System Programming
Midterm Project Report
Spring 2023

Barış Ayyıldız 1901042252

Client Side

Functions

The main function accepts two arguments: the first argument specifies whether the client wants to connect or try to connect to the server, and the second argument is the process ID of the server.

void generateRequest(char* buffer): This function is used to generate a Request structure based on the user's input. It creates a Request structure using malloc() and sets the pid, payloadSize, payload, and type fields based on the user's input. It uses strtok() to tokenize the user's input and extract the necessary information.

int createFifo(int pid): This function is used to create a named pipe (FIFO) for inter-process communication. It takes the process ID as input and generates a FIFO path based on the ID. It then creates the FIFO using mkfifo() and sets the appropriate permissions.

int isValidRequest(Request *req): This function checks whether a Request structure is valid or not. It returns 0 if the type field of the Request is empty and 1 otherwise.

The main function first checks whether the arguments are valid or not. If they are, it creates a named pipe for the client and sends a Connect or tryConnect request to the server, depending on the first argument. It then waits for a response from the server and prints the result. If the response is **CONNECTION_ESTABLISHED**, it means that the client has successfully connected to the server. If the response is anything else, it means that the server was unable to establish a connection with the client.

Finally, the main function reads commands from the user and sends them to the server using named pipes. It then waits for a response from the server and prints it. If the response is **ERROR**, it means that the server was unable to process the command. If the response is anything else, it means that the server has successfully processed the command.

Server Side

Global Variables

parent_pid: A global variable of type pid_t that stores the process ID of the parent process.
rootAddress: A character array of size 256 that stores the path to the server directory.

Functions

void writeLog(char* message): This function is used to write log messages to a log file named "logfile.txt". It takes a string message as input and writes it to the log file. The function also implements file locking using the flock() function to prevent concurrent writes to the log file.

void sigint_handler(int signum): This function is a signal handler for the SIGINT signal, which is sent to the process when the user presses Ctrl+C. The function sends the SIGTERM signal to all child processes and exits the parent process.

void server_send(Request req, char* message): This function is used to send a response message to a client through a FIFO (named pipe). It takes a Request structure and a string message as input. It creates a FIFO with a name that includes the process ID of the client and writes the message to the FIFO.

char* getManualText(Request req): This function returns a help message based on the client's request. It takes a Request structure as input and returns a string message. If the payloadSize is 0, it returns a list of available commands. Otherwise, it returns a description of the specified command.

void printManual(Request req): This function is used to print the help message to the client. It takes a Request structure as input, retrieves the help message using the getManualText() function, and sends it to the client using server_send() function.

void printListOfFiles(Request req): This function is used to retrieve the list of files in the server directory and send it to the client. It takes a Request structure as input and retrieves the list of files using the "ls" command. It then sends the list to the client using server_send() function.

bool isNumber(char *number): This function returns true if the given input is numeric else returns false

void readFileLine(Request req, int target): This function reads a file from the server's file system and sends its contents back to the client through a named pipe (FIFO). It creates a child process to execute the cat command and redirects its output to the named pipe. It waits for the child process to finish executing before returning. The function also uses file locking to ensure only one process writes to the named pipe at a time.

void writeFileEOF(Request req): This function appends the text specified in the req.payload[1] argument to the end of the file specified in req.payload[0]. It also uses a named pipe to send a response to the client. The function uses file locking to prevent multiple processes from writing to the same file simultaneously.

void writeFileLine(Request req): This function is used to write data to a specific line in a file. The function takes in a Request object that contains information about the file path, the line number, and the data to be written. The function opens the file, reads each line, and writes the data to the specified line number. The function also uses file locking to ensure that no other process can access the file while it is being modified. Finally, the original file is deleted, and the temporary file is renamed to the original file name.

void downloadFile(Request req): This function is used to download a file from the server. The function takes in a Request object that contains information about the file path. Server reads from the requested file in 1kb chunks and writes it into the clients folder. Both reading and writing operations are handled in the server.

void uploadFile(Request req): This function is used to upload a file to the server. The function takes in a Request object that contains information about the file path. Clients send the file to the server in 1kb chunks.

char* getProcessName(int pid, int (*clients)[2], int* clientsCounter, sem_t* clients_sem, sem_t* clientsCounter_sem): This function is used to retrieve the name of a process given its process ID (pid). The function takes in an integer pid, an array of client processes clients, the number of clients clientsCounter, and two semaphores clients_sem and clientsCounter_sem. The function searches through the clients array to find the client process with the given pid and returns its name.

void clientQuit(Request req, int* clientsCounter, int (*clients)[2], sem_t* clients_sem, sem_t* clientsCounter_sem, sem_t* isAvailable_sem): This function is used to handle a client quitting the server. The function takes in a Request object that contains information about the quitting client. The function removes the quitting client from the clients array and sends a success message to the client. The function also logs the client's quitting event and updates the number of clients in the clientsCounter variable. The function uses three semaphores to ensure that the critical section of code is executed atomically and avoids race conditions.

void handleRequest(Request req, int* clientsCounter, int (*clients)[2], sem_t* clients_sem, sem_t* clientsCounter_sem, sem_t* isAvailable_sem): This function is used to handle a client request. The function takes in a Request object, the number of clients clientsCounter, an array of client processes clients, and three semaphores clients_sem, clientsCounter_sem, and isAvailable_sem. The function first checks the type of the request and calls the appropriate function to handle the request. If the request is not recognized, the function sends an error message to the client. The function uses semaphores to ensure that the critical section of code is executed atomically and avoids race conditions.
In the main function **fd, fdClient, and fdLog** variables are file descriptors for the server, client, and log files, respectively. The clients variable is a two-dimensional integer array that stores information about the connected clients.

The **rootAddress** variable is a string that represents the root directory of the server. If the first command-line argument is **"Here" or "."**, then the root directory is set to an empty string. Otherwise, the root directory is set to the value of the first command-line argument with a forward slash appended to it.

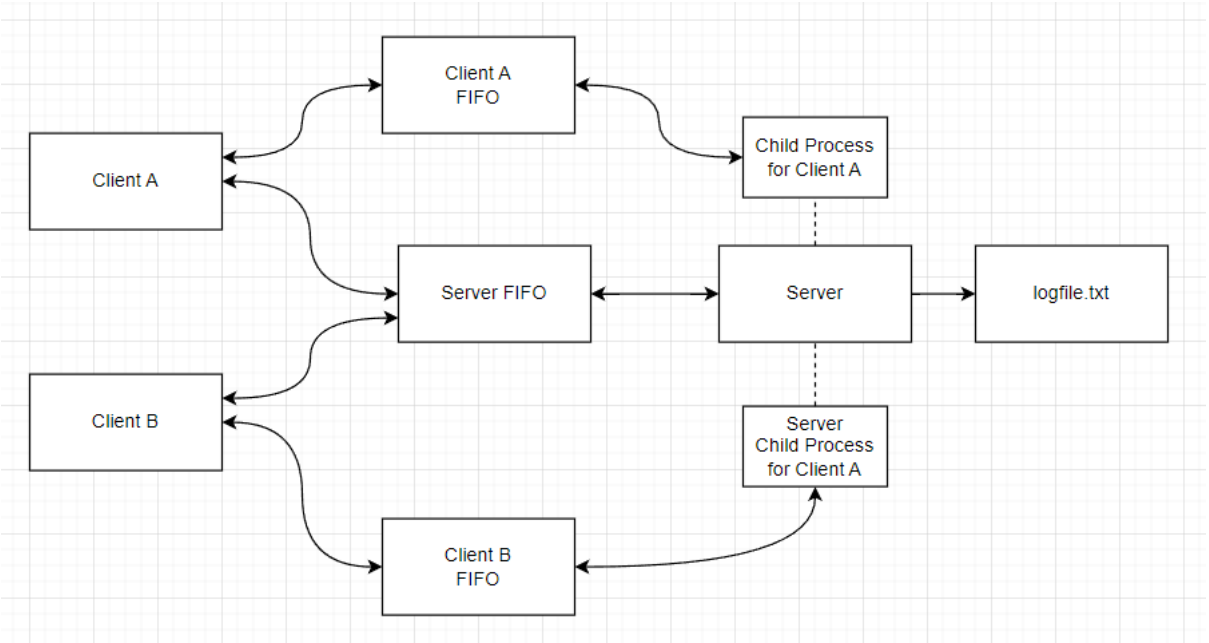
mkdir function creates a directory with the name specified in rootAddress with read, write, and execute permissions. Also a file called **logfile.txt** is created here. It holds the information of connected and disconnected clients.

MAX_NUMBER_OF_CLIENTS is an integer variable that represents the maximum number of clients that can be connected to the server. The value is obtained from the second command-line argument and is converted from a string to an integer using the atoi function.

The **clientsCounter**, **clients**, and **totalCounter** variables are pointers to shared memory areas. clientsCounter is used to keep track of the number of connected clients, clients is an array that stores information about the connected clients id and a counter value, and totalCounter keeps track of the total number of requests processed by the server. This is used to generate a label for the client.

The **clients_sem**, **clientsCounter_sem**, **isAvailable_sem**, **totalCounter_sem**, request_sem are semaphores. clients_sem locks any process that wants to access the clients array when it's used by another process. clientsCounter_sem does the same thing for clientsCounter and totalCounter_sem for the totalCounter. **isAvailable_sem** semaphore is initialized with the MAX_NUMBER_OF_CLINETS. Whenever a client is connected to the server we decrement this semaphore's value by one and whenever a client is disconnected from the client it gets incremented again by one. So when the server's capacity is full isAvailable_sem becomes 0 and it locks the client that wants to access and it has to wait until a client is disconnected from the server. **This prevents the race condition**.

Architecture of my System



In my architecture there is a named pipe called Server FIFO. This fifo accepts **Connect** and **tryConnect** requests and server reads from it. If the server can handle that client, it sends **CONNECTION_ESTABLISHED** message to that client. At that point the server creates a new process with fork system call. That child process handles user requests and writes to client fifos.

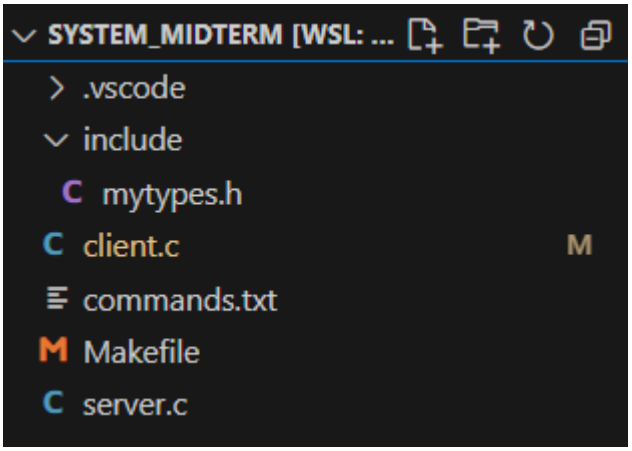
If a client sends a tryConnect request and if the server is out of capacity, the server sends **SERVER_IS_OUT_OF_CAPACITY** and the client terminates.

When a user wants to quit, it sends a quit message to its fifo. Child process takes it and kills itself. When a child process dies, the server decrements the counter by one. When the server gets a killServer, it kills all of its child processes and itself with **raise(SIGINT)** and this code calls **sigint_handler** function.

Server also writes all the Connect, tryConnect and quit requests and the logfile.txt. When the server needs to write that file it first locks it with **flock** function, so no other process can write it at that moment, and unlocks it when it’s finished.

Test Cases

This is the folder structure of my project. mytypes.h contains some macros an a struct named Request and Block




```c
typedef struct Request {
  int pid;
  char type[BUF_SIZE];
  char payload[3][BUF_SIZE];
  int payloadSize;
} Request;
```

```c
You, 1 hour ago | 1 author (You)
typedef struct Block {
  int status;
  char buffer[BUF_SIZE];
} Block;
```

Request struct has this shape. pid keeps the process id, type is the type of the request. It can be **Connect, tryConnect, help, list, readF, writeT, upload or download**. payload is the extra parameters. For example if the input is **"help writeT"**, payload becomes **{writeT}** and in that case **payloadSize** becomes 1.

Block struct is used when downloading or uploading a file.

Now I will call **make server** to compile server.c and call **make client** to compile client.c.

```
barisayyildiz@DESKTOP-2V8A48Q:~/system_midterm$ make server
gcc server.c -o biboServer
barisayyildiz@DESKTOP-2V8A48Q:~/system_midterm$
```

```
barisayyildiz@DESKTOP-2V8A48Q:~/system_midterm$ make client
gcc client.c -o biboClient
barisayyildiz@DESKTOP-2V8A48Q:~/system_midterm$
```

I will run the server with the capacity of 2 and with the dirname dir.

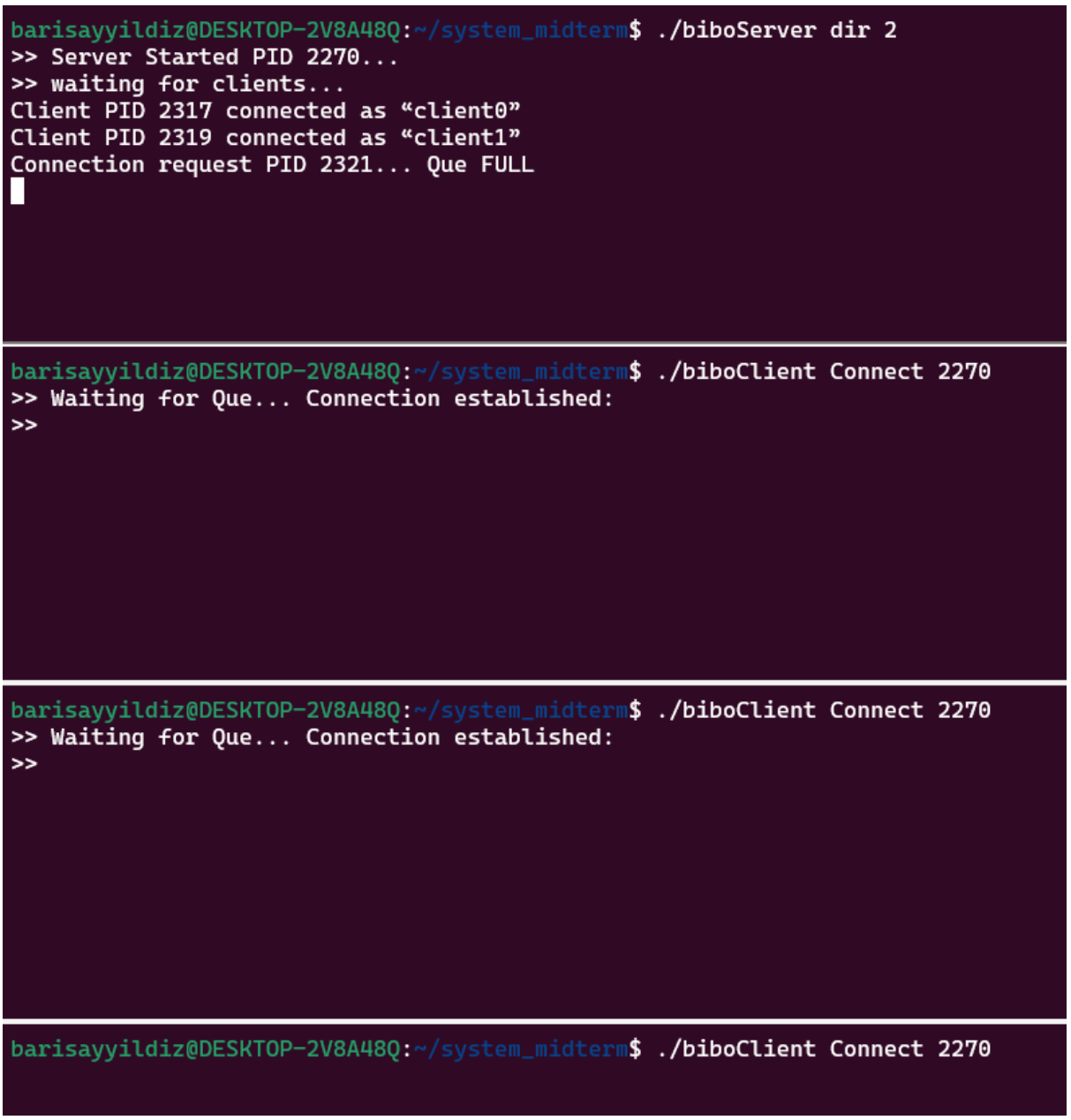
```
barisayyildiz@DESKTOP-2V8A48Q:~/system_midterm$ ./biboServer dir 2
>> Server Started PID 2270...
>> waiting for clients...
```

As you can see server created a directory named dir, its process id is 2270 and it created a fifo file for the server with the format **"biboServer_{PROCESS_ID}"**

```
barisayyildiz@DESKTOP-2V8A48Q:/tmp$ ls -d bibo*
biboServer_2270
barisayyildiz@DESKTOP-2V8A48Q:/tmp$
```

```
barisayyildiz@DESKTOP-2V8A48Q:/tmp$ ps -C biboServer
  PID TTY          TIME CMD
 2270 tty1     00:00:00 biboServer
barisayyildiz@DESKTOP-2V8A48Q:/tmp$
```

Now I will try to create 3 processes:



As you can see two of them successfully connected with the names “**client0**” and “**client1**”. And the third client couldn’t connect because there was no room for it.

And you can see that there are 3 biboServer processes, 1 for the main process and 2 for the children processes. And again we have 2 child processes.

```
barisayyildiz@DESKTOP-2V8A48Q:/tmp$ ps -C biboServer
  PID TTY          TIME CMD
 3070 tty1     00:00:00 biboServer
 3072 tty1     00:00:00 biboServer
 3074 tty1     00:00:00 biboServer
barisayyildiz@DESKTOP-2V8A48Q:/tmp$ ps -C biboClient
  PID TTY          TIME CMD
 3071 tty6     00:00:00 biboClient
 3073 tty2     00:00:00 biboClient
barisayyildiz@DESKTOP-2V8A48Q:/tmp$
```

After that I quit client0

```
barisayyildiz@DESKTOP-2V8A48Q:~/system_midterm$ ./biboServer dir 2
>> Server Started PID 2270...
>> waiting for clients...
Client PID 2317 connected as “client0”
Client PID 2319 connected as “client1”
Connection request PID 2321... Que FULL
client0 disconnected...
Client PID 2321 connected as “client2”
```

```
barisayyildiz@DESKTOP-2V8A48Q:~/system_midterm$ ./biboClient Connect 2270
>> Waiting for Que... Connection established:
>> quit
barisayyildiz@DESKTOP-2V8A48Q:~/system_midterm$
```

```
barisayyildiz@DESKTOP-2V8A48Q:~/system_midterm$ ./biboClient Connect 2270
>> Waiting for Que... Connection established:
>>
```

```
barisayyildiz@DESKTOP-2V8A48Q:~/system_midterm$ ./biboClient Connect 2270
>> Waiting for Que... Connection established:
>>
```

And client2 is connected to the server.

```
barisayyildiz@DESKTOP-2V8A48Q:~/system_midterm$ ./biboServer dir 2
>> Server Started PID 2270...
>> waiting for clients...
Client PID 2317 connected as “client0”
Client PID 2319 connected as “client1”
Connection request PID 2321... Que FULL
client0 disconnected...
Client PID 2321 connected as “client2”
Try Connect request PID 2323... Que FULL... exiting
```

```
barisayyildiz@DESKTOP-2V8A48Q:~/system_midterm$ ./biboClient Connect 2270
>> Waiting for Que... Connection established:
>> quit
barisayyildiz@DESKTOP-2V8A48Q:~/system_midterm$ ./biboClient tryConnect 2270
>> Waiting for Que... server is out of capacity, closing....
barisayyildiz@DESKTOP-2V8A48Q:~/system_midterm$
```

A client used tryConnect but since the server was out of capacity it couldn’t connect and got terminated.

Now let’s test some of the commands

Commands:

help

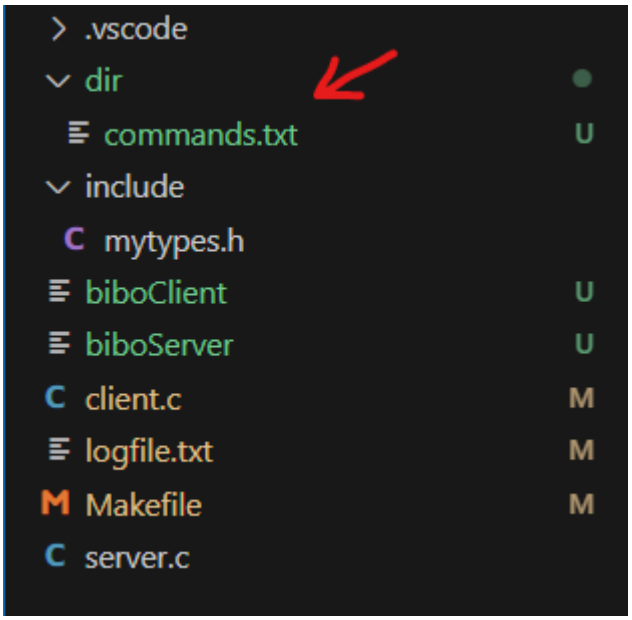
```
>> help

        Available comments are :
                help, list, readF, writeT, upload, download, quit, killServer

>>
```

```
>> help writeT

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

upload

commands.txt from root directory moved to folder dir

```
>> upload commands.txt

file uploaded successfully

>>
```



As you can see commands.txt is in directory dir

But for large files, uploaded files get a little corrupted. For example

```
>> upload server.c

file uploaded successfully
>>
```

Server.c under dir directory

```
761     close(fd);
762     munmap(clients, sizeof(int) * 5);
763     munmap(clientsCounter, sizeof(int));
764
765     return 0;
766   }
767
768
769   ildPid);
770                 sprintf(fifoPath, fifoPathTemplate, req.pid);
771                 fdClient = open(fifoPath, O_RDONLY);
772                 if(read(fdClient, &req, sizeof(Request)) == -1){
773                   perror("read");
774                   exit(EXIT_FAILURE);
775                 }
776                 close(fdClient);
777                 // printf("child_process req.type : %s\n", req.type);
778
779                 if(strcmp(req.type, "upload") == 0){
780                   uploadFile(req);
781           0){
782                   uploadFile(req);
783         6DEL
```

Actually server.c ends at line 766 but for some reason I get this extra part. It always fails at the last chunk of data. From my understanding if the last buffer's 1KB is not fully allocated it prints some random stuff.

And for files that is larger than 10MB

```
-rw-r--r-- 1 barisayyildiz barisayyildiz 11919600 May 16 20:45 commands_10MB.txt
```

```
>> upload commands_10MB.txt

file uploaded successfully
>>
```

Original

```
866877    upload <file>
866878    download <file>
866879    quit
866880    killServer
866881
```

Uploaded

```
866934    list
866935    readF <file> <line #>
866936    writeT <file> <line #> <string>
866937    upload <file>
866938    download <fi
```

In that case it didn’t write the last chunk of data at all.

readF

```
>> readF commands.txt

help
list
readF <file> <line #>
writeT <file> <line #> <string>
upload <file>
download <file>
quit
killServer
```

```
>> readF commands.txt 3

readF <file> <line #>
```

```
>> readF commands.txt 1000

requested file is smaller than the target size...
>>
```

writeF

```
>> writeT commands.txt helllooooooooo

text appended...
>>
```

dir > commands.txt

```
1  help
2  list
3  readF <file> <line #>
4  writeT <file> <line #> <string>
5  upload <file>
6  download <file>
7  quit
8  killServer
9  hellllooooooooo
```

```
>> writeT commands.txt 3 hellloo_woorrrllddd

text appended...
>>
```

dir > commands.txt

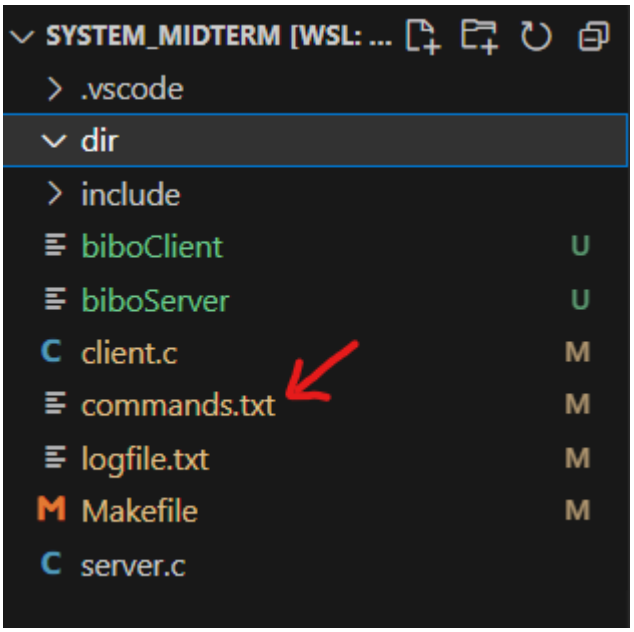
```
1  help
2  list
3  hellloo_woorrrllddd
4  writeT <file> <line #> <string>
5  upload <file>
6  download <file>
7  quit
8  killServer
9  hellllooooooooo
```

list

```
>> list

commands.txt
...
>>
```

download

```
>> download commands.txt

file downloaded successfully

>>
```



commands.txt is back to the root directory.

Now let’s download commands_10MB.txt

```
barisayyildiz@DESKTOP-2V8A48Q:~/system_midterm$ ./biboServer dir 2
>> Server Started PID 18860...
>> waiting for clients...
Client PID 18861 connected as “client0”
Client PID 18863 connected as “client1”
File copied successfully.
File copied successfully.
```

```
>> ^C
barisayyildiz@DESKTOP-2V8A48Q:~/system_midterm$ make client
gcc client.c -o biboClient
barisayyildiz@DESKTOP-2V8A48Q:~/system_midterm$ ./biboClient Connect 18860
>> Waiting for Que... Connection established:
>> download commands2_10MB.txt

file downloaded successfully..

>>
```

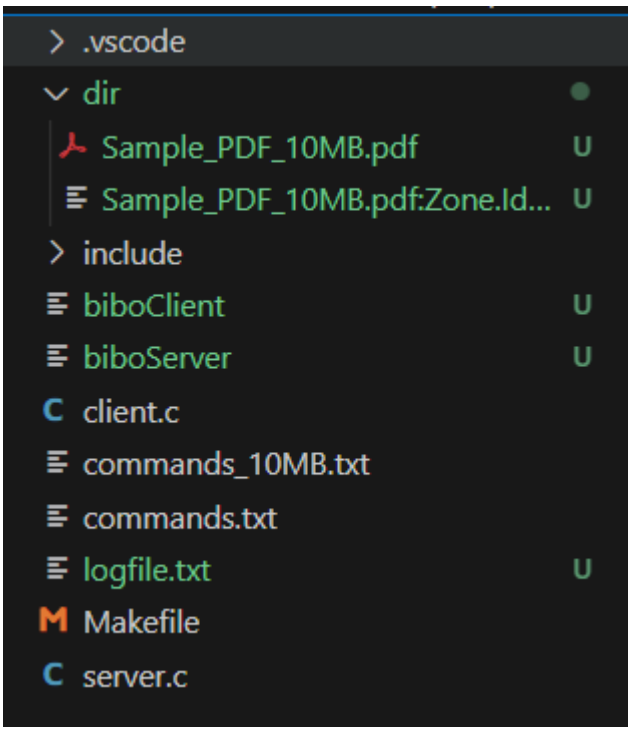
```
file downloaded successfully..

>> ^C
barisayyildiz@DESKTOP-2V8A48Q:~/system_midterm$ ./biboClient Connect 18860
>> Waiting for Que... Connection established:
>> download commands2_10MB.txt

file downloaded successfully..

>>
```

I have downloaded the same file at the same time but since I use file locks, there was no race condition and I was able to download the file without any corruption

Downloaded:

```
866877  upload <file>
866878  download <file>
866879  quit
866880  killServer
866881
```

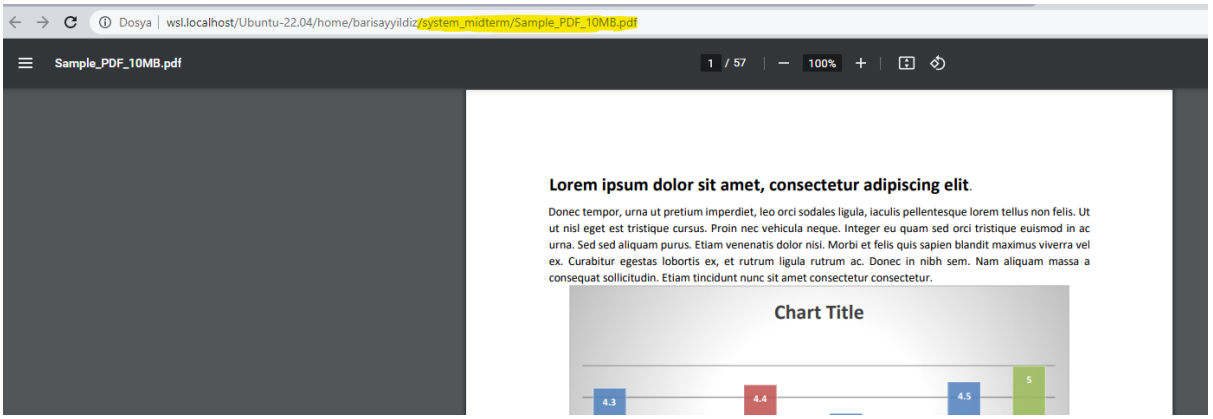
Now let’s download this pdf file




```
barisayyildiz@DESKTOP-2V8A48Q:~/system_midterm$ make server
gcc server.c -o biboServer
barisayyildiz@DESKTOP-2V8A48Q:~/system_midterm$ ./biboServer dir 2
>> Server Started PID 22785...
>> waiting for clients...
Client PID 22793 connected as “client0”
File copied successfully.
```

```
barisayyildiz@DESKTOP-2V8A48Q:~/system_midterm$ make client
gcc client.c -o biboClient
barisayyildiz@DESKTOP-2V8A48Q:~/system_midterm$ ./biboClient Connect 22785
>> Waiting for Que... Connection established:
>> download Sample_PDF_10MB.pdf

file downloaded successfully..

>>
```

As you can see we are able to open the pdf file from the root directory

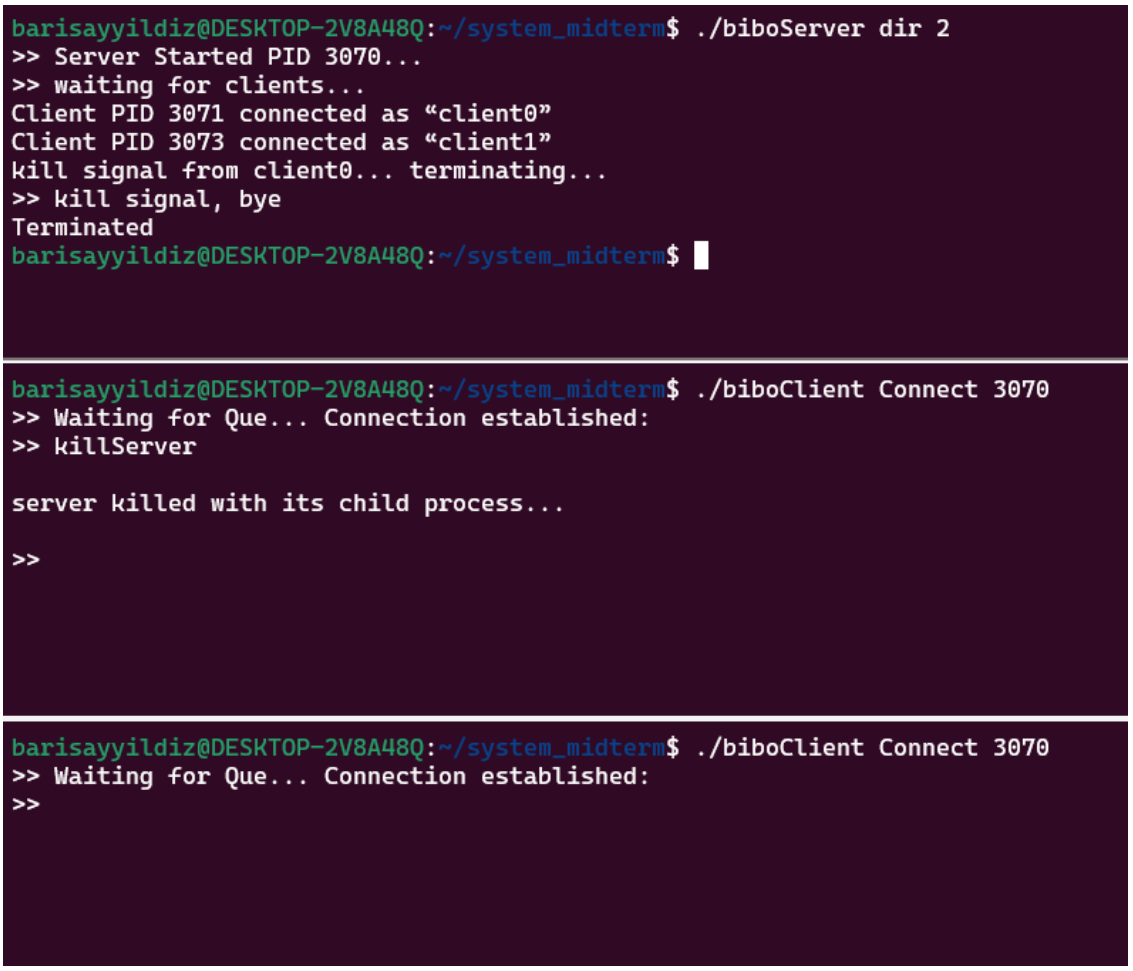


quit

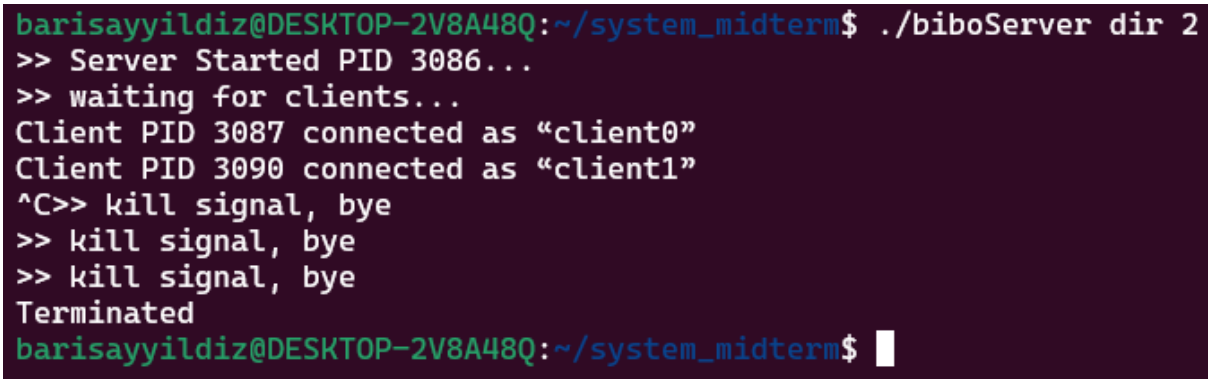
```
barisayyildiz@DESKTOP-2V8A48Q:~/system_midterm$ ./biboServer dir 2
>> Server Started PID 3018...
>> waiting for clients...
Client PID 3019 connected as “client0”
Client PID 3045 connected as “client1”
client1 disconnected...
Client PID 3047 connected as “client2”
client0 disconnected...
```

```
        Available comments are :
                help, list, readF, writeT, upload, download, quit, killServer

>> help

        Available comments are :
                help, list, readF, writeT, upload, download, quit, killServer

>> quit
barisayyildiz@DESKTOP-2V8A48Q:~/system_midterm$
```

killServer



ctrl+c in server



Logfile

```
Client PID 4124 connected as “client0”
Client PID 4130 connected as “client1”
client0 disconnected...
Client PID 4224 connected as “client2”
client2 disconnected...
Client PID 4343 connected as “client3”       You, 56 mi
kill signal, bye
kill signal, bye
kill signal, bye
Client PID 4386 connected as “client0”
kill signal, bye
kill signal, bye
Client PID 4423 connected as “client0”
Try Connect request PID 4425... Que FULL... exiting
client0 disconnected...
kill signal, bye
kill signal, bye
Client PID 4429 connected as “client0”
Client PID 4431 connected as “client1”
Connection request PID 4433... Que FULL
client0 disconnected...
Client PID 4433 connected as “client2”
client2 disconnected...
kill signal, bye
kill signal, bye
Client PID 18425 connected as “client0”
client0 disconnected...
kill signal, bye
Client PID 18463 connected as “client0”
```

How to run:

make server
	This will create executable biboServer
make client
	This will create executable biboClient
make clear
	This will remove biboServer and biboClient

./biboServer <dirname> <max #of client>
./biboClient ServerPID

Example:

./biboServer dir 2
./biboServer Here 5
./biboServer . 4

Here and . will not create a directory for server

./biboClient Connect <PID>
./biboClient tryConnect <PID>