

Gebze Technical University

Department Of Computer Engineering

CSE 344 Spring 2023

System Programming

Midterm

Due Date: 17.05.2023

Abdurrahman Bulut

1901042258

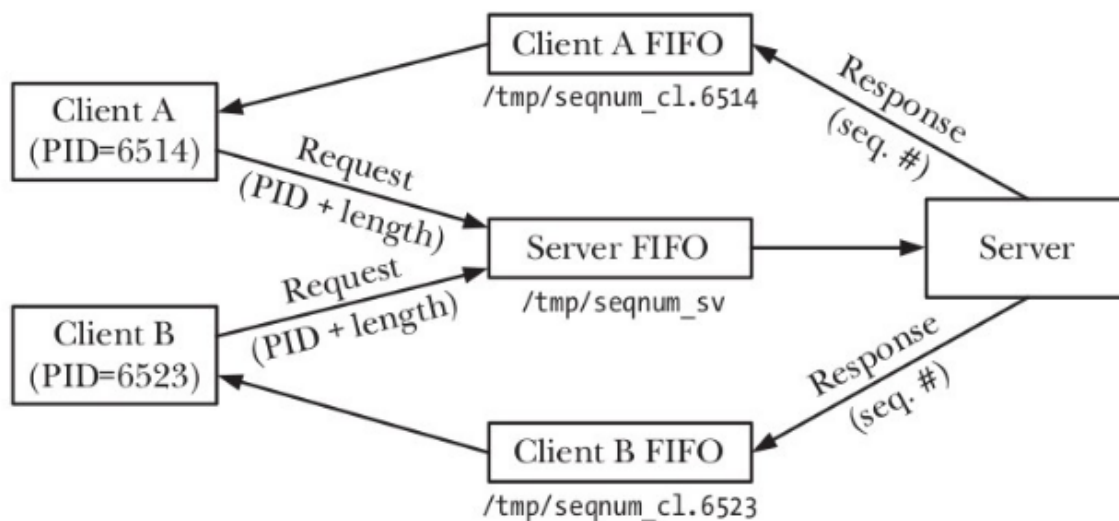
## Introduction

In this project, we are 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.

This project is about inter-process communication and multiple synchronization primitives. I used Fifos for IPC and semaphores for synchronization.

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. It has a master server process that controls client connections and creates a child process for each client that is linked to it. A queue is used by the server to control client connections. 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.

In order to respond to the requests of its linked clients, each child process runs independently. 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.

```
15
16 #define SERVER_FIFO "/tmp/seqnum_sv"
17 #define CLIENT_FIFO_TEMPLATE "/tmp/seqnum_cl.%d"
18 #define CLIENT_FIFO_NAME_LEN (sizeof(CLIENT_FIFO_TEMPLATE) + 20)
19 #define BUFFER_SIZE 1024
20 #define MAX_QUEUE_SIZE 1024
21 #define MAX_CLIENTS 2048
22 #define LOG_FILE "log.txt"
23 #define SHARED_MEM_SIZE (sizeof(sem_t) + MAX_CLIENTS * sizeof(int))
24
25 sem_t semaphore;
26
27 struct request
28 {
29     pid_t pid;
30     char command[BUFFER_SIZE];
31     int command_len;
32     int connected; // 0 for not connected, 1 for connected
33 };
34
35 struct response
36 {
37     int status;
38     char message[BUFFER_SIZE];
39 };
40
41 typedef struct
42 {
43     int pid;
44     int connected; // 0 is waiting, 1 is connected, -1 is empty.
45 } Client;
46
47 Client clients[MAX_QUEUE_SIZE];
48
49
50 pid_t child_pids[MAX_CLIENTS]; // Array to store child process pids
```

I created request and response structures. A client will send a request to the server with its pid. Variables such as “max\_clients”, “current\_client”, “serverFd”, “dummyFd”, “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.

```
if (sem_init(&semaphore, 1, 1) == -1)
{
    perror("sem_init");
    exit(1);
}
```

A semaphore is initialized using `sem_init()`. This semaphore will be used to synchronize access to shared resources, such as the client queue.

The server will work with common “./biboServer Cloud 3”. The cloud word is the directory. 3 is maximum number of clients that can connect to the server at a time. If more than this number client tries to connect to the server. The server puts them to 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()`.

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

```

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);
    }
}

```

```

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.

As last steps the server creates a new process to handle each client request. This enables the server to manage numerous client requests simultaneously. After handling the client request, the child process ends. More client requests are still being awaited by the parent process. The server looks to see if any child processes have ended. The server locates the appropriate client slot and modifies the client status in the shared memory if a child process has terminated. The client's disconnect is then indicated by a message printed after that. Lastly, 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`. `clientFd` 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.

```
#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 clientFifo[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
};

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.



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:

```
bulut@bulut:~/Desktop/system$ ./biboServer Dosya 5
Request FIFO path: /tmp/seqnum_sv
Server Started at PID 190122...
waiting for clients...

client pid: 190196
server pid: 190122
client connected: 0
client command: list
command listargs (null)client pid: 190196
server pid: 190122
client connected: 0
client command: help readF
command helpargs readFclient pid: 190392
server pid: 190122
client connected: -2086690464
client command: list
command listargs (null)client pid: 190392
server pid: 190122
client connected: -2086690464
client command: help upload
command helpargs upload
```

```
bulut@bulut:~/Desktop/system$ ./biboClient Connect 190122
>> fifo path for this client: /tmp/seqnum_cl.190196
You are Connected
Status : 1
>> Enter command: list
command: list
Server responded with status 0 and message: ..
.

>> Enter command: help readF
command: help readF
Server responded with status 0 and message: readF <file> <l
ine #>
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$ ./biboClient Connect 190122
>> fifo path for this client: /tmp/seqnum_cl.190392
You are Connected
Status : 1
>> Enter command: list
command: list
Server responded with status 0 and message: ..
.

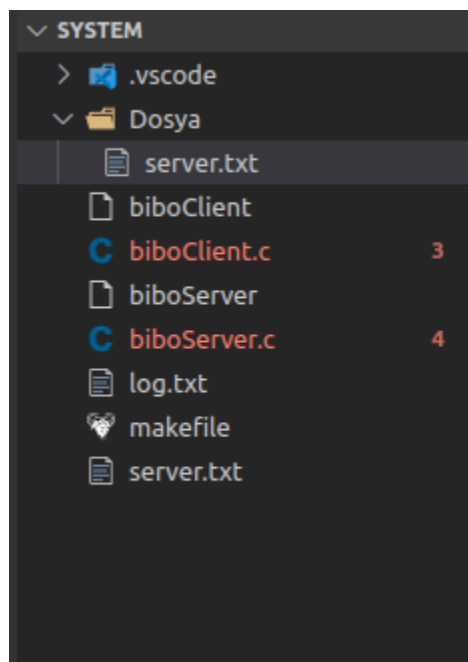
>> Enter command: help upload
command: help upload
Server responded with status 0 and message: upload <file>
Uploads the file from the current working directory of cli
ent to the Servers directory (beware of the cases no file
in clients current working directory and file with the sam
e name on Servers side)
>> Enter command: █
```

```
PROBLEMS 7 OUTPUT DEBUG CONSOLE TERMINAL
Server Started at PID 190122...
Waiting for clients...

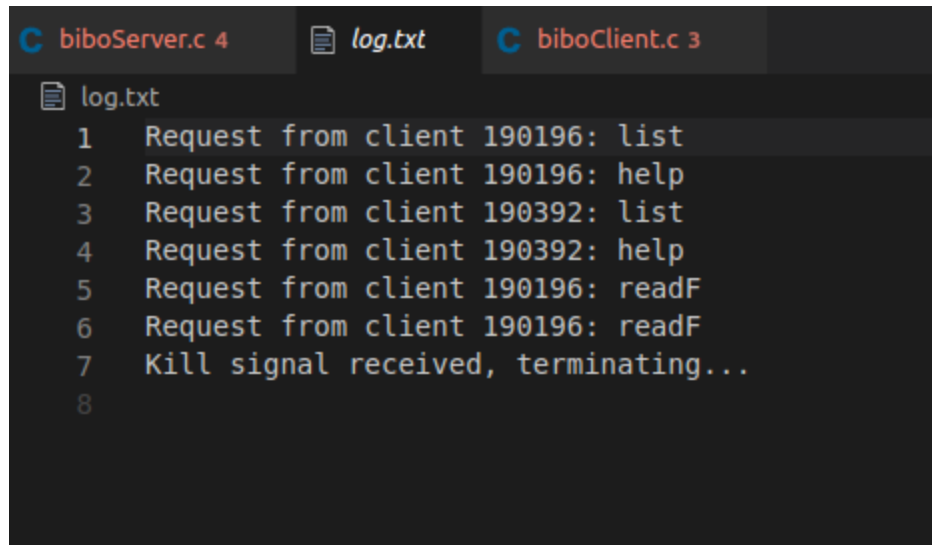
Client pid: 190196
Server pid: 190122
Client connected: 0
Client command: list
Command listargs (null)client pid: 190196
Server pid: 190122
Client connected: 0
Client command: help readF
Command helpargs readFclient pid: 190392
Server pid: 190122
Client connected: -2086690464
Client command: list
Command listargs (null)client pid: 190392
Server pid: 190122
Client connected: -2086690464
Client command: help upload
Command helpargs uploadclient pid: 190196
Server pid: 190122
Client connected: 0
Client command: readF 2
Write to client FIFO: Bad file descriptor
Command readFargs 2

bulut@bulut:~/Desktop/system$ ./biboClient Connect 190122
>> fifo path for this client: /tmp/seqnum_cl.190196
You are Connected
Status : 1
>> Enter command: list
command: list
Server responded with status 0 and message: ..
.

>> Enter command: help readF
command: help readF
Server responded with status 0 and message: 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: readF 2
command: readF 2
Server responded with status 0 and message: Failed to open
file '2'
>> Enter command: download server.txt
command: download server.txt
```



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.

A screenshot of a code editor with three tabs at the top: 'biboServer.c 4', 'log.txt', and 'biboClient.c 3'. The 'log.txt' tab is active, showing a list of log entries. The entries are numbered 1 through 8. Lines 1-6 show requests from clients 190196 and 190392 for 'list', 'help', and 'readF'. Line 7 shows a 'Kill signal received, terminating...' message. Line 8 is empty.

```
log.txt
1 Request from client 190196: list
2 Request from client 190196: help
3 Request from client 190392: list
4 Request from client 190392: help
5 Request from client 190196: readF
6 Request from client 190196: readF
7 Kill signal received, terminating...
8
```

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

## 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. Finally, I have 3 files. First one is server.c, second one is client.c and last one is makefile.