Assignment 2
Shell Part 1

Operating Systems and Networks

Monsoon 2021

Deadline: 15th September, 11:55 PM

There would be **NO** deadline extensions

Assignment 3 would be a ***direct continuation*** of this assignment

TOTAL MARKS: 100

Task:

Implement a shell that supports a **semi-colon** separated list of commands. Use ***'strtok'*** to tokenize the command. Also, support ***'&'*** operator which lets a program run in the background after printing the process id of the newly created process. Write this code in a **modular fashion**. In the next assignment, you will add more features to your shell.

The goal of the assignment is to create a user-defined interactive shell program that can create and manage new processes. The shell should be able to create a process out of a system program like emacs, vi, or any user-defined executable

The following are the specifications for the assignment. For each of the requirements, an appropriate example is given along with it.

Specification 1: Display requirement [10 marks]

When you execute your code, a shell prompt of the following form must appear along with it. Do *NOT* hard-code the user name and system name here.

<username@system_name:curr_dir>

Example:

```
<Name@UBUNTU:~>
```

The directory from which the shell is invoked will be the home directory of the shell and should be indicated by **"~"**. If the user executes "cd" i.e changes the directory, then the corresponding change must be reflected in the shell as well.

Example:

```
./a.out
<Name@UBUNTU:~> cd newdir/test
<Name@UBUNTU:~/newdir/test> cd ../..
<Name@UBUNTU:~>
```

If your current working directory is the directory from which your shell is invoked, then on executing command "cd .." your shell should display the absolute path of the current directory from the root.
Example: If your shell executable resides in '/home/user/shell' then on executing 'cd ..', the command prompt should display '/home/user' as the cwd.

Note:
Your shell **should** account for random spaces and tabs.
Example:

```
./a.out
<Name@UBUNTU:~> vim        &
<Name@UBUNTU:~/newdir/test> cd         ..
<Name@UBUNTU:~> ls       -a       -l   .   ;   cd       test
<Name@UBUNTU:~/test>
```

Any such commands should work. This is NOT hard to implement, just tokenize the input string appropriately.

Specification 2: Builtin commands [25 marks]

[cd - 15 marks, echo - 5 marks, pwd - 5 marks]

Builtin commands are contained within the shell itself. Checkout 'type command-name' in the terminal (eg. 'type echo'). When the name of a built-in command is used as the first word of a simple command the shell executes the command directly, without invoking another program. Builtin commands are necessary to implement functionality impossible or inconvenient to obtain with separate utilities.
Make sure you implement **cd**, **pwd**, and **echo**. *DON'T* use *'execvp'* or similar commands for implementing these commands.

Note :

- For echo, handling multi-line strings and environmental variables is not a requirement
- You do not need to handle escape flags and quotes. You can print the string as it is. However, you must handle tabs and spaces

  Example:

  ```
  <Name@UBUNTU:~> echo "abc      'ac'          abc"
  "abc 'ac' abc"
  ```
- For cd apart from the basic functionality, implement the flags "**.**", "**..**", "**-**" and "**~**".
- It is an error for a cd command to have more than one command-line argument. If no argument is present then you must cd into the home directory.

Specification 3: ls command [15 marks]

Implement the ***ls*** command with its two flags *"-a"* and *"-l"*. You should be able to handle all the following cases also:

- ls
- ls -a
- ls -l
- ls .
- ls ..
- ls ~
- ls -a -l
- ls -la / ls -al
- ls <Directory/File_name>
- ls -<flags> <Directory/File_name>

Example:

```
<Name@UBUNTU:~> ls -al test_dir
```

Note:

- For ls, ls -a and ls <Directory_name> outputting the entries in a single column is fine.
- You can assume that the directory name would not contain any spaces.
- For the "**-l**", kindly, print the **exact** same content as displayed by your actual Shell. You can leave out the colors but the content should be the same.
- ***DON'T*** use *'execvp'* or similar commands for implementing this.
- **Multiple** flags and directory names can be tested. Your shell should also account for these arguments in any order.

Example:

```
<Name@UBUNTU:~> ls -l <dir_1> -a <dir_2>
<Name@UBUNTU:~> ls -la <dir_1> <dir_2>
```

Specification 4: System commands with and without arguments [20 marks]

All other commands are treated as system commands like emacs, vi, and so on. The shell must be able to execute them either in the background or in the foreground.

Foreground processes: For example, executing a "vi" command in the foreground implies that your shell will wait for this process to complete and regain control when this process exits. **[10 marks]**

Background processes: Any command invoked with **"&"** is treated as a background command. This implies that your shell will spawn that process and doesn't wait for the process to exit. It will keep taking other user commands. Whenever a new background process is started, print the PID of the newly created background process on your shell also. **[10 marks]**

Example:

```
<Name@UBUNTU:~> gedit &
456
<Name@UBUNTU:~> ls -l -a
        .
        .
        . Execute other commands
        .
        .
<Name@UBUNTU:~> echo hello
```

Note:

- You do *NOT* have to handle background processing for built-in commands (ls, echo, cd, pwd, pinfo). Commands not implemented by you should be runnable in the background.
- Your shell should be able to run multiple background processes, and not just one. Running pinfo on each of these should work as well.

Specification 5: pinfo command (user-defined) [15 marks]

-***pinfo***: This prints the process-related info of your shell program. Use of ***"popen()"*** for implementing *pinfo* is NOT allowed

Example:

```
<Name@UBUNTU:~>pinfo
pid -- 231
Process Status -- {R/S/S+/Z}
memory -- 67854 {Virtual Memory}
Executable Path -- ~/a.out
```

-***pinfo <pid>***: This prints the process info about the given PID.

Example:

```
<Name@UBUNTU:~>pinfo 7777
pid -- 7777
Process Status -- {R/S/S+/Z}
memory -- 123456 {Virtual Memory}
Executable Path -- /usr/bin/gcc
```

Process status codes:

1. R/R+: Running
2. S/S+: Sleeping in an interruptible wait
3. Z: Zombie
4. T: Stopped (on a signal)

Note: "+" must be added to the status code if the process is in the foreground

Specification 6: Finished Background Processes [10 marks]

If the background process exits then the shell must display the appropriate message to the user.

Example:

After gedit exits, your shell program should check the exit status and print it on stderr.

```
<Name@UBUNTU:~> gedit &
<Name@UBUNTU:~> cd test
<Name@UBUNTU:~/test>
gedit with pid 456 exited normally/abnormally
<Name@UBUNTU:~/test>
```

Specification 7: repeat Command [5 marks]

Implement the ***repeat*** command. The command is responsible for executing the given instruction multiple times. The first argument to the command specifies the number of times the following command is to be run. This would ***NOT*** be tested with background processes.

Example:

```
<Name@UBUNTU:~/newdir/test> repeat 2 cd ..
<Name@UBUNTU:~> repeat 3 echo hello
hello
hello
hello
<Name@UBUNTU:~> repeat 2 sleep 4 ( Sleep for 4 seconds twice)
<Name@UBUNTU:~>
```

Bonus (Optional) [20 marks]

history: [5 marks]

Implement a *'history'* command which is similar to the actual history command. The maximum number of commands it should output is 10. The **maximum** number of commands your shell should store is 20. You must overwrite the oldest commands if more than 20 commands are entered. You should track the commands across all sessions and not just one.

Example:

```
<Name@UBUNTU:~> ls
<Name@UBUNTU:~> cd
<Name@UBUNTU:~> cd
<Name@UBUNTU:~> history
ls
cd
history
<Name@UBUNTU:~> exit
```

When you run the shell again

```
<Name@UBUNTU:~> history
ls
cd
history
exit
```

```
history
```

Extensions of the above command are:

a. - ***history <num>***: Display only latest <num> commands. **[5 marks]**

   Example:

   ```
   <Name@UBUNTU:~> ls
   <Name@UBUNTU:~> cd
   <Name@UBUNTU:~> ls
   <Name@UBUNTU:~> history
   ls
   cd
   ls
   history
   <Name@UBUNTU:~> history 3
   ls
   history
   history 3
   ```

   Note: <num> for this part can be more than 10 but will always be less than 20 which is the total number of commands your shell remembers

b. ***Up Arrow Key***: On clicking the *UP* arrow key, you must loop over the previous commands present in your shell's history and show them on the prompt. In case you reach the first command or have no history, then stay on the same command if *UP* is pressed. **[10 marks]**

   Example:

   ```
   <Name@UBUNTU:~> ls
   <Name@UBUNTU:~> cd
   <Name@UBUNTU:~> echo hello
   <Name@UBUNTU:~>
   ```

   Now if we press UP once, "echo hello" should be displayed as the command on the prompt. If we press UP again, your shell should show "cd". If we press UP again, show "ls". Now, as this is the last command in history, nothing will happen on pressing UP again and the shell will continue to display "ls".

Note:

- After a command is shown on the prompt using the UP arrow key, it can be modified. For Example, after getting "cd" in the above example, we can add a directory name in front of it to change it to "cd <dir>" and then **execute it** by pressing the Enter key.
- UP arrow key will *ONLY* be pressed when the prompt is empty i.e., no other input is written in the prompt

General notes

Useful commands/structs/files:

uname, hostname, signal, waitpid, getpid, kill, execvp, strtok, fork, getopt, readdir, opendir, readdir, closedir, getcwd, sleep, watch, struct stat, struct dirent, /proc/interrupts, fopen, chdir, getopt, pwd.h (to obtain username), /proc/loadavg, etc.

Type: man/man 2 <command_name> to learn of all possible invocations/variants of these general commands. Pay specific attention to the various data types used within these commands and explore the various header files needed to use these commands.

Guidelines:

1. The Assignment must ONLY be done in **C.** *NO* other languages are allowed.
2. If the command cannot be run or returns an error it should be handled appropriately. Look at ***"perror.h"*** for appropriate routines to handle errors.
3. You ***MUST*** do error handling for both user-defined and system commands.
4. Use of *system()* call is prohibited.
5. You can use both "*printf*" and "*scanf*" for this assignment.
6. The user can type the command anywhere in the command line i.e., by giving spaces, tabs, etc. Your shell should be able to handle such scenarios appropriately.
7. The user can type in any command, including, ./a.out, which starts a new process out of your shell program. In all cases, your shell program must be able to execute the command or show the error message if the command cannot be executed.
8. If the code doesn't compile, no marks will be rewarded.
9. Segmentation faults at the time of grading will be penalized.
10. Bonus is an ***optional*** task. It can cover the marks for only this assignment if you have lost any.
11. **Do NOT take codes from seniors or your batchmates, by any chance. We will extensively evaluate cheating scenarios along with the previous few year's submissions.**

12. **Viva will be conducted during the evaluations, related to your code and also the logic/concept involved. If you're unable to answer them, you'll get no marks for that feature/topic that you've implemented.**

Submission Guidelines

1. Upload format <**Roll-No>_Assignment2.tar.gz** . Make sure you write a **makefile** for compiling all your code (with appropriate flags and linker options)
   Kindly adhere to the following naming guidelines and directory structure:
   <Roll-No>_Assignment2
   ├── README.md
   ├── makefile
   └── Other files and Directories

2. Include a **README.md file** briefly describing your work and which file corresponds to what part. Including a README file is *NECESSARY.*