MULTI THREADED SERVER

Compilation Instructions

- To compile the code , run `g++ -o client client.cpp` in the terminal, to create a client executable.
- run the command `g++ -o server server.cpp` in the terminal to create a server executable.
- In separate terminal sessions , run the above generated executables in the order :
    1. `./server` to create a local server for the code
    2. `./client` to create multiple clients (simulation) and give the input , for client to interact with the server.

The code Implementation is explained below :

1. Client Side

1. The multiple clients are simulated using the threads, wherein each thread represents an user, to simulate a scenario such that , multiple clients , interacting with the server.

2. I created a client request object as :

```
typedef struct client_request
 {
 int i;
 pthread_t tid;
 int client_fd;
 int req_time;
 string command;
 pthread_mutex_t mutex;
 } client_request;
 vector<client_request> clients;
```

   wherein `clients` is the array of the user_threads , and stores , the related information of the user_threads.

3. Taking the **input** from the user :

```
for (int i = 0; i < num_clients; i++)
    {
        string str, time_req = "";
        getline(cin, str);
        auto itr = str.begin();
        while (*itr != ' ')
        {
            time_req += *itr;
            itr++;
```

```
        }
        itr++;
        client_request tmp;
        tmp.req_time = stoi(time_req);
        string command(itr, str.end());
        tmp.command = command;
        clients.push_back(tmp);
    }
```

say the input command is `4 insert 1 hello` , I parse the string to get the wait time first as 4 , and separate the main command from the wait time for the thread. using `stoi` , convert string object to int and initialised the client object and stored it in the list of user_clients `clients`

4. After generating the threads , for each user , we make the thread execute the `client_thread` function , and the simulation begins.

```
void *client_thread(void *(arg))
{
int idx = *(int *)arg;
clients[idx].client_fd = get_client_socket_fd();
int client_fd = clients[idx].client_fd;
int req_time = clients[idx].req_time;
string command = clients[idx].command;
sleep(req_time);
pthread_mutex_lock(&clients[idx].mutex);
int x = write(client_fd, command.c_str(), command.length());
// send command to server
if (x < 0)
{
    cerr << "Failed To communicate with the server" << endl;
    pthread_mutex_unlock(&clients[idx].mutex);
    return NULL;
}
pthread_mutex_unlock(&clients[idx].mutex);

// read response from the server

pthread_mutex_lock(&clients[idx].mutex);

string buffer;
buffer.resize(buffer_max_sz);
int byte_read = read(client_fd, &buffer[0], buffer_max_sz - 1);
buffer[byte_read] = '\0';
buffer.resize(byte_read);
if (byte_read <= 0)
{
    cerr << "Failed To communicate with the server" << endl;
    pthread_mutex_unlock(&clients[idx].mutex);
    return NULL;
}
pthread_mutex_lock(&terminal);
```

```
cout << clients[idx].i << " : " << gettid() << " : " << buffer <<
endl;
pthread_mutex_unlock(&terminal);

pthread_mutex_unlock(&clients[idx].mutex);

return NULL;
}
```

5. In the `client_thread` function , we establish a different socket for each of the use thread , and generate a socket_fd for each of the client in the function `get_client_socket_fd` which established connection for the client with the server hosted on port `8001`. This function returns the generated socket_fd for the client which is used ahead for communicating with the server.

```
int get_client_socket_fd()
{
struct sockaddr_in server_obj;
int socket_fd = socket(AF_INET, SOCK_STREAM, 0);
if (socket_fd < 0)
{
    perror("Error in socket creation for CLIENT");
    exit(-1);
}
int port_num = server_port;
memset(&server_obj, 0, sizeof(server_obj)); // Zero out structure
server_obj.sin_family = AF_INET;
server_obj.sin_port = htons(port_num); // convert to big-endian order
if (connect(socket_fd, (struct sockaddr *)&server_obj,
sizeof(server_obj)) < 0)
{
    perror("Problem in connecting to the server");
    exit(-1);
}
return socket_fd;
}
```

6. In the main thread function , we `send` command to the server and wait on the blocking call `read` until it recieves the response from the server.

7. prints the response recieved from the server on the terminal and exits the simulation.

8. **NOTE THAT : AS MULTIPLE THREADS COULD HAVE WRITTEN ON THE TERMINAL AT THE SAME TIME , I HAVE USED A MUTEX LOCK `terminal` so that at a time only one client thread could write to the terminal**

---

2. Server Side

Initialising the server

- The server is setup and hosted on the PORT_ARG defined as macro (8001) using the system calls - `bind` , `socket` , `listen` , and it basically hosts the server program and accepts the incoming client requests using the `accept` system call in the main server loop.
- This is done in the function named as `init_server_socket()`
- whenever , a new client gets connected , we get the socket_fd of the client in the main server loop , that we use to communicate b/w the client and the server , we add the given file descriptor to the queue (or `push` it). and signal the waiting threads , waiting on the conditional variable `cond_var`.

Worker Threads

- The worker threads are initialized to the count given by argument while compiling the server side.
- In basic functioning , all of them go on waiting (to avoid busy waiting I have used a conditional variable cond_var) , that is signalled and makes the worker thread wake up and execute the incoming request.
- The worker threads , pop from the `queue` of clients fd , if the queue is empty it means there is no client and they go to wait until a client is added.

```
void *worker_thread(void *arg)
{
    // int sockfd = *((int *)arg);
    char buffer[buff_sz];
    int n;
    while (1)
    {
        pthread_mutex_lock(&queue_lock);
        while (q.empty())
        {
            pthread_cond_wait(&cond_var, &queue_lock);
        }
        int *client_sockfd = q.front();
        q.pop();
        pthread_mutex_unlock(&queue_lock);
        handle_client(*client_sockfd);
    }
    return NULL;
}
```

- Once any of the worker threads , wakes up it pops the client fd from the queue and calls the handle client function that handles the client requests.
- The handle client function reads the input command from the client socket and calls the function handler with that command , to execute the command accordingly.
- function handler basically parses the iunput command using `strtok_r` etc. and executes the command and sends back the response to the client 😀.

The Queue Data Structure

- whenever , a new client connection is established , we add the user_fd to the queue , and signal the waiting threads which are waiting for serve the client requests.
- The worker threads pop these requests , and serve the client as explained above

The dictionary

- the dictionary is implemented using the array of struct and the struct object is defined as :

```
struct dictionary_node
{
string str;
int id;
int is_active;
pthread_mutex_t mutex;
};
```

where **is_active** is boolean variable , depicting whether or not the dictionary key is active or has been deleted (1 for active and 0 for inactive), lock (or mutex) is defined so that multiple operations cant be carried out on the same key node 😃.