Gebze Technical University

Department Of Computer Engineering

CSE 344 Spring 2023

System Programming

Homework 4

Due Date: 22.05.2023

Abdurrahman Bulut
1901042258

Introduction

In our midterm project, we were asked to write a client and a server program. This server enables multiple clients to connect, access and modify the contents of files in a specific directory.

That project was about inter-process communication and multiple synchronization primitives. I used Fifos for IPC and semaphores for synchronization. It asked us to use processes for each client. Each client runs its requests in a child process.Therefore I used fork() system call to create it for each client.

But for this homework (Hw4), it is asked to us to use thread instead of processes. So, I created a thread pool. Instead of creating a separate process for each request, a thread from the thread pool will be assigned to handle each request.

I took the code from the book as a basis.

Client - Server IPC: taken from the book.

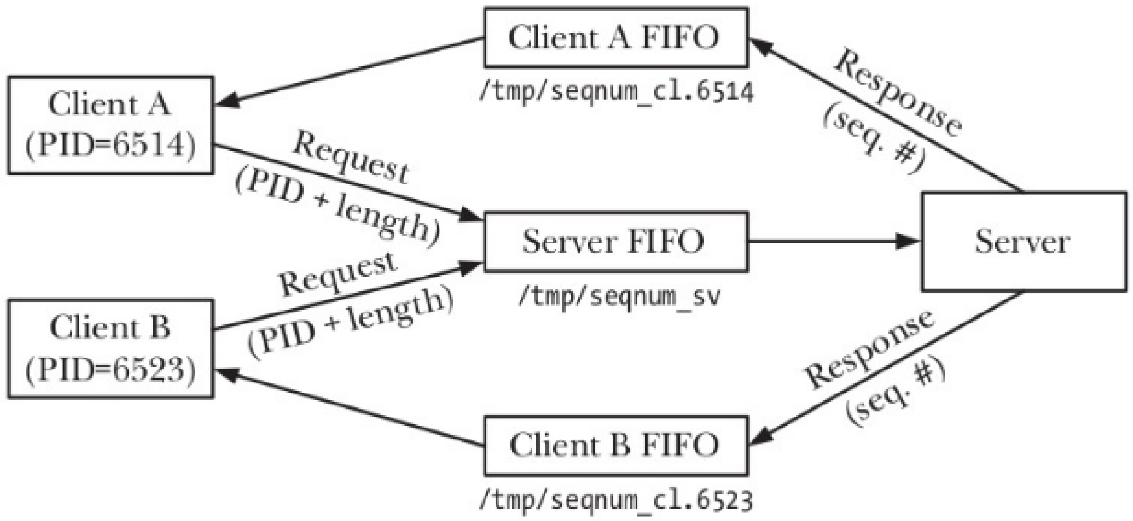


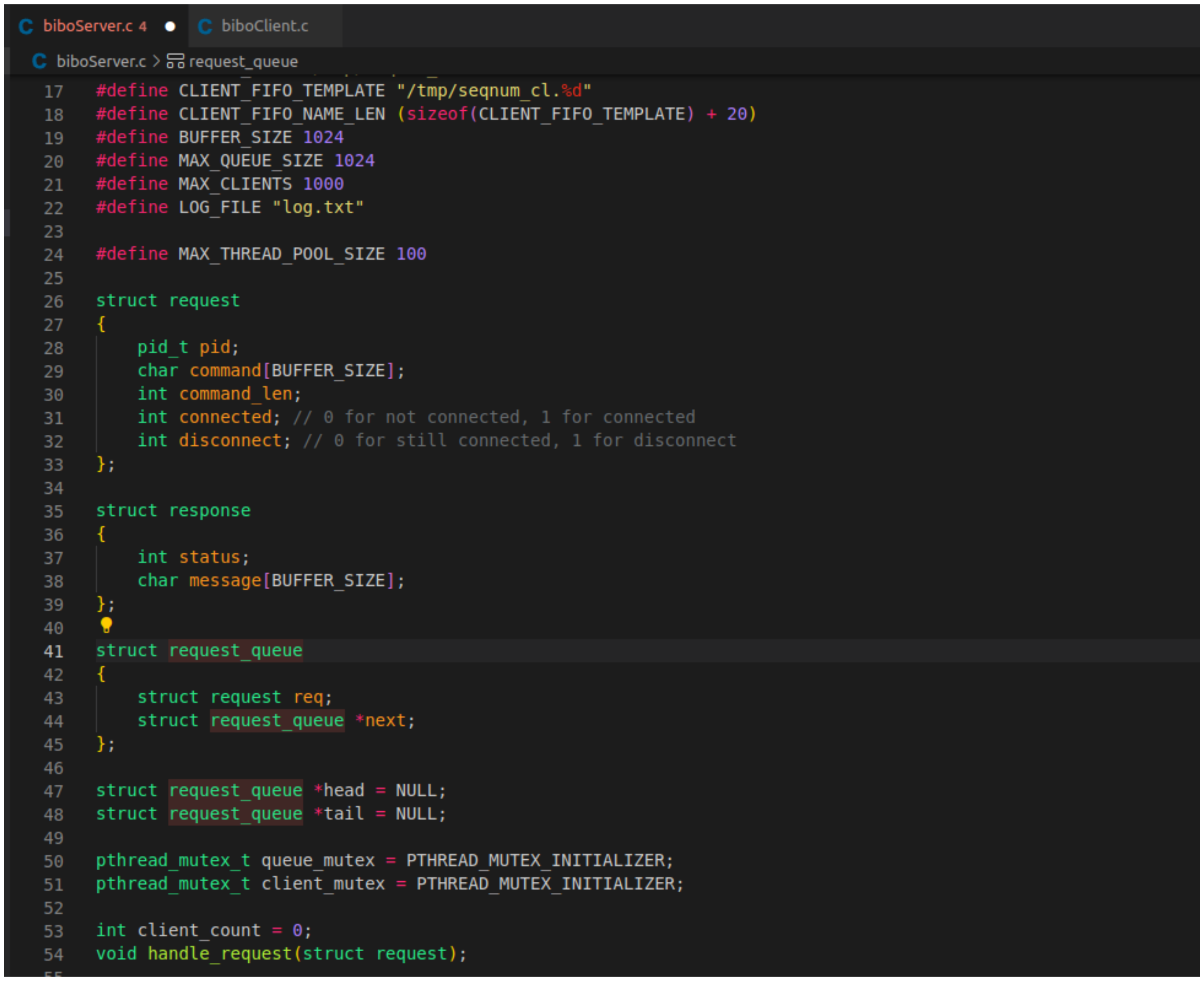
Figure 44-6: Using FIFOs in a single-server, multiple-client application

Above picture shows that each client will have a pid and will send its request via server fifo. Server will read the request from the client, process it and send a response to the related client with client fifo. Client fifos names with their pid numbers. For each request, the client will send its pid also. Server will recognize the client with its pid. My code exactly does this.

Server-side:

Multiple clients can connect, access, and alter the contents of files in a certain directory concurrently thanks to the architecture's server-side design. The server maintains a pool of worker threads which handle incoming client requests. The pool size is adjustable, providing a level of concurrency control. When the server receives a client request, it assigns a worker thread to process it. The server operates continuously, constantly listening for client requests. To allow multiple clients to connect at the same time, the server implements a processing queue and a waiting queue. When a client request is received, it is enqueued in the processing queue if the number of currently connected clients has not reached the server's maximum capacity. If the server is at full capacity, the request is enqueued in the waiting queue instead. A queue is used by the server to control client connections. When a client disconnects, the server removes it from the processing queue and checks the waiting queue for any client requests. If any are found, the first request in the queue is moved to the processing queue and processed accordingly. The maximum number of clients that can connect at once is reflected in the queue's maximum size. The server determines whether there is room in the queue when a client tries to join. The server will either force the client to wait (Connect option) or notify them that the server is currently full (TryConnect option) if the queue is full. Among these requests are those to list the files in the server directory, read from and write to files, upload and download files, and list the files in the server directory.

In my code, the server fifo directory is “/tmp/seqnum_sv”. Client's directory is “/tmp/seqnum_cl.%d”. %d is for the pid of the client. I assumed the maximum client number was 1024.


biboServer.c 4 biboClient.c
biboServer.c > request_queue

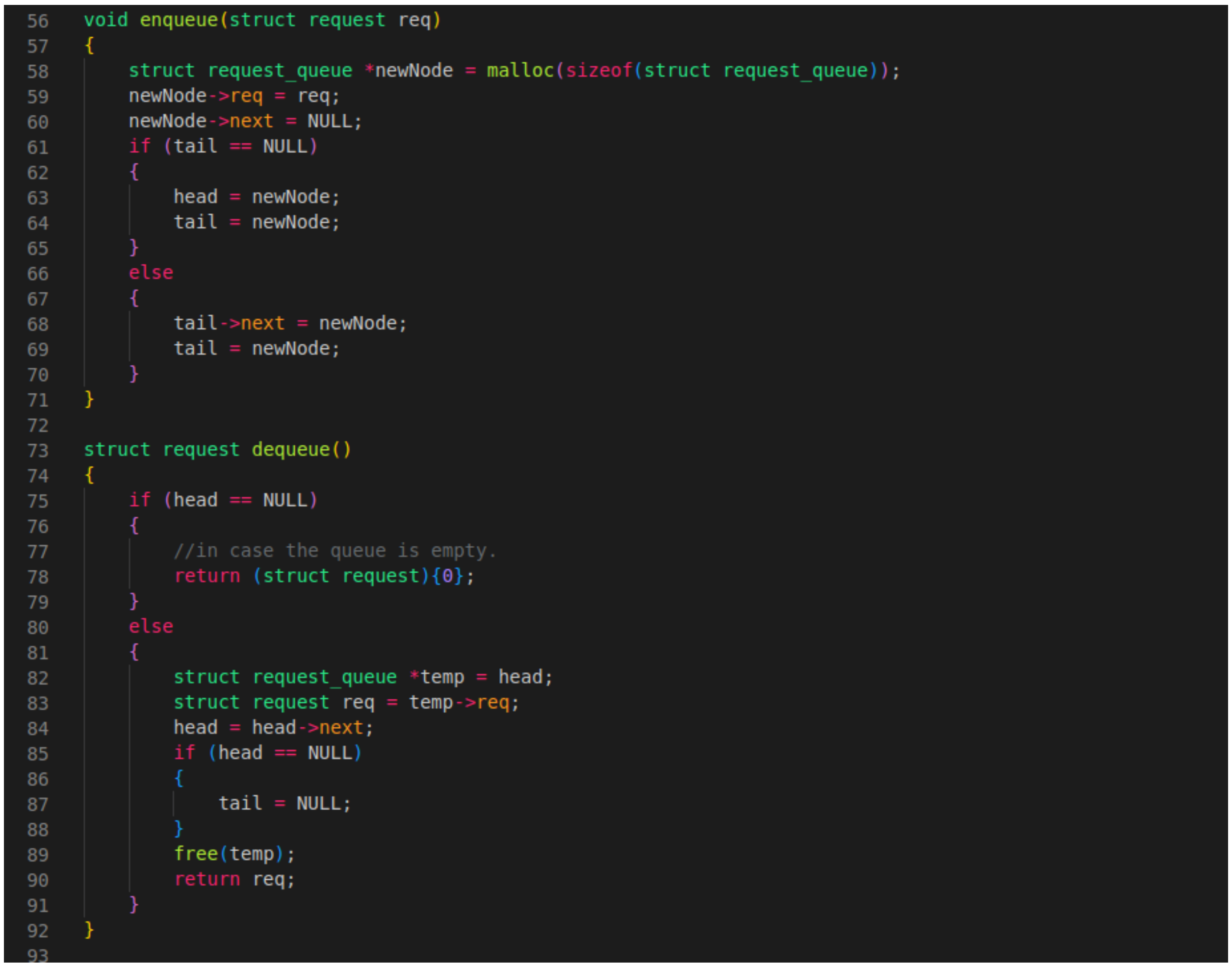
```c
#define CLIENT_FIFO_TEMPLATE "/tmp/seqnum_cl.%d"
#define CLIENT_FIFO_NAME_LEN (sizeof(CLIENT_FIFO_TEMPLATE) + 20)
#define BUFFER_SIZE 1024
#define MAX_QUEUE_SIZE 1024
#define MAX_CLIENTS 1000
#define LOG_FILE "log.txt"

#define MAX_THREAD_POOL_SIZE 100

struct request
{
    pid_t pid;
    char command[BUFFER_SIZE];
    int command_len;
    int connected; // 0 for not connected, 1 for connected
    int disconnect; // 0 for still connected, 1 for disconnect
};

struct response
{
    int status;
    char message[BUFFER_SIZE];
};

struct request_queue
{
    struct request req;
    struct request_queue *next;
};

struct request_queue *head = NULL;
struct request_queue *tail = NULL;

pthread_mutex_t queue_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t client_mutex = PTHREAD_MUTEX_INITIALIZER;

int client_count = 0;
void handle_request(struct request);
```

I created request and response structures. Struct request and struct response are two data structures that define the format of requests and responses. A client will send a request to the server with its pid. Variables such as “max_clients”, “current_client”, “serverFd”, “pool_size”, “clientFifo”, “req”, and “resp” are initialized. These variables are used to store information such as the maximum number of clients the server can handle, the current number of connected clients, file descriptors for the server and clients, and structures for client requests and server responses.

Mutexes are used to prevent data races. A data race occurs when two concurrent threads are competing to access and modify shared data simultaneously and leading to undefined behavior and bugs. In my server code, mutexes are used to protect shared resources that are accessed and modified by multiple threads. In this case, the shared resources are the connected_clients count, the processing_queue, and the waiting_queue.

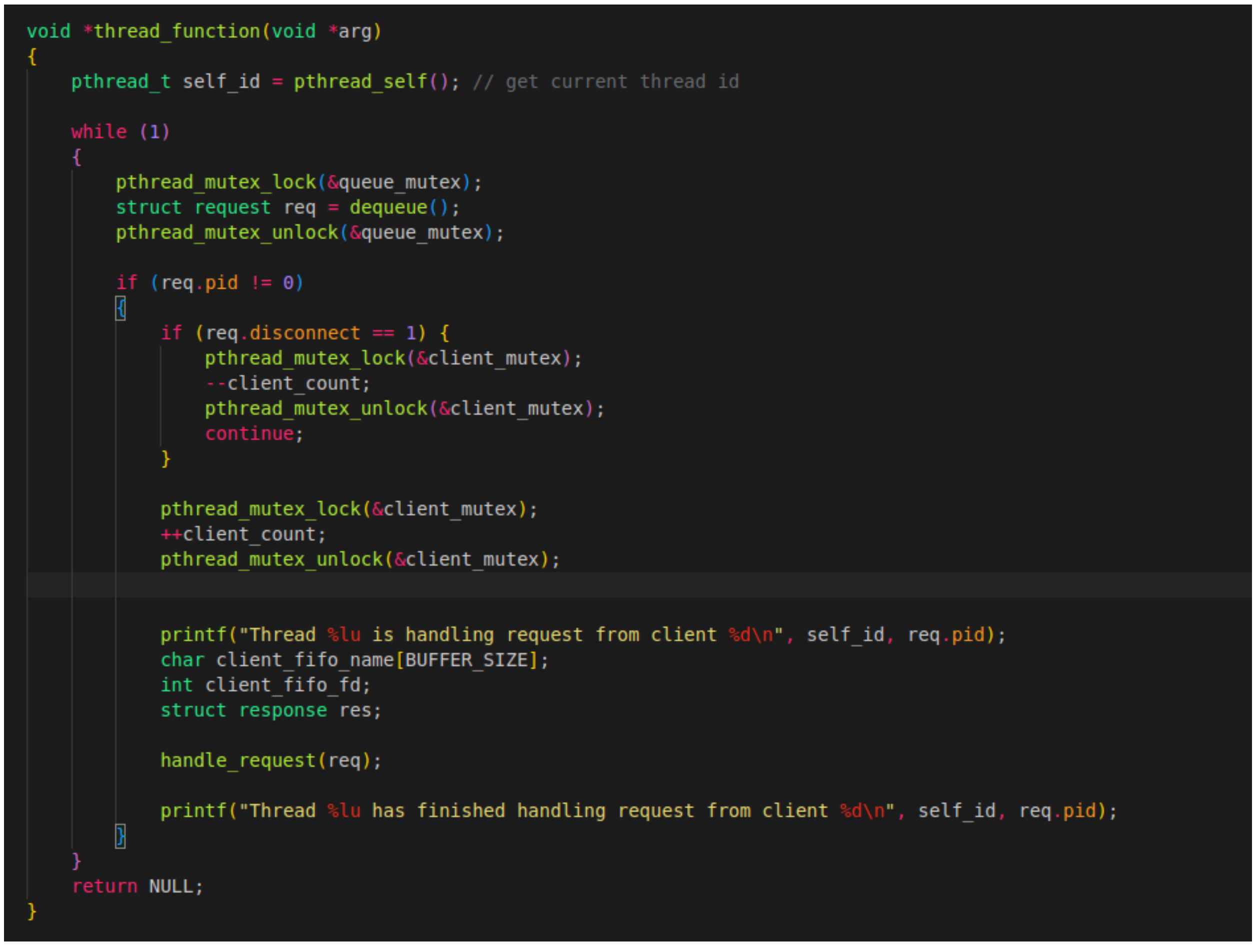
“struct request_queue” is used to create a queue of requests. Each node in the queue contains a request and a pointer to the next node in the queue. head and tail keep track of the start and end of the request queue. “pthread_mutex_t queue_mutex” and “client_mutex” are mutex variables used for synchronization. They are used to ensure that only one thread at a time can access the shared request queue or modify the client count.

The server will work with “./biboServer Cloud 3 4”. The cloud word is the directory. 3 is the maximum number of clients that can connect to the server at a time. 4 is the thread pool size. If more than this number, the client tries to connect to the server. The server puts them in the queue. The server checks if it received exactly two command-line arguments (the directory name and the maximum number of clients). If not, it prints a usage message and exits. The server opens a log file for writing. If this operation fails, the server prints an error message and exits. The server creates a new directory with the name provided as a command-line argument using mkdir(), and then changes to this directory using chdir().



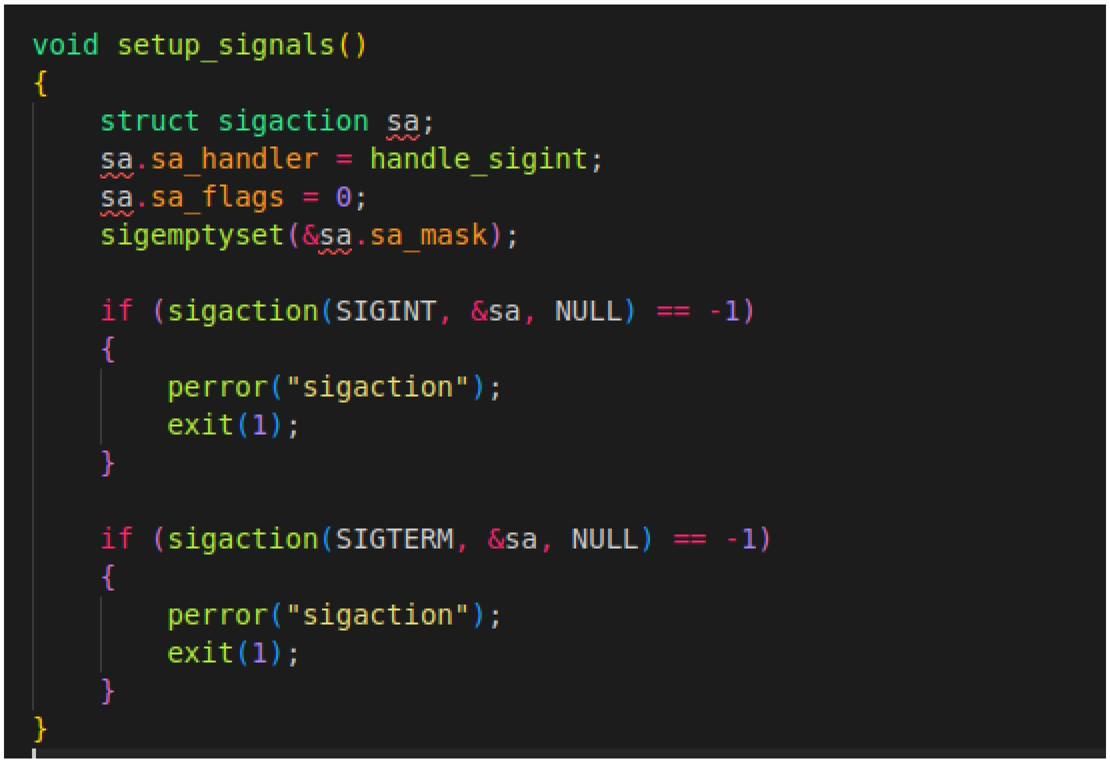
```c
void enqueue(struct request req)
{
    struct request_queue *newNode = malloc(sizeof(struct request_queue));
    newNode->req = req;
    newNode->next = NULL;
    if (tail == NULL)
    {
        head = newNode;
        tail = newNode;
    }
    else
    {
        tail->next = newNode;
        tail = newNode;
    }
}

struct request dequeue()
{
    if (head == NULL)
    {
        //in case the queue is empty.
        return (struct request){0};
    }
    else
    {
        struct request_queue *temp = head;
        struct request req = temp->req;
        head = head->next;
        if (head == NULL)
        {
            tail = NULL;
        }
        free(temp);
        return req;
    }
}
```

“enqueue “and “dequeue” are used to manage the queue of requests. The enqueue function is used to add new requests to the queue. The dequeue function is used to remove a request from the queue when it is ready to be processed.



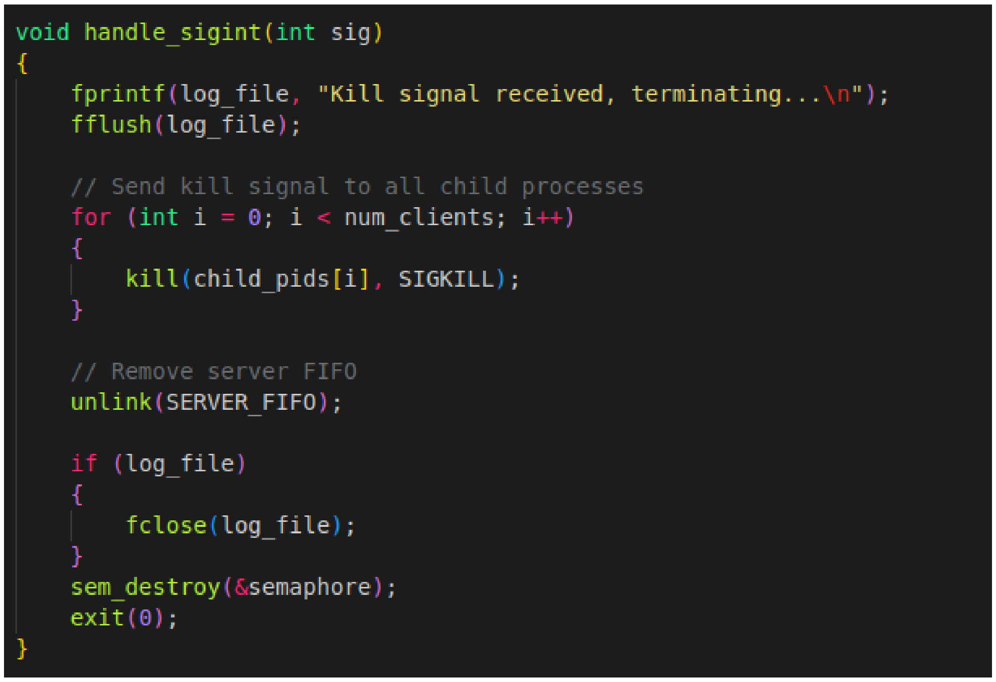
```c
void *thread_function(void *arg)
{
    pthread_t self_id = pthread_self(); // get current thread id

    while (1)
    {
        pthread_mutex_lock(&queue_mutex);
        struct request req = dequeue();
        pthread_mutex_unlock(&queue_mutex);

        if (req.pid != 0)
        {
            if (req.disconnect == 1) {
                pthread_mutex_lock(&client_mutex);
                --client_count;
                pthread_mutex_unlock(&client_mutex);
                continue;
            }

            pthread_mutex_lock(&client_mutex);
            ++client_count;
            pthread_mutex_unlock(&client_mutex);


            printf("Thread %lu is handling request from client %d\n", self_id, req.pid);
            char client_fifo_name[BUFFER_SIZE];
            int client_fifo_fd;
            struct response res;

            handle_request(req);

            printf("Thread %lu has finished handling request from client %d\n", self_id, req.pid);
        }
    }
    return NULL;
}
```

Each thread in the thread pool of the server runs the "thread_function" function. Client requests are continuously processed from a common request queue. For logging reasons, this method first obtains the current thread ID. The shared request queue is then locked, a request is retrieved and removed, and the queue is then made available to other threads once again. At this point, the process continues indefinitely. Depending on whether a valid request signals a client disconnect or not, the function modifies the client count if one is received. The request is then processed, and after the task has been completed, a log entry is made. The server may process several requests concurrently thanks to this function, increasing its effectiveness.



```c
void setup_signals()
{
    struct sigaction sa;
    sa.sa_handler = handle_sigint;
    sa.sa_flags = 0;
    sigemptyset(&sa.sa_mask);

    if (sigaction(SIGINT, &sa, NULL) == -1)
    {
        perror("sigaction");
        exit(1);
    }

    if (sigaction(SIGTERM, &sa, NULL) == -1)
    {
        perror("sigaction");
        exit(1);
    }
}
```

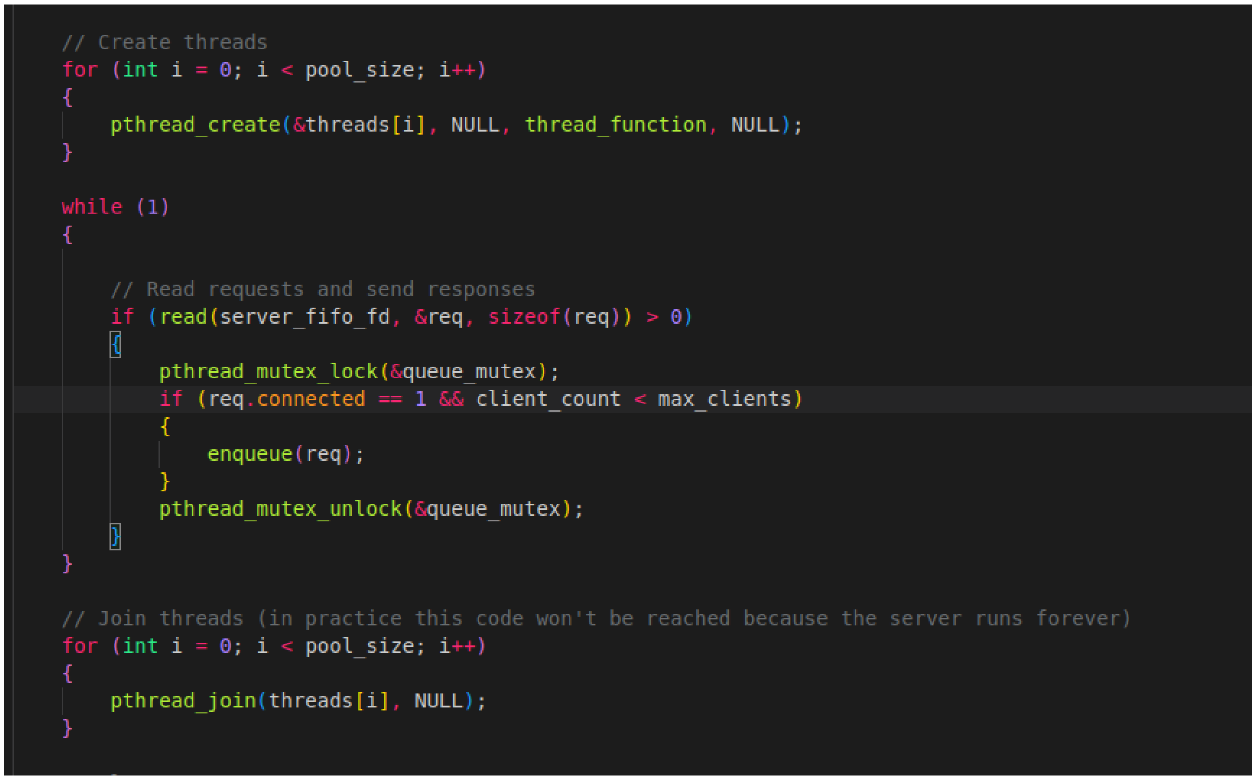
The server sets up signal handlers to handle various signals properly with setup signals function.



```c
void handle_sigint(int sig)
{
    fprintf(log_file, "Kill signal received, terminating...\n");
    fflush(log_file);

    // Send kill signal to all child processes
    for (int i = 0; i < num_clients; i++)
    {
        kill(child_pids[i], SIGKILL);
    }

    // Remove server FIFO
    unlink(SERVER_FIFO);

    if (log_file)
    {
        fclose(log_file);
    }
    sem_destroy(&semaphore);
    exit(0);
}
```

A named pipe (FIFO) with a particular file path and permissions is created by the server. The server emits an error message and terminates if this operation is unsuccessful.The server goes into an endless cycle of waiting for client requests. The server reads into the request structure for each request from the server FIFO. The server determines if any slots are open in the client queue if the request is a "Connect" command. If there are, the client is included in the list and receives a message confirming its connection. The client receives a response stating that it is in the queue if the client queue is full. When a "tryConnect" command is requested, the server determines whether the client queue is full. If it is, the server sends a response to the client indicating that the queue is full, and the client is expected to exit.

I created a client structure but it doesn't work as we expected. I tried to create a code structure that does if more than the maximum client number that server accepts client try to connect to the server, those clients should enter the queue with waiting status. If a client finishes its work, it will be removed from the client array and a waiting client will go to the client array. That's why I created some helper functions.



```
// Create threads
for (int i = 0; i < pool_size; i++)
{
    pthread_create(&threads[i], NULL, thread_function, NULL);
}

while (1)
{

    // Read requests and send responses
    if (read(server_fifo_fd, &req, sizeof(req)) > 0)
    {
        pthread_mutex_lock(&queue_mutex);
        if (req.connected == 1 && client_count < max_clients)
        {
            enqueue(req);
        }
        pthread_mutex_unlock(&queue_mutex);
    }
}

// Join threads (in practice this code won't be reached because the server runs forever)
for (int i = 0; i < pool_size; i++)
{
    pthread_join(threads[i], NULL);
}
```

The server generates a thread pool as its initial last step, and each thread is given the task of running the thread_function. It is in charge of responding to client queries. Following this initialization, the server starts an indefinite loop where it constantly reads and validates client requests. The request is added to the shared queue for the threads to handle if it comes from a new connection and the number of clients is below the set limit. Although there is a mechanism to rejoin the threads to the main thread at the end, in reality, this portion is not available since the server is built to run forever, enabling it to continue handling client requests in a multi-threaded way. Lastyly, The server cleans up resources such as the server FIFO, the log file, and the semaphore when it is prepared to shut down.

I created some helper methods to manage clients such as add_clients, remove_client, num_of_connected_clients, initialize_client and available_slots.

The handle_request() function is made to handle a range of client requests. The parameters for this function are a struct request object and an integer clientFd. cliendFd is an opened client Fifo descriptor. The client's command is included in the struct request object req, which the server must handle, and clientFd is the file descriptor describing the client connection. The client submits a request to the server, which receives it, processes it, and then replies to the client. It first parses the command that comes from the client. It assigns them to command and args variables.

Command Execution: Depending on the parsed command, the function performs different operations.

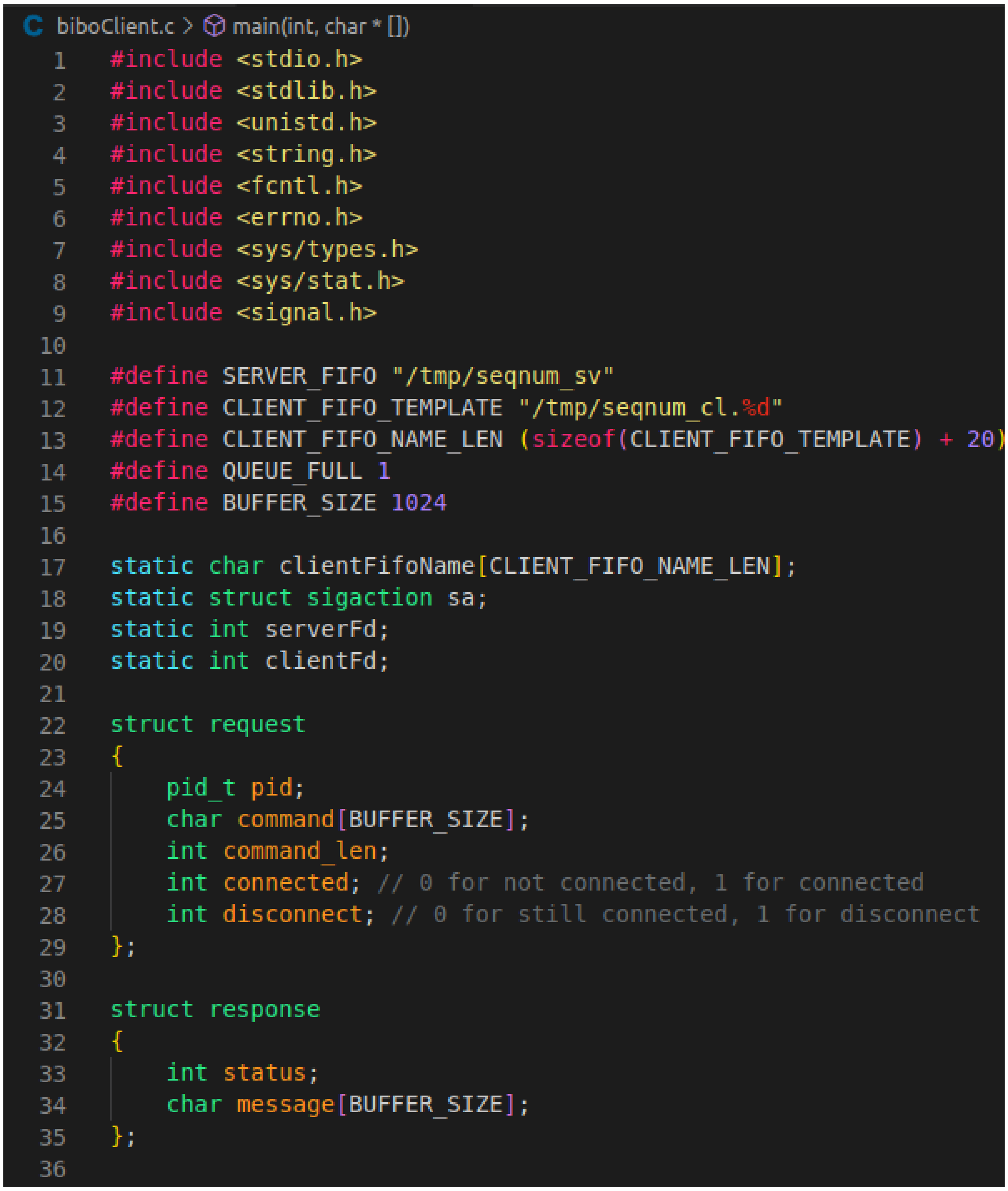
- If the command is killServer, the server is instructed to terminate itself.
- If the command is list, the server lists the files in its current directory and sends this list back to the client.
- If the command is help, the server sends back a message to the client detailing how to use different commands.
- If the command is readF, the server reads a file as per the client's request and sends back the file's contents.
- If the command is writeT, the server writes to a specified file based on the client's request.
- If the command is upload, the server receives a file from the client and stores it.
- If the command is download, the server sends a specified file to the client.
- If the command is quit, the server logs that the client has disconnected and closes the connection.

- If the command is killServer, the server logs a message that it has received a kill request, terminates all connections, frees up resources, and then shuts down.

The server prepares a response and sends it back to the client after processing the client's command. A "Command Not Found!" message is returned if the command is unidentified. The function adds a new record to a log file following each request. It saves the PID of the client together with the command that was sent. The function incorporates error handling for a number of potential problems, including difficulty with accessing a file or writing to a client FIFO.

Client-side:

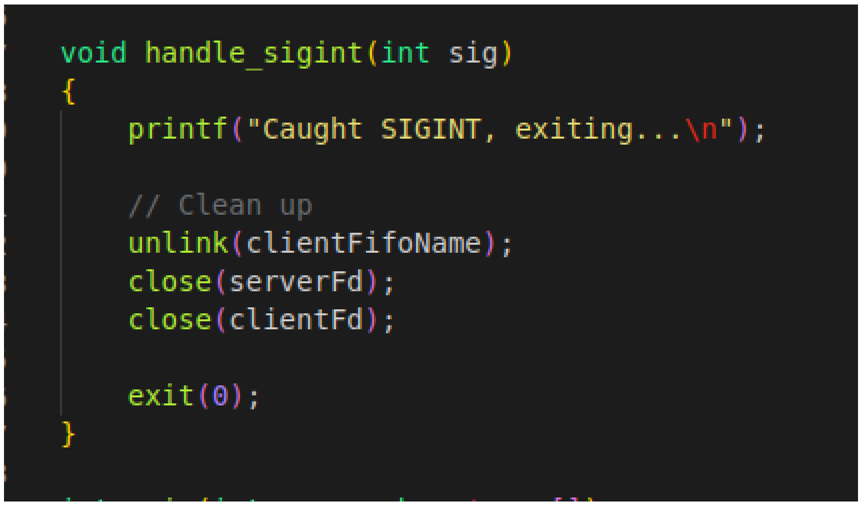
My client code uses the same request-response structures and same fifo file name prefixes. I did not change my client code for homework 4. It is the same as midterm project.



biboClient.c > main(int, char * [])

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <signal.h>

#define SERVER_FIFO "/tmp/seqnum_sv"
#define CLIENT_FIFO_TEMPLATE "/tmp/seqnum_cl.%d"
#define CLIENT_FIFO_NAME_LEN (sizeof(CLIENT_FIFO_TEMPLATE) + 20)
#define QUEUE_FULL 1
#define BUFFER_SIZE 1024

static char clientFifoName[CLIENT_FIFO_NAME_LEN];
static struct sigaction sa;
static int serverFd;
static int clientFd;

struct request
{
    pid_t pid;
    char command[BUFFER_SIZE];
    int command_len;
    int connected; // 0 for not connected, 1 for connected
    int disconnect; // 0 for still connected, 1 for disconnect
};

struct response
{
    int status;
    char message[BUFFER_SIZE];
};

```

I created a handle_sigint() function to handle SIGINT signals (Ctrl+C). This function performs cleanup by deleting the client's FIFO and closing the file descriptors before exiting.

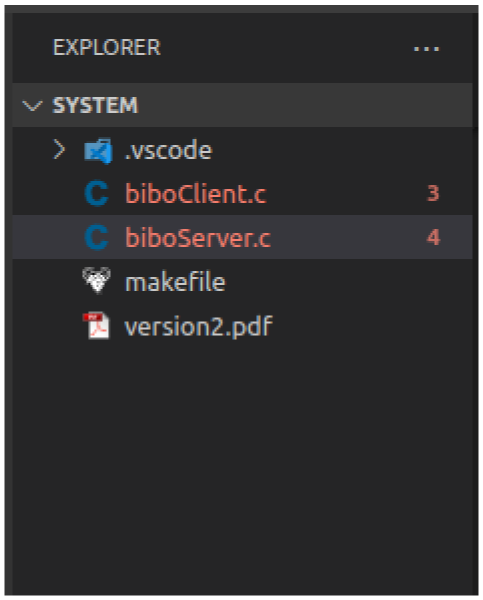


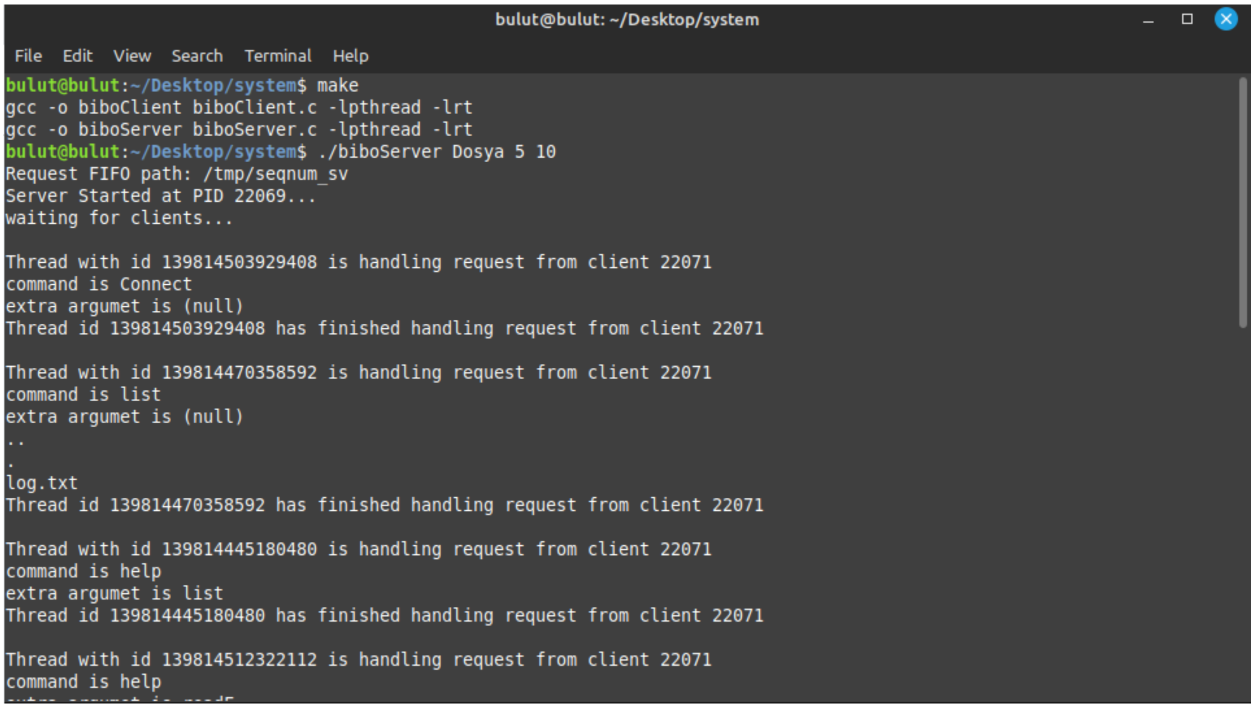
```c
void handle_sigint(int sig)
{
    printf("Caught SIGINT, exiting...\n");

    // Clean up
    unlink(clientFifoName);
    close(serverFd);
    close(clientFd);

    exit(0);
}
```

The main function checks command-line arguments: The client expects two arguments: the server command and the server PID. Initializes a request to the server: It fills a request struct with the client PID and the command. Creates a client FIFO: The FIFO's name is based on the client's PID. The client sends this name to the server when making a request. Sends the request to the server: It opens the server's FIFO in write mode and writes the request struct to it. Receives the response from the server: It opens the client's FIFO in read mode and reads the response struct from it. Then it enters a loop where it reads user commands from the standard input, sends them to the server, and handles the responses. Certain special commands like 'quit', 'killServer', 'upload', and 'download' have specific behaviors. If the user enters the 'quit' command or the loop is otherwise broken (e.g., server not found), it sends a 'quit' command to the server, removes the client's FIFO, and closes the file descriptors.

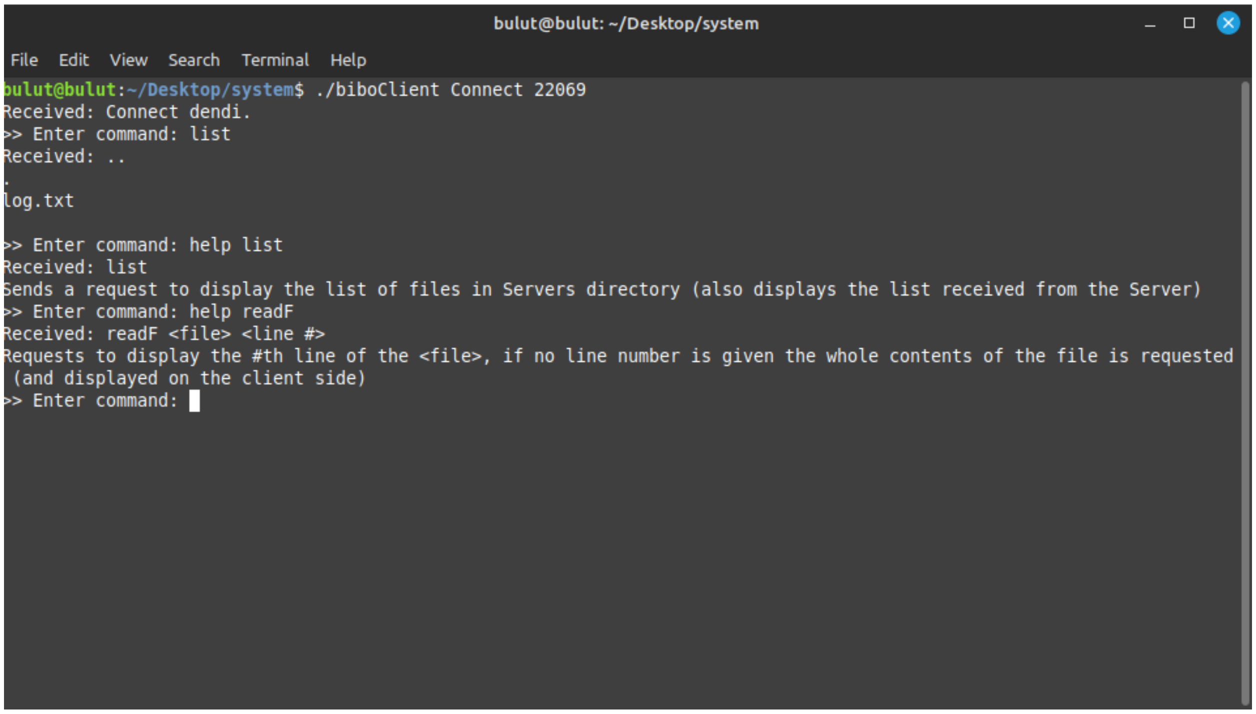
Tests:

My initial file structure is this:

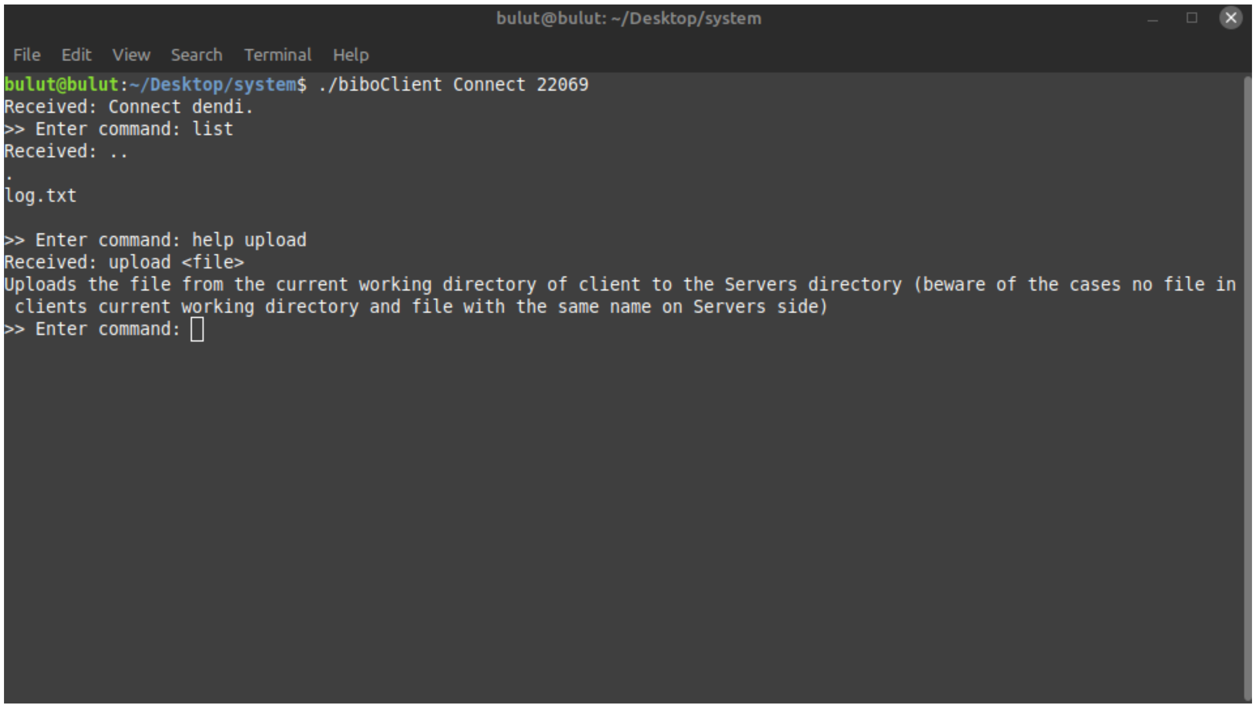



bulut@bulut: ~/Desktop/system
File Edit View Search Terminal Help

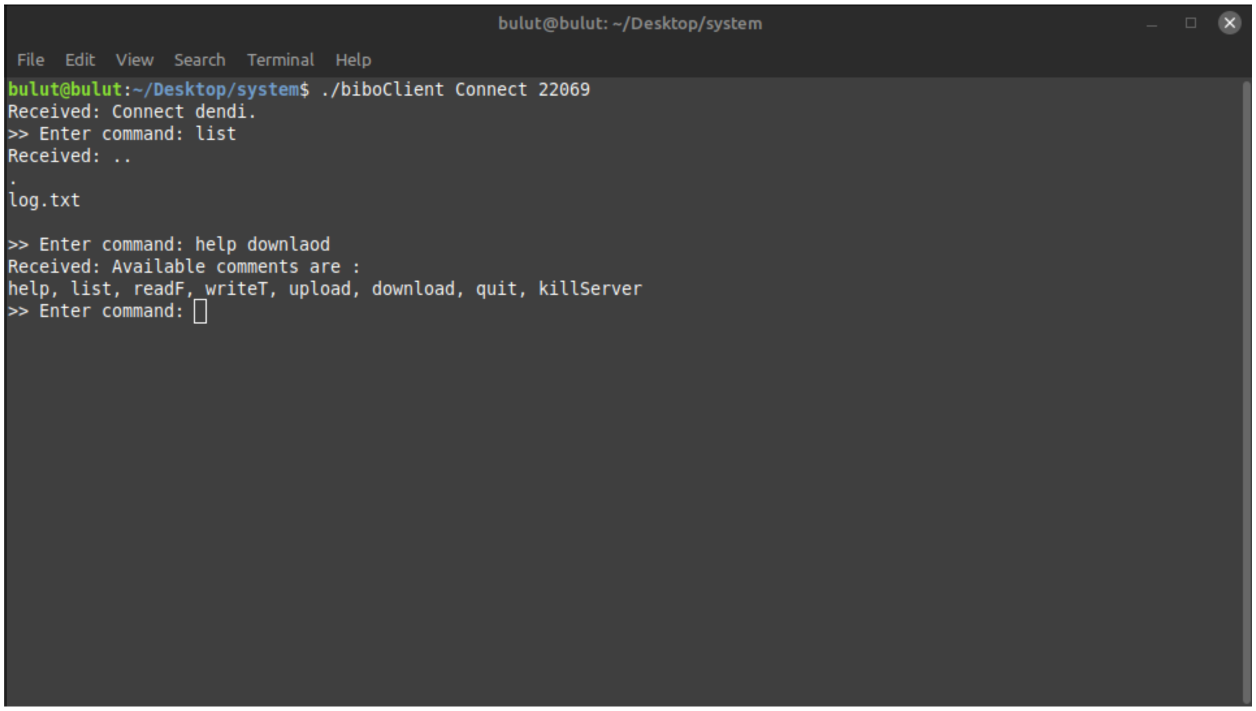
```
bulut@bulut:~/Desktop/system$ make
gcc -o biboClient biboClient.c -lpthread -lrt
gcc -o biboServer biboServer.c -lpthread -lrt
bulut@bulut:~/Desktop/system$ ./biboServer Dosya 5 10
Request FIFO path: /tmp/seqnum_sv
Server Started at PID 22069...
waiting for clients...

Thread with id 139814503929408 is handling request from client 22071
command is Connect
extra argumet is (null)
Thread id 139814503929408 has finished handling request from client 22071

Thread with id 139814470358592 is handling request from client 22071
command is list
extra argumet is (null)
..
.
log.txt
Thread id 139814470358592 has finished handling request from client 22071

Thread with id 139814445180480 is handling request from client 22071
command is help
extra argumet is list
Thread id 139814445180480 has finished handling request from client 22071

Thread with id 139814512322112 is handling request from client 22071
command is help
```


bulut@bulut: ~/Desktop/system
File Edit View Search Terminal Help

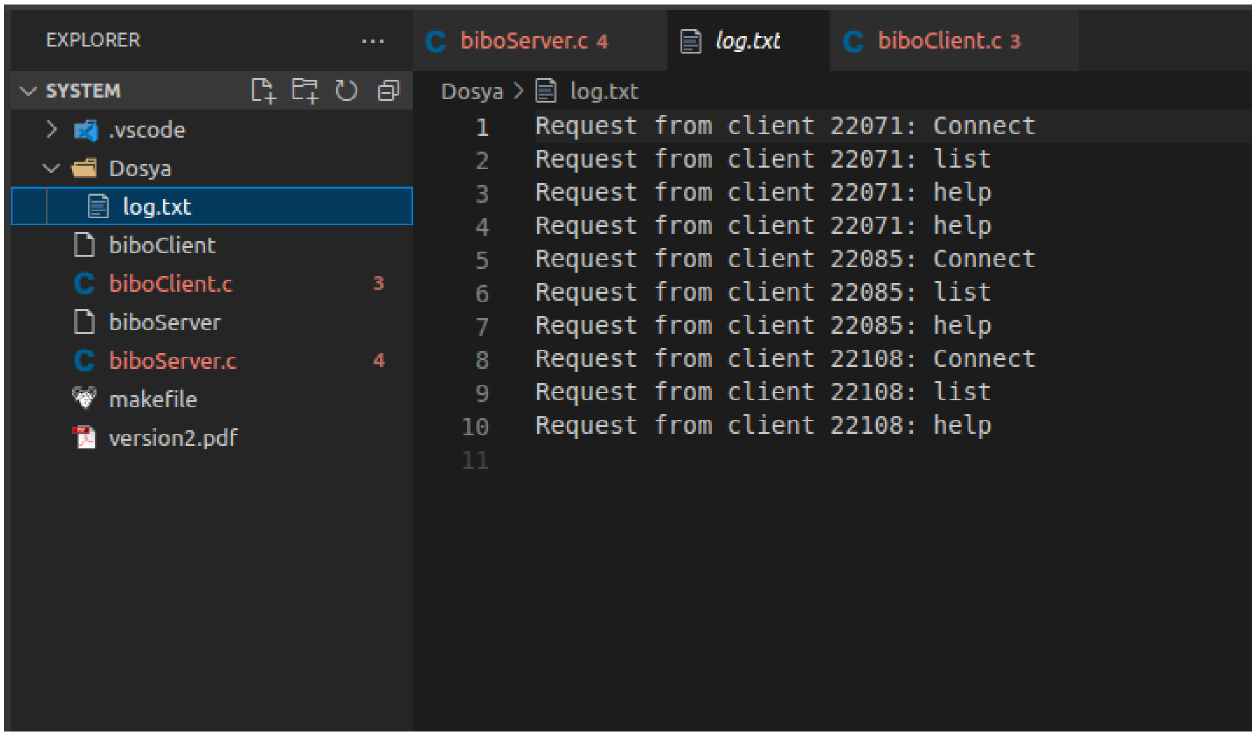
```
bulut@bulut:~/Desktop/system$ ./biboClient Connect 22069
Received: Connect dendi.
>> Enter command: list
Received: ..
.
log.txt

>> Enter command: help list
Received: list
Sends a request to display the list of files in Servers directory (also displays the list received from the Server)
>> Enter command: help readF
Received: readF <file> <line #>
Requests to display the #th line of the <file>, if no line number is given the whole contents of the file is requested
 (and displayed on the client side)
>> Enter command: 
```


bulut@bulut: ~/Desktop/system
File Edit View Search Terminal Help

```
bulut@bulut:~/Desktop/system$ ./biboClient Connect 22069
Received: Connect dendi.
>> Enter command: list
Received: ..
.
log.txt

>> Enter command: help upload
Received: upload <file>
Uploads the file from the current working directory of client to the Servers directory (beware of the cases no file in
 clients current working directory and file with the same name on Servers side)
>> Enter command: 
```


bulut@bulut: ~/Desktop/system
File Edit View Search Terminal Help

```
bulut@bulut:~/Desktop/system$ ./biboClient Connect 22069
Received: Connect dendi.
>> Enter command: list
Received: ..
.
log.txt

>> Enter command: help downlaod
Received: Available comments are :
help, list, readF, writeT, upload, download, quit, killServer
>> Enter command: 
```



```
Request from client 22071: Connect
Request from client 22071: list
Request from client 22071: help
Request from client 22071: help
Request from client 22085: Connect
Request from client 22085: list
Request from client 22085: help
Request from client 22108: Connect
Request from client 22108: list
Request from client 22108: help
```

This is the log file. It is common to all clients.

The errors that you see are not a problem. They are sigaction errors. But the code is working. I think it is because of the visual studio code environment.

Conclusion

As a conclusion, I created a concurrent file access system for the client and server. I also created a makefile. In this project I learned how to use Fifos and semaphores. I tried to manage clients with threads. Finally, I have 3 files. First one is biboServer.c, second one is biboClient.c and last one is makefile.