CSE 3033 - OPERATING SYSTEMS
Programming Assignment # 2
DUE DATE: 19/12/2022 - 23:59PM

This programming assignment is related with writing a simple shell by considering the outline program given in the lab sessions.

The main() function of your program presents the command line prompt "**myshell: "** and then invokes setup() function which waits for the user to enter a command. The *setup function* (given in the outline program of your textbook) reads the user's next command and parses it into separate tokens that are used to fill the argument vector for the command to be executed. It can also detect background processes. This program is terminated when the user enters **^D (<CONTROL><D>);** and setup function then invokes exit. The contents of the command entered by the user is loaded into the *args* array. You may assume that a line of input will contain no more than 128 characters or more than 32 distinct arguments.

Necessary functionalities and components of your shell is listed below:

A. It will take the command as input and will execute that in a new process. When your program gets the program name, it will create a new process using **`fork()`** system call, and the new process (child) will execute the program. The child will use the `execl()` function in the below to execute a new program.

- Use **`execv()`** instead of **`execvp(),`** which means that you will have to read the **PATH** environment variable, then search each directory in the **PATH** for the command file name that appears on the command line.

- **<u>Important Notes:</u>**

  1. Using the "`system()`" function is not allowed for part A!

  2. In the project, you need to handle foreground and background processes. When a process run in foreground, your shell should wait for the task to complete, then immediately prompt the user for another command.

```
myshell: gedit
```

A background process is indicated by placing an ampersand ('&') character at the end of an input line. When a process run in background, your shell should not wait for the task to complete, but immediately prompt the user for another command.

```
myshell: gedit &
```

With background processes, you will need to modify your use of the wait() system call so that you check the process id that it returns.

B. It must support the following internal (*built-in*) commands. *Note that an internal command is the one for which no new process is created but instead the functionality is built directly into the shell itself.*

- **`history –`** seethe list of and execute the last 10 commands. See the following example for the use of these commands.

Example:

```
myshell> history
       0 ps
       1 ls
       2 ls -l
       3 who
       4 ps -a
       5 chmod 777 a.txt
       6 ls | wc
       7 ps – ef
       8 ps
       9 ls -al
myshell> ls /bin
myshell> history
       0 ls /bin
       1 ps
       2 ls
       3 ls -l
       4 who
       5 ps -a
       6 chmod 777 a.txt
       7 ls | wc
       8 ps – ef
       9 ps
myshell> history -i 9
       PID TTY          TIME CMD
       6052 pts/0    00:00:00 ps
myshell> history
       0 ps
       1 ls /bin
       2 ps
       3 ls
       4 ls -l
       5 who
       6 ps -a
       7 chmod 777 a.txt
       8 ls | wc
       9 ps – ef
```

In the first line, after issuing **`history`** command, the lastly executed 10 commands are printed on th screen. After executing **`ls /bin`**, you can see that the history is updated. With **`history -i num`**, we can execute the command at num index. After executing the command at some index, the history table is updated again.

To implement this part, you cannot use history command of Linux. You have to implement the functionality by yourself, creating data structures to store commands as necessary. When we want to execute a command at an index, execute it using the A part of the homework.

- **^Z** - Stop the currently running foreground process, as well as any descendants of that process (e.g., any child processes that it forked). If there is no foreground process, then the signal should have no effect.

- **fg %num** - Move the background process with process id num to the forground. Note that for this, you have to keep track of all the background processes.

- **exit -** Terminate your shell process. If the user chooses to exit while there are background processes, notify the user that there are background processes still running and do not terminate the shell process unless the user terminates all background processes.

C. I/O Redirection

The shell must support I/O-redirection on either or both *stdin* and/or *stdout* and it can include arguments as well. For example, if you have the following commands at the command line:

- **myshell: myprog [args] > file.out**
  Writes the standard output of **myprog** to the file **file.out.** **file.out** is created if it does not exist and truncated if it does.

- **myshell: myprog [args] >> file.out**
  Appends the standard output of **myprog** to the file **file.out.** **file.out** is created if it does not exist and appended to if it does.

- **myshell: myprog [args] < file.in**
  Uses the contents of the file **file.in** as the standard input to program **myprog**.

- **myshell: myprog [args] 2> file.out**
  Writes the standard error of **myprog** to the file **file.out**.

- **myshell: myprog [args] < file.in > file.out**
  Executes the command **myprog** which will read input from **file.in** and stdout of the command is directed to the file **file.out**

Notes:

- You should use the skeleton program provided as a starting point for your implementation. The skeleton program reads the next command line, parses and separates it into distinct arguments using blanks as delimiters. You will implement the action that needs to be taken based on the command and its arguments entered to ***myshell***. Feel free to modify the command line parser as you wish.
- You can assume that all command line arguments will be delimited from other command line arguments by white space – one or more spaces and/or tabs.
- For this project, the error messages should be printed to **stderr.**
- Take into account materials and examples covered in the lab sessions. As a starting point, you can consider the example programs given in course web site.

- Consider all necessary error checking for the programs.
- No late homework will be accepted!
- In case of any form of **copying** and **cheating** on solutions, all parties/groups will get ZERO grade. You should submit your own work.
- You have to work in groups of two or three.

What to submit?

A softcopy of your *source codes* which are extensively commented and appropriately structured and a minimum 3-page report should be emailed to **cse333.projects@gmail.com** in a zip file. Make sure that your zip file name contains student IDs (Student#1_Student#2_Project2.zip)!