent command in the history is executed.
When the user enters a single **!** followed by an integer N, the N th command in the history is executed starting from 0.

Continuing our example from above, if the user enters !! , the ps command will be performed; if the user enters **!3** , the command cal will be executed. Any command executed in this fashion should be echoed on the user's screen. The command should also be placed in the history buffer as the next command.

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

Project 2 —Linux Kernel Module for Listing Tasks

In this project, you will write a kernel module that lists all current tasks in a Linux system. Be sure to review the programming project in Chapter 2, which deals with creating Linux kernel modules, before you begin this project. The project can be completed using the Linux virtual machine provided with this text

Part I—Iterating over Tasks Linearly

As illustrated in Section 3.1, the PCB in Linux is represented by the structure task struct , which is found in the <linux/sched.h> include file. In Linux, the for each process() macro easily allows iteration over all current tasks in the system:

```
#include <linux/sched.h>
struct task struct *task;
        for each process(task) {
        /       * on each iteration task points to the next task */
        }
```

The various fields in task struct can then be displayed as the program loops through the for each process() macro.

TASK:

Design a kernel module that iterates through all tasks in the system using the for each process() macro. In particular, output the task name (known as executable name), state, and process id of each task. (You will probably have to read through the task struct structure in <linux/sched.h> to obtain the names of these fields.) Write this code in the module entry point so that its contents will appear in the kernel log buffer, which can be viewed using the dmesg command. To verify that your code is working correctly, compare the contents of the kernel log buffer with the output of the following command, which lists all tasks in the system:

```
ps -el
```

The two values should be very similar. Because tasks are dynamic, however, it is possible that a few tasks may appear in one listing but not the other.

Part II—Iterating over Tasks with a Depth-First Search Tree

The second portion of this project involves iterating over all tasks in the system using a **Depth-first search (DFS)** tree.

Linux maintains its process tree as a series of lists. Examining the task_struct in <linux/sched.h>, we see two struct list head objects:

children

and

sibling

These objects are pointers to a list of the task's children, as well as its siblings. Linux also maintains references to the init task (struct task_struct init_task). Using this information as well as macro operations on lists, we can iterate over the children of init as follows:

```
struct task struct *task;
struct list head *list;

list_for_each(list, &init_task->children) {
        task = list_entry(list, struct task_struct, sibling);
        /* task points to the next child in the list */
}
```

The list_for_each() macro is passed two parameters, both of type struct list head :

- A pointer to the head of the list to be traversed
- A pointer to the head node of the list to be traversed

At each iteration of list_for_each() , the first parameter is set to the list structure of the next child. We then use this value to obtain each structure in the list using the list_entry() macro.

TASK:

Beginning from the init task, design a kernel module that iterates over all tasks in the system using a DFS tree. Just as in the first part of this project, 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 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.