Assignment 2

Operating Systems and Networks, Monsoon 2022

Deadline: 5th September, 5 PM

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, gedit, or any user-defined executable.

The following are the specifications for the assignment

Specification 1: Display requirement [8 marks]

When you execute your code, a shell prompt of the following form must appear along with it. Do not hard-code the username and the 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.

Note:

- If your current working directory is the directory from which your shell is invoked, then on executing the command "cd .." your shell should display the absolute path of the current directory from the root.
- Your shell should support a ';' or '&' separated list of commands. Use ***'strtok'*** to tokenize the command.
- ***'&'*** operator run the command preceding it in the background after printing the process id of the newly created process.
- Your shell should also account for random spaces and tabs.

  Example:

  ```
  ./a.out
  <Name@UBUNTU:~> vim      &
  [1] 35006
  <Name@UBUNTU:~>sleep 5 & echo hello
  [2] 35036
  hello
  # sleep command runs in the background while echo runs in the foreground
  sleep with pid 35036 exited normally # after 5 seconds
  <Name@UBUNTU:~/newdir/test>   cd    ..
  <Name@UBUNTU:~> ls     -a  -l  . ;cd         test
  <Name@UBUNTU:~/test>

  # for more clarity on background processes, refer to specification 6
  ```

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

Specification 2: Builtin commands [20 marks]

[cd - 10 marks, echo - 5, marks, pwd - 5 marks]

Builtin commands are contained within the shell itself. Check 'type command-name' in the terminal (eg. 'type echo'). When the name of a shell 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.

You have to implement cd, echo and pwd.

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.
- For cd apart from the basic functionality, implement the flags: '.', '..', '-', '~'.
- 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.
- The pwd command writes the full pathname of the current working directory to the standard output. Basically, the absolute path of the directory.

Example:

```
<Name@UBUNTU:~>pwd
/home/user
<Name@UBUNTU:~>cd test
<Name@UBUNTU:~/test>cd ~
<Name@UBUNTU:~>cd -
/home/user/test
<Name@UBUNTU:~/test>
```

Specification 3: ls command [15 marks]

Implement the ls command which lists all the files and directories in the specified directory in **alphabetical order**.

Flags

-l : displays extra information regarding files

-a : display all files, including hidden files

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_name>
- ls <directory_path>
- ls <file_name>
- ls <flags> <directory/file_name>

- ls <dir_1> -l <dir_2> -a <file>

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**", print the **exact** same content as displayed by your actual shell.
- ***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.
- Use specific **colour** coding to differentiate between file names, directories and executables in the output. [eg. green for executables, white for files and blue for directories etc.]

Example:

```
# The ordering of the flags and file/directory names/paths does not matter.
# The command should print the desired output in all cases
ls -l dir_1 -a dir_2
ls -la dir_1 dir_2_path
ls dir_1 -al
ls dir_1 -l -a dir_2 # both flags are applicable for both the directories, consider
# the complete command and ignore the order
```

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

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

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

Time taken by the foreground process should be printed in the next prompt if the process takes at least 1 second to run.

Example:

```
<Name@UBUNTU:~>sleep 2
# sleeps for 2 seconds
<Name@UBUNTU:~took 2s>
```

Background processes [10 marks]: 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]**

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.
- Printing the process name along with its pid when a background process finishes is compulsory.

```
<Name@UBUNTU:~>sleep 10&
[1] 85435
<Name@UBUNTU:~>
sleep with pid = 85435 exited normally # After 10 seconds
<Name@UBUNTU:~>sleep 2
```

```
# sleeps for 2 seconds
<Name@UBUNTU:~>
```

Specification 5: pinfo command (user-defined) [10 marks]

pinfo prints the process-related info of your shell program.

Example:

```
<Name@UBUNTU:~>pinfo
pid : 231
process status : {R/S/S+/Z}
memory : 67854 {Virtual Memory}
executable path : ~/a.out
```

pinfo <pid> prints the process info of 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

Note:

- “+” must be added to the status code if the process is in the foreground.
- Use of “popen()” for implementing pinfo is NOT allowed.

Specification 6: Finished Background Processes [5 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: Discover command [15 marks]

Create a custom discover command which emulates the basics of the find command. The command should search for files in a directory hierarchy.

The command will have the following **optional command line arguments**:

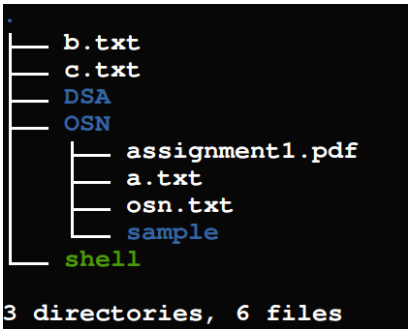
discover <target_dir> <type_flags> <file_name>

- target directory specified using ‘.’, ‘..’, ‘~’ or directory_name or directory_path. [if this argument is not present, consider the current directory as the target directory]

- the following type flags can be used with the command
  - -d → searches for all directories
  - -f → searches for all files
- name of the file/directory to be found in the given directory hierarchy.
- If neither the file type nor the file name is provided as a command line argument, print all the contents of the target directory.
- The ordering of the arguments doesn't matter.

Example:

Consider the given directory structure,




```
<Name@UBUNTU:~>discover . "osn.txt"
./OSN/osn.txt

<Name@UBUNTU:~>discover ./OSN "a.txt"
./OSN/a.txt

<Name@UBUNTU:~>discover -d
.
./DSA
./OSN
./OSN/sample
# the command searches for all the directories in the current directory hierarchy

<Name@UBUNTU:~>discover
.
./b.txt
./c.txt
./DSA
./shell
./OSN
./OSN/a.txt
./OSN/assignment1.pdf
./OSN/osn.txt

<Name@UBUNTU:~>discover -d -f
.
./b.txt
./c.txt
./DSA
./shell
./OSN
./OSN/a.txt
./OSN/assignment1.pdf
./OSN/osn.txt

<Name@UBUNTU:~>discover ./OSN -f
./OSN/a.txt
./OSN/assignment1.pdf
./OSN/osn.txt
```

Specification 8: history [7 marks]

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

DO NOT store the command in history if it is the exactly same as the previously entered command.

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
```

Bonus (Optional)

All bonus questions in this and future assignments will be **ungraded**. These are just some cool features you can try to implement. We won’t be giving any marks for this and it won’t be considered to fill in for lack of other features.

Aliases

- Aliases are custom shortcuts used to represent a command (or set of commands). An alias replaces a string that invokes a command in the shell with another user-defined string. Aliases are mostly used to replace long commands, improving efficiency and avoiding potential spelling errors.
- Implement the `alias` command. In case there are no command line arguments, print all the existing aliases in your shell. Check examples to see the formatting
- In case a command line argument of the format <key> ”<value>”, create a new alias <key> for the command <value>.
- Implement another command `dalias` which has a compulsory command line argument <key>, `dalias <key>` deletes the alias <key>.
- Modifying existing aliases is not allowed.
- Aliases are temporary and are not stored across multiple sessions.

Example:

```
<Name@UBUNTU:~>alias ABC "ls" # creates alias abc which is ls
<Name@UBUNTU:~>ABC            # executes ls command
<Name@UBUNTU:~>ABC -l         # executes ls -l
<Name@UBUNTU:~>dalias ABC     # removes ABC as an alias
<Name@UBUNTU:~>alias cd "pwd" # creates alias cd
<Name@UBUNTU:~>cd             # executes pwd
<Name@UBUNTU:~>cd ..          # error: pwd doesn't take any arguments
<Name@UBUNTU:~>alias cd "ls"  # error: you cannot modify an existing alias
<Name@UBUNTU:~>alias
cd -> "ls"
<Name@UBUNTU:~>
```

Autosuggestion

While you type an input command on the prompt, the shell should give a suggestion to complete the command using the history. The shell suggests the latest command from history which matches the substring you have entered.

The suggestion should be visible in faded colour and on pressing <Ctrl-n>, the suggested command should be autocompleted on the prompt.

Example:

<Name@UBUNTU:~>history
echo hello
ls ..
cd test
echo abc
type pwd
echo hi

<Name@UBUNTU:~>echo hi

'ec' is the current prompt, the latest command which has the
substring 'ec' is 'echo hi'. So, 'ho hi' will be displayed in faded color
<Name@UBUNTU:~>echo hi
#after pressing <Ctrl-n>, the command is autocompleted on the prompt
<Name@UBUNTU:~>type pwd

Up Arrow

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.

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.

Useful commands/structs/files:

uname, hostname, signal, waitpid, getpid, kill, execvp, strtok, fork, getopt, readdir, opendir, readdir, closedir, getcwd, sleep, 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. All C standard library functions are allowed unless explicitly banned. Third-party libraries are not allowed.
3. 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.
4. You ***MUST*** do error handling for both user-defined and system commands.
5. Use specific color coding for error messages, prompts, etc.
6. You can use both "*printf*" and "*scanf*" for this assignment.
7. 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.
8. 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.
9. If the code doesn't compile, no marks will be rewarded.
10. Segmentation faults at the time of grading will be penalized.
11. The bonus section is **optional** and ungraded.
12. Write this code in a **modular fashion**. In the next assignment, you will add more features to your shell.

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.

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**
2. Make sure you write a **makefile** for compiling all your code (with appropriate flags and linker options).
3. Kindly adhere to the following naming guidelines and directory structure:

```
<Roll-No>_Assignment2
├── README.md
├── makefile
└── Other files and Directories
```

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