Bhavicka Mohta (bm3001)
Maisha Mahrin (mm10294)

Operating Systems Project Report

Basic CLI Implementation in Server:

In this phase, we built upon phase 1 and created a socket communication model where the shell was usable from the client end.

We also added a help function (which is called by entering “help”) to give the user more details about what can and cannot be entered. Currently all compound commands work if they’re entered singly without pipes. A thing we wanted to implement but didn’t get the time to was combining compound commands (like “ls -lh”) using pipes. It’s definitely one of the changes we would like to make in the future if we’re able to fit it in.

The user defined functions used in the code:

```
void init_shell()
```

It prints the username of the PC running the shell for which we use the getenv() function.

```
void printDir()
```

Everytime to when the user is prompted to type a shell command, the directory of the shell file where the shell script is running gets printed.

```
commandList()
```

Prints all the available commands

```
void parseSpace(char* inputString,char** parsedArgs )
```

Parses the spaces in single line commands

```
void execArgs(char** parsedArgs, int newsocket)
```

This function executes the user commands for single line commands using execvp in a child process.

So, we pass a vector as well as the name of the command to execvp to run the command.

```
int parsePipe(char* inputString, char** parsedArgsPiped)
```

Parses the pipes in piped commands : We then pass our input string into a function which counts the number of pipes ("|") in the input. If there are 0 pipes, which means it's a normal separate command, it is passed into a functions which parses the input according to space and then these parsed arguments are passed into a method called execArgs which forks a child and uses execvp inside it to execute the commands.

In the case that there are pipes, we create pipe mechanisms to write the output of the first child to the read end of the next pipe/parent, whatever the case might be.

```
void execArgsPiped1(char** parsedArgsPiped, int newsocket)
```

This function executes the user commands for one piped commands using execvp in 2 separate child processes. So, we pass a vector as well as the name of the command to execvp to run the command. We redirect the output to the client using dup2();

```
void execArgsPiped2(char** parsedpipe, int newsocket)
```

This function executes the user commands for two piped commands using execvp in 3 separate child processes. So, we pass a vector as well as the name of the command to execvp to run the command. We redirect the output to the client using dup2();

```
void execArgsPiped3(char** parsedpipe, int newsocket)
```

This function executes the user commands for two piped commands using execvp in 4 separate child processes. So, we pass a vector as well as the name of the command to execvp to run the command. We redirect the output to the client using dup2();

Server-Client Communication:

Bhavicka Mohta (bm3001)
Maisha Mahrin (mm10294)

The goal of this phase was to create a server which could provide remote accessibility to our shell. The client could send a request to the server and it would get the output from there. This is how we went about implementing that:

Socket creation:

int server_fd = socket(domain, type, protocol)

server_fd: socket descriptor, an integer (like a file-handle)

domain: integer, specifies communication domain. Since we are communicating between processes on different hosts connected by IPV4, we use AF_INET

type: communication type SOCK_STREAM: TCP(reliable, connection oriented)
SOCK_DGRAM: UDP(unreliable, connectionless), we build a tcp socket, so use SOCK_STREAM.

protocol: Protocol value for Internet Protocol(IP), which is 0. This is the same number which appears on protocol field in the IP header of a packet

Bind:

int bind(int server_fd, const struct sockaddr *addr, socklen_t addrlen);

After creation of the socket, bind function binds the socket to the address and port number specified in addr(custom data structure).

Listen:

int listen(int server_fd, int backlog);

It puts the server socket in a passive mode, where it waits for the client to approach the server to make a connection. The backlog defines the maximum length to which the queue of pending connections for sockfd may grow. In our code, we have used 10 because if a connection request

arrives when the queue is full, the client may receive an error with an indication of ECONNREFUSED.

Accept:

int new_socket= accept(int server_fd, struct sockaddr *addr, socklen_t *addrlen);

It extracts the first connection request on the queue of pending connections for the listening socket, server_fd, creates a new connected socket, and returns a new file descriptor referring to that socket. At this point, connection is established between client and server, and they are ready to transfer data.

Reading the message:

Here int read(new_socket, message, 1024); Reads the message from the client using socket new_socket and stores the string in the message.

Stages for Client

Socket connection: Exactly same as that of server's socket creation

Connect:

int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);

The connect() system call connects the socket referred to by the file descriptor sockfd to the address specified by addr. Server's address and port is specified in addr.

How to Run

We've provided a Makefile with our two C files. Once you have downloaded them all in the same place. Open two terminal windows. On one, write Make client, on the other, run Make Server. Make sure they are all in the same directory.

Running without a Makefile: You can run without a Makefile, you just need to have two terminals open, and then run both the client.c and the server.c in two separate terminals. Give your CLI command inputs on the client terminal, which is processed in server and you see the results in client terminal. Then on the client terminal, you can enter the commands you want and you will get the outputs there. If you get the error "bind failed: Address already in use" enter "kill -9 $(lsof -t -i:5564)" and rerun the commands.

Input Validation for the Project:

```
parsed[m][i]=='\"' || parsed[m][i]==' ' || parsed[m][i]=='\\' || parsed[m][i]=='\'' ||
parsed[m][i]=='\n' || parsed[m][i]=='\t' || parsed[m][i]=='\r'  || parsed[m][i]=='\b'
|| parsed[m][i]=='\v'  || parsed[m][i]=='\a'
```

Special characters are limited in the single line commands so that the commands function properly. Commands such as cd, ping and man have been limited because they would break the functionality of the server. Ctrl+C doesn't work sometimes so it has been warned against.

```
if(strcmp(parsed[0]=='\"' || parsed[0]==' ' || parsed[0]=='\\' || parsed[0]=='\'' ||
parsed[0]=='\n' || parsed[0]=='\t' || parsed[0]=='\r'  || parsed[0]=='\b'  ||
parsed[0]=='\v'  || parsed[0]=='\a')
```

No special characters or spaces are allowed for piped commands. The client will keep asking for input till there's proper input in all cases.

```
if(!((inputString[j]>='a' && inputString[j]<='z')  || (inputString[j]>='A' &&
inputString[j]<='Z') ||inputString[j]=='|'))
```

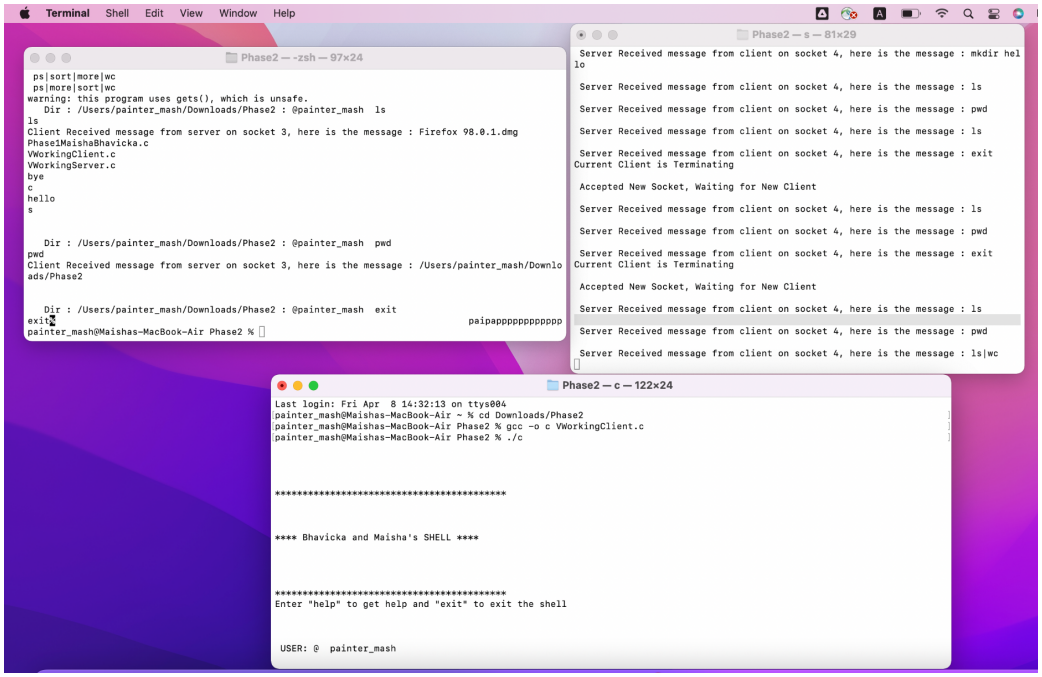
Bhavicka Mohta (bm3001)
Maisha Mahrin (mm10294)

How to exit Client:

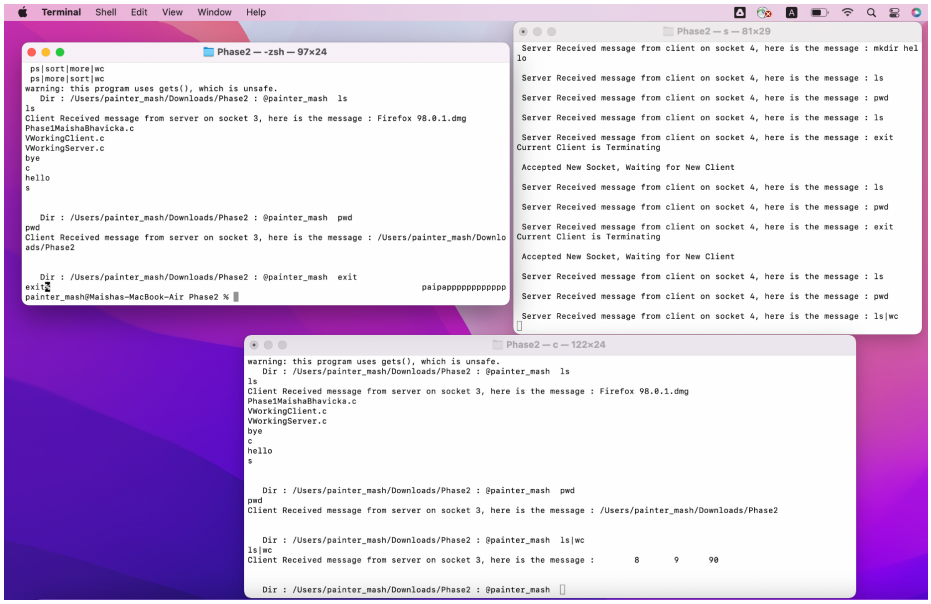
To get out of the terminal, use *exit* command. This will exit the current client and the server will wait for the connection of a new client.

Adding a New Client after Exiting from an old one:

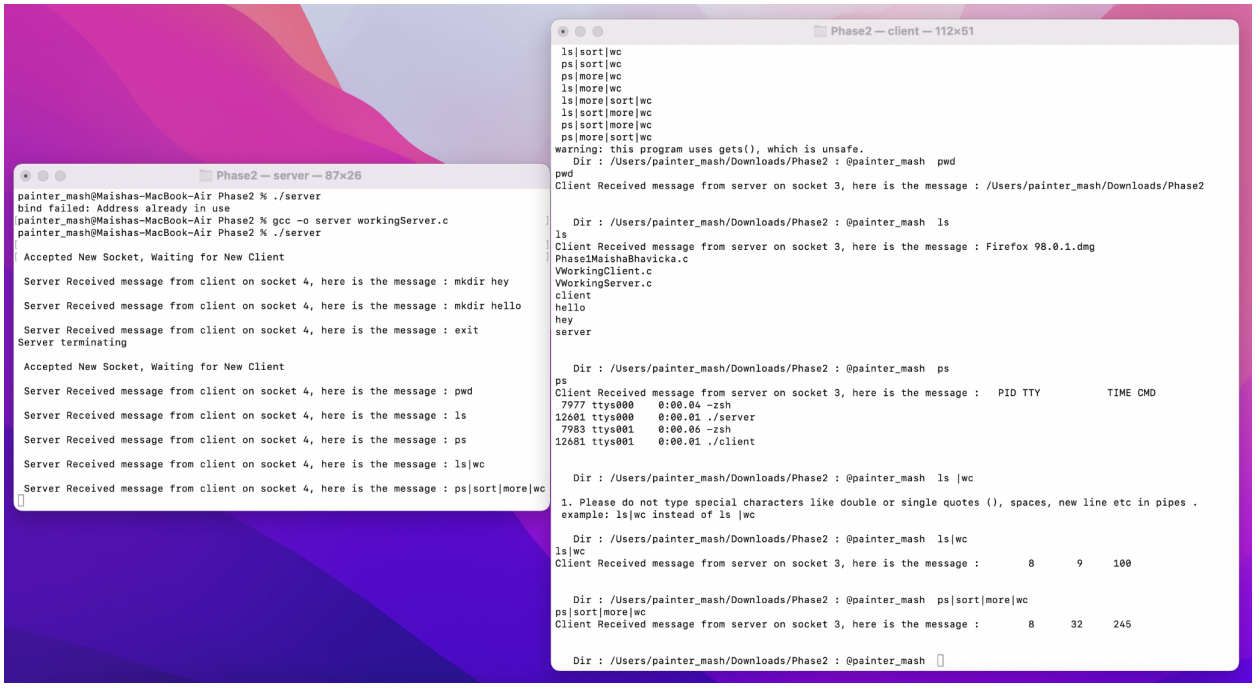
Take out a separate terminal and run the .c file using gcc -o c VWorkingClient.c. Start using the new client without an issue. The same terminal as the former exited client might not work in the remote server. So, nevertheless, take out a new terminal everytime you add a new client.



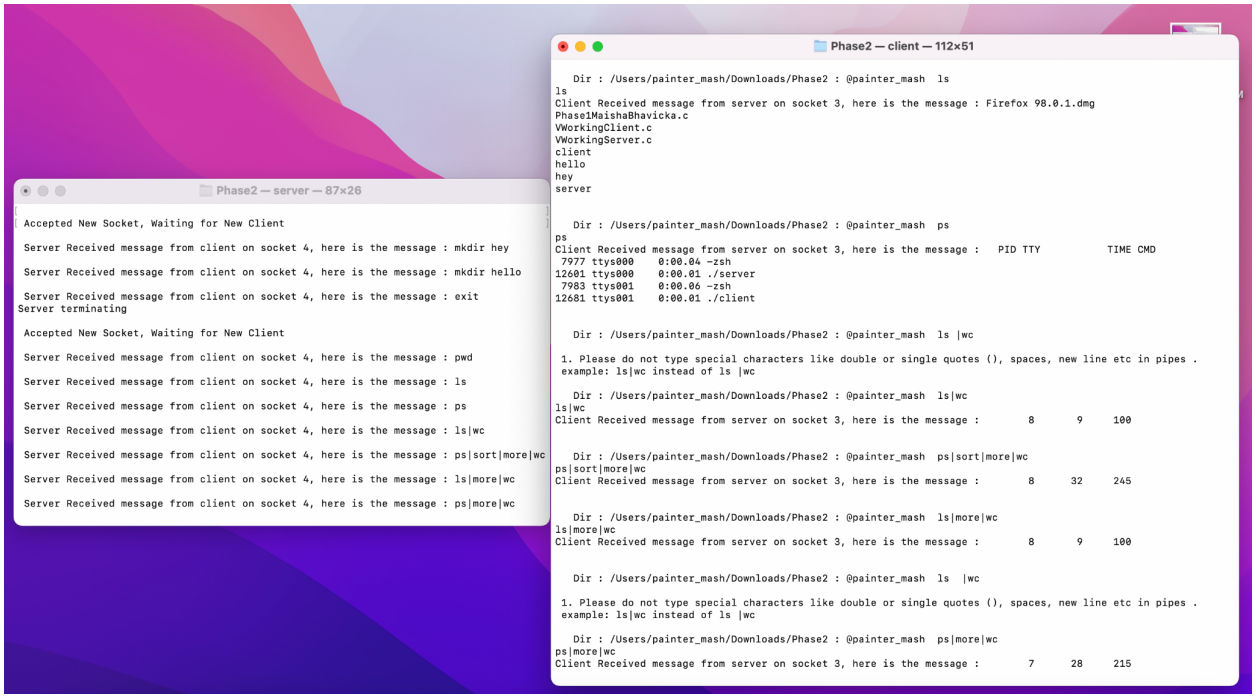
Bhavicka Mohta (bm3001)
Maisha Mahrin (mm10294)



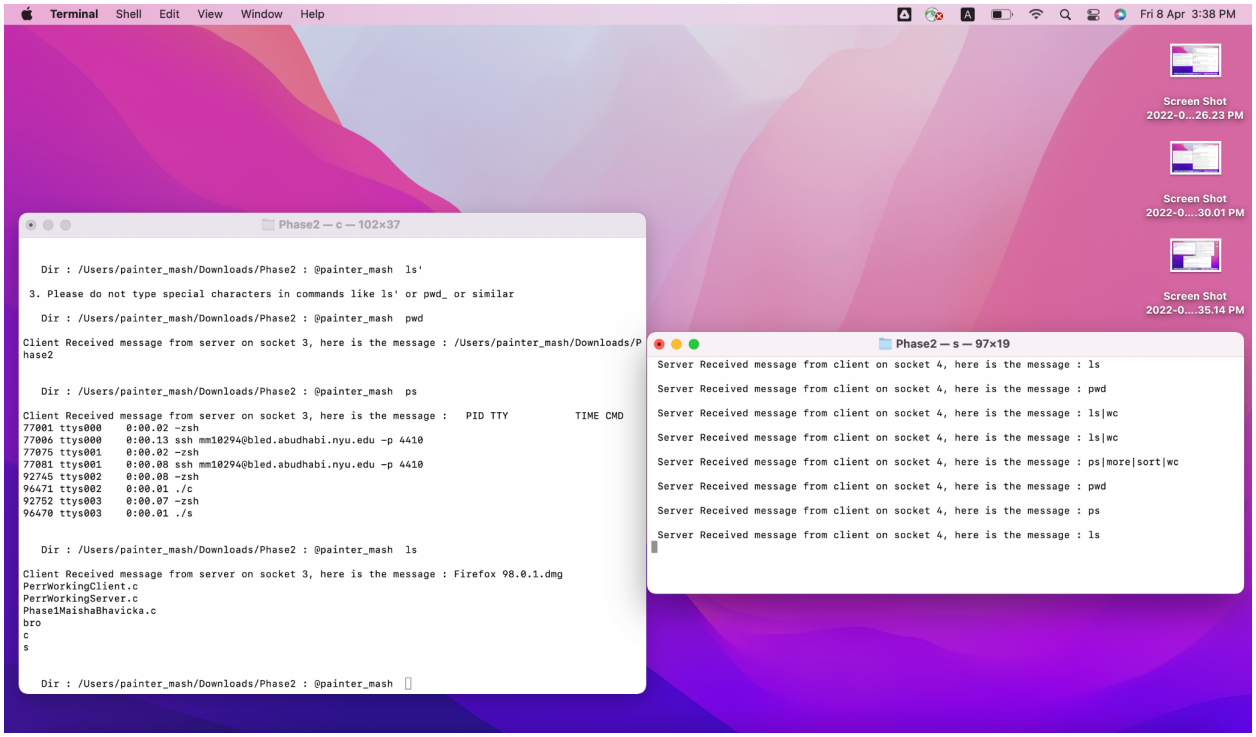
Test Cases: Some Commands Without Error or Typo:



Bhavicka Mohta (bm3001)
Maisha Mahrin (mm10294)



Test Cases: Some Commands With Error or Typo, extra space, special character etc.:



Bhavicka Mohta (bm3001)
Maisha Mahrin (mm10294)

Comparison that the bash terminal and the normal shell give the same output as our built shell:

Aside from the limitation on commands and limitation on Ctrl+C, our shell in the server-client communication works smoothly.

Overall working on this phase of the project was a great learning experience in both making a shell and also debugging. We feel better prepared for the next one now. We would also like to thank our TA Shan Randhawa for his constant help and guidance.

Reference:

1. www.geeksforgeeks.org/taking-string-input-space-c-3-different-methods
2. https://www.geeksforgeeks.org/socket-programming-cc/
3. Codegrepper for exit keyword comparison
4. https://web.stonehill.edu/compsci/CS314/Assignments/Assignment2.pdf
5. https://en.wikipedia.org/wiki/Escape_sequences_in_C#Table_of_escape_sequences