CSE344 Midterm
Concurrent File Access System

Eren ÇAKAR
1901042656

May 4, 2024

1 Introduction

The concurrent file access system described in this report addresses the need for a robust file server that enables multiple clients to connect, access, modify, and archive files in a specific directory located on the server side. This system is designed to facilitate seamless communication and file management between clients and the server, while ensuring data consistency, integrity, and concurrency control.

1.1 Objective

The primary objective of this project is to design and implement a file server capable of handling concurrent access from multiple clients. The server should enforce mutual exclusion to prevent race conditions, ensure data consistency and integrity, and support various file formats and sizes.

1.2 Scope

The scope of this project includes the design and implementation of both server-side and client-side programs. The server program is responsible for managing client connections, handling file operations, enforcing synchronization, and archiving files. On the other hand, the client program allows users to connect to the server, perform file operations, and interact with the server's file system. **The system was designed with the scope of server directory operating with no subfolders/subdirectories in it for preventing path complexities.**

The following sections provide a comprehensive overview of the concurrent file access system, including its architecture, implementation details, testing methodology, and conclusions.

2 System Architecture

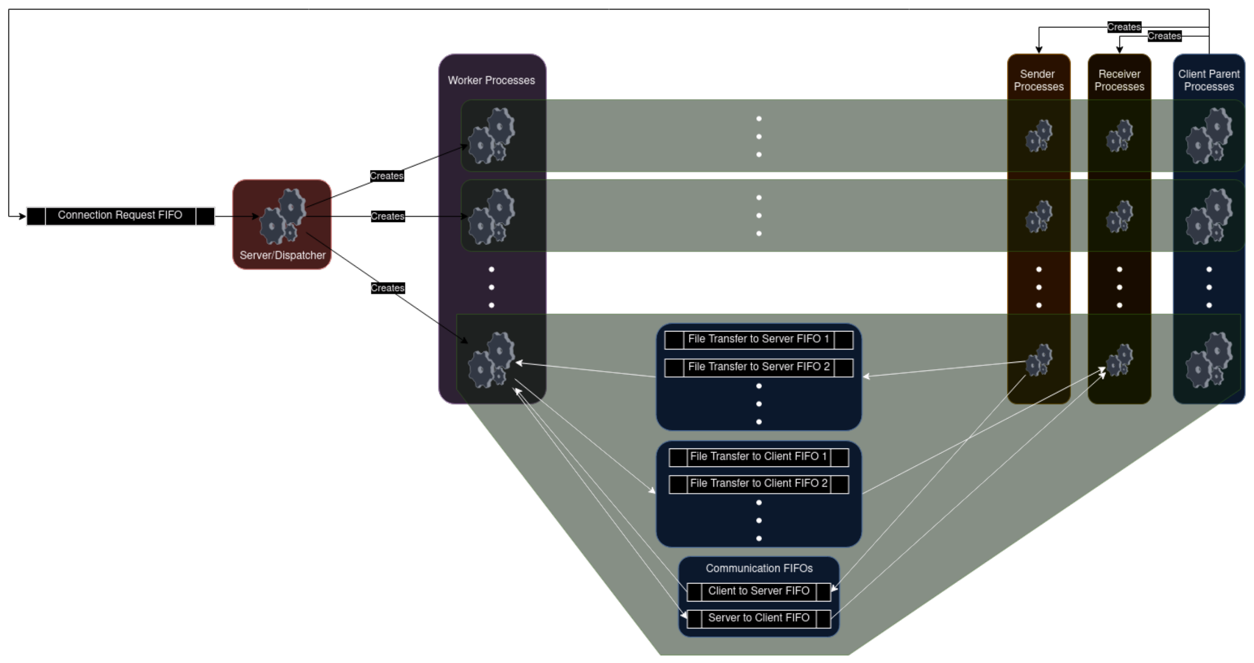


Figure 1: System Architecture

The concurrent file access system consists of several types of processes responsible for managing client-server communication and file operations. The architecture includes:

2.1 Server/Dispatcher Process

The server/dispatcher process is responsible for managing client connections and dispatching worker processes to handle client requests. It creates a FIFO (named pipe) and waits for connection requests from clients. Upon receiving a request, the dispatcher checks if there's enough room to create a worker process. If space is available, it dispatches a worker process to serve the client; otherwise, it creates a special worker process called a decliner to decline the client or queues the client, depending on the specified behavior in the request.

2.2 Worker Process

Worker processes handle client requests by reading messages from the client, executing commands, and returning appropriate responses. Each client is assigned a dedicated worker process. Communication between the worker process and the client is facilitated through two FIFOs: one for messages from the client to the server and the other for responses from the server to the client. This separation of communication direction enables clients to queue commands and helps prevent FIFO data corruption. Also the logging of client and server communications is done by respective worker processes just after each sending and receiving of messages in either direction.

Additionally, a special type of worker process, called a decliner, is responsible for gracefully declining client connections by instructing them to terminate and closing the FIFOs.

2.3 Client Parent Process

The client parent process is responsible for creating client sender and receiver processes, enabling independent sending of commands and receiving of responses. This design prevents the server from blocking client command transmission.

2.4 Client Sender Process

The client sender process reads user input from the terminal and sends commands to the server (worker process) through the FIFO dedicated to client-server communication. Additionally, it manages the creation of client-to-server file transmission FIFOs, particularly in cases involving file uploads to the server.

2.5 Client Receiver Process

The client receiver process reads responses from the server through the FIFO dedicated to server-client communication and prints them to the terminal. It operates in a loop, continuously receiving and processing server responses. Special cases, such as receiving keywords for client termination or connection validation, are also handled by this process.

In summary, the system architecture comprises processes responsible for managing client-server communication, executing commands, and handling file data transfer through distinct FIFO channels.

3 Implementation Details

3.1 Mutual Exclusions

Named semaphores are used when implementing needed mutexes. There are total of three distinct type of mutexes throughout the system; file mutex, server mutex, and connection mutex.

3.1.1 File Mutex

File mutexes are used whenever there's a write or read execution on server files. This enables that only one process can only either read or write to the same file at the same time. This prevent data corruption. Also every file has its own named semaphore, so it's not like if a worker writes on fileA the others need to wait to read or write files other than fileA.

3.1.2 Server Mutex

Server mutex is used for waiting for write calls on files or archiving process on server whenever the other is in the execution. This way archived `.tar` file is not corrupted. There's only one server named semaphore per server for this purpose, but the name is still special to the server (using its PID), so it's possible to run several server processes.

3.1.3 Connection Mutex

This is the only mutex that is accessible from client-side, which is inevitable since the purpose of this mutex is to prevent connection requests intervene with each other. Since server is only informed after it looks at the FIFO for connection requests it can't handle mutual exclusion for clients on this FIFO. But the creation and deletion (unlinking) of this named semaphore is handled by the server, which is only appropriate.

3.2 Server/Dispatcher Process

The server/dispatcher process serves as the central component of the file server system, responsible for managing client connections and coordinating the execution of client requests. This process is implemented in the `server.c` file.

3.2.1 Initialization

Upon startup, the server/dispatcher process initializes various data structures and resources required for its operation. These include:

- **Server FIFO (Named Pipe):** The process creates a FIFO with its PID, enabling communication with clients. This FIFO serves as the primary channel for receiving connection requests from clients.

- **Worker Process Management:** An array of worker process IDs (`worker_pids`) is initialized to keep track of active client connections. Each slot in the array corresponds to a client, with zero indicating an available slot.
- **Client Queue:** A linked list (`client_queue`) is initialized to manage client connection requests in case the server reaches its maximum capacity. Clients are queued up for processing when all worker slots are occupied.
- **Signal Handlers:** Signal handlers are set up to handle SIGCHLD signals, indicating child process termination, and SIGINT signals, enabling graceful server shutdown.

3.2.2 Connection Handling

The server/dispatcher process waits for incoming connection requests from clients via the server FIFO. Upon receiving a request, it checks if there's available capacity to serve the client:

1. If there are free worker slots, the dispatcher dispatches a worker process to handle the client's request. The worker process is responsible for executing commands and communicating with the client.
2. If the server is at maximum capacity, the dispatcher may either decline the connection or queue the client for later processing, depending on the specified behavior in the request.

3.2.3 Dynamic Process Management

The server/dispatcher process dynamically manages worker processes to ensure efficient resource utilization. When a worker process terminates (e.g., due to client disconnection), the dispatcher frees the corresponding worker slot and attempts to serve any queued clients waiting for connection.

3.2.4 Cleanup and Shutdown

Upon receiving a SIGINT signal or when exiting, the server/dispatcher process performs cleanup tasks, including terminating active worker processes, closing the server FIFO, and releasing allocated resources.

The server/dispatcher process plays a critical role in facilitating client-server communication and ensuring the smooth operation of the concurrent file access system.

3.3 Worker Process

The worker process handles client requests and performs file operations on behalf of the client. This process is implemented in the `worker.c` file.

3.3.1 Initialization

The worker process initializes various resources and performs setup tasks before executing client requests:

- **Argument Validation:** Validates command-line arguments to ensure proper usage.
- **Server Directory Validation:** Checks if the specified server directory path is valid.
- **Client ID Validation:** Validates the client ID received as a command-line argument.
- **Client Status Check:** Verifies if the client is still alive.
- **Signal Handler Setup:** Connects a signal handler to handle SIGINT signals for graceful shutdown.
- **Server Directory Opening:** Opens the server directory for file operations.
- **Logs Directory Preparation:** Prepares the directory structure for logging client activities.
- **Logs Directory Opening:** Opens the logs directory for writing log files.
- **Log File Opening:** Opens a log file for the current client session.

3.3.2 Client Request Handling

Upon establishing a connection with the client, the worker process enters a loop to handle incoming client requests:

- **FIFO Opening:** Opens FIFOs for communication with the client, enabling bidirectional message exchange.
- **Connection Establishment:** Sends a connection acceptance message to the client via the server-to-client FIFO.
- **Main Loop:** Enters a loop to continuously read incoming messages from the client.
- **Message Processing:** Parses client messages and executes corresponding commands, such as reading, writing, or deleting files.
- **Error Handling:** Handles errors encountered during message processing and reports them to the client.

The worker process remains active until the client closes the connection or terminates, ensuring responsive handling of client requests and maintaining robust communication with the file server system.

3.4 Decliner Process

The decliner process is responsible for declining connection requests from clients. This process is implemented in the `decliner.c` file.

3.4.1 Initialization

At startup, the decliner process initializes the necessary resources for declining client connections:

- **Server-to-Client FIFO (Named Pipe):** The process opens the FIFO with the client's PID for writing decline messages. This FIFO serves as the communication channel for sending decline responses to clients.

3.4.2 Declining Client Connections

The decliner process receives command-line arguments whilst being initialized by dispatcher process specifying the client PID and decline reason. It then proceeds to send a decline message to the specified client through the server-to-client FIFO. If the client is no longer alive, it exits gracefully without sending any message.

3.4.3 Main Function

The main function of the decliner process orchestrates the entire decline process:

1. **Command-Line Argument Parsing:** Parses command-line arguments to extract client PID and decline reason.
2. **Client Status Check:** Checks if the specified client is still alive by verifying the existence of the server-to-client FIFO.
3. **Constructing Decline Message:** Constructs a decline message based on the provided decline reason.
4. **Declining Client Connection:** Sends the decline message to the client through the server-to-client FIFO.
5. **Exiting the Process:** Gracefully exits the decliner process.

3.5 Client Parent Process

3.5.1 Initialization

The client parent process initializes by parsing command-line arguments to determine the operation mode and server PID. It validates the server's availability and starts sender and receiver worker processes. Signal handlers are set up to manage process termination and signals.

3.5.2 Child Process Management

The client parent process oversees the sender and receiver child processes responsible for client-server communication. It coordinates the creation and termination of these processes to ensure smooth communication.

3.5.3 Connection Establishment

Based on the specified command, the client parent process sends a connection request to the server and waits for a response. It handles the connection process according to the server's availability and the chosen operation mode.

3.5.4 Cleanup

Upon receiving termination signals or exiting, the client parent process cleans up by terminating child processes and releasing allocated resources.

3.6 Client Sender Process

3.6.1 Initialization

The client sender process initializes resources and sets up signal handlers for termination, SIGINT signals, and start signals from the client parent process.

3.6.2 FIFO Management

It manages FIFOs for communication with the server, including creating and opening the client-to-server FIFO for writing.

3.6.3 Command Transmission

Upon receiving start signals from the client parent process, the sender process enters a loop to read commands from standard input, send them to the server via FIFO, and handle client-side command processing.

3.6.4 Cleanup

After completing tasks or encountering errors, the sender process closes and unlinks the client-to-server FIFO.

3.7 Client Receiver Process

3.7.1 Initialization

The client receiver process initializes resources and sets up signal handlers for termination, SIGINT signals, and potential errors during initialization.

3.7.2 FIFO Management

It manages FIFOs for communication with the server, including creating and opening the server-to-client FIFO for reading.

3.7.3 Response Handling

The receiver process enters a loop to read responses from the server via FIFO. It processes incoming data and handles server responses according to the client's logic.

3.7.4 Cleanup

After completing tasks or encountering errors, the receiver process closes and unlinks the server-to-client FIFO.

4 Testing

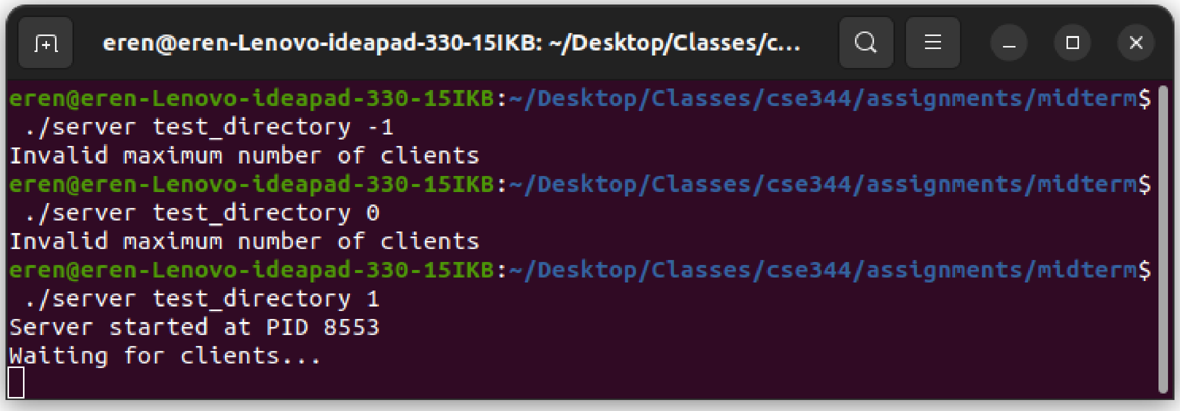
In this section every edge case of the system will be explored in respective subsections.

4.1 Command Executions

In this subsection the execution of client commands are explored for the edge cases special to that client command and not issues such as 'several client trying to write to a file' which is not due to a client command but client and system behaviour etc.

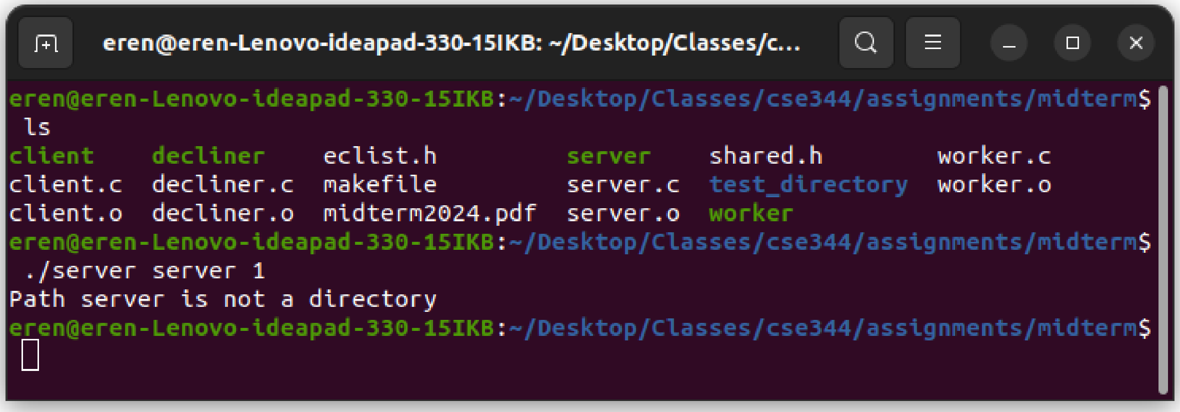
4.1.1 Server Initialization

In figures 2 and 3 you can see validation of server process arguments.



```
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse344/assignments/midterm$
 ./server test_directory -1
Invalid maximum number of clients
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse344/assignments/midterm$
 ./server test_directory 0
Invalid maximum number of clients
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse344/assignments/midterm$
 ./server test_directory 1
Server started at PID 8553
Waiting for clients...
```

Figure 2: Client count validation

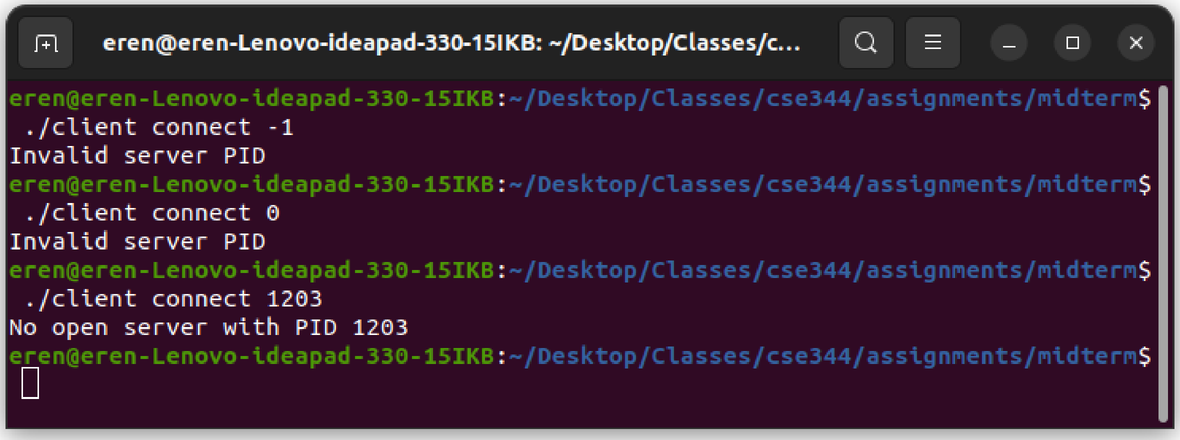


```
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse344/assignments/midterm$
 ls
client    decliner    eclist.h         server    shared.h        worker.c
client.c  decliner.c  makefile         server.c  test_directory  worker.o
client.o  decliner.o  midterm2024.pdf  server.o  worker
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse344/assignments/midterm$
 ./server server 1
Path server is not a directory
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse344/assignments/midterm$
```

Figure 3: Server directory path validation

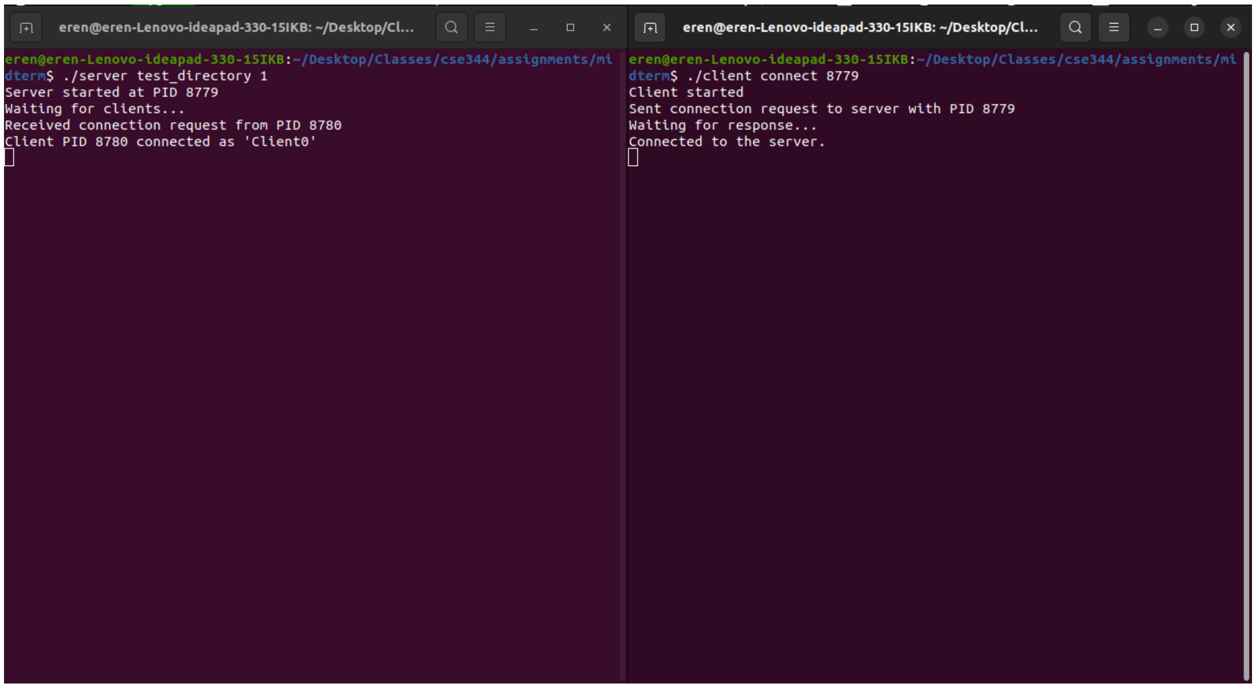
In figure 3 the given path points to the executable named `server` (which is the compiled executable we use), therefore process exit with shown warning.

4.1.2 Client Initialization



```
eren@eren-Lenovo-ideapad-330-15IKB: ~/Desktop/Classes/c...
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse344/assignments/midterm$
 ./client connect -1
Invalid server PID
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse344/assignments/midterm$
 ./client connect 0
Invalid server PID
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse344/assignments/midterm$
 ./client connect 1203
No open server with PID 1203
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse344/assignments/midterm$
```

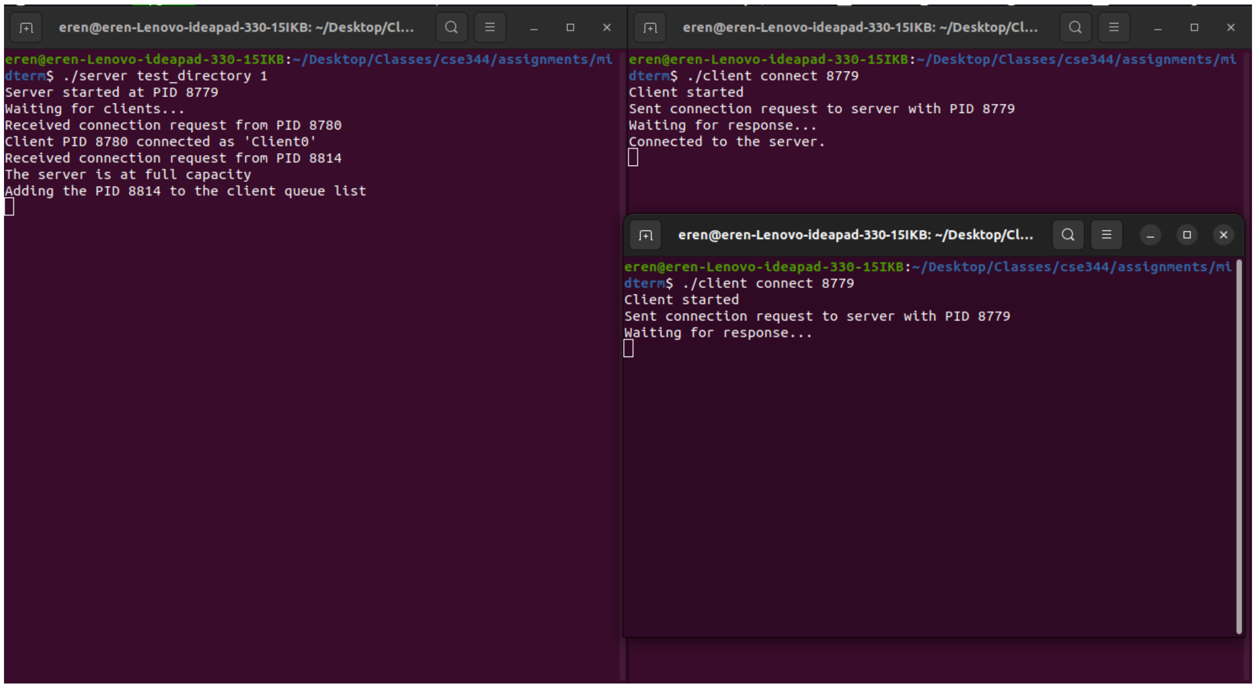
Figure 4: Example of invalid PID values



```
eren@eren-Lenovo-ideapad-330-15IKB: ~/Desktop/Cl...
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse344/assignments/mi
dterm$ ./server test_directory 1
Server started at PID 8779
Waiting for clients...
Received connection request from PID 8780
Client PID 8780 connected as 'Client0'
```

```
eren@eren-Lenovo-ideapad-330-15IKB: ~/Desktop/Cl...
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse344/assignments/mi
dterm$ ./client connect 8779
Client started
Sent connection request to server with PID 8779
Waiting for response...
Connected to the server.
```

Figure 5: Successful connection with 'connect' argument

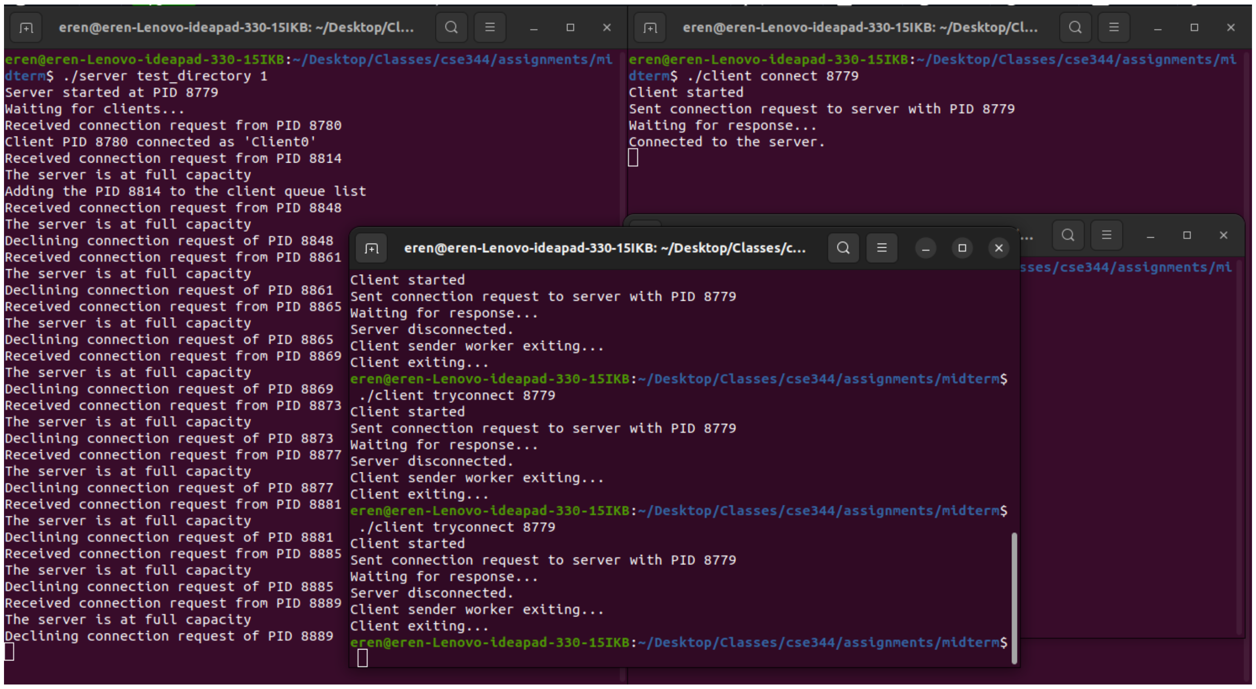


```
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse344/assignments/mi
dterm$ ./server test_directory 1
Server started at PID 8779
Waiting for clients...
Received connection request from PID 8780
Client PID 8780 connected as 'Client0'
Received connection request from PID 8814
The server is at full capacity
Adding the PID 8814 to the client queue list
```

```
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse344/assignments/mi
dterm$ ./client connect 8779
Client started
Sent connection request to server with PID 8779
Waiting for response...
Connected to the server.
```

```
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse344/assignments/mi
dterm$ ./client connect 8779
Client started
Sent connection request to server with PID 8779
Waiting for response...
```

Figure 6: Queueing up for connection with 'connect' argument



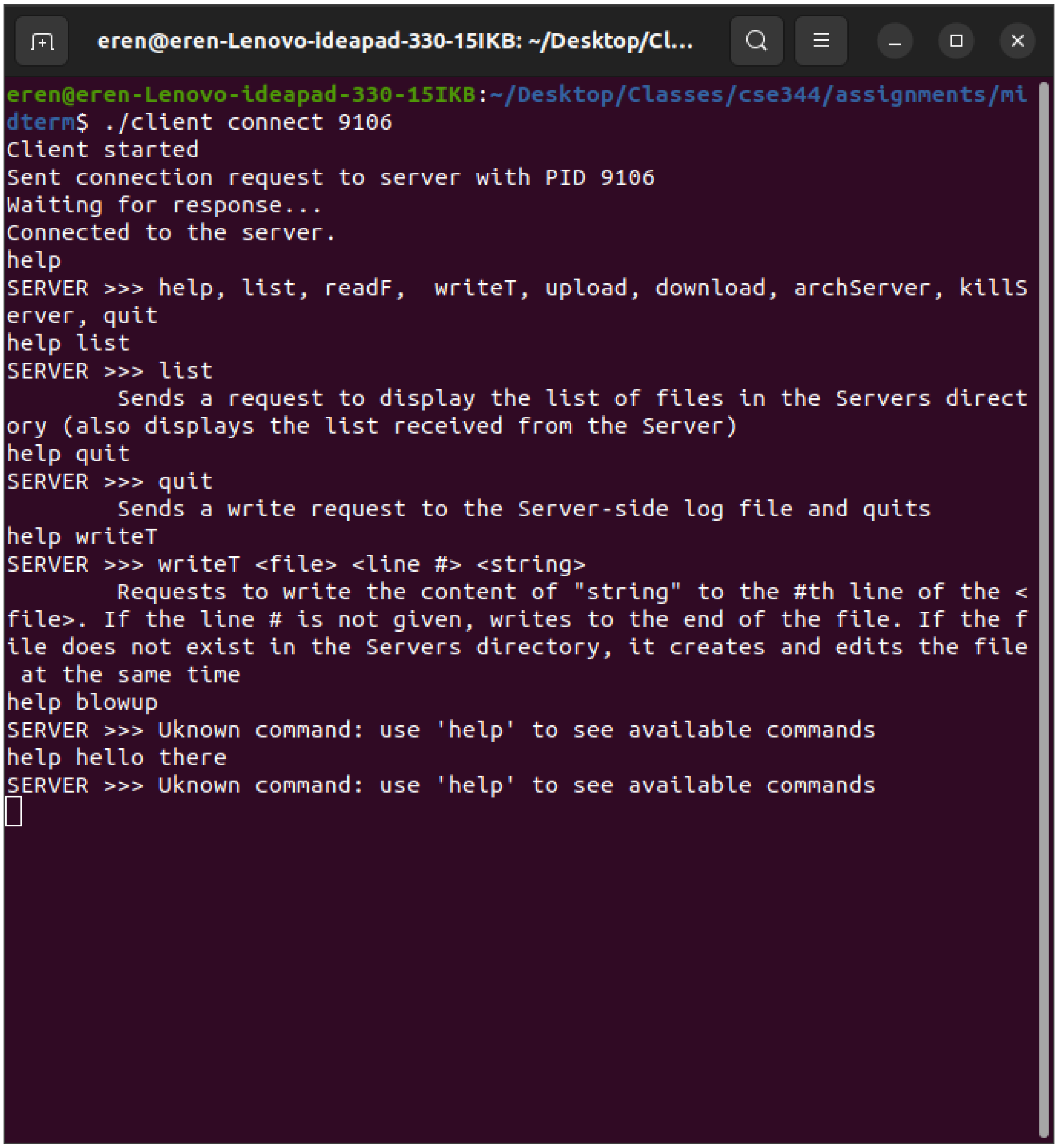
```
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse344/assignments/mi
dterm$ ./server test_directory 1
Server started at PID 8779
Waiting for clients...
Received connection request from PID 8780
Client PID 8780 connected as 'Client0'
Received connection request from PID 8814
The server is at full capacity
Adding the PID 8814 to the client queue list
Received connection request from PID 8848
The server is at full capacity
Declining connection request of PID 8848
Received connection request from PID 8861
The server is at full capacity
Declining connection request of PID 8861
Received connection request from PID 8865
The server is at full capacity
Declining connection request of PID 8865
Received connection request from PID 8869
The server is at full capacity
Declining connection request of PID 8869
Received connection request from PID 8873
The server is at full capacity
Declining connection request of PID 8873
Received connection request from PID 8877
The server is at full capacity
Declining connection request of PID 8877
Received connection request from PID 8881
The server is at full capacity
Declining connection request of PID 8881
Received connection request from PID 8885
The server is at full capacity
Declining connection request of PID 8885
Received connection request from PID 8889
The server is at full capacity
Declining connection request of PID 8889
```

```
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse344/assignments/mi
dterm$ ./client connect 8779
Client started
Sent connection request to server with PID 8779
Waiting for response...
Connected to the server.
```

```
Client started
Sent connection request to server with PID 8779
Waiting for response...
Server disconnected.
Client sender worker exiting...
Client exiting...
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse344/assignments/midterm$
 ./client tryconnect 8779
Client started
Sent connection request to server with PID 8779
Waiting for response...
Server disconnected.
Client sender worker exiting...
Client exiting...
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse344/assignments/midterm$
 ./client tryconnect 8779
Client started
Sent connection request to server with PID 8779
Waiting for response...
Server disconnected.
Client sender worker exiting...
Client exiting...
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse344/assignments/midterm$
```

Figure 7: Getting shut down in the case of full capacity server with 'tryconnect' argument

4.1.3 Help Command

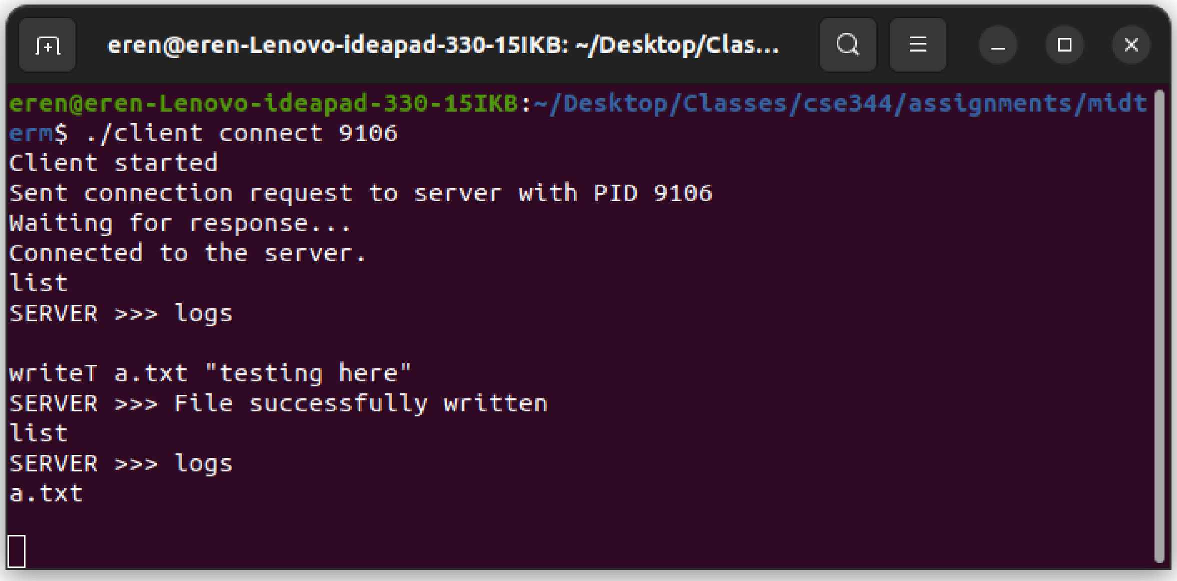


eren@eren-Lenovo-ideapad-330-15IKB: ~/Desktop/Cl...

```
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse344/assignments/mi
dterm$ ./client connect 9106
Client started
Sent connection request to server with PID 9106
Waiting for response...
Connected to the server.
help
SERVER >>> help, list, readF,  writeT, upload, download, archServer, killS
erver, quit
help list
SERVER >>> list
        Sends a request to display the list of files in the Servers direct
ory (also displays the list received from the Server)
help quit
SERVER >>> quit
        Sends a write request to the Server-side log file and quits
help writeT
SERVER >>> writeT <file> <line #> <string>
        Requests to write the content of "string" to the #th line of the <
file>. If the line # is not given, writes to the end of the file. If the f
ile does not exist in the Servers directory, it creates and edits the file
 at the same time
help blowup
SERVER >>> Uknown command: use 'help' to see available commands
help hello there
SERVER >>> Uknown command: use 'help' to see available commands
```

Figure 8: Correct and wrong usages of help command

4.1.4 List Command

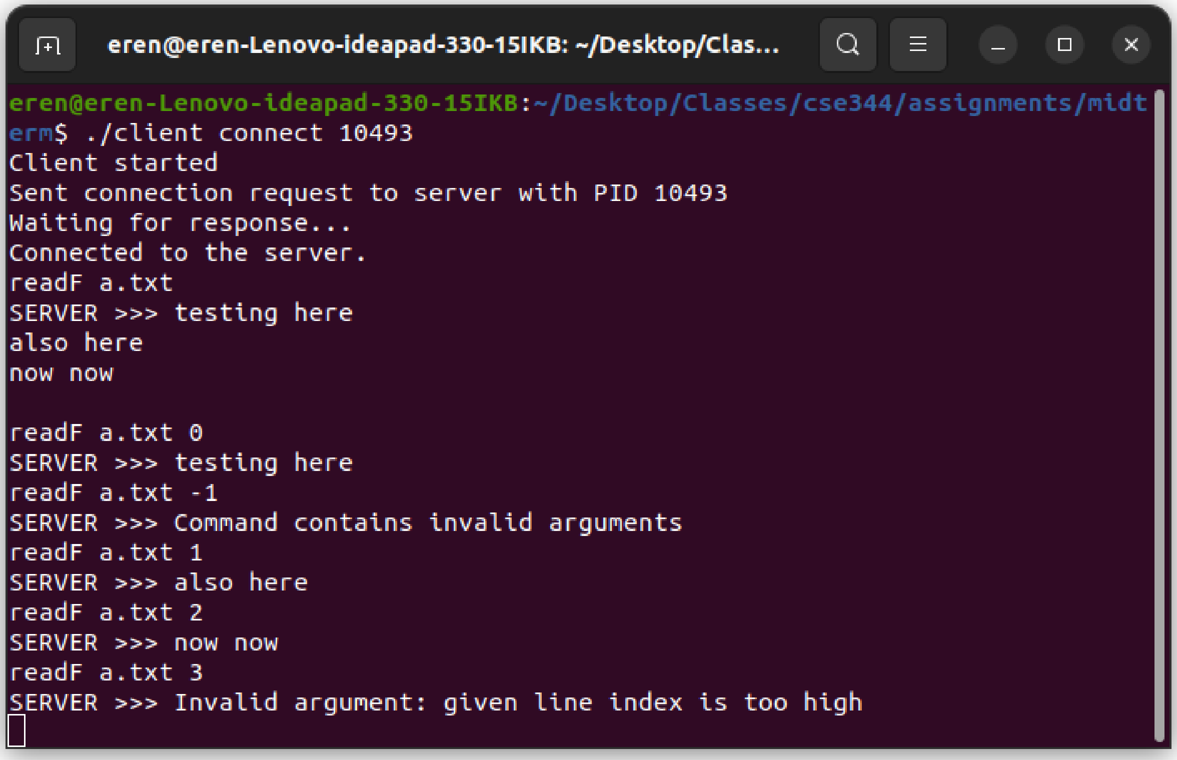


```
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse344/assignments/midt
erm$ ./client connect 9106
Client started
Sent connection request to server with PID 9106
Waiting for response...
Connected to the server.
list
SERVER >>> logs

writeT a.txt "testing here"
SERVER >>> File successfully written
list
SERVER >>> logs
a.txt
```

Figure 9: Listing initial contents of server directory and listing again after writing a new file

Every server directory has a directory in it for logs of clients that is created automatically.

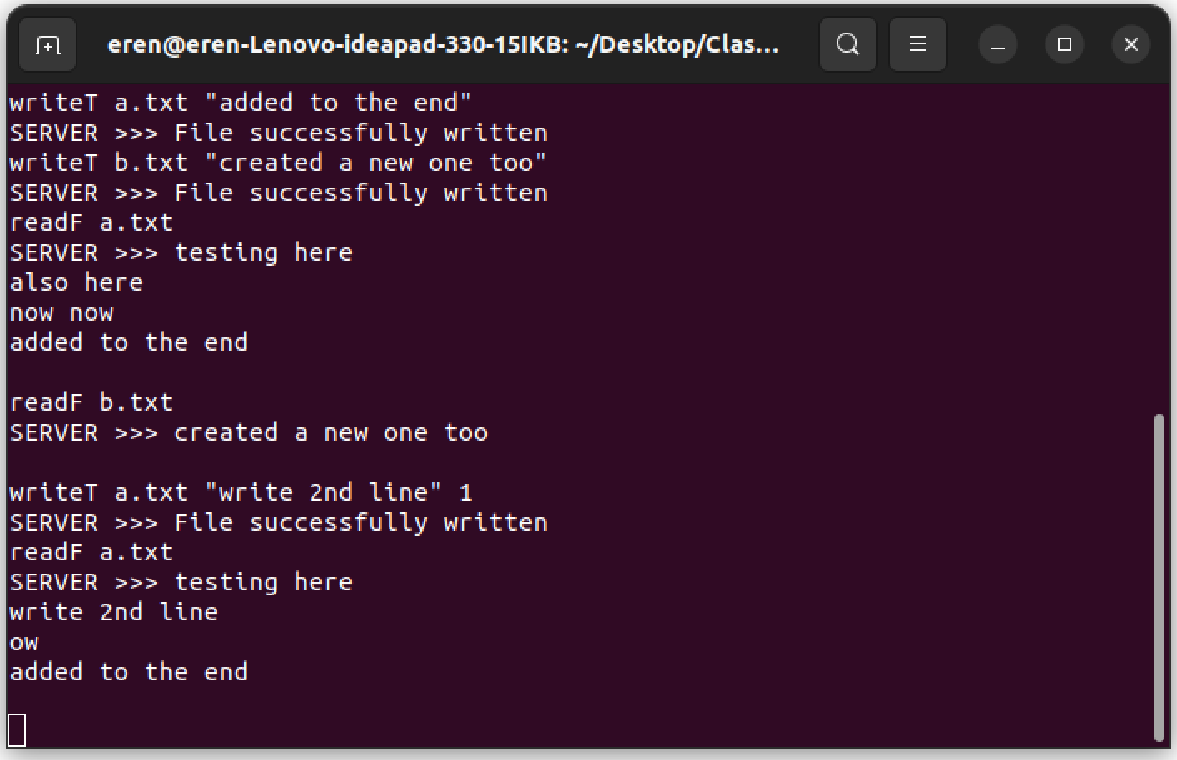
4.1.5 ReadF Command



```
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse344/assignments/midt
erm$ ./client connect 10493
Client started
Sent connection request to server with PID 10493
Waiting for response...
Connected to the server.
readF a.txt
SERVER >>> testing here
also here
now now

readF a.txt 0
SERVER >>> testing here
readF a.txt -1
SERVER >>> Command contains invalid arguments
readF a.txt 1
SERVER >>> also here
readF a.txt 2
SERVER >>> now now
readF a.txt 3
SERVER >>> Invalid argument: given line index is too high
```

Figure 10: ReadF command with and without line index with valid and invalid values

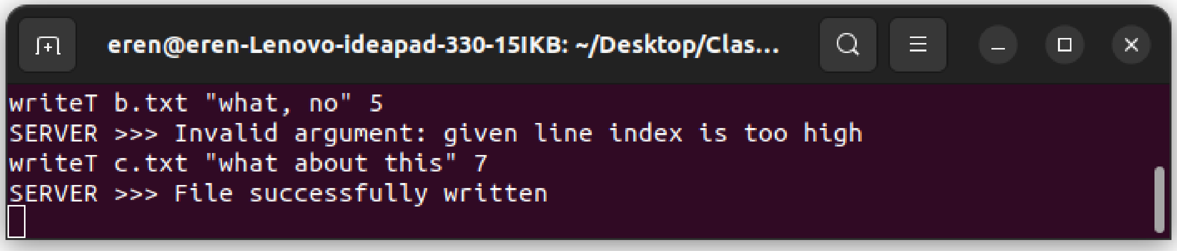
4.1.6 WriteT Command


eren@eren-Lenovo-ideapad-330-15IKB: ~/Desktop/Clas...

```
writeT a.txt "added to the end"
SERVER >>> File successfully written
writeT b.txt "created a new one too"
SERVER >>> File successfully written
readF a.txt
SERVER >>> testing here
also here
now now
added to the end

readF b.txt
SERVER >>> created a new one too

writeT a.txt "write 2nd line" 1
SERVER >>> File successfully written
readF a.txt
SERVER >>> testing here
write 2nd line
ow
added to the end
```

Figure 11: Writing to end of file and creation of file. Also writing on given line.

In Figure 11 it can be seen that when a line index is provided to `writeT` command it writes at given line but replacing/onto existing characters. This behaviour was default to standard `write()` function provided, it is left as it is a design choice rather than a bug. However it is important to note newline character is added to the written line, so any leftover from the overwritten line(s) is carried below to a newline.

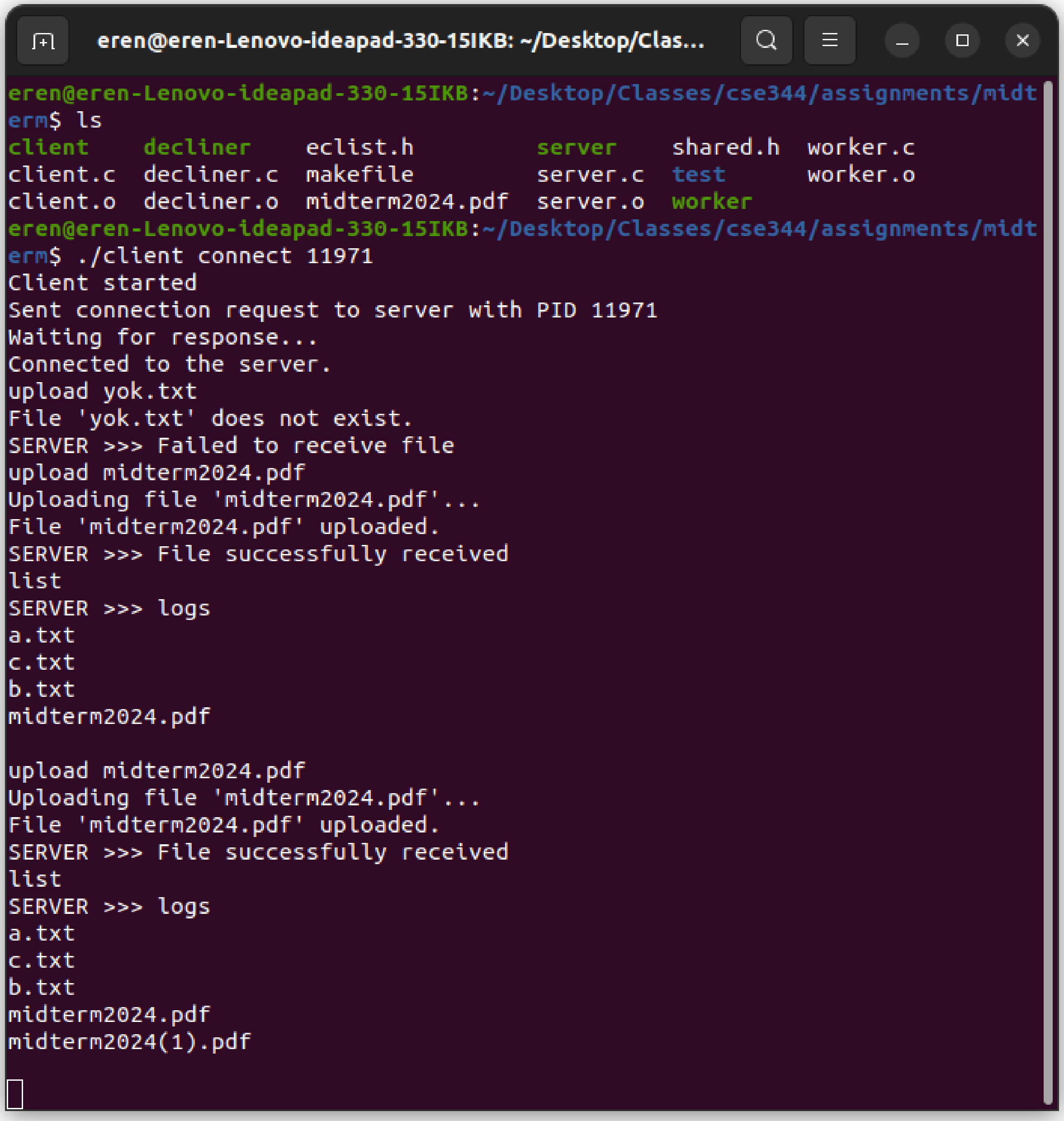

eren@eren-Lenovo-ideapad-330-15IKB: ~/Desktop/Clas...

```
writeT b.txt "what, no" 5
SERVER >>> Invalid argument: given line index is too high
writeT c.txt "what about this" 7
SERVER >>> File successfully written
```

Figure 12: Specifying line index

In Figure 12 it is seen whenever an index line is given to write to an existing file it is checked if it's a valid value, however it is ignored if the given path/filename does not exists but instead is created by the command and the given string is just written.

4.1.7 Upload Command



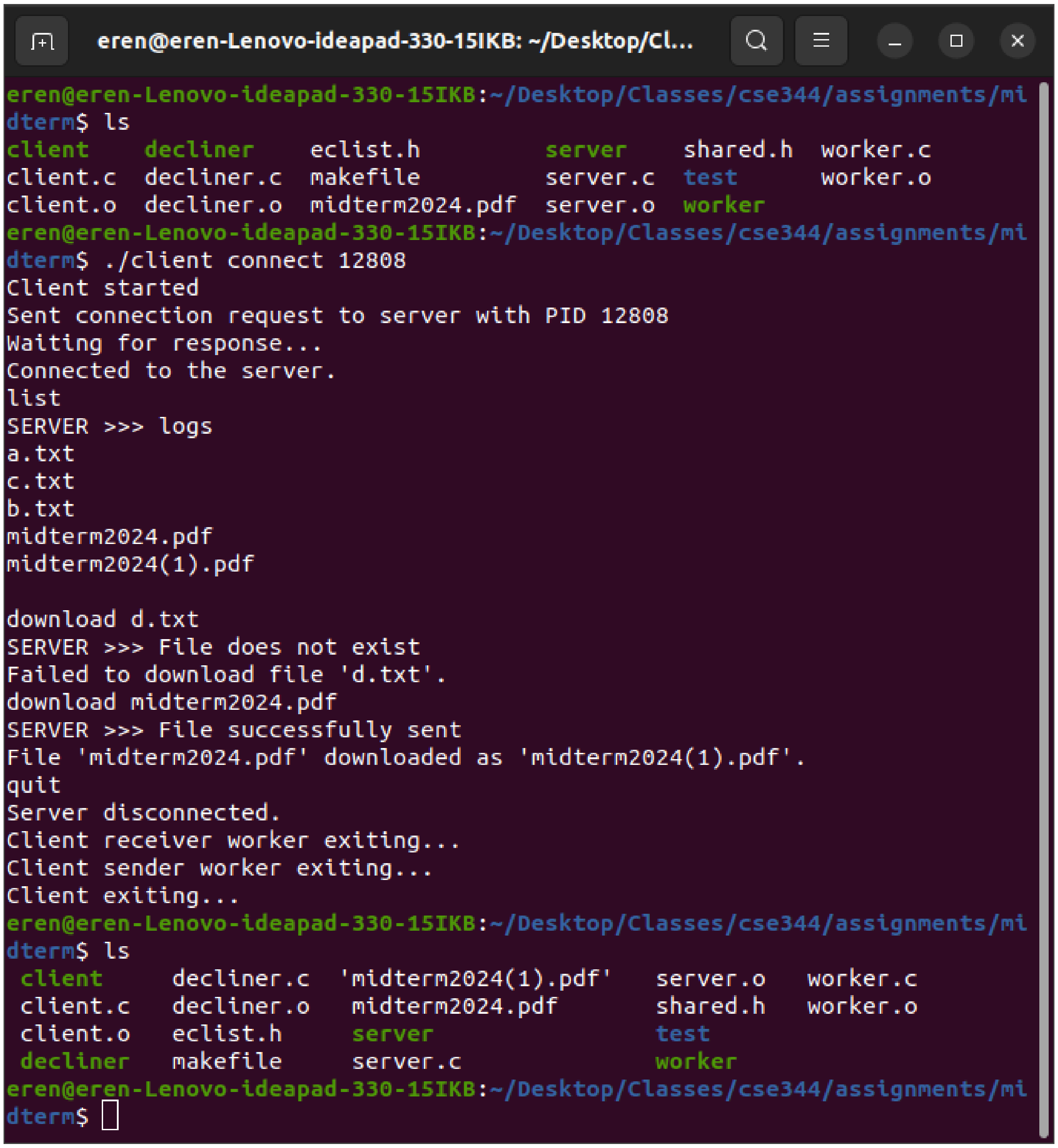
```
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse344/assignments/midt
erm$ ls
client     decliner     eclist.h        server     shared.h  worker.c
client.c   decliner.c   makefile        server.c   test      worker.o
client.o   decliner.o   midterm2024.pdf server.o   worker
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse344/assignments/midt
erm$ ./client connect 11971
Client started
Sent connection request to server with PID 11971
Waiting for response...
Connected to the server.
upload yok.txt
File 'yok.txt' does not exist.
SERVER >>> Failed to receive file
upload midterm2024.pdf
Uploading file 'midterm2024.pdf'...
File 'midterm2024.pdf' uploaded.
SERVER >>> File successfully received
list
SERVER >>> logs
a.txt
c.txt
b.txt
midterm2024.pdf

upload midterm2024.pdf
Uploading file 'midterm2024.pdf'...
File 'midterm2024.pdf' uploaded.
SERVER >>> File successfully received
list
SERVER >>> logs
a.txt
c.txt
b.txt
midterm2024.pdf
midterm2024(1).pdf
```

Figure 13: File uploading

In Figure 13 it is shown trying to upload a non existent file fails and uploading a file with same makes it upload with a slightly different name, similar how modern browsers handle downloaded file names.

It is also seen that file type does not matter when it comes down to file transfer.

4.1.8 Download Command



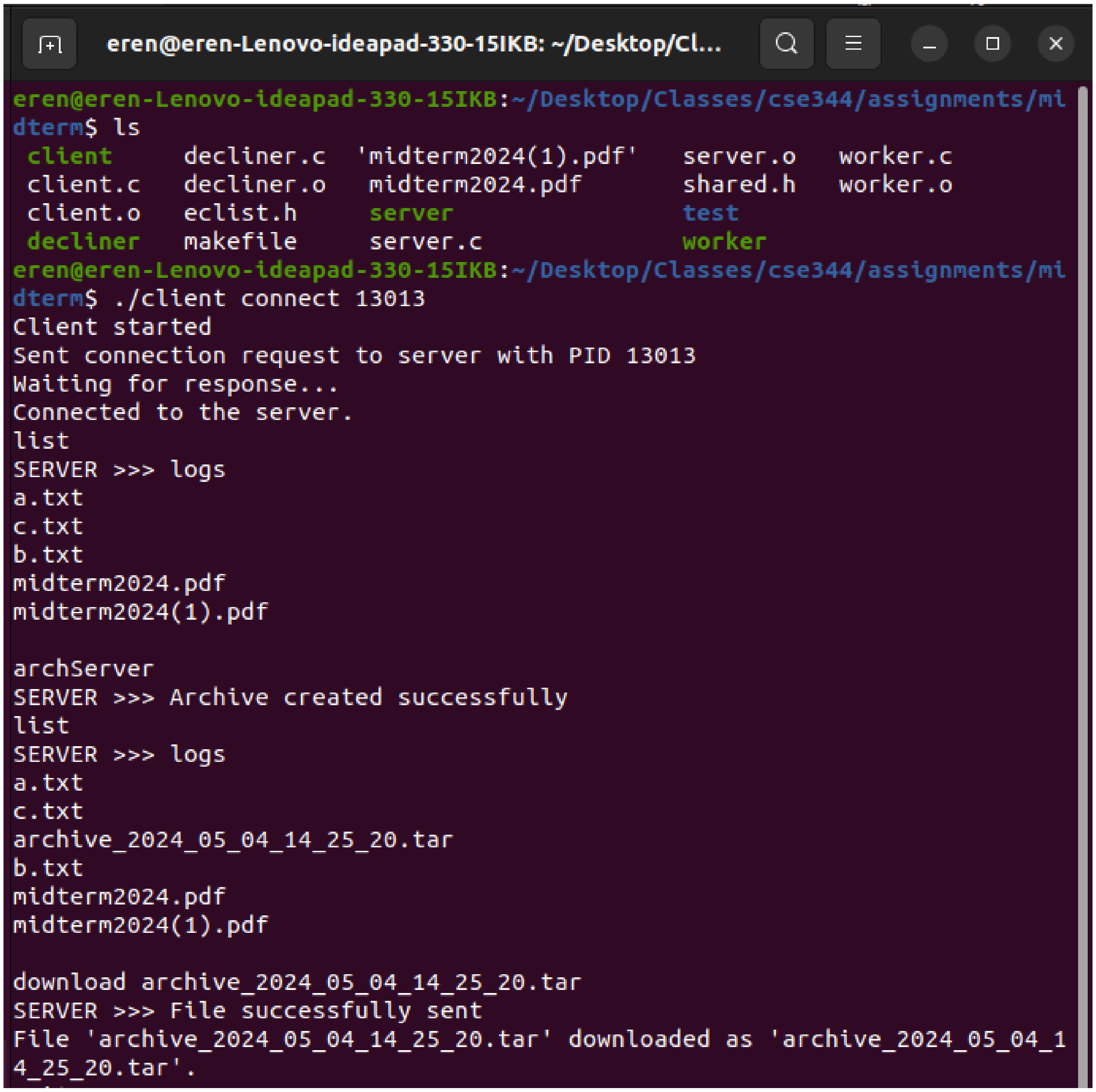
eren@eren-Lenovo-ideapad-330-15IKB: ~/Desktop/Cl...

```
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse344/assignments/mi
dterm$ ls
client     decliner     eclist.h        server     shared.h  worker.c
client.c   decliner.c   makefile        server.c   test      worker.o
client.o   decliner.o   midterm2024.pdf server.o   worker
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse344/assignments/mi
dterm$ ./client connect 12808
Client started
Sent connection request to server with PID 12808
Waiting for response...
Connected to the server.
list
SERVER >>> logs
a.txt
c.txt
b.txt
midterm2024.pdf
midterm2024(1).pdf

download d.txt
SERVER >>> File does not exist
Failed to download file 'd.txt'.
download midterm2024.pdf
SERVER >>> File successfully sent
File 'midterm2024.pdf' downloaded as 'midterm2024(1).pdf'.
quit
Server disconnected.
Client receiver worker exiting...
Client sender worker exiting...
Client exiting...
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse344/assignments/mi
dterm$ ls
 client     decliner.c  'midterm2024(1).pdf'  server.o   worker.c
 client.c   decliner.o   midterm2024.pdf      shared.h   worker.o
 client.o   eclist.h     server               test
 decliner   makefile     server.c             worker
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse344/assignments/mi
dterm$ 
```

Figure 14: File downloading

In Figure 14 it is shown when given path/filename is not in the server directory the download fails. And if the downloaded file shares its name with another already existing file in the directory of the client then the downloaded file's name is changed appropriately, similar to how modern browsers handle this issue.

4.1.9 ArchServer Command

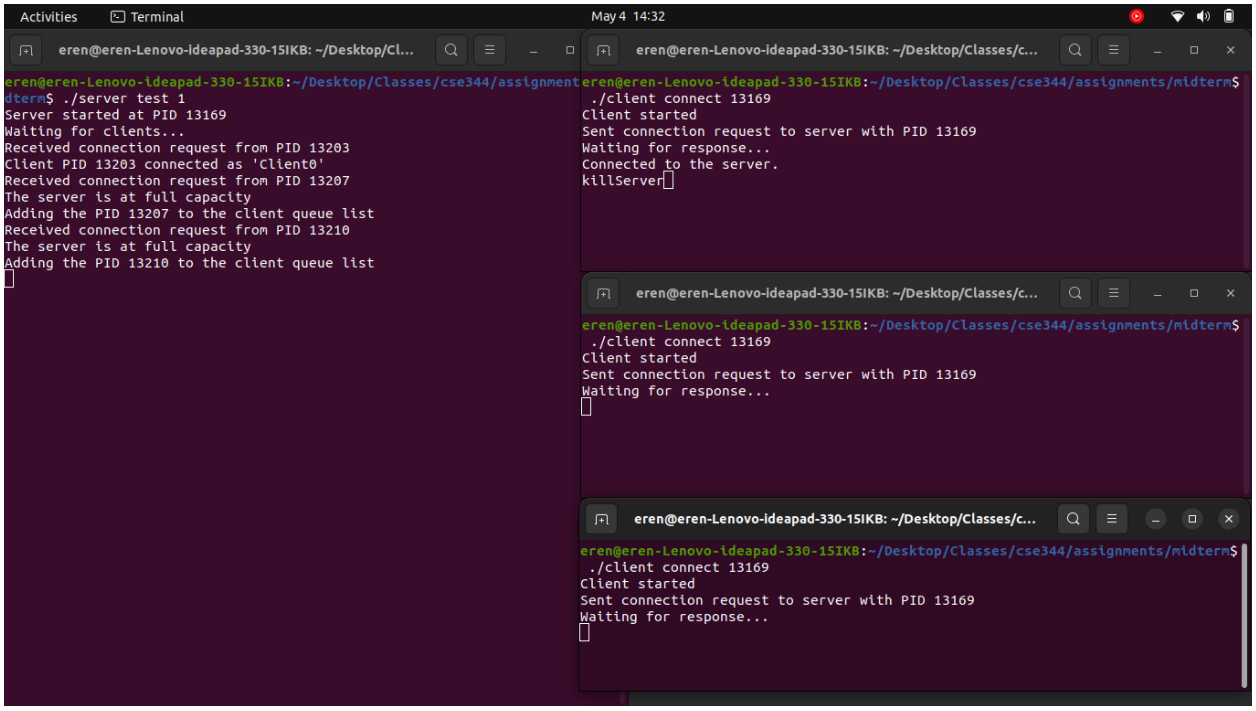


```
eren@eren-Lenovo-ideapad-330-15IKB: ~/Desktop/Cl...
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse344/assignments/mi
dterm$ ls
 client     decliner.c  'midterm2024(1).pdf'   server.o   worker.c
 client.c   decliner.o   midterm2024.pdf       shared.h   worker.o
 client.o   eclist.h     server                test
 decliner   makefile     server.c              worker
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse344/assignments/mi
dterm$ ./client connect 13013
Client started
Sent connection request to server with PID 13013
Waiting for response...
Connected to the server.
list
SERVER >>> logs
a.txt
c.txt
b.txt
midterm2024.pdf
midterm2024(1).pdf

archServer
SERVER >>> Archive created successfully
list
SERVER >>> logs
a.txt
c.txt
archive_2024_05_04_14_25_20.tar
b.txt
midterm2024.pdf
midterm2024(1).pdf

download archive_2024_05_04_14_25_20.tar
SERVER >>> File successfully sent
File 'archive_2024_05_04_14_25_20.tar' downloaded as 'archive_2024_05_04_1
4_25_20.tar'.
```

Figure 15: Archiving the server directory

In Figure 15 it is shown the archiving of the server is done and the archive is put into the server directory. The archiving process ignores already existing `.tar` files so this is not a problem. After archiving the user can download the `.tar` file if they want to.

4.1.10 KillServer Command



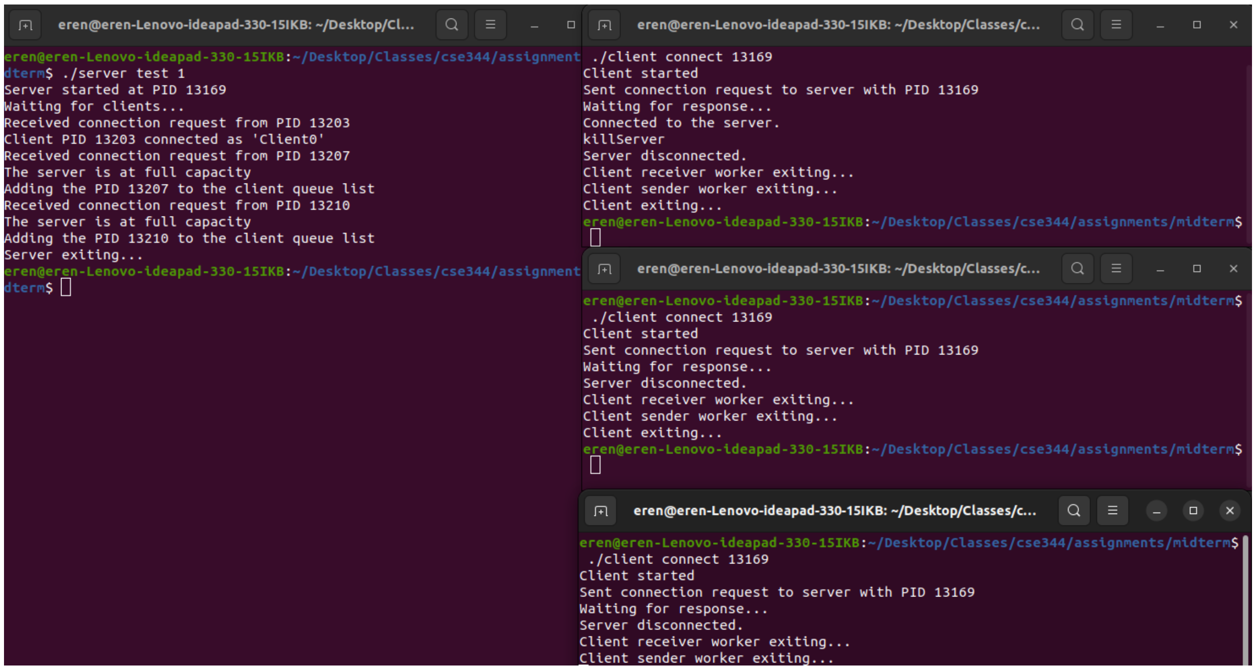
```
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse344/assignment
dterm$ ./server test 1
Server started at PID 13169
Waiting for clients...
Received connection request from PID 13203
Client PID 13203 connected as 'Client0'
Received connection request from PID 13207
The server is at full capacity
Adding the PID 13207 to the client queue list
Received connection request from PID 13210
The server is at full capacity
Adding the PID 13210 to the client queue list
```

```
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse344/assignments/midterm$
 ./client connect 13169
Client started
Sent connection request to server with PID 13169
Waiting for response...
Connected to the server.
killServer
```

```
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse344/assignments/midterm$
 ./client connect 13169
Client started
Sent connection request to server with PID 13169
Waiting for response...
```

```
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse344/assignments/midterm$
 ./client connect 13169
Client started
Sent connection request to server with PID 13169
Waiting for response...
```

Figure 16: Before killing the server



```
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse344/assignment
dterm$ ./server test 1
Server started at PID 13169
Waiting for clients...
Received connection request from PID 13203
Client PID 13203 connected as 'Client0'
Received connection request from PID 13207
The server is at full capacity
Adding the PID 13207 to the client queue list
Received connection request from PID 13210
The server is at full capacity
Adding the PID 13210 to the client queue list
Server exiting...
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse344/assignment
dterm$
```

```
 ./client connect 13169
Client started
Sent connection request to server with PID 13169
Waiting for response...
Connected to the server.
killServer
Server disconnected.
Client receiver worker exiting...
Client sender worker exiting...
Client exiting...
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse344/assignments/midterm$
```

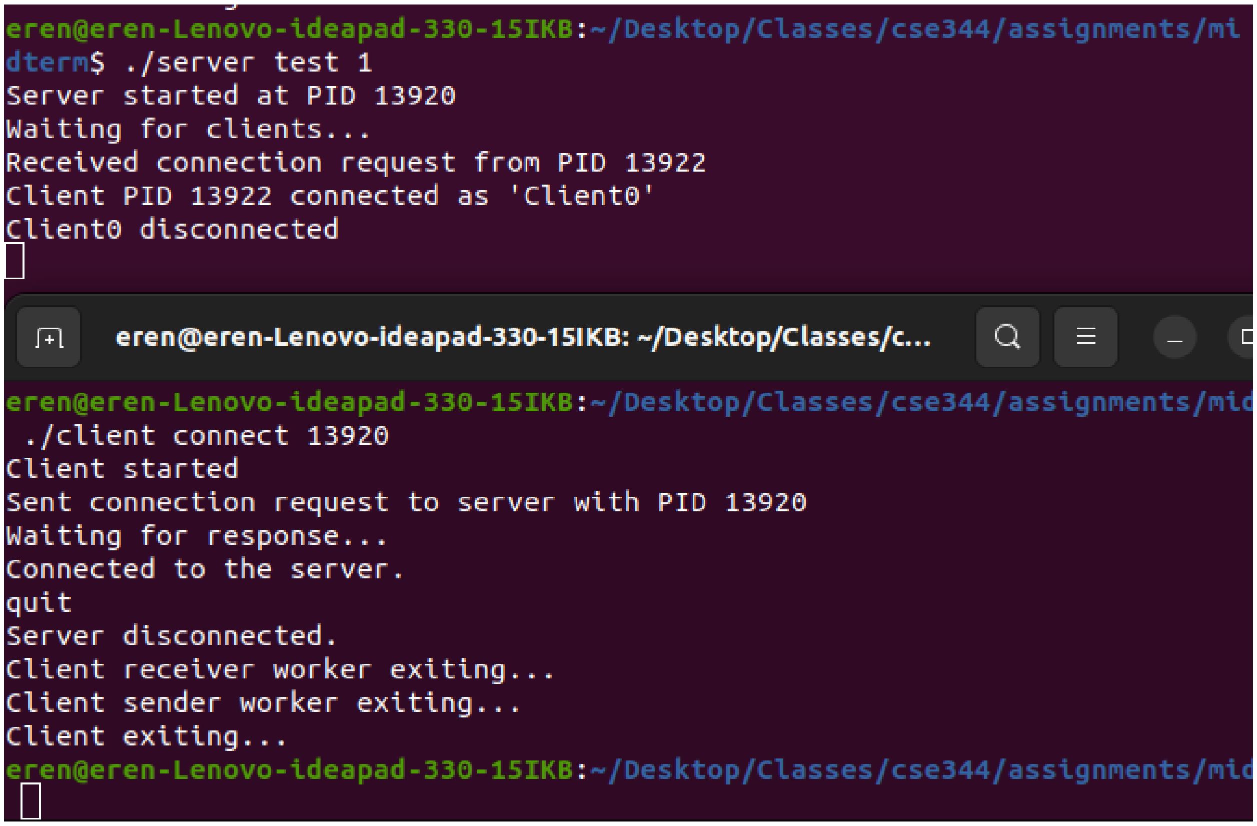
```
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse344/assignments/midterm$
 ./client connect 13169
Client started
Sent connection request to server with PID 13169
Waiting for response...
Server disconnected.
Client receiver worker exiting...
Client sender worker exiting...
Client exiting...
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse344/assignments/midterm$
```

```
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse344/assignments/midterm$
 ./client connect 13169
Client started
Sent connection request to server with PID 13169
Waiting for response...
Server disconnected.
Client receiver worker exiting...
Client sender worker exiting...
```

Figure 17: After killing the server

In figures 16 and 17 it can be seen that upon server's receival of command `killServer` it first shuts down clients in queue and then the connected clients and finally itself.

4.1.11 Quit Command



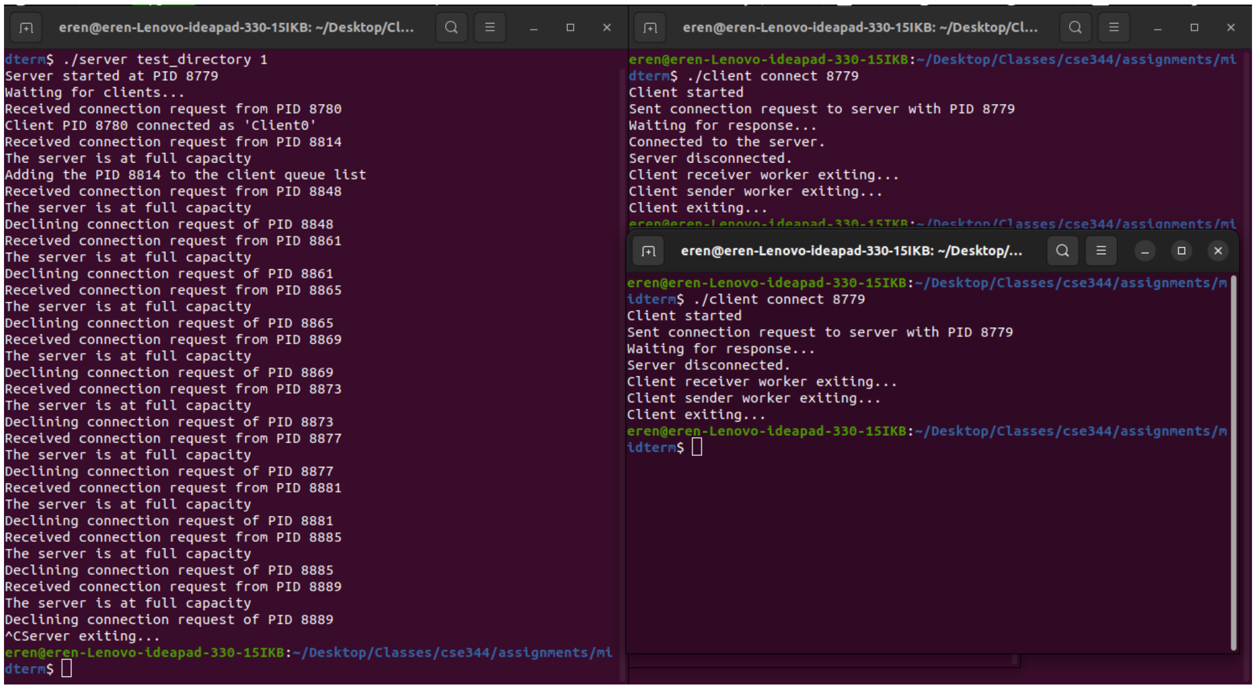
```
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse344/assignments/mi
dterm$ ./server test 1
Server started at PID 13920
Waiting for clients...
Received connection request from PID 13922
Client PID 13922 connected as 'Client0'
Client0 disconnected
```

```
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse344/assignments/mid
 ./client connect 13920
Client started
Sent connection request to server with PID 13920
Waiting for response...
Connected to the server.
quit
Server disconnected.
Client receiver worker exiting...
Client sender worker exiting...
Client exiting...
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse344/assignments/mid
```

Figure 18: Client quitting

The `quit` command is straightforward.

4.2 Signal Handlings

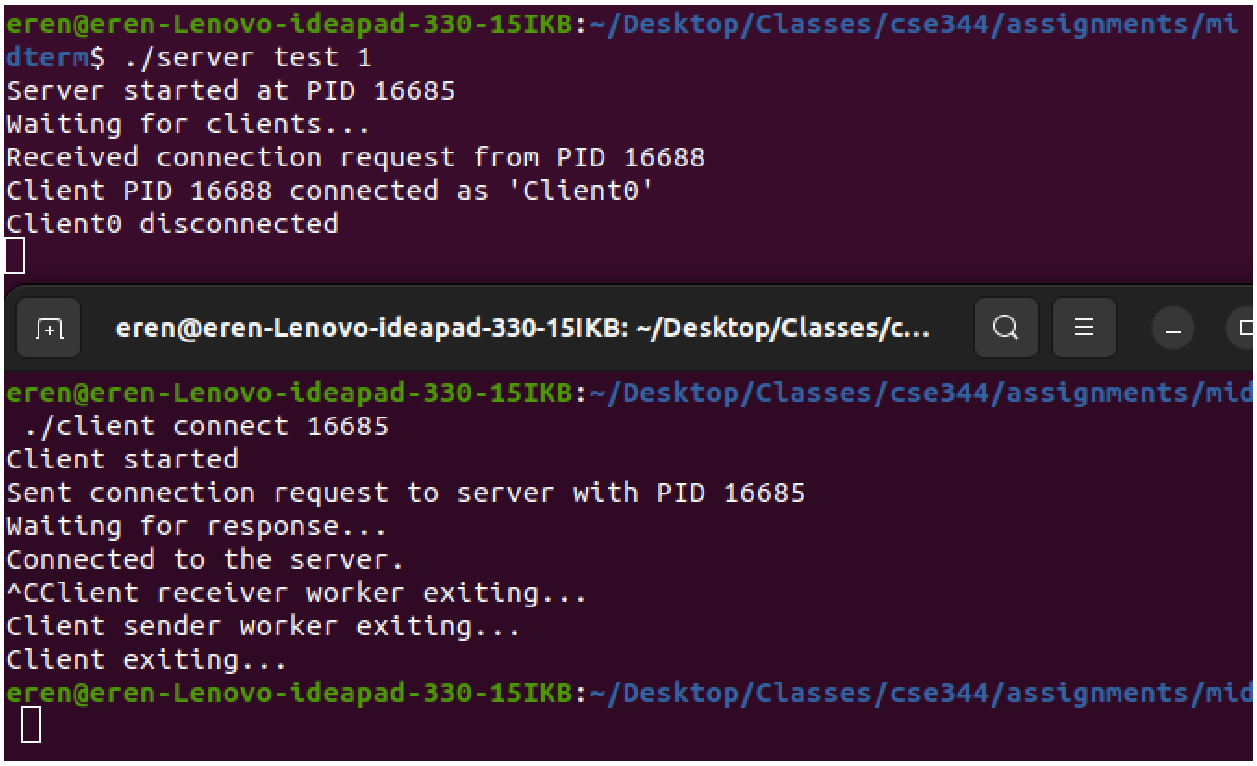

eren@eren-Lenovo-ideapad-330-15IKB: ~/Desktop/Cl...
eren@eren-Lenovo-ideapad-330-15IKB: ~/Desktop/Cl...
eren@eren-Lenovo-ideapad-330-15IKB: ~/Desktop/...

```
dterm$ ./server test_directory 1
Server started at PID 8779
Waiting for clients...
Received connection request from PID 8780
Client PID 8780 connected as 'Client0'
Received connection request from PID 8814
The server is at full capacity
Adding the PID 8814 to the client queue list
Received connection request from PID 8848
The server is at full capacity
Declining connection request of PID 8848
Received connection request from PID 8861
The server is at full capacity
Declining connection request of PID 8861
Received connection request from PID 8865
The server is at full capacity
Declining connection request of PID 8865
Received connection request from PID 8869
The server is at full capacity
Declining connection request of PID 8869
Received connection request from PID 8873
The server is at full capacity
Declining connection request of PID 8873
Received connection request from PID 8877
The server is at full capacity
Declining connection request of PID 8877
Received connection request from PID 8881
The server is at full capacity
Declining connection request of PID 8881
Received connection request from PID 8885
The server is at full capacity
Declining connection request of PID 8885
Received connection request from PID 8889
The server is at full capacity
Declining connection request of PID 8889
^CServer exiting...
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse344/assignments/mi
dterm$
```

```
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse344/assignments/mi
dterm$ ./client connect 8779
Client started
Sent connection request to server with PID 8779
Waiting for response...
Connected to the server.
Server disconnected.
Client receiver worker exiting...
Client sender worker exiting...
Client exiting...
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse344/assignments/mi
```

```
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse344/assignments/m
idterm$ ./client connect 8779
Client started
Sent connection request to server with PID 8779
Waiting for response...
Server disconnected.
Client receiver worker exiting...
Client sender worker exiting...
Client exiting...
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse344/assignments/m
idterm$
```

Figure 19: Server side SIGINT handling



```
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse344/assignments/mi
dterm$ ./server test 1
Server started at PID 16685
Waiting for clients...
Received connection request from PID 16688
Client PID 16688 connected as 'Client0'
Client0 disconnected
```

```
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse344/assignments/mid
 ./client connect 16685
Client started
Sent connection request to server with PID 16685
Waiting for response...
Connected to the server.
^CClient receiver worker exiting...
Client sender worker exiting...
Client exiting...
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse344/assignments/mid
```

Figure 20: Client side SIGINT handling

4.3 Mutual Exclusion

It is practically very hard to produce race condition cases, therefore instead of screenshots the source code segments are provided in this section. And the irrelevant parts are represented with comments //....

```
// client.c
int send_connection_request(int server_pid, int nonblock)
{
    // variable inits...

    char mutex_path[MAX_FILE_NAME_LENGTH];
    sem_t *sem = sem_open(get_server_fifo_mutex_path(server_pid, mutex_path), 0);
    if (sem == SEM_FAILED)
    {
        perror("sem_open");
        return -1;
    }
    sem_wait(sem);

    // connection request sending...

    sem_post(sem);
    sem_close(sem);

    return 0;
}
```

And below is

```
// server.c

void cleanup()
{
    // ...
    sem_close(connection_mutex);
    char mutex_path[MAX_FILE_NAME_LENGTH];
    sem_unlink(get_server_fifo_mutex_path(getpid(), mutex_path));
}

int main(int argc, char *argv[])
{
    // ...
    char mutex_path[MAX_FILE_NAME_LENGTH];
    connection_mutex = sem_open(get_server_fifo_mutex_path(getpid(), mutex_path), O_CREAT | O_EXCL, 066
    if (connection_mutex == SEM_FAILED)
    {
        perror("sem_open");
        return 1;
    }
    // ...
    return 0;
}
```

Above is the handling of the race condition of connection request message sending of the clients, since there is only one FIFO for receiving connection requests it's possible that two or more `write` calls on the FIFO interfering each other. These code parts prevent that.

```
// worker.c
int download_file(const char *file_path)
{
    // ...
    char archive_mutex_path[MAX_FILE_NAME_LENGTH];
    sem_t *archive_sem = sem_open(get_server_data_mutex_path(getppid(), archive_mutex_path), O_CREAT, 06
    sem_wait(archive_sem);

    char mutex_path[MAX_FILE_NAME_LENGTH];
    sem_t *sem = sem_open(get_file_data_mutex_path(file_path, mutex_path), O_CREAT, 0666, 1);
    sem_wait(sem);

    // ...

    sem_post(sem);
    sem_close(sem);
    sem_unlink(mutex_path);
    sem_post(archive_sem);
    sem_close(archive_sem);
    sem_unlink(archive_mutex_path);
    return 0;
}

int handle_readF_command(const char **parsed_command)
{
    // ...
    char mutex_path[MAX_FILE_NAME_LENGTH];
    sem_t *sem = sem_open(get_file_data_mutex_path(file_path, mutex_path), O_CREAT, 0666, 1);
    sem_wait(sem);

    // ...

    sem_post(sem);
    sem_close(sem);
    sem_unlink(mutex_path);
    return 0;
}

int handle_writeT_command(const char **parsed_command)
{
    // ...

    char archive_mutex_path[MAX_FILE_NAME_LENGTH];
    sem_t *archive_sem = sem_open(get_server_data_mutex_path(getppid(), archive_mutex_path), O_CREAT, 06
    sem_wait(archive_sem);

    char mutex_path[MAX_FILE_NAME_LENGTH];
    sem_t *sem = sem_open(get_file_data_mutex_path(file_path, mutex_path), O_CREAT, 0666, 1);
    sem_wait(sem);

    // ...

    sem_post(sem);
    sem_close(sem);
    sem_unlink(mutex_path);
    sem_post(archive_sem);
    sem_close(archive_sem);
    sem_unlink(archive_mutex_path);

    // ...

    return 0;
}

int handle_archServer_command(const char **parsed_command)
{
    char archive_mutex_path[MAX_FILE_NAME_LENGTH];
    sem_t *archive_sem = sem_open(get_server_data_mutex_path(getppid(), archive_mutex_path), O_CREAT, 0
    sem_wait(archive_sem);

    // ...

    sem_post(archive_sem);
    sem_close(archive_sem);
    sem_unlink(archive_mutex_path);
    return 0;
}

int handle_killServer_command(const char **parsed_command)
{
    char archive_mutex_path[MAX_FILE_NAME_LENGTH];
    sem_t *archive_sem = sem_open(get_server_data_mutex_path(getppid(), archive_mutex_path), O_CREAT, 0
    sem_wait(archive_sem);

    kill(getppid(), SIGINT);

    sem_post(archive_sem);
    sem_close(archive_sem);
    sem_unlink(archive_mutex_path);
    return 0;
}
```

Code parts of `worker.c` given above ensures the data integrity and prevention of data corruptions, making use of respective file data mutexes and server archive mutex.

The creation and deletion of the mutexes in the same function prevent dangling named semaphores, and whenever another worker process may try to create an existing mutex, it just opens and given value and flag is ignored. This behaviour is mentioned in man page of `sem_open`.