Mini Project 1 : Building your own shell

Instructions

In this Mini Project, you will be building your own shell using C. As the scale of code will be very large, it is **necessary to write modular code. Do NOT write monolithic code.** Following is the definition of what modular code should look like : your codebase must be separated in different C files based on the tasks that it is intended to perform. Create header files to include headers from C library. You will be penalized heavily if your code is non modular.

Part A : Basic System Calls

Specification 1 : Display Requirement [5]

On every line when waiting for user input, a shell prompt of the following form must appear along with it. Do NOT hard code the username/system name here. The current directory address should be shown in the prompt as well.

```
<Username@SystemName:~>
```

For example, if system name is “SYS”, username is “JohnDoe”, and the user is currently in the home directory, then prompt looks like this :

```
<JohnDoe@SYS:~>
```

The directory from which shell is invoked becomes the home directory for the shell and represented with “~”. All paths inside this directory should be shown relative to it. Absolute path of a directory/file must be shown when outside the home directory.

When user changes the working directory, the corresponding change in path of directory must be reflected in the next prompt. For example, on going to the parent directory of the home directory of shell, following form of prompt is expected :

```
<JohnDoe@SYS:/home/johndoe/sem3>
```

Specification 2 : Input Requirements [5]

Keep in mind the following requirements for input when implementing your shell :

- Your shell should support a ‘;’ or ‘&’ separated list of commands. You can use ‘**strtok**’ to tokenize the input.
- Your shell should account for random spaces and tabs when taking input.

- The “;” command can be used to give multiple commands at the same time. This works similar to how “;” works in Bash.
- ‘&’ operator runs the command preceding it in the background after printing the process id of the newly created process. **(Refer to Specification 4 for more details on this)**

```
./a.out
<JohnDoe@SYS:~> vim &
[1] 35006
<JohnDoe@SYS:~> sleep 5 & echo "Lorem ipsum"
[2] 35036
Lorem ipsum
# sleep runs in the background while echo runs in the foreground
<JohnDoe@SYS:~> hop test ; pwd
sleep with pid 35036 exited normally # after 5 seconds
~/test
<JohnDoe@SYS:~/test>
```

If any command is erroneous, then error should be printed.

```
<JohnDoe@SYS:~> sleeeep 6
ERROR : 'sleeeep' is not a valid command
```

Specification 3 : hop [5]

‘hop’ command changes the directory that the shell is currently in. It should also print the full path of working directory after changing. The directory path/name can be provided as argument to this command.

- You are also expected to implement the “.”, “..”, “~”, and “-” flags in hop. ~ represents the home directory of shell (refer to specification 1).
- You should support both absolute and relative paths, along with paths from home directory.
- If more than one argument is present, execute hop sequentially with all of them being the argument one by one (from left to right).
- If no argument is present, then hop into the home directory.

```
<JohnDoe@SYS:~> hop test
/home/johndoe/test
<JohnDoe@SYS:~/test> hop assignment
```

```
/home/johndoe/test/assignment
<JohnDoe@SYS:~/test/assignment> hop ~
/home/johndoe
<JohnDoe@SYS:~> hop -
/home/johndoe/test/assignment
<JohnDoe@SYS:~/test/assignment> hop .. tutorial
/home/johndoe/test
/home/johndoe/test/tutorial
<JohnDoe@SYS:~/test/tutorial> hop ~/project
/home/johndoe/project
<JohnDoe@SYS:~/project>
```

Note :

- You can assume that the paths/names will not contain any whitespace characters.
- DON’T use ‘execvp’ or similar commands for implementing this.

Specification 4 : reveal [8]

‘reveal’ command lists all the files and directories in the specified directories in lexicographic order (default reveal does not show hidden files). You should support the -a and -l flags.

- -l : displays extra information
- -a : displays all files, including hidden files

Similar to hop, you are expected to support “.”, “..”, “~”, and “-” symbols.

- Support both relative and absolute paths.
- If no argument is given, you should reveal at the current working directory.
- Multiple arguments will not be given as input.

The input will always be in the format :

```
reveal <flags> <path/name>
```

Handle the following cases also in case of flags :

- reveal -a <path/name>
- reveal -l <path/name>
- reveal -a -l <path/name>
- reveal -l -a <path/name>

- reveal -la <path/name>
- reveal -al <path/name>
- **Handle the cases where there are multiple l & a like:-**
    - reveal -lala <path/name>

Note :

- You can assume that the paths/names will not contain any whitespace characters.
- DON’T use ‘execvp’ or similar commands for implementing this.
- Use specific color coding to differentiate between file names, directories and executables in the output [green for executables, white for files and blue for directories].
- Print a list of file/folders separated by newline characters.
- The details printed with -l should be the same as the ls command present in Bash.

Specification 5 : log commands [8]

log

Implement a ‘log’ command which is similar to the actual history command in bash. The default number of commands it should store and output is 15 (max). 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. The commands should be printed from oldest to recent moving top to down (Refer example below).

Note :

- DO NOT store a command in log if it is the exactly same as the previously entered command.
- Store the arguments along with the command
- Commands separated by ; or & are supposed to be stored in a single line.
- Erroneous commands can also be stored (this is not a strict requirement, and you can choose to not store them). Please mention whatever you choose in README accordingly.
- Do NOT store the log command in log. In cases where multiple commands separated by ; or & contain a log command, you are expected to not store the entire command string. For eg. in case of `hop .. ; echo "OSN" ; log`, the log will not be updated at all since log is present in the command string.

log purge

Clears all the log currently stored. Do not store this command in the log.

log execute <index>

Execute the command at

```
<JohnDoe@SYS:~> reveal test
```

```
# output
<JohnDoe@SYS:~> sleep 5
# output
<JohnDoe@SYS:~> sleep 5
# output
<JohnDoe@SYS:~/test> echo "Lorem ipsum"
# output
<JohnDoe@SYS:~> log
reveal test
sleep 5
echo "Lorem ipsum"
<JohnDoe@SYS:~> log execute 1
# output of echo "Lorem ipsum"
<JohnDoe@SYS:~> log
reveal test
sleep 5
echo "Lorem ipsum"
<JohnDoe@SYS:~> log execute 3
# output of reveal test
<JohnDoe@SYS:~> log
reveal test
sleep 5
echo "Lorem ipsum"
<JohnDoe@SYS:~> log purge
<JohnDoe@SYS:~> log
<JohnDoe@SYS:~>     //outputs nothing since log has been purged
<JohnDoe@SYS:~> sleep 5 ; reveal
# output
<JohnDoe@SYS:~> log
sleep 5 ; reveal
<JohnDoe@SYS:~> sleep 5 ; revealll
# error mesage
```

```
<JohnDoe@SYS:~> sleep 5 ; echo "Lorem Ipsum" ; log

<JohnDoe@SYS:~> log

sleep 5 ; reveal

sleep 5 ; revealll
```

Specification 6 : System commands [12]

Your shell must be able to execute the other system commands present in Bash as well like emacs, gedit etc. This should be possible in both foreground and background processes.

Foreground Process

Executing a command in foreground means the shell will wait for that process to complete and regain control afterwards. Control of terminal is handed over to this process for the time being while it is running.

Time taken by the foreground process and the name of the process should be printed in the next prompt if process takes > 2 seconds to run. Round the time down to integer before printing in prompt.

```
<JohnDoe@SYS:~> sleep 5

# sleeps for 5 seconds

<JohnDoe@SYS:~ sleep : 5s>
```

Background Process

Any command invoked with “&” is treated as a background command. This implies that your shell will spawn that process but doesn’t hand the control of terminal to it. Shell 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.

```
<JohnDoe@SYS:~> sleep 10 &

13027

<JohnDoe@SYS:~> sleep 20 & # After 10 seconds

Sleep exited normally (13027)

13054

<JohnDoe@SYS:~> echo "Lorem Ipsum" # After 20 seconds

Sleep exited normally (13054)

Lorem Ipsum
```

Note :

- No need to handle background processes for any commands implemented by yourself (hop, reveal, log etc.)
- You should be able to run multiple background processes.
- Whenever the background process finishes, display a message to the user **autonomously, without any futher interaction required by the user.**
- Print process name along with pid when background process ends. Also mention if the process ended normally or abnormally.

Specification 7 : proclore [5]

proclore is used to obtain information regarding a process. If an argument is missing, print the information of your shell.

Information required to print :

- pid
- Process Status (R/R+/S/S+/Z)
- Process group
- Virtual Memory
- Executable path of process

```
<JohnDoe@SYS:~> proclore
pid : 210
process status : R+
Process Group : 210
Virtual memory : 167142
executable path : ~/a.out


<JohnDoe@SYS:~> proclore 301
pid : 301
process Status : R
Process Group : 243
Virtual memory : 177013
executable Path : /usr/bin/gcc
```

Process states :

- R/R+ : Running
- S/S+ : Sleeping in an interruptible wait

- Z : Zombie

The “+” signifies whether it is a foreground or background process.

Specification 8 : seek [8]

‘seek’ command looks for a file/directory in the specified target directory (or current if no directory is specified). It returns a list of relative paths (from target directory) of all matching files/directories **(files in green and directories in blue)** separated with a newline character.

Note that by files, the text here refers to non-directory files.

Flags :

- -d : Only look for directories (ignore files even if name matches)
- -f : Only look for files (ignore directories even if name matches)
- -e : This flag is effective only when a single file or a single directory with the name is found. If only one file (and no directories) is found, then print it’s output. If only one directory (and no files) is found, then change current working directory to it. Otherwise, the flag has no effect. This flag should work with -d and -f flags.

If -e flag is enabled but the directory does not have access permission (execute) or file does not have read permission, then output **“Missing permissions for task!”**

Argument 1 :

The target that the user is looking for. A name **without whitespace characters** will be given here. You have to look for a file/folder with the exact name, or one which contains this target as a prefix in its name.

Argument 2 (optional) :

The path to target directory where the search will be performed (this path can have symbols like . and ~ as explained in the reveal command). If this argument is missing, target directory is the current working directory. The target directory’s tree must be searched (and not just the directory).

```
<JohnDoe@SYS:~> seek newfolder

./newfolder

./doe/newfolder

./doe/newfolder/newfolder.txt

<JohnDoe@SYS:~> seek newfolder ./doe

./newfolder # This is relative to ./doe

./newfolder/newfolder.txt

<JohnDoe@SYS:~> seek -d newfolder ./doe

./newfolder

<JohnDoe@SYS:~> seek -f newfolder ./doe
```

```
./newfolder/newfolder.txt

<JohnDoe@SYS:~> seek newfolder ./john

No match found!

<JohnDoe@SYS:~> seek -d -f newfolder ./doe

Invalid flags!

<JohnDoe@SYS:~> seek -e -f newfolder ./doe

./newfolder/newfolder.txt

This is a new folder! # Content of newfolder.txt

<JohnDoe@SYS:~> seek -e -d newfolder ./doe

./newfolder/

<JohnDoe@SYS:~/doe/newfolder>
```

Print "**No match found!**" in case no matching files/directories is found. Note that -d and -f flag can't be used at the same time and must return error message "**Invalid flags!**".

A call to this command will always be in the format :

seek <flags> <search> <target_directory>

Part B : Processes, Files and Misc.

Note : Please read specification 10-12 together before attempting as one's implementation will affect the other. The evaluation will be done separately, this note is just for your convenience.

Specification 9 : .myshrc [2 + 2 [BONUS]]

The `myshrc` file is a customized shell configuration file, similar to `.bashrc`, tailored to enhance your command-line experience. It includes useful aliases and functions to streamline your workflow and improve efficiency. In this you need to create your own .bashrc file which will use **aliases** and **functions**.

In this case, implementing *any* alias (not just the ones mentioned in the example below) will award you with **2 points**. All you need to do is to map a single word alias to some command. You are not expected to handle aliases mapping to any custom functions. You do not need to worry about multi-word aliases.

Additionally, implementing the two functions mentioned in the example below (*mk_hop* and *hop_seek*) will award you with **2 BONUS points** (this is entirely **optional**).

```
<JohnDoe@SYS:~> nano .myshrc
// File Contents are given below (feel free to make edits to it coz its yours :)
//# You can create aliases without the alias keyword
reveall = reveal -l

//OR
```

```
// You can create aliases with a predefined keyword, say, 'alias' as well
(Whatever suits you! Note: You aren't expected to handle both ways; either way
is fine.)
alias reveala = reveal -a
alias home = hop ~

// Functions (BONUS)
mk_hop()
{
        mkdir "$1" # Create the directory
        hop "$1" # Change into the directory
}

hop_seek()
{
    hop "$1" # Hop into this directory
    seek "$1" # search for files/directories with the same name as the directory
you just hopped into.
}

// OR
// You can create functions with a predefined keyword, say, 'func' as well
(Whatever suits you! Note: You aren't expected to handle both ways; either way
is fine.)
func mk_hop()
{
        mkdir "$1" # Create the directory
        hop "$1" # Change into the directory
}

func hop_seek()
{
    hop "$1" # Hop into this directory
    seek "$1" # search for files/directories with the same name as the directory
you just hopped into.
}


// File Content Ends
<JohnDoe@SYS:~> reveall
// Output is same as what reveal -l usually does

// assuming test dir doesnt exist
<JohnDoe@SYS:~> mk_hop test
// makes test dir
<JohnDoe@SYS:~/test>
```

Specification 10 : I/O Redirection [10]

I/O Redirection is when you change the default input/output (which is the terminal) to another file. This file can be used to read input into a program or to capture the output of a program.

Your shell should support >, <, » (< should work with both > and »).

- > : Outputs to the filename following “>”.
- >> : Similar to “>” but appends instead of overwriting if the file already exists.
- < : Reads input from the filename following “<”.

Your shell should handle these cases appropriately:

- An error message “No such input file found!” should be displayed if the input file does not exist.
- The output file should be created (with permissions 0644) if it does not already exist in both > and ».
- In case the output file already exists, it should be overwritten in case of > and appended to in case of ».

You are NOT required to handle multiple inputs and outputs.

```
<JohnDoe@SYS:~> echo "Hello world" > newfile.txt
<JohnDoe@SYS:~> cat newfile.txt
Hello world
<JohnDoe@SYS:~> wc < a.txt
1 2 12 # There can be extra spaces
<JohnDoe@SYS:~> echo "Lorem ipsum" > newfile.txt
<JohnDoe@SYS:~> cat newfile.txt
Lorem ipsum
<JohnDoe@SYS:~> echo "dolor sit amet" >> newfile.txt; cat newfile.txt
Lorem ipsum
dolor sit amet
<JohnDoe@SYS:~> wc < newfile.txt > a.txt
<JohnDoe@SYS:~> cat a.txt
2 5 27 # There can be extra spaces
```

Specification 11 : Pipes [12]

Pipes are used to pass information between commands. It takes the output from command on left and passes it as standard input to the command on right. Your shell should support any number of pipes.

Note :

- Return error “Invalid use of pipe”, if there is nothing to the left or to the right of a pipe (”|”).
- Run all the commands sequentially from left to right if pipes are present.

```
<JohnDoe@SYS:~> echo "Lorem Ipsum" | wc
1 2 12 # extra spaces can be present
```

Specification 12 : Redirection along with pipes [8]

This specification requires you to be able to run I/O redirection along with pipes. It should support any number of pipes (but not multiple inputs and outputs from I/O redirection). In short, you are required to make sure that Specification 10 and Specification 11 work when given as input together.

```
<JohnDoe@SYS:~> cat a.txt

Lorem Ipsum

<JohnDoe@SYS:~> cat < a.txt | wc | cat > b.txt

<JohnDoe@SYS:~> cat b.txt

1 2 12  # note that extra spaces can be present
```

Specification 13 : activities [5]

This specification requires you to print a list of all the processes currently running that were spawned by your shell in lexicographic order. This list should contain the following information about all processes :

- Command Name
- pid
- state : running or stopped

```
<JohnDoe@SYS:~> activities

221 : emacs new.txt - Running

430 : vim - Stopped

620 : gedit - Stopped
```

Format of an entry should be :

[pid] : [command name] - [State]

Specification 14 : Signals [7]

ping <pid> <signal_number>

ping command is used to send signals to processes. Take the pid of a process and send a signal to it which corresponds to the signal number (which is provided as an argument). Print error “No such process found”, if process with given pid does not exist. You should take signal number’s modulo with 32 before checking which signal it belongs to (assuming x86/ARM machine). Check man page for signal for an exhaustive list of all signals present.

```
<JohnDoe@SYS:~> activities

221 : emacs new.txt - Running
```

```
430 : vim - Running

620 : gedit - Stopped

<JohnDoe@SYS:~> ping 221 9                                    # 9 is for
SIGKILL

Sent signal 9 to process with pid 221

<JohnDoe@SYS:~> activities

430 : vim - Running

620 : gedit - Stopped

<JohnDoe@SYS:~> ping 430 47

Sent signal 15 to process with pid 430          # 15 is for SIGTERM

<JohnDoe@SYS:~> activities

430 : vim - Stopped

620 : gedit - Stopped
```

Following 3 commands are direct keyboard input where Ctrl is Control key on keyboard (or it's equivalent).

Ctrl - C

Interrupt any currently running foreground process by sending it the SIGINT signal. It has no effect if no foreground process is currently running.

Ctrl - D

Log out of your shell (after killing all processes) while having no effect on the actual terminal.

Ctrl - Z

Push the (if any) running foreground process to the background and change it's state from "Running" to "Stopped". It has no effect on the shell if no foreground process is running.

Specification 15 : fg and bg [10]

fg <pid>

Brings the running or stopped background process with corresponding pid to foreground, handing it the control of terminal. Print "No such process found", if no process with given pid exists.

```
<JohnDoe@SYS:~> activities

620 : gedit - Stopped

<JohnDoe@SYS:~> fg 620

# brings gedit [620] to foreground and change it's state to Running
```

bg <pid>

Changes the state of a stopped background process to running (in the background). If a process with given pid does not exist, print “No such process found” to the terminal.

```
<JohnDoe@SYS:~> activities

620 : gedit - Stopped

<JohnDoe@SYS:~> bg 620

# Changes [620] gedit's state to Running (in the background)
```

Specification 16 : Neonate [7] [BONUS]

Command : **neonate -n [time_arg]**

The command prints the Process-ID of the most recently created process on the system (you are not allowed to use system programs), this pid will be printed every [time_arg] seconds until the key ‘x’ is pressed.

```
<JohnDoe@SYS:~> neonate -n 4

# A line containing the pid should be printed

# every 4 seconds until the user

# presses the key: 'x'.

11810

11811

11811

11812

11813 # key 'x' is pressed at this moment terminating the printing
```

Part C : Networking

Specification 17 : iMan [4]

iMan fetches man pages from the internet using sockets and outputs it to the terminal (stdout). In this case, you are required to use the website : http://man.he.net/ to get the man pages.

You do not need to handle “man page not found” error separately and you can print the html tags as well if there are any.

Do not print the header received during the GET request. (Check example given below for exact output).

iMan <command_name>

<command_name> is the name of the man page that you want to fetch.

This should fetch the man page for the given command from http://man.he.net/, if the page does not exist, you can continue to print the response from the page.

```
<JohnDoe@SYS:~> iMan sleep

# expected output here :
sleep



sleep

SLEEP(1)                         User Commands                         SLEEP(1)

NAME
       sleep - delay for a specified amount of time

SYNOPSIS
       sleep NUMBER[SUFFIX]...
       sleep OPTION

DESCRIPTION
       Pause for NUMBER seconds.  SUFFIX may be 's' for seconds (the default),
       'm' for minutes, 'h' for hours or 'd' for days.  Unlike most  implemen-
       tations  that require NUMBER be an integer, here NUMBER may be an arbi-
       trary floating point number.  Given two or more  arguments,  pause  for
       the amount of time specified by the sum of their values.

       --help display this help and exit

       --version
              output version information and exit

AUTHOR
       Written by Jim Meyering and Paul Eggert.

REPORTING BUGS
       GNU coreutils online help:
&lt;https://www.gnu.org/software/coreutils/&gt;
       Report sleep translation bugs to
&lt;https://translationproject.org/team/&gt;

COPYRIGHT
       Copyright  (C) 2018 Free Software Foundation, Inc.  License GPLv3+: GNU
       GPL version 3 or later &lt;https://gnu.org/licenses/gpl.html&gt;.
       This is free software: you are free  to  change  and  redistribute  it.
       There is NO WARRANTY, to the extent permitted by law.

SEE ALSO
       sleep(3)

       Full documentation at:
&lt;https://www.gnu.org/software/coreutils/sleep&gt;
       or available locally via: info '(coreutils) sleep invocation'

GNU coreutils 8.30              September 2019                       SLEEP(1)

Man Pages Copyright Respective Owners.  Site Copyright (C) 1994 - 2024
Hurricane Electric.
All Rights Reserved.
```

```
o or more  arguments,  pause  for
       the amount of time specified by the sum of their values.

       --help display this help and exit

       --version
              output version information and exit

AUTHOR
       Written by Jim Meyering and Paul Eggert.

REPORTING BUGS
       GNU coreutils online help: &lt;ht

// expected output ends

<JohnDoe@SYS:~> iMan invalid_command


# expected output here :


invalid_command




invalid_command
No matches for "invalid_command"
Search Again

Man Pages Copyright Respective Owners.  Site Copyright (C) 1994 - 2024
Hurricane Electric.
All Rights Reserved.


ERROR: No matches for "invalid_command" command

// expected output ends
```

If there are more than 1 argument, iMan considers only the first argument and ignores the rest. For example :

```
"iMan sleep" is the same as "iMan sleep extra"
```

Guidelines

1. The submission must be done in C. No other languages are allowed.
2. All C standard library functions are allowed unless explicitly mentioned. You can find the list of header files in C standard library here : https://en.cppreference.com/w/c/header
3. Third party libraries are not allowed.

4. If there is an error while running a command, it **should not cause the shell to crash**. Handle errors as they are mentioned in the Mini Project document. If an **error is not mentioned here, you should still handle it appropriately**. Look at perror.h for handling error routines.
5. Do error handling for both user defined and system commands.
6. Use specific color coding for error messages and prompts (You can choose the colors)
7. As mentioned in Input Requirements, there can be multiple random tabs and spaces, you should handle these (and still run the entered command).
8. The user can execute other files from within the shell (including the shell). Your program must be able to execute them or print error if it is unable to do so.
9. Use of popen, pclose and system() calls is not permitted.
10. Use the exec family of command to execute the system commands. Errors should be handled appropriately.
11. You are not required to implement background functionality for user commands (commands implemented by you) in this shell (hop, reveal, seek, proclore etc.)
12. However, piping and redirection must work with both user and system commands.
13. Use signal handlers to handle signals when switching between foreground and background or when a background process exits.
14. Your code MUST compile for any marks to be given. Use version control to save working versions of code before messing around with it and write modular code.
15. Segmentation faults and other crashes while your shell is running will be penalized.
16. The symbols <, >, », &, |, ; , - would always correspond to their special meaning and would not appear otherwise, such as in inputs to echo etc.
17. You are expected to implement everything except Specification 6 without using execvp.

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.

Also check out termios.h functions like tcgetattr(), tcsetattr() for terminal setting (raw mode/ cooked mode).

Refer to man pages on internet 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.

Submission Guidelines

1. Make a **makefile** for compiling all your code (with appropriate flags and linker options). This makefile is expected to generate an **executable “a.out” on running the “make”** command. Executing a.out should start your shell.
2. Include a README.md describing which file corresponds to what part and any assumptions that you made. Any assumptions not written in README.md may not be considered during manual evaluation.
3. Submit modular code only.
4. Following is the expected directory structure for your submission :

```
├── README.md
├── makefile
└── Other files and directories
```