GTU Department of Computer Engineering
CSE 344- Spring 2023
MIDTERM Report

Mehmet Mete Şamlıoğlu
200104004093

Server Side (biboServer)

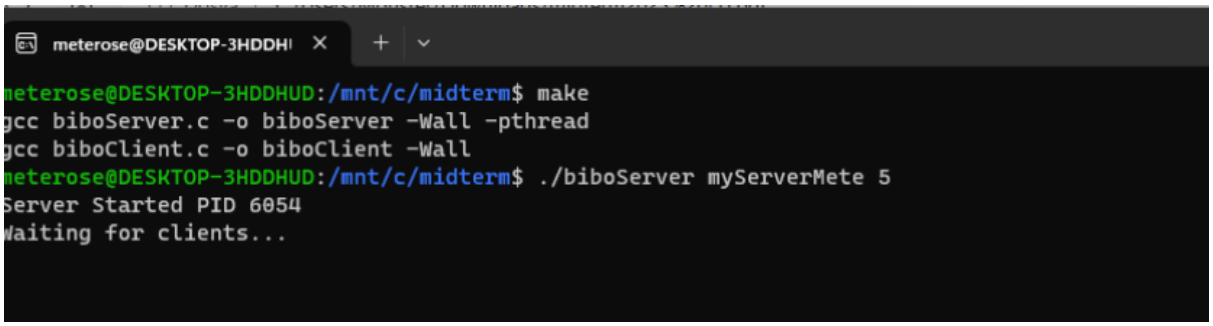
On the server side of the program, I used FIFO for IPC and semaphores to provide synchronization. In the beginning, I tried to provide communication between the client and server using sockets but later on, I was told that is not allowed, I tried to convert my algorithm to work with semaphores instead of sockets. I have used two helper functions to manage two operations. To handle the client's operations I wrote the void client_commands function and also to void handle signals I wrote the signal_handler( ) function.

I also declared global variables to manage semaphores and client operation. I declared four global variables which are in below;

```
#define BUFFER_SIZE 256
#define MAX_CLIENTS 10



sem_t sem_client_count;
int client_count = 0;
int client_fifo_fds[MAX_CLIENTS];
volatile sig_atomic_t condition_break = 1;
```

I used sem_client_count to keep track of the clients, int client_count to store the client_count, and also client_fifo_fds to keep all the client_fifos to make the system work with more than one client. It is fixes sized and set to the MAX_CLIENTS count that is limited by the system.

To start a new server the code that is supposed to enter is as below;



The second argument is the name of the server and the third one is for to determine max_client_count in the system. After the running command, the server will be waiting for clients. All the clients will use the server's PID id to connect to the server.

In the very first step, I separate the running command and declare the max_client_size and also the directory for the server. I create a directory that has the same name as the name of the server, this directory will keep the log_files of all clients in it. After that, by using the PID of the server, I created a file in **/tmp/** directory with the name **fifo_(pid_number9)** to use for IPC. The client will find that file in /tmp/ and connect to the server by using it.

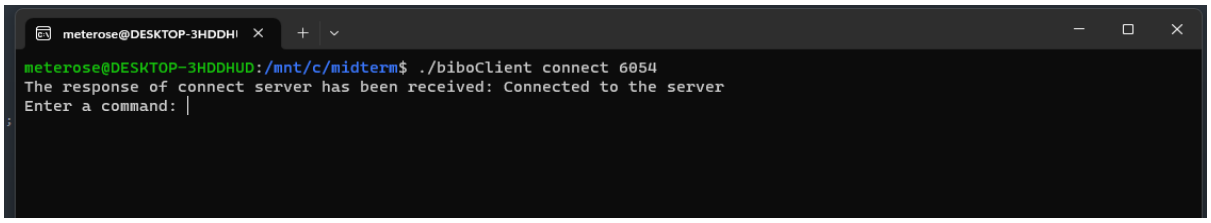
After creating the necessary files to provide FIFO, the main loop will start iterating, the condition for the main loop is a volatile sig_atomic_t variable condition_break. It is initially assigned to 1, and according to the signal that will be caught later on, the value of it will be zero which is an end condition. This was my goal to terminate the iteration but I could not handle it properly. Even though I successfully send the SIGTERM signal the loop does not terminate. So the iteration starts with the read function that is reading the fifo_directory and saves the content of it to a string called **client_fifo.** If the read operation is successful, a new file descriptor is created and opens the client_fo in read and write mode.

```
char client_fifo[BUFFER_SIZE];
ssize_t bytes_received = read(fd_fifo, client_fifo, sizeof(client_fifo) - 1);
if (bytes_received <= 0) {
    perror("FIFO read failed");
    break;
}
client_fifo[bytes_received] = '\0';

// Open the client FIFO for writing
int client_fifo_fd = open(client_fifo, O_RDWR);
if (client_fifo_fd == -1) {
    perror("Client FIFO open failed");
    return 1;
}

sem_wait(&sem_client_count);
```

After this, the sem_wait statement starts computing, and the server waits for the clients, If a client is connected to the server, it is PID and the count that is assigned to that client will be printed and it will stay connected until it quits.

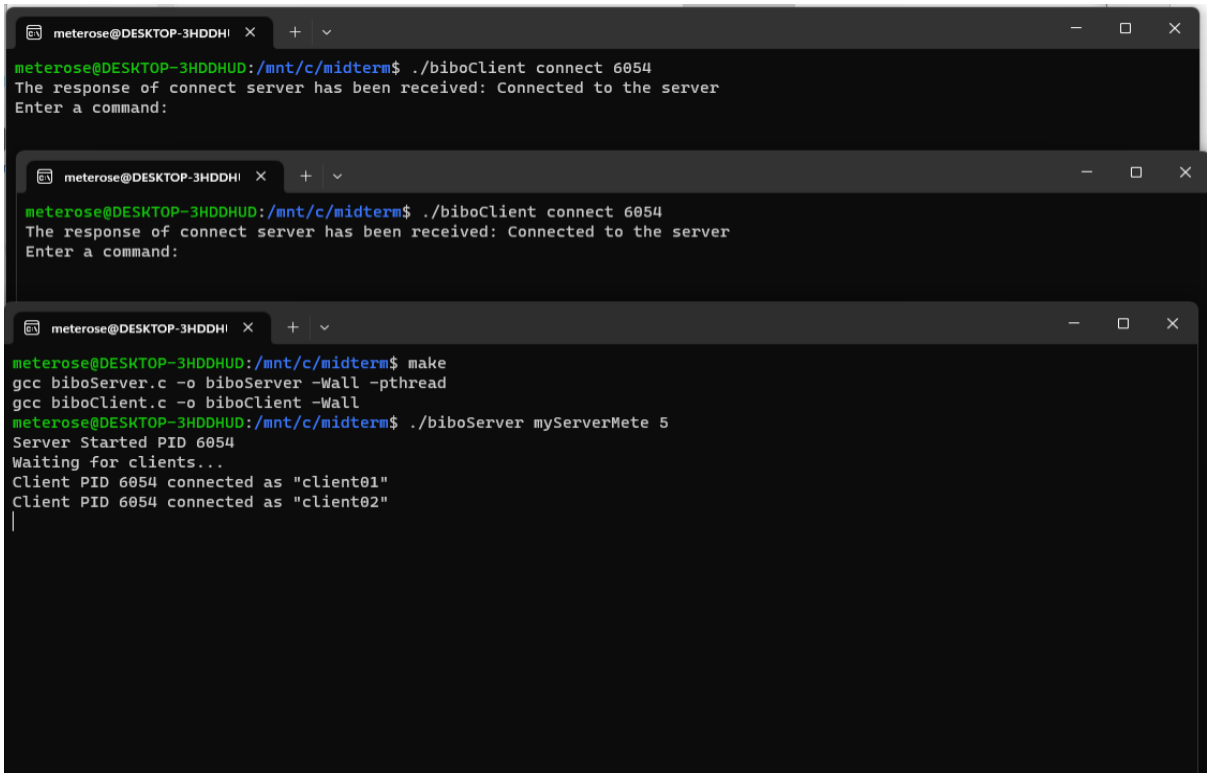
If a client connects to the server via another terminal by using the ./biboClient connect pid_no comment, it will directly be connected to the server and both terminals will display that information.

Terminal Client



Terminal System

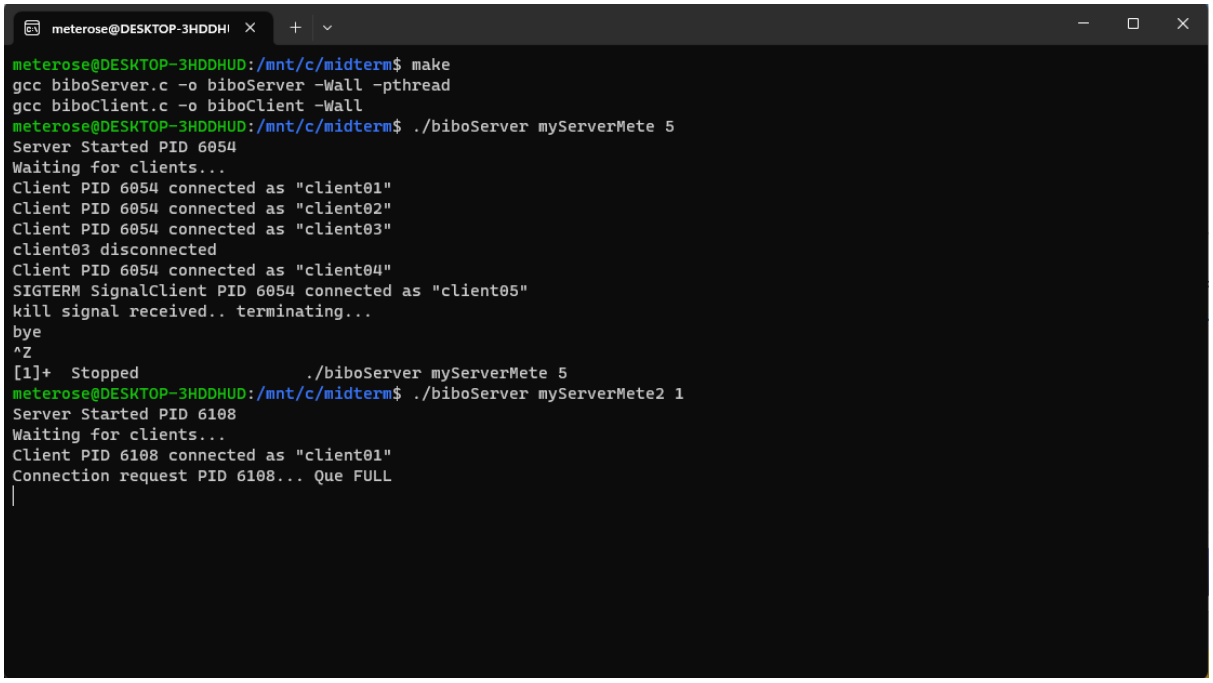
As you can see above, the terminal shows the client that is connected to the system**. Let's connect another client to the system and check the Terminal system.**



As you can see above both clients are connected to the system, and both of them are ready to command. The system will be able to keep as many as clients that are limited by max_client. So we decide how many clients are going to be stored in the system at the beginning of the system. Even though this is determined by the user, I had to limit that count because of control of the array that stores the client_fifos.

MAX_CLIENT CONTROL

I started the system with a capacity of 1 and connect two clients, Let's see what is going to happen.



As you can see, the server does not let the next client connect because it is full. I handled it in the main iteration as you can see below.

```
sem_wait(&sem_client_count);
if (client_count >= max_clients) {
    printf("Connection request PID %d... Que FULL\n", getpid());
    close(client_fifo_fd);
    sem_post(&sem_client_count);
    continue;
}
client_count++;
sem_post(&sem_client_count);
```

So whenever a new client wants to connect, the server checks the current client count and compares that with the max_client value that is limited in the beginning. It is greater or equal to the client max_clients count, it is not going to be connected

and kept waiting. I could not handle the part that requires a queue to store the next client in it until one of the other clients disconnect. The algorithm that I use did not achieve that task. **So I basically do not allow the client to connect if the server is full instead of putting it into the queue.**

CLIENT COMMANDS

The client will be sent to the cliend_commands function to perform the commands. This will be done by using forks(). So If the client is connected to the system without any problem, the fork will compute and in the child process the client_commands function will be called with client_index directory name variables and it will compute in that function until it quits from the server.

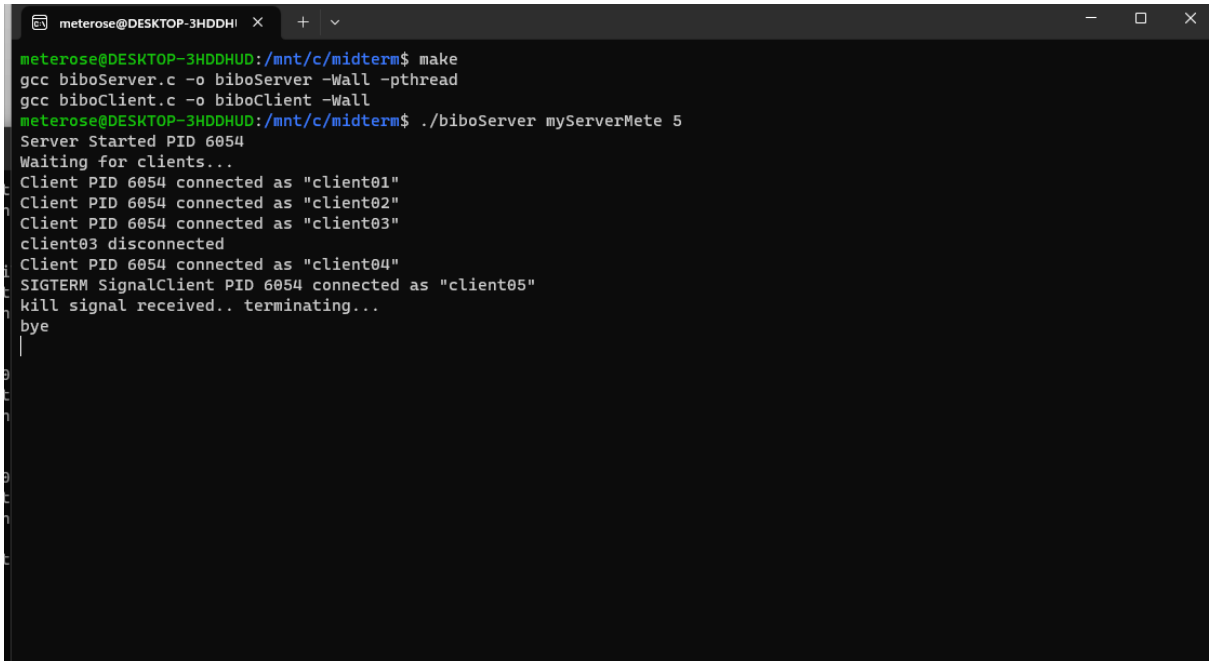
```
        printf("Client PID %d connected as \"client%02d\"\n", getpid(), client_index + 1);
        pid_t pid = fork();
        if (pid == -1) {
            perror("Fork");
            break;
        }
        else if (pid == 0)
        {
            // Child process
            close(fd_fifo);
            client_commands(client_index, directory_name);
            exit(0);
        } else {
            // Parent process
            close(client_fifo_fd); // Close the client FIFO in the parent process
        }
    }
```

```
void client_commands(int client_index, char *directory_name) {
    int client_fifo_fd = client_fifo_fds[client_index];

    // Read from client FIFO
    char command[BUFFER_SIZE];
    char *server_response = "Connected to the server";
    write(client_fifo_fd, server_response, strlen(server_response));
    usleep(100000);
    while (1)
    {
        ssize_t bytes_received = read(client_fifo_fd, command, sizeof(command));
        command[strcspn(command, "\n")] = '\0';
        if (bytes_received <= 0) {
            if (bytes_received == -1) {
                perror("Failed FIFO read operation");
            }
            break;
        }
        else
        {

            if(strcmp(command, "help") == 0 || strcmp(command, "help help") == 0)
            {
                char *str = "Available comments are :\nhelp, list, readF, writeT, upload, download, quit, killServer";
                write(client_fifo_fd, str, strlen(str));
            }
            if(strcmp(command,"quit") == 0)
            {
                printf("client%02d disconnected\n",client_index + 1);
                char filename[128];
                char path[256];
                snprintf(filename, sizeof(filename), "logfile_%d", (client_index + 1));
                snprintf(path, sizeof(path), "./%s/%s", directory_name, filename);
                FILE* file = fopen(path, "w");
                if (file == NULL)
                {
                    perror("File is not created");
                }
                fclose(file);
                break;
            }
            if(strcmp(command,"list") == 0)
            {
                printf("Ben");
                char list_command[256];
                snprintf(list_command, sizeof(list_command), "ls %s", directory_name);
                int command_check = system(command);
                if(command_check == -1)
                {
                    perror("Command");
                }
            }
            if( strcmp(command, "killServer") == 0)
            {
                kill(getppid(), SIGTERM);
                break;
            }

        }

        memset(command, 0, sizeof(command)); /* Clear command*/
        usleep(100000);
    }
```

So the client_Commands function communicates with the client via FIFO and performs the commands accordingly, I did not handle getting synchronized commands from the client. I spent most of my time on it but all the commands that I took from the client repeatedly were not accurate. For example, if clients send 7 commands I took 5 of them in this function because of the synchronization issue. Therefore I did not take multiple commands from the user, I took only one command instead. So let's try some of the methods that work properly.

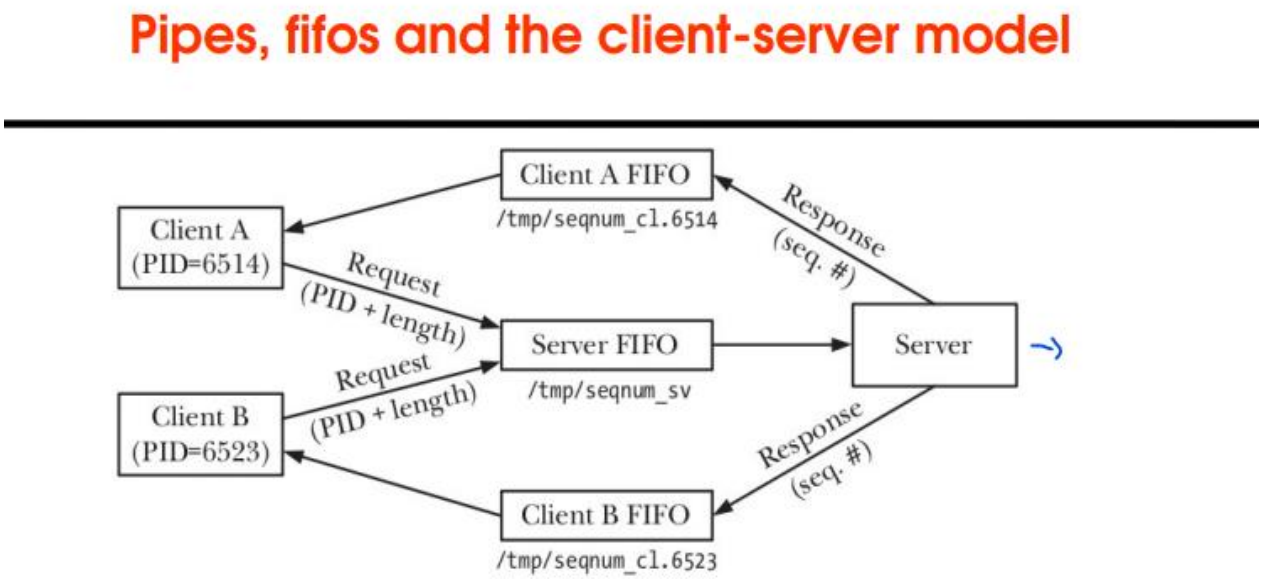
Help

Quit and killServer



I did not perform the rest of it because of the lack of time and also the problem that is related to the synchronization problem that I mentioned above.

The mode that I used to build that algorithm is exactly the same as the one that we saw in the lecture which is this.

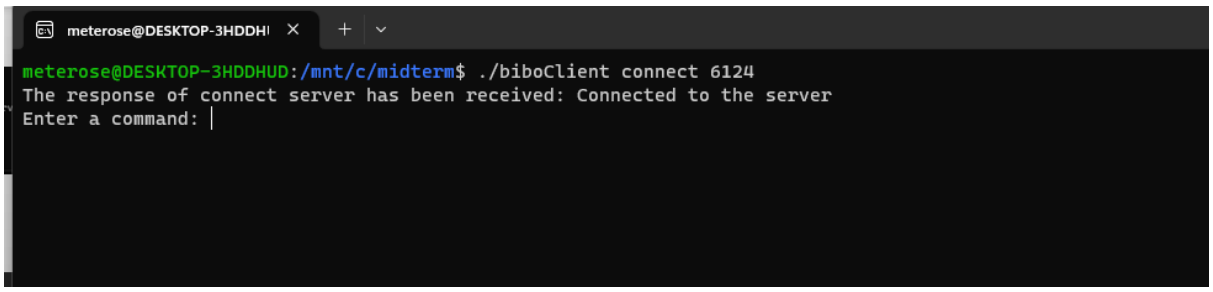


This was my architecture model when I started to create an algorithm, All the algorithm is built upon this architecture. The Server represents biboServer and the client represents biboClients. By using the function(client_commands) that I put above, they communicate with each other and perform the required tasks accordingly.

Client-side (client Server)

The client part of the system was relatively easy when compared to the server side because all I needed to was communicate with the client_commands function, to run the client server there are two running commands which are

- ./biboClient connect server_pıd
- ./biboClient try connect server_pid



In my client code, only the connect version of it works because as I mentioned in the server side I could handle queue operation therefore I did not compute the tryConnect version of the running command.

So the client-side simply communicates with the client_commands function from the server side and the client_command function performs the commands that are given by the biboClient.

```
    printf("Enter a command: ");
    fgets(command, sizeof(command), stdin);
    command[strcspn(command, "\n")] = '\0';
    write(client_fifo_fd, command, strlen(command));
    usleep(100000);
    ssize_t server_response = read(client_fifo_fd, response_message, sizeof(response_message));
    if (server_response == -1 && errno != EAGAIN)
    {
        perror("Server response read failed");
    }
    response_message[server_response] = '\0';
    printf("Server response: %s\n", response_message); /* Server Response */
    memset(command, 0, sizeof(command));


close(client_fifo_fd);
unlink(path_client_fifo);
close(fifo_server);
```

In this part of the biboClient code, the server communicates with the client, the response message is the message that is taken by the client_command function via FIFO and the command is a string that contains the task that is going to be performed by the server-side. It is clear that the IPC is handled with FIFOs. The problem was determining when the server will read and write to the FIFO and the same question for the client. That was the part that I was not able to handle. Therefore my code is not complete.