Operating Systems and Networks

Mini Project 1 : Building your own shell

Before starting, make sure to read this document **completely** including **Instructions, Guidelines, Plagiarism Policy and Submission Guidelines described at the end** before asking any doubts as they might already be covered here. Additionally, you are recommended to read these to avoid unnecessary penalties.

GitHub Classroom

Follow this link to accept the assignment. You will then be assigned a private repository on GitHub. This is where you will be working on the mini project. All relevant instructions regarding this project can be found below.

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.

It is highly recommended to start this mini project early to complete it in time as it will be quite intensive. There are 3 parts to this assignment with two deadlines in total (including final deadline). More clearly, you are required to submit your code 2 times in the duration of this assignment with the last deadline being the final submission deadline (Parts A,B,C combined) before getting penalized for submitting after this till the hard deadline which is 1 week after it i.e. 19 Sep 2023

- 1st deadline : Only part A is required to be submitted 50% points of Part A will be given based on this submission.
- 2nd deadline : All Parts A, B and C must be submitted. 50% points of Part A and 100% points of Part B,C will be given based on this submission.

For easier understanding, let’s say Part A is worth 100 points total. If your submission is worth 20 points from Part A during 1st deadline, and you submit it completely during 2nd deadline (100 points), you will get a total of (20/2) + (100/2) = 60 points. If you submit Part A

completely in both deadlines, you will get full marks. For Part B,C: whatever you submit during 2nd deadline will be used to award you points. **Thus, in case of missing 1st deadline, 50% points of Part A will be lost even if that part is submitted at a later deadline.**

Deadlines

Intermediary deadline : 11:59 PM - 25 Aug, 2023

Final Soft deadline : 11:59 PM - 12 Sep, 2023

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.

```
./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:~> warp 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 : warp [5]

'warp' 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 warp. ~ 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 warp sequentially with all of them being the argument one by one (from left to right).

If no argument is present, then warp into the home directory.

```
<JohnDoe@SYS:~> warp test
/home/johndoe/test
<JohnDoe@SYS:~/test> warp assignment
/home/johndoe/test/assignment
<JohnDoe@SYS:~/test/assignment> warp ~
/home/johndoe
<JohnDoe@SYS:~> warp -
/home/johndoe/test/assignment
<JohnDoe@SYS:~/test/assignment> warp .. tutorial
/home/johndoe/test
/home/johndoe/test/tutorial
<JohnDoe@SYS:~/test/tutorial> warp ~/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 : peek [8]

'peek' command lists all the files and directories in the specified directories in lexicographic order (default peek 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 warp, you are expected to support ".", "..", "~", and "-" symbols.

Support both relative and absolute paths.

If no argument is given, you should peek at the current working directory.

Multiple arguments will not be given as input.

The input will always be in the format :

```
peek <flags> <path/name>
```

Handle the following cases also in case of flags :

peek -a <path/name>

peek -l <path/name>

peek -a -l <path/name>

peek -l -a <path/name>

peek -la <path/name>

peek -al <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 : pastevents commands [8]

pastevents

Implement a 'pastevents' command which is similar to the actual history command in bash. Your implementation should store (and output) the 15 most recent command statements given as input to the shell based on some constraints. You must overwrite the oldest commands if more than the set number (15) of commands are entered. `pastevents` is persistent over different shell runs, i.e., the most recent command statements should be stored when the shell is exited and be retrieved later on when the shell is opened.

Note :

DO NOT store a command in pastevents if it is the exactly same as the previously entered command. (You may use direct string comparison, `strcmp()`, for this)

Store the arguments along with the command

(*Edit*) Store all statements except commands that include pastevents or pastevents purge.

pastevents purge

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

pastevents execute <index>

Execute the command at position in pastevents (ordered from most recent to oldest). Do not store statements containing $p * eventsexecute$ if it matches the most recent command.

```
<JohnDoe@SYS:~> peek test
# output
<JohnDoe@SYS:~> sleep 5
# output
<JohnDoe@SYS:~> sleep 5
# output
<JohnDoe@SYS:~/test> echo "Lorem ipsum"
# output
<JohnDoe@SYS:~> pastevents
peek test
sleep 5
echo "Lorem ipsum"
<JohnDoe@SYS:~> pastevents execute 1
# output of echo "Lorem ipsum"
<JohnDoe@SYS:~> pastevents
peek test
sleep 5
echo "Lorem ipsum"
<JohnDoe@SYS:~> pastevents execute 3
# output of peek test
<JohnDoe@SYS:~> pastevents
peek test
sleep 5
echo "Lorem ipsum"
peek test
<JohnDoe@SYS:~> pastevents purge
<JohnDoe@SYS:~> pastevents
<JohnDoe@SYS:~>
```

(*Edit*)

```
<JohnDoe@SYS:~> pastevents # Assuming this is the first time the shell is run
<JohnDoe@SYS:~> sleep 1
<JohnDoe@SYS:~> sleep 2
<JohnDoe@SYS:~> pastevents
```

```
sleep 1
sleep 2
<JohnDoe@SYS:~> sleep 2
sleep 1
sleep 2  # sleep 2 is not repeated
<JohnDoe@SYS:~> sleep 1; pastevents
sleep 1
sleep 2
<JohnDoe@SYS:~> pastevents # Notice how 'sleep 1; pastevents' is not considere
sleep 1
sleep 2

*** Exit Shell ***

<JohnDoe@SYS:~> sleep 2
<JohnDoe@SYS:~> pastevents
sleep 1
sleep 2
exit
sleep 2
<JohnDoe@SYS:~> sleep 3; pastevents execute 1
<JohnDoe@SYS:~> sleep 3; sleep 2
<JohnDoe@SYS:~> sleep 3
<JohnDoe@SYS:~> sleep 3; sleep 2
<JohnDoe@SYS:~> pastevents
sleep 1
sleep 2
exit
sleep 2
sleep 3; sleep 2 # Only output once as 'sleep 3; pastevents execute 1' is stor
sleep 3
sleep 3; sleep 2
<JohnDoe@SYS:~> pastevents purge; sleep 1
<JohnDoe@SYS:~> pastevents # No output as the previous command contained paste
<JohnDoe@SYS:~> sleep 1
<JohnDoe@SYS:~> pastevents
sleep 1
```

NOTE - handling omissions in the case of background processes is left to student (as you need not handle background execution for user defined commands). However, ensure that

you write the assumptions made in the README and that there is no infinite recursion (pastevents execute calling pastevents execute calling pastevents execute…)

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 (warp, peek, pastevents etc.)

You should be able to run multiple background processes.

Whenever background process finishes, display message to user (after user enters any command, display all ended between last command run and this).

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, i.e., add “+” only if it is a *foreground* 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 as this.

Argument 2 :

The path to target directory where the search will be performed (this path can have symbols like . and ~ as explained in the peek 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 9-11 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 : I/O Redirection [8]

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. This specification should work for all commands - user defined as well as system commands defined in bash. 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
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 10 : Pipes [8]

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. This specification should work for all commands - user defined as well as system commands defined in bash.

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
<JohnDoe@SYS:~> echo "Lorem Ipsum" | wc | sed 's/ //g'
1212
```

Specification 11 : Redirection along with pipes [5]

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 8 and Specification 9 work when given as input together.

```
<JohnDoe@SYS:~> cat a.txt
Lorem Ipsum
<JohnDoe@SYS:~> cat < a.txt | wc | sed 's/ //g' | cat > b.txt
<JohnDoe@SYS:~> cat b.txt
1212
```

Specification 12 : 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 13 : Signals [12]

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 - Stopped
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 - Stopped
620 : gedit - Stopped
<JohnDoe@SYS:~> ping 430 41
Sent signal 9 to process with pid 430
<JohnDoe@SYS:~> activities
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 14 : fg and bg [8]

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 15 : Neonate [8]

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
```

The starter code for this specification can be found here. Please also note a few things:

1. By default , I/O of the teminal is line-buffered, i.e, input is guaranteed to be flushed/sent to your program once a line is terminated.
2. termios.h, a POSIX-standard header file, allows you to get the tty into raw mode whereas it is generally in cooked mode. Reading it’s documentation/man pages is suggested.
3. Getting the terminal into raw mode allows it to be such that at soon as a key is pressed, the input signal is sent to your program, along with a lot of other default features like echoing being disabled.

Part C : Networking

Specification 16 : iMan [12]

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 must implement it in the following manner:

1. Do DNS resolution for man.he.net
2. Open a TCP socket to the IP address
3. Send a Get request to the website's server
4. Read the body of the website
5. Close socket

Libraries that could be used are:

1. <unistd.h>
2. <netdb.h>
3. <arpa/inet.h>

along with general libraries such as <stdlib.h>, <string.h>, <stdio.h> and so on. Relevent functions will be discussed in the tutorial.

iMan <command_name>

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

This should fetch the man page (*atleast* the name, synopsis and description of the command) for the given command from http://man.he.net/. Print an error statement if the page does not exist.

```
<JohnDoe@SYS:~> iMan sleep
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
<JohnDoe@SYS:~> iMan invalid_command
ERROR
         No such command
```

Outputs given above may be formatted in any manner as long as the information by the website is unaltered.

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.

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 (warp, peek, 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.

Plagiarism Policy

Do NOT take code from your seniors or batchmates as we will do extensively evaluation for any plagiarism. Heavy Moss will be run against the previous few year’s submissions as well, and any case of **plagiarism will be punished heavily according to the course policy.**

There will be a viva conducted during manual evaluations, related to your code and the logic/concept involved in it. You will get a penalty of up to 100% if you are unable to answer these questions regarding the working of your code and the concept behind it.

Takeaway : Do not copy, it won’t get you marks but there is a high chance for it to get you trouble.

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. Following is the expected directory structure for your submission :

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