

CSE 344 System Programming

Final Project

Kahraman Arda KILIÇ

1901042648

CONTENTS

1-Server-Client Communication

2-First synchronization of directories

3-Added/Removed/Edited Files Checking

4-Inner directories

5-Server

5a-Thread Pool

5b-Socket Creation

5c-Receiving Data

5d-File locks between threads

5e-SIGINT

6-Client

6a-Socket creation

6b-Receiving Data

6c-SIGINT

7-Test Cases

1- Server – Client Communication

This project aims to establish socket communication between a Server and Client to ensure data flow. The Server and Client sides need to transfer data to each other in a synchronized manner over the socket. In this project, a continuous flow of commands is maintained between the Server and Client. As shown in the example diagrams, the Server and Client continuously send specific commands to each other. Each command has a corresponding response on the other side. The use of `recv()` and `send()` is limited to once each for command transmission, thereby resolving synchronization issues between the sockets.

The messages sent by the Server or Client are stored in an array of size 1024. Within this array, there are three pieces of information. The `encoder()` method in `utils.h` is used to encode the message before calling `send()`. The receiving side uses the `decoder()` method, also found in the same file, to decode the received message and separate the information. The information contained in the message is as follows:

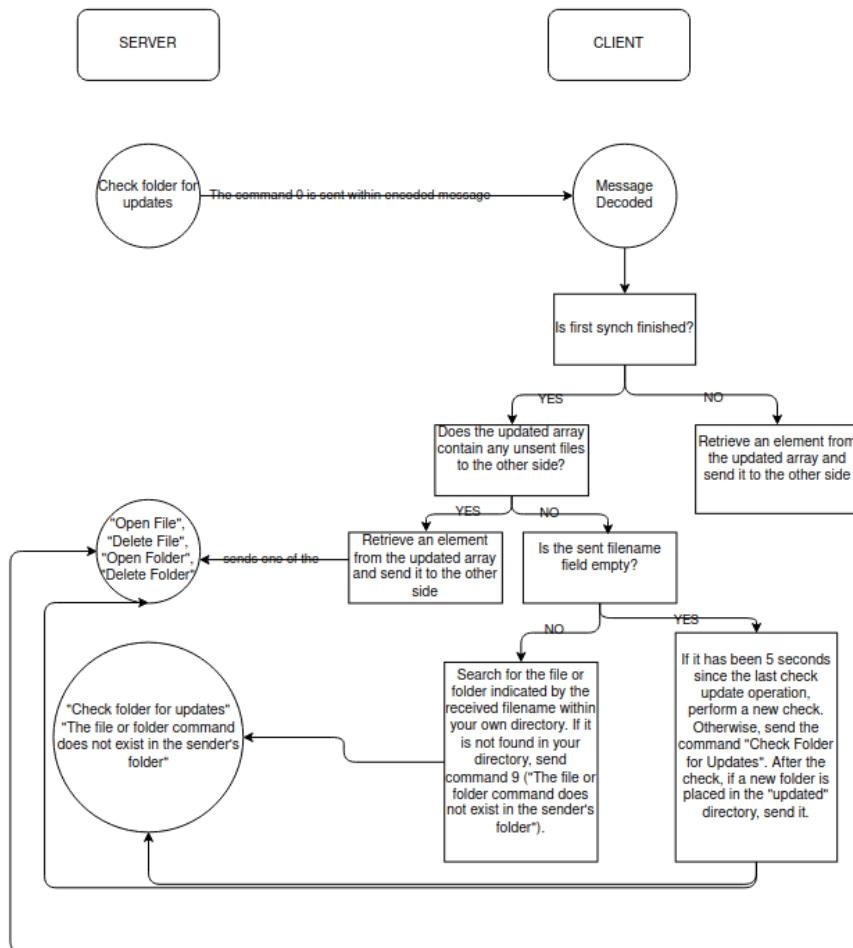
Command: This information determines what the other side wants. There are a total of 10 commands ranging from 0 to 9. They are: "Check folder for updates - 0", "Open the received file - 1", "Write the received message to the opened file - 2", "Close the opened file - 3", "Delete the received file name - 4", "Open the received folder - 5", "Delete the received folder and its contents - 6", "I have written the message you sent to my file. If there are still unread buffers in the file, continue sending them to me - 7", "I have opened the file you sent, open and start sending me the file you will send - 8", "The file or folder command does not exist in the sender's folder - 9".

Is Directory: It takes values 0 and 1. It determines whether the given filename in the Filename section is a folder. This allows the use of `mkdir()`. If it is a folder, it takes the value 1; otherwise, it takes the value 0.

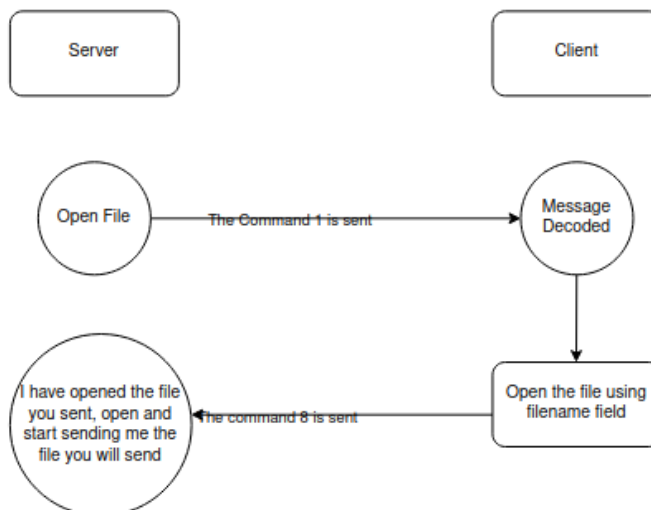
Message/Filename: This section contains the desired message to be sent (a buffer read from a file) or the filename. Both cannot be present at the same time.

A message of 1024 bytes is sent, consisting of 1 byte for the Command, 1 byte for Is Directory, and 1022 bytes for Message/Filename.

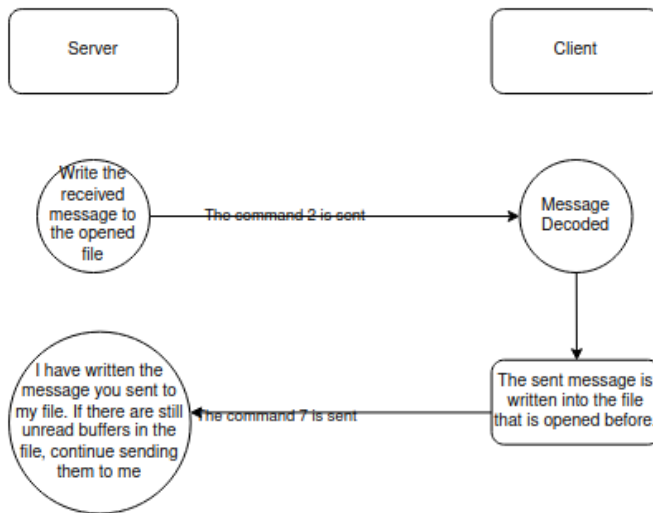
Command 0 – Check folder for updates



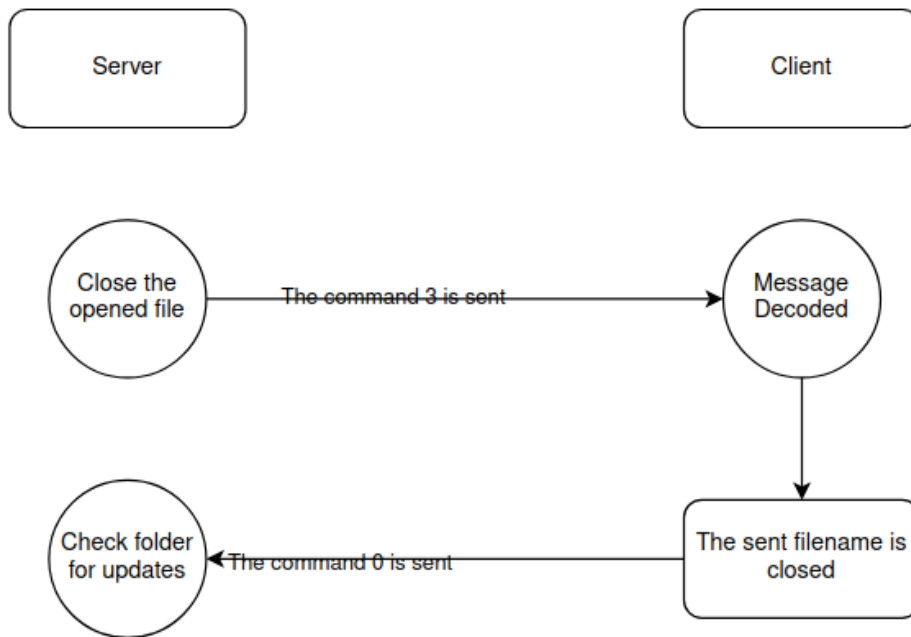
1- Open the received file



2- Write the received message to the opened file



3- Close the opened file



As depicted in the diagrams, the Server and Client continuously send commands to each other in a loop. The waiting period, which can be referred to as "waiting", occurs when the command 0 is exchanged between them. For example, in the diagram for Command 3, after receiving the command to close the file, the receiving side closes the opened file and sends a command to the other side to check their files. This ensures that they continuously send the command 0 to each other during times when there are no updates.

2- First Synchronization of directories

During the initial synchronization process, when the program is first executed, the current states of the folders should be communicated to each other. The primary operation I take during the first synchronization is the addition operation. If a file exists on one side but not on the other side, it should be added to that side. By sending the files that are not present in both sides during the initial execution of the program, synchronization is achieved. If there are two files with the same name but different contents, the contents of these files are shared with each other during the initial synchronization process. At this point, the content of the server takes priority.

3-Added/Removed/Edited Files Checking

There is an array called `currentFiles` to track the modified/added/deleted files within the folder. After each update check on the folder, the current state of the folder is added to the `currentFiles` array. However, any changes made to the folder during the program's runtime are not automatically added to this array. If another update check is to be performed, a comparison is made with the `currentFiles` array. If a file in the array is not present in the current state of the folder, it indicates a deletion. If a file not found in the `currentFiles` array is present in the current state of the folder, it indicates an addition. If a file with the same filepath exists in the `currentFiles` array and the current state of the folder, but their last modification time values differ, it indicates an edit operation. The identified files based on these operations are then added to the updated array.

4-Inner directories

In this program, the implementation of nested folders, referred to as Inner Directories, has been done. When we read the files in a folder using `readdir()` function, we can determine whether it is a folder or not using `IS_DIR()` function. The implementation of nested folders can be observed in the `traverseDirectory()` method in the `utils.h` file. If a file read by `readdir()` is a folder, `traverseDirectory()` is recursively called. This allows accessing all the nested files and folders. During the deletion process of a folder, the deletion is also performed recursively, followed by using `rmdir()` function to delete the desired folder.

5-Server

a-Thread Pool

On the server side, there is an array called `pthread_t *threads`. For each new client that connects, a thread is added to this array. The thread pool size is obtained as an argument from the console. If there is no space available in the threads array for a connected client, their connection is terminated. Each thread is specific to a client. If a client disconnects, the corresponding thread should be removed from the thread pool. Instead of removing threads from the thread pool after each client disconnects, the program checks the threads in the thread pool when a new client attempts to connect. If any of the threads have exited using `pthread_exit()`, the space of that thread is cleared, and all the remaining threads in the thread pool are shifted once to create an available space. This way, the newly connected client can start using its dedicated thread.

The completion of the tasks of the threads in the thread pool can be determined using `pthread_join()`. However, if the 5th thread in the threads array has exited, it requires waiting for all the threads in the threads array using `pthread_join()`. Therefore, in my research, I found `pthread_tryjoin_np()`. This method allows checking if a thread is still running, and if it is, the waiting process is not performed. This way, I was able to rearrange the thread pool by performing the check once.

`pthread_join()` is also used for the threads. After receiving a SIGINT signal, it waits for all active threads to finish.

`pthread_tryjoin_np()` is used in `cleanThreadPool()`, and `pthread_join()` is used in `sigIntHandler()`.

b-Socket Creation

On the server side, when opening the socket, `INADDR_ANY` is used. The port value is obtained from the user. The socket connection is established with the client by calling `bind()`, `listen()`, and `accept()` in sequence. When a client connects, the received `client_socket` value is passed as an argument to the thread created using `pthread_create()`. Inside the thread function, this socket value is used for `recv()` and `send()` operations. In this project, one `recv()` and one `send()` are used within the thread function. Since the messages are encoded, there is no need to perform multiple `recv()` and `send()` operations. This eliminates potential asynchronous situations.

On the server side, first, `recv()` is performed to receive a command from the client. This command is processed, and the necessary response is stored in the `server_message` buffer. Then, this buffer is sent using `send()`. During socket communication, 1024-byte buffers are used.

c-Receiving Data

After establishing the socket connection and creating the thread on the server side, there is a waiting process in the thread function during `recv()`. It waits until a message is received from the client. When the client sends a message, it is received using `recv()`. Then, the received encoded message is decoded by calling the `decoder()` method. Based on the command value, the corresponding operation is performed (for example, if the command in the received message is 3, an already open file is closed). After the operation, the desired message to be sent is placed in the `server_message` array using the `encoder()` function, and then it is sent using `send()` as a 1024-byte message.

d-File locks between threads

In the initial stages of the project, I considered solving the file locking issue using `fcntl()` and `flock()`. However, in my further research, I learned that these two approaches are process-based. Any lock applied with `fcntl()` is shared among threads, which means I had to choose between compromising the concurrent nature of threads or allowing two different threads to write to the same file simultaneously. The second option would result in data corruption, so I decided to implement the first approach despite its lower efficiency.

When a "FILE OPEN (1)" command is received from the client, I used a mutex to create a lock specific to that thread. The thread holds this mutex lock until a "FILE CLOSE (3)" command is received from the client. As a result, only one thread can perform a write operation on any file within the server at a given time. Although I am aware that this approach is less efficient and impacts concurrency, I chose it to prevent data loss.

During my research on how to address this issue, I came across articles suggesting the use of an intermediate thread layer for file reading/writing. However, I decided not to implement this approach to avoid opening too many threads from the thread pool.

```

/* If the message is OPEN FILE */
else if(strcmp(parsedCommand[0],"1") == 0){
    char combinedPath[MAX_FILENAME];
    snprintf(combinedPath, MAX_FILENAME, "%s/%s", directoryPath, parsedCommand[2]);
    fd = open(combinedPath, O_WRONLY | O_CREAT | O_TRUNC, 0666);
    if (fd == -1) {
        perror("Failed to open file");
        exit(-1);
    }
    pthread_mutex_lock(&mutexFileLock); //MUTEX LOCKED
    char * encodedMessage = encoder('8','0',parsedCommand[2]);
    strcpy(server_message,encodedMessage);
    free(encodedMessage);
    strcpy(filePath, parsedCommand[2]);
}

/* If the message is WRITE INTO FILE */
else if(strcmp(parsedCommand[0],"2") == 0){
    ssize_t bytesWritten = write(fd, parsedCommand[2], strlen(parsedCommand[2]));
    if (bytesWritten == -1) {
        perror("Error writing to the file");
        close(fd);
        //return 1;
    }
    char * encodedMessage = encoder('7','0',filePath);
    strcpy(server_message,encodedMessage);
    free(encodedMessage);
}

/* If the message is CLOSE FILE */
else if(strcmp(parsedCommand[0],"3") == 0){
    close(fd);
    pthread_mutex_unlock(&mutexFileLock); // MUTEX UNLOCKED
    getCurrentFilesFromDirectory(directoryPath, &currentFiles);
    modifyFileNames(currentFiles,directoryPath);
    char * encodedMessage = encoder('0','0',"");
    strcpy(server_message,encodedMessage);
    free(encodedMessage);
}
}

```

e-SIGINT

When a SIGINT signal is received in the server, the following steps should be taken:

- 4- Close all threads in the thread pool using pthread_exit().
- 5- Remove previously closed threads from the thread pool.
- 6- Ensure that all threads created in the thread pool are closed using pthread_join().
- 7- Free the dynamically allocated memory for the thread pool array using free().
- 8- Free the dynamically allocated memory for the directory path array using free().
- 9- Destroy any used mutexes using pthread_mutex_destroy().
- 10- Close the server socket using close().
- 11- Exit the program.

```

void sigIntHandler(int signum) {
    pthread_mutex_lock(&mutexsigIntReceived);
    sigIntReceived = 1;
    cleanThreadPool();
    pthread_mutex_unlock(&mutexsigIntReceived);
    for (int i = 0; i < currentThreadIndex; i++) {
        pthread_join(threads[i], NULL);
        printf("Thread %d joined\n", i);
    }
    free(threads);
    free(directoryPath);
    pthread_mutex_destroy(&mutexsigIntReceived);
    pthread_mutex_destroy(&mutexFileLock);
    close(server_socket);
    exit(-1);
}

```

The sigIntReceived flag is set to 1 within the SIGINT signal. This flag is used to check if a SIGINT signal was received within the threads.

```

// If SIGINT received, then terminate it.
pthread_mutex_lock(&mutexsigIntReceived);
if(sigIntReceived){
    pthread_mutex_unlock(&mutexsigIntReceived);
    close(client_socket);
    free2DArray(parsedCommand);
    freeFileList(updated);
    freeFileList(currentFiles);
    freeFileList(firstStartCurrentFiles);
    pthread_exit(NULL);
}
pthread_mutex_unlock(&mutexsigIntReceived);

```

6-Client

a-Socket creation

Creating a socket on the client side is indeed easier compared to the server side. The only difference is that on the server side, the IP address is typically set to `INADDR_ANY`, while on the client side, an optional server IP address can be provided. If no IP address is specified, the client will connect to the server using the localhost. If a server IP address is provided, the connection will be established using the given IP address and port. In tests conducted on two different devices, a 192.168.x.x formatted IP address was used, making it mandatory to provide such an IP address on the client side.

b-Receiving Data

The client side is always responsible for initiating communication with the server. On the client side, there are typically 2 `send()` operations and 1 `recv()` operation. Before entering an infinite `while()` loop, as soon as the connection is established, the client sends a message to the server. The server then receives this message, performs the necessary operations, and sends a response back. The client continues the communication with the server using the `recv()` and `send()` functions within the while loop.

The received data on the client side is also stored in a buffer of size 1024, similar to the server side. This encoded message is then decoded, and the appropriate actions are taken based on the given command.

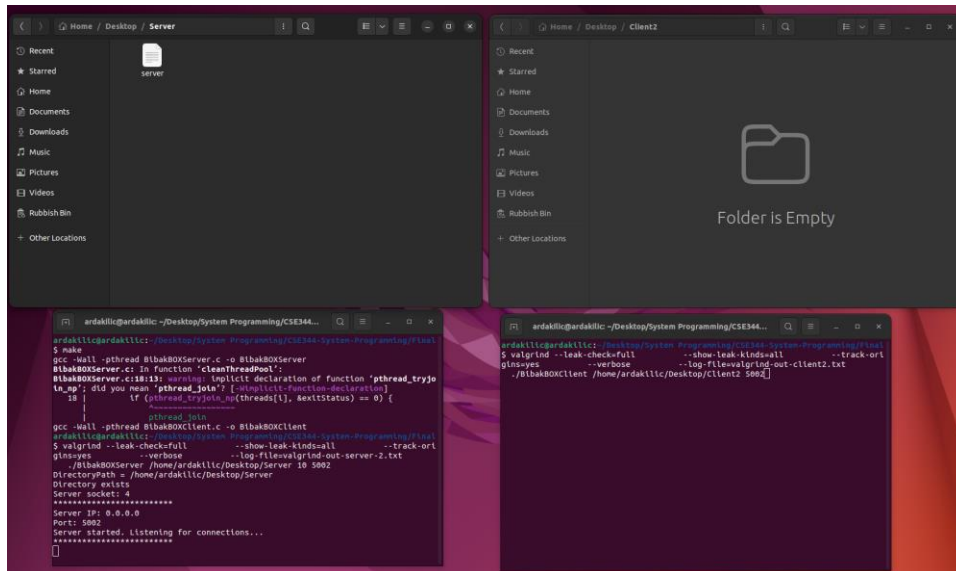
c-SIGINT

```
void sigIntHandler(int signum) {
    fprintf(logFile, "SIGINT Received\n");
    free(directoryPath);
    free(serverAddress);
    fprintf(logFile, "Dynamically allocated char pointers are freed.\n");
    freeFileList(updated);
    freeFileList(currentFiles);
    freeFileList(firstStartCurrentFiles);
    fprintf(logFile, "Dynamically allocated struct FileNode* (Linked Lists) are freed.\n");
    free2DArray(parsedCommand);
    fprintf(logFile, "Dynamically allocated 2D Char array (parsedCommand) is freed.\n");
    close(client_socket);
    fprintf(logFile, "Client socket is closed\n");
    fclose(logFile);
    exit(EXIT_FAILURE);
}
```

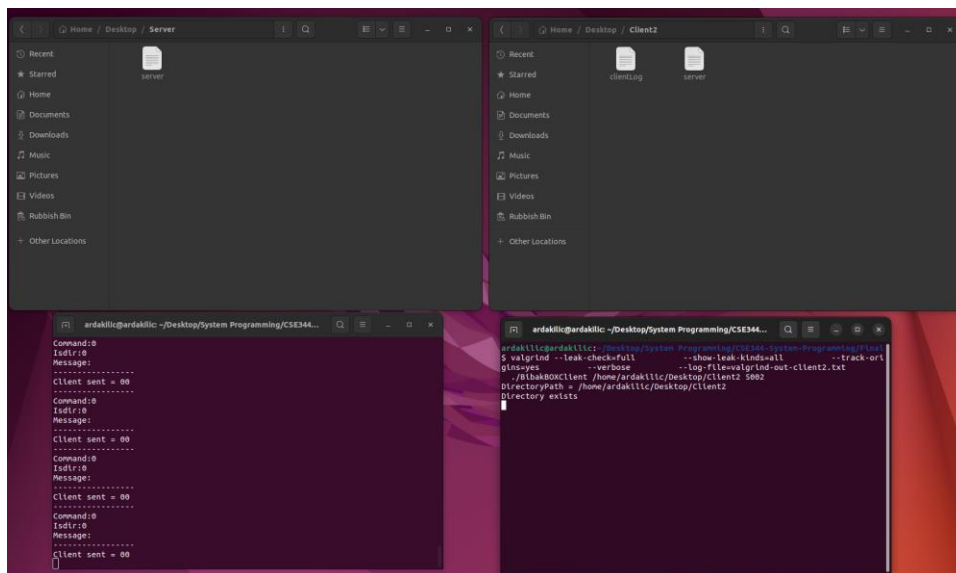
In the client side, since there is no thread, there is no need for `pthread_join()` in the SIGINT handler. The pointers used during program execution in the client side, which were allocated using `malloc()`, have been freed. Then, the `client_socket` is closed. The log file, if opened, is closed, and finally, the program exits using `exit()`.

7-Test Cases

a. Initial synchronization on startup

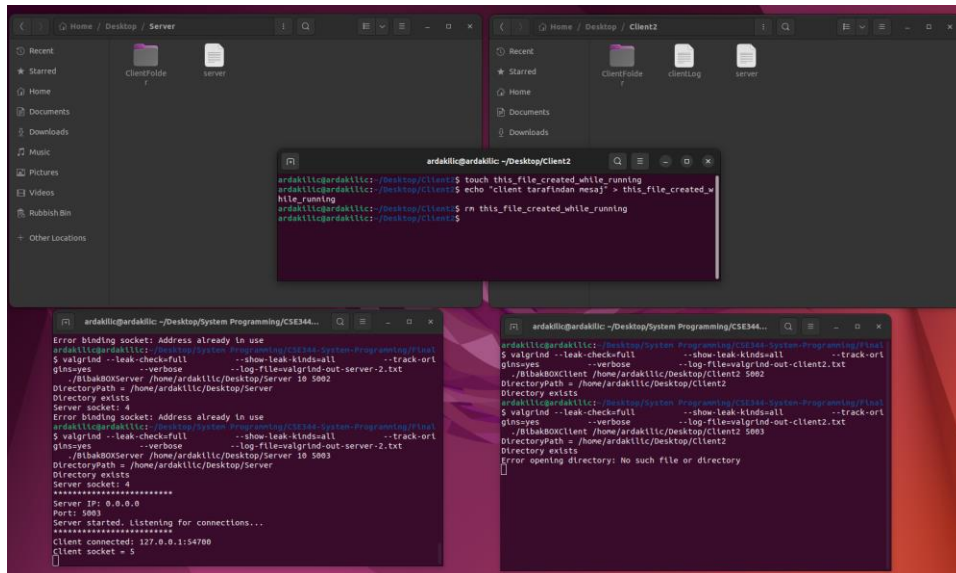


Before starting the client program, the folders are like this.

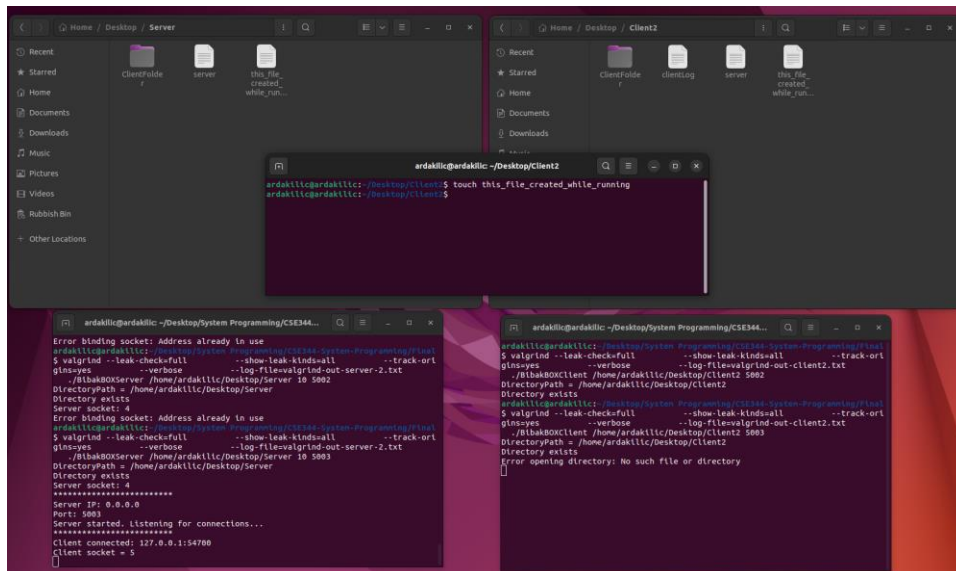


The server sent "server" file to client. The content of the file is copied to the client folder

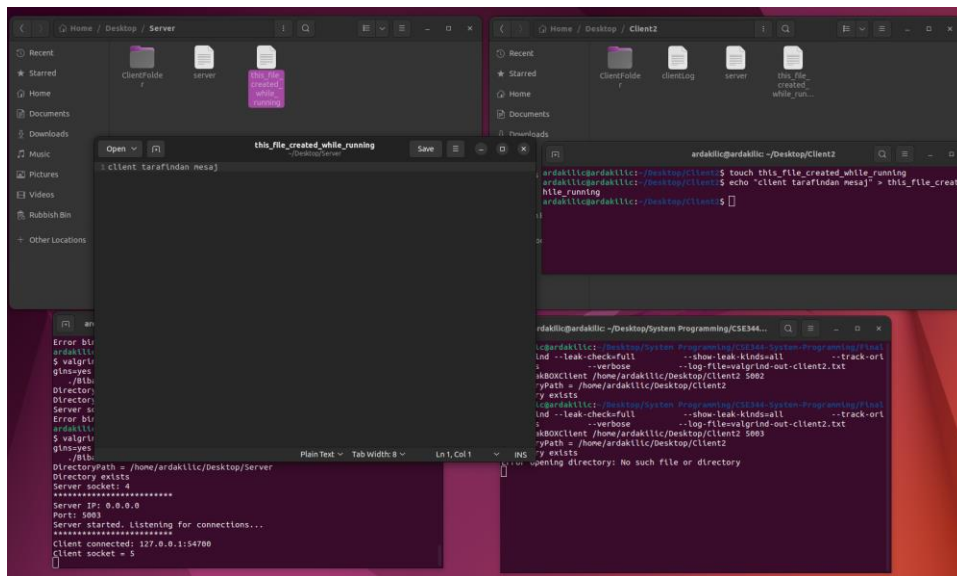
b. Deleting a file in the client folder



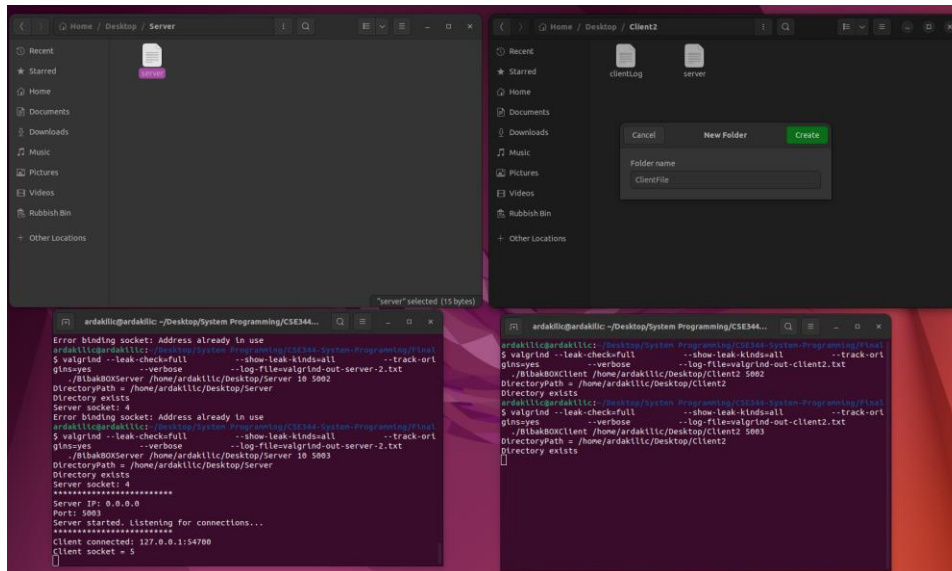
c. Creating a file in the client folder



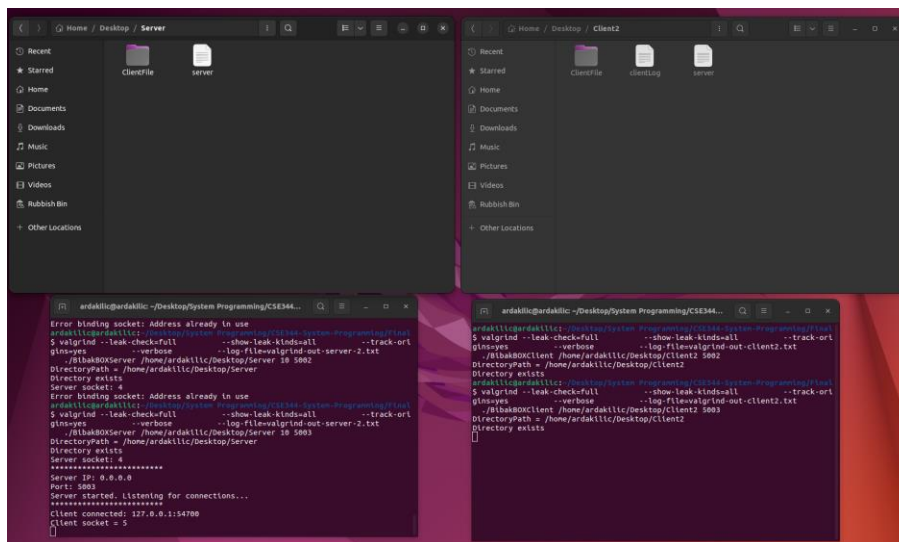
d. Editing a file in the client folder



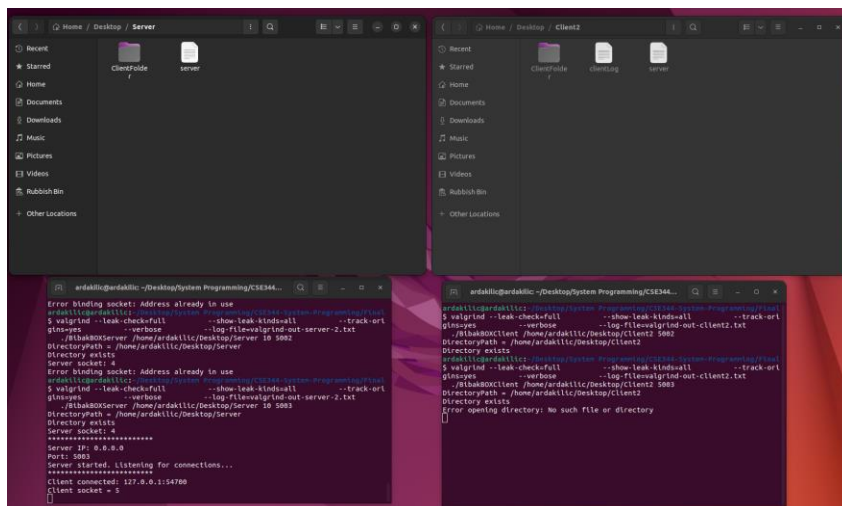
e. Creating a folder in the client folder



Creating a folder in Client while program are running.

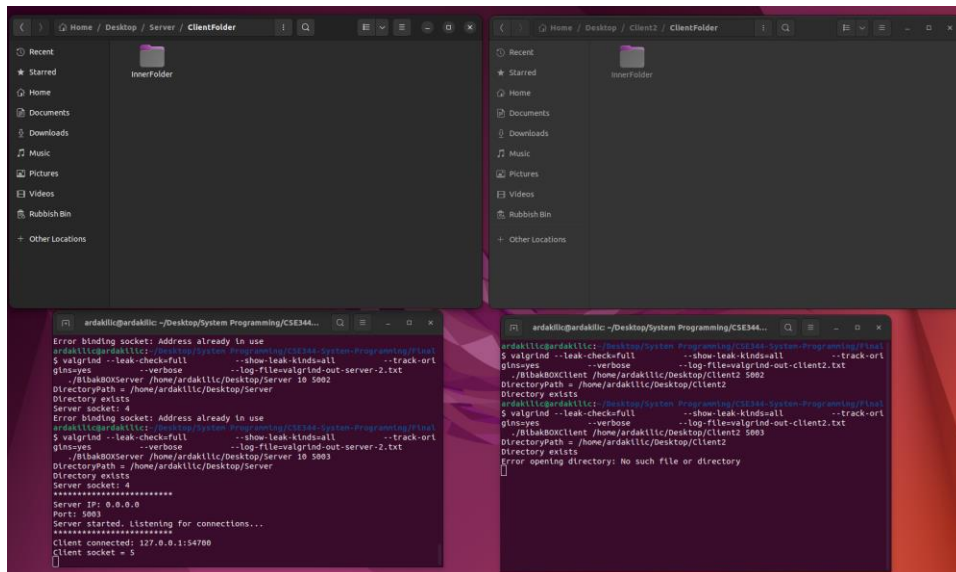


Folder is sent to server.



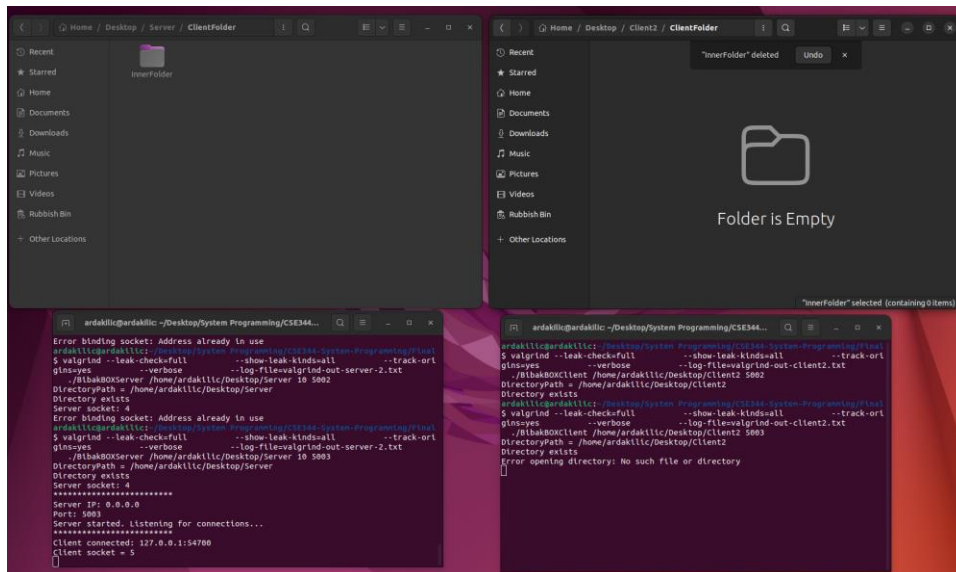
Renaming the folder in client side. Changes are applied to server.

f. Creating an inner directory in the client folder

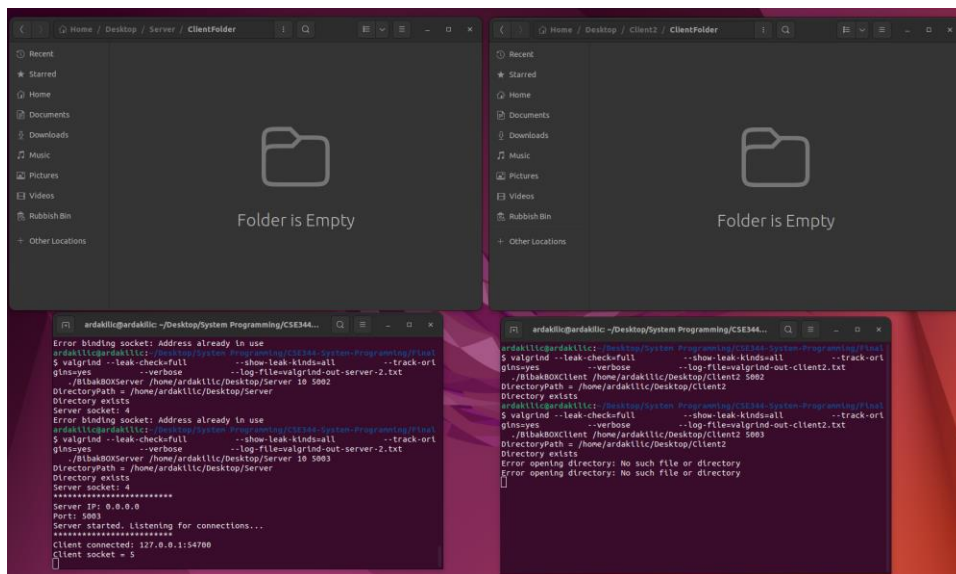


Client created a new folder in ClientFolder directory. Server also created InnerFolder in its /home/ardakili/Desktop/Server/ClientFolder

g. Deleting an inner directory in the client folder

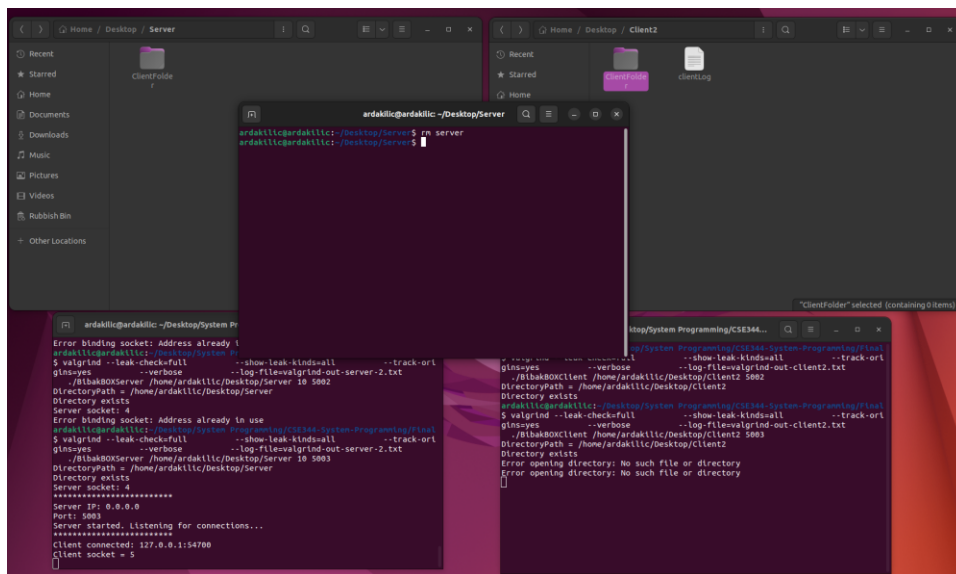


The directory is removed in Client folder, immediately screenshot is taken before server removes its folder.

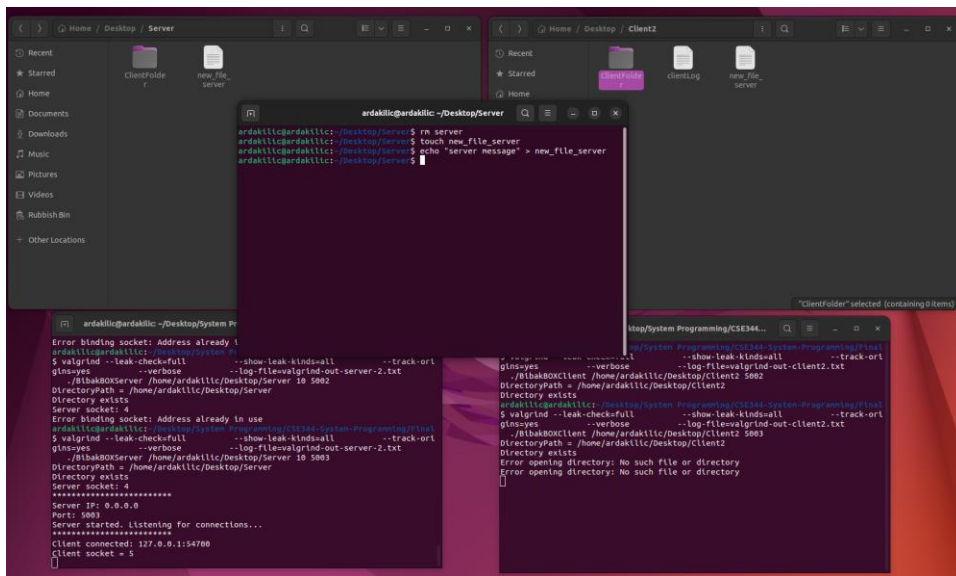


The server is removed the inner folder

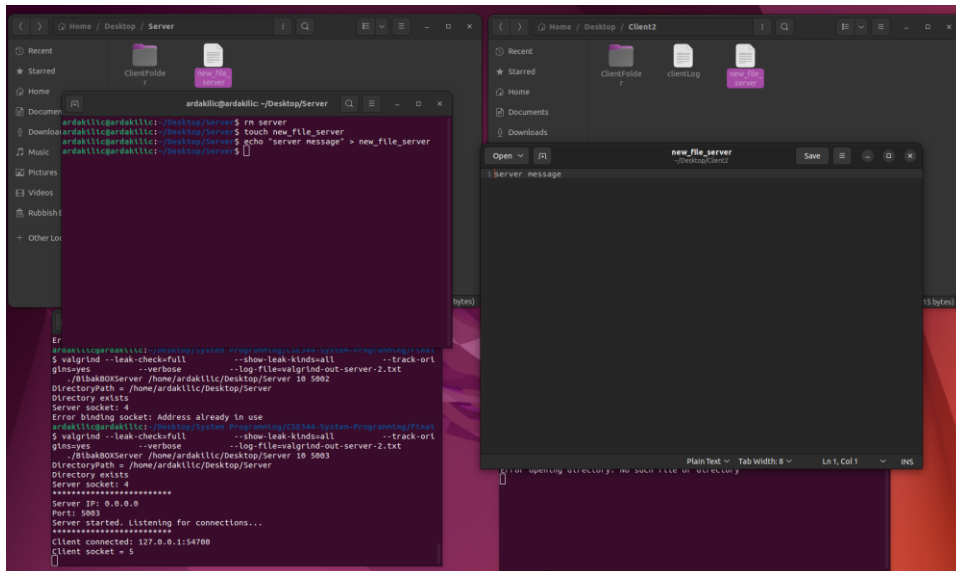
h. Deleting a file in the server folder



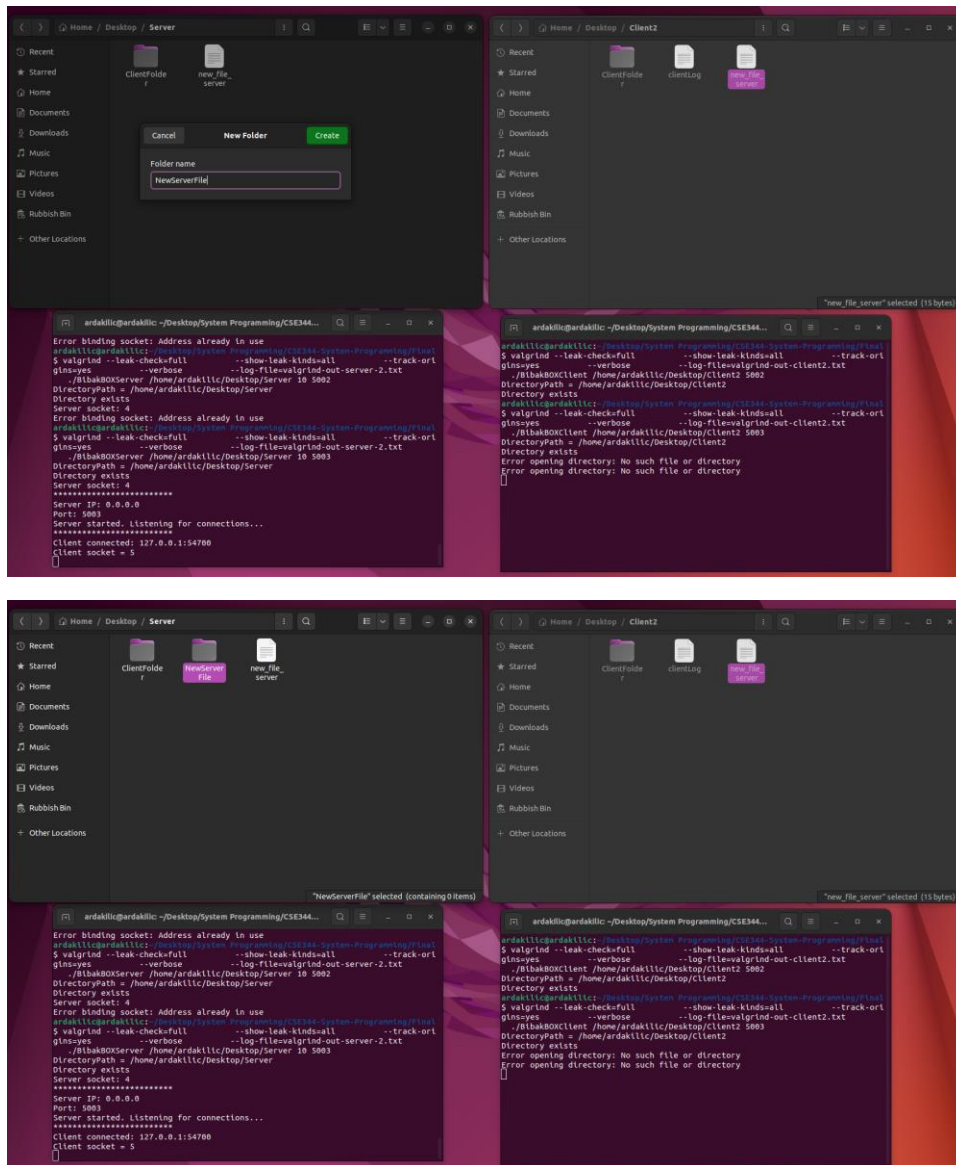
i. Creating a file in the server folder

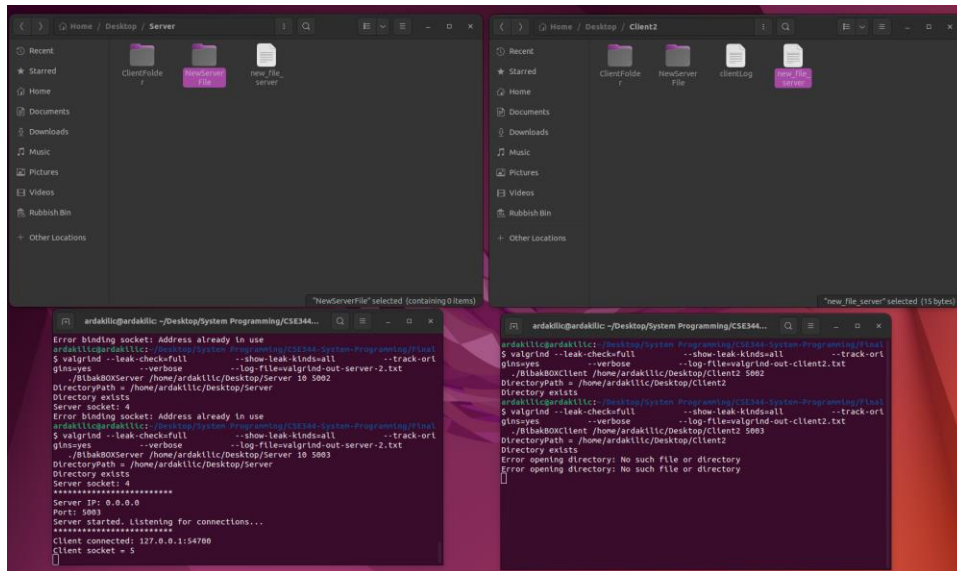


j. Editing a file in the server folder

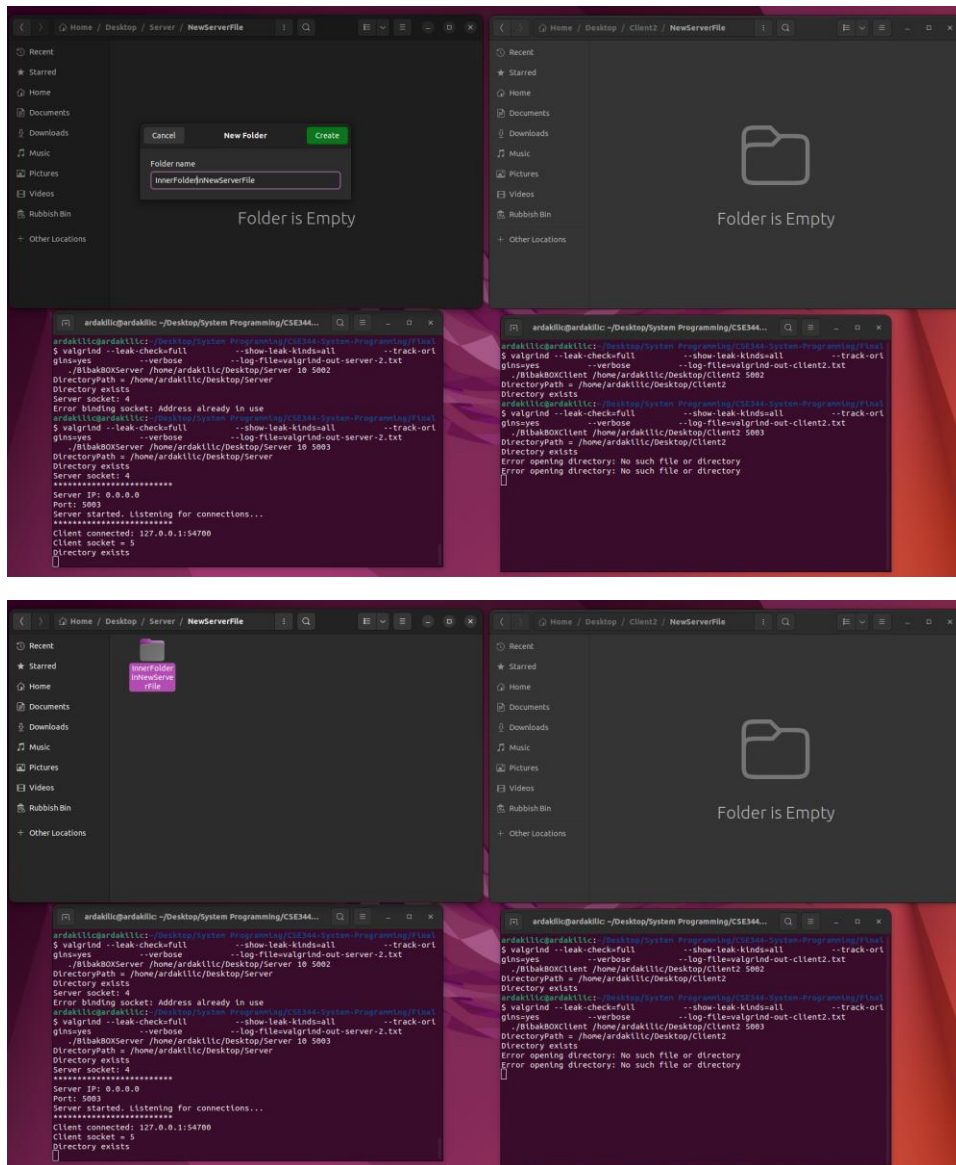


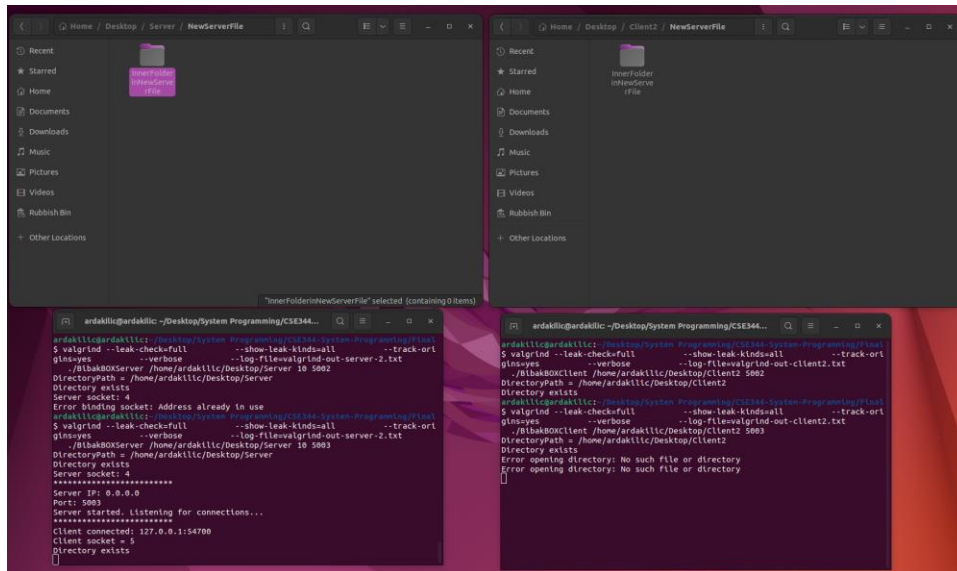
k. Creating a folder in the server folder



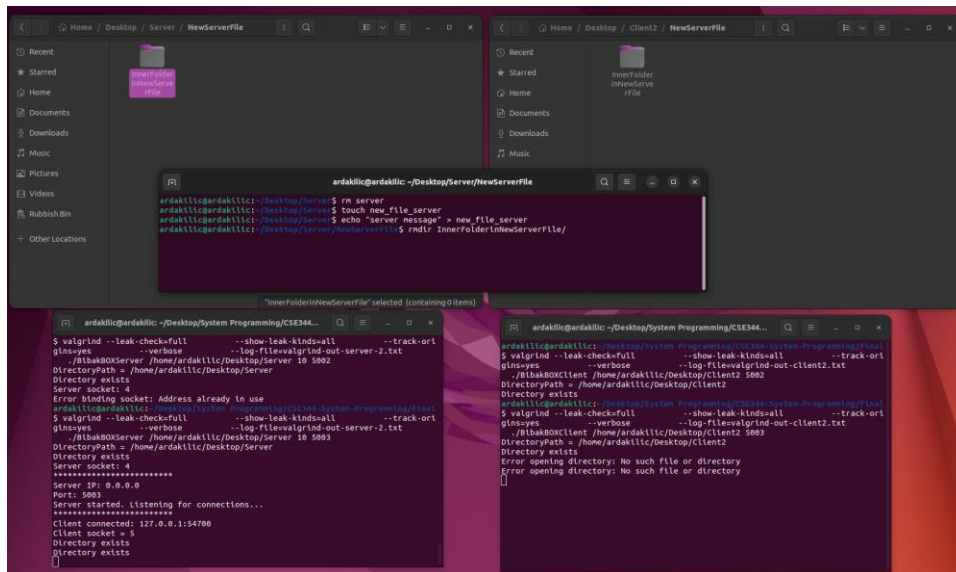


I. Creating an inner directory in the server folder





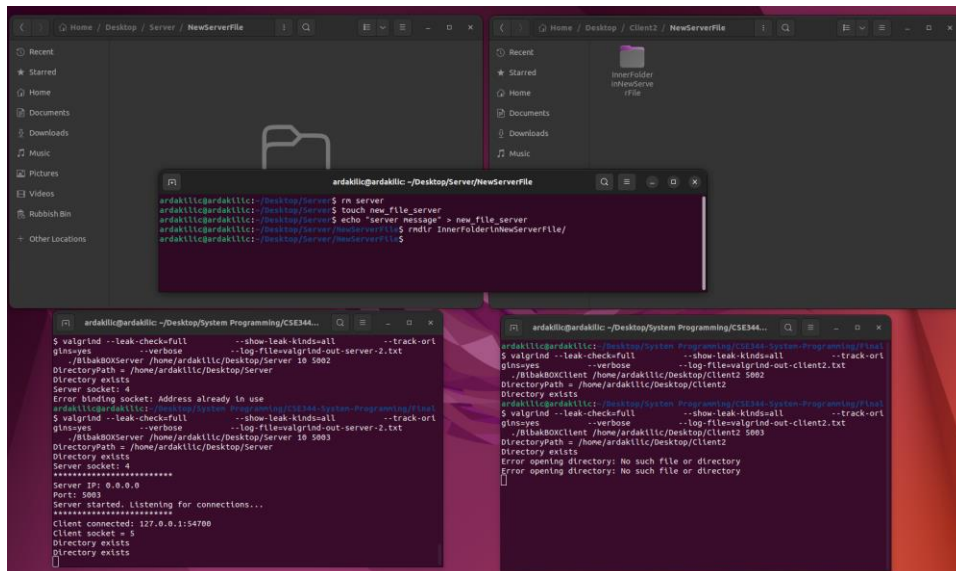
m. Deleting an inner directory in the server folder



The screenshot shows a terminal window with the following commands and output:

```
ardaklilc@ardaklilc:~/Desktop/Server/NewServerFile$ rm server
ardaklilc@ardaklilc:~/Desktop/Server/NewServerFile$ touch new_file_server
ardaklilc@ardaklilc:~/Desktop/Server/NewServerFile$ echo "Server message" > new_file_server
ardaklilc@ardaklilc:~/Desktop/Server/NewServerFile$ rmdir innerFolderInNewServerFile/
```

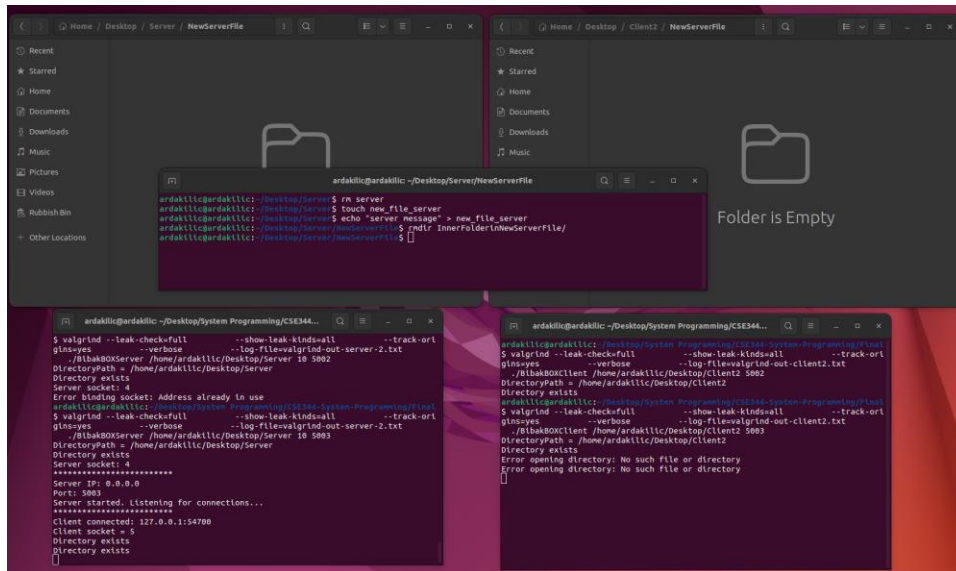
The output of the `rmdir` command is: `rmdir: failed to remove 'innerFolderInNewServerFile/': Directory not empty`.



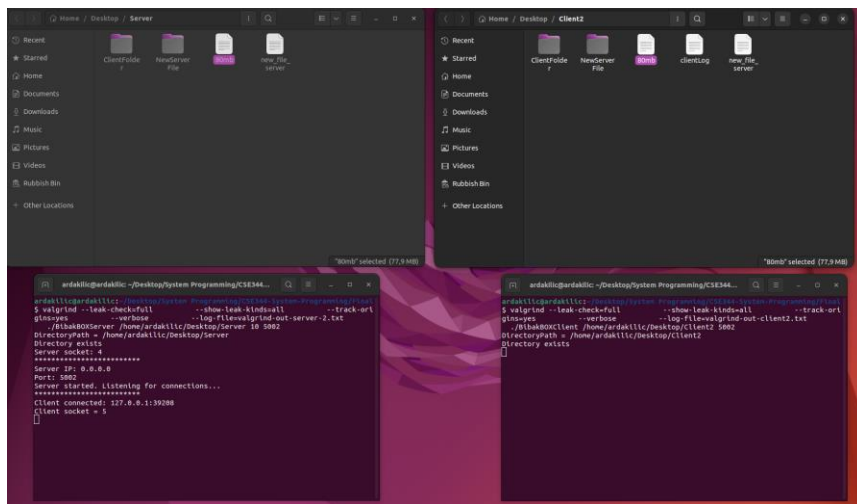
The screenshot shows a terminal window with the following commands and output:

```
ardaklilc@ardaklilc:~/Desktop/Server/NewServerFile$ rm server
ardaklilc@ardaklilc:~/Desktop/Server/NewServerFile$ touch new_file_server
ardaklilc@ardaklilc:~/Desktop/Server/NewServerFile$ echo "Server message" > new_file_server
ardaklilc@ardaklilc:~/Desktop/Server/NewServerFile$ rmdir innerFolderInNewServerFile/
```

The output of the `rmdir` command is: `rmdir: failed to remove 'innerFolderInNewServerFile/': Directory not empty`.

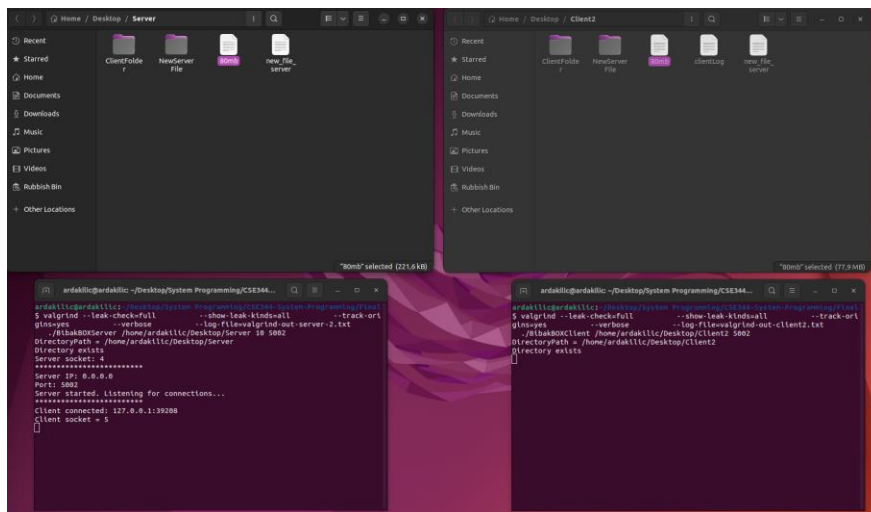


n. Large file transfer (80MB) from server to client

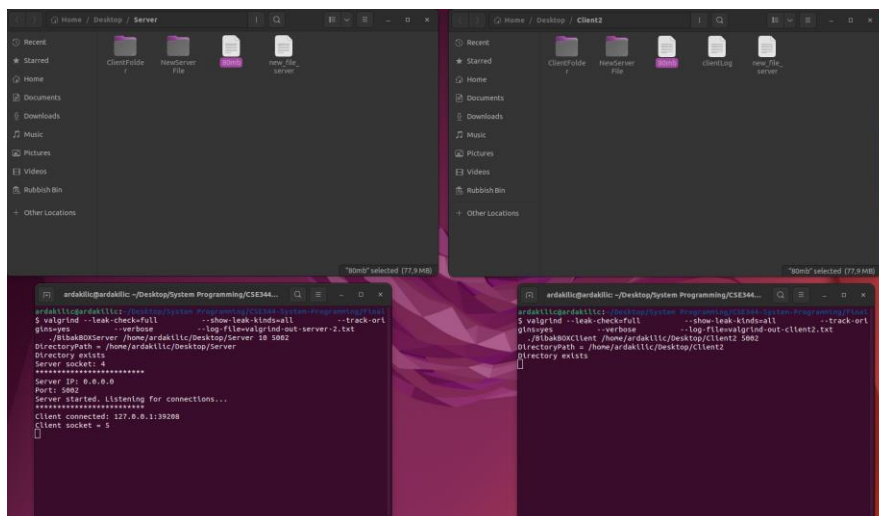


39.97 seconds for sending 77.9MB file from Server to Client

o. Large file transfer (80MB) from client to server

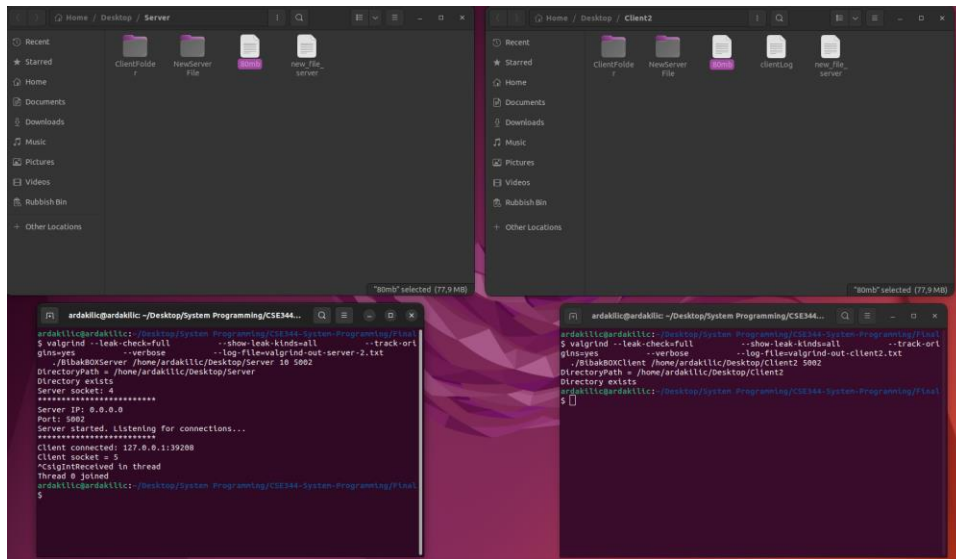


During file transfer



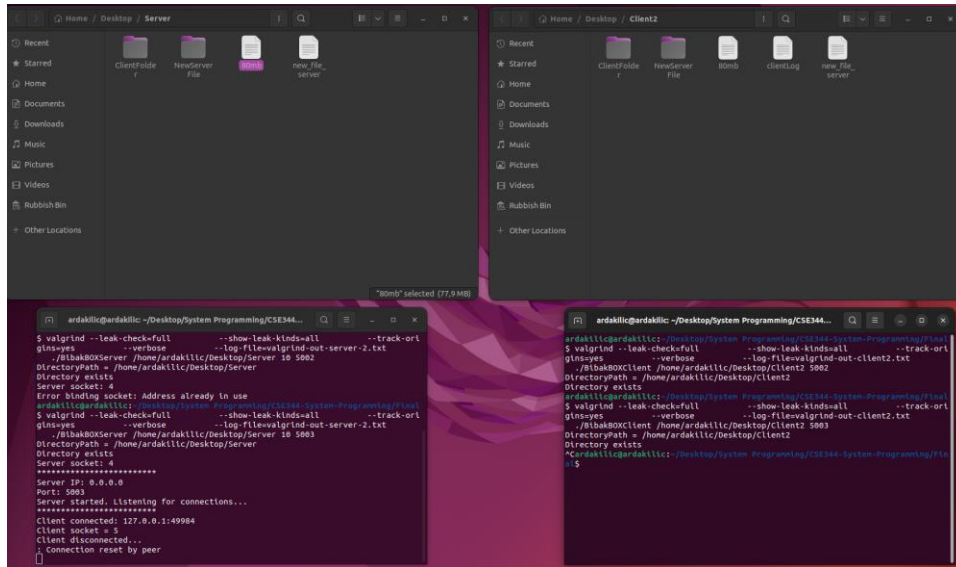
File transfer is completed within 38.25 seconds

p. Sending a SIGINT signal to the server



```
2829425 [SEND]: Check your folder for updates
2829426 The message sent from server is decoded into parsedCommand array.
2829427 [SEND]: Check your folder for updates
2829428 The message sent from server is decoded into parsedCommand array.
2829429 [SEND]: Check your folder for updates
2829430 The message sent from server is decoded into parsedCommand array.
2829431 [SEND]: Check your folder for updates
2829432 The message sent from server is decoded into parsedCommand array.
2829433 [SEND]: Check your folder for updates
2829434 The message sent from server is decoded into parsedCommand array.
2829435 [SEND]: Check your folder for updates
2829436 The message sent from server is decoded into parsedCommand array.
2829437 [SEND]: Check your folder for updates
2829438 The message sent from server is decoded into parsedCommand array.
2829439 [SEND]: Check your folder for updates
2829440 The message sent from server is decoded into parsedCommand array.
2829441 [SEND]: Check your folder for updates
2829442 The message sent from server is decoded into parsedCommand array.
2829443 [SEND]: Check your folder for updates
2829444 The message sent from server is decoded into parsedCommand array.
2829445 [SEND]: Check your folder for updates
2829446 The message sent from server is decoded into parsedCommand array.
2829447 [SEND]: Check your folder for updates
2829448 The message sent from server is decoded into parsedCommand array.
2829449 [SEND]: Check your folder for updates
2829450 Server closed the connection.
2829451
```

q. Sending a SIGINT signal to the client

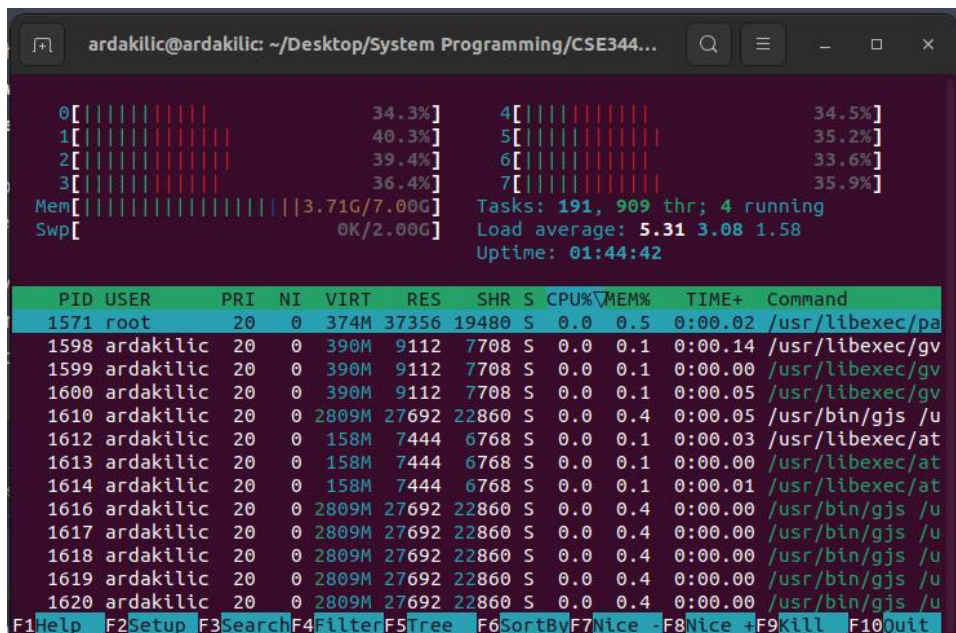


```
47034 Writing into the file...
47035 [SEND]: The sent buffer is written into file. Server can send more data
47036 The message sent from server is decoded into parsedCommand array.
47037 Writing into the file...
47038 [SEND]: The sent buffer is written into file. Server can send more data
47039 The message sent from server is decoded into parsedCommand array.
47040 Writing into the file...
47041 [SEND]: The sent buffer is written into file. Server can send more data
47042 The message sent from server is decoded into parsedCommand array.
47043 Writing into the file...
47044 [SEND]: The sent buffer is written into file. Server can send more data
47045 SIGINT Received
47046 Dynamically allocated char pointers are freed.
47047 Dynamically allocated struct FileNode* (Linked Lists) are freed.
47048 Dynamically allocated 2D Char array (parsedCommand) is freed.
47049 Client socket is closed
47050
```


r. Connecting 20 clients to the server

20 clients connected to the server with different file names. At the end of the process, synchronization was achieved among all clients and the server's files.

When running 20 clients, the CPU usage was monitored using "htop". In this project, sleep was not used. Instead, to reduce memory usage, the files were checked every 5 seconds. The elapsed time was measured using gettimeofday().



s. Testing file transfer between different computers

A test was conducted by running a server and client on two different computers. The IP address of the server side was determined using "ifconfig" command, and it was passed as an argument to the Server Address when the client connected.

The socket connection between the two computers is similar to the connection on localhost. The only difference is that when the client connects, `sin_addr.s_addr = inet_addr(Server IP Address)` is used. For localhost connection, this value is `sin_addr.s_addr = inet_addr("127.0.0.1")`.

Data sharing between computers is much slower compared to localhost. This was observed during the testing process.

The `directoryPath` values have been modified to be relative, allowing the server or client to be opened with any file path. This ensures that each computer can run the program with its own specific directory path.

