CSE344

Midterm

Report

1901042676

Mustafa Mercan

Assignment:

In this assignment, we are asked to create a server and a client, where we need to handle the requests made by the client on the server side and send the responses back to the client. Additionally, specific operations are required to be performed, and these operations should be carried out by the server.

Server Side:

The server is expected to operate as "biboServer <dirname> <max. #ofClients>". Here, "dirname" represents the file on which the client will perform its operations, and "max. #ofClients" represents the maximum number of clients that can connect to this server.

In the server part, a separate child process should be created for each client while performing operations, and when a client exits, this event should be recorded in the log file.

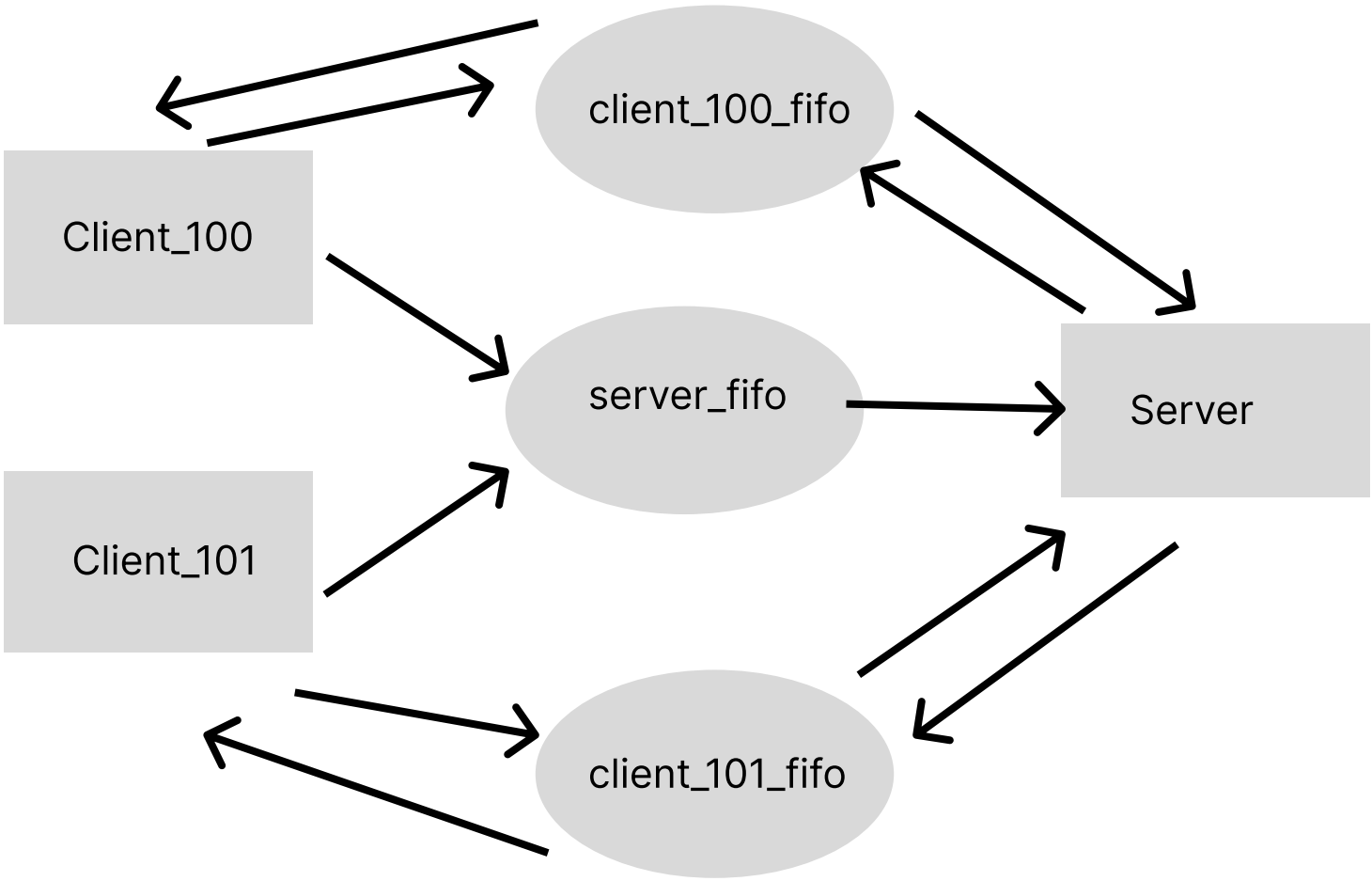
Client Side:

The client is expected to operate as "biboClient <Connect/tryConnect> ServerPID". "ServerPID" represents the process ID of the server to connect to. If the "Connect" parameter is used, the client attempts to establish a connection. If the number of connected clients is equal to or exceeds the maximum client limit, the client is added to the client queue and waits for its turn to connect. If the "tryConnect" parameter is used and the maximum client limit has been reached, the client terminates without establishing a connection.

Approach to the Problem:

In this assignment, we are asked to create a server and a client, where we need to handle the requests made by the client on the server side and send the responses back to the client. Additionally, specific operations are required to be performed, and these operations should be carried out by the server.

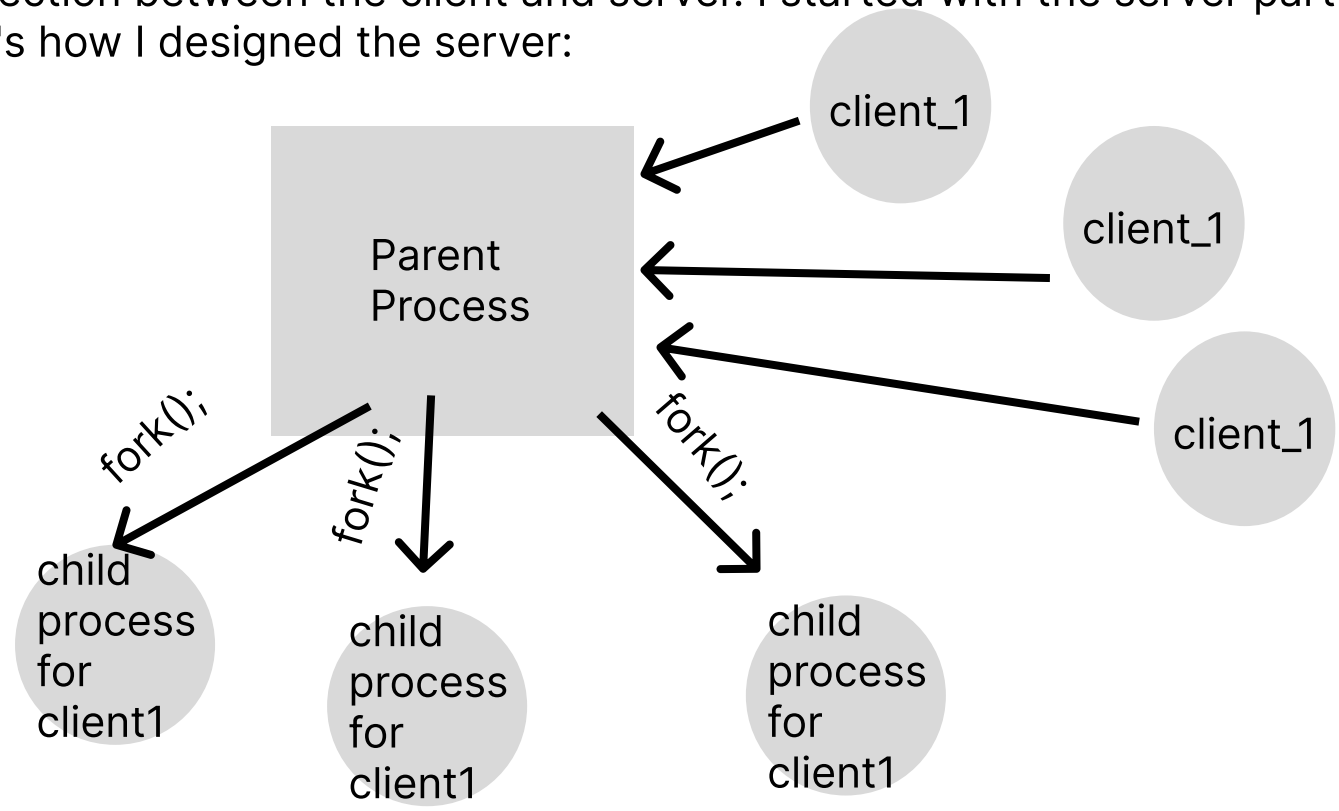
Firstly, I have created a system to solve the problem. The system I have developed works as follows:



I have decided to establish communication between the client and server using FIFO files. When the server starts running, it creates a FIFO file. The purpose of this FIFO file is to store the client's process ID (PID) when the client starts running, and to store the server's PID provided as a parameter. The server then retrieves this data and compares the provided server PID with its own PID. If the PID numbers match, the client establishes the connection. If they don't match, a response is sent to the client through a dedicated FIFO file. Based on these conditions, the client either continues running or terminates the process.

After designing this structure, I decided to create a file organization based on the Divide and Conquer principle. I divided the problem into smaller parts and decided to organize the files accordingly.
Firstly, I created the files server.c and client.c. These files contain the code for running the server and client, respectively.
Next, since the task requires multiple commands, I created another file called command_function.c. In this file, I implemented functions to process the requests sent from the client on the server side and generate the corresponding responses based on the output.
I also created a file named command_utils.c, which contains helper functions used within command_functions.c.
Lastly, I created the utils.c file. This file contains utility functions that perform basic operations.

After completing all these steps, my first goal was to establish a connection between the client and server. I started with the server part. Here's how I designed the server:



According to this design model, the server verifies whether the serverPID parameter is correctly provided by each client. If it is correct, the server performs a fork() operation within the parent process, creating a child process. This child process becomes dedicated to the connected client, handling request/response tasks.

After completing this design, I proceeded to the coding phase of server.c. First, the server starts by receiving the max_client value. Then, it checks if the file provided as a parameter exists in the directory. If it doesn't exist, the server creates the file. Once this step is completed, I began working on creating the content of the client.c file.

On the client side, I first created variables to hold the necessary paths. These paths allowed me to write data to the FIFO file created by the server. I wrote the client's PID and the server PID obtained as a parameter to the server_fifo. Then, I read and parsed this data on the server side. After parsing, I compared the provided serverPID with the actual PID of the server. If the value was correct, I printed "Client connected" to the screen and proceeded with the fork operation.If the serverPID was entered incorrectly, I handled the situation by sending an appropriate response and printing it on the screen.

After completing these steps, I created the dedicated FIFO file for the client. Now, the communication between the client and the child process will be conducted through this FIFO file. On the client side, we can now start receiving input from the user. As the user enters input, the corresponding command is written to the client's dedicated FIFO file. Then, the child process waits for data to arrive in this file. Once data is received, the child process reads the command, parses it, and calls the necessary function accordingly.

After processing the command received from the client, a response is generated based on the specific situation. This response is written to the client's dedicated FIFO file, and then it is read by the client and displayed on the client's screen.

After successfully implementing the basic functionalities, I attempted to disconnect the client and notify the server about it. For this purpose, I followed the following approach. On the server side, I defined a signal handler with signal(SIGUSR1, decrease_client);. This signal handling serves the following purpose: when the client enters the "quit" command, it sends a SIGUSR1 signal to the server. Upon receiving this signal, the parameter that keeps track of the total number of connected clients is updated, and the child process handles this command by deleting the dedicated FIFO file associated with that client.

Additionally, I used this signal to decrement the semaphore I created when the client process terminates. This way, the clients waiting in line to connect to the server can establish the connection one by one as users disconnect from the server.This approach ensures that clients waiting in the queue can establish connections in a sequential manner as users disconnect from the server.

After completing this part, I focused on establishing the connection on the client side based on the connect or tryConnect parameters. First, I implemented two signal handling operations. They are as follows:

```
signal(SIGUSR1,queue_connect);
signal(SIGUSR2,queue_connect);
```

I implemented the queue_connect signal handler to perform actions based on the connect type. On the server side, when the initial connection is established, the client receives either the SIGUSR1 or SIGUSR2 signal. The SIGUSR1 signal indicates that the maximum number of clients (max_client) has been reached. If the connect_type is "connect," the client prints an informational message and then enters a pause() state. If a client disconnects from the server, the client receives the SIGUSR1 signal, which ends the pause() state. If the connect_type is "tryConnect," the client prints an informational message without entering the pause() state and then terminates.

```c
void queue_connect(int pid)
{
    if(pid == SIGUSR1)
    {
        if(strcmp(connect_type,"connect") == 0)
        {
            printf("The queue is full. Waiting for the queue to be available.\n");
            pause();
        }
        else if(strcmp(connect_type,"tryConnect") == 0)
        {
            printf("The queue is full.The client is terminating.\n");
            exit(1);
        }
        else
        {
            printf("Please enter connect/tryConnect\n");
            exit(1);
        }
    }
    else if(pid == SIGUSR2)
    {
        printf("Connection established.\n");
    }
}
int main(int arg, char**args)
```

To handle the termination of the client when the user presses Ctrl+C, and to ensure that the server terminates the corresponding child process and allows the next client in the queue to connect, I created the following signal:

```c
signal(SIGINT,quit_signal);
```

As a result, an informational message is printed to the client, and a signal is sent to the server. This leads to the termination of the child process on the server side and the termination of the client process. Additionally, a client log is written.

In the client side, when the "killServer" command is entered, a SIGINT signal is sent to the server. This causes the server to terminate, along with all its child processes.

```c
else if(strcmp(input,"killServer") == 0)
{
    kill(atoi(args[2]),SIGINT);
    printf("Server is being terminated\n");
    return 1;
}
```

On the server side, I handled the SIGINT signal to ensure the termination of all child processes.

```
signal(SIGINT,quit_signal);
```

```
void quit_signal(int pid)
{
    printf("Terminating the program, child process and clients\n");
    int parent_pid = getpid();
    kill(-parent_pid,SIGTERM);
    for(int i = 0 ; i < current_client_number ; i++)
    {
        kill(pid_client[i],SIGTERM);
    }

    exit(1);
}
```

"After completing all these processes, I have successfully completed my project."