CSE 344 – System Programming
Midterm Project Report

Süleyman Burak Yaşar
1901042662

Content:

1. System Design and Architecture

The system consists of two parts: one client and one server. Clients connect to the server to perform various file operations. FIFOs are used for communication between them. These FIFOs facilitate bidirectional communication between the client and the server.

Server Side:

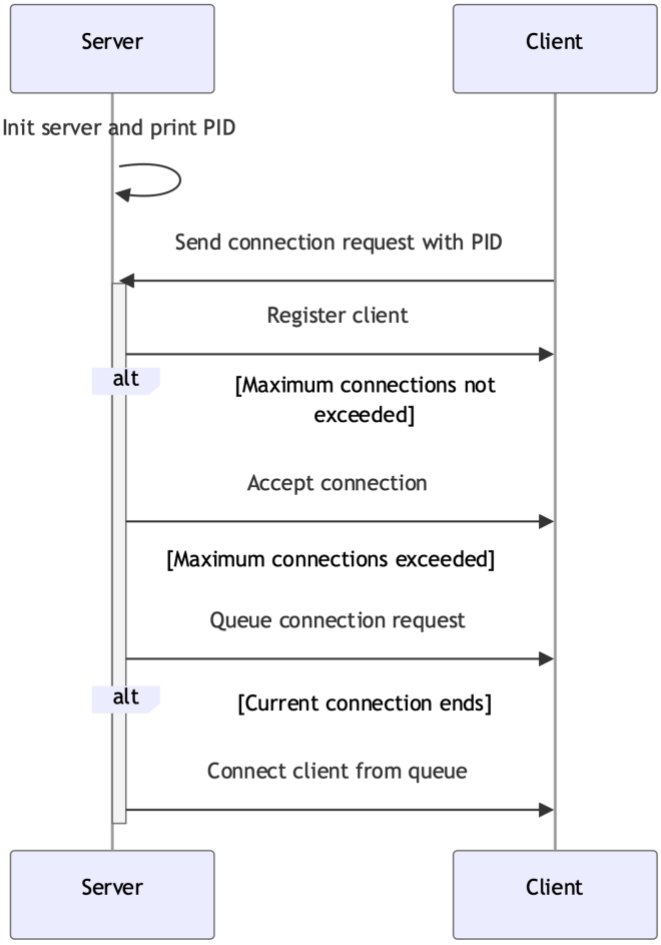
The server has its own directory where it stores files. Additionally, the server is equipped with one main FIFO which serves multiple purposes. This FIFO is responsible for managing connection requests from clients as well as handling exit processes from either the clients or the server's own child processes.

Client Side:

The client has two FIFOs. The first one is the request FIFO and the second is the response FIFO. The first FIFO is unidirectional from the client to the server. Through this FIFO, requests are made to the server, and it is also used for data transmission when necessary. The second FIFO is unidirectional from the server to the client. This FIFO is used for data transfer from the server to the client.

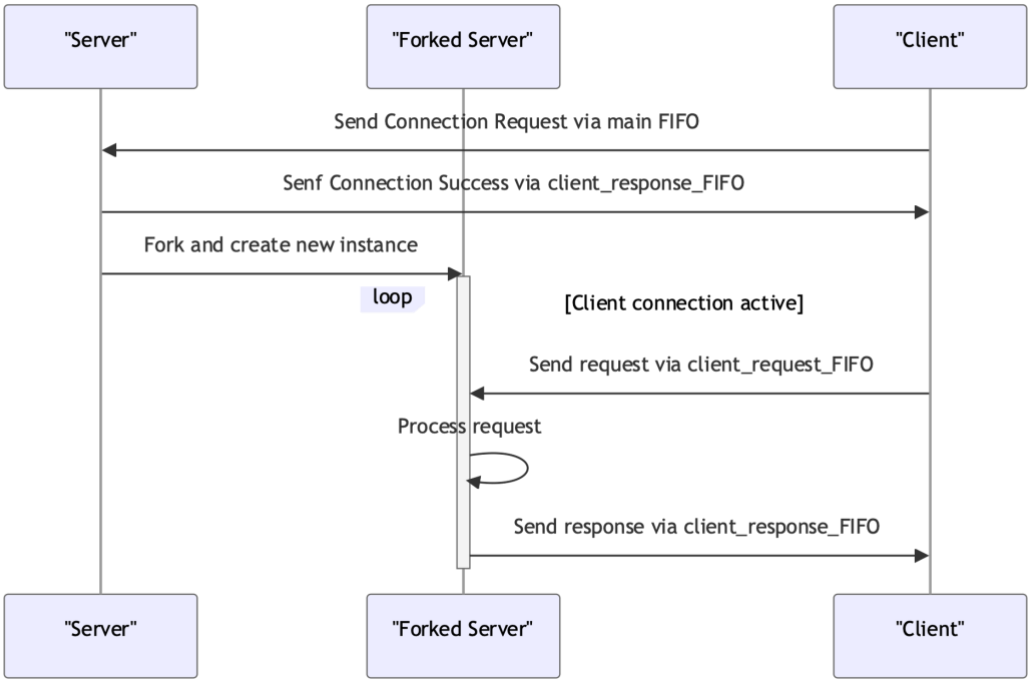
Connection Design:

- First, the server part is initiated and the server's PID is printed.
- Using the server's PID, clients send connection requests through the main FIFO on the server.
- These connection requests also include their own PIDs.
- The server uses the client PIDs received from the requests to send responses to those clients using the client_response_FIFO.
- Clients connecting to the server are registered.
- If the maximum number of connections to the server is exceeded, new connection requests are queued.
- If a current connection ends, the client at the front of the queue can connect to the server.



Communication Design:

- After establishing a connection, the server forks to create another instance of itself. This new instance is responsible for handling incoming requests and responses, and it is directed to the client_request_handler.
- Requests from clients are received by the server via the client_request_FIFO.
- The server processes these requests and sends responses to the client via the client_response_FIFO.
- These operations continue until the client connection is terminated.



Client Disconnect Design:

- When a client initiates a disconnect, the child process responsible for that client notifies the parent server process via the main FIFO that the connection with this client has ended.
- The child process terminates itself.
- The parent server process processes the disconnect request.
- If there is a pending request in the queue, it connects that request to the server.

Server Terminate Design:

If the client needs to terminate the server due to a "killServer" command, a signal, or any other reason:

- The parent process of the server sends a termination request to all active child processes.
- It waits for all child processes to finish their tasks.
- After all child processes have terminated, it closes any open files, releases memory resources appropriately, and finally terminates itself.

Request And Response Structure:

1. **Request:**

   1.1. Types of Request

   Our system defines various types of requests, which are specified in an enum named RequestType. This enumeration includes the following:

   - **CONNECTION_REQUEST (0):** Represents a connection request where the client initiates its first connection to the server.
   - **CHILD_PARENT (1):** This type is often used for internal system communications, particularly for managing the relationship between child processes and their parent processes.
   - **REQUEST (2):** A general request type used for various operations within the system.

   1.2. Command Types

   The system utilizes an enum called CommandType to execute a range of commands. The commands and their index values are listed below:

   - CONNECT (0): Command to connect to the server.
   - TRY_CONNECT (1): Command to attempt connection, useful after failed attempts.
   - DISCONNECT (2): Command to disconnect the current connection.

- KILL (3): Command to terminate a process.
- LIST (4): Command to list files or processes in the system.
- READFILE (5): Command to read a file.
- WRITEFILE (6): Command to write to a file.
- UPLOAD (7): Command to upload a file.
- DOWNLOAD (8): Command to download a file.
- ARCH_SERVER (9): Command to archive the server.
- QUIT (10): Command to exit the system.

1.3. Request Structure

Requests within the system are represented by a structure named Request. This structure consists of the following fields:

- **requestType:** A field of type RequestType that indicates the type of request.
- **PID:** The process identifier for the client or child process.
- **command:** A field of type CommandType that specifies the command to be executed.
- **parameter1, parameter2, parameter3:** Arrays each of 256 characters in length that carry parameters required by the command.

2. Response:

The FileResponse structure is designed to help manage the process of sending files over a FIFO in smaller parts, called chunks. Here's what each part of this structure is used for:

- **totalChunks:** This tells you how many pieces the complete file is divided into. It helps in understanding how many times data will be sent to get the whole file.

- **chunkNumber:** This number shows the order of the current piece being sent. It helps ensure that the pieces of the file are put back together in the right order when they reach their destination.

- **size:** This indicates how much data is in the current chunk. This is important because the last piece of the file might be smaller than the others.

- **data:** This is a large array where the actual file data for the current chunk is stored. The size of the array is big enough to handle a good amount of data at once, making it easier to manage large files.

Race Condition:

Shared memory was utilized to create a semaphore, which effectively addressed race conditions and synchronization challenges among processes. This semaphore mechanism ensured that critical sections of code were executed in a mutually exclusive manner, preventing conflicts and ensuring proper coordination between concurrent processes. By leveraging shared memory and semaphores, efficient communication and synchronization were achieved, enhancing the overall reliability and performance of the system.

2. Implementation Details

Initial Setups:

Server Side:

Servers starts by establishing a way to handle interruption signals, which are typically generated when the user wants to terminate the process.

Next, it creates a shared semaphore, which is a way to control access to a common resource by multiple processes in a concurrent system such as a multitasking operating system.

Server then creates main FIFO for inter-process communication. The fifo is named with the server's PID to ensure uniqueness.

Afterwards, it prepares a directory for the server's operation. The directory name is provided as an input to the server.

The Server then prepares to manage client connections. It creates an array to store the identifiers of the clients and a queue to manage the order of client servicing. The maximum number of clients is also provided as an input to the server program.

Finally, it sets up a log file for the server to record its operation. The log file is located in the server's directory. After ensuring the log file can be opened, it records the start of the server and its PID.

Client Side:

Initially, it sets up signal handlers for SIGINT and SIGTERM signals. These handlers allow the client to respond to interrupt signals (like when a user presses Ctrl+C) or termination signals (sent by other processes or the system).

Next, it checks the command-line arguments to ensure they are correct. The client expects two arguments: the connection type (either "Connect" or "tryConnect") and the server's process ID (PID). If the arguments are not as expected, it prints a usage message and exits.

Then, it constructs the name of the server's FIFO communication channel using the server's PID. This is the main channel the client will use to send requests to the server.

Finally, it creates two FIFOs specific to this client: one for sending responses to the server and one for receiving requests from the server. These FIFOs are named using the client's PID to ensure uniqueness. The server's PID is also stored for later use.

Connection:

Server Side:

The server first opens a FIFO named MAIN_FIFO in read-only mode.

If the FIFO opens successfully, the file descriptor is stored in global_main_fifo_fd and the server attempts to read a Request from the FIFO.

The server then checks if the requestType of the received Request is CONNECTION_REQUEST. If it is, the server prepares a ConnectionResponse to send back to the client. The server constructs the name of the client's FIFO (which it will use to send the response) by appending the client's PID to a base string. The server then opens the client's FIFO in write-only mode.

Next, the server checks if the maximum number of clients (MAX_CLIENTS) has been reached.

If the maximum has been reached and the client's command is TRY_CONNECT, the server sends a response indicating that the server is full and the client is rejected.

If the maximum has been reached and the client's command is CONNECT, the server sends a response indicating that the client is in a queue waiting for a connection.

In both cases, the server then closes the client's FIFO and the main FIFO, and continues to the next iteration of the loop.

If the maximum number of clients has not been reached, the server sends a response indicating a successful connection. The server stores the client's PID in client_pid_to_connect, and closes the client's FIFO.

Client Side:

It prepares a Request structure named connection_request, setting the request type to CONNECTION_REQUEST, the client's PID, and the command**.**

It opens the server's FIFO in write-only mode and writes the connection_request to it.

It then opens the client's response FIFO in read-only mode and reads a ConnectionResponse from it.

It checks the status of the ConnectionResponse. If the status is SUCCESS, it means the client has successfully connected to the server. If the status is IN_QUEUE, it means the client is in a queue waiting to connect to the server. If the status is neither SUCCESS nor IN_QUEUE, it means the server is full and the client's connection request is rejected.

If the client's connection request is rejected,clients terminate.

Client – Child Server Communication

Server:

After a successful connection from the client, the server forks to clone itself. The new child process enters the client_child function. Here, the client's two FIFOs are opened. Then, a loop is entered. Within this loop, the client_request_FIFO is read to await requests from the client. Requests are processed based on their commands. Responses to the client are provided through the client_response_FIFO. This process continues until the client disconnects.

Client:

After successfully connecting, the client opens its two dedicated FIFOs. It enters a loop to receive commands from the user. When the user inputs a command, the relevant function is called to process it. The client then sends a request to the server via the client_request_FIFO and receives responses through the client_response_FIFO. This loop continues until the client disconnects.

List

The list request from the client is sent to the server via the client_request_FIFO. The server processes this request by listing the files in the server directory and sends the list to the client through the client_response_FIFO. The client then displays this list to the user.

ReadF

with line number

The client sends the file name and line number to the server. The server reads the file name. If it exists, the file is opened; otherwise, an error message is sent to the client. Once opened, the file is read line by line. When the specified line is reached, it is saved to the buffer. The server then sends this line to the client. If the line does not exist, a message indicating that the line was not found is sent. The client displays the requested line to the user.

Without line number

The client sends the file name to the server. The server reads the file name. If it exists, the file is opened; otherwise, an error message is sent to the client. The size of the opened file is determined. It is then divided into chunks, each of which will be 32KB in size. The client is informed about the total number of chunks and the total size of the file. The server begins sending the file to the client in chunks. As each chunk is read by the client, it is displayed on the screen.

writeF

with line number

The client sends the file name, line number, and the line to be added to the server. The server reads the file name. If it exists, the file is opened; otherwise, an error

message is sent to the client. The file is then read using a buffer. The number of newlines and bytes read up to that point is recorded. Each byte read is rewritten using the `lseek()` function. When the specified line is reached, the line sent by the client is written to the file. Subsequently, any remaining unread data is read and rewritten. If the line is successfully written, a message indicating success is sent from the server to the client. If the line does not exist, a message indicating that the line was not found is sent.

without line number

The client sends the file name and the line to be appended to the server. The server reads the file name. If it exists, the file is opened in append mode; otherwise, an error message is sent to the client. The line is then written to the opened file. A message indicating the successful writing is sent to the client.

upload

The client attempts to open the file. If the file cannot be opened, an error message is returned, and the process terminates. The client sends the file name to the server. The server reads the file name and tries to create the file in the server directory. If a file with the same name already exists, the server tries again with an incremented file name.

The client determines the size of the opened file. It then divides the file into chunks, each 32KB in size. The total number of chunks and the total size of the file are sent to the server.

The file is sent from the client to the server in chunks. The server receives each chunk and writes it to the file it has created. Once all chunks have been received, the file upload is complete. A message indicating the successful upload is sent to the client.

download

Client sends the file name to the server. The server reads the file name. If it exists, the file is opened; otherwise, an error message is sent to the client. The size of the opened file is determined. It is then divided into chunks, each of which will be 32KB in size. The client is informed about the total number of chunks and the total size of the file.

The client attempts to create the file on its side. If a file with the same name already exists, the client tries again with an incremented file name. As the server begins sending the file to the client in chunks, the client receives each chunk and writes it to the file it has created. Once all chunks have been sent, the file download is complete.

archServer

A folder is created with the name entered by the user, and the names of the files in the server directory are retrieved using the list operation. Subsequently, these file names are downloaded to the created folder. Forking is then initiated. Within the child process, the tar program is executed using exec(), allowing it to archive the files within the folder and save them to the location of the client file before terminating itself. Meanwhile, the parent process waits for the child process to complete using waitpid. Upon the child process's termination, both the folder it opened and its contents are deleted, marking the completion of the operation.

Quit

The client sends a quit request to the server. Upon receiving this request, the child process in the server informs the parent process that the connection with the client has been terminated. The server exits the loop and performs the necessary file closing operations before terminating itself.

In the client, the necessary file closing operations are conducted, and the FIFOs are unlinked. Finally, the program terminates.

killServer

The client sends a "killServer" request to the server's main FIFO. Upon receiving this request, the parent process in the server sends a kill signal to all of its child processes and waits for all of them to terminate. Once all child processes have terminated, the parent process sends a kill signal to all connected clients. It then proceeds to perform the necessary file closing operations, unlinks the main FIFO, and terminates itself.

Signal Handling

Server Side:

Child

When one of the server's children attempts to close due to a signal, the child_handler function is called. This function sends a signal to the parent's

main FIFO indicating that the connection has been terminated. Then, it proceeds to perform the necessary file closing operations before terminating itself.

Parent

the parent process in the server sends a kill signal to all of its child processes and waits for all of them to terminate. Once all child processes have terminated, the parent process sends a kill signal to all connected clients. It then proceeds to perform the necessary file closing operations, unlinks the main FIFO, and terminates itself.

Client Side

Sends kill signals to server child process the necessary file closing operations are conducted, and the FIFOs are unlinked. Finally, the program terminates.

Semaphore

A single semaphore is utilized. It is locked (or "down") upon receiving each request from the client, ensuring that operations are serialized.

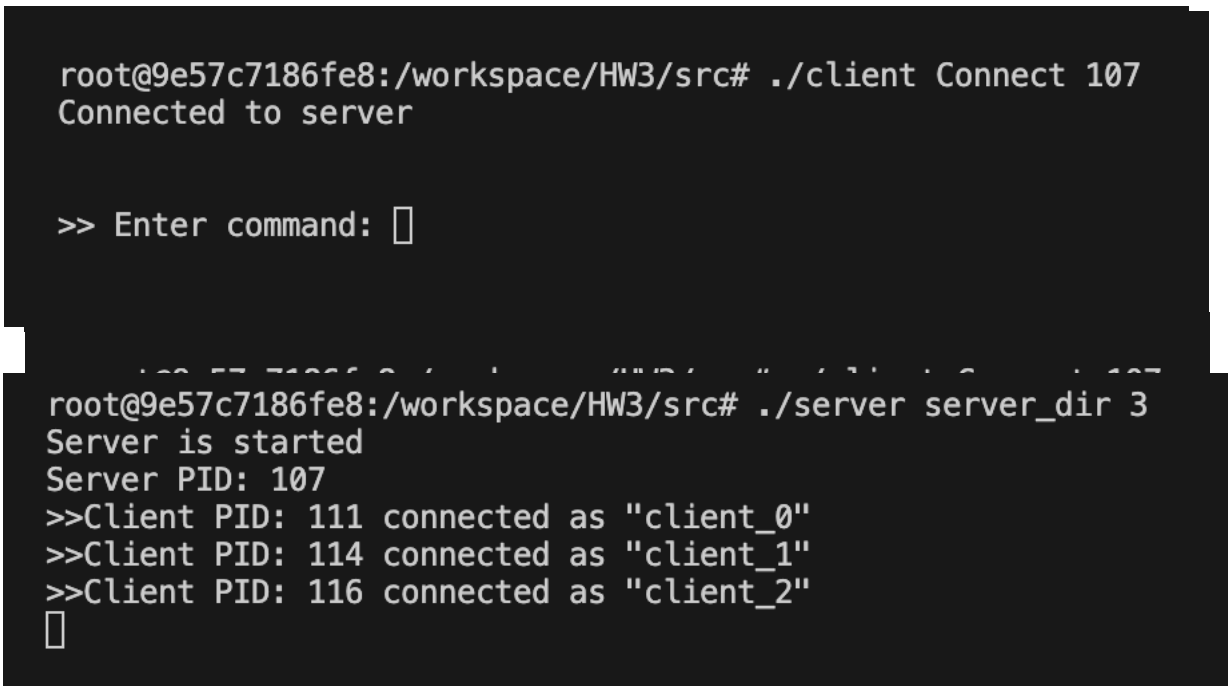
3. Tests and Results

Connection:

Success:

Wrong PID

Multiple Connection



Reject MAX Client

```
root@9e57c7186fe8:/workspace/HW3/src# ./client tryConnect 107
Server is full. You are rejected

root@9e57c7186fe8:/workspace/HW3/src#
```

```
root@9e57c7186fe8:/workspace/HW3/src# ./server server_dir 3
Server is started
Server PID: 107
>>Client PID: 111 connected as "client_0"
>>Client PID: 114 connected as "client_1"
>>Client PID: 116 connected as "client_2"
>>Client PID: 129 Server is full. You are rejected
```

Quee

```
root@9e57c7186fe8:/workspace/HW3/src# ./client Connect 217
You are in the queue
```

```
root@9e57c7186fe8:/workspace/HW3/src# make
gcc server.c -o server
gcc client.c -o client
root@9e57c7186fe8:/workspace/HW3/src# ./server server_dir 2
Server is started
Server PID: 217
>>Client PID: 220 connected as "client_0"
>>Client PID: 226 connected as "client_1"
>>Client PID: 229 Server is full. Connection request is added to the queue
```

One Client disconnected

```
root@9e57c7186fe8:/workspace/HW3/src# ./client Connect 217
You are in the queue

Connected to server


>> Enter command:
```

```
root@9e57c7186fe8:/workspace/HW3/src# make
gcc server.c -o server
gcc client.c -o client
root@9e57c7186fe8:/workspace/HW3/src# ./server server_dir 2
Server is started
Server PID: 217
>>Client PID: 220 connected as "client_0"
>>Client PID: 226 connected as "client_1"
>>Client PID: 229 Server is full. Connection request is added to the queue
Client PID: 226 -> QUIT
Quit
>>Client PID: 226 is disconnected
>>Client PID: 229 connected as "client_2"
```

Help

```
root@9e57c7186fe8:/workspace/HW3/src# ./client Connect 217
Connected to server


>> Enter command: help

     Available commands are:
           help <command#>
           list
           readF <filename> <line#>
           writeF <filename> <line#> "<string>"
           upload <filename>
           download <filename>
           archServer <tarfilename>
           quit
           killServer


>> Enter command:
```

Help with Command

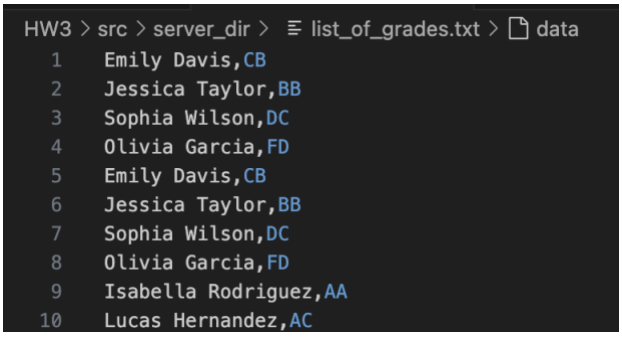
```
>> Enter command: help readF

     readF <filename> <line#>
          Read a file from the server
          <filename> is the name of the file to read
          <line#> is the line number to read (optional)

>> Enter command:
```

```
>> Enter command: help writeF

     writeF <filename> <line#> "<string>"
          Write a file to the server
          <filename> is the name of the file to write
          <line#> is the line number to write (optional)
          <string> is the string to write
```

List

```
>> Enter command: list

     list_of_grades.txt
     serverLog.log
```

readF with line number




```
>> Enter command: readF list_of_grades.txt 5

Emily Davis,CB

>> Enter command:
```

readF without line number

```
>> Enter command: readF list_of_grades.txt
Emily Davis,CB
Jessica Taylor,BB
Sophia Wilson,DC
Olivia Garcia,FD
Emily Davis,CB
Jessica Taylor,BB
Sophia Wilson,DC
Olivia Garcia,FD
Isabella Rodriguez,AA
Lucas Hernandez,AC
Alexander Gonzalez,AE
Ethan Young,AG
Jacob Allen,AI
Michael King,AK
Benjamin Green,BA
```

Invalid readF

```
>> Enter command: readF
Too few or too many arguments
>> Enter command: readF none.txt
No such file or directory
>> Enter command: readF ist_of_grades.txt 0
Invalid command Second parameter should be a number and greater than 0
>> Enter command: readF ist_of_grades.txt a
Invalid command Second parameter should be a number and greater than 0
>> Enter command: readF ist_of_grades.txt 5 hello
Too few or too many arguments
```

writeF with line number

HW3 > src > server_dir > list_of_grades.txt

```
1   Emily Davis,CB
2   Jessica Taylor,BB
3   Sophia Wilson,DC
4   Olivia Garcia,FD
5   i am the new line baby
6   Emily Davis,CB
7   Jessica Taylor,BB
8   Sophia Wilson,DC
9   Olivia Garcia,FD
10  Isabella Rodriguez,AA
11  Lucas Hernandez,AC
```

```
>> Enter command: writeF list_of_grades.txt 5 "i am the new line baby"
Success

>> Enter command:
```

writeF without line number

```
>> Enter command: writeF list_of_grades.txt "i am the end"
Success
```

```
19   Madison Harris,BG
20   Grace Walker,BI
21   Victoria Clark,CA
22   Ella Perez,CC
23   Nora Carter,CE
24   Camila Roberts,CG
25   Lauren Jackson,CI
26   Liam Thompson,CK
27   Oliver Martinez,DA
28   Brooklyn White,DC
29   Zoe Brooks,DE
30   i am the end
```

writeF Invalid

```
>> Enter command: writeF
Token count: 1
Too few or too many arguments

>> Enter command: writeF "selam"
Token count: 2
Too few or too many arguments

>> Enter command: writeF list_of_grades.txt
Token count: 2
Too few or too many arguments

>> Enter command: writeF list_of_grades.txt 5
Token count: 3
Invalid command

>> Enter command:
```

Upload

```
root@9e57c7186fe8:/workspace/HW3/src# ls -l data.csv
-rw-r--r-- 1 root root 100913007 May  5 02:54 data.csv
root@9e57c7186fe8:/workspace/HW3/src# ./client Connect 753
Connected to server


>> Enter command: upload data.csv
File uploaded successfully
100913007 bytes is uploaded

>> Enter command: quit
Success
Exiting...
root@9e57c7186fe8:/workspace/HW3/src# ls -l server_dir/data.csv
-rw-r--r-- 1 root root 100913007 May  5 03:07 server_dir/data.csv
root@9e57c7186fe8:/workspace/HW3/src#
```

Invalid Upload

```
>> Enter command: upload
Too few or too many arguments

>> Enter command: upload benYokum.txt
Failed to open file: No such file or directory

>> Enter command:
```

upload a file that is already in the server directory

```
>> Enter command: list

    list_of_grades.txt
    data.csv
    ne.txt
    serverLog.log

>> Enter command: upload data.csv
File uploaded successfully
100913007 bytes is uploaded

>> Enter command: list

    list_of_grades.txt
    data.csv
    ne.txt
    serverLog.log
    data(1).csv

>> Enter command:
```

download

```
root@9e57c7186fe8:/workspace/HW3/src# ls -l server_dir/kitap.pdf
-rw-r--r-- 1 root root 7260882 May  5 03:38 server_dir/kitap.pdf
root@9e57c7186fe8:/workspace/HW3/src# ./client Connect 1090
Connected to server


>> Enter command: download kitap.pdf
Download kitap.pdf (7260882 Byte) started
File downloaded successfully

>> Enter command: quit

Exiting...
root@9e57c7186fe8:/workspace/HW3/src# ls -l kitap.pdf
-rw-r--r-- 1 root root 7260882 May  5 04:05 kitap.pdf
root@9e57c7186fe8:/workspace/HW3/src#
```

download a file that is already in the client directory

```
root@9e57c7186fe8:/workspace/HW3/src# ls kitap*
kitap.pdf
root@9e57c7186fe8:/workspace/HW3/src# ./client Connect 1090
Connected to server


>> Enter command: download kitap.pdf
Download kitap.pdf(7260882 Byte) started
File downloaded successfully

>> Enter command: quit

Exiting...
root@9e57c7186fe8:/workspace/HW3/src# ls kitap*
kitap.pdf  kitap_1.pdf
root@9e57c7186fe8:/workspace/HW3/src#
```

Invaild download

```
root@9e57c7186fe8:/workspace/HW3/src# ./client Connect 1090
Connected to server


>> Enter command: list

    list_of_grades.txt
    kitap.pdf
    data.csv
    ne.txt
    serverLog.log

>> Enter command: download benYokum.log
No such file or directory

>> Enter command:
```

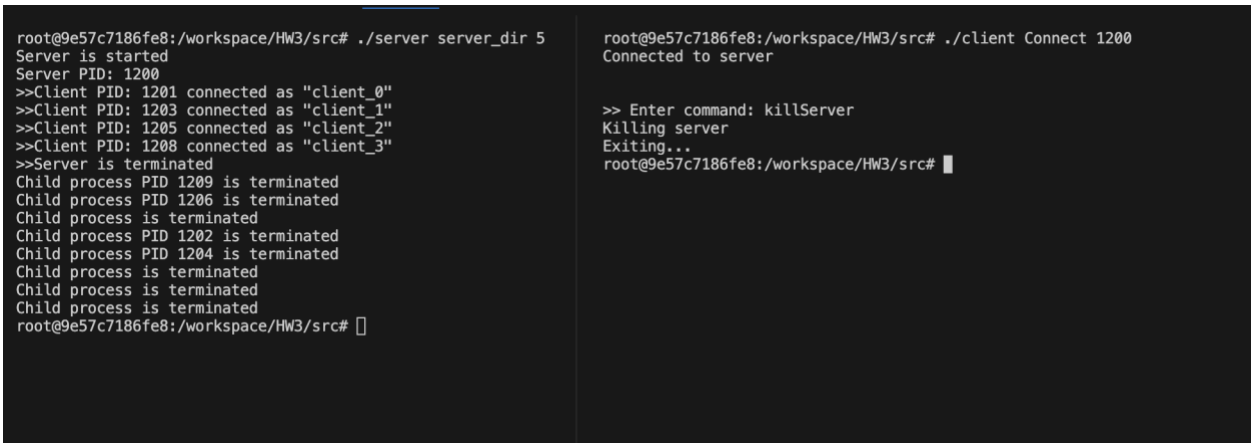
archServer

```
>> Enter command: archServer topla.tar


    list_of_grades.txt
    kitap.pdf
    ne.txt
    serverLog.log
Creating directory topla

Download list_of_grades.txt(537 Byte) started
File downloaded successfully

Download kitap.pdf(7260882 Byte) started
File downloaded successfully

Download ne.txt(146 Byte) started
File downloaded successfully

Download serverLog.log(13723 Byte) started
File downloaded successfully
Calling tar utility .. child PID 1173
child returned with SUCCESS...
Directory removed

>> Enter command:
```

Quit


root@9e57c7186fe8:/workspace/HW3/src# ./server server_dir 2
Server is started
Server PID: 1175
>>Client PID: 1179 connected as "client_0"
Client PID: 1179 -> QUIT
Quit
>>Client PID: 1179 is disconnected
root@9e57c7186fe8:/workspace/HW3/src# ./client Connect 1175
Connected to server
>> Enter command: quit
Exiting...
root@9e57c7186fe8:/workspace/HW3/src#

killServer


root@9e57c7186fe8:/workspace/HW3/src# ./server server_dir 5
Server is started
Server PID: 1200
>>Client PID: 1201 connected as "client_0"
>>Client PID: 1203 connected as "client_1"
>>Client PID: 1205 connected as "client_2"
>>Client PID: 1208 connected as "client_3"
>>Server is terminated
Child process PID 1209 is terminated
Child process PID 1206 is terminated
Child process is terminated
Child process PID 1202 is terminated
Child process PID 1204 is terminated
Child process is terminated
Child process is terminated
Child process is terminated
root@9e57c7186fe8:/workspace/HW3/src#
root@9e57c7186fe8:/workspace/HW3/src# ./client Connect 1200
Connected to server
>> Enter command: killServer
Killing server
Exiting...
root@9e57c7186fe8:/workspace/HW3/src#