CSE344 HOMEWORK 4 REPORT

Ayşe Begüm Nur

1901042613

Introduction:

To execute this program, four command-line arguments are required. Here is a clearer description of each argument:

1. The first argument: The program executable itself.
2. The second argument: The path to the new server directory. This directory will be used by the server to store files or perform operations related to the server's functionality.
3. The third argument: The maximum number of clients that the server can handle concurrently. This determines the limit on the number of client connections the server can accept and process simultaneously.
4. The fourth argument: The size of the thread pool, which specifies the number of worker threads that will be created by the server. These threads will be responsible for handling client requests in parallel.

In summary, when running the program, you would provide the executable name, the path to the server directory, the maximum number of concurrent clients, and the thread pool size as command-line arguments.

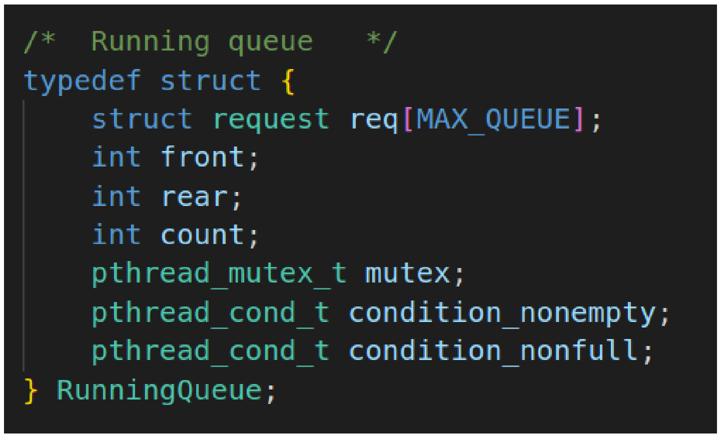
1) Server and Threads

I implemented a thread pool for handling client requests in a server. To accommodate multiple clients, I made changes to the structure by introducing two queue structures: a "request queue" for holding incoming connection requests and a "running queue" for managing the requests being processed. The request queue is modified only by the main thread, while the running queue is accessed by all threads in the pool.

To ensure thread safety, I added a mutex and two condition variables for the running queue functions (enqueue and dequeue).

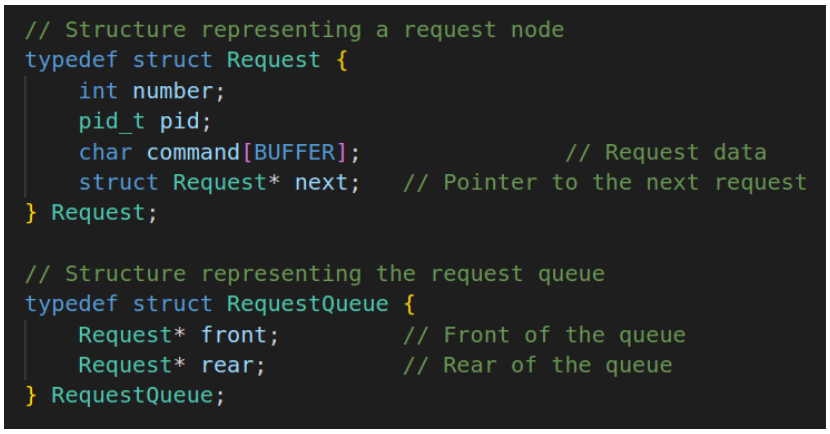
Here's how the process works:

1. The main thread receives a client's connection request from the server fifo and adds it to the request queue (provided that the running queue is not full and the request is not a "tryConnect" request).
2. In the critical section, if the number of running requests is less than the maximum allowed clients, the main thread dequeues the first element from the request queue and adds it to the running queue.
3. To prevent race conditions between the main thread and worker threads, a mutex and condition variable are used. The main thread locks the mutex and checks if the running requests exceed the maximum allowed clients. If not, it adds a new element to the running queue. If the limit is reached, the main thread waits on the condition variable until a worker thread receives a disconnect request from a client and signals the condition variable.



```
/*  Running queue   */
typedef struct {
    struct request req[MAX_QUEUE];
    int front;
    int rear;
    int count;
    pthread_mutex_t mutex;
    pthread_cond_t condition_nonempty;
    pthread_cond_t condition_nonfull;
} RunningQueue;
```

Running Queue Structure

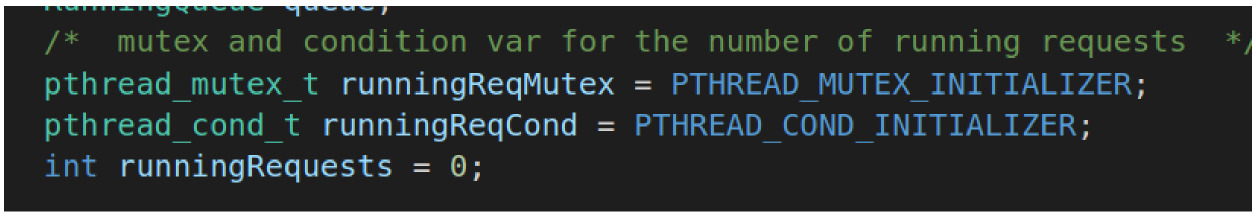


```c
// Structure representing a request node
typedef struct Request {
    int number;
    pid_t pid;
    char command[BUFFER];               // Request data
    struct Request* next;   // Pointer to the next request
} Request;

// Structure representing the request queue
typedef struct RequestQueue {
    Request* front;         // Front of the queue
    Request* rear;          // Rear of the queue
} RequestQueue;
```

Request Queue Structure

Additionally, to avoid race conditions, I used a separate mutex for the "readF" and "upload" comments, as well as for the "writeT" and "download" comments.



```c
RunningQueue queue;
/*  mutex and condition var for the number of running requests  */
pthread_mutex_t runningReqMutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t runningReqCond = PTHREAD_COND_INITIALIZER;
int runningRequests = 0;
```

runningRequest variable, mutex and condition variable

For communication between the server, threads, and clients:

- The server opens a fifo to receive the first connection requests from clients.
- Each client opens a unique fifo to read responses from the worker thread.
- Each worker thread opens a unique server fifo for a specific client, where it reads client requests and the client writes its requests.

2) Client Side

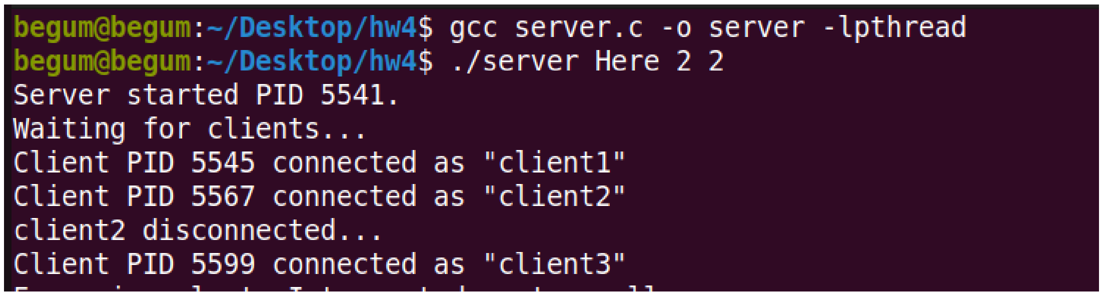
Implementing the client side was relatively straightforward. The client's main tasks involve obtaining requests from the standard input, writing the initial connection request to the server fifo, and retrieving responses from the client fifo. s

Here's a clearer description:

1. The client reads requests from the standard input (stdin).
2. It writes the first connection request to the server fifo, indicating its intention to establish a connection with the server.
3. If the server accepts the connection request, the client proceeds to open a new fifo (thread fifo). This fifo serves as the communication channel for sending actual requests to the server.
4. The client then writes subsequent requests to the thread fifo.
5. Meanwhile, the client constantly listens for responses from the server, reading them from the client fifo.

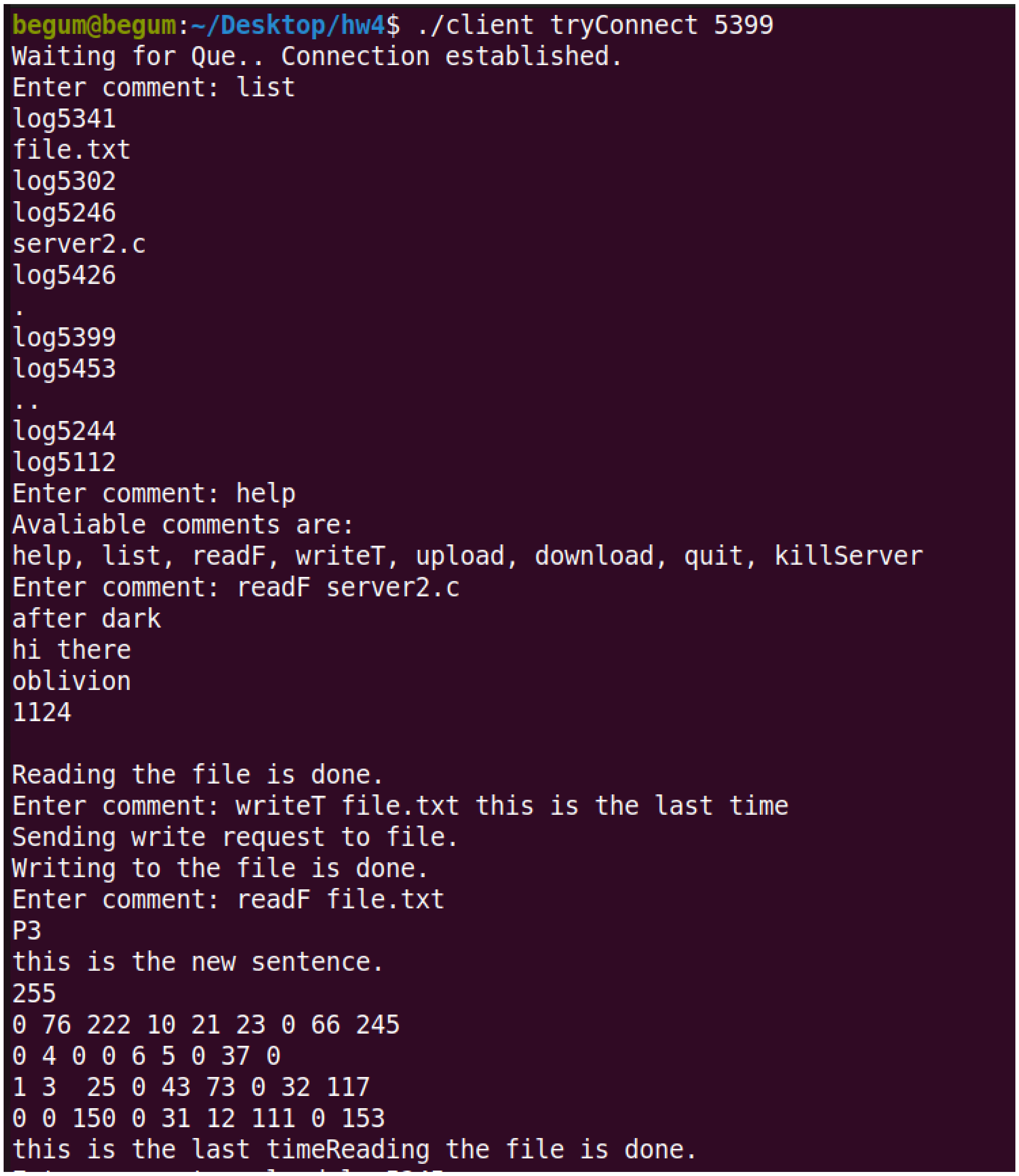
Overall, the client's role mainly revolves around gathering requests, initiating the connection with the server, creating a dedicated fifo for further communication, and exchanging requests and responses through the respective fifos.

3) Tests



```
begum@begum:~/Desktop/hw4$ gcc server.c -o server -lpthread
begum@begum:~/Desktop/hw4$ ./server Here 2 2
Server started PID 5541.
Waiting for clients...
Client PID 5545 connected as "client1"
Client PID 5567 connected as "client2"
client2 disconnected...
Client PID 5599 connected as "client3"
```

Server Output



```
begum@begum:~/Desktop/hw4$ ./client tryConnect 5399
Waiting for Que.. Connection established.
Enter comment: list
log5341
file.txt
log5302
log5246
server2.c
log5426
.
log5399
log5453
..
log5244
log5112
Enter comment: help
Avaliable comments are:
help, list, readF, writeT, upload, download, quit, killServer
Enter comment: readF server2.c
after dark
hi there
oblivion
1124

Reading the file is done.
Enter comment: writeT file.txt this is the last time
Sending write request to file.
Writing to the file is done.
Enter comment: readF file.txt
P3
this is the new sentence.
255
0 76 222 10 21 23 0 66 245
0 4 0 0 6 5 0 37 0
1 3  25 0 43 73 0 32 117
0 0 150 0 31 12 111 0 153
this is the last timeReading the file is done.
```

Client Output