

SHELL - GSSH

A CUMULATIVE REPORT ON THE
IMPLEMENTATION OF A WORKING
SHELL FOR THE COURSE OPERATING
SYSTEM (CS-3010)

Prepared by: Sarah Al-Towaity
& Gautham Dinesh

01

INTRODUCTION

- 02 - Project Introduction
- 03 - Code File Summaries
- 06 - Execution Instructions

PHASE 1

A Local Command Line Interface Shell

- 08 - Implementation
- 12 - Test Cases

PHASE 2

A Remote Command Line Interface Shell

- 21 - Implementation
- 28 - Test Cases

PHASE 4

A Multithreaded Command Line Interface Shell with Scheduling Capabilities

- 47 - Implementation
- 56 - Test Cases with Demo

02

ABOUT OUR PROJECT

Our shell is called the gssh shell and it is a command line shell implemented with the intention of replicating command execution in a Linux CLI shell. This project is executed in 4 phases. In the first phase, gssh is only a local shell. Implementation is primarily dependent on process creation. In the second phase, gssh becomes a remote shell, handling one individual command at a time via socket communication. The third phase integrated multithreading to allow the server shell, gssh, to handle more than one client simultaneously. The last phase upgrades the shell with scheduling capabilities.

OUR TEAM



SARAH AL-TOWAITY



GAUTHAM DINESH

CODE FILE SUMMARIES

In order to create a modular structure for our implementation, our code has been divided into 3 main modules: 1) Input parsing module, 2) Execution module, and 3) a module that includes the main code implementation for the shell. In total, we have 6 main files at the end of phase 1: 1) main.c, 2) input_parsing.c, 3) input_parsing.h, 4) exec.c, 5) exec.h, and, finally, the 6) Makefile. In phase 2, which upgrades the local shell created in phase 1, a new file, client.c is added.

input_parsing.c and Input_parsing.h

The c file, input_parsing.c includes four functions, which are declared in the associated header file, input_parsing.h, that essentially perform the function of retrieving input from the user, parsing it, and tokenizing it based on special delimiters.

1. read_input(): This function dynamically allocated a buffer that stores user input. The function also dynamically expands the size of the buffer if exceeded. The function then returns the string read from the user.
2. tokenize(): This function takes in a string and dynamically allocates an array that will hold string elements. The function will then parse through the string, separating it by white space characters. This function also performs dynamic memory expansion to the array holding the string tokens, if necessary.

Essentially, this function will be used to retrieve the arguments in vector format to be passed to `execvp` in the `main.c` file.

3. `tokenizePipes()`: This function essentially performs all the functions of `tokenize()`, but with the pipe character, '|', as a delimiter, instead. With the addition of taking an extra parameter, a pointer to an integer, where the number of commands joined by the pipes is saved.

4. `remove_special_chars()`: This function was added after the implementation of phase 2. This function takes a string and removes single and double quotes from the string.

exec.c and exec.h

The `c` file, `exec.c` includes four functions that essentially perform the execution of commands entered by the user depending on the number of pipes in the command. All four functions use the function `execvp`, where the first argument is the command and the second argument is a vector (array) of the command's arguments:

1. `exec()`: This function executes pipeless commands.
2. `execPipe()`: This function executes commands with one pipe.
3. `execTwoPipes()`: This function executes commands with two pipes.
4. `execThreePipes()`: This function executes commands with three pipes.
5. `cd_command()`: This function executes the change of directory command, `cd`.

main.c

The main code for the shell is in the `C` file, `main.c`. More specifically, the shell loop code is wrapped in the function `gss_loop()`. The general structure of the loop depends on the variable `res`, which is set to 1, allowing the loop to run.

05

This variable is maintained by the value, 1, which is always returned by the call to the functions in `exec.c` in the case of normal execution. Within the loop, the type of `exec` function to run depends on the number of commands that is determined by the `tokenizePipes()` function.

client.c

This file is added in phase 2 as a result of adding remote capabilities to the `gssh` shell via socket communication. This file creates a socket that connects to the server with every command that the user enters to be served by the server, our main shell.

Makefile

This file will allow us to compile all our `c` code and header files by simply running the command `make`. Our `Makefile` will be routinely updated in every phase in anticipation of the addition of more files, if necessary.

06

EXECUTION INSTRUCTIONS

Phase 1:

1. Change the current directory to OS_Shell
2. Compile code by running the command "make".
3. Execute using "./shell"

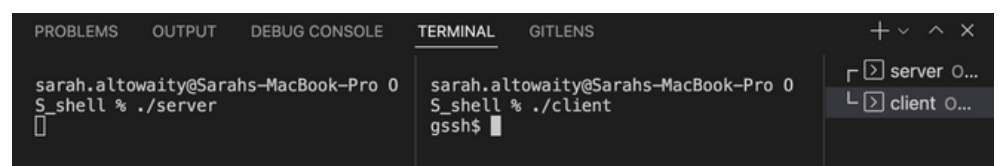
```
sarah.altowaity@Sarahs-MacBook-Pro OS_shell % make
gcc -c main.c
cc -c -o input_parsing.o input_parsing.c
cc -c -o exec.o exec.c
gcc -o shell main.o input_parsing.o exec.o
sarah.altowaity@Sarahs-MacBook-Pro OS_shell %
```

```
sarah.altowaity@Sarahs-MacBook-Pro OS_shell % ./shell
gssh$
```

Phase 2:

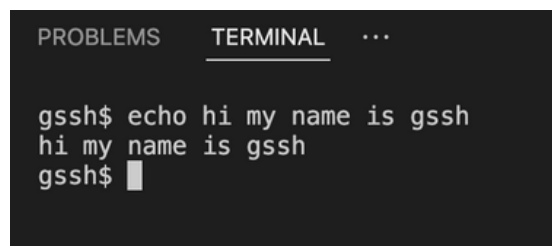
1. Change the current directory to OS_Shell
2. Compile code by running the command "make".
3. Open a shell window and execute the server using "./server".
4. Open another shell window and execute the client using "./client"
5. Type commands in the shell running the client file

```
sarah.altowaity@Sarahs-MacBook-Pro OS_shell % make
cc -c -o client.o client.c
cc -c -o input_parsing.o input_parsing.c
gcc -o client client.o input_parsing.o
gcc -c main.c
cc -c -o exec.o exec.c
gcc -o server main.o input_parsing.o exec.o
sarah.altowaity@Sarahs-MacBook-Pro OS_shell %
```



PHASE 1

THIS PHASE REQUIRES THE IMPLEMENTATION OF A LOCAL SHELL/COMMAND LINE INTERFACE THAT REPLICATES SOME FEATURES AND COMMANDS FROM LINUX. IN ADDITION TO SUPPORTING THE EXECUTION OF AT LEAST 15 COMMANDS, THE WORKING SHELL SHOULD SUPPORT SOME COMPOSED COMMANDS. OUR SHELL WILL SUPPORT THE PIPE COMMAND (UP TO THREE PIPES).

A terminal window with a dark background. At the top, there are three tabs: 'PROBLEMS', 'TERMINAL' (which is selected and underlined), and '...'. The terminal shows the following text: 'gssh\$ echo hi my name is gssh', followed by the output 'hi my name is gssh' on the next line, and then 'gssh\$' followed by a cursor on the third line.

```
PROBLEMS  TERMINAL  ...
gssh$ echo hi my name is gssh
hi my name is gssh
gssh$
```

Our shell is called the gssh shell, the basic skeleton of the implementation of which involves tokenizing the input and then executing the parsed and tokenized input, with specific handling of pipes in both the tokenization and the execution. In this phase, our basic structure of the shell involves a running loop that tokenizes and executes user commands locally. Built within the loop is the added ability to exit our shell by simply inputting the user command "exit".

08

IMPLEMENTATION

Main Shell Loop

Our shell is essentially executed by the main while loop in the function `gss_loop()` in `main.c`.

The function declares the following variables:

1. `line`: this is a string that will hold the line read from the user as input
2. `cmds`: this is an array of strings that will come to hold the command and associated arguments of every entered commands.
3. `piped_array`: this is an array of strings that have been separated by pipe. Each element of the array is a command and, if available, its arguments in string format.
4. `no_cmds`: an integer that holds the number of commands in the string read from user input.
5. `res`: is an integer, first initialized to 1 and maintained by the `exec` function, that controls the main loop.

The while loop first prints the shell prompt. "gssh", reads input from the user, and tokenizes it by pipe, splitting the string based on the '|' character. The `tokenizePipes()` function performing this operation takes in a pointer to `no_cmds` and sets its value to the number of commands the user inputted based on the number of pipes.

From there, we enter a switch statement based on the number of commands, executing the suitable version of the `exec.c` functions based on how many pipes the user entered. In every iteration of the loop, we free `piped_array` and `cmds`, which are dynamically allocated in every invocation of the `tokenize` variant functions.

Input Reading

Input is read from the user character by character using the `getchar()` function until the end of file character or the newline character at which point the null character is added to the end of the string, which is dynamically allocated in the function, `read_input()`.

Input Tokenization

Input is first tokenized based on the pipe character as a delimiter to extract commands. Commands are then tokenized based on whitespace characters, `'\n'` and `'\t'`, to get them in a format ready to be passed into the arguments of the `execvp()` system call in the `exec` functions in `exec.c`

The `tokenize` functions dynamically allocate an array of strings and use `strtok()` from the `string.h` header file to populate the array with the tokens. The functions also make sure to dynamically expand the array holding the tokens, if necessary using `realloc()`. The `tokenize` functions then return the array of strings.

10

Command Execution

Commands are executed by the `exec` functions in `exec.c` depending on the number of pipes.

1.No pipes

The command is executed by invoking the `exec()` function in `exec.c` which creates a child that invokes the `execvp()` system call using the first string in the passed array and the entire array as the second argument.

2. Multiple Pipes (> 1)

These commands are handled by the other variants of the `exec` function depending on the number of pipes. The basic idea is to create a child process to handle every command in the `piped_array`. For every child process created to handle the commands in the middle of the `piped_array` (not the first or the last), we re-direct input from standard input to read from one pipe (which will be the output of the previous command) and re-direct output from standard output to the other pipe (which will read this output as input to the next command). The pipe handling of the first command only re-directs standard output so that output is written on the pipe of the second command. As for the last command, standard input is re-directed from the standard input stream to the pipe, where the second-to-last output has been written, for reading. The final output is displayed next.

Error Handling

The implementation of this phase handles cases of failed forking and failed calls to `execvp()` by displaying the error prompt to the user. See the next section for illustrations.

Shell Termination

The shell terminates once the user enters the command "exit".

12

TEST CASES

Note: All the examples shown below have also outputted the same result when the shell was executed in the remote server.

15 single commands

1. mkdir

2. ls

```
gssh$ mkdir test
gssh$ ls
Makefile
Operating System - Phase 1.pdf
README.md
client
client.c
client.o
exec.c
exec.h
exec.o
input_parsing.c
input_parsing.h
input_parsing.o
main.c
main.o
server
shell
shell.zip
test
gssh$
```

output of both ls and mkdir

3. cat

```
gssh$ cat input_parsing.h
#ifndef INPUT_PARSING
#define INPUT_PARSING

char* read_input();
char** tokenize(char*);
char** tokenizePipes(char*, int*);
void remove_special_chars(char*);

#endifgssh$
```

13

4. sort

```
gssh$ ls -l | sort
-rw-r--r-- 1 sarah.altowaity staff 165 Apr 5 04:19
input_parsing.h
-rw-r--r-- 1 sarah.altowaity staff 337 Apr 5 04:19
Makefile
-rw-r--r-- 1 sarah.altowaity staff 340 Apr 5 04:19
exec.h
-rw-r--r-- 1 sarah.altowaity staff 1663 Mar 29 14:08
README.md
-rw-r--r-- 1 sarah.altowaity staff 2040 Apr 5 04:19
client.o
-rw-r--r-- 1 sarah.altowaity staff 2152 Apr 5 04:19
main.o
-rw-r--r-- 1 sarah.altowaity staff 2288 Apr 5 04:19
input_parsing.o
-rw-r--r-- 1 sarah.altowaity staff 2408 Apr 5 04:19
client.c
-rw-r--r-- 1 sarah.altowaity staff 2745 Apr 5 04:19
main.c
-rw-r--r-- 1 sarah.altowaity staff 3634 Apr 5 04:19
input_parsing.c
-rw-r--r-- 1 sarah.altowaity staff 5816 Apr 5 04:19
exec.o
-rw-r--r-- 1 sarah.altowaity staff 12597 Apr 5 04:19
exec.c
-rw-r--r-- 1 sarah.altowaity staff 259966 Mar 29 14:08
shell.zip
-rw-r--r--@ 1 sarah.altowaity staff 273104 Mar 29 14:08
Operating System - Phase 1.pdf
-rwxr-xr-x 1 sarah.altowaity staff 50528 Apr 5 04:17
shell
-rwxr-xr-x 1 sarah.altowaity staff 50552 Apr 5 04:19
client
-rwxr-xr-x 1 sarah.altowaity staff 50848 Apr 5 04:19
server
drwxr-xr-x@ 2 sarah.altowaity staff 64 Apr 5 10:30
test
total 1488
gssh$
```

5. grep

```
gssh$ ls | grep h
Operating System - Phase 1.pdf
exec.h
input_parsing.h
shell
shell.zip
gssh$
```

6. echo

```
gssh$ echo hi my name is gssh
hi my name is gssh
gssh$
```

14

7. touch

```
gssh$ touch test.txt
gssh$ ls
Makefile
Operating System - Phase 1.pdf
README.md
client
client.c
client.o
exec.c
exec.h
exec.o
input_parsing.c
input_parsing.h
input_parsing.o
main.c
main.o
server
shell
shell.zip
test
test.txt
gssh$ █
```

8. mv

```
gssh$ mv test.txt test/
gssh$ ls
Makefile
Operating System - Phase 1.pdf
README.md
client
client.c
client.o
exec.c
exec.h
exec.o
input_parsing.c
input_parsing.h
input_parsing.o
main.c
main.o
server
shell
shell.zip
test
gssh$ ls test
test.txt
gssh$ █
```

9. cp

```
gssh$ ls
Makefile
Operating System - Phase 1.pdf
README.md
client
client.c
client.o
exec.c
exec.h
exec.o
input_parsing.c
input_parsing.h
input_parsing.o
main.c
main.o
server
shell
shell.zip
test
test.txt
gssh$ cp test.txt test/
gssh$ ls test
test.txt
gssh$ █
```

```
gssh$ ls
Makefile
Operating System - Phase 1.pdf
README.md
client
client.c
client.o
exec.c
exec.h
exec.o
input_parsing.c
input_parsing.h
input_parsing.o
main.c
main.o
server
shell
shell.zip
test
test.txt
gssh$ █
```

15

10. rm

The list of files in the working directory is displayed before and after running "rm test.txt"

```
gssh$ ls
Makefile
Operating System - Phase 1.pdf
README.md
client
client.c
client.o
exec.c
exec.h
exec.o
input_parsing.c
input_parsing.h
input_parsing.o
main.c
main.o
server
shell
shell.zip
test
test.txt
gssh$ rm test.txt
gssh$ ls
Makefile
Operating System - Phase 1.pdf
README.md
client
client.c
client.o
exec.c
exec.h
exec.o
input_parsing.c
input_parsing.h
input_parsing.o
main.c
main.o
server
shell
shell.zip
test
gssh$
```

12. rmdir

```
gssh$ ls
Makefile
Operating System - Phase 1.pdf
README.md
client
client.c
client.o
exec.c
exec.h
exec.o
input_parsing.c
input_parsing.h
input_parsing.o
main.c
main.o
server
shell
shell.zip
test
gssh$ rmdir test
gssh$ ls
Makefile
Operating System - Phase 1.pdf
README.md
client
client.c
client.o
exec.c
exec.h
exec.o
input_parsing.c
input_parsing.h
input_parsing.o
main.c
main.o
server
shell
shell.zip
gssh$
```

13. wc

```
gssh$ wc Makefile
  18   42  337 Makefile
gssh$
```

14. date

```
gssh$ date
Wed Mar 16 11:47:44 +04 2022
gssh$
```


16

14. pwd

```
gssh$ pwd
/Users/sarah.altowaity/Documents/OperatingSys
tems/OS_shell
gssh$
```

15. ps

```
gssh$ ps
  PID TTY          TIME CMD
 34623 ttys001      0:00.11 /bin/zsh -l
 41011 ttys001      0:00.03 ./server
 40961 ttys003      0:00.02 /bin/zsh -l
 41019 ttys003      0:00.02 ./client
gssh$
```

16. man

```
gssh$ man ps
PS(1)                                General Commands
Manual                               PS(1)

NAME
  ps - process status

SYNOPSIS
  ps [-AaCcEefhjlMmrSTvwXx] [-O fmt | -ot]
    [-G gid[,gid...]]
      [-g grp[,grp...]] [-u uid[,uid...]] [
-p pid[,pid...]] [-t tty[,tty...]]
    [-U user[,user...]]
  ps [-L]

DESCRIPTION
  The ps utility displays a header line, f
  ollowed by lines containing
  information about all of your processes
  that have controlling terminals.

  A different set of processes can be sele
  cted for display by using any
  combination of the -a, -G, -g, -p, -T,t,
  -U, and -u options.  If more
  than one of these options are given, the
  n ps will select all processes
  which are matched by at least one of the
  given options.
```

17

17. ping

```
gssh$ ping google.com
PING google.com (142.250.181.110): 56 data by
tes
64 bytes from 142.250.181.110: icmp_seq=0 ttl
=115 time=74.225 ms
gssh$ █
```

18. tee

```
./client
gssh$ echo hello | tee test.txt
hello
gssh$ cat test.txt
hello
gssh$ █
```

Pipe Instructions

One Pipe:

1. ls | grep "exec"

```
gssh$ ls | grep "exec"
exec.c
exec.h
exec.o
gssh$ █
```

2. cat Makefile | grep o

```
gssh$ cat Makefile | grep o
client: client.o input_parsing.o
      gcc -o client client.o input_parsing.o
server: main.o input_parsing.o exec.o
      gcc -o server main.o input_parsing.o exec.o
main.o: main.c
      rm *.o shell
gssh$ █
```

18

Two Pipes:

1. `cat Makefile | grep main | wc`

```
gssh$ cat Makefile | grep main | wc
      4      15     113
gssh$ █
```

2. `ls | grep main | tee main.txt`

```
gssh$ ls | grep main | tee main.txt
main.c
main.o
gssh$ cat main.txt
main.c
main.o
gssh$ █
```

Three Pipes:

1. `cat Makefile | grep main | grep gcc | wc`

```
gssh$ cat Makefile | grep main | grep gcc | wc
      2      9     60
gssh$ █
```

2. `ls | grep inp | tee input.txt | wc`

```
gssh$ ls | grep inp | tee input.txt | wc
      4      4     58
gssh$ █
```

Error Handling

Invalid Commands

```
gssh$ toch
gssh: command not found: No such file or directory
gssh$ █
```

```
gssh$ a
gsh: command not found: No such file or directory
gssh$ a | grep a
gsh: command not found: No such file or directory
gssh$ █
```

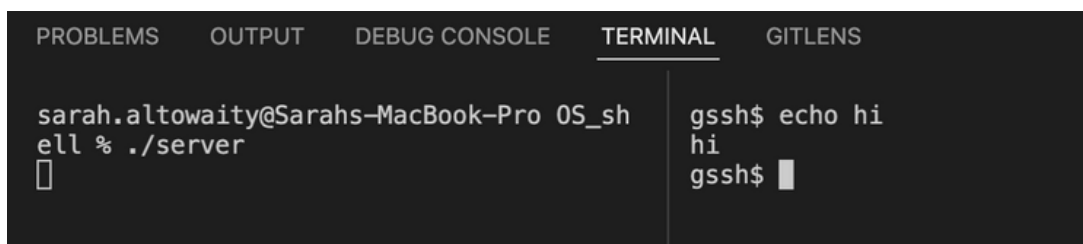
Shell Termination

```
gssh$ ls
Makefile
Operating System – Phase 1.pdf
README.md
client
client.o
exec.c
exec.h
exec.o
input_parsing.c
input_parsing.h
input_parsing.o
main.c
main.o
server
shell
shell.zip
test
gssh$ exit
sarah.altowaity@Sarahs-MacBook-Pro OS_shell % █
```

PHASE 2

THIS PHASE REQUIRES THE IMPLEMENTATION OF A REMOTE SHELL/COMMAND LINE INTERFACE USING SOCKET COMMUNICATION. ESSENTIALLY, OUR SHELL FROM PHASE 1 IS UPGRADED TO HAVE REMOTE CAPABILITIES TO SERVE COMMANDS REMOTELY.

In this phase, we upgrade the shell so that it becomes a server and each command is served as a client through socket communication. At any given time, only one command (client) can be served by our shell. We also have a limit of 10 commands in the queue to be served by our shell. In the next phase, we will be able to handle more than one command, connecting through a different server, remotely, by our shell through multithreading.



The screenshot shows a terminal window with a dark background and light gray text. At the top, there are tabs for 'PROBLEMS', 'OUTPUT', 'DEBUG CONSOLE', 'TERMINAL' (which is selected), and 'GITLENS'. The terminal content is split into two panes. The left pane shows the prompt 'sarah.altowaity@Sarahs-MacBook-Pro OS_sh' followed by the command 'ell % ./server' and a cursor. The right pane shows a remote shell prompt 'gssh\$' followed by the command 'echo hi', the output 'hi', and another 'gssh\$' prompt with a cursor.

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  GITLENS

sarah.altowaity@Sarahs-MacBook-Pro OS_sh
ell % ./server
█

gssh$ echo hi
hi
gssh$ █
```

21

IMPLEMENTATION

Overview

In order for us to have our shell offer remote services to the user, our chosen strategy was to augment our `main.c` file with code that creates a server to handle clients. We have also added a `client.c` file that will be responsible for sending and receiving commands from the user. This file creates a client for every command to be serviced by the user.

Server Implementation

We implement our server by creating two sockets, `sock1` (the main server) and `sock2` (the accepted client), in the `gss_loop()` function in the `main.c` file. We then proceed to add code that sets the communication protocols, binds the socket to the address, and listens for connections. We amend our while loop so that it no longer depends on the value of `res`, but rather runs indefinitely – as a server should.

Within the while loop, we accept any connecting clients in the variable `sock2`. We, then, receive the user input from the client and tokenize it based on pipes using the `tokenizePipes()` function. The `exec` functions called within the while loop are responsible for closing `sock2`. Once the while is broken out of, because of the `exit` command, we close `sock1`. However, this is never reached as our code only allows the `client.c` code to terminate. The server is

kept alive as it awaits further connections (other rounds of the execution of the file `client.c`).

Client Implementation

To implement the client, we had to add another file to our shell repository. The file, called `client.c`, creates a socket, called `sock`, used to connect to the server remotely. It displays the shell prompt, "gssh", reads user's input, sends the input to the server for execution, receives the response from the user, and displays it to the client.

The `client.c` code structure involves a loop controlled by a variable called `exit`, initially set to 0 (false). The loop runs as long as the condition "`!exit`" is true. If the user enters the command "exit", the variable is set to 1, and the loop stops in the next iteration, terminating the client program.

The redirection from the server output stream to the socket for display in the client is discussed next.

Output Re-Direction

To send the response over for reading to the client socket, standard output had to be re-directed to the client socket. This is done in the `exec.c` functions, which have been modified to receive an additional argument – a pointer to an integer, the socket file descriptor. To stop the server from displaying the results of `execvp()`, we re-direct standard output to the file descriptor associated with the client socket. Additionally, we close the client in every forked process the functions create.

23

Error Handling Implementation

In this phase, we added more specific error detection mechanisms to catch invalid pipe commands where one command is empty, e.g. " | ls | " or "| | | " to match Linux's error "parse error near `|': Invalid argument". Check the coming section on test cases for illustrations.

Input Parsing Modifications

We also added a function, `remove_special_chars()` to `input_parsing.c`, which is a function that removes the special characters `\"` and `\'`. `\n` and `\t` are already removed by the `tokenize()` function in `input_parsing.c`. This allows commands such as `echo "hi"` and `grep 'e'` to execute as they would in Linux, while they caused errors previously. Check the coming section on test cases for illustrations.

Implementation of cd

In this phase, we enhanced the services of our shell by giving it the ability to change the working directory. This is done through a call to `cd_command()` in `exec.c`. This function takes in a path, locating a directory. It opens the directory using `fopen()` to check the validity of the path. If `fopen()` returns `NULL`, the path is invalid and an error message is displayed to the client after re-directing standard error to the server (shell) socket. In the case of a valid path, `chdir()` is called with the path as the argument. The directory and the socket are subsequently closed.

Shell Termination

Just like in phase 1, the shell is terminated by entering the "exit" command. However, in phase 2, the "exit" command kills one session of the shell, the client, but does not kill the server, which is kept alive, listening to any other request to respawn the shell and handle commands. This is done by changing the value of the variable `exit` that controls the while loop in the client implementation from 0 to 1, making the condition `!exit` false, terminating the loop. See next section for illustrations.

25

TEST CASES

Note: All the examples shown below have also outputted the same result when the shell was executed in the remote server.

15 Single Commands

1.mkdir

2.ls (shows output of previous command)

```
sarah.altowaity@Sarahs-MacBook-Pro OS_shell % ./server

gssh$ mkdir test
gssh$ ls
Makefile
Operating System - Phase 1.pdf
README.md
client
client.c
client.o
exec.c
exec.h
exec.o
input_parsing.c
input_parsing.h
input_parsing.o
main.c
main.o
server
shell
shell.zip
test
gssh$
```

3. cat

```
sarah.altowaity@Sarahs-MacBook-Pro OS_shell % ./server

gssh$ cat input_parsing.h
#ifndef INPUT_PARSING
#define INPUT_PARSING

char* read_input();
char** tokenize(char*);
char** tokenizePipes(char*, int*);
void remove_special_chars(char*);

#endif
gssh$
```

26

TEST CASES

4. sort

```
sarah.altowaity@Sarahs-MacBook-Pro OS_shell % ./server
[]

gssh$ ls -l | sort
-rw-r--r-- 1 sarah.altowaity staff 165 Apr  5 04:19
input_parsing.h
-rw-r--r-- 1 sarah.altowaity staff 337 Apr  5 04:19
Makefile
-rw-r--r-- 1 sarah.altowaity staff 340 Apr  5 04:19
exec.h
-rw-r--r-- 1 sarah.altowaity staff 1663 Mar 29 14:08
README.md
-rw-r--r-- 1 sarah.altowaity staff 2040 Apr  5 04:19
client.o
-rw-r--r-- 1 sarah.altowaity staff 2152 Apr  5 04:19
main.o
-rw-r--r-- 1 sarah.altowaity staff 2288 Apr  5 04:19
input_parsing.o
-rw-r--r-- 1 sarah.altowaity staff 2408 Apr  5 04:19
client.c
-rw-r--r-- 1 sarah.altowaity staff 2745 Apr  5 04:19
main.c
-rw-r--r-- 1 sarah.altowaity staff 3634 Apr  5 04:19
input_parsing.c
-rw-r--r-- 1 sarah.altowaity staff 5816 Apr  5 04:19
exec.o
-rw-r--r-- 1 sarah.altowaity staff 12597 Apr  5 04:19
exec.c
-rw-r--r-- 1 sarah.altowaity staff 259966 Mar 29 14:08
shell.zip
-rw-r--r--@ 1 sarah.altowaity staff 273104 Mar 29 14:08
Operating System - Phase 1.pdf
-rwxr-xr-x 1 sarah.altowaity staff 50528 Apr  5 04:17
shell
-rwxr-xr-x 1 sarah.altowaity staff 50552 Apr  5 04:19
client
-rwxr-xr-x 1 sarah.altowaity staff 50848 Apr  5 04:19
server
drwxr-xr-x@ 2 sarah.altowaity staff 64 Apr  5 10:30
test
total 1488
gssh$
```

5. grep

```
sarah.altowaity@Sarahs-MacBook-Pro OS_shell % ./server
[]

gssh$ ls | grep h
Operating System - Phase 1.pdf
exec.h
input_parsing.h
shell
shell.zip
gssh$
```

```
sarah.altowaity@Sarahs-MacBook-Pro OS_shell % ./server
[]

gssh$ ls | grep 'h'
Operating System - Phase 1.pdf
exec.h
input_parsing.h
shell
shell.zip
gssh$
```

27

6. echo

<pre>sarah.altowaity@Sarahs-MacBook-Pro OS_shell % ./server █</pre>	<pre>gssh\$ echo hi my name is gssh hi my name is gssh gssh\$ █</pre>
<pre>sarah.altowaity@Sarahs-MacBook-Pro OS_shell % ./server █</pre>	<pre>gssh\$ echo 'hi my name is gssh' hi my name is gssh gssh\$ █</pre>

7. touch

<pre>sarah.altowaity@Sarahs-MacBook-Pro OS_shell % ./server █</pre>	<pre>gssh\$ touch test.txt gssh\$ ls Makefile Operating System - Phase 1.pdf README.md client client.c client.o exec.c exec.h exec.o input_parsing.c input_parsing.h input_parsing.o main.c main.o server shell shell.zip test test.txt gssh\$ █</pre>
---	--

8. mv

<pre>sarah.altowaity@Sarahs-MacBook-Pro OS_shell % ./server █</pre>	<pre>gssh\$ mv test.txt test/ gssh\$ ls Makefile Operating System - Phase 1.pdf README.md client client.c client.o exec.c exec.h exec.o input_parsing.c input_parsing.h input_parsing.o main.c main.o server shell shell.zip test gssh\$ ls test test.txt gssh\$ █</pre>
---	--

28

9. cp

```
./server
█ gssh$ ls
  Makefile
  Operating System - Phase 1.pdf
  README.md
  client
  client.c
  client.o
  exec.c
  exec.h
  exec.o
  input_parsing.c
  input_parsing.h
  input_parsing.o
  main.c
  main.o
  server
  shell
  shell.zip
  test
  test.txt
gssh$ cp test.txt test/
gssh$ ls test
test.txt
gssh$ █
```

```
./server
█ gssh$ ls
  Makefile
  Operating System - Phase 1.pdf
  README.md
  client
  client.c
  client.o
  exec.c
  exec.h
  exec.o
  input_parsing.c
  input_parsing.h
  input_parsing.o
  main.c
  main.o
  server
  shell
  shell.zip
  test
  test.txt
gssh$ █
```

29

10. rm

list of files and directories before removing test.txt

```
./server
█

gssh$ ls
Makefile
Operating System - Phase 1.pdf
README.md
client
client.c
client.o
exec.c
exec.h
exec.o
input_parsing.c
input_parsing.h
input_parsing.o
main.c
main.o
server
shell
shell.zip
test
test.txt
gssh$ rm test.txt
gssh$ ls
Makefile
Operating System - Phase 1.pdf
README.md
client
client.c
client.o
exec.c
exec.h
exec.o
input_parsing.c
input_parsing.h
input_parsing.o
main.c
main.o
server
shell
shell.zip
test
gssh$ █
```

11. wc

```
./server
gssh$ wc Makefile
      18      42     337 Makefile
gssh$
```

12. date

```
./server
gssh$ date
Tue Apr  5 11:00:14 +04 2022
gssh$
```

13. rmdir

```
./server
gssh$ ls
Makefile
Operating System - Phase 1.pdf
README.md
client
client.c
client.o
exec.c
exec.h
exec.o
input_parsing.c
input_parsing.h
input_parsing.o
main.c
main.o
server
shell
shell.zip
test
gssh$ rmdir test
gssh$ ls
Makefile
Operating System - Phase 1.pdf
README.md
client
client.c
client.o
exec.c
exec.h
exec.o
input_parsing.c
input_parsing.h
input_parsing.o
main.c
main.o
server
shell
shell.zip
gssh$
```

31

14. pwd

<pre>./server █</pre>	<pre>gssh\$ pwd /Users/sarah.altowaity/Documents/OperatingSystems/OS_shell gssh\$ █</pre>
-----------------------	---

15. ps

<pre>./server █</pre>	<pre>gssh\$ ps PID TTY TIME CMD 34623 ttys001 0:00.11 /bin/zsh -l 41011 ttys001 0:00.03 ./server 40961 ttys003 0:00.02 /bin/zsh -l 41019 ttys003 0:00.02 ./client gssh\$ █</pre>
-----------------------	---

16. man

<pre>./server █</pre>	<pre>gssh\$ man grep GREP(1) General Commands Manual GREP(1) NAME grep, egrep, fgrep, rgrep, bzgrep, bzegrep, bzip, zgrep, zgrep, zfgrep - file pattern searcher SYNOPSIS grep [-abcdDEFGHhIiJLlMmnOopqRSsUVvwXxZz] [-A m] [-B num] [-C [num]] [-e pattern] [-f file] [--binary-files=vae] [--color=[wh en]] [--colour=[when]] [--context=[num]] [--lal] [--line-buff ered] [--null] [pattern] [file ...] DESCRIPTION The grep utility searches any givgssh\$ █</pre>
-----------------------	---

32

17. ping

<pre>./server █</pre>	<pre>gssh\$ ping google.com PING google.com (142.250.181.110): 56 data by tes 64 bytes from 142.250.181.110: icmp_seq=0 ttl =115 time=74.225 ms gssh\$ █</pre>
-----------------------	--

18. tee

<pre>shell % ./server █</pre>	<pre>./client gssh\$ echo hello tee test.txt hello gssh\$ cat test.txt hello gssh\$ █</pre>
-------------------------------	---

Pipe Instructions

One Pipe:

1. `ls | grep "exec"`

<pre>./server █</pre>	<pre>gssh\$ ls grep "exec" exec.c exec.h exec.o gssh\$ █</pre>
-----------------------	--

2. `cat Makefile | grep o`

<pre>./server █</pre>	<pre>gssh\$ cat Makefile grep o client: client.o input_parsing.o gcc -o client client.o input_parsing.o server: main.o input_parsing.o exec.o gcc -o server main.o input_parsing.o exec.o main.o: main.c rm *.o shell gssh\$ █</pre>
-----------------------	--

Two Pipes:

1. cat Makefile | grep main | wc

<pre>./server █</pre>	<pre>gssh\$ cat Makefile grep main wc 4 15 113 gssh\$ █</pre>
-----------------------	--

2. ls | grep main | tee main.txt

<pre>./server █</pre>	<pre>gssh\$ ls grep main tee main.txt main.c main.o gssh\$ cat main.txt main.c main.o gssh\$ █</pre>
-----------------------	--

Three Pipes:

1. cat Makefile | grep main | grep gcc | wc

<pre>./server █</pre>	<pre>gssh\$ cat Makefile grep main grep gcc wc 2 9 60 gssh\$ █</pre>
-----------------------	--

2. ls | grep inp | tee input.txt | wc

<pre>./server █</pre>	<pre>gssh\$ ls grep inp tee input.txt wc 4 4 58 gssh\$ █</pre>
-----------------------	--

34

Error Handling

Invalid Commands

<pre>% ./server █</pre>	<pre>gssh\$ toch gssh: command not found: No such file or directory gssh\$ █</pre>
-------------------------	--

<pre>./server █</pre>	<pre>gssh\$ a b c gssh: command not found: No such file or directory gssh\$ █</pre>
-----------------------	---

Pipe Syntax Errors

<pre>server █</pre>	<pre>gssh\$ ls gssh: parse error near ` ': Invalid argument gssh\$ █</pre>
---------------------	--

<pre>./server █</pre>	<pre>gssh\$ ls gssh: parse error near ` ': Invalid argument gssh\$ █</pre>
-----------------------	--

Shell Termination

<pre>sarah.altowaity@Sarahs-MacBook-Pro OS_shell % . ./server █</pre>	<pre>gssh\$ wc exec.c 438 1306 13160 exec.c gssh\$ exit sarah.altowaity@Sarahs-MacBook-Pro OS_shell % █</pre>
---	---

Change Directory

<pre>./server █</pre>	<pre>/Users/sarah.altowaity/Documents/OperatingSystems/OS_shell gssh\$ cd test gssh\$ pwd /Users/sarah.altowaity/Documents/OperatingSystems/OS_shell/test gssh\$ cd ../../OS_shell gssh\$ pwd /Users/sarah.altowaity/Documents/OperatingSystems/OS_shell</pre>
-----------------------	--

<pre>./server █</pre>	<pre>gssh\$ cd akjsaajc cd: : No such file or directory gssh\$ █</pre>
-----------------------	--

PHASE 3

IN THIS PHASE, WE UPGRADE OUR SHELL WITH MULTITHREADING SO THAT IT CAN SERVE MORE THAN ONE CONNECTED SIMULTANEOUSLY

In this phase, we upgrade the shell server so that it can serve more than one connected client in a different thread simultaneously. We also provide the shell server and the client with the ability to exit with Ctrl-C.

```
sarah.altowaity@Sarahs-MacBook-Pro OS_shell % ./server
gssh$ ls
Makefile
README.md
client
client.c
client.o
exec.c
exec.h
exec.o
input.txt
input_parsing.c
input_parsing.h
input_parsing.o
main.c
main.o
server
shell
shell.pdf
test
gssh$

gssh$ pwd
/Users/sarah.altowaity/Documents/OperatingSystem
gssh$

gssh$ ls -l
total 6864
-rw-r--r--  1 sarah.altowaity  staff   337
Apr  6 02:46 Makefile
-rw-r--r--  1 sarah.altowaity  staff  1663
Mar 29 14:08 README.md
-rwxr-xr-x  1 sarah.altowaity  staff 50960
Apr 26 12:58 client
-rw-r--r--  1 sarah.altowaity  staff   4118
Apr 26 10:28 client.c
-rw-r--r--  1 sarah.altowaity  staff   3648
Apr 26 12:58 client.o
-rw-r--r--  1 sarah.altowaity  staff  14181
Apr 24 22:11 exec.c
-rw-r--r--  1 sarah.altowaity  staff    424
gssh$ ls -l
total 6864
-rw-r--r--  1 sarah.altowaity  staff   337
Apr  6 02:46 Makefile
-rw-r--r--  1 sarah.altowaity  staff  1663
Mar 29 14:08 README.md
-rwxr-xr-x  1 sarah.altowaity  staff 50960
Apr 26 12:58 client
```

gssh accommodating 3 clients at the same time

36

IMPLEMENTATION

Overview

In order for us to have multithreading capabilities, we had to create a client handler function that will be responsible for handling the threads.

Thread Creation

In the `gssh_loop()` function, we start by setting declaring our thread and the struct holding its attributes. We set the threads to be detached as each client is independent and does not to be synchronized or joined with any other thread. We then create a thread for every accepted connection.

Client (Thread) Handling Function

The function returning a void pointer passed to the thread creation function is called `clientHandler` and it takes in a void pointer to the socket handling the connection. In essence, this routine engulfs the functional aspect of the `gssh` loop that handled the commands (previously only done in one main thread) to be utilized by every thread in the handling of clients. The function ends with `pthread_exit(NULL)` to terminate the thread if the connection to the client is terminated.

Dynamic Receiving of Input

In order to have our server receive the full input sent by clients more dynamically, regardless of size, we implemented the function that Shan provided. The `recv_timeout` function works by using a non-blocking pipe and a timeout value. The function runs in a loop.

1. It gets the time difference between the current time and when the function first begins receiving.
2. If we receive any data, we print it and reset the beginning time for receiving.
3. If there is data received but the time difference is greater than the timeout, we stop receiving.
4. If there is no data received, we wait for twice the timeout before we exit the functions.

Exiting Client and Server with Ctrl-C

We included an exiting routine to handle exiting from both the client and the server. When the user enters Ctrl-C on the client program, the client sends an "exit" string to the server, so that the server can close the socket dedicated to this connection, display an exiting message, and closes the socket in the client program. The server also closes with Ctrl-C, prints an exiting messages, and closes the socket. Both of the closing routines are in client handler functions, `clientExitHandler` and `serverExitHandler`.

38

Changing Directories with `cd` in Our Multithreading Environment

Using `cd`, in our implementation, with various threads, will change the directory of the server program as well as all the threads that are currently being serviced by the thread (see test cases for illustrations). Any impending communications will also be in the working directory of all the other threads, including the main server thread, at the time of the connection.

Issues with Commands with Inherent Blocking

In our implementation, we have changed the socket to be nonblocking to enable the receiving of large input without having to specify the size. Because of that, certain commands executed with `execvp` that wait for arguments to completely execute, in which case `execvp` executes successfully to run the program and does not run `-1`, cause our server socket to lose the ability to service subsequent commands. We have fixed it for commands like `cat`, where we handled these cases by avoiding the execution of `execvp`, but a general solution is a work in progress.

39

TEST CASES

15 Single Commands

1. mkdir

2. ls

<pre>> ./server █</pre>	<pre>> ./client gssh\$ mkdir test gssh\$ █</pre>	<pre>> ./client gssh\$ ls Makefile README.md client client.c client.o exec.c exec.h exec.o input.txt input_parsing.c input_parsing.h input_parsing.o main.c main.o server shell.pdf test gssh\$ █</pre>
----------------------------	---	--

3. cat

<pre>> ./server █</pre>	<pre>> ./client gssh\$ mkdir test gssh\$ cat input_parsing.h #ifndef INPUT_PARSING #define INPUT_PARSING char* read_input(); char** tokenize(char*); char** tokenizePipes(char*, int*); void remove_special_chars(char*); #endifgssh\$ █</pre>	<pre>> ./client gssh\$ ls Makefile README.md client client.c client.o exec.c exec.h exec.o input.txt input_parsing.c input_parsing.h input_parsing.o</pre>
----------------------------	---	---

40

TEST CASES

4. sort

5. grep

```

> ./server
gssh$ ls -l | sort
-rw-r--r-- 1 gautham staff 48 Apr 24 21:56 input.txt
-rw-r--r-- 1 gautham staff 165 Apr 5 23:49 input_parsing.h
-rw-r--r-- 1 gautham staff 337 Apr 23 16:07 Makefile
-rw-r--r-- 1 gautham staff 424 Apr 5 23:59 exec.h
-rw-r--r-- 1 gautham staff 1663 Mar 16 13:46 README.md
-rw-r--r-- 1 gautham staff 2288 Apr 5 23:57 input_parsing.o
-rw-r--r-- 1 gautham staff 3560 Apr 27 22:27 main.o
-rw-r--r-- 1 gautham staff 3634 Apr 5 23:49 input_parsing.c
-rw-r--r-- 1 gautham staff 3648 Apr 27 22:27 client.o
-rw-r--r-- 1 gautham staff 4118 Apr 25 22:49 client.c
-rw-r--r-- 1 gautham staff 5463 Apr 27 22:31 main.c
-rw-r--r-- 1 gautham staff 6144 Apr 24 21:23 exec.o
-rw-r--r-- 1 gautham staff 14181 Apr 24 21:17 exec.c
-rw-r--r-- 1 gautham staff 3274761 Apr 6 21:34 shell.pdf
-rwxr-xr-x 1 gautham staff 50960 Apr 27 22:27 client
-rwxr-xr-x 1 gautham staff 51704 Apr 27 22:27 server
drwxr-xr-x 2 gautham staff 64 Apr 27 22:37 test
total 6760
gssh$

gssh$ ls | grep h
exec.h
input_parsing.h
shell.pdf
gssh$

```

6. echo

7. wc

```

> ./server
gssh$ echo hi my name is gssh
hi my name is gssh
gssh$

gssh$ wc Makefile
18 42 337 Makefile
gssh$

```

8. touch

list of files and directories after creating test.txt is shown in the second client

```

> ./server
gssh$ touch test.txt
gssh$

gssh$ ls
Makefile
README.md
client
client.c
client.o
exec.c
exec.h
exec.o
input.txt
input_parsing.c
input_parsing.h
input_parsing.o
main.c
main.o
server
shell.pdf
test
test.txt
gssh$

```

41

TEST CASES

9. cp

<pre>> ./server []</pre>	<pre>gssh\$ cp test/test.txt ./ gssh\$ ls Makefile README.md client client.c client.o exec.c exec.h exec.o input.txt input_parsing.c input_parsing.h input_parsing.o main.c main.o server shell.pdf test test.txt gssh\$ []</pre>	<pre>gssh\$ ls test test.txt gssh\$ []</pre>
-----------------------------	---	--

10. mv

moving of test.txt into the directory test is displayed on the second client

<pre>> ./server []</pre>	<pre>gssh\$ mv test.txt test/ gssh\$ []</pre>	<pre>gssh\$ ls test test.txt gssh\$ []</pre>
-----------------------------	---	--

11. rmdir

<pre>> ./server []</pre>	<pre>gssh\$ rmdir test gssh\$ []</pre>	<pre>gssh\$ ls Makefile README.md client client.c client.o exec.c exec.h exec.o input.txt input_parsing.c input_parsing.h input_parsing.o main.c main.o server shell.pdf test.txt gssh\$ []</pre>
-----------------------------	--	---

42

TEST CASES

12. rm

<pre>> ./server █</pre>	<pre>gssh\$ rm test/test.txt gssh\$ █</pre>	<pre>gssh\$ ls test gssh\$ █</pre>
----------------------------	---	------------------------------------

13. date

<pre>> ./server █</pre>	<pre>gssh\$ date Wed Apr 27 22:44:19 +04 2022 gssh\$ █</pre>	<pre>gssh\$ ls Makefile README.md client client.c client.o exec.c</pre>
----------------------------	--	---

14. pwd

<pre>> ./server █</pre>	<pre>gssh\$ pwd /Users/gautham/Documents/OS/OS_shell gssh\$ █</pre>
----------------------------	---

15. ps

<pre>> ./server █</pre>	<pre>gssh\$ pwd /Users/gautham/Documents/OS/OS_shell gssh\$ █</pre>	<pre>38862 ttys012 0:23.53 /bin/zsh 44811 ttys012 0:00.00 /bin/zsh 44813 ttys012 0:02.03 /bin/zsh 44820 ttys012 0:00.01 /bin/zsh 44843 ttys012 0:07.72 /Users/gautham/Documents/OS/OS_shell 63405 ttys013 0:01.68 /bin/zsh -l 63410 ttys013 0:00.01 /bin/zsh -l 63962 ttys013 0:00.00 /bin/zsh -l 63963 ttys013 0:00.02 /bin/zsh -l 63965 ttys013 0:00.09 /Users/gautham/Documents/OS/OS_shell 67647 ttys013 0:02.39 ./server 65958 ttys014 0:00.88 /bin/zsh -l 65963 ttys014 0:00.01 /bin/zsh -l 66506 ttys014 0:00.00 /bin/zsh -l 66507 ttys014 0:00.01 /bin/zsh -l 66509 ttys014 0:00.06 /Users/gautham/Documents/OS/OS_shell 67800 ttys014 0:00.01 ./client 67807 ttys015 0:00.49 /bin/zsh -l 67811 ttys015 0:00.01 /bin/zsh -l 68368 ttys015 0:00.00 /bin/zsh -l 68369 ttys015 0:00.01 /bin/zsh -l 68371 ttys015 0:00.03 /Users/gautham/Documents/OS/OS_shell 68388 ttys015 0:00.01 ./client gssh\$ █</pre>
----------------------------	---	--

Connect Live Share -- NORMAL --

43

TEST CASES

16. tee

```
> ./server
gssh$ cat test.txt
hello
gssh$
gssh$ echo hello | tee test.txt
hello
gssh$
```

17. man

```
> ./server
magssh$ ^R
man ps
PS(1)                                General Commans
ds Manual                            PS(1)

NAME
    ps - process status

SYNOPSIS
    ps [-AaCcEefhjLMmrSTvwXx] [-O fmt | o
fmt] [-G gid[,gid...]]
        [-g grp[,grp...]] [-u uid[,uid...]]
        [-p pid[,pid...]] [-t tty[,tty...]]
        [-U user[,user...]]
    ps [-L]
```

44

TEST CASES

Pipe Instructions

One Pipe:

```
➤ ./server
gssh$ ls | grep "exec"
exec.c
exec.h
exec.o
gssh$

gssh$ cat Makefile | grep o
client: client.o input_parsing.o
gcc -o client client.o input_parsing.o
server: main.o input_parsing.o exec.o
gcc -o server main.o input_parsing.o exec.o
main.o: main.c
rm *.o shell
gssh$
```

Two Pipes:

```
➤ ./server
gssh$ cat Makefile | grep main | wc
      4      15     113
gssh$

gssh$ ls | grep main | tee main.txt
main.c
main.o
gssh$
```

Three Pipes:

```
➤ ./server
gssh$ cat Makefile | grep main | grep gcc | wc
      2      9     60
gssh$

gssh$ ls | grep inp | tee input.txt | wc
      4      4     58
gssh$
```

Error Handling

Invalid Commands

```
➤ ./server
gssh$ toch
gssh: command not found: No such file or directory
gssh$

gssh$ a | grep a
gssh: command not found: No such file or directory
gssh$
```

Pipe Syntax Errors

```
➤ ./server
gssh$ ls |
gssh$ parse error near `|': Invalid argument
gssh$
```

45

TEST CASES

Client and Server Exiting with Ctrl -C

<pre>./server ^C Exiting server. sarah.altowaity@Sarahs-MacBook-Pro OS_shell %</pre>	<pre>gssh\$ ^C Exiting client. sarah.altowaity@Sarahs-MacBook-Pro OS_shell %</pre>	<pre>gssh\$ ^C Exiting client. sarah.altowaity@Sarahs-MacBook-Pro OS_shell %</pre>
--	--	--

Change Directory

<pre>./server </pre>	<pre>gssh\$ pwd /Users/sarah.altowaity/Documents/OperatingSystems/OS_shell gssh\$ cd test gssh\$ pwd /Users/sarah.altowaity/Documents/OperatingSystems/OS_shell/test gssh\$</pre>	<pre>gssh\$ pwd /Users/sarah.altowaity/Documents/OperatingSystems/OS_shell/test gssh\$</pre>
----------------------	---	--

PHASE 4

THIS PHASE REQUIRES THE IMPLEMENTATION OF A SCHEDULING ALGORITHM THAT SCHEDULES THE RUNNING OF COMMANDS BASED ON A PARTICULAR COMBINATION OF SCHEDULING ALGORITHM.

In this phase, we upgrade the shell with scheduling capabilities that demo the scheduling algorithms in real multiprogramming systems. The particular scheduling algorithm we implement is a combination of the Shortest Job Remaining First (SJRF) and Round Robin (RR).

47

IMPLEMENTATION

Overview

In order for us to simulate the scheduling capabilities of our shell, we create a demo program that executes with a command line argument indicating the time it would take to run. Our shell, at any given time, will maintain a job queue, implemented as a priority queue for the implementation of SRTF and three active threads: 1) a thread that creates a dedicated communication with connecting client, 2) a scheduler, and 3) the thread scheduled to run by the scheduler. We also define a Request struct object to hold all the necessary information about the requests by clients to be scheduled by the scheduler.

Algorithm Behavior and Scheduling Conditions

1. Applying preemptive Shortest Remaining Time First (SRTF) to select a process from the priority queue whenever a process enters the queue.
2. If a request arrives while another is in execution, the current request is preempted to allow the next thread with the shortest remaining time to run.
3. A thread is not allowed to run twice consecutively.
4. A thread can only run for a maximum of a quantum time (Round Robin) after which it will be stopped and the next *thread* with the shortest time remaining request will be scheduled. The quantum is set to 3 in the first round of execution of any request and 7 for the subsequent rounds.
5. A thread cannot make another request if it has a pending request to be completed in the queue.

Priority Queue Data Structure and STRF

All the threads executing in the shell share a priority queue data structure of an array of pointers to Request objects. Each Request object has the following attributes: 1) a string called command holding the command and any associated parameters and/or flags, 2) an int variable called time that holds the value of the burst (execution) time of the task – normal shell commands such as "ls" and "pwd" have a default time of 1, 3) a pthread_id variable called threadid that holds the thread ID of the thread waiting to execute this specific request, 4) an int called socket holding the file descriptor value of the socket associated with the client requesting to execute the command, 5) a pointer to sem_t called sem which will store a pointer to the semaphore associated with each client thread, and 6) an integer called round that records the current round the request is at (the number of times this request has been executed so far). This attribute will dictate the quantum time for the request to run in the next round of execution. Every time a client on a specific thread creates a request, given that the client has not previously made a request pending execution by the scheduler, a Request object is created with the necessary initializations of attributes and inserted into the queue. We define the following functions for the proper working and maintenance of the priority queue:

1. swap(): swap two Request pointer objects.
2. heapify(Request * array[], int size, int i): an $O(\log n)$ algorithm to heapify the array based on the time property. The heap this function builds is a min heap as it gives priority to requests with shorter time.
3. insert(Request * array[], Request * newRequest): inserts the new request at the end of the passed array and

49

re-heapifies the queue by looping over the elements from indices 0 to $\text{size}/2 - 1$ in a $O(n \log n)$ fashion.

4) `delete(Request * array[], pthread_t threadid)`: Finds the Request to be removed from the queue based on the threadid, since a thread can only have one request in the command at any given time, swaps the last element with the element to be deleted, decrements the size. and heapifies the queue (excluding the element stored at the index corresponding to the initial size).

Our priority queue data structure aids the scheduler in knowing the order of requests in order of increasing execution time.

Use of Semaphores

Our program makes use of three named semaphores shared by all threads (defined as global variables) and a local unnamed semaphore created with every connecting client thread. The use of each is as follows:

SHARED NAMED SEMAPHORES

1) `/global`: a named semaphore initialized to zero and is used to indicate if a demo program is currently in execution. This semaphore is used by the scheduler to stop the demo program currently in execution.

2) `/newArrival`: a named semaphore initialized to 0. This semaphore signals to the scheduler that a new request has entered the queue.

3) /processRe: this named semaphore is initialized to 0 and is used to prevent the scheduler from scheduling the next process until the task that was interrupted is returned back to the queue.

UNNAMED LOCAL SEMAPHORES

1) local: an unnamed semaphore created in every thread that is initialized to zero. The client waits on this semaphore before the running of its request. In order for the request to execute, the scheduler will have to call `sem_post` on this semaphore to allow it to execute by either calling `runExec` for normal commands or `execDemo` for the running of the demo program.

Use of Mutexes

We use one mutex lock, `queue_lock` for insertion and deletion from the queue, defined as a global variable to be shared by all the threads, to only allow one thread to modify the queue at any given time.

Demo.c and RR

To simulate the working of a scheduler, we create a separate .c file called `demo.c`. This program takes the two command line arguments, the remaining execution time and the current round of the request. Depending on the round, the `QUANTUM` variable is set: 3 in the first round, and 7 in the rest. The program opens the semaphores and behaves according to their values or whether the quantum is reached.

51

The demo program is basically a for loop that runs from N , the remaining execution time, down to $N - \text{QUANTUM}$ if $N \geq \text{QUANTUM}$ or 0 if $N < \text{QUANTUM}$. The program prints the current iteration number if the value of the named semaphore `/global`, the semaphore responsible for the execution of the program, is greater than 0, indicating that the program is allowed to execute, followed by a sleep for a second. In the case that the quantum is reached, we return the current iteration, which is used to create a new Request object to be inserted in the queue with the time attribute being equal to the value of the counter variable in the for loop at the time of interruption and the round attribute being set to the current value incremented by 1. We then post the `/newArrival` semaphore to indicate the arrival of the new request into the queue. In the case of another client requesting to run `demo.c` from another thread and the scheduler waiting on `/global` making it 0, the program returns the current iteration of the for loop. Effectively, any thread running `demo` will be interrupted upon the arrival of a new request from another client. In the case the for loop is executed in its entirety without returning, the request has been completed. At that point, if the value of `/global` is not 0, we wait on the semaphore to decrement its value to 0 to indicate that the demo is no longer running and there is no other process in the queue. If the scheduler has changed the value of `global` to 0 and there is no remaining execution time, nothing will be reinserted into the queue. Therefore, the `sem process_re` will not be signaled back to 1 and the scheduler will call wait on the semaphore and will block. Hence, we need to post `process_re` for the scheduler to be able to scheduler the next process. We finally close the semaphores and return `i` (which will be 0).

52

Scheduler

Our scheduler is the running thread that is responsible of retrieving commands from the priority queue and signaling the appropriate threads to run if scheduled. The thread is created upon the execution of the server and performs the scheduling if the queue is not empty. The scheduler keeps a variable that stores the socket file descriptor of the socket on which the currently running thread is establishing its communication. If there are requests in the queue, the scheduler begins by waiting on the `/newArrival` semaphore which allows the scheduler to stop scheduling in the case a new request is being inserted in the queue. Once the process is inserted, the associated thread will signal the `/newArrival` semaphore, incrementing its value back to 0, and the scheduler will be able to start the scheduling of the new request. In the case that another thread is executing `demo.c`, the `/global` semaphore will be set to 1, at which case, the scheduler will wait on it to make its value 0, stopping the currently executing thread from running. The scheduler then waits on the semaphore `/processRe` that decrements the value of the semaphore. This stops the scheduler from scheduling the next command until the interrupted process finishes returning its request into the queue (or deleting its request if it finished execution). The scheduler then schedules the next process to run under the condition that the task does not belong to the thread that requested the task that finished execution. To do that, the scheduler retrieves the next process from the queue and posts the semaphore associated with the thread that made the request. The scheduler also posts the `/global` semaphore if the request is to run `demo.c` to allow the execution of the program.

53

In the case that the value of `/global` is 0, no thread is currently in execution and we do not need to issue an interruption through semaphores. Hence, the scheduler simply retrieves the next request in priority from the priority queue and posts the associated semaphore of the thread that created the request. In the case the request is to run a demo program, the `/global` semaphore is posted too to change its value to 1 and to allow the execution of the demo program once executed by the scheduled process. Finally, the value of the current socket is updated to the socket attribute of the scheduled process.

Currently executing thread

The client handler thread is created every time a client establishes a connection with the server. The client handler thread creates a local unnamed semaphore and initializes to 0. This semaphore will be used to initialize the `sem` attribute for all the requests a particular thread is making. If the client connects for the first time, the client handler receives the command and creates a Request pointer object and initialize the attributes. We set the round to 1 to indicate that this specific request is running for the first time. We then go into another while loop that is based on the boolean variable, first set to 1, to indicate the assumption that this request will need more than one round to execute. Within the while loop, we first insert the Request pointer object into the queue, this is a critical section and hence is protected by mutex locks. If the command is repeating we post the `/processRe` semaphore to indicate that the Request pointer object that was inserted into the queue is the remainder of a previously executed request. This allows the scheduler to schedule the next request. If not, we post the `/newArrival` semaphore to indicate that this is a new request that is entering the queue for the first time. The client then waits on the unnamed semaphore to stop

54

Our client from executing the command. The client handler is allowed to move on with execution if the scheduler signals the semaphore associated with the client. Once signaled, the client executes the request. If the command is a normal command line task, we call `runExec()`, that is the execution function from previous phases. We then delete the request from the queue after the request is executed. Otherwise, we call `execDemo()`, a refurbishment of `runExec()` that returns the request command that holds the updated information of the request after the end of the program's execution (either because of the end of the round, the running of the full execution time of the request, or the arrival of a new process). In the case the remaining time of the updated command is 0, the request is removed from the queue, and the client moves on to accepting a new request from the user.

Client-Socket Communication

We have made our receiving from the socket in `client.c` non-blocking in phase 3 because of issues related to dupping the output onto the various thread sockets. In this phase, we have made the disabled blocking for normal command line requests with a set timeout and kept the receiving blocking for the execution of `demo.c`. The iterations are also printed on the server.

55

Assumptions

For flawless execution and demonstration of the scheduling capabilities our shell, we make the following assumptions:

1. No new request is made at the simultaneous instance of the end of the execution of the current request or at the end of the quantum of the currently executing request.
2. When executing /demo, we assume the user inputs a nonnegative integer parameter argument as part of the request

Execution

Some semaphore functions are not supported on MacOS. To execute the shell, we set up a Linux environment using a Docker container

56

TEST CASES DEMO

To help demonstrate the scheduling capabilities of the shell, view the following demo video:

Normal commands, ./demo, and improper commands

https://drive.google.com/file/d/1UwoJTPf2UyKlXIbIFVAuPwE1DfhZRKh_/view?usp=sharing

Piped commands

https://drive.google.com/file/d/1Ex--PQwAAIIL-Qm2olaXciWPjRGHJ6b_/view?usp=sharing