# Operating Systems
# Final Project Report

Spring 2022

Maisha Mahrin
Bhavicka Mohta

Bhavicka Mohta (bm3001)
Maisha Mahrin (mm10294)

# Introduction

In this project we had to create our own remote shell that simulates some services of Operating Systems including the “current Linux shell features, processes, threads, communication, scheduling, etc.” We achieved this through 4 phases throughout.

# Phase 1 Report

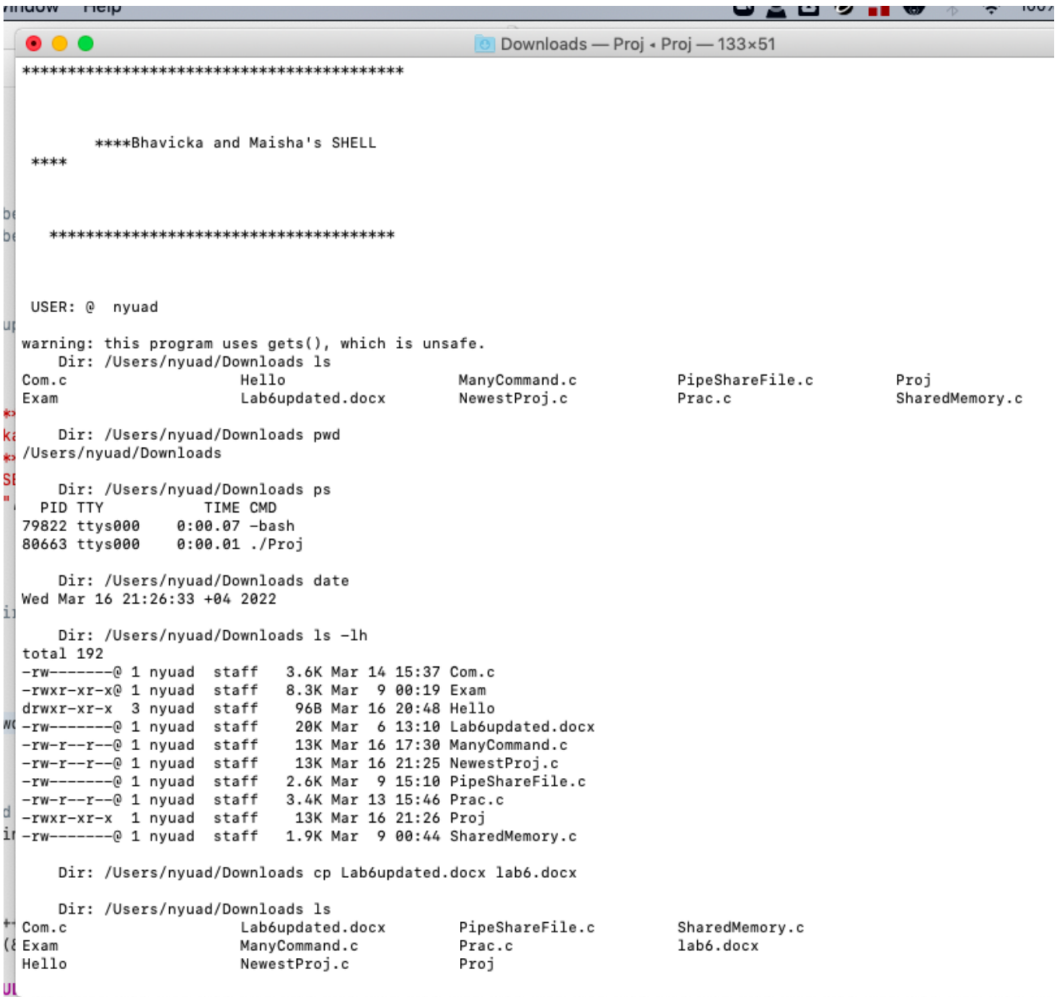
To complete phase 1 of the project, we started off by researching how to implement a basic shell in C. This gave us an idea of what to do, and we started off with an infinite loop in main() which would take continuous user input. We also added the condition that if the input was “exit” the user will exit our shell.

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.

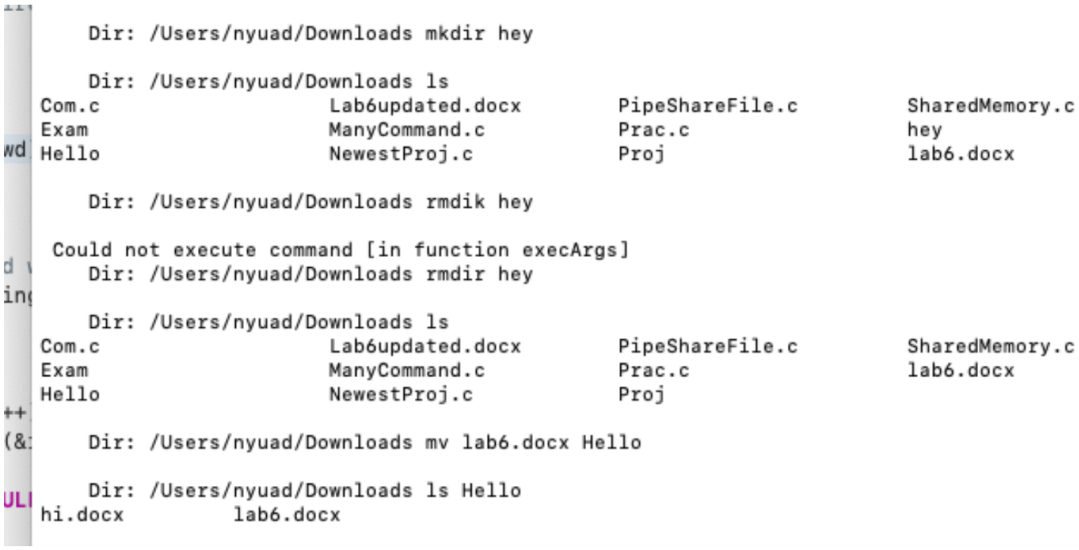
We also added a help function (which is called by entering “help”) to give the user more details about what can and cannot be entered. Currently all compound commands work if they’re entered singly without pipes. A thing we wanted to implement but didn’t get the time to was combining compound commands (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.

Bhavicka Mohta (bm3001)
Maisha Mahrin (mm10294)

Here are some of the sample input and output test cases to see for yourself:



Bhavicka Mohta (bm3001)
Maisha Mahrin (mm10294)



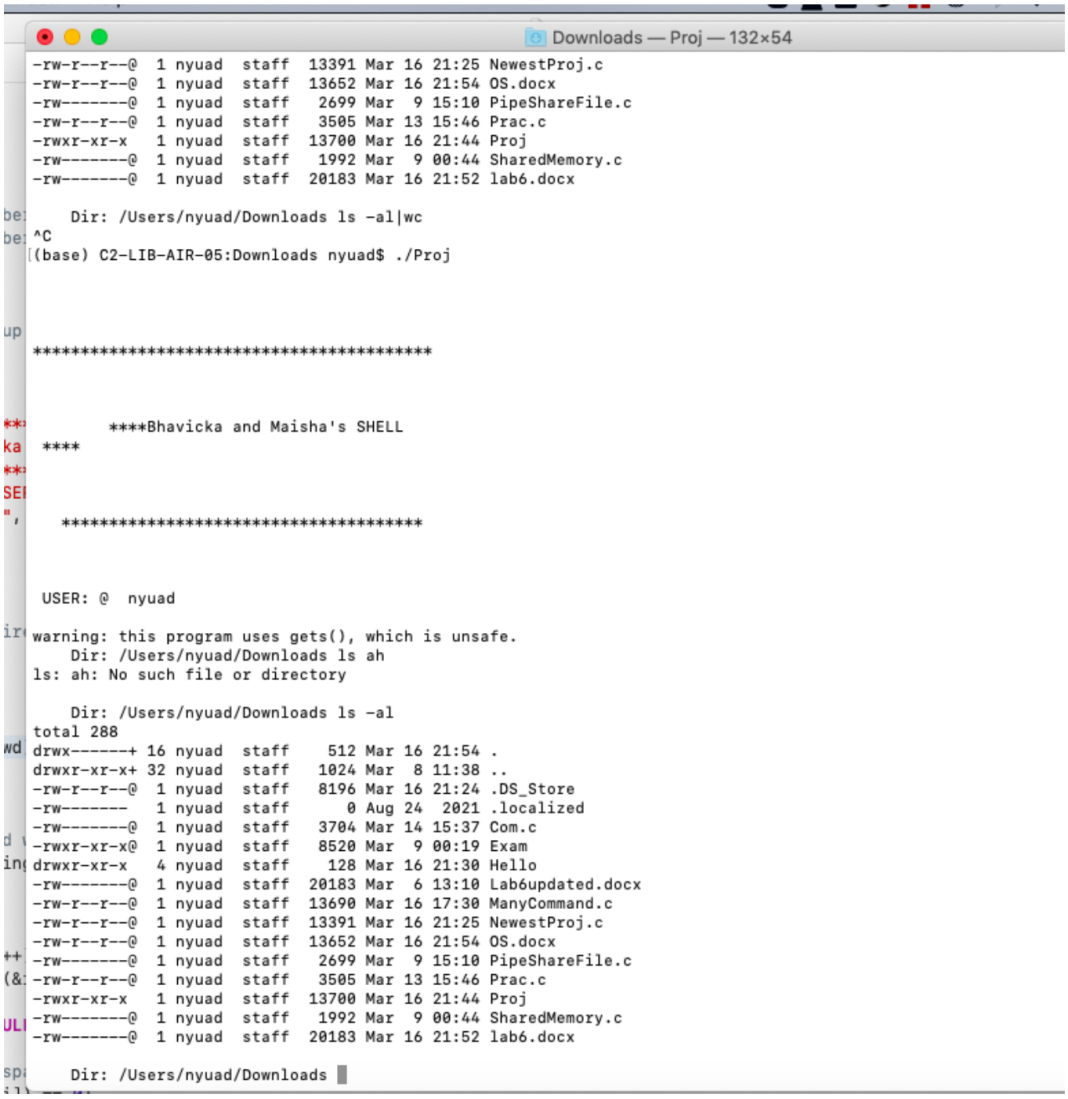
```
    Dir: /Users/nyuad/Downloads mkdir hey

    Dir: /Users/nyuad/Downloads ls
Com.c                   Lab6updated.docx        PipeShareFile.c         SharedMemory.c
Exam                    ManyCommand.c           Prac.c                  hey
Hello                   NewestProj.c            Proj                    lab6.docx

    Dir: /Users/nyuad/Downloads rmdik hey

 Could not execute command [in function execArgs]
    Dir: /Users/nyuad/Downloads rmdir hey

    Dir: /Users/nyuad/Downloads ls
Com.c                   Lab6updated.docx        PipeShareFile.c         SharedMemory.c
Exam                    ManyCommand.c           Prac.c                  lab6.docx
Hello                   NewestProj.c            Proj
    Dir: /Users/nyuad/Downloads mv lab6.docx Hello

    Dir: /Users/nyuad/Downloads ls Hello
hi.docx         lab6.docx
```

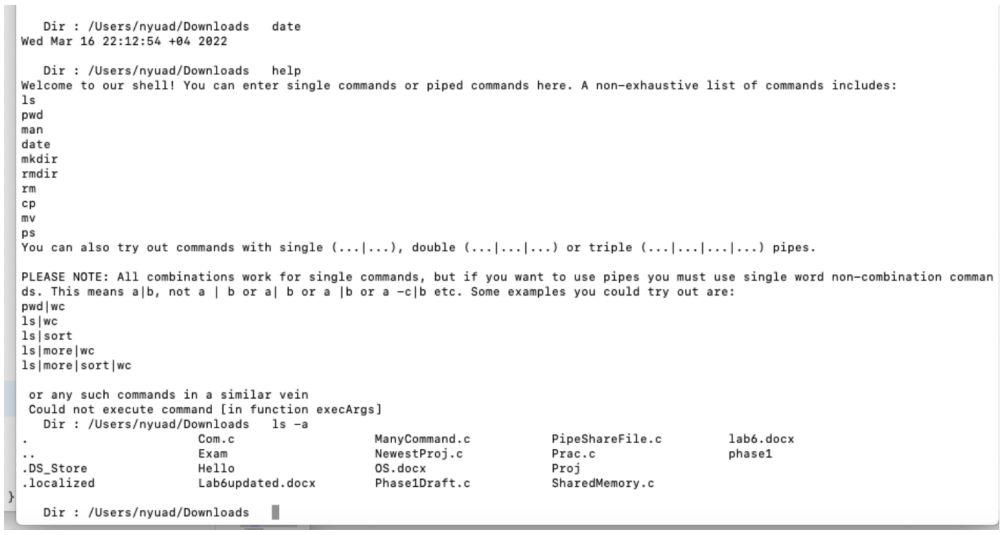
Bhavicka Mohta (bm3001)
Maisha Mahrin (mm10294)

```
    Dir: /Users/nyuad/Downloads ls|wc
      10       10      104

    Dir: /Users/nyuad/Downloads pwd|wc
       1        1       23

    Dir: /Users/nyuad/Downloads ls|sort
Com.c
Exam
Hello
Lab6updated.docx
ManyCommand.c
NewestProj.c
PipeShareFile.c
Prac.c
Proj
SharedMemory.c

    Dir: /Users/nyuad/Downloads ls|sort|wc
      10       10      104

    Dir: /Users/nyuad/Downloads ls|more
Com.c
Exam
Hello
Lab6updated.docx
ManyCommand.c
NewestProj.c
PipeShareFile.c
Prac.c
Proj
SharedMemory.c

    Dir: /Users/nyuad/Downloads ls|more|sort|wc
      10       10      104
```

```
    Dir: /Users/nyuad/Downloads cp Lab6updated.docx lab6.docs

    Dir: /Users/nyuad/Downloads ls
Com.c                   Lab6updated.docx        PipeShareFile.c         SharedMemory.c
Exam                    ManyCommand.c           Prac.c                  lab6.docs
Hello                   NewestProj.c            Proj

    Dir: /Users/nyuad/Downloads rm lab6.docs

    Dir: /Users/nyuad/Downloads ls
Com.c                   Hello                   ManyCommand.c           PipeShareFile.c         Proj
Exam                    Lab6updated.docx        NewestProj.c            Prac.c                  SharedMemory.c

    Dir: /Users/nyuad/Downloads
```

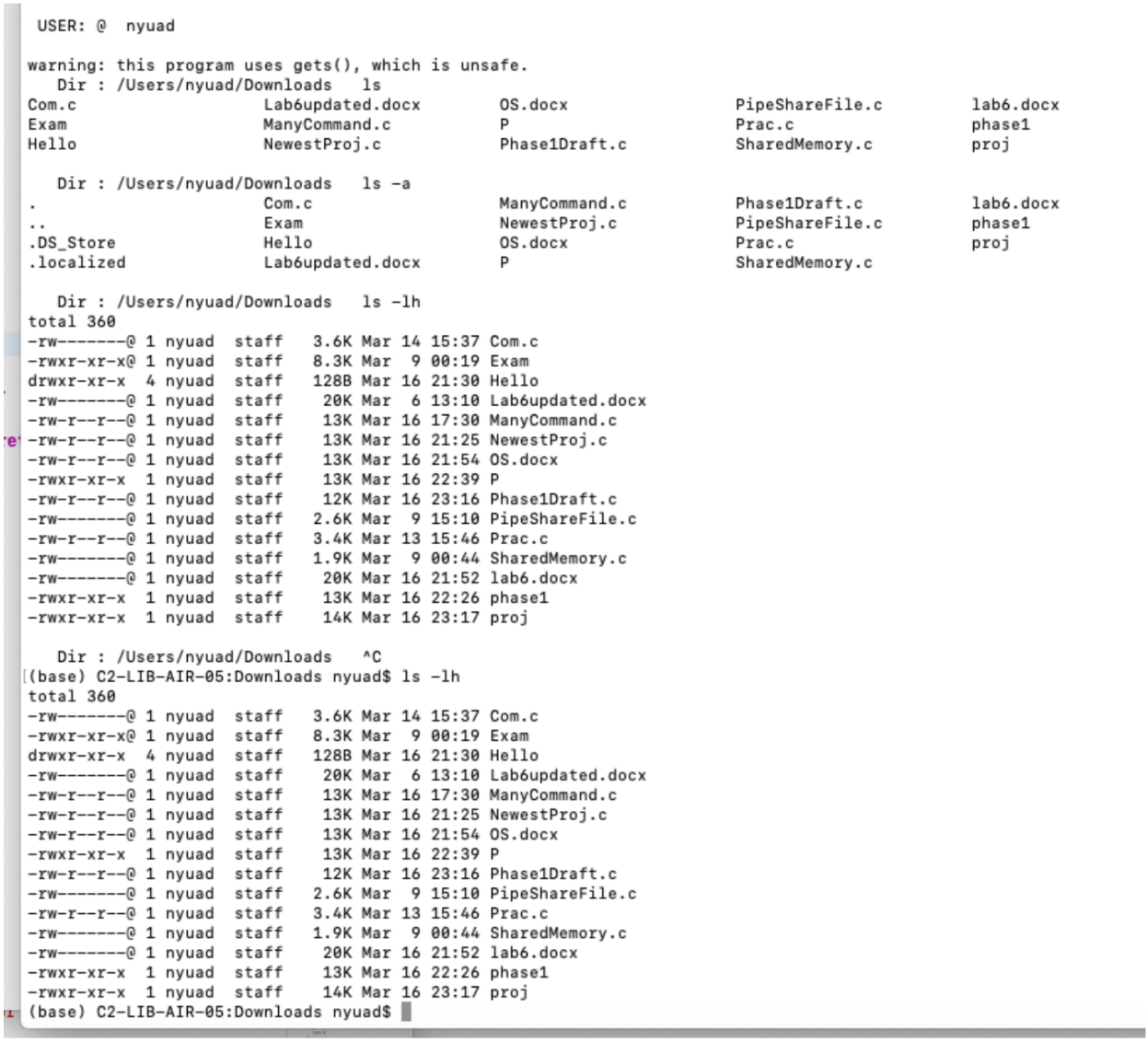
Bhavicka Mohta (bm3001)
Maisha Mahrin (mm10294)



Bhavicka Mohta (bm3001)
Maisha Mahrin (mm10294)

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



```
 USER: @  nyuad

warning: this program uses gets(), which is unsafe.
   Dir : /Users/nyuad/Downloads   ls
Com.c                   Lab6updated.docx        OS.docx                 PipeShareFile.c         lab6.docx
Exam                    ManyCommand.c           P                       Prac.c                  phase1
Hello                   NewestProj.c            Phase1Draft.c           SharedMemory.c          proj

   Dir : /Users/nyuad/Downloads   ls -a
.                       Com.c                   ManyCommand.c           Phase1Draft.c           lab6.docx
..                      Exam                    NewestProj.c            PipeShareFile.c         phase1
.DS_Store               Hello                   OS.docx                 Prac.c                  proj
.localized              Lab6updated.docx        P                       SharedMemory.c

   Dir : /Users/nyuad/Downloads   ls -lh
total 360
-rw-------@ 1 nyuad  staff   3.6K Mar 14 15:37 Com.c
-rwxr-xr-x@ 1 nyuad  staff   8.3K Mar  9 00:19 Exam
drwxr-xr-x  4 nyuad  staff   128B Mar 16 21:30 Hello
-rw-------@ 1 nyuad  staff    20K Mar  6 13:10 Lab6updated.docx
-rw-r--r--@ 1 nyuad  staff    13K Mar 16 17:30 ManyCommand.c
-rw-r--r--@ 1 nyuad  staff    13K Mar 16 21:25 NewestProj.c
-rw-r--r--@ 1 nyuad  staff    13K Mar 16 21:54 OS.docx
-rwxr-xr-x  1 nyuad  staff    13K Mar 16 22:39 P
-rw-r--r--@ 1 nyuad  staff    12K Mar 16 23:16 Phase1Draft.c
-rw-------@ 1 nyuad  staff   2.6K Mar  9 15:10 PipeShareFile.c
-rw-r--r--@ 1 nyuad  staff   3.4K Mar 13 15:46 Prac.c
-rw-------@ 1 nyuad  staff   1.9K Mar  9 00:44 SharedMemory.c
-rw-------@ 1 nyuad  staff    20K Mar 16 21:52 lab6.docx
-rwxr-xr-x  1 nyuad  staff    13K Mar 16 22:26 phase1
-rwxr-xr-x  1 nyuad  staff    14K Mar 16 23:17 proj

   Dir : /Users/nyuad/Downloads   ^C
(base) C2-LIB-AIR-05:Downloads nyuad$ ls -lh
total 360
-rw-------@ 1 nyuad  staff   3.6K Mar 14 15:37 Com.c
-rwxr-xr-x@ 1 nyuad  staff   8.3K Mar  9 00:19 Exam
drwxr-xr-x  4 nyuad  staff   128B Mar 16 21:30 Hello
-rw-------@ 1 nyuad  staff    20K Mar  6 13:10 Lab6updated.docx
-rw-r--r--@ 1 nyuad  staff    13K Mar 16 17:30 ManyCommand.c
-rw-r--r--@ 1 nyuad  staff    13K Mar 16 21:25 NewestProj.c
-rw-r--r--@ 1 nyuad  staff    13K Mar 16 21:54 OS.docx
-rwxr-xr-x  1 nyuad  staff    13K Mar 16 22:39 P
-rw-r--r--@ 1 nyuad  staff    12K Mar 16 23:16 Phase1Draft.c
-rw-------@ 1 nyuad  staff   2.6K Mar  9 15:10 PipeShareFile.c
-rw-r--r--@ 1 nyuad  staff   3.4K Mar 13 15:46 Prac.c
-rw-------@ 1 nyuad  staff   1.9K Mar  9 00:44 SharedMemory.c
-rw-------@ 1 nyuad  staff    20K Mar 16 21:52 lab6.docx
-rwxr-xr-x  1 nyuad  staff    13K Mar 16 22:26 phase1
-rwxr-xr-x  1 nyuad  staff    14K Mar 16 23:17 proj
(base) C2-LIB-AIR-05:Downloads nyuad$
```

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

Bhavicka Mohta (bm3001)
Maisha Mahrin (mm10294)

2. Codegrepper for exit keyword comparison

3. https://web.stonehill.edu/compsci/CS314/Assignments/Assignment2.pdf

# Phase 2 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:**

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).

Bhavicka Mohta (bm3001)
Maisha Mahrin (mm10294)

**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);

Bhavicka Mohta (bm3001)
Maisha Mahrin (mm10294)

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.

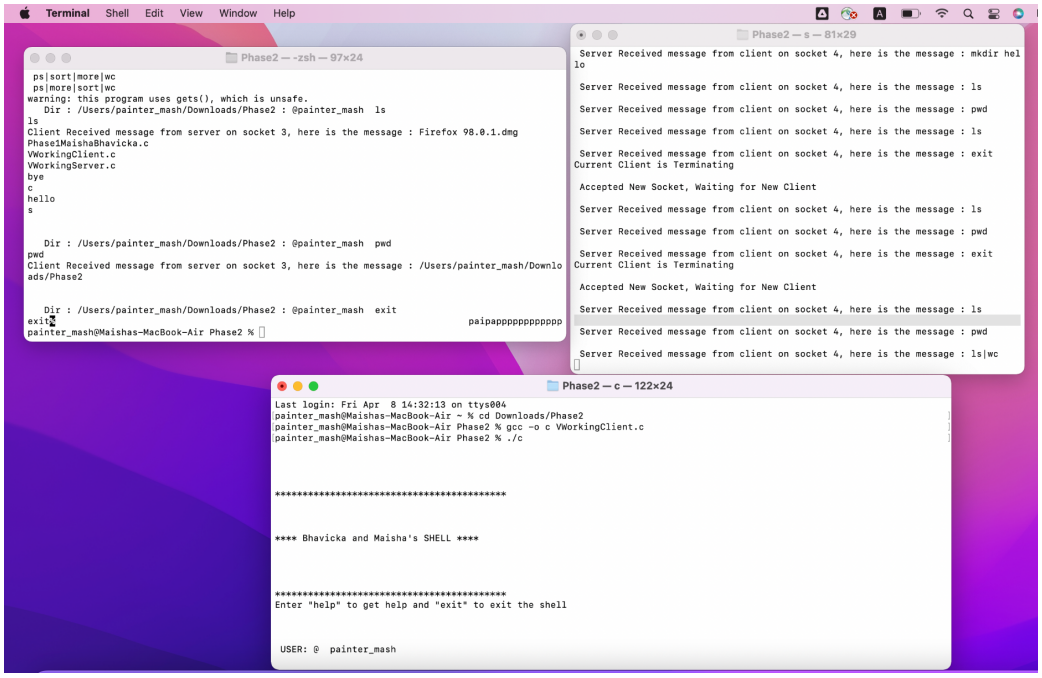
```
if(!((inputString[j]>='a' && inputString[j]<='z')  || (inputString[j]>='A' && inputString[j]<='Z') ||inputString[j]=='|'))
```

**How to exit Client:**

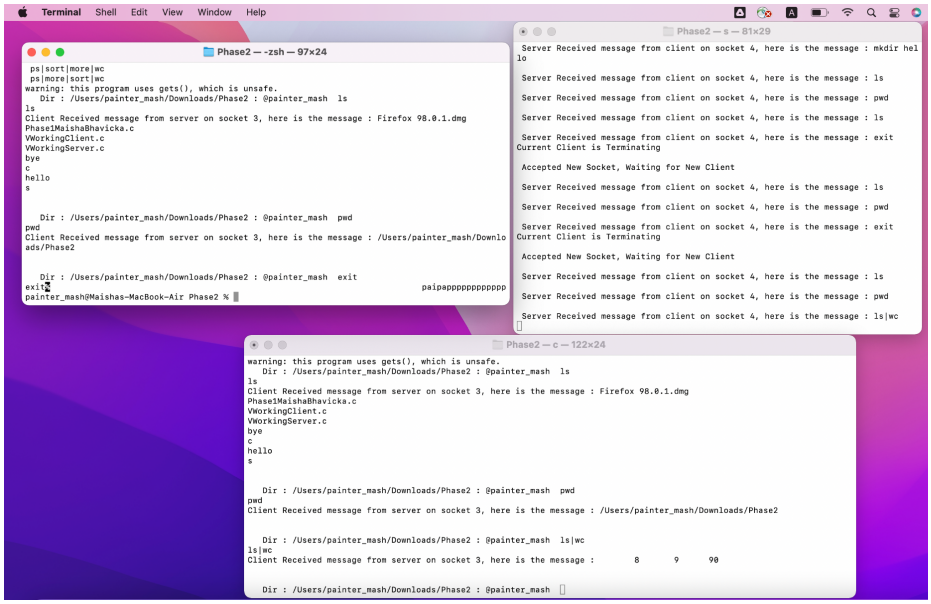
To get out of the terminal, use *exit* command. This will exit the current client and the server will wait for the connection of a new client.

**Adding a New Client after Exiting from an old one:**

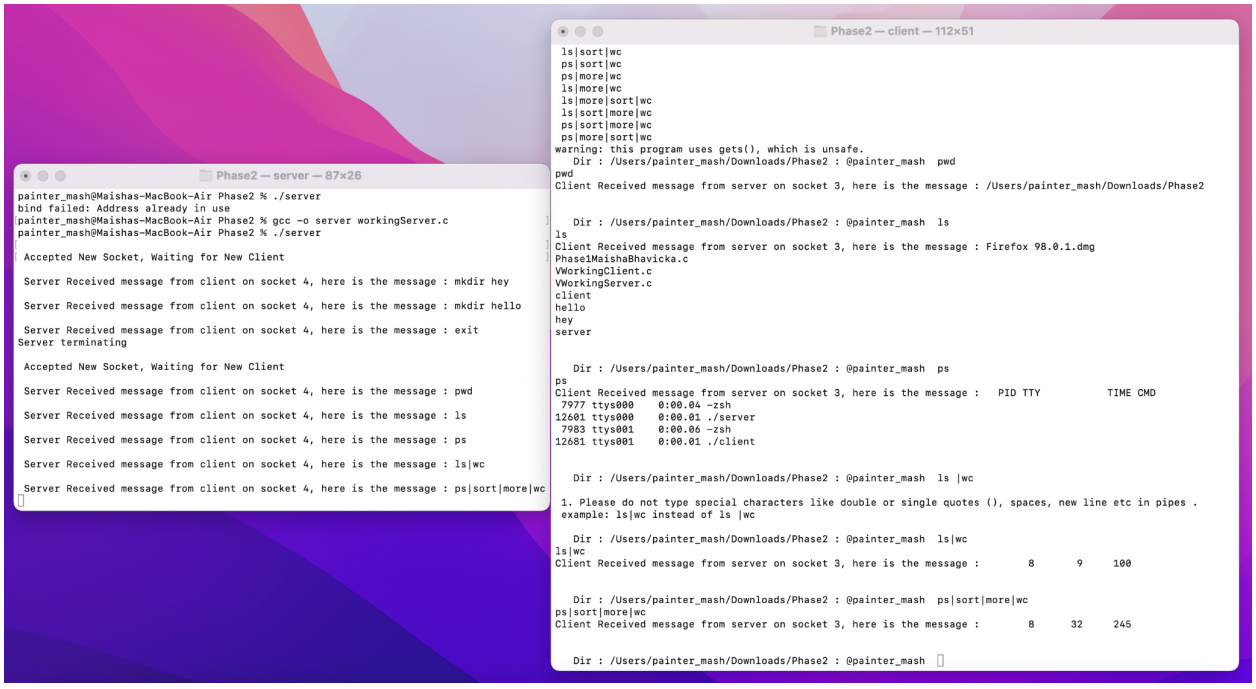
Take out a separate terminal and run the .c file using gcc -o c VWorkingClient.c. Start using the new client without an issue. <u>The same terminal as the former exited client might not work in the remote server. So, nevertheless, take out a new terminal everytime you add a new client.</u>



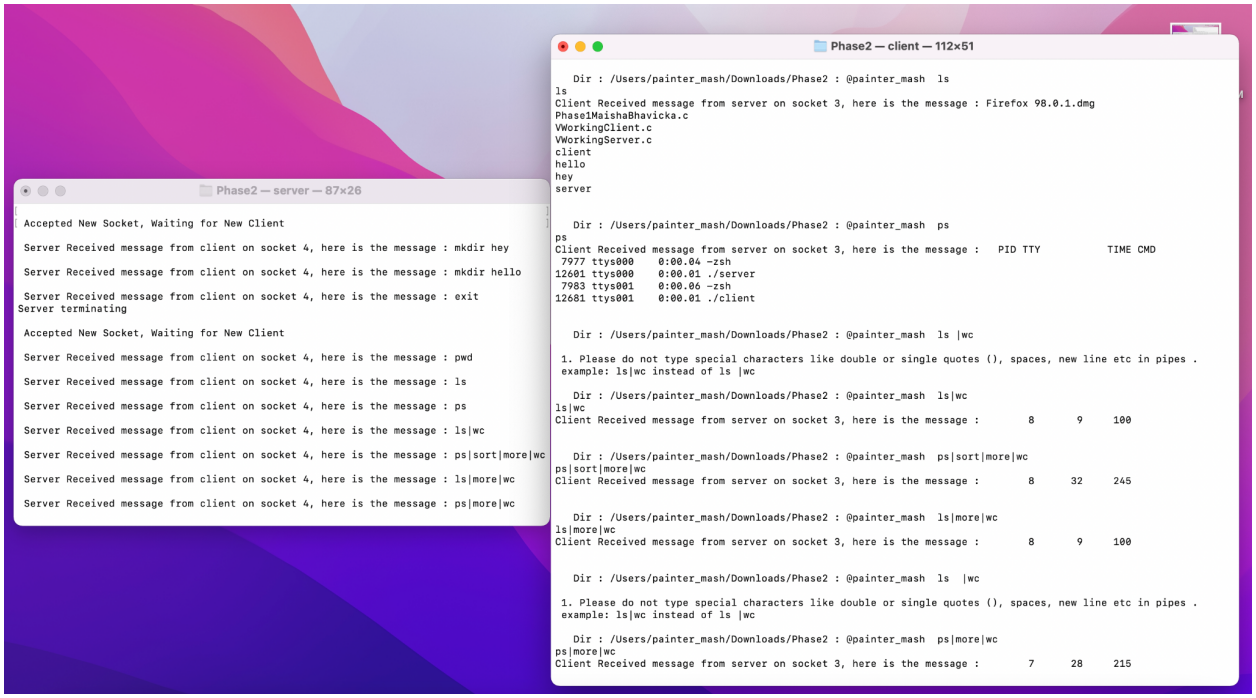
Bhavicka Mohta (bm3001)
Maisha Mahrin (mm10294)



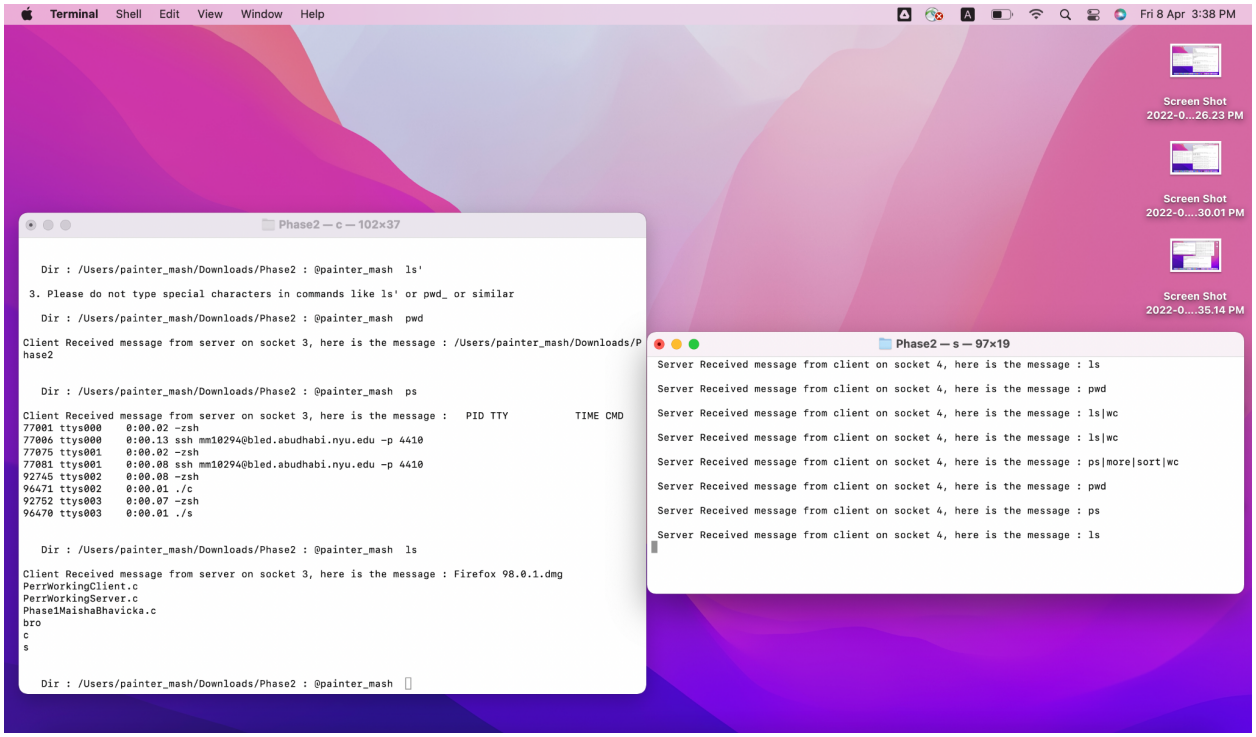

## Test Cases: Some Commands Without Error or Typo:



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.

Problems Faced and Solution:

1. *How to exit from Parent in the main function of the server.c, when the child in the main is handling every operation [here, we actually are terminating the functions with this specific client.c]?*

   We added a global variable flag that becomes 1 in the child when exit has been typed, as it's a global variable, it's value in parent also gets updated and in parent we also close socket if flag==1, after which at the beginning of the loop which keeps server alive till infinity we add flag=0 to initiate the start of a new client.

2. *Changes that allowed us to stop the break in client / server when we input wrong command:*

   We forked a child in the main and the whole program of phase 2 is handled in the child, and we don't have any wait functions in the execArgs functions, aside from the parent in the main function

Bhavicka Mohta (bm3001)
Maisha Mahrin (mm10294)

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

# Phase 3 Report

**Basic CLI Implementation in Server with Multiple Clients (phase 3):**

In this phase, we built upon phase 1 and phase 2 and created a socket communication model where the CLI shell was usable from the client end, and multiple clients could access the server at the same time.

**Overview of Phase 1:**

In phase 1, we built a CLI shell that supported  single, double or triple piped commands, general command line Linus commands etc.

**Overview of Phase 2:**

Bhavicka Mohta (bm3001)
Maisha Mahrin (mm10294)

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

**Code Supports:**

1. Ctrl+C to exit
2. “exit” command to exit
3. Single, double or triple piped commands like:

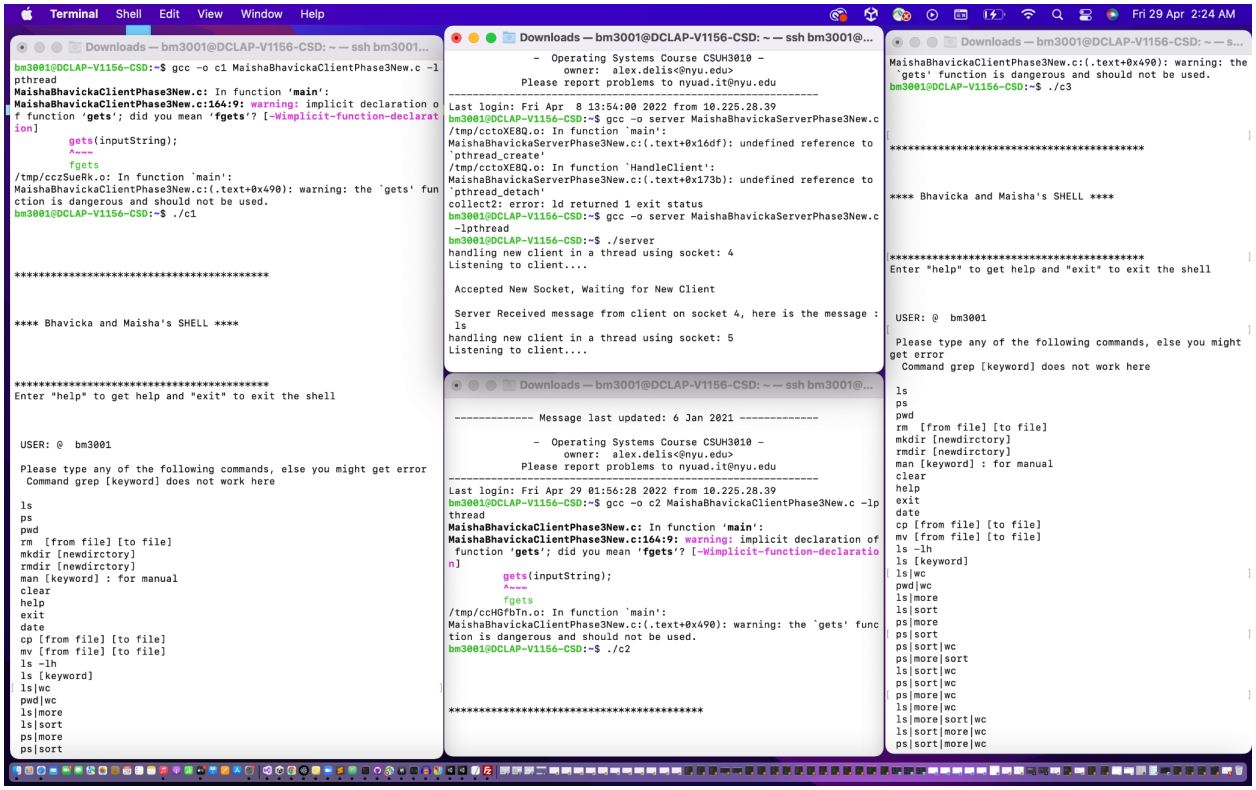
   cat file3.txt | grep “yasin” | tee file4.txt | wc –l
4. Supports error checking from input
5. Inputting invalid commands wait a bit and direct you to inputting the correct command again
6. Handles empty command

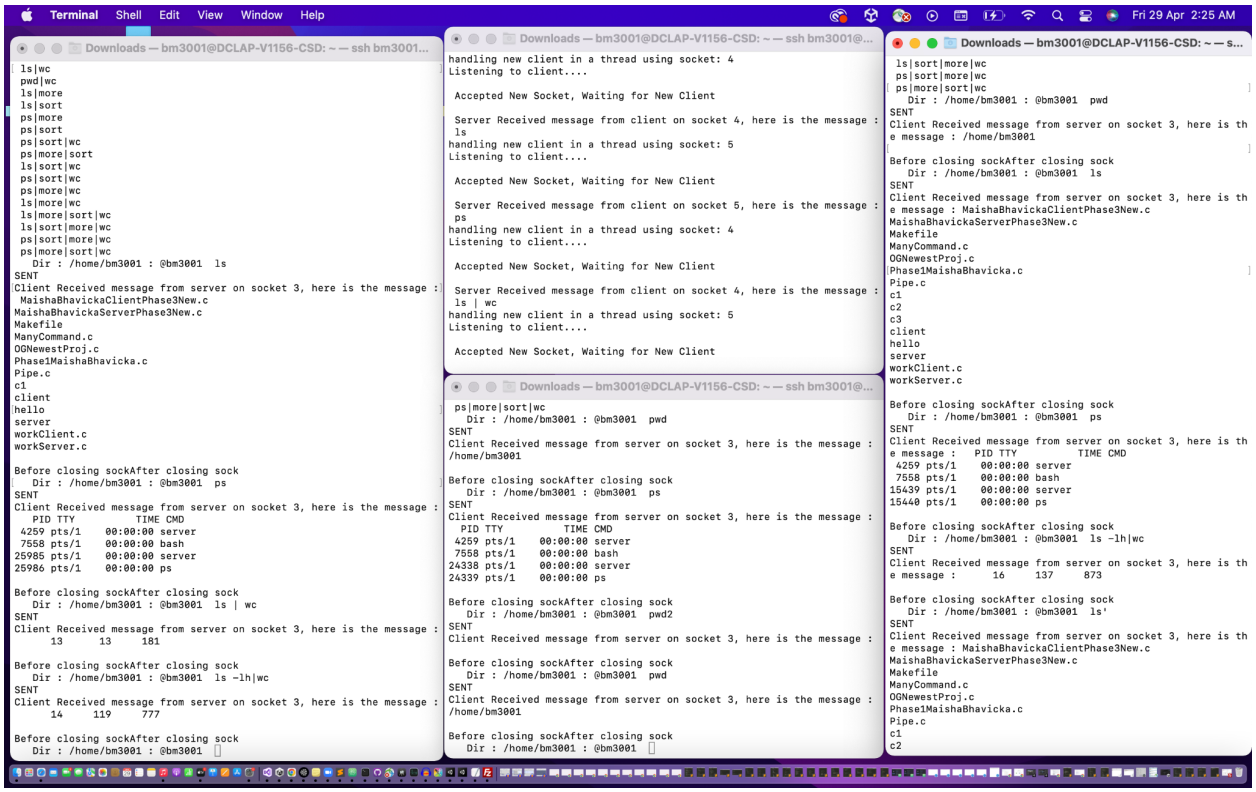
**List of Unsupported Commands:**

1. cd
2. ping
3. man

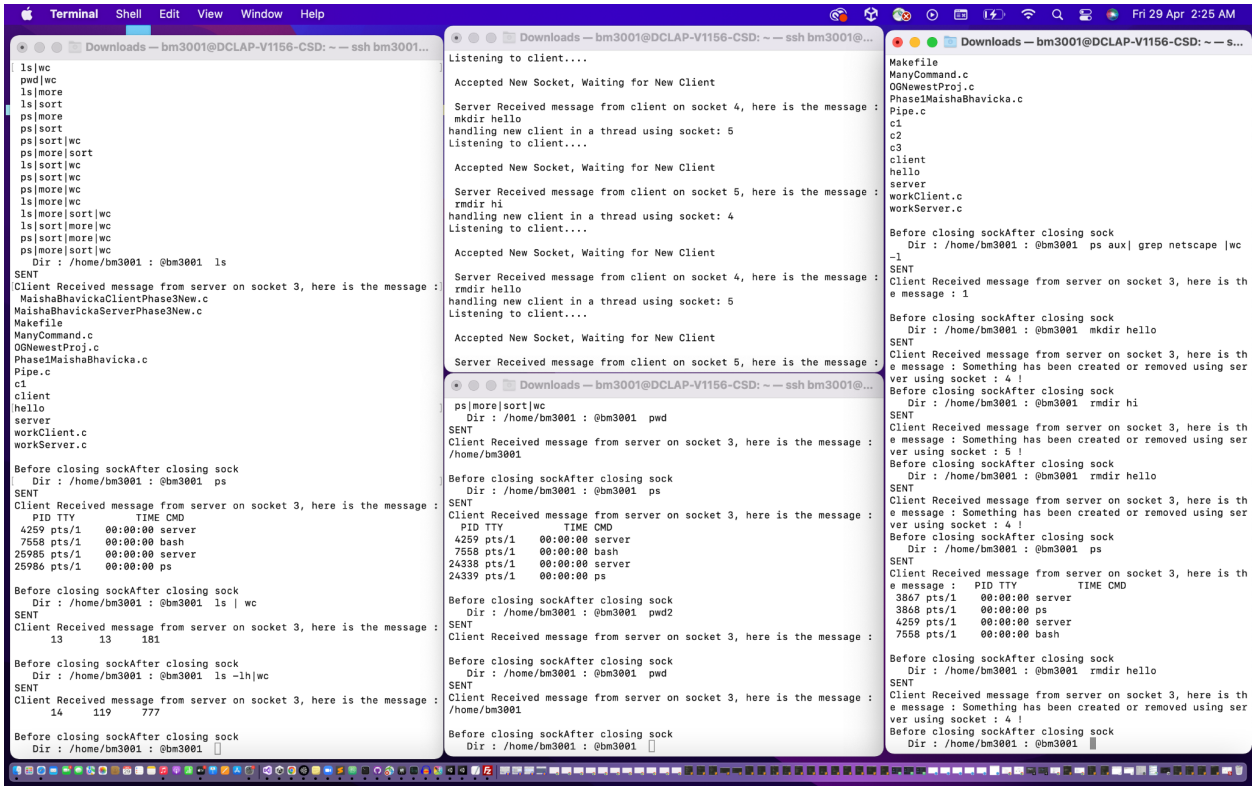
**Multiple Client Code Testing:** With and without error in input:

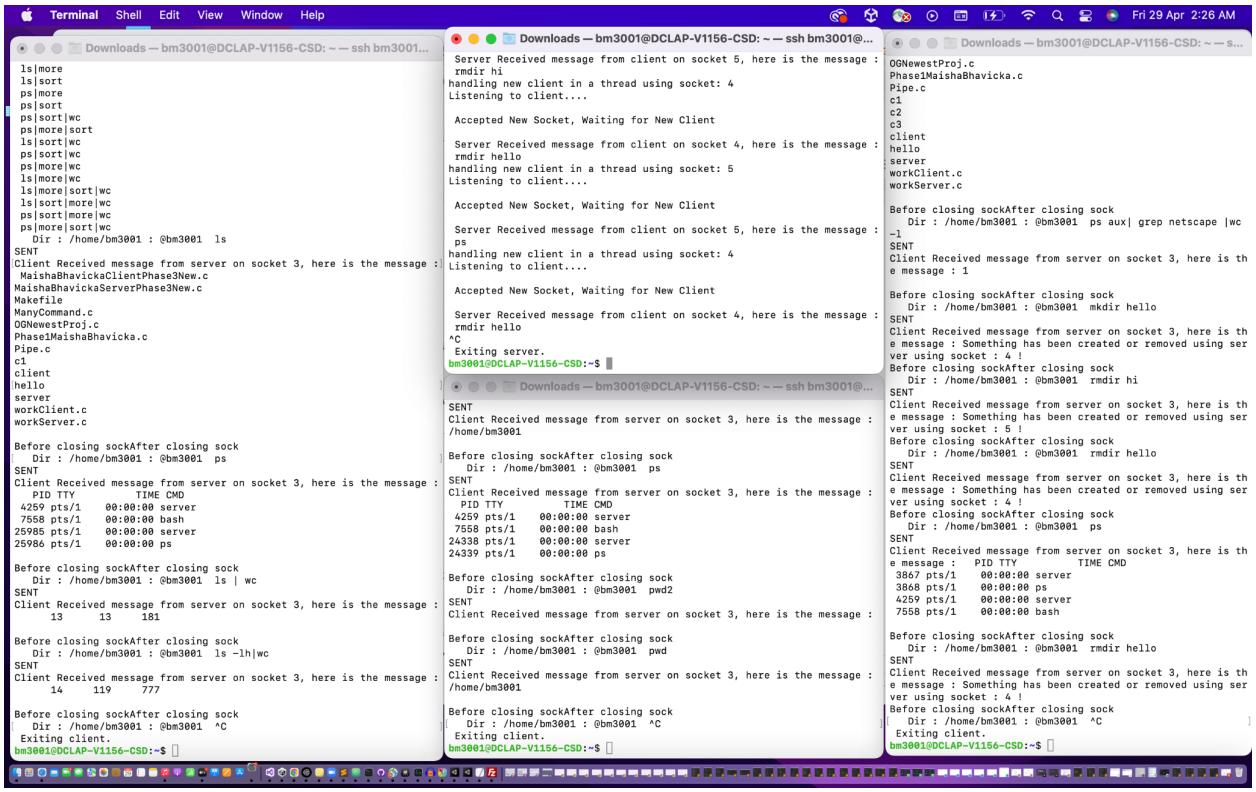
Bhavicka Mohta (bm3001)
Maisha Mahrin (mm10294)



Bhavicka Mohta (bm3001)
Maisha Mahrin (mm10294)



Bhavicka Mohta (bm3001)
Maisha Mahrin (mm10294)

**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 (or more) terminal windows. On one (or more), write make client, on the other, run make server. Make sure they are all in the same directory. After this, you can enter ./server and ./client to run the server and client respectively. Make sure to run the server first, otherwise the connection will fail. You can run make clean to get rid of the object files afterwards. Alternatively, you can also just run make at the beginning which will make all files and then run ./server and ./client on different terminals.

Running without a Makefile: You can run without a Makefile, you just need to have two or more terminals open, and then run both the client.c and the server.c in separate terminals. Make sure to run the server first. 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.

**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.

Also, using Ctrl+C will exit the client or server.

**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)

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:**

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.

**Problems Faced and Solution:**

1. *How to exit from Parent in the main function of the server.c, when the child in the main is handling every operation [here, we actually are terminating the functions with this specific client.c]?*

   We added a global variable flag that becomes 1 in the child when exit has been typed, as it's a global variable, it's value in parent also gets updated and in parent we also close socket if flag==1, after which at the beginning of the loop which keeps server alive till infinity we add flag=0 to initiate the start of a new client.
2. *Changes that allowed us to stop the break in client / server when we input wrong command:*

We forked a child in the main and the whole program of phase 2 is handled in the child, and we don't have any wait functions in the execArgs functions, aside from the parent in the main function

3. *Buffer Overflow and Buffer Not being emptied:*

   We used the buffer polling technique of using the recv function to force read all the characters in the buffer to empty out the buffer before a new request is sent.

4. *Exiting with Ctrl+C:*

   Using signal.h header file and signal(SIGINT, serverExitHandler) function handled this signal operation

# Phase 4 Report

In this phase, we had to upgrade Phase3 with scheduling capabilities in the server, which should simulate the role of scheduler to provide concurrency for serving several clients.

**Data Structure employed:** We use a linked queue where each node is a struct made out of 5 elements: the pointer to the next Node, the thread ID, the round it is on, the semaphore value and the job time remaining. It uses the standard procedures to add and delete nodes.

**Thread Scheduler Function:**

```
void *ThreadScheduler(void *arg)
```

Bhavicka Mohta (bm3001)
Maisha Mahrin (mm10294)

In this phase, we were trying to implement the Shorted Time Remaining First algorithm in Round Robin. Our current implementation has the Shortest Job First, which we were going to upgrade later.

This function is only used to schedule the dummy programs. Normal shell commands proceed as usual.

We have a fixed quantum value of 7. Inside this function, this value is decreased under specific conditions. It finds the shortest job and then lets the dummy program for that run in the critical section of the semaphore.

**Client Handler Function:**

```
// Function that handles the client thread
void *HandleClient(void *arg)
```

In this function, apart from the previously described functionality in phase 3, our shell decides whether it will need to run the scheduler or not, depending on whether any dummy program is running.

**Dummy Program:** Will run for a specified amount of job time (and has a thread ID too) and keep updating job time remaining in a while loop (using sleep(1) to count).

## How to Run

We've provided a Makefile with our three C files. Once you have downloaded them all in the same place, open two (or more) terminal windows. On one (or more), write make client, on the other, run make server. Make sure they are all in the same directory. After this, you can enter

./server and ./client to run the server and client respectively. Make sure to run the server first, otherwise the connection will fail. You can run make clean to get rid of the object files afterwards. To run the simulation on the client, write make dummyProgram.o on the original client terminal, and then enter "./dummyProgram.o" once our shell is running. Alternatively, you can also just run make at the beginning which will make all files and then run ./server and ./client on different terminals, and once our client is running you can run ./dummyProgram.o on there.

Running without a Makefile: You can run without a Makefile, you just need to have two or more terminals open, and then run both the client.c and the server.c in separate terminals using gcc commands which you can refer to the makefile for. Make sure to run the server first. 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.

**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.

Also, using Ctrl+C will exit the client or server

## Test Cases