Assignment #2 – Shell

Due date: Feb 20, 2023 at 11:59 pm

Description: Generally, a shell is a program that executes other programs in response to text commands. A sophisticated shell can also change the environment in which other programs execute by passing named variables, a parameter list, or an input source. For this assignment, you are going to design and implement a C program to serve as an oversimplified shell interface that accepts user commands and then executes each command in a separate process. This assignment may use the `fork()`, `exec()`, `wait()`, and other system calls and can be completed on any Linux, UNIX, or macOS system.

UNIX Shell Overview: 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.

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 `exec()` system calls.

Figure 1 shows the pseudo-code of a C program that provides the general operations of a command-line 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, the program will set `should_run` to 0 and terminate.

```c
#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 1 Pseudo-code of a shell program in C

---should have more functions ----

* Use gets()
* ask for args until arg[n] = NULL

Required Tasks I: The first task is to complete the code in Figure 1 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 1). 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
```

← user input is finished

This `args` array will be passed to the `execvp()` function, which has the following prototype: `execvp(char *command, char *params[])`, where the `command` represents the command to be performed, and `params` store the parameters of this command. For this project, the `execvp()` function should be invoked as `execvp(args[0], args)`.

Required Tasks II: UNIX shells also allow the child process to run in the background or concurrently by adding an ampersand (`&`) at the end of the command. The second task is to allow `execvp()` to run in the background (i.e., concurrently with the parent). To achieve this, be sure to check whether the user included `&` to determine whether the parent process is to wait for the child to exit.

Required Tasks III: Typically, the UNIX shell provides a history feature to allow a user to execute the most recent command by entering `!!`. For example, if a user enters the command `ls -l`, the user may execute the command again by entering `!!` at the prompt. Any command executed in this fashion should be echoed on the user's screen, and the command should also be placed in the history buffer as the next command. The program should also manage basic error handling. If there is no recent command in the history, entering `!!` should result in a message "`No commands in history.`"

Submission Instruction:

Please submit two files through Blackboard.

1. Name your C program *myshell.c* file and submit it in Blackboard.
2. Make a video recording to quickly demonstrate your shell, and submit the recording in Blackboard.

Rubric:

- Comments in code 5 points
- Task I 50 points
- Task II 15 points
- Task III 15 points
- Video demonstration 15 points