PROGRAMMING ASSIGNMENT #2
Microshell

CSCI 480 100 points Spring 2022

Check Blackboard for due date. Due by 11:59 PM.
Late penalty (and early submission credit) as in syllabus.

Specifications

This project consists of designing a C/C++ program to serve as a shell interface that accepts user commands and then executes each command in a separate process. Your implementation will support input and output redirection. Completing this project will involve using the fork(), exec(), wait(), dup2() system calls.

This assignment has several components and is bigger than assignment 1. Read the writeup and the programming hints document carefully. *Please start early and allocate enough time for this assignment.*

A shell interface gives the user a prompt, after which the next command is entered. The example below illustrates the prompt mysh> and the user's next command: cat prog.c. (This command displays the file prog.c on the terminal using the LINUX cat command.)

```
mysh>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. The parent process waits for the child to exit before continuing.

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 (Do not use *system()* in the assignment).

Below is the outline of the main() function of a simple microshell, which presents the prompt and then take the steps based on the input from the user has been read. Read "Programming Hints" for example code.

```
int main(void)
{
  //declare variables such as arrays for command line arguments.
  //A main loop
  {
        //print the prompt
        //read the user input
        //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()
  }
  exit(0);
}
```

This project is organized into several parts:

1. Creating the child process and executing the command in the child
2. Providing a history feature
3. Adding support of input and output redirection

Tackle the problem step by step. First of all make sure that your shell is taking inputs correctly. Then test the execution of various commands. Then add the history feature. Then add the implementation of output redirection and input redirection.

Executing Command in a Child Process

The first task is to modify the main() function in the outline 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. For example, if the user enters the command ps -ael at the mysh> 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).

Creating a History Feature

The next task is to modify the shell interface program so that it 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, she can then execute that 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.

If the user enters command “history”, your shell will display all saved commands. You can use the C++ STL vector to implement this “history” function, in which case you do not need to worry about the maximum number of commands saved in history. If you implement using an array, store up to 10 commands.

Your 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.”

Your shell is *not* required to support the function of running a particular command in the history (e.g. !5).

Redirecting Input and Output

Your shell should then be modified to support the ‘>’ and ‘<’ redirection operators, where ‘>’ redirects the output of a command to a file and ‘<’ redirects the input to a command from a file. For example, if a user enters

```
mysh>ls > out.txt
```

the output from the ls command will be redirected to the file out.txt. Similarly, input can be redirected as well. For example, if the user enters

```
mysh>sort < in.txt
```

the file in.txt will serve as input to the sort command.

Managing the redirection of both input and output will involve using the dup2() function, which duplicates an existing file descriptor to another file descriptor. For example, if fd is a file descriptor to the file out.txt, the call

```
dup2(fd, STDOUT_FILENO);
```

duplicates fd to standard output (the terminal). This means that any writes to standard output will in fact be sent to the out.txt file.

You can assume that commands will contain either one input or one output redirection and will not contain both. In other words, you do not have to be concerned with command sequences such as sort < in.txt > out.txt. You can assume that there is a whitespace before and after the redirection sign. Ideally (not required), your shell allows optional whitespace before or after the redirection sign.

Test Cases

Your shell will be tested by these basic test cases during grading. TA may test more.

- Print a prompt "mysh>" and wait for your input; (test case 1)
- The shell understands the command "quit" or "q", as the special commands to exit. (test case 2)
- Execute the command you type in after the prompt (e.g. ls, ps, ls -l) and print a new prompt. If the user enters a return without input command, the shell repeats the prompt. (test case 3 and 3b)
- The shell understands command "history", which prints out the recent commands in the history buffer. (test case 4)
- The shell understands !! which will run the last valid command. If there is no recent command in the history, entering !! should result in a message "No commands in history." (test case 5 and 5b)
- The shell understands output redirection using '>'. (test case 6)
- The shell understands input redirection using '<'. (test case 7)

Example Output:

turing%>./ **z1234567_project2**
mysh>quit
turing%>./ **z1234567_project2**
mysh>ps
…
mysh>ls -l
…
mysh>history
1 ps
2 ls -l
3 history
mysh>!!
…
mysh>ls –l > output.txt
mysh>ps > output.txt
mysh> sort < output.txt
…
mysh>hello
Couldn't execute the command hello

Requirements:

The programs should 1) work according to the specifications; 2) be comprehensible and well commented; 3) check the return value of the system calls and have proper error handling.

Submission:

Submission requirement is the same as the 1st assignment. Note that the directory must be called: "z1234567_project2_dir", all in lowercase. Important steps are repeated below:

You will submit a compressed file through Blackboard, using the "Assignment" functionality.

The compressed file contains your source code and a Makefile. It should be named as "your-zid_project2.tar" and must be created following the procedure described below:

1. Put all your source code files (NO OBJECT or EXECUTABLE FILES) and your Makefile in a directory called "your-zid_project2_dir". Example: z1234567_project2_dir. **Note**: 'z' must be in lower case.

   In your Makefile, you need to make sure your compilation produces the executable file called "your-zid_project2". For a student with z1234567 as her zid, the executable would be *z1234567_project2*.

2. In the parent directory of your-zid_project2_dir, compress this whole subdirectory by the following command:

   tar –cvvf your-zid_project2.tar your-zid_project2_dir

   Example:

   tar –cvvf z1234567_project2.tar z1234567_project2_dir

"your-zid_project2.tar" is now the compressed file containing the whole subdirectory of your files. You can then transfer (e.g. using an ftp client) the tar file from turing (or hopper) to a computer on which you can open a web browser for your final submission to the Blackboard system.