CSE 344 – Midterm Report

Ersel Celal Eren – 1901042673

In this homework, we are asked to design a server-client logic. To handle this task, I followed the path below:

First, I created two different files ‘server.c’ and ‘client.c’. This two program or process communicates through FIFOs(named pipe). According to my design, there is a ***server_fifo*** that all client use for connection request. At server side, server reads PID of clients, checks availability of connection in terms of server capacity. Meanwhile also client specific FIFOs are created. For each client, there are two FIFO, one for data transfer from server to client other is data transfer. Each client has its own FIFO with name **‘server_to_client_fifo_1234’** and **‘client_to_server_fifo_1234’**. Here 1234 is PID of client. At server side after controls child process is being created for connected client and then this child process handles requests of client. This is the overview of my design.

Before explaining more detailed I want to point out that I could not fulfill all the required features in the PDF. My missing features are these : writeT, upload, download and log files. Now I am going to explain in detail from code.

---

server.c

I define and decleare a struct for properties of server. Also implement a queue structure to hold waiting clients. There is an another struct definition too.

```c
typedef struct {
    int max_clients;
    pid_t* connected_clients;
    int num_children;
    int empty_slot_count;
    pid_t server_pid;
    char directory_path[100];

} ServerStruct;

ServerStruct *server;
```

I use shared memory to hold ServerStruct. It gives me flexiblity when I am calling function from child processes. After allocation in create_shared_struct(…) function, I initialized values of server. I create a directory with the given path in command-line argument. Third command-line argument is assigned as capacity of server.

```c
//-------------------- Create Server Struct -----------------//
server = create_shared_struct(atoi(argv[2]));
if (server == NULL) {
    return 1;
}
server->max_clients = atoi(argv[2]);
server->empty_slot_count = server->max_clients;
server->num_children = 0;
server->server_pid = getpid();
strncpy(server->directory_path, argv[1], sizeof(server->directory_path) - 1);
server->directory_path[sizeof(server->directory_path) - 1] = '\0';
// Check if the specified directory exists, create it if it doesn't
struct stat st;
if (stat(server->directory_path, &st) == -1) {
    // Directory does not exist, create it
    if (mkdir(server->directory_path, 0700) == -1) {
        perror("Error creating directory");
        return 1;
    }
    printf("Directory '%s' created.\n", server->directory_path);
} else {
    // Directory already exists
    if (!S_ISDIR(st.st_mode)) {
        printf("Specified dirname is not a directory.\n");
        return 1;
    }
    printf("Directory '%s' already exists.\n", server->directory_path);
}
```

After creating server struct, I creat FIFO for server. This FIFO gets connection requests.

```c
//>>>>>>>>>>>>>>>>>>>>>>>> Server FIFO <<<<<<<<<<<<<<<<<<<<<<<<<<<
pid_t server_pid = getpid();
printf("Server PID: %d || Client Capacity: %d \n", server->server_pid,server->empty_slot_count);
mkfifo(SERVER_FIFO, 0666); // Create the server FIFO
int server_fifo_fd = open(SERVER_FIFO, O_RDONLY | O_NONBLOCK); // Open the server FIFO
```

In a while loop, I first check queue. If there is a client waiting, first client at queue will connect. Send_notificiation() function sends three different integer value: 0 for server is full, 1 for you are in queue, 2 for you are connected. Client waits this response from server after it tries to connect server. If client gets 0, it terminates program. If client gets 1, wait until another client is done. If it gets 2, starts giving command to server.

```c
while (1) {

    if(!isEmpty(&q) && server->empty_slot_count >0){ //if there is a client at queue and there is an empty slot
        client_pid = dequeue(&q);
        printf("Client %d is dequeued\n", client_pid);
        send_notification(client_pid, 2);
    }
```

If there is no client in queue or there is no empty slot for new client, it goes to else block. In these code block, It reads a Client PID from server_fifo. If there is data written into that FIFO, it checks slots in add_client() function. This function returns a boolean. Client also writes connection_mode to fifo. 0 for “connect”, 1 for “tryConnect”. If there is no space and clients wants “connect”, 0 will be returned as response. If there is no space and clients wants “tryConnect”, 1 will be returned as response and client will be added to queue. Regardless of connection_mode, if

there is a space in server, client will be connected and 2 will be returned as response which means successfull response.

```
else{
    bytes_read = read(server_fifo_fd, &client_pid, sizeof(client_pid));

    if(bytes_read > 0){
        read(server_fifo_fd, &connection_mode, sizeof(connection_mode));

        bool can_connect = add_client(client_pid); // Check if the client can be connected

        // If the client wants to connect immediately and the server is full,
        // close the connection and continue the loop
        if (connection_mode == 0 && !can_connect) {
            printf("Client %d could not connect. Server is full.\n", client_pid);
            send_notification(client_pid, 0); // Notify the client that the server is full
            continue;
        }

        // If the client wants to wait in the queue, add the client to the waiting queue
        if (connection_mode == 1 && !can_connect) {
            printf("Client %d is waiting in the queue.\n", client_pid);
            send_notification(client_pid, 1);
            enqueue(&q, client_pid);
            continue;
        }

        // Print a message indicating a new client has connected
        send_notification(client_pid, 2);
        printf("Client %d connected\n", client_pid);
    }
```

After successful connection, child process is being created and empty_slot_count is decreased. All requests from client will be handled in handle_client(…) function. Client can exit from this function with “quit” command. Then it calls remove_client(…) and be removed from server client list. Also returns response to client side.

```
// Create a new process to handle this client
pid_t fork_pid = fork();

if (fork_pid < 0) {
    perror("fork");
    exit(EXIT_FAILURE);
}

// Child process
if (fork_pid == 0) {
    server->empty_slot_count = server->empty_slot_count - 1;
    handle_client(client_pid);
    remove_client(client_pid);
    server->empty_slot_count = server->empty_slot_count + 1;
    exit(EXIT_SUCCESS);
}
```

In the end, it waits all childs to terminate and closes and unlinks server_fifo FIFO.

```
// Wait for all child processes to terminate
while (waitpid(-1, NULL, 0) > 0);

// Close and remove the named pipe
close(server_fifo_fd);
unlink(SERVER_FIFO);
```

In main function, I decleared two sigaction. First one is for handling Ctrl+C presses on client side. Second is for command “killServer”. When a client inputs “killServer” it sends SIGUSR2 signal and with this way I handled termination of all process in Server(Parent) process.

```
//-------------------- CTRL+C handler from clients ----------------//
struct sigaction sa;
memset(&sa, 0, sizeof(sa));
sa.sa_sigaction = signal_handler;
sa.sa_flags = SA_SIGINFO;

if (sigaction(SIGUSR1, &sa, NULL) == -1) {
    perror("Error setting up SIGUSR1 handler");
    return 1;
}
//-----------------------------------------------------------------//


//-------------------- killServer handler from clients -------------//
struct sigaction sa2;
memset(&sa2, 0, sizeof(sa2));
sa2.sa_sigaction = signal_handler2;
sa2.sa_flags = SA_SIGINFO;

if (sigaction(SIGUSR2, &sa2, NULL) == -1) {
    perror("Error setting up SIGUSR1 handler");
    return 1;
}
//-----------------------------------------------------------------//
```

---

client.c

At first, I define some macros and decleare variables.

```
#define SERVER_FIFO "/tmp/server_fifo"
#define NOT_FIFO "notification_fifo"
#define MAX_MESSAGE_LEN 1024
#define SEM_NAME "/my_semaphore"

pid_t server_pid;
int client_to_server_fifo_fd;
char client_to_server_fifo_name[35];
```

In main function, I first check if command-line arguments are correct or not. Then decide connection_mode of client.

```
// Check the number of command line arguments
if (argc != 3) {
    printf("Usage: ./client <connect|tryConnect> <server_pid>\n");
    return 1;
}
//print pid of client
printf("Client PID: %d\n", getpid());

//Get the connection mode and server PID from the command line arguments
int connection_mode = strcmp(argv[1], "tryConnect") == 0 ? 1 : 0; // 0 for connect, 1 for tryConnect
server_pid = atoi(argv[2]);
```

After that, I set the signal handlers for “killServer” and Ctrl+C press.

```
struct sigaction sa_term, sa_int;

// Set up the SIGTERM signal handler
sa_term.sa_handler = sigterm_handler;
sigemptyset(&sa_term.sa_mask);
sa_term.sa_flags = 0;

if (sigaction(SIGTERM, &sa_term, NULL) == -1) {
    perror("Failed to register SIGTERM handler");
    return 1;
}

// Set up the SIGINT signal handler
sa_int.sa_handler = signal_handler;
sigemptyset(&sa_int.sa_mask);
sa_int.sa_flags = 0;

if (sigaction(SIGINT, &sa_int, NULL) == -1) {
    perror("Failed to register SIGINT handler");
    return 1;
}
```

I open server_fifo for “write-only” and write PID of this client and connection_mode to FIFO. At server side, Server will read this PID and add it to connected_clients if there is an empty slot.

```c
// Open the server's named pipe for writing
int server_fifo_fd = open(SERVER_FIFO, O_WRONLY);

sem_t *semaphore = sem_open(SEM_NAME, O_CREAT, 0644, 1);
if (semaphore == SEM_FAILED) {
    perror("sem_open");
    return 1;
}

// Acquire the semaphore
if (sem_wait(semaphore) == -1) {
    perror("sem_wait");
    return 1;
}

// Send the client's PID and connection mode to the server
pid_t client_pid = getpid();
write(server_fifo_fd, &client_pid, sizeof(client_pid));
write(server_fifo_fd, &connection_mode, sizeof(connection_mode));

// Release the semaphore
if (sem_post(semaphore) == -1) {
    perror("sem_post");
    return 1;
}
```

After connection request, I create FIFO for getting responses from server side. Then start to wait connection status. If there is an empty slot in this code block will be passed. If there isn't empty slot but Client wants to wait then it starts waiting until server returns a success response.

```c
// OPEN THE SERVER TO CLIENT FIFO
char server_to_client_fifo_name[35];
sprintf(server_to_client_fifo_name, "/tmp/server_to_client_%d_fifo", client_pid);
mkfifo(server_to_client_fifo_name, 0666);
int server_to_client_fifo_fd = open(server_to_client_fifo_name, O_RDONLY);
if(server_to_client_fifo_fd == -1){
    printf("Error opening server to client fifo\n");
    exit(1);
}

int status = get_connection_status(client_pid, server_to_client_fifo_fd, connection_mode);

if(connection_mode == 1 && status != 2){
    while(1){
        if(get_connection_status(client_pid, server_to_client_fifo_fd, connection_mode)==2){
            break;
        }
    }
}
```

Later, if server accepts client, I create FIFO for sending message from client to server and open it 'write-only'.

```
// OPEN THE CLIENT TO SERVER FIFO

sprintf(client_to_server_fifo_name, "/tmp/client_to_server_%d_fifo", client_pid);
mkfifo(client_to_server_fifo_name, 0666);
client_to_server_fifo_fd = open(client_to_server_fifo_name, O_WRONLY);
```

Then starts getting input from user. Each input that client get from user is being written into client_to_server_fifo. If user inputs “exit” , it ends loop. Otherwise, Client starts to wait response from Server in handle_response(…) function.

```
while (1) {
    char message[MAX_MESSAGE_LEN];
    printf("Enter your message to the server (type 'exit' to quit): ");
    fgets(message, MAX_MESSAGE_LEN, stdin);

    // Remove the newline character at the end of the message
    message[strcspn(message, "\n")] = '\0';

    // Send the message length and then the message to the server
    int message_len = strlen(message);
    write(client_to_server_fifo_fd, &message_len, sizeof(message_len));
    write(client_to_server_fifo_fd, message, message_len);

    if(strcmp(message, "exit") == 0){
        break;
    }

    handle_response(message, server_to_client_fifo_fd,client_to_server_fifo_name);

}
```

After Client is done, closes and unlinks FIFOs.

```
// Close the named pipes
close(server_fifo_fd);
close(client_to_server_fifo_fd);
close(server_to_client_fifo_fd);
// Remove the client-specific named pipe
unlink(client_to_server_fifo_name);
unlink(server_to_client_fifo_name);


// Close the semaphore
if (sem_close(semaphore) == -1) {
    perror("sem_close");
    return 1;
}

return 0;
```

---

1)

Server:

```
erselwsl@DESKTOP-T4MTM6P:~/VSCODE_SRC/midterm$ ./biboServer test 1
Directory 'test' created.
Server PID: 9086 || Client Capacity: 1
```

Client 1:

```
erselwsl@DESKTOP-T4MTM6P:~/VSCODE_SRC/midterm$ ./biboClient connect 9086
Client PID: 9178
Response from server : 2
Enter your message to the server (type 'exit' to quit):
```

---

2)

Client 2:

```
erselwsl@DESKTOP-T4MTM6P:~/VSCODE_SRC/midterm$ ./biboClient connect 9086
Client PID: 9274
Response from server : 0
```

Server:

```
erselwsl@DESKTOP-T4MTM6P:~/VSCODE_SRC/midterm$ ./biboServer test 1
Directory 'test' created.
Server PID: 9086 || Client Capacity: 1
Client 9178 connected
Client 9274 could not connect. Server is full.
```

---

3)

Client 3:

```
erselwsl@DESKTOP-T4MTM6P:~/VSCODE_SRC/midterm$ ./biboClient tryConnect 9086
Client PID: 9418
Response from server : 1
```

Server:

```
erselwsl@DESKTOP-T4MTM6P:~/VSCODE_SRC/midterm$ ./biboServer test 1
Directory 'test' created.
Server PID: 9086 || Client Capacity: 1
Client 9178 connected
Client 9274 could not connect. Server is full.
Received SIGUSR1! Client PID: 9274
Client 9418 is waiting in the queue.
```

---

4)

Server:

```
erselwsl@DESKTOP-T4MTM6P:~/VSCODE_SRC/midterm$ ./biboServer test 2
Directory 'test' already exists.
Server PID: 9719 || Client Capacity: 2
Client 9758 connected
Received message from client 9758: test message
Received message from client 9758: test message 2
```

Client 1:

```
erselwsl@DESKTOP-T4MTM6P:~/VSCODE_SRC/midterm$ ./biboClient connect 9719
Client PID: 9758
Response from server : 2
Enter your message to the server (type 'exit' to quit): test message
not-command
Enter your message to the server (type 'exit' to quit): test message 2
not-command
Enter your message to the server (type 'exit' to quit):
```

---

5)

Client 1:

```
Enter your message to the server (type 'exit' to quit): help
Available commands: help, list, quit, killServer, readF, writeF, upload, downlo
ad.

Enter your message to the server (type 'exit' to quit):
```

---

6)

Client 1:

```
Enter your message to the server (type 'exit' to quit): list
biboClient
biboClient.c
biboServer
biboServer.c
client
server
test

Enter your message to the server (type 'exit' to quit):
```

---

7)

Client 1:

```
Enter your message to the server (type 'exit' to quit): help list
list: Sends a request to display the list of files in Servers directory.
(also displays the list received from the Server)

Enter your message to the server (type 'exit' to quit): help readF
readF <file> <line #> :
 Requests to display the # line of the <file>, if no line number is given the w
hole contents of the file is requested (and displayed on the client side)

Enter your message to the server (type 'exit' to quit): help killServer
killServer : Sends a kill request to the Server.

Enter your message to the server (type 'exit' to quit):
```

---

8)

Client 1:

```
Enter your message to the server (type 'exit' to quit): exit
Notification from server : This client has been disconnected.
```

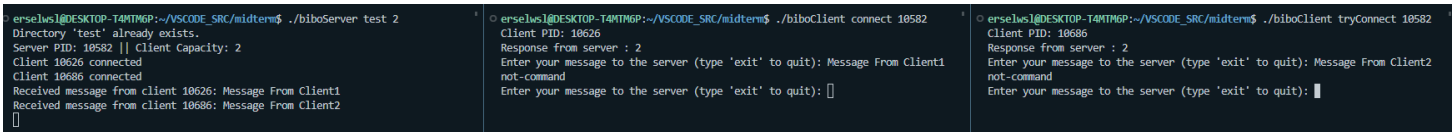
Server:

```
erselwsl@DESKTOP-T4MTM6P:~/VSCODE_SRC/midterm$ ./biboServer test 2
Directory 'test' already exists.
Server PID: 9719 || Client Capacity: 2
Client 9758 connected
Received message from client 9758: test message
Received message from client 9758: test message 2

Client 9758 has disconnected.
```

---

9)



---

10)

Client 1:

```
erselwsl@DESKTOP-T4MTM6P:~/VSCODE_SRC/midterm$ ./biboClient connect 10582
Client PID: 10626
Response from server : 2
Enter your message to the server (type 'exit' to quit): Message From Client1
not-command
Enter your message to the server (type 'exit' to quit): killServer
Received SIGTERM. Exiting...
erselwsl@DESKTOP-T4MTM6P:~/VSCODE_SRC/midterm$
```

Server :

```
erselwsl@DESKTOP-T4MTM6P:~/VSCODE_SRC/midterm$ ./biboServer test 2
Directory 'test' already exists.
Server PID: 10582 || Client Capacity: 2
Client 10626 connected
Client 10686 connected
Received message from client 10626: Message From Client1
Received message from client 10686: Message From Client2
Killing child process 10626
Killing child process 10686
erselwsl@DESKTOP-T4MTM6P:~/VSCODE_SRC/midterm$
```

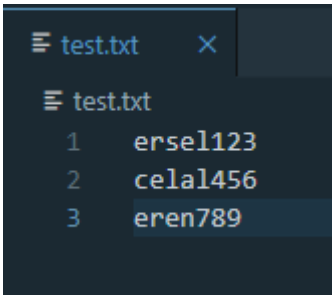
Client 2:

```
erselwsl@DESKTOP-T4MTM6P:~/VSCODE_SRC/midterm$ ./biboClient tryConnect 10582
Client PID: 10686
Response from server : 2
Enter your message to the server (type 'exit' to quit): Message From Client2
not-command
Enter your message to the server (type 'exit' to quit): Received SIGTERM. Exi
ting...
erselwsl@DESKTOP-T4MTM6P:~/VSCODE_SRC/midterm$
```

---

11)

Test.txt :




```
erselwsl@DESKTOP-T4MTM6P:~/VSCODE_SRC/midterm$ ./biboServer test 2
Directory 'test' already exists.
Server PID: 11198 || Client Capacity: 2
Client 11242 connected
Filename: |test.txt|
Line 2: celal456
```

```
erselwsl@DESKTOP-T4MTM6P:~/VSCODE_SRC/midterm$ ./biboClient connect 11198
Client PID: 11242
Response from server : 2
Enter your message to the server (type 'exit' to quit): readF test.txt 2
Line 2: celal456

Enter your message to the server (type 'exit' to quit):
```