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 ***`execl()`*** instead of ***`execlp(),`*** 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.

- **Important Notes:**

    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.*

- ***`alias/unalias`*** **–** set/remove an alias for a command. See the following example for the use of these commands.

  Example:

  ```
  myshell> alias “ls -l” list
  myshell> alias "ps -a” proc
  myshell> alias -l
          list "ls -l"
          proc "ps -a"
  myshell> proc
          PID TTY          TIME CMD
          6052 pts/0    00:00:00 ps
  myshell> unalias list
  myshell> alias -l
          proc "ps -a"
  ```

  In the first line, **`ls -l`** is set to have the alias **`list`,** and **`ps -a`** command set to have the alias **`ps -a`.** Using *-l* (lowercase letter L) lists all the aliases added to **myshell.** When one of the aliases added to the **myshell** is entered, the corresponding command has to be executed. To remove an alias from **myshell,** ***`unalias`*** command has to be used. When user removes an alias, it is removed from the aliases list and that alias cannot be used.

- ***`^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.

- ***`clr`*** - clear the screen.
  - You can use `system()` function to execute `clear` command.

- ***`fg`*** **-** Move all the background processes to foreground. 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`***

Bonus (10%): You will get extra credit if your shell supports pipe operator “|”. For a pipe in the command line, you need to take care of connecting stdout of the left command to stdin of the command following the "|". For example, if the user types "`ls -al | sort`", then the “`ls`” command is run with stdout directed to a Unix pipe, and that the `sort` command is run with stdin coming from that same pipe.