May 19, 2021

Gebze Technical University
Computer Engineering
CSE344 - Spring 2021
Final Project

Nevzat Seferoglu
171044024

1 Problem Definition

In this project, there is server which can handle multiple clients concurrently. There are clients which are requested for some sort of data from server. Server needed to work in synchronize for answering client queries. There can be lots of clients which request for data. The data needed to be kept in table. Because, int the project, general use-case of that model was representing as a **sql-server**. Data can be updated or selected more than one times from unlimited clients. Clients are not restricted. Server was needed to be handled in an elegant way.

2 Solution

Project contains some tricky implementation like unbounded producer-consumer and reader-writer. Synchronization part was also needed to be handled in a good way. Therefore, **conditional variables and mutexes** were my best gun for that issue. Server rules lots of clients at the same time (almost). Server's send its rules to clients with conditional variable. Server also receives some state information from clients (in a sense of act), according to states behaves differently. Especially escaping mechanism were crucial. Interrupts or signals can be received in any line of code in server also in clients. When something go wrongs server was needed to handle that problem elegantly. **atexit** function is quite uselful for that kind of situations.
I detailed my solution in different parts and explain each of them one by one.

2.1 Design and Solutions

2.1.1 Representing CSV in C

There were several option to kept csv's data in C. I could use **HashMap**, **LinkedList**, **ArrayList** or even some kind of graph structure. Problem was huge and we are not adding new records to our database. Therefore I choosed **string **table** structure. First pointer represents row, second one represent columns of csv file. Each cell is stand for data from left to right and top to buttom in order.
- My cells can contain any kind of characters except double-quate.
- I represents blank records as null.
- My token which are surrounded with double-quate can contain any characters.
For making possible those kind of problem, I designed my own **split** function instead of using strtok. I can parse the string with or without and any character which may surround.

```
char** split(char *str, char c, char out, int *size);
char** split2(char *str, char c, char opt, int *size);
```

These function return newly allocated two dimentional array. Each element of an array represents tokens. I also send the size as pointer and receive the size of token after returning back to caller. Therefore I can traverse on tokens.
I also neded to implement **free** functions that can free the allocated tokens array.

```
/* release tokens */
void freepath(char **path, int size) {
    int i = 0;
    if (path != NULL) {
        for(i = 0; i<size; ++i) {
            free(*(path+i));
            *(path+i) = NULL;
        }
        free(path);
        path = NULL;
    }
}

/* release database table */
void release_table() {
    if (table != NULL && release_check) {
        for (int i=0; i<rSize; ++i) {
            if (table[i] != NULL) {
                for (int j=0; j<cSize; ++j) {
                    free(table[i][j]);
                }
            }
        }
        for (int i=0; i<rSize; ++i) {
           free(table[i]);
        }
```

```
        free(table);
    }
}
```

As a result, I achieved to convert csv files to **C type of array structure**. There is no missing instruction for that part.

2.1.2 Server Design

Main server component

```
typedef struct ThreadArgument {
    int i;
} ThreadArgument;

typedef struct Thread {
    pthread_t threadID;
    ThreadArgument args;
} Thread;

Thread *threads = NULL;
Queue *queue = NULL;

// socket components
int sockfd;
struct sockaddr_in server_addr;
struct sockaddr_in client_addr;
socklen_t socklen;
int clientfd;
```

I kept a queue for incoming connection to server. Server initialize a pool of threads for handling incoming request. There was a **synchronization barrier** for archived that. After I ensures that all threads are created and ready for taking their jobs from queue.
My queue contains the file descriptor of each incoming connection to server. I enqueue that file descriptor to queue and wait for threads the dequeue and handle the request. My assigned thread finishes all jobs of connected client and return back to pool to get a new job.
Unbounded Producer-Consumer
We are concurrently accessing queue in that design. Each thread can enqueue and dequeu in any time. Even main thread can enqueue to our queue structure. Therefore, I save my design with cinditional variable and mutext.

```
/* My queue also kept size in it. I exploits of that variable to determine conditional varable. */
pthread_mutex_t mQueue;
pthread_cond_t cQueue = PTHREAD_COND_INITIALIZER;
```

Reader-Writer
Each thread needed parse and evaulate given query in our common database. Therefore every possible running thread can access database, read and write some value to it. It is obvious that we need to save our structure with reader-writer synchronization.

```
int AR = 0;
int AW = 0;
int WR = 0;
int WW = 0;
pthread_mutex_t mRW;
pthread_cond_t okToRead = PTHREAD_COND_INITIALIZER;
pthread_cond_t okToWrite = PTHREAD_COND_INITIALIZER;
```

Are threads are available in meanwhile ?
I also use another conditional variable and mutex combination to determine whether any thread available. If thread available en I cannot go for next acception, otherwise server need to sleep for waiting at least one thread is done and ready for next client connection.

```
// available threads
int available = 0;
pthread_mutex_t mAvailable;
pthread_cond_t cAvailable = PTHREAD_COND_INITIALIZER;
```

I am in need of keeping another global value which is **liveThreadNum**. It keeps the number of threads which are alive at that moment. I exploited that value to post finito semaphore in case of it is the last alive thread. If it is last, main thread will be aware of that. Because it is already waiting for the working thread to be finished. Then It can exit safely.

2.1.3 How communicate with client ?

There are two distinc messages for starting end ending of communication. My server (agent thread) send a message to client to say it is ready to handle its job. Client receive that message and read its next query from file. Send the query to alreadt connected thread.
After that point, appointed thread parse the query and evaluate it to inform client. After parsing, according the query type there is two different option to send it:
- If it is SELECT type, thread first send the records number to user client, then return entire answer to client **line by line**. Because, writing to sockets is not atomic until some vertain value.
- If it is UPDATE, thread returns updated column to client as a message.

```
void server_loop() {
    char *time = NULL;

    for (;;) {

        clientfd = x_accept(sockfd, (SA*)(&client_addr), &socklen, logFile);
        if (interrupt && clientfd == -1) {
            time = current_timestamp();
            fprintf(logFile, "%s Termination signal received, waiting for ongoing threads to complete.\n",
     time);
            free(time);
            x_pthread_cond_broadcast(&cQueue, logFile);
            break;
        }

        x_pthread_mutex_lock(&mAvailable, logFile);
        while (available <=0) {
            time = current_timestamp();
            fprintf(logFile, "%s No thread is available! Waiting \n", time);
            free(time);

            x_pthread_cond_wait(&cAvailable, &mAvailable, logFile);
            if (interrupt) {

                time = current_timestamp();
                fprintf(logFile, "%s Termination signal received, waiting for ongoing threads to complete
    .\n", time);
                free(time);
                x_pthread_cond_broadcast(&cAvailable, logFile);
                break;
            }
        }

        --available;
        x_pthread_mutex_unlock(&mAvailable, logFile);

        x_pthread_mutex_lock(&mQueue, logFile);
        enqueue(queue, clientfd);

        if (queue == NULL) {
            x_pthread_mutex_unlock(&mQueue, logFile);
            x_pthread_cond_broadcast(&cQueue, logFile);
            break;
        }

        x_pthread_cond_broadcast(&cQueue, logFile);
        x_pthread_mutex_unlock(&mQueue, logFile);
    }
}
```

After all query is done. Client send a ending message to server (thread) for closing already opened file descriptor and let thread leave for taking new client connection jobs. Taking new jobs contains some sub task. First thread needed signal itself as available. Then leave them to main thread to be assign a new job.

```
x_pthread_mutex_lock(&mQueue, logFile);
while (queue->size == 0) {
    x_pthread_cond_wait(&cQueue, &mQueue, logFile);
    if (interrupt) {
```

```
        x_pthread_mutex_unlock(&mQueue, logFile);
        x_pthread_cond_broadcast(&cQueue, logFile);
        break;
    }
}
if (interrupt) {
    break;
}
fd = dequeue(queue);
if (fd == -1) {
    x_pthread_mutex_unlock(&mQueue, logFile);
    break;
}
x_pthread_mutex_unlock(&mQueue, logFile);
```

2.1.4 Client Desing

Clients are less complicated than server. Each client takes its own query from file. Send them through the connection. Every query are being sent from the same connection.

From the client perspective,

- Receive the starting information to send its query.
- After recive start notification, send query to waiting thread to be eveluated.
- Receives an integer value, if the value equals to zero, invalid query has been given and nothing is done by the server.
- Receives an integer value if the value more than ones, it will represended as a row number and print some information about return state. It can be table or it can also be updated information.

```
void client_request() {
    char line[BUFSIZ];
    char actualQuery[BUFSIZ];
    int queryID, numOfQuery = 0;
    int numOfRecords;

    if (interrupt) {
        exit(EXIT_FAILURE);
    }

    memset(line, 0, BUFSIZ);
    x_read(sockfd, line, BUFSIZ, stderr);
    if (strcmp (line, START_NOTIFICATION) == 0) {
        fprintf(stdout, "%s Client-%d connecting to %s:%d\n", current_timestamp(), id, IPv4, PORT);
    } else {
        fprintf(stdout, "%s Client-%d cannot connect to %s:%d\n", current_timestamp(), id, IPv4, PORT);
    }

    fptr = x_fopen(pathToQueryFile, stderr);

    memset(line, 0, BUFSIZ);
    while (fgets(line, BUFSIZ, fptr) != NULL) {

        if (interrupt) {
            exit(EXIT_FAILURE);
        }

        memset(actualQuery, 0, BUFSIZ);
        sscanf (line, "%d %[^\n]", &queryID, actualQuery);

        if (id != queryID)
            continue;

        calcTime.start = clock();
        fprintf(stdout, "%s Client-%d connected and sending query \'%s\'\n", current_timestamp(), id,
    actualQuery);
        if (write(sockfd, actualQuery, BUFSIZ) == -1) {
            fprintf(stderr, "%s Error, %s: %s with [errno:%s]\n", __func__, "Query cannot be sent to
    server from client!\n", current_timestamp(), strerror(errno));
            exit(EXIT_FAILURE);
        }

        x_read(sockfd, &numOfRecords, sizeof(int), stderr);
```

```
        if (numOfRecords != 0) {
            calcTime.end = clock() - calcTime.start;
            calcTime.elapsedTime = ((double)calcTime.end)/CLOCKS_PER_SEC;
            fprintf(stdout, "%s Server's response to Client-%d is %d records, and arrived in %.5lf seconds.\n",
                current_timestamp(), id, (numOfRecords-1), (calcTime.elapsedTime));

            for (int i=0; i<numOfRecords; ++i) {
                memset(line, 0, BUFSIZ);
                x_read(sockfd, line, BUFSIZ, stderr);
                fprintf(stdout, "%s\n", line);
            }
        }

        memset(line, 0, BUFSIZ);
        ++numOfQuery;
    }

    if (write(sockfd, END_NOTIFICATION, strlen(END_NOTIFICATION)+1) == -1) {
        fprintf(stderr, "%s Error, %s: %s with [errno:%s]\n", __func__, "END_NOTIFICATION cannot be sent to server from client!\n", current_timestamp(), strerror(errno));
        exit(EXIT_FAILURE);
    }

    x_close(sockfd, stderr);

    fprintf(stdout, "%s A total of %d queries were executed, Client-%d is terminating.\n",
    current_timestamp(), numOfQuery, id);
}
```

After everything is done, client need to inform the thread for saying 'I am done'. Thread will recognize that state and go for next connection by altering its conditional variable.

2.1.5 CTRL+C Handling

Server Part

There is a general handler for the entire process. There are certain points which are in a scope op conditonal variable and their its waiting stage. I needed to consider these interrupts. I informed the worker threads with the help of broadcasting. After the threads receive signal then I check the global variable of interrupt handler. If it is distinct, return back to main as soon as possible to exiting from the server. My threads were joinable and main thread waits for them to be returned.

```
/* one of my escaping mechanism */
x_pthread_mutex_lock(&mQueue, logFile);
while (queue->size == 0) {
    x_pthread_cond_wait(&cQueue, &mQueue, logFile);
    if (interrupt) {
        x_pthread_mutex_unlock(&mQueue, logFile);
        x_pthread_cond_broadcast(&cQueue, logFile);
        break;
    }
}
```

Client Part

Client part is much more easier than server's. There is no concurrent access to client and I checked for certain point to be exited from client. I did not interrupt the existing connection until the queries are done and return back to client. After the query is done, I checked for an interrupt, if it is happened, try to exit gracefully.

```
while (fgets(line, BUFSIZ, fptr) != NULL) {

    if (interrupt) {
        exit(EXIT_FAILURE);
    }
    /* Queries are handled */
}
```

3 Which requirements I achieved ?

- **I archived all requirements denoted in assignment file**.
- Except SELECT DISTINCT. (Please do not send this query to server.)