

Md Sarwar Hossain

Id: 161044121

Client-Server Communication in C Programming

**system architecture:**

This is continuation of the midterm project. This time I used Threads instead of process. In a server with a thread pool, the server initializes a fixed number of worker threads and starts listening for client requests. These threads initially wait for tasks to be available. Then Server opens a FIFO as a medium of communication with the client. When a client request comes in, the server needs to handle the request. Rather than creating a new thread for each request (which can be expensive in terms of resources), the server will use one of the already existing threads in the thread pool. When a request arrives, the server will put the request into a job queue. This queue is shared among all threads in the pool. Each request is a job for the threads. The threads in the pool are in a continuous loop where they check the job queue for any available jobs. If there's a job in the queue, a thread will take it off the queue and start processing it. If there are no jobs in the queue, the thread will wait until a job becomes available. When a worker thread takes a request from the job queue, it processes the request. This might involve reading data from a database, performing some computations, or anything else that the server needs to do. When the thread is done processing the request, it sends a response back to the client. After sending the response, the thread is now free to process more requests, so it goes back to the job queue to look for more work. This design allows a server to handle multiple requests concurrently, while limiting resource usage by reusing a fixed set of worker threads. It's more efficient than creating a new thread for each request, especially for servers that handle a large number of requests. We can say that this design is much more sophisticated than using a dedicated process for each client request.

## Design:

As I designed this system on top of the existing system many of the design patterns are same. Although there are huge differences, how they function. First of all, let's talk about the server side.

At first the server program initializes some structures based on the command line inputs. No. of thread & no. Of max clients. After getting these two I initialize two structures, 'Connclients' and 'pthread\_t' which uses array under the hood. The connclients keeps track of no. Of connected clients on the other hand pthread\_t was used for thread pool.


After initializing these two structures, I create and switch to the server directory provided by command line arguments. Server files will be in this directory. Then I initialized signal handling, specifically SININT, SIGTERM & SIGCHLD. When these signals will arrive we perform some cleanup to give the resources back to the system.

Now server starts listening to incoming client requests in a infinite loop. We will get back how to kill a server. At first we read the request, analyze what clients want then response accordingly.

We were supposed to put the user in waiting queue if the max client is reached. However, this server does not accept anyone if max client is reached.

So if there is a space in connectionpool (a hashset would be a better choice here, as we need some fast operation like frequent lookup for clients) we accept the client. If the client is already connected we directly proceed in serving them. With some condition we determine if a client is eligible for service.

Meanwhile, We started our worker threads. They are constantly looking for job to assign to them. If there is no job they are put to sleep. This has been done by mutex + condition variable (monitor).



After server received the request from client and find that it's eligible, the server add the request in job\_queue so the threads can take over. So server is responsible for queuing job for the threads. And worker threads execute the task.

Let's now talk about client side before we focus on other design parts. The client program is responsible for sending request to the server using a Custom Protocol (more on that later). The server uses the same protocol to talk back to the client. After verifying the server connection, the client opens a FIFO (determined by template). The server program knows about the client fifo and response to the client through that fifo. So, after sending requests, the client program waits for response before sending another request. If server can fulfill clients request it answers positively with resources else it fails. The client program runs till a user decides to quit.

Protocols are base of a communication, without this the client won't understand what server is saying vice versa. There is a very simple protocol, which I used to make sure the understanding between client and server is stable.

```

typedef enum response_protocol{
    connEstablished=1,
    connWaiting=2,
    connClosed=3,
    connDeclined=4,
    resBuffer=5,
    resFail=6,
    resComplete=7,|
}resprotocol_t;

typedef enum reqprotocol{
    invalid=-1,
    Connect=1,
    tryConnect=2,
    help=3,
    list=4,
    readF=5,
    writeT=6,
    upload=7,
    download=8,
    quit=9,
    killServer=10,
}

```

Now for the request and response object I used this protocol.

```

typedef struct request{
    pid_t client_pid;
    reqprotocol_t action_code;
    char cmd[CMD_LEN];
} request_t;

typedef struct response{
    resprotocol_t rescode;
    char body[RESPONSE_LEN];
}response_t;

```

There is a server config file. These file is created so the client can read server configuration and connect easily to the server.

```
struct server_config{
    int serverpid;
    char serverdir[100];
};
```

## Commands

There are no. Of request that the server can handle. The first being read and write to a file.

Since both these operations are on the files in the same directory and no. Of worker threads are interested in them , I had to be very careful in their synchronization mechanism. The problem is very much similar to reader-writer problem. Here are the some of the items I used,

```
pthread_mutex_t mutex_jqueue = PTHREAD_MUTEX_INITIALIZER;
```

```
pthread_mutex_t mutex_clients = PTHREAD_MUTEX_INITIALIZER;
```

```
pthread_mutex_t mutex_rw = PTHREAD_MUTEX_INITIALIZER;
```

```
//condition variables
```

```
pthread_cond_t cv_emptyq = PTHREAD_COND_INITIALIZER;
```

```
pthread_cond_t can_read = PTHREAD_COND_INITIALIZER;
```

```
pthread_cond_t can_write = PTHREAD_COND_INITIALIZER;
```

**The reading part is** thread-safe with the use of a mutex lock. It waits until there are no active or waiting writers, then increments the active reader count. If a specific line number is given, it reads only that line; otherwise, it reads the whole

file and sends the data line-by-line to the client process. The function also handles EOF and errors appropriately, sending corresponding messages to the client.

I used mutex lock to ensure thread safety along with condition variables. The writer thread waits until there are no active readers or writers, then increments the active writer count. If a specific line number is given, it appends data to that line; otherwise, it writes data to the end of the file. It sends the client process a "writing" message before it starts writing and a "done writing" message after it finishes.

The upload and download function however uses filelocks. This way is also effective as file locks are very powerful.

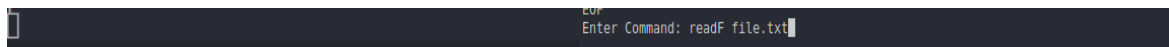
The Server keeps a log of each request. When a client quits it download a log of a clients request in Client directory.

The killServer request kill the server along with all the worker threads.

### **Drawbacks:**

Both server and client program is not robust in handling signals. The edge cases for the request are not properly implemented. So if the request is not according to the format it might generate weird output. The client program needs more improvement. The communication could be much better.

### **Running Program:**



```

cu cursus. Feugiat vivamus at augue eget arcu dictum. Ac tortor digni
ssim convallis aenean et tortor at risus. Lorem ipsum dolor sit amet
. Massa placerat dui ultricies lacus sed turpis tincidunt id. Faucib
us interdum posuere lorem ipsum dolor. Proin libero nunc consequat in
terdum varius sit amet mattis. Vel quam elementum pulvinar etiam non
quam lacus suspendisse faucibus. Viverra ipsum nunc aliquet bibendum.
Libero volutpat sed cras ornare arcu dui vivamus arcu felis. Cras se
d felis eget velit aliquet sagittis id consectetur purus. In hendreri
c gravida rutrum quisque non. Eu non diam phasellus vestibulum lorem
sed. Bibendum est ultricies integer quis.

This is the last line."hello world" sodales neque. Nec ultrices dui sapien eget mi
di. Phasellus id enim. Quis tincidunt fermentum dui. Tortor dignissim convallis aene
an. Proin sapien malesuada ornare tincidunt. Tellus in metus vulputate eu.

'This is another write test'

'This is another attempt to write'
EOF
Enter Command: █
    rtor dignissim convallis aenean et tortor at risus. Lorem ipsum dolor sit amet. Massa pla
t id. Faucibus interdum posuere lorem ipsum dolor. Proin libero nunc consequat interdum v
inar etiam non quam lacus suspendisse faucibus. Viverra ipsum nunc aliquet bibendum. Libe
rcu felis. Cras sed felis eget velit aliquet sagittis id consectetur purus. In hendrerit
lus vestibulum lorem sed. Bibendum est ultricies integer quis.

This is the last line."hello world"

'This is another write test'

'This is another attempt to write'
EOF
Enter Command: █

```

Two thread reading simultaneously

id:202405 cmd:5	This is another attempt to write
id:202405 cmd:5	EOF
id:202405 cmd:9	Enter Command: quit
Client 202405 disconnected	server: Log file created. Bye....
	o sarwar@sarwar-ThinkPad-T470s:~/Deskt