GEBZE TECHNICAL UNIVERSITY

Computer Engineering Department

CSE 344 – 2024 Spring

Midterm Project Report

Serdar Genç

210104004023

1) Connection:

The initial connection of a client and the server is provided by the Server FIFO which is contained inside the “server_directory”. This FIFO obeys the name pattern “server_%d.fifo” (%d = server pid).

“server_directory” is received from command line argument, but for program to work the SERVER_DIR definition in client.c must be same with server directory name.

Client program sends a ***client*** into the Server FIFO and the server reads it. After reading, server creates a Client FIFO with pattern “client_%d.fifo” (%d = client pid).

```
struct client {
   pid_t pid;
   int tryConnect;
};
```

1.1) tryConnect vs Connect:

tryConnect is set to 1 if connection type is “tryConnect”. If tryConnect is 1 and the server is currently full, the server sends information to the client that tells server is full via Client FIFO. If the server is full, then tryConnect client terminates without waiting que. Else it is sent to child_process function to become an active client.

If tryConnect is set to 0 this means connection type is “Connect”. If tryConnect is 0 and the server is currently full, the server enqueues the client into wait_que which is a instance of struct queue.

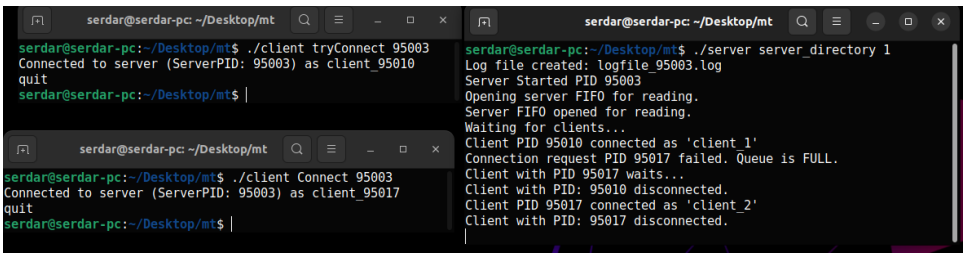
```
struct queue {
   int front;
   int rear;
   int capacity;
   int size;
   struct client **elements;
```

};

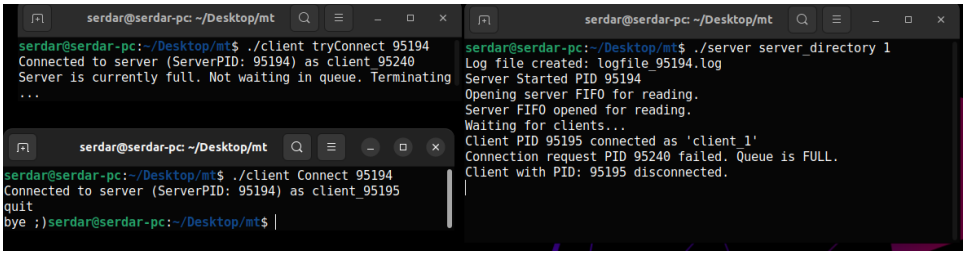
When a client is enqueued it waits for its turn. Whenever an active client is terminated, if there is a queue, the front of the queue is dequeued and it is sent to child_process function to become an active client.

This is how i check this condition:

while (1) { if (num_clients < max_clients && wait_que.size > 0){ ..



(left top: client_1[tryconnect], left bottom: client_2[connect], right: server terminal)



(left top: [tryconnect], left bottom: client_1[connect, right: server terminal)

As you can see, Connect client waits for its turn but tryConnect client does not.

2) Client Requests and Server Responses:

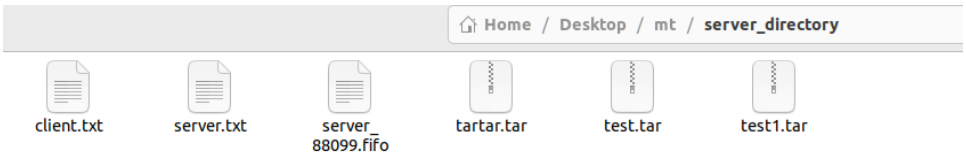
A client sends requests in a while loop which does not stop until kill signal, stop signal or quit command. In each iteration it checks if the server is still online by checking the Server FIFO status. If Server FIFO is still existing, then it waits for user to enter command and sends the command to the server via Client FIFO. Then client opens read end of Client FIFO and waits for the server to write a response. At the same time the server receives the command and extract arguments from it. Then it executes command, creates a response and writes it back into Client FIFO. Client FIFO receives the response and prints it.

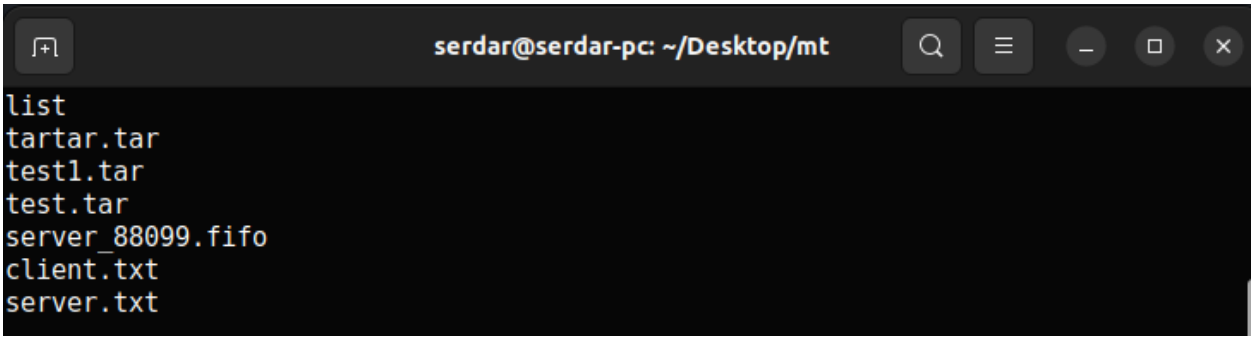
2.1) ‘help’ commands:

```
help
Available commands are:

help, list, readF, writeT, upload, download, archServer, killServer, quit
help readF


readF <file> <line #>
Display the #th line of the <file>, returns with an error if <file> does not exists.

help writeT


writeT <file> <line #> <string>
Write the  content of “string” to the  #th  line the <file>, if the line # is not given writes to the end of file. If the file does not exists in Servers directory creates and edits thefile at the same time.

help upload


upload <file>
Uploads the file from the current working directory of client to the Servers directory.

help download


download <file>
Receive <file> from Servers directory to client side

help archServer


archServer <fileName>.tar
Using fork, exec and tar utilities create a child process that will collect all the files currently available on the the  Server side and store them in the <filename>.tar archive

help killServer


killServer
Sends a kill request to the Server
```

help, help list, help readF, help writeT, help upload, help download, help archServer, help killServer commands can be used to receive information about functions and how functions work.

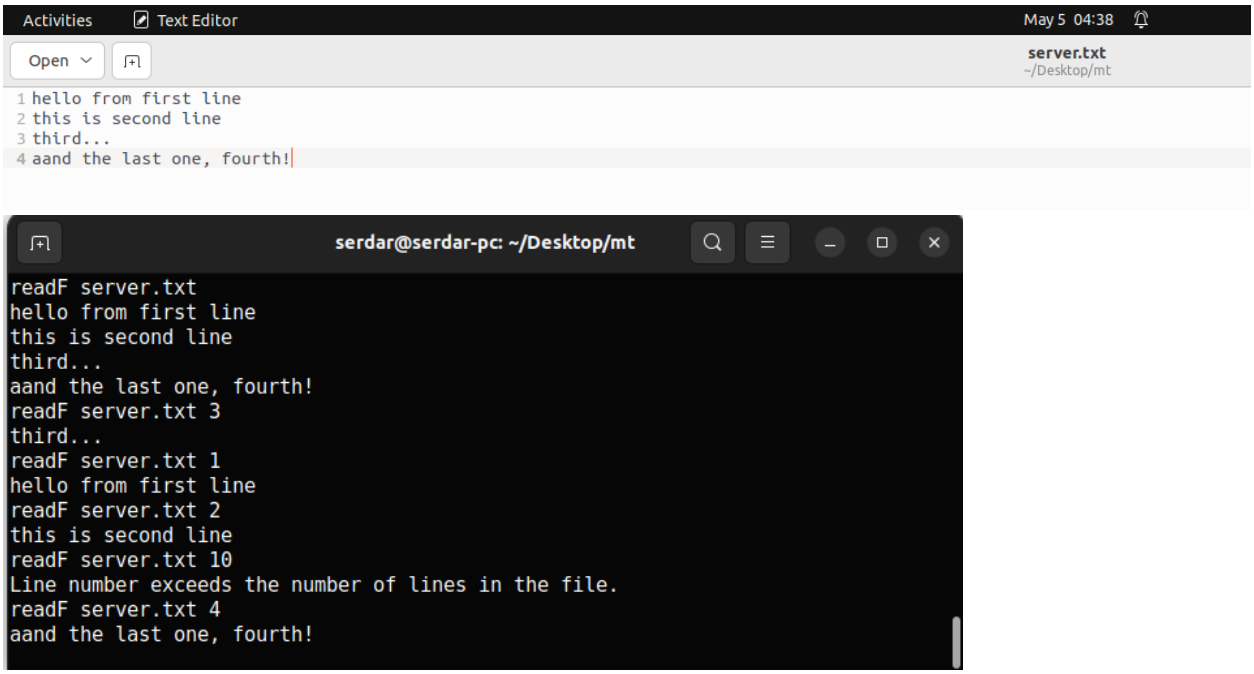
2.2) ‘list’ command:





list command opens “server_directory” and reads directory using dirent struct and readdir() function (<dirent.h> library). Then sends file names as response one by one.

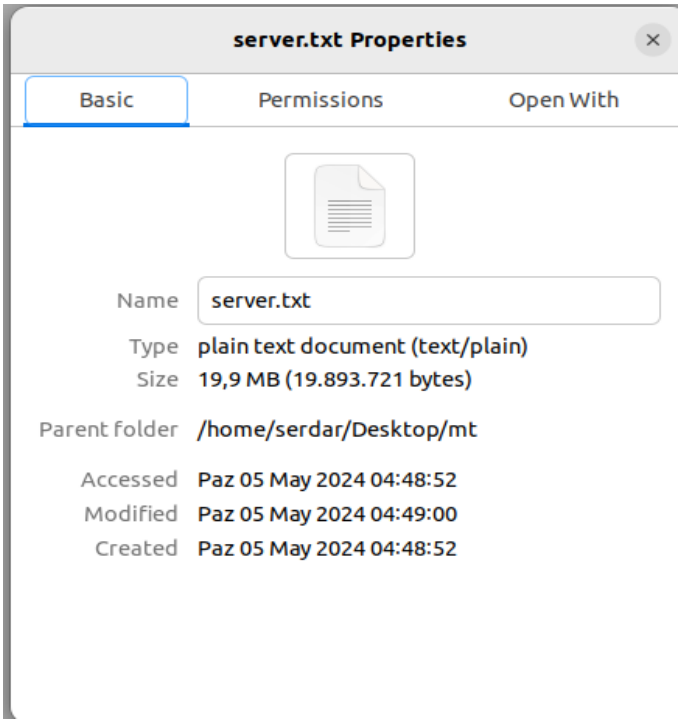
2.3) ‘readF’ command:

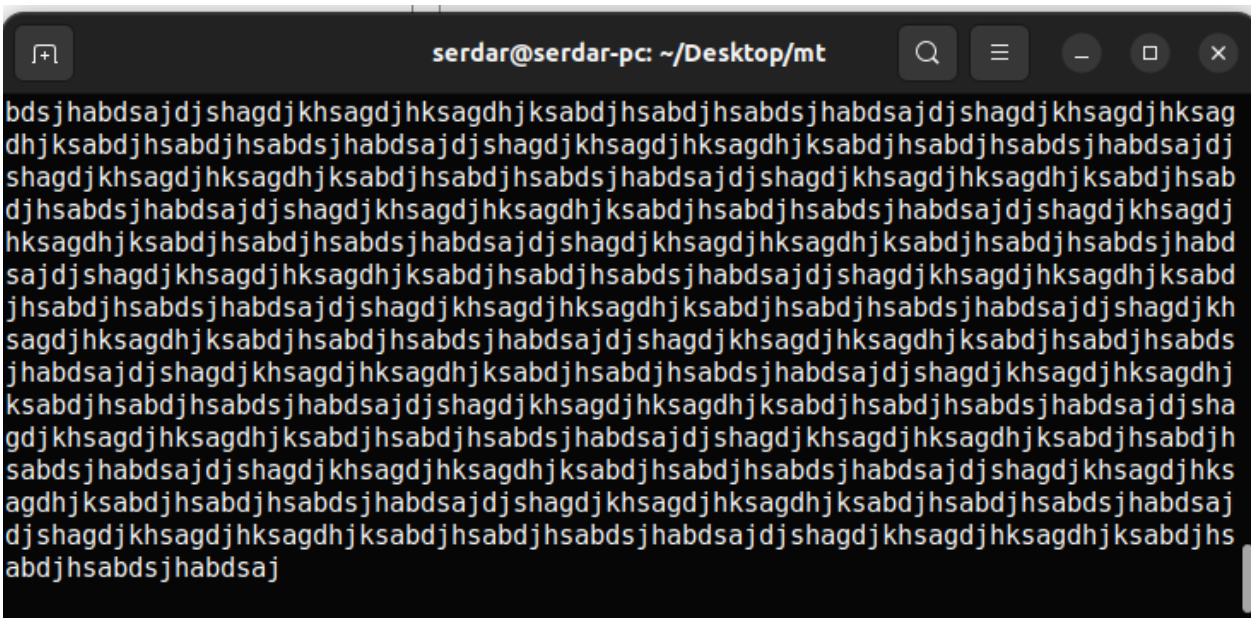


readF <filename> <line> command reads all lines when; line argument is smaller than 1, or line argument is null. If line argument is bigger than the number of lines in the file server sends response: “Line number exceeds the number of lines in the file.” Otherwise, it reads the line from <line> of file.
I show the test with 20 MB file in writeT part.

For this operation I read file char by char and increment current_line when ‘\n’ is encountered. If current_line is selected_line, I send that line as response to Client FIFO.

2.4) ‘writeT’ command:





(a part of output from readF for 20MB file)

```
djshagdjkhsagdjhksagdhjksabdjhsabdjhsabd
abdjhsabdsjhabdsaj
writeT server.txt lastline!
Succesfully written into the file.
```

```
serdar@serdar-pc: ~/Desktop/mt
dhjksabdjhsabdjhsabdsjhabdsajdjshagdjkhsagdjhksagdhjksabdjhsabdjhsabdsjhabdsajdj
shagdjkhsagdjhksagdhjksabdjhsabdjhsabdsjhabdsajdjshagdjkhsagdjhksagdhjksabdjhsab
djhsabdsjhabdsajdjshagdjkhsagdjhksagdhjksabdjhsabdjhsabdsjhabdsajdjshagdjkhsagdj
hksagdhjksabdjhsabdjhsabdsjhabdsajdjshagdjkhsagdjhksagdhjksabdjhsabdjhsabdsjhabd
sajdjshagdjkhsagdjhksagdhjksabdjhsabdjhsabdsjhabdsajdjshagdjkhsagdjhksagdhjksabd
jhsabdjhsabdsjhabdsajdjshagdjkhsagdjhksagdhjksabdjhsabdjhsabdsjhabdsajdjshagdjkh
sagdjhksagdhjksabdjhsabdjhsabdsjhabdsajdjshagdjkhsagdjhksagdhjksabdjhsabdjhsabds
jhabdsajdjshagdjkhsagdjhksagdhjksabdjhsabdjhsabdsjhabdsajdjshagdjkhsagdjhksagdhj
ksabdjhsabdjhsabdsjhabdsajdjshagdjkhsagdjhksagdhjksabdjhsabdjhsabdsjhabdsajdjsha
gdjkhsagdjhksagdhjksabdjhsabdjhsabdsjhabdsajdjshagdjkhsagdjhksagdhjksabdjhsabdjh
sabdsjhabdsajdjshagdjkhsagdjhksagdhjksabdjhsabdjhsabdsjhabdsajdjshagdjkhsagdjhks
agdhjksabdjhsabdjhsabdsjhabdsajdjshagdjkhsagdjhksagdhjksabdjhsabdjhsabdsjhabdsaj
djshagdjkhsagdjhksagdhjksabdjhsabdjhsabdsjhabdsajdjshagdjkhsagdjhksagdhjksabdjhs
abdjhsabdsjhabdsaj
lastline!
```

(output from readF for 20MB file after 'writeT server.txt lastline!' command)

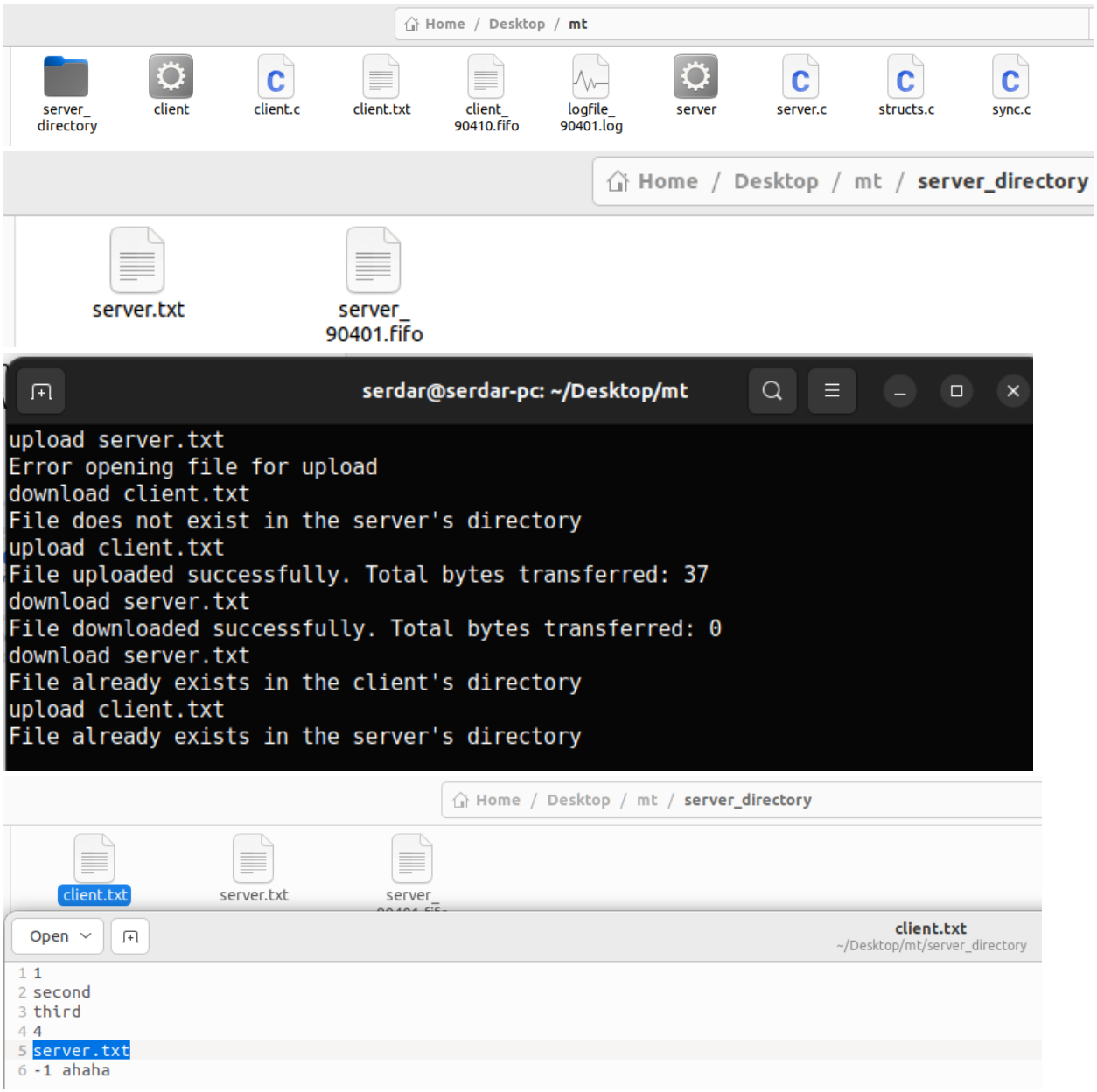
```
readF server.txt
1
2
3
4
writeT server.txt 2 second
Succesfully written into the file.
writeT server.txt 3 third
Succesfully written into the file.
readF server.txt
1
second
third
4
```

writeT <filename> <line> <string> command writes <string> at the end of file when; line argument is smaller than 0, or line argument is null. If line argument is bigger than the number of lines in the file server sends response:

“Line number exceeds the number of lines in the file.” Otherwise, it writes <string> to the line from <line> of file.

For this operation I open a temp file, I write into temp until the selected line. When selected line == current line I write selected line, then I write the rest of the file. At the last part, I rewrite the file with write buffer which reads its data from temp file.

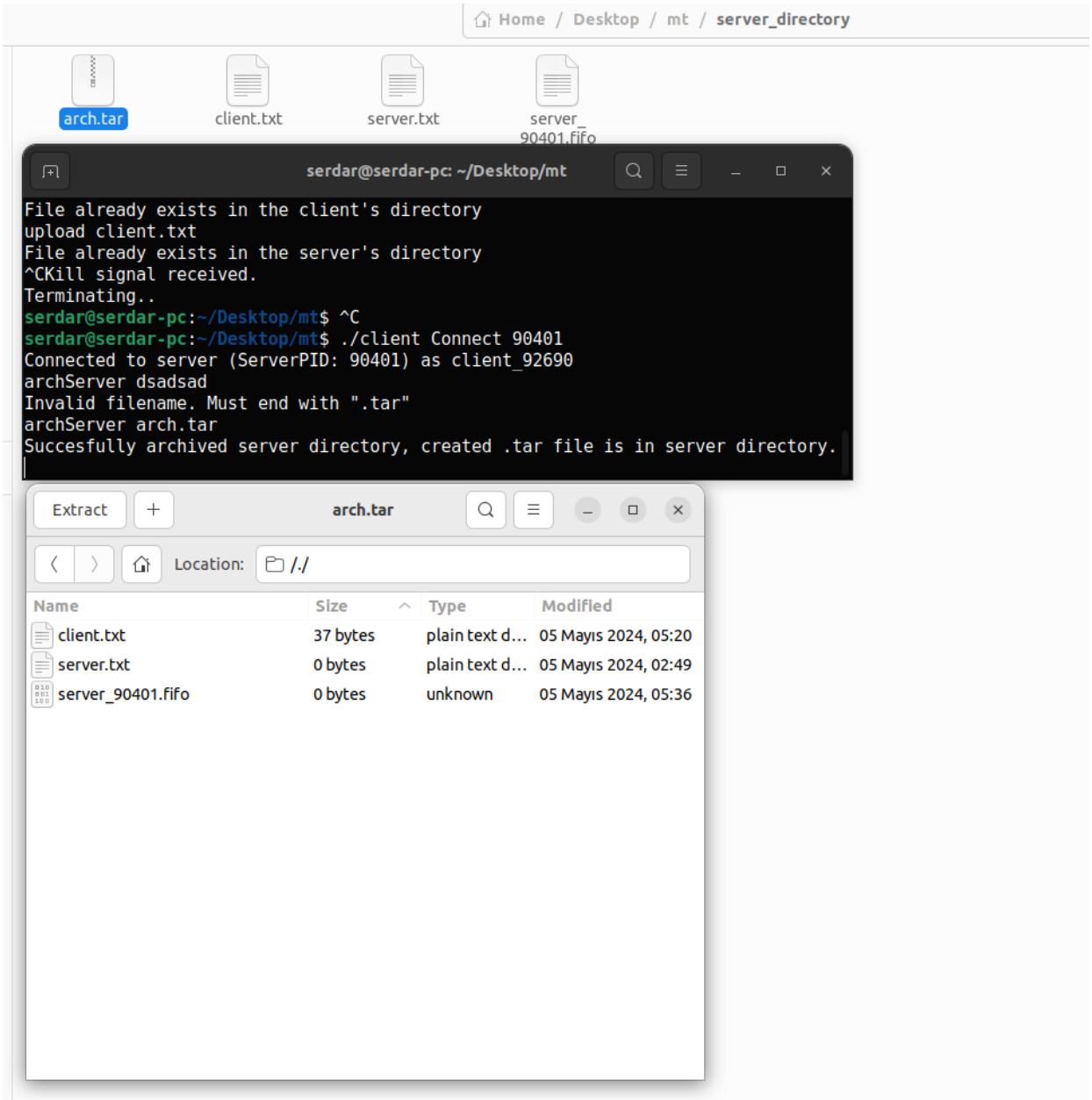
2.5) ‘upload’ and ‘download’ commands:



upload <filename> command creates a file in “server_directory” and copies the data from original file to newly created file.
When successful it prints: ‘File uploaded successfully. Total bytes transferred: %d’ (%d = bytes transferred).
If the original file does not exist, then it prints: ‘Error opening file for upload’.
If the file has already been uploaded to server’s directory before then it prints: ’ File already exists in the server's directory’.

Download <filename> command creates a file in current_directory(the directory that we can writeT into files or readF from files, and the directory that contains log files and client fifos.) and copies the data from the original file (the file from server directory) to newly created file.
When successful it prints: ‘File downloaded successfully. Total bytes transferred: %d’ (%d = bytes transferred).
If the original file does not exist, then it prints: ‘File does not exist in the server's directory’.
If the file has already been uploaded to client’s directory before then it prints: ’ File already exists in the client’s directory’.

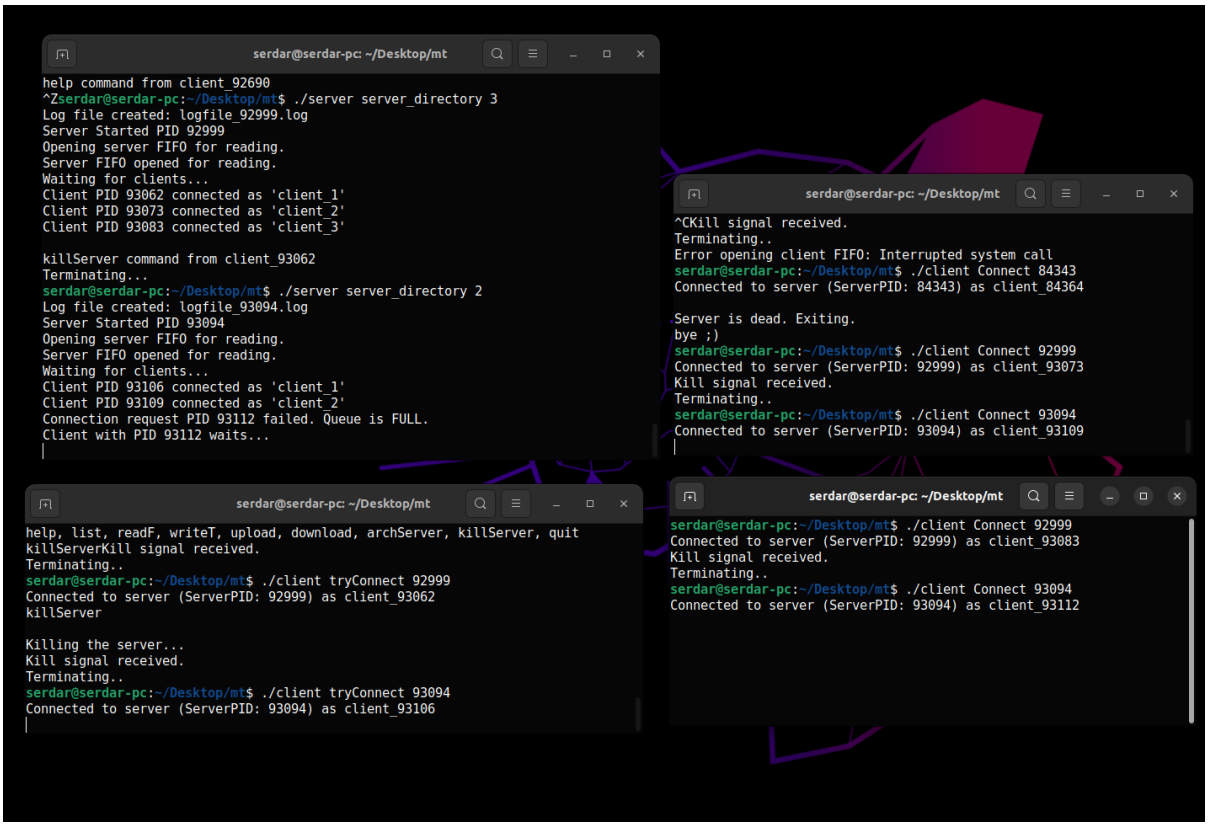
2.6) archServer command:



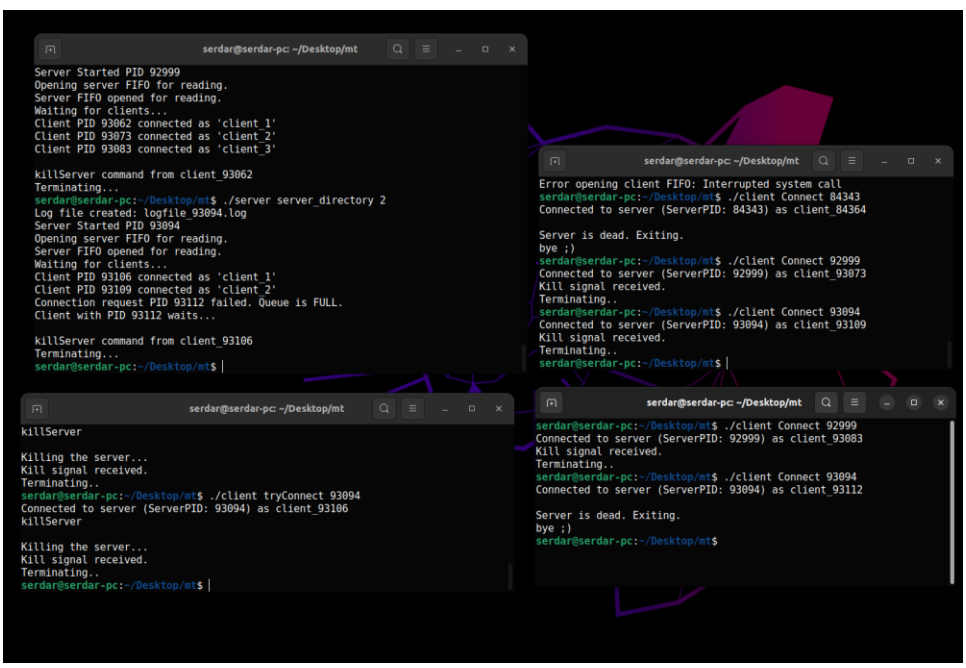
archServer <archiveName> function only accepts archive names that end with '.tar'.

Then it creates a child process and in child process it opens "server_directory" using chdir() function. After changing directory, it calls **execlp("tar", "tar", "-cf", archiveName, ".", NULL);** to create the .tar archive.

2.7) killServer command:



(before killServer, 2 active clients and 1 client waiting in line)



(after killServer in an active client, all clients and server are terminated)

Note: For clients that were waiting in wait_que before termination, it is necessary to press enter in their terminal for terminating safely and unlinking their fifos.

3) Synchronization:

I have a shared memory and in that shared memory I store ***Shared* sd;***

```
typedef struct shared_memory_files {
    int capacity;
    int size;
    File files[64];
} Shared;
typedef struct {
    char name[255];
    int reader_count;
    sem_t reader_mutex;
    sem_t writer_mutex;
} File;
```

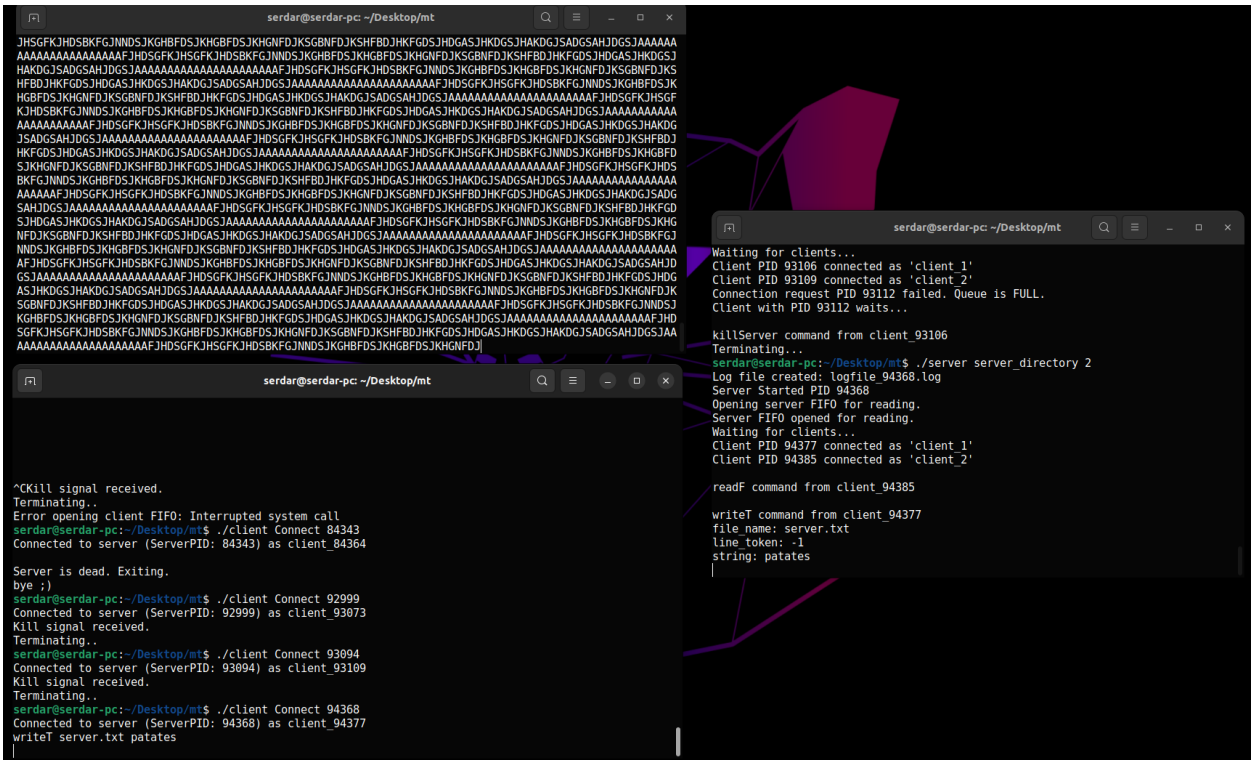
This ***sd*** pointer points to a struct that stores array of File structs which store file names, an integer that holds the count of readers currently reading that file and semaphores to synchronize read and write operations.

At the start of my main code, I initialize files inside my current_directory to shared memory and I always initialize newly created files inside my current_directory (the only way of initialization in runtime is calling writeT with non-existing filename).

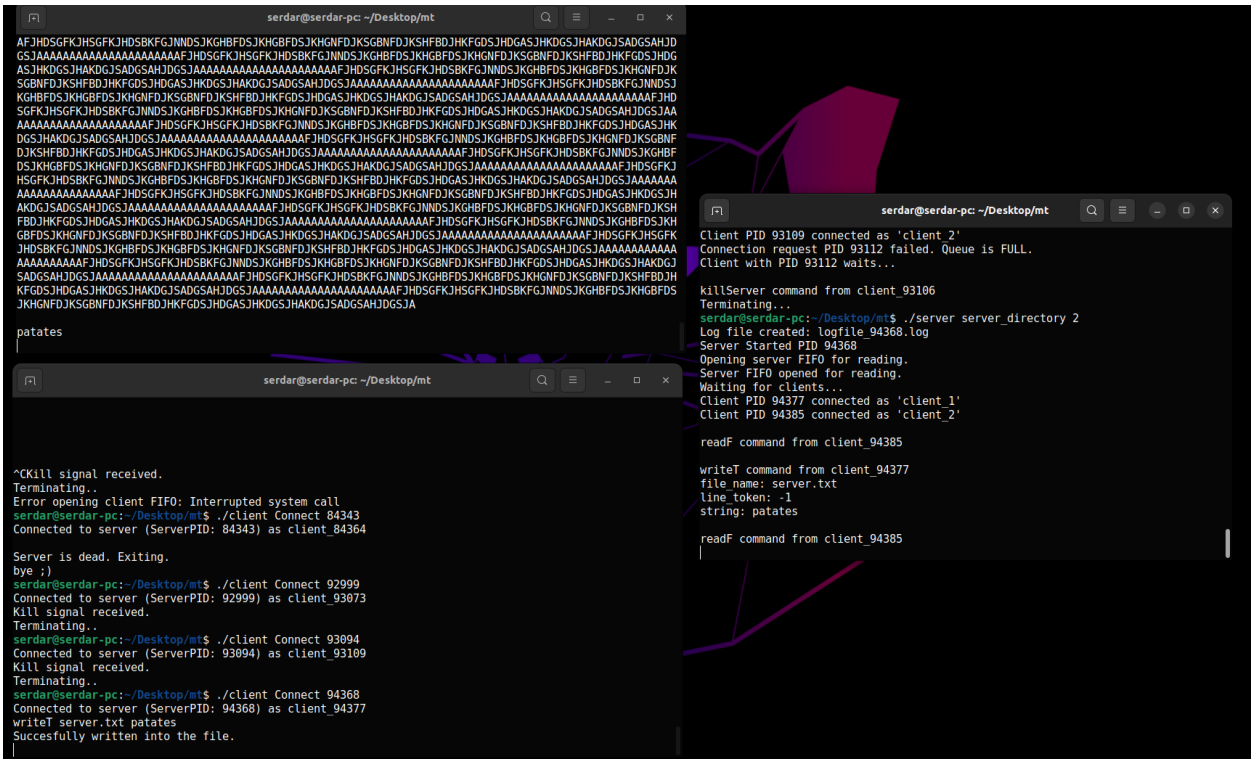
I used semaphores to synchronize my code, I configured my semaphores to allow many amounts of readers but only one writer at the same time. If there is a writer, there can't be any more writers or readers. I defined and used

reader_enter, reader_exit, writer_enter, writer_exit semaphore configurations in writeT, readF, upload and download functions.

3.1) Proof of my synch working:



The client in upper terminal is currently reading a very big file (76 MB) and the client in lower terminal has sent its writeT command, on the right terminal (server terminal) you can see that writeT command is succesfully received but it is not being written yet because it waits for reading to be over.



Here you can see that after reading is done, writeT stops waiting and executes successfully.