_tmp _tmp _tmp _tmp _tmp _tmp _tmp _tmp _tmp _tmp
_tmp _tmp _tmp _tmp _tmp _tmp _tmp _tmp _tmp _tmp _tmp _tmp _tmp
_tmp

**Gebze Technical University**

---

# CSE344 System Programming

MIDTERM PROJECT

2023-24

---

**Lecturer: Erkan Zergeroglu**

Emre YILMAZ 1901042606

# Contents

# 1 Important Notes

1. **Maxiumum queue size is considered as 30. It means that at most 30 client can wait in queue that waits for execution.**
2. No semaphores are used between clients and servers. I assume they run on different computers.
3. Instead of using a single semaphore for read-write operations, a separate semaphore set has been created for each file in the target directory. I have never been convinced that it makes sense to lock the entire directory just because read-write operations are being performed on one file. File-based locking is much more efficient.
4. You can check the log txt files to see which file entered the critical section and when.
5. Initially, semaphores were used for connection-registration processes, but they were later deactivated. You will see this in the following sections. The reason for this deactivation is that clients attempting to connect in tryConnect mode do not receive a response when the server's main process is blocked. Therefore, semaphores were disabled in favor of spin locks. I do not remove them from project.
6. It has been assumed that file names will not contain space characters.
7. **When you examine the examples, you will see that operations on different types of files (such as binary and text) have been successfully completed. Additionally, there are examples of large (greater than 10MB) file transfers.**
8. I've tried to test edge cases as much as possible, which you can see in the examples. Additionally, I made sure to unlink all FIFOs when clients die or when the server receives a CTRL-C signal. You can also see this in the outputs.
9. I have sent three Valgrind outputs, the first one is for the server, and the others are for clients.
10. **I changed the names of the files generated by the make command while preparing the report. Although you might see different behaviors of the client and server in some outputs, please run the program using the `neHosClient` and `neHosServer` commands.**
11. For any problem: eyilmaz2019@gtu.edu.tr, +905319346629
12. Best Regards.

## 2 How to Run

In the main directory (inside the 1901042606_midterm folder), compile with make. Afterwards, in the same directory, you can start the client with ./neHosClient <connect mode> <server pid> and the server with ./neHosServer <directory> <max client>

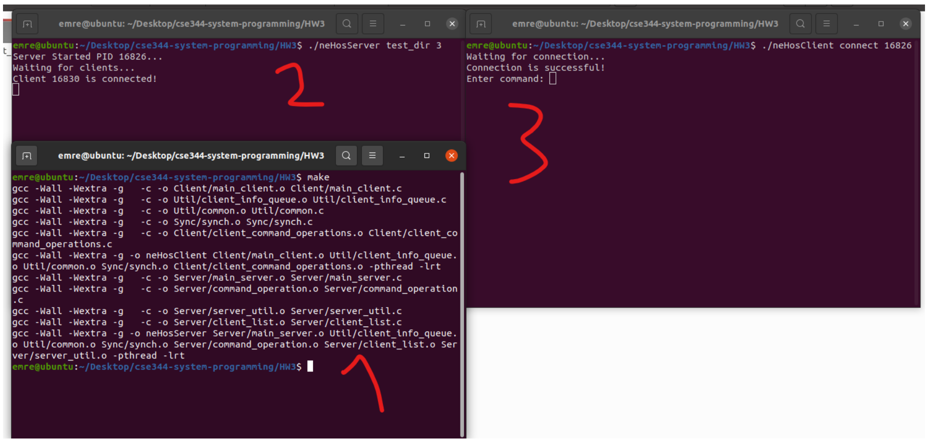


Fig: How to Run

# 3 Communication Architecture

There is a FIFO that is unique and exclusive to the server. Client registrations are done through this command. While the server is reading, all clients registered to that server are writing to this FIFO.When creating this server-specific FIFO, the server's process ID was used.

Apart from the registration process, all other communication (sending commands, sending results to the server, receiving results from the server) is carried out through two other FIFO type. But this 2 type is unique for each client. I mean, every client has 2 unique FIFO. One of these can be called the request FIFO and the other the response FIFO. The request FIFO is open in read mode on the server side and in write mode on the client side. The response FIFO is open in write mode on the server side and in read mode on the client side. All communication between the client and the server, except for the registration process, takes place through these two FIFOs. These FIFOs are created using the process IDs of the connected client.

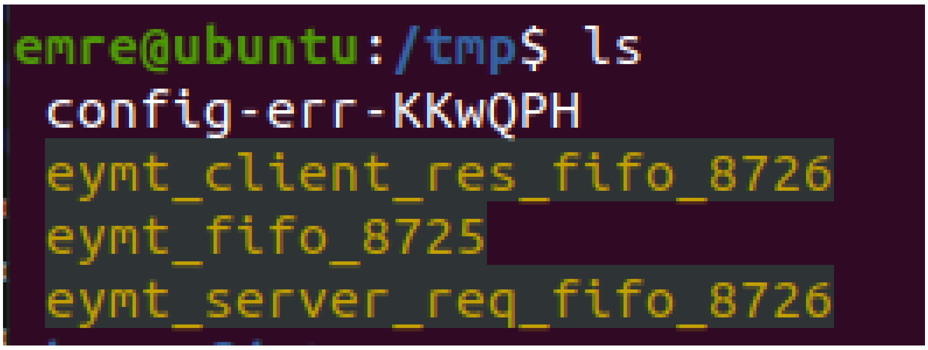


Fig-0.1: FIFOs While 1 Client is Active

# 4 Registering Clients and Queue Mechanic

There are two separate data structures where clients are stored.

## 4.1 Client Queue

This queue contains clients waiting to be executed that are connected or not connected. Whenever a spot opens up, the next user in the queue connects to the server.

## 4.2 Client List

This list contains all clients that are currently being executed. When a client sends a "Quit" command, receives a SIGINT command with CTRL-C, or encounters an error, it sends the necessary information to the server, and the server removes the relevant client from this list named "client list".

## 4.3 Registration Algorithm

The registration algorithm is based on busy waiting. It constantly attempts to read, and upon successfully reading, it dequeues the client to try to connect them to the server. If there is no space in the queue, the necessary warning messages are printed, and the loop continues.

You can see from the sample outputs that simultaneous connection requests are also successfully handled.

The algorithm for clients connecting to the server is as follows:

- If the FIFO read operation is not zero, we add the incoming client to the queue and the client list. An important point in this item is that if the incoming client tries to connect in tryConnect mode, we do not add that client anywhere at this step.
- Next, we find out the total number of clients that have already connected and check if this exceeds the maximum client count. If it does and the number of bytes read is not zero (indicating a new request), we print the necessary warning on the screen and check the client's connection mode. If it is in tryConnect mode, we send the information that the connection was unsuccessful to the client's FIFO.
- If the maximum client count has not been reached, we dequeue, send a message to the client's FIFO indicating that the user has connected, and start the operation. It should also be noted that before this step, it is checked whether the user is in tryConnect mode. If so, they are added to the queue and the client list, and then the dequeue operation takes place.

```c
ssize_t num_read = read(server_fd, &cli_info, sizeof(cli_info));
if (num_read == -1)
{
    perror("Failed to read from server FIFO");
    cleanup(server_fd, -1, -1, dir_syncs);
    exit(EXIT_FAILURE);
}
else if (num_read == 0 && is_queue_empty(&client_queue))
{
    // If no data is read and the client queue is empty, continue to the next iteration
    continue;
}
else if (num_read != 0)
{
    // Enqueue the client info into the client queue and add the client to the list of clients
    char log_message[256];
    sprintf(log_message, "Client %d is queued.\n", cli_info.pid);
    write_log_file(log_message, dir_syncs);
    if(cli_info.mode!=1)
    {
        enqueue(&client_queue, cli_info);
        add_client(list_clients, cli_info.pid);
    }
}
```

Fig-1: Registration Algo-1

```c
if (working_clients(cli_info) == max_clients)
{
    char log_message[256];
    if(num_read!=0)
    {
        sprintf(log_message, "Connection request %d is refused. Queue is full.\n", cli_info.pid);
        write_log_file(log_message, dir_syncs);
        printf("Connection request %d is refused. Queue is full. \n", cli_info.pid);
    }

    if(cli_info.mode==1 && cli_info.pid != -1)
    {
        char log_message[256];
        sprintf(log_message, "Client %d is refused. Its mode is tryConnect and queue is full.\n", cli_info.pid);
        printf("Client %d is refused. Its mode is tryConnect and queue is full.\n", cli_info.pid);
        write_log_file(log_message, dir_syncs);
        char client_res_fifo[256]; // Client response FIFO
        sprintf(client_res_fifo, CLIENT_RES_FIFO, cli_info.pid); // Create the client response FIFO
        int client_res_fd = open(client_res_fifo, O_WRONLY); // Open the client response FIFO

        if (client_res_fd == -1)
        {
            perror("Failed to open client response FIFO");
            cleanup(server_fd, client_res_fd, -1, dir_syncs);
            exit(EXIT_FAILURE);
        }

        char response = '0'; // This will mean that connection is successful
        ssize_t num_written = write(client_res_fd, &response, sizeof(response)); // Write the response to the client response FIFO
        if (num_written == -1)
        {
            perror("Failed to write to client response FIFO");
            cleanup(server_fd, client_res_fd, -1, dir_syncs);
            exit(EXIT_FAILURE);
        }
        continue;
    }
    else
    {
        continue;
    }
```

Fig-2: Registration Algo-2

```c
else if(cli_info.pid!=-1 && cli_info.mode==1)
{
    enqueue(&client_queue, cli_info);
    add_client(list_clients, cli_info.pid);
}

cli_info = dequeue(&client_queue);
char log_message[256];
sprintf(log_message, "Client %d is dequeued and connected. It is ready for operations.\n", cli_info.pid);
write_log_file(log_message, dir_syncs);

pid_t child_pid = fork();
if (child_pid == -1)
{
    perror("Failed to fork child process");
    cleanup(server_fd, -1, -1, dir_syncs);
    exit(EXIT_FAILURE);
}
```

Fig-3: Registration Algo-3

## 4.4 Output of Registrations

### 4.4.1 Queue Test

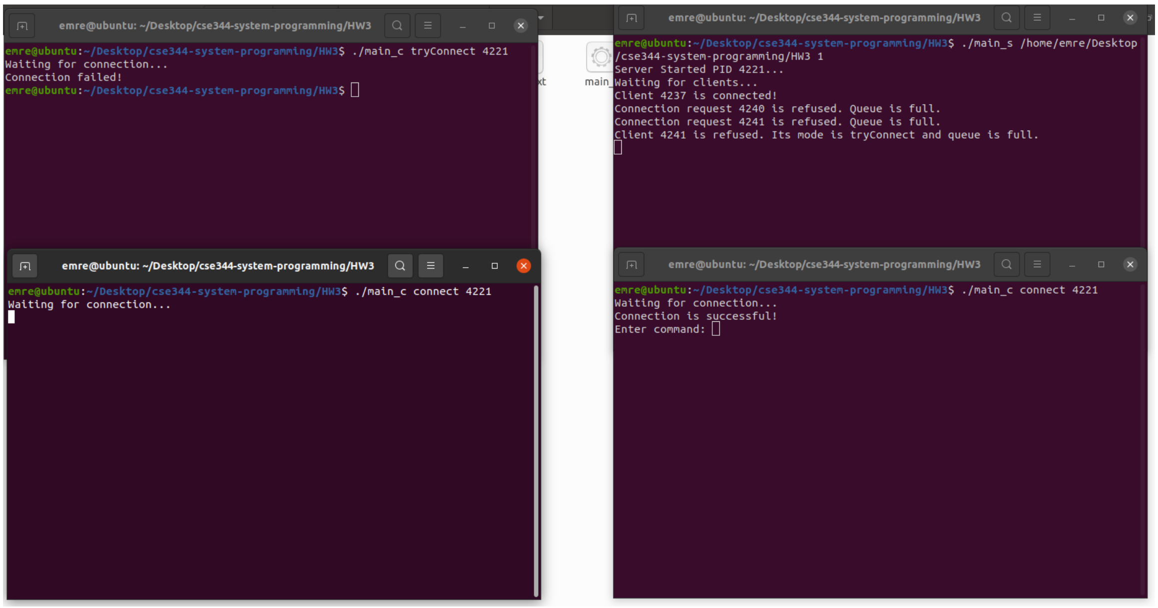


Fig-4: Output Example of Registration

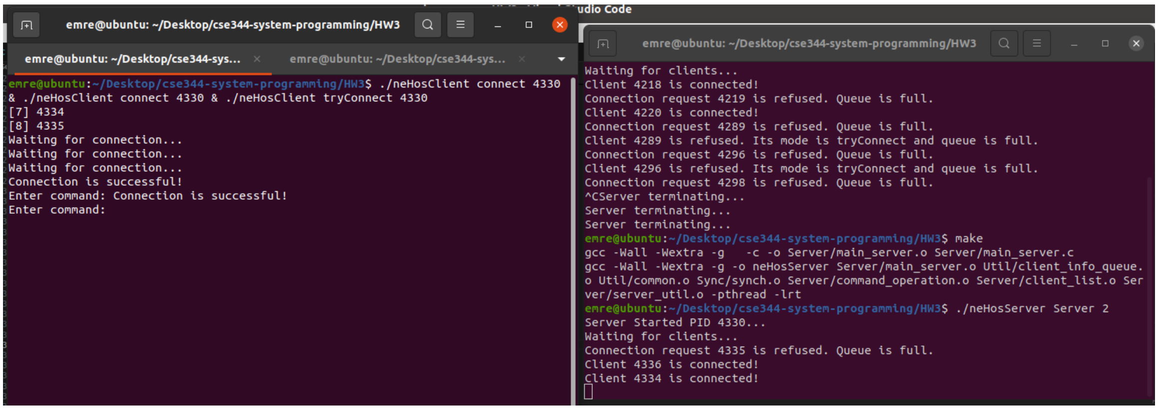


Fig-4.1: Output Example of Simultanuous Registration

**NOTE**: In this simultaneous registration process in Fig-4.1, the command I entered runs the processes as background processes and returns control to the shell. Therefore, you might not be able to send commands effectively. After testing the simultaneous connection, I strongly recommend that you start the client normally from the terminal with single execution and then enter the commands. This ensures that the commands are handled correctly without the complications of background processing, allowing for more stable and predictable interactions with the server.

### 4.4.2 Enqueue Test

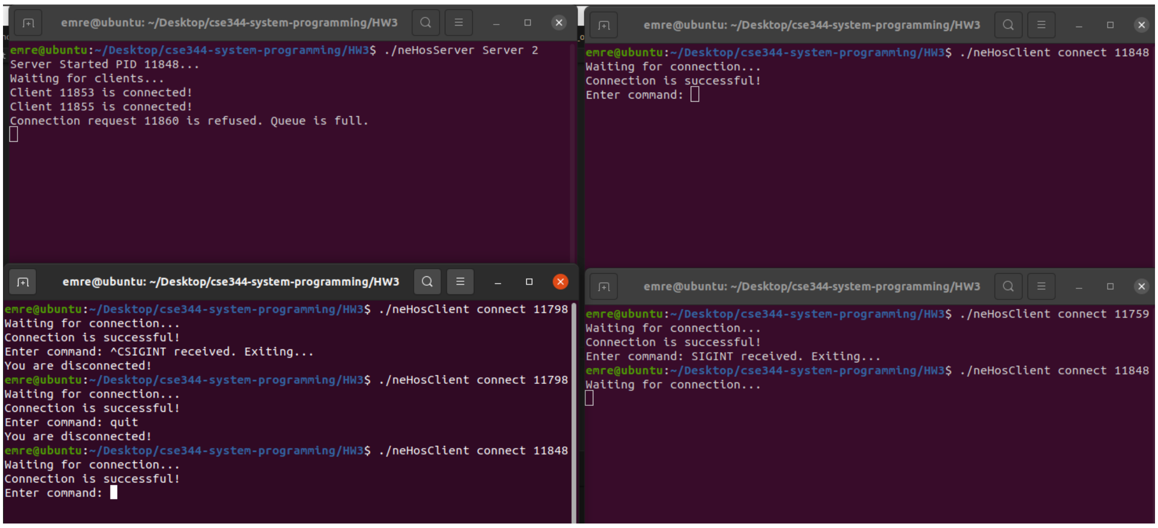


Fig-4.2: Queue Situation Before One of Them Exiting

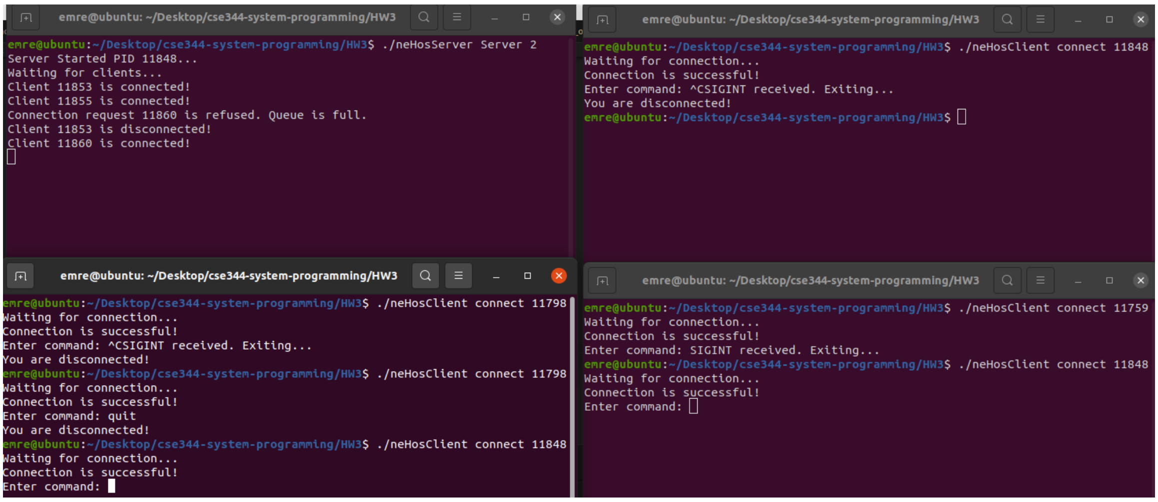


Fig-4.3: Queue Situation After One of Them Exiting

As you can see in Fig 4.2 and Fig 4.3, when one of the clients dies, the first client waiting in the queue is processed and execution begins.

# 5 Synchronization Module

## 5.1 Synchronization for Registration

There is no synchronization for client registration, and there are several reasons for this:

1. I did not find it appropriate to use semaphores between clients. It seems reasonable to assume that they operate on different machines.
2. For the same reason mentioned above, I did not use any semaphores between the client and the server.
3. In most systems, FIFO write operations are atomic up to 4096 bytes. And the size that the client writes and the server reads for registration is only about 2 integers (8 bytes), so we can say it is guaranteed to be atomic. You can see in the later sections that the test results for simultaneous registration have been successfully completed.

## 5.2 Synchronization for File Operations

```c
#define FILE_NAME_LENGTH 256
#define FILE_LIMIT 256

#define UPLOAD_AND_LIST_SYNC_FILE "05319346629"

/**
 * @struct Semaphores
 * @brief Structure to hold the semaphores used for synchronization.
 *
 * This structure holds the semaphores used for synchronization in the system.
 * It includes semaphores for mutex, empty, full, and list directory mutex.
 */
typedef struct {
    sem_t mutex;            /**< Semaphore for mutual exclusion */
    sem_t empty;            /**< Semaphore for empty condition */
    sem_t full;             /**< Semaphore for full condition */
    sem_t list_dir_mutex;   /**< Semaphore for list directory mutex */
} Semaphores;

/**
 * @struct file_sync
 * @brief Structure to represent a safe file.
 *
 * This structure represents a safe file in the directory.
 * It includes the file name, reader count, writer count, and various semaphores for synchronization.
 */
struct file_sync {
    sem_t readTry;          /**< Semaphore for readers trying to enter the critical region */
    sem_t rMutex;           /**< Semaphore for readers mutual exclusion */
    sem_t wMutex;           /**< Semaphore for writers mutual exclusion */
    sem_t lock;              /**< Semaphore for readers synchronization */
    char fname[FILE_NAME_LENGTH];  /**< File name */
    int readerCount;        /**< Number of readers */
    int writerCount;        /**< Number of writers */
};

/**
 * @struct dir_sync
 * @brief Structure to represent a safe directory.
 *
 * This structure represents a safe directory.
 * It includes an array of safe files, semaphores for synchronization, and other metadata.
 */
struct dir_sync {
    struct file_sync files[FILE_LIMIT];   /**< Array of safe files in the directory */
    Semaphores sems;                           /**< Semaphores for synchronization */
    int size;                                  /**< Current number of files in the directory */
    int capacity;                              /**< Maximum capacity of the directory */
};
```

Fig-5: Synchronization

The synchronization module is straightforward. It tries to handle the classic reader-writer problem, with assistance taken from an Operating Systems course textbook. The main idea is to maintain a separate synchronization structure for each file, as it was deemed impractical to lock the entire directory for all clients just because one user is reading. Here are the key points of the module:

- The main structure for synchronization and security is called `dir_sync`. It contains one `Semaphore` struct and an array called `file_sync`, which will manage synchronization operations for each file.
- The `Semaphore` struct was initially used for connections but was later deactivated. It is not used anywhere in the project currently; however, it has not been removed, thinking it might be needed in the future.

Let me clarify sync variables:

1. `readTry`: This semaphore is used to control when readers can try to access the shared resource. It primarily serves to prevent new readers from starting while a writer is waiting, thus avoiding potential reader preference and writer starvation.
2. `rMutex`: This semaphore provides mutual exclusion among readers specifically for accessing and modifying the readerCount variable.
3. `wMutex`: This semaphore is used for mutual exclusion among writers. It ensures that only one writer at a time can check or modify the writerCount, which helps coordinate access to the resource.
4. `lock`: This semaphore is crucial for controlling actual access to the shared resource. It prevents writers from writing while any readers are active and vice versa.

**Important Note:** For observations related to the critical section, please review the log file. If you want to see it in real-time, you can try to write to the file a.txt with another client before the reading process is completed by issuing the readF command. I tried this and observed that it waited in the critical section.

Additionally, the critical section algorithm dynamically adds and removes files to and from a structure dir_sync. The program also operates correctly with file additions from external sources.

### 5.2.1 How to Test File Synchronization

1. Both the /Server and `/target_dir` folders contain a text file named `a.txt`, which is around 13 megabytes in size. After starting the server, if you enter the command `"readF a.txt"` from one client and then quickly enter the command `"writeF a.txt appendToEnd"` from another client, you will observe that the write command has to wait for a while to enter the critical section.
2. For a clearer observation, you can enable the sleep function located on line 286 in the `"Server/command_operation.c"` file, which is currently commented out. This will cause a 10-second delay before exiting the critical section when you want to read the entire file. You will be able to observe this delay easily when you try to write to the same file from another client.
3. If you also examine the log files, you can see that the synchronization has been successfully completed.

# 6 Shared Memory

Shared memory has been utilized for two distinct structures.

1. The first is semaphores. Since unnamed semaphores are used, shared memory is employed to share these semaphores among children and parent processes.

2. The second use is for what's referred to as the `client_list`, a list that holds all currently active clients. The reason for storing this list in shared memory is to enable the termination of all clients when necessary. This setup allows any process that has access to the shared memory to quickly and efficiently update, read, or manage the list of clients, ensuring coordinated control over client processes.

# 7 List Command

## 7.1 Implementation

After the list command reaches the server, the server retrieves each file located in the directory and writes them to the FIFO, appending a newline character after each file. This allows the client to easily print them out. Once the server finishes writing, it closes the `client response` FIFO, which causes the client's response FIFO read function to return 0, ending the reading process. Finally, the server attempts to read from the `client request FIFO` and blocks. After the client finishes reading, it writes a value to this FIFO, unlocking the block. This unlocking triggers the server's code block to reopen the `client response` FIFO.

## 7.2 List Test

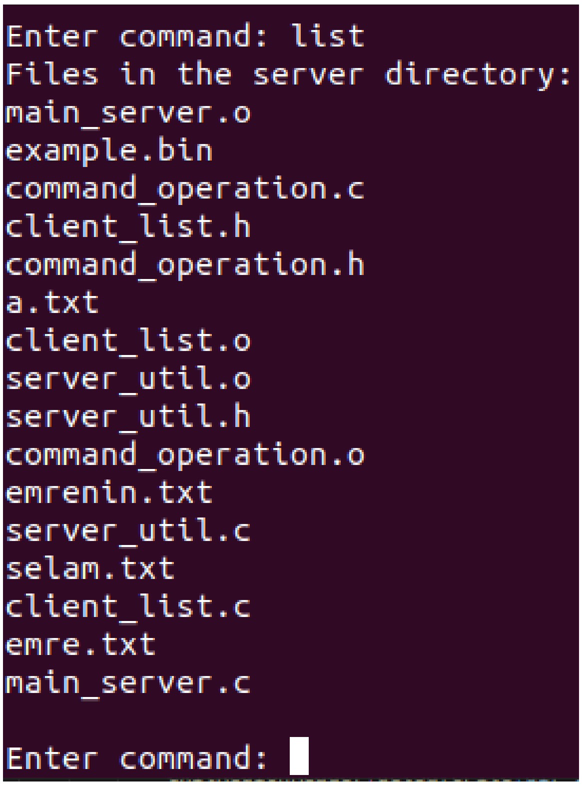


Fig-6: List Command Result

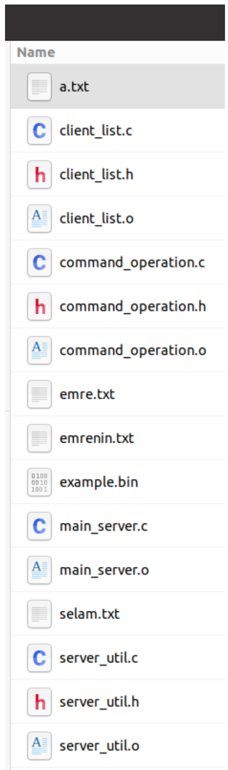


Fig-7: Server directory

# 8 ReadF Command

## 8.1 Implementation

In the `readF` function on the server side, the `handle_readF_command` function is triggered and starts running. Depending on whether it attempts to read the entire line or not, it calls the helper functions `handle_whole_file` and `handle_specific_line`.

Both of these functions work in a similar manner. The function starts operating while the two FIFOs belonging to the client (response and request FIFOs) are open. It opens the file in read mode using fopen, writes to the `client_response` FIFO using write. After the writing process is completed, it closes the `client_response` FIFO, which causes the client's FIFO read function to return 0, indicating that the reading process is finished. After the client completes reading, it writes a value to the server's FIFO. Once the server reads this value, it reopens the `request_fifo` and thus the writing process is completed.

The `handle_specific_line` function finds newline character pointers and uses `lseek` to handle the line reading operations.

## 8.2 readF Test

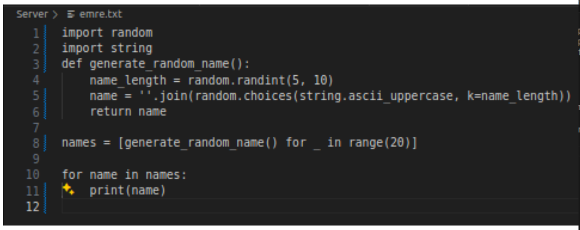


Fig-8: Initial text of emre.txt


Fig-9: Reading whole file

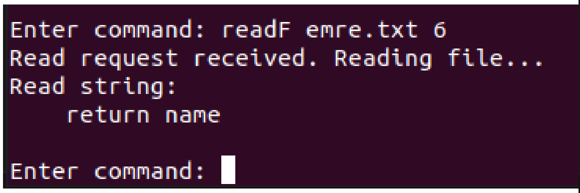


Fig-10: Reading specific line

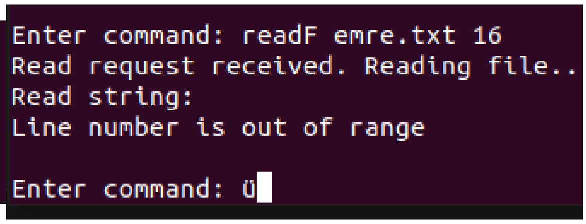


Fig-11: Error handling example

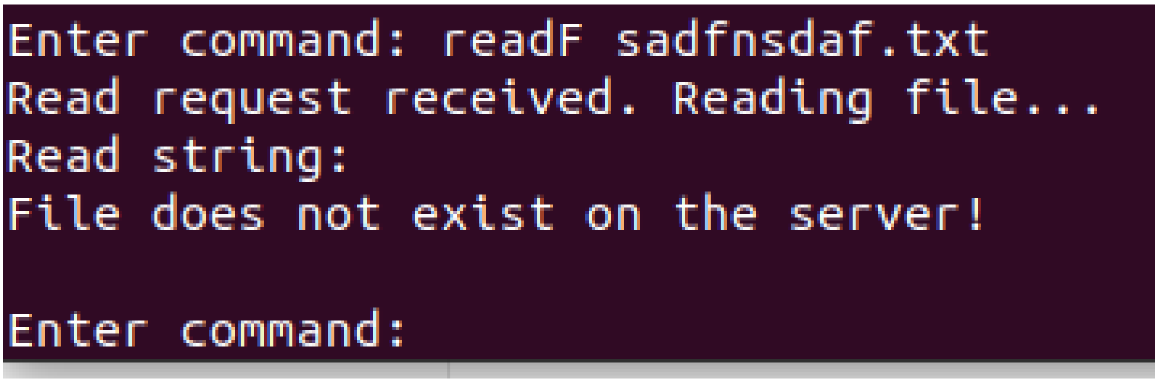


Fig-11.2: Trying to Read A File Does not Exist

As you see in Fig-11 and Fig-11.2, if you try to read a file that does not exist or try to read a line number that is not valid, you will encounter an error.

# 9 WriteF Command

## 9.1 Implementation

When a command is sent from the client to the server, the client specifies which file it wants to write to, and if applicable, the line number, and then closes the `client_request` FIFO. Immediately afterwards, it tries to read from the `client_response` FIFO and becomes blocked. On the server side, the relevant function is directed with the parameters. If no line number parameter is given, the function is straightforward. If the relevant file is found, it opens the file in append mode and appends the necessary text to the end. Then, it sends feedback to the client about the success of the write operation.

If there is a line number parameter, the situation is a bit more complex. Reading from the file and writing to a temporary file continues until the specified line is found. After the line number is reached, the message from the client is added to this temp file, and the read-write operations continue. This way, the message is inserted into the specified line. Finally, the file that the client wants to modify is deleted from the system, and the name of the temp file is changed to this file name, achieving the final desired state of the file. At the end of this function, feedback is sent to the client regarding whether the write operation was successful or not. The client prints this message on the screen, and the operation is completed.

## 9.2 writeF Test

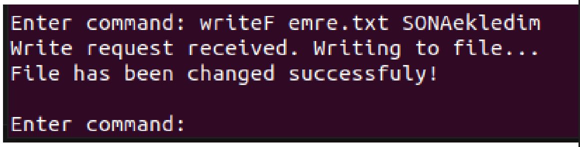


Fig-12: writeF Without Line Number

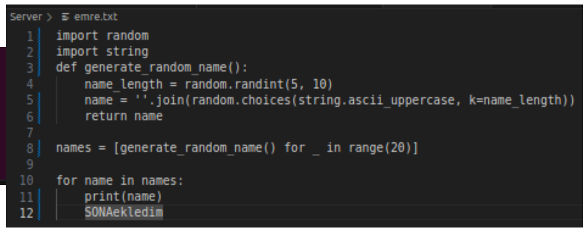


Fig-13: Result of writeF in Fig-12

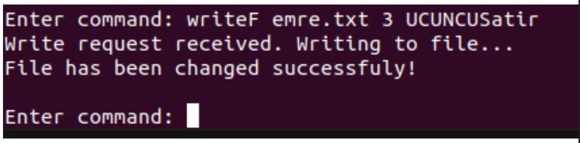


Fig-14: writeF With Line Number

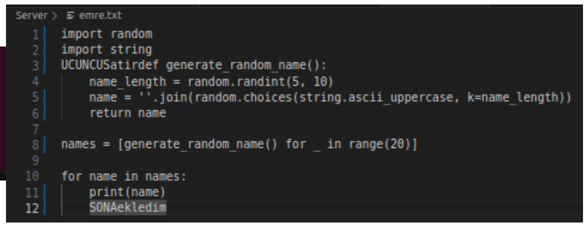


Fig-15: Result of writeF in Fig-14

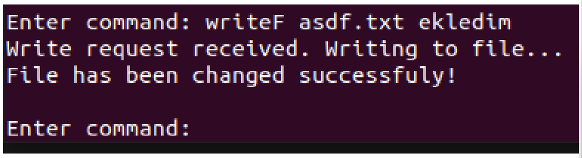


Fig-16: writeF and New File Creation

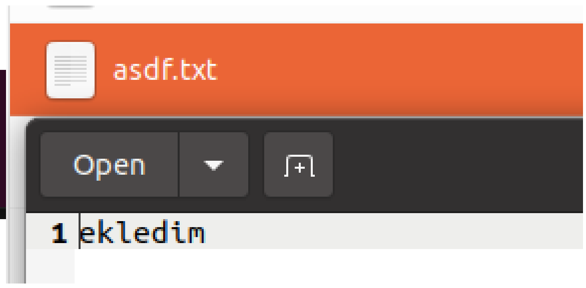


Fig-17: Result of writeF in Fig-16

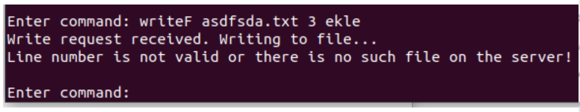


Fig-17.2: File Does not Exist

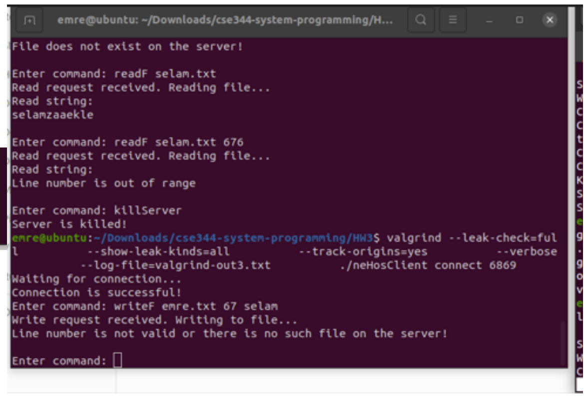


Fig-17.3: Invalid Line Number

As you see in Fig 17.2 and Fig-17.3, if there is no such file that user want to write OR if the line number that user wants to write is not valid, it will be encountered an error.

# 10 Download Command

## 10.1 Implementation

It is the same implementation as readF command. The only difference is in client side. Client do not print the response of server, instead it prints it to the file that client wants to download. If there is no such file in server side, it prints the error message. Additionally, it checks if the downloaded file name exists on the client side; if it does, it appends "(1)" to the downloaded file name.

## 10.2 download Test

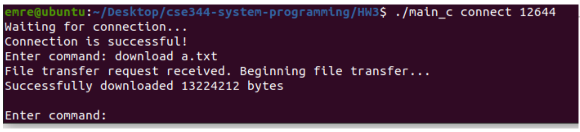


Fig-18: download Command for 13.2MB Text File

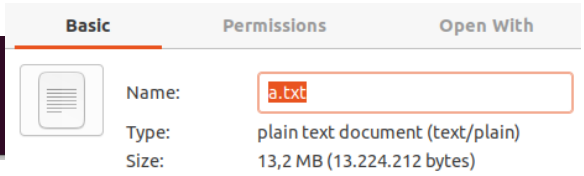


Fig-19: Size of Downloaded File in Server Side

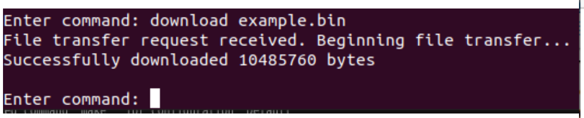


Fig-20: download Command for 10.5MB Binary File

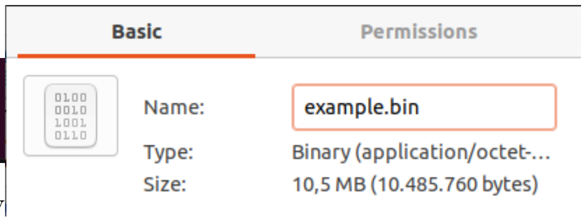


Fig-21: Size of Downloaded Binary File in Server Side

As you can see in Fig-18, Fig-19, Fig-20 and Fig-21, program can download binary and text files successfuly. Note that the binary file size is 10.5 MB and text file size is 13.2 MB.

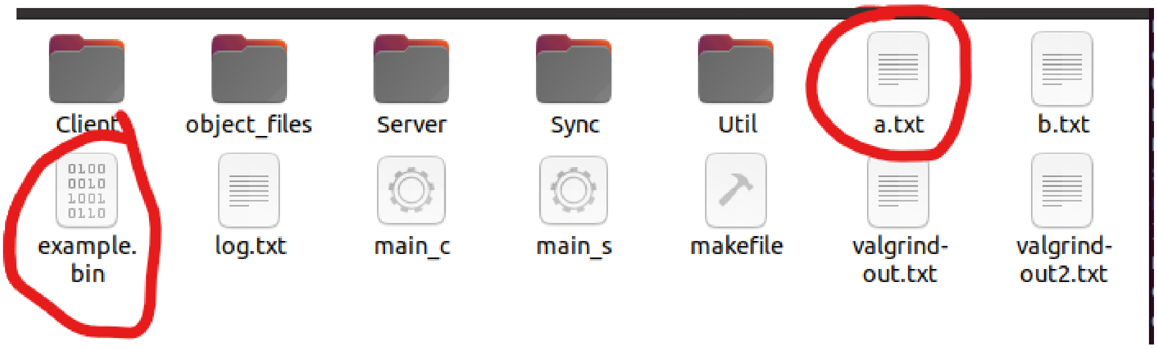


Fig-21.2: Client Directory After Downloading Txt and Bin

- **You can see in Fig-23, there are files a.txt and a(1).txt.**

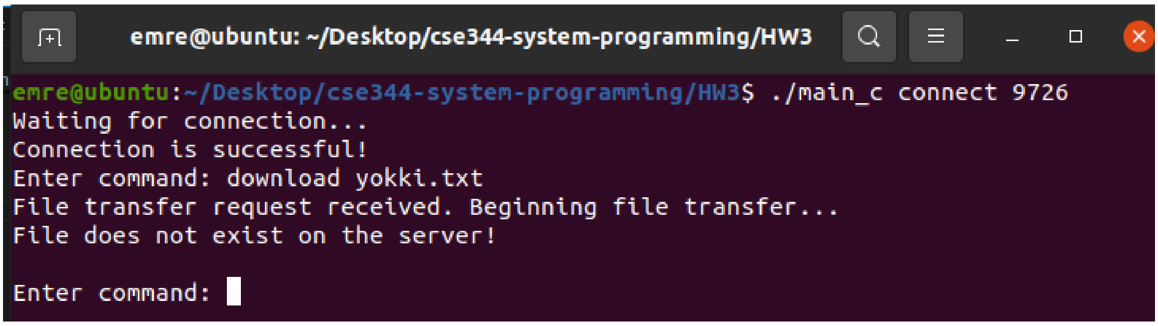


Fig-23.1: Trying to Download No-Existed File

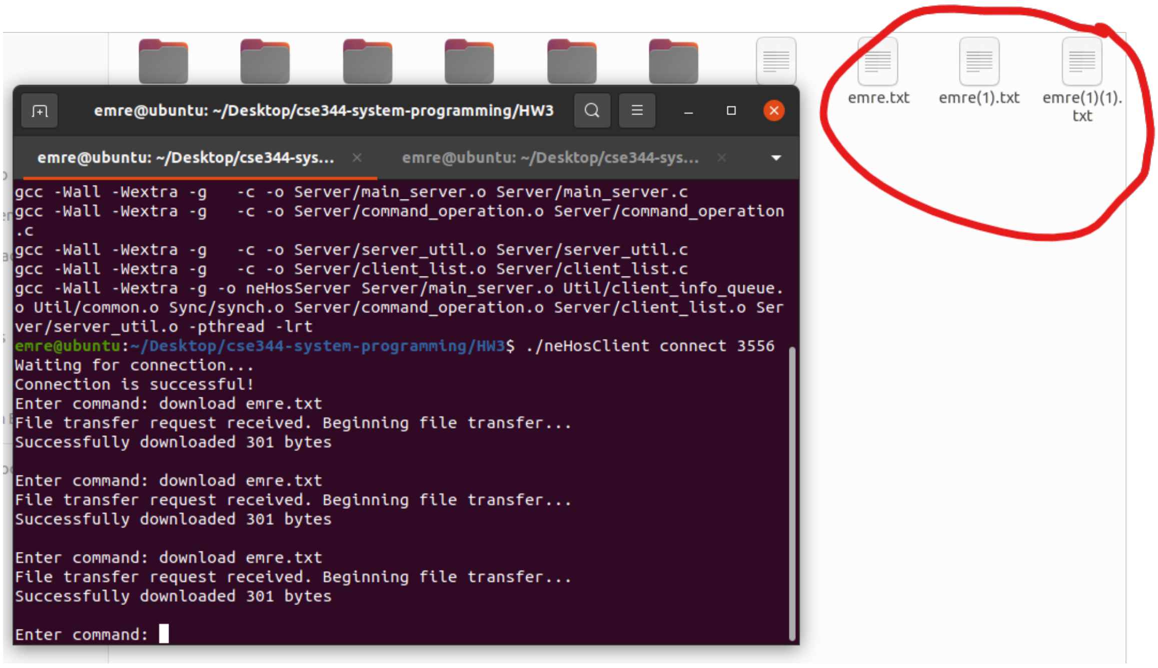


Fig-23.2: Trying to Download a File 3 Times

As you see, if you download same file 3 times, it changes its name each time. Also, if there is no such file in server-side, we get an error message. Program can handle different data types as you see in Fig-18 and Fig-20.

# 11 Upload Command

## 11.1 Implementation

For this command, the client first opens the file in read mode ('R') and performs the reading using the read function. It writes the read content to the `client_req` FIFO. After the writing process is completed, the client closes the `client_req` FIFO, which signals to the server that the reading is finished since the read function returns 0 and the reading operation ends. Once the server completes its reading operation, it sends a message to the `client_res` FIFO indicating that the reading is complete. Upon receiving this message, the client reopens the `client_req` FIFO, and the upload process is successfully completed. When saving the file to its directory, the server first checks for the existence of the file; if a file with the same name exists, it saves it by adding the string `"(1)"` to the name.

## 11.2 Upload Test

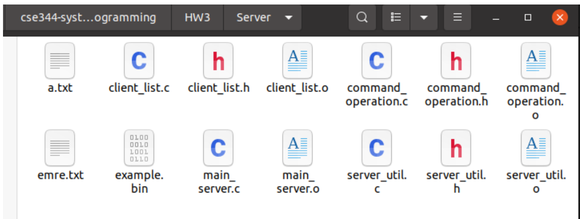


Fig-23: Initial Directory of Server Side

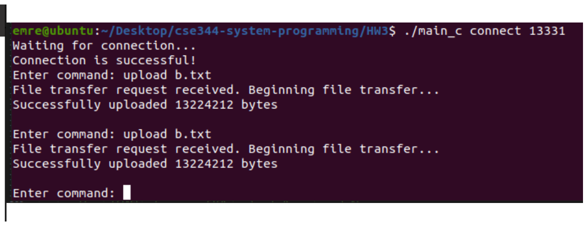


Fig-24: Two Upload Command for Same File

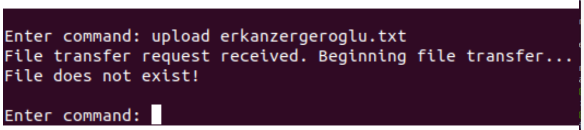


Fig-25: Try to Upload No-Existing File

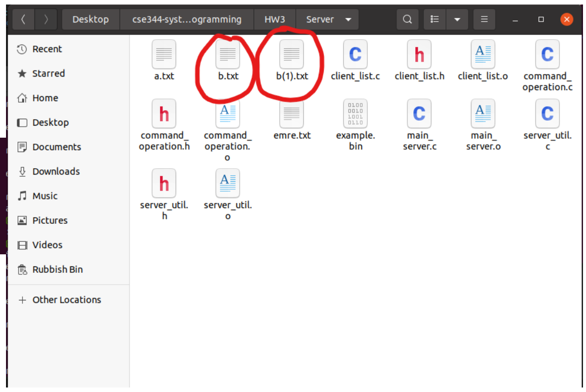


Fig-26: Final Situation of Server Side

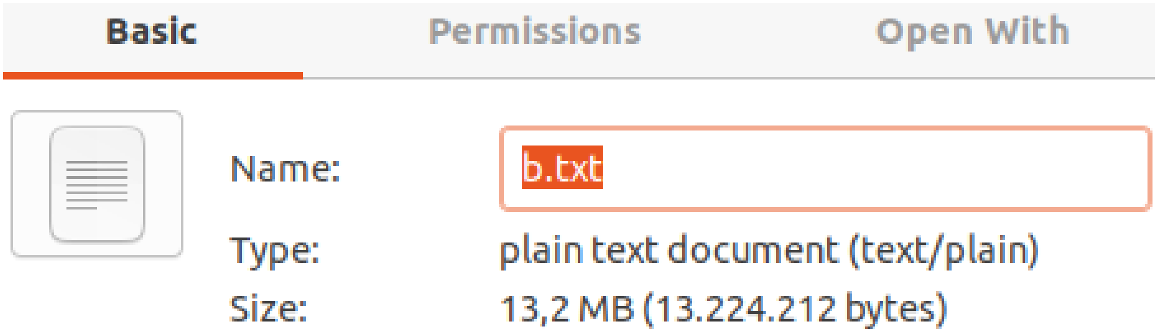


Fig-27: Uploaded File Size

# 12 KillServer Command

## 12.1 Implementation

First, a kill command is sent to the server. After receiving the command, the server sends a message back to the client confirming that it has received the command. Upon receiving this message, the client immediately terminates its own process. The server then sends a SIGINT signal to each member of the process group, effectively terminating them one by one. The parent process, upon receiving the SIGINT signal, sends a SIGINT command to each of the client processes that are currently running and stored in shared memory, terminating them as well. This ensures that there are no running client or server processes left.

## 12.2 KillServer Test

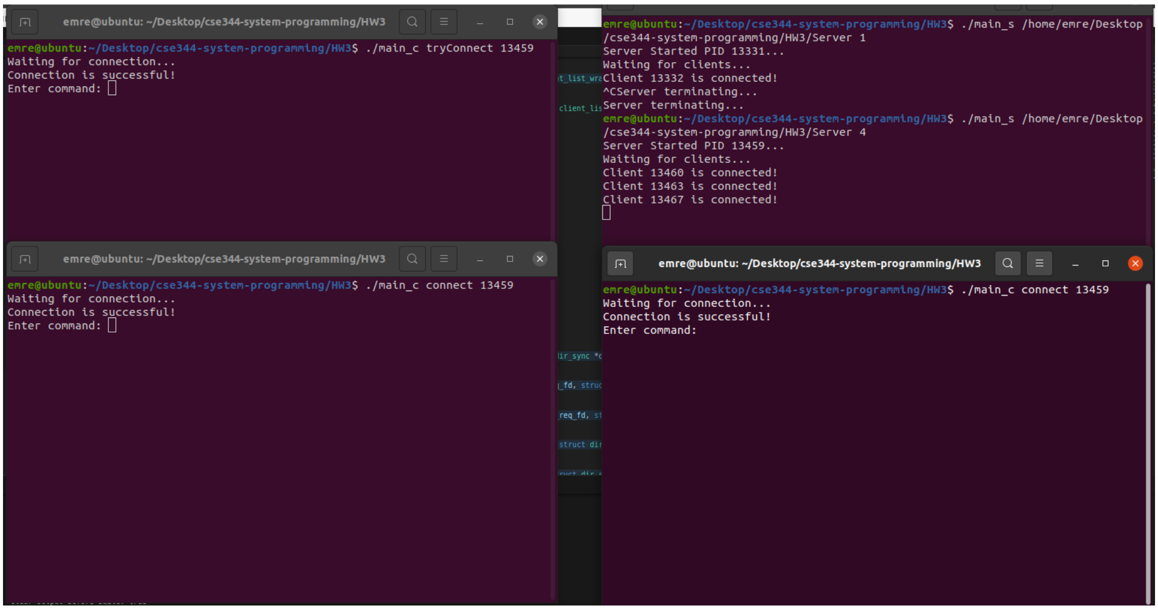


Fig-28: Before the killServer Command

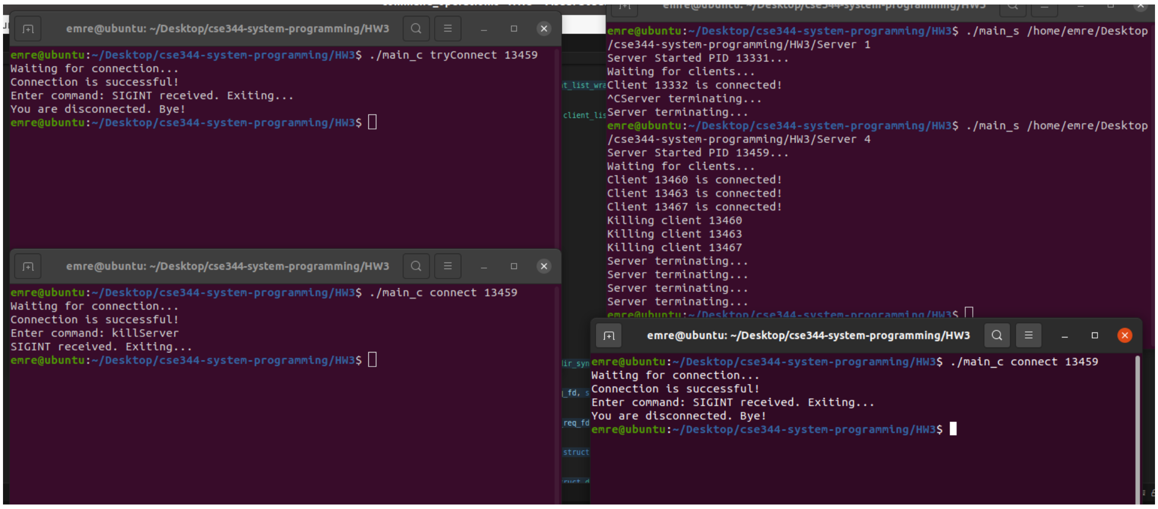


Fig-29: After the killServer Command

# 13 Quit Command

## 13.1 Implementation

When the parent process receives the quit signal, it removes the client from the list of processes held in shared memory. It notifies the child that the quit command has been successfully received, cleans up its own resources, and then terminates itself with an exit signal. Upon learning that the removal process was successful, the client then terminates itself, stopping its operation.

## 13.2 Quit Test

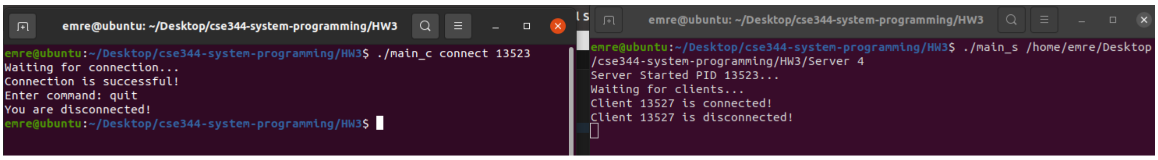


Fig-30: Quiz Command

# 14 Archive Command

## 14.1 Implementation

This is handled by the combined use of two commands. The first command is the `archServer` command. When this command reaches the server side, the handler function performs a `fork`, and then uses `execve` to save the entire directory into a TAR file. Once the operation is complete, it sends a message to the client over the FIFO, informing them that the TAR process is finished. Upon receiving this message, the client learns that the TAR operation is complete and initiates the download command to download the file.

During the TAR process, all files located on the server are placed into a critical section. This prevents disruption of synchronization.

## 14.2 archServer Test

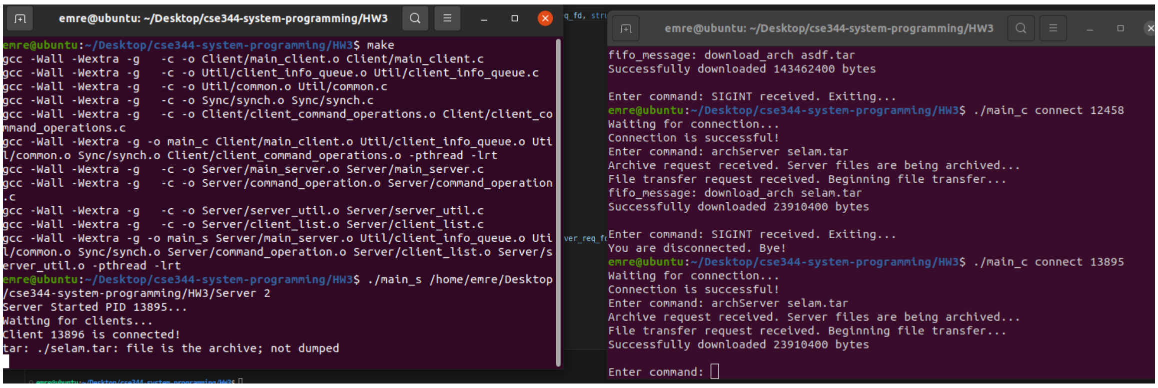


Fig-31: Archive Command

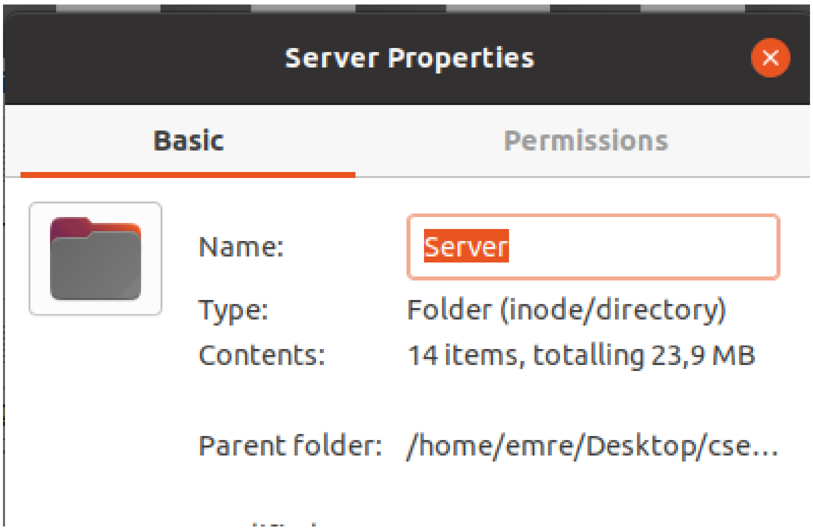


Fig-32: Size of Server Dir

If you try to test it, you will see that .tar file is located in client directory and there is no .tar file in server directory since it is deleted after downloading operation.

# 15 Help Command

There is nothing special in this command. Just prints the avaliable commands and if necessary specific command descriptions.

## 15.1 Help Test

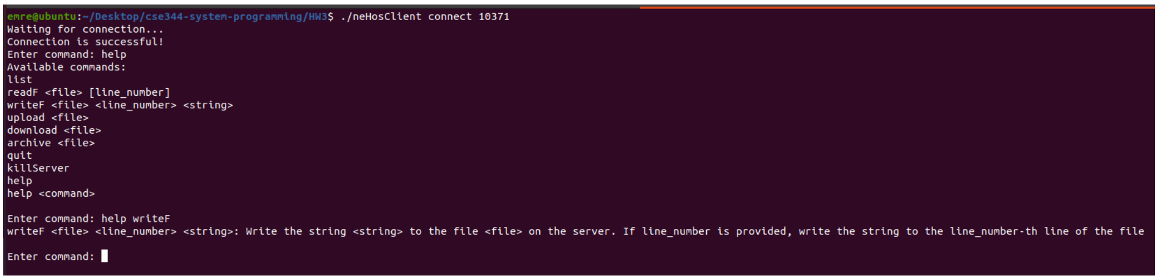


Fig-33: Example of Help Command

# 16 Handling CTRL-C Signals

When the server receives a CTRL-C signal, it terminates all currently active clients, **including those in the queue**, before shutting itself down.

The client, on the other hand, closes after sending information to the server.

## 16.1 CTRL-C Signal From Server

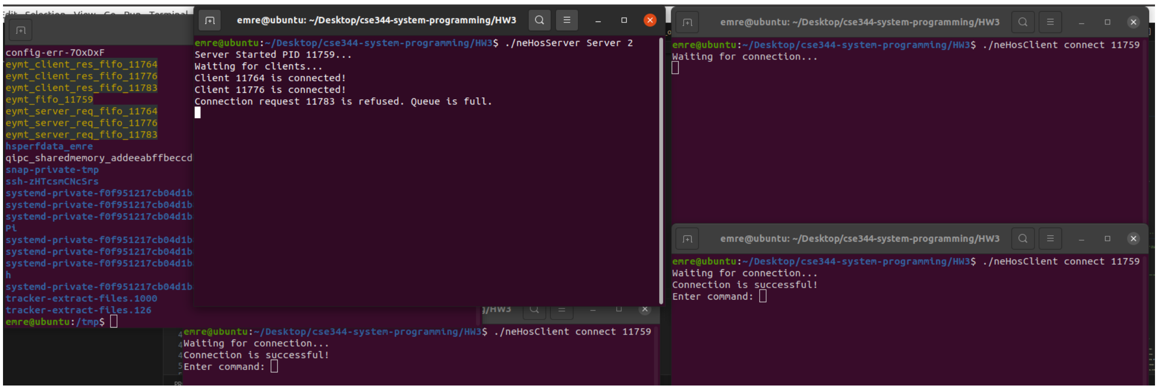


Fig-34: Before CTRL-C Command from Server

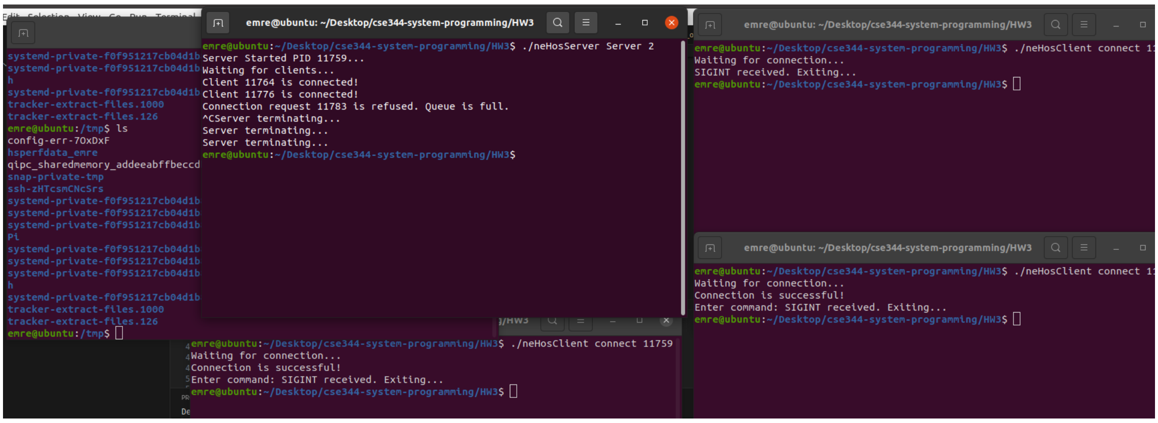


Fig-35: After CTRL-C Command from Server

As you can see, when the server generates a CTRL-C signal, all clients are terminated and no FIFOs remain open. You can verify this from the example.

## 16.2 CTRL-C Signal From Child

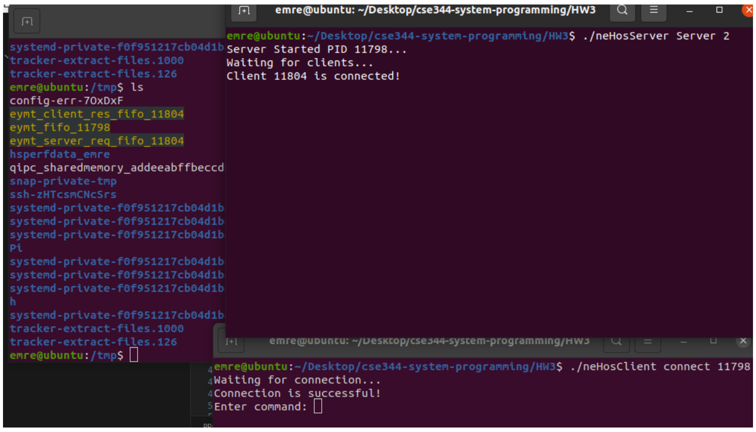


Fig-36: Before CTRL-C Command from Child

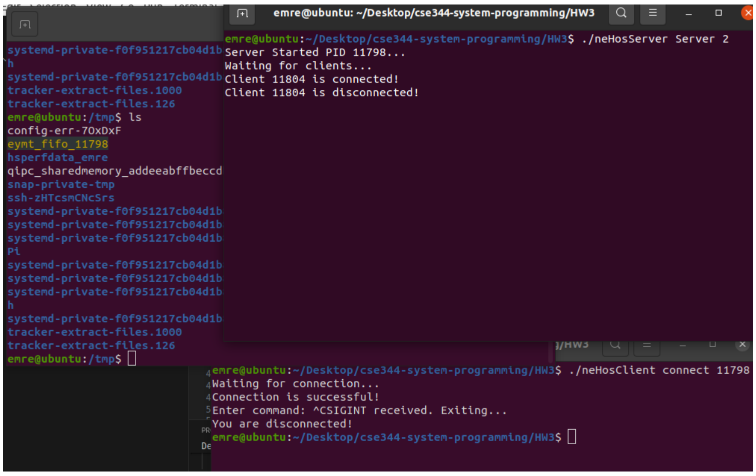


Fig-37: After CTRL-C Command from Child

As you can see in Fig-36 and Fig-37, the child has been successfully terminated. The FIFOs have been cleaned up, and the server has been informed of the client's termination.