# Multithreaded Concurrent TCP Database Server

## Student Details


| Name | Reg No. | Roll No. | Section |
|---|---|---|---|
| Tejas Hegde | 190905342 | 48 | D |
| Swapnodip Chakrabarti | 190905124 | 14 | D |
| Akshita Anand | 190905039 | 04 | D |
| Devyani Goil | 190905340 | 47 | D |


## Abstract


A simple way to go about building a concurrent TCP server would be to create a new process to handle each client connection. Each process is isolated from the other process. When the CPU has to switch between 2 processes, context switching takes place. This is expensive as the old registers need to be saved, and new registers are loaded up. We make use of threads to handle each connection. All threads belonging to the same process share the same memory and resources. This eliminates the context switch issue.
An issue while building a database is consistency of data. When mulitple operations are performed on the same data, there are chances of inconsistencies and race conditions. Databases implement complex lock mechanizms to deal with this issue. In our simple implementation we make use of a mutex lock to prevent race conditions.


**Programming language used:** C

## Implementation

Repository: https://github.com/Swapnodip/os-project.git

**Client**

```
#include <stdio.h>
#include <sys/socket.h>
#include <unistd.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <string.h>
#include <pthread.h>
#include "os_proj_types.c"
#define PORT 8000

void initRequest(struct data *request)
```

```
{
    request->create = 0;
    request->delete = 0;
    request->read = 0;
    request->update = 0;
}

void displayUser(struct User user)
{
    if (user.id == -1)
    {
        printf("Invalid operation\n");
        return;
    }
    printf("ID: %d\nName: %s\nEmail: %s\n", user.id, user.name, user.email);
}

int main()
{
    int sd, n, id;
    struct sockaddr_in seraddress;
    struct data req;
    int requestType;
    struct data request, response;
    struct User user;
    char buf[30];
    printf("Enter 1 to create, 2 to read, 3 to update, 4 to delete\n");
    initRequest(&request);
    scanf("%d", &requestType);
    if (requestType == 1)
    {
        printf("Enter name, email and password of new user\n");
        scanf("%s", buf);
        strcpy(user.name, buf);
        scanf("%s", buf);
        strcpy(user.email, buf);
        scanf("%s", buf);
        strcpy(user.password, buf);
        request.create = 1;
    }
    else if (requestType == 2)
    {
        printf("Enter id of user to read\n");
        scanf("%d", &id);
        user.id = id;
        request.read = 1;
    }
    else if (requestType == 3)
    {
        printf("Enter ID, name, email, and password of user to be updated\n");
        scanf("%d", &user.id);
        scanf("%s", buf);
```

```
        strcpy(user.name, buf);
        scanf("%s", buf);
        strcpy(user.email, buf);
        scanf("%s", buf);
        strcpy(user.password, buf);
        request.update = 1;
    }
    else if (requestType == 4)
    {
        printf("Enter id of user to read\n");
        scanf("%d", &id);
        user.id = id;
        request.delete = 1;
    }
    request.user = user;
    sd = socket(AF_INET, SOCK_STREAM, 0);
    seraddress.sin_family = AF_INET;
    seraddress.sin_addr.s_addr = INADDR_ANY;
    seraddress.sin_port = htons(PORT);
    connect(sd, (struct sockaddr *)&seraddress, sizeof(seraddress));
    n = write(sd, (struct data *)&request, sizeof(request));
    if (n < 0)
    {
        printf("Write error\n");
    }
    n = read(sd, (struct data *)&response, sizeof(response));
    if (n < 0)
    {
        printf("Read error\n");
    }
    printf("---------- RESPONSE ----------\n");
    displayUser(response.user);
    close(sd);
}
```

**Server**

```
#include <stdio.h>
#include <sys/socket.h>
#include <unistd.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <string.h>
#include <pthread.h>
#include "os_proj_types.c"
#define PORT 8000
#define MAX_USERS 10

struct User users[MAX_USERS];
pthread_mutex_t lock;
int pointer = -1;
uint id = -1;
```

```
// Pointer value won't always stay unique because of deletion of users. Hence for a uniquie ID we
// use a global variable

uint getID()
{
    return ++id;
}

struct User createUser(struct User user)
{
    if (pointer == MAX_USERS - 1)
    {
        printf("Max limit reached, cannot create new user\n");
        user.id = -1;
        return user;
    }
    pthread_mutex_lock(&lock);
    user.id = getID();
    users[++pointer] = user;
    pthread_mutex_unlock(&lock);
    return user;
}

//Only user.id is relevant for delete and read,
//the remaining fields of user do not need to be initialised for these operations

struct User deleteUser(struct User user)
{
    if (user.id > pointer || user.id < 0)
    {
        printf("Entry does not exist\n");
        user.id = -1;
        return user;
    }
    pthread_mutex_lock(&lock);
    struct User ret = users[user.id];
    for (int i = user.id; i < pointer; i++)
    {
        users[i] = users[i + 1];
    }
    pointer--;
    pthread_mutex_unlock(&lock);
    return ret;
}

//Mutex lock is not needed for read operation because db is not being updated

struct User readUser(struct User user)
{
    if (user.id > pointer || user.id < 0)
    {
```

```
        printf("Entry does not exist\n");
        user.id = -1;
        return user;
    }
    for (int i = 0; i <= pointer; i++)
    {
        if (users[i].id == user.id)
        {
            return users[i];
        }
    }
    printf("User not found\n");
    user.id = -1;
    return user;
}

// Find user by ID and update all fields

struct User updateUser(struct User user)
{
    if (user.id > pointer || user.id < 0)
    {
        printf("Entry does not exist\n");
        user.id = -1;
        return user;
    }
    uint i = user.id;
    strcpy(users[i].name, user.name);
    strcpy(users[i].email, user.email);
    strcpy(users[i].password, user.password);
    return users[i];
}

void *serveThread(void *args)
{
    struct data received_data;
    int new_socket = (*(int *)args);
    read(new_socket, (struct data *)&received_data, sizeof(struct data));
    struct data send_data;
    if (received_data.create)
    {
        // Create new User
        send_data.user = createUser(received_data.user);
    }
    else if (received_data.read)
    {
        send_data.user = readUser(received_data.user);
    }
    else if (received_data.update)
    {
        send_data.user = updateUser(received_data.user);
    }
```

```
    else if (received_data.delete)
    {
        send_data.user = deleteUser(received_data.user);
    }
    write(new_socket, (struct data *)&send_data, sizeof(send_data));
    close(new_socket);
}

void serve(int new_socket)
{
    pthread_t newThread;
    pthread_create(&newThread, NULL, &serveThread, (void *)&new_socket);
}

int main(int argc, char const *argv[])
{
    int server_fd, new_socket;
    struct sockaddr_in address;
    int addrlen = sizeof(address);
    pthread_mutex_init(&lock, NULL);
    // Creating socket file descriptor
    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0)
    {
        perror("In socket");
        exit(EXIT_FAILURE);
    }
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT);
    memset(address.sin_zero, '\0', sizeof address.sin_zero);
    if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) < 0)
    {
        perror("In bind");
        exit(EXIT_FAILURE);
    }
    if (listen(server_fd, 10) < 0)
    {
        perror("In listen");
        exit(EXIT_FAILURE);
    }
    while (1)
    {
        if ((new_socket = accept(server_fd, (struct sockaddr *)&address, (socklen_t *)&addrlen)) < 0)
        {
            perror("In accept");
            exit(EXIT_FAILURE);
        }
        serve(new_socket);
    }
    close(server_fd);
    pthread_mutex_destroy(&lock);
    return 0;
```

```
}
```

**types.h** (structs used)

```
#include <stdlib.h>

struct User
{
   uint id;
   char name[30];
   char email[30];
   char password[30];
};

struct data
{
   int create;
   int read;
   int update;
   int delete;
   struct User user;
};
```

**Explanation:**
We have implemented a non persistent, in memory database. It stores user data in an array and supports create, read, update, and delete operations. The server is a concurrent server, meaning it can process multiple requests at a time. Concurrency has been achieved using thread creation instead of fork().
Since read operation does not make any changes to the code, it doesn't make use of mutex lock. Create, update and delete operations use mutex lock, to prevent race conditions during execution of their critical section.

Note: Do refer dummy code from the repository. Running it many times with and without mutex lock demonstrates how mutex lock helps in thread synchronization.

## Contribution From Each Student

Tejas Hegde – Concept and idea
Swapnodip – Client side code
Devyani – Server side code to accept requests
Akshita – Database operations code

## Learning Outcome

How to create a thread, and why threads can be better than processes, in terms of performance.

How to achieve thread synchronization through the use of mutex locks.

# References

[1] https://www.baeldung.com/linux/process-vs-thread

[2] https://www.cplusplus.com/reference/mutex/mutex/lock/