Report

Overview :

There are 4 files in this question :

- headerq3client.h
- headerq3server.h
- client_sim.cpp
- server_prog.cpp

The program can be run as follows (client side) :

```
g++ -pthread client_sim.cpp -o client
./client
```

The program can be run as follows (server side) :

```
g++ -pthread server_prog.cpp -o server
./server X
```

X can be any integer.

Explanation for q3 :

1. headerq3client.h :

In this file, I have declared one mutex, an array for client threads and a structure for the clients.

- Clients :

```
struct clients {
    string command;             // the command associated with that client.
    int client_id;              // the id assigned to the client.
    int sleeptime;              // sleep time for that client.
    int fd;                     // socket fd.
};
```

the mutex and thread array are as follows :

```
pthread_t clientthreads[SIZE];
pthread_mutex_t outputlock;
```

2. client_sim.cpp :

We take the input from the user and parse the input accordingly and store all the parsed values inside the structure. We then spawn all the m threads for the clients into the handleclients() function. The only function that i have added in this file is the handleclients() function, rest of the functions are same as the tutorial code.

2.1 handleclients() :

In this function, we first extract the client corresponding to that thread. After acquiring the client, we get a socket for this client and make the client go to sleep (as per input).

```
curr_client->fd = get_socket_fd();
sleep(curr_client->sleeptime);
```

The socket that we received is used for a 2 way communication by the client with the server.

We can now write the command to be performed by this client into the socket, the socket is accepted by the server side using the accept() function.

```
if(write(curr_client->fd,(curr_client->command).c_str(),(curr_client->command).length())<0) {
    cerr << "Failed to SEND DATA on socket.\n";
    return NULL;
}
```

Now we also read from the socket after the server sends the response via the socket. The read here is a blocking call and thus only executes when something is sent from the server side.

we can read the data as follows :

```
int bytes = buff_sz;
std::string output;
output.resize(bytes);

int bytes_received = read(curr_client->fd, &output[0], bytes - 1);
// debug(bytes_received);
if (bytes_received <= 0)
{
    cerr << "Failed to establish communication with the server.\n";
    // return "
    return NULL;
}
```

```
// debug(output);
output[bytes_received] = '\0';
output.resize(bytes_received);
```

After the read, we just have to print the output message onto the terminal, here i have used a common lock, as at times interrupts can happen while the cout is being executed, and thus to prevent this, a lock has been used here.

```
pthread_mutex_lock(&outputlock);
cout<<curr_client->client_id<<":"<<pthread_self()<<":"<<output<<"\n";
pthread_mutex_unlock(&outputlock);
```

We are done with the client side now.

3. headerq3server.h :

In this file, I have declared a queue to store the client file decriptors, an array of mutexes for the dictionary, a mutex for the queue and a conditional variable for the queue to check if its empty or has some clients to be processed.

the declarations are as follows :

```
string dict[101];            // key val thing
queue<int> que;
pthread_mutex_t que_lock;
pthread_cond_t qempty;
pthread_t serverthreads[SIZE];
pthread_mutex_t dictmutex[SIZE];
```

4. server_prog.cpp :

In this file, we int int main() we initialise the mutex array and the que_lock and spawn the server threads into the handleworker function. We take the number of worker threads directly form the terminal. We also initialise the dictionary to empty string.

```
int i, j, k, t, n;
n = stoi(argv[1]);            // worker threads
for(int i=0;i<101;i++) {
    dict[i]="";
    pthread_mutex_init(dictmutex + i, NULL);
}
pthread_mutex_init(&que_lock,NULL);
if (pthread_cond_init(&qempty, NULL) != 0) {
    perror("pthread_cond_init() error");
    exit(1);
```

```
}
for(int i=0;i<n;i++) {
    // cout<<"spawning threads\n";
    pthread_create(serverthreads+i,NULL,handleworker,NULL);
}
```

We also add all the clients that have been read by the server into a queue. The queue contains the socket file descriptors of all the clients. We acquire the queue lock and push the socket_fd into it and unlock the queue lock. We also conditional signal to the qempty conditional variable to cond_wait in handleworker function.

```
pthread_mutex_lock(&que_lock);
que.push(client_socket_fd);
pthread_mutex_unlock(&que_lock);
pthread_cond_signal(&qempty);
```

4.1 handleworker :

In this function, we first want to acquire a client to be processed. This is done as follows :

```
pthread_mutex_lock(&que_lock);
while(que.empty()) {
    pthread_cond_wait(&qempty,&que_lock);
}
int client_fd = que.front();
que.pop();
pthread_mutex_unlock(&que_lock);
```

Every time we get a client_fd, we process it, else the server waits until a client_fd is sent.

We now have the client_fd with us, and thus we can execute the client request, for which we need to read via the socket.

```
std::string output;
int bytes = buff_sz;
output.resize(bytes);

int bytes_received = read(client_fd, &output[0], bytes - 1);
//debug(bytes_received);
if (bytes_received <= 0)
{
    cerr << "Failed to read data from socket. \n";
    goto S;
}

output[bytes_received] = '\0';
output.resize(bytes_received);
```

This output that we received now need to be parsed accordingly and needs to be passed to its respective function.

There are 5 functions that a client can ask the server to perform, these are as follows :

All these operations are done on the common dict that we had declared, thus before every operation we need to acquire its lock and release after the operation is complete.

4.1.1 Insert :

Insert is as follows :

```
if(!strcmp(commands[1],"insert")) {
    if(no_of_args!=4) {
        cout<<"Invalid no of args (insert)\n";
        goto S;
    }
    int key = atoi(commands[2]);
    pthread_mutex_lock(dictmutex+key);
    if(dict[key]=="") {
        dict[key]=commands[3];
        send_string_on_socket(client_fd,"Insertion Successful");
    }
    else send_string_on_socket(client_fd,"Key already exists");
    pthread_mutex_unlock(dictmutex+key);
}
```

4.1.2 Delete :

Delete is as follows :

```
if(!strcmp(commands[1],"delete")) {
    if(no_of_args!=3) {
        cout<<"Invalid no of args (delete)\n";
        goto S;
    }
    int key = atoi(commands[2]);
    pthread_mutex_lock(dictmutex+key);
    if(dict[key]!="") {
        dict[key]="";
        send_string_on_socket(client_fd,"Deletion successful");
    }
    else send_string_on_socket(client_fd,"No such key exists");
    pthread_mutex_unlock(dictmutex+key);
}
```

4.1.3 Update :

Update is as follows :

```
if(!strcmp(commands[1],"update")) {
    if(no_of_args!=4) {
        cout<<"Invalid no of args (update)\n";
        goto S;
    }
    int key = atoi(commands[2]);
    pthread_mutex_lock(dictmutex+key);
    if(dict[key]!="") {
        dict[key]=commands[3];
        send_string_on_socket(client_fd,dict[key]);
    }
    else send_string_on_socket(client_fd,"No such key exists");
    pthread_mutex_unlock(dictmutex+key);
}
```

4.1.4 Concat :

Concatenation is as follows :

```
if(!strcmp(commands[1],"concat")) {
    if(no_of_args!=4) {
        cout<<"Invalid no of args (concat)\n";
        goto S;
    }
    int key_1 = atoi(commands[2]);
    int key_2 = atoi(commands[3]);
    pthread_mutex_lock(dictmutex+key_1); pthread_mutex_lock(dictmutex+key_2);
    if(dict[key_1]!="" && dict[key_2]!="") {
        string temp_1 = dict[key_1];
        string temp_2 = dict[key_2];
        dict[key_1]=dict[key_1]+temp_2;
        dict[key_2]=dict[key_2]+temp_1;
        send_string_on_socket(client_fd,dict[key_2]);
    }
    else send_string_on_socket(client_fd,"Concat failed as at least one of the keys does not exist");
    pthread_mutex_unlock(dictmutex+key_1);pthread_mutex_unlock(dictmutex+key_2);
}
```

4.1.5 Fetch :

Fetch is as follows :

```
if(!strcmp(commands[1],"fetch")) {
    if(no_of_args!=3) {
        cout<<"Invalid no of args (fetch)\n";
        goto S;
    }
    int key = atoi(commands[2]);
    pthread_mutex_lock(dictmutex+key);
    if(dict[key]!="") send_string_on_socket(client_fd,dict[key]);
    else send_string_on_socket(client_fd,""“Key does not exist");
```

```
    pthread_mutex_unlock(dictmutex+key);
}
```

These are all the commands that client can request the server for.