CSE 344
System Programming
Report

Yavuz Selim İkizler
1901042617

Code Design

General Code Structures:

The program consists of two parts, server and client. In the server part, named pipes are created (server reads data from the client, client writes data to the client) .Data communication occurs for each client within the client_handler function, within a while loop. The client and server exchange data within this while loop.The function is called for each thread created and joined in the main function. When creating clients, the server does not receive requests, and there is no ID or maximum client control.

Client:

```
void sigintHandler(int sig)
```

The program sets up a signal handler for the SIGINT signal (triggered by pressing Ctrl+C). When the signal is received, the sigintHandler function is called. In the handler, it writes a "close" message to the client fifo using the client_pipe_fd file descriptor and prints the value of client_pipe_fd. Then, it exits the program.

```
int main(int argc, char *argv[])
```

The program enters a loop where it repeatedly prompts the user for commands, sends the commands to the server via the server fifo, and receives and processes responses from the server.

The program uses fgets to read the user's input command from the standard input. It removes the newline character from the end of the command and writes the command to the server fifo using the write function. If the write operation fails, an error message is displayed, and the loop is exited.

After sending the command, the program attempts to read the response from the server using the read function. If the read operation fails or returns a non-positive value, it displays a connection failure message. If the response is "quit," the program sets the is_quit flag to 1 to exit the loop. Otherwise, it prints the response to the standard output.

Once the loop is exited, the program closes the client and server fifo using the close function and returns 0 to indicate successful execution.

Server:

The server code you described starts by creating a server directory using the name obtained from the command line arguments. It checks if the named pipes (FIFOs) exist or not, and if they don't, it creates them. The code then enters a loop to create threads.The loop Iterates over the range of max_clients.
pthread_create(&threads[i], NULL, client_handler, &params). Creates a new thread and passes the address of the client_handler function as the thread routine. The &params argument is a pointer to the params struct, which is passed as an argument to client_handler.
&threads[i] stores the thread ID for the newly created thread.
client_handler is the function that will be executed in the new thread.
&params passes the address of params as an argument to client_handler.
If pthread_create fails, an error message is printed, and the program returns with an error code.

```
splitString(buffer, &numTokens);
```

The splitString function takes a string as a parameter. It uses strtok to split the string into tokens based on a delimiter and stores the split strings in a char pointer array. Finally, it returns the array containing the split strings.

```
void* client_handler(void* args)
```

It is the function where the commands received from the clients are read, processed, and returned. It takes a void pointer as a parameter and then casts this pointer to a ThreadParams structure. The params variable in the structure contains the dirname and max_client variables obtained from the argv in the main function. At the beginning of the function, the dir_name is assigned to a char pointer, which will be used as the server directory variable later on. Then, separate server read and client write descriptors are opened for each client to perform read and write operations, and data exchange within the loop is done using these descriptors. By using a sem_wait call before entering the while loop and a sem_post call at the end of the function, a race condition between clients is prevented.

Operations:

List:

The code snippet provided opens the current directory specified by the command-line argument argv[1]. If the directory fails to open, an error message is displayed and the program exits.

Next, the code reads the entries in the directory using readdir in a loop. It skips the entries for the current directory (.) and the parent directory (..). For each valid entry, it concatenates the entry name with a newline character and appends it to the response string. This creates a directory listing with each entry on a new line.

After processing all the directory entries, the code writes the response string, containing the directory listing, to the client using the file descriptor client_write_fd.

Finally, the code closes the directory using closedir.

WriteT:

It retrieves the file name and the line number from the tokens array obtained earlier. The file name is stored in tokens[1], and the line number is converted from a string to an integer using atoi(tokens[2]).

It constructs the file path by copying the directory name specified in dir_name to file_path, and then appends the file name to the end of the path using strcat.

It attempts to open the file in "r+" mode. If the file doesn't exist, it tries to create it using "w+" mode. If opening or creating the file fails, an error message is displayed, and the program exits.

If the number of tokens is 4 (indicating a line replacement operation), it reads the file line by line until it reaches the desired line number. If the line number is found, it replaces the line with the string from tokens[3] by seeking back to the beginning of the line and using fprintf to overwrite it. If the line number exceeds the current number of lines in the file, it appends the string at the end using fseek and fprintf.

If the number of tokens is not 4 (indicating an append operation), it appends the string from tokens[2] at the end of the file using fseek and fprintf.

After performing the file write operation, the file is closed using fclose.

A response message is constructed as "write operation done" using snprintf and stored in the response array.

The response message is written to the client using write and the file descriptor client_write_fd. If the write operation fails, an error message is displayed, and the program exits.

ReadF:

The code initializes variables, including file_name (the name of the file to read), and num (the line number to retrieve).

It creates a buffer file_path to store the complete file path by concatenating the directory name from dir_name and the file name from file_name.

It attempts to open the file using fopen and checks if the file is successfully opened. If not, it prints an error message and exits.

Inside the while loop, it reads each line of the file using fgets until either the specified line is found or the end of the file is reached.

If the current line number matches the desired line number (num), it prepares a response by copying the line into response. It then writes the response to the client using the file descriptor client_write_fd. Finally, it cleans up allocated memory and breaks out of the loop.

If the desired line is not found (line_exist is 0), it prepares a response by copying the entire file content into newstr. It then writes the response to the client.

After the loop ends, if the desired line was not found, it prepares a response by copying the content of newstr (the entire file) into response and writes it to the client.

quit:

The "quit" command sends the message to client "quit". After receiving the message, the client exits the loop.

Help:

The "help" command sends the available commands as a message to the client.

Download:

The "download" command concatenates the string obtained from the dir_name with the string received from the client using the "strcat" function. It then opens the file based on this path and reads it line by line within a while loop. After reading each line, it writes the read strings to the file opened with the string received from the client. Finally, it sends a feedback message to the client indicating the number of bytes read.

Upload:

The "upload" command performs the opposite operation of "download." It reads the file from the client and creates a file in the server's directory to write the contents.

Test Outputs

Connect:

```
mrmiles001@LAPTOP-TCVBO9A5:/mnt/c/Users/yavuz/Desktop/1901042617_system_midterm$ ./biboServer deneme 3
Server Started PID 506...
Waiting for clients...
Client PID 508 connected as "client01"
```

```
fd:4
mrmiles001@LAPTOP-TCVBO9A5:/mnt/c/Users/yavuz/Desktop/1901042617_system_midterm$ ./biboClient tryConnect 500
Connected to server (PID: 506). Enter your commands:
```

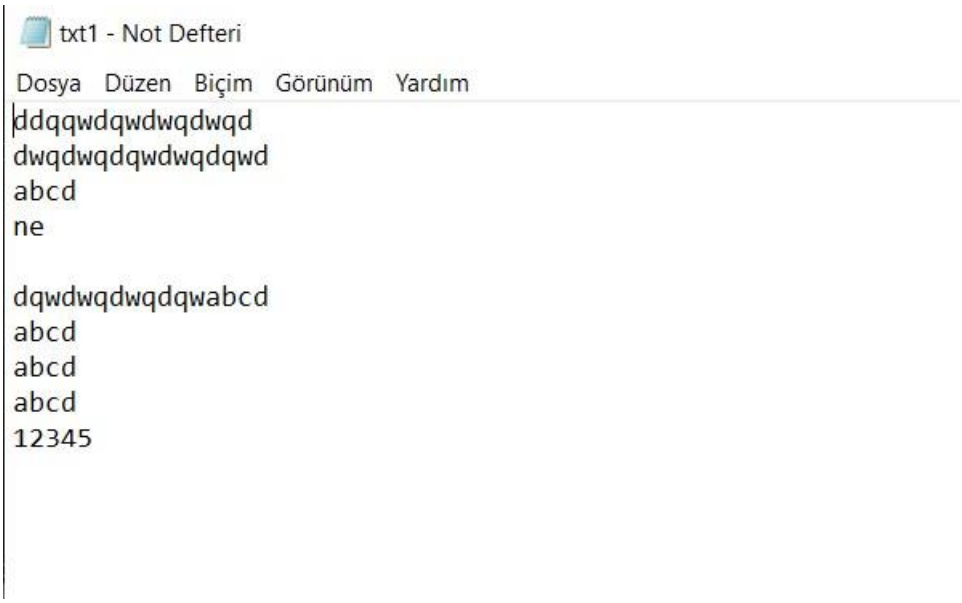
List:

```
Connected to server (PID: 431). Enter your commands:
> list
txt1.txt
txt2.txt
```

Help:

```
> help
list
 readF
 writeT
 upload
 download
 killServer
```

WriteT:




```
writeT txt1.txt newstr
```

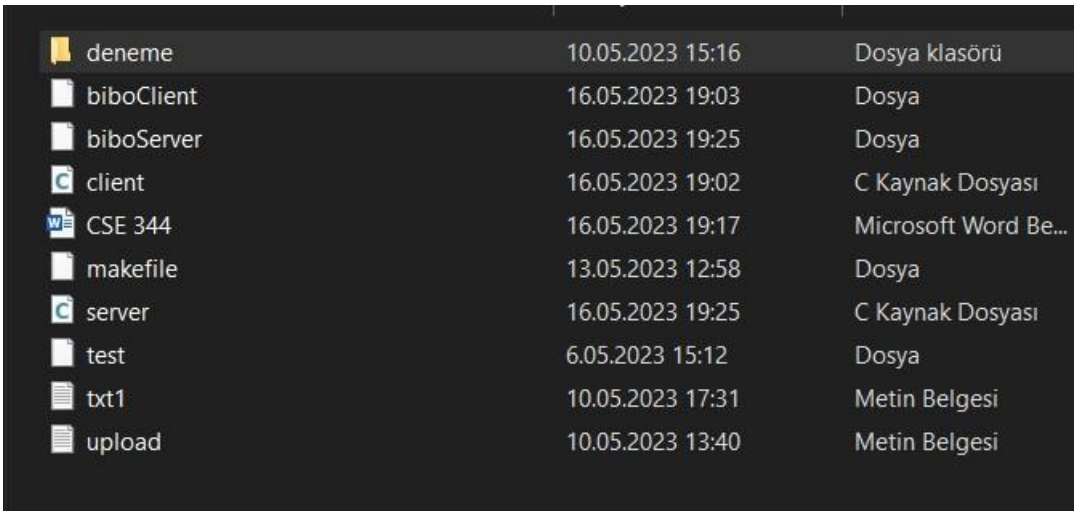
```
ddqqwdqwdwqdwqd
dwqdwqdqwdwqdqwd
abcd
ne

dqwdwqdwqdqwabcd
abcd
abcd
abcd
12345
newstr
```

```
> writeT txt1.txt 4 newstr2
write operation done
```

ReadF:

```
> readF txt1.txt 3
abcd
```

Upload:



deneme	10.05.2023 15:16	Dosya klasörü
biboClient	16.05.2023 19:03	Dosya
biboServer	16.05.2023 19:25	Dosya
client	16.05.2023 19:02	C Kaynak Dosyası
CSE 344	16.05.2023 19:17	Microsoft Word Be...
makefile	13.05.2023 12:58	Dosya
server	16.05.2023 19:25	C Kaynak Dosyası
test	6.05.2023 15:12	Dosya
txt1	10.05.2023 17:31	Metin Belgesi
upload	10.05.2023 13:40	Metin Belgesi

```
> upload upload.txt
47 bytes transferred >
```

txt1	15.05.2023 14:41	Metin Belgesi
txt2	7.05.2023 15:43	Metin Belgesi
upload	16.05.2023 20:24	Metin Belgesi

Download:

```
> download txt1.txt
90 bytes transferred
```

deneme	16.05.2023 20:24	Dosya klasörü
biboClient	16.05.2023 20:29	Dosya
biboServer	16.05.2023 19:25	Dosya
client	16.05.2023 20:29	C Kaynak Dosyası
CSE 344	16.05.2023 19:17	Microsoft Word Be...
makefile	13.05.2023 12:58	Dosya
server	16.05.2023 19:25	C Kaynak Dosyası
test	6.05.2023 15:12	Dosya
txt1	16.05.2023 20:35	Metin Belgesi
upload	10.05.2023 13:40	Metin Belgesi

Quit:

```
> quit
mrmiles001@LAPTOP-TCV8O9A5:/mnt/c/Users/yavuz/Desktop/1901042617_system_midterm$
```

Kill Signal:

Server:

```
Server Started PID 496...
Waiting for clients...
^C
Kill signal received. Terminating...
```

Client:

```
Connected to server (PID: 474). Enter your commands:
> ^C
```

Multiple Client: