CSE 344 SYSTEM PROGRAMMING FINAL PROJECT REPORT

OZAN GEÇKİN

1801042103

1)Overview

My project includes 2 programs. A client and a server . The server run as a single instance, open up a stream socket and wait for clients to connect and conduct SQL queries. The server runs in the background and prints a log file by logging the results of the operations performed. The client run multiple instance because multiple clients can make requests to the server at the same time. The client reads SQL queries from a file, sends them to the server and can print the received responses. After making all the queries, the clients log out. I used mutex, condition variables and sockets while doing my project. I implemented mutex and condition variables to ensure synchronization. I used the socket to establish communication between programs. These sockets allowed me to make TCP connections.

Server

The server program runs in the background. So it cannot be controlled from the terminal. While running, it takes some parameters from the command line;

```
./server -p PORT -o pathToLogFile –l poolSize –d datasetPath
```

PORT: this is the port number the server will use for incoming connections.

pathToLogFile: is the relative or absolute path of the log file to which the server daemon will write all of its output (normal output & errors).

poolSize: the number of threads in the pool (>= 2)

datasetPath: is the relative or absolute path of a csv file containing a single table, where the first row contains the column names

The main thread of the server reads the dataset and puts it in memory. I have created struct structures to keep it in memory so that it is easy to access queries. Pool threads all execute the same function. Each waits for the main thread to give them work in an endless loop. If there is nothing to do, they will be blocked. They can get a job again after their job is done. It only supports 2 commands select and update. The various steps taken by the servers' threads will be printed to the log file.

Client

The client's job is easy. The query file will contain an arbitrary number of queries for each client process.

```
./client –i id -a 127.0.0.1 -p PORT -o pathToQueryFile
```

-a: IPv4 address of the machine running the server

-p: port number at which the server waits for connections (>1000, the first 1000 are reserved to kernel processes)

-o: relative or absolute path of the file containing an arbitrary number of queries

-i: integer id of the client (>=1)

The query file will contain one query per line. The line will start with the client id, have a single space, and the SQL query. The client process will read the file, and for every query with its ID, it will send it to the server, and print the result that it will receive. Once all its queries are completed, it will terminate.

2)How you solved this problem ?

In my project, the major problem I had to solve was the synchronization issue. Because the threads are accessing and reading and writing the same places. I used mutex and condition variables for these sync issues. I was very careful when using them. I would like to explain my solution way more, If two thread will try to do operations in the same memory at the same time, segfault will occur. For example if you have 2 threads and one shared variable between those threads and one thread will try to write something and another thread will try to read something from this variable, this will cause serious memory problems so linux kernel will not allow as to corrupt memory data and will stop our program with reason of segmentation fault or corrupted memory. To avoid conditions I explained above db server is using two types of locks ; pthread_rwlock_t and pthread_mutex_t ,rwlock is used for database and mutex is used for any other thread operations which may not be thread safe. This lock is acquaired when selecting or updating something in database. If rdlock (read lock) is used for SELECT and rwlock (write lock) is used for UPDATE. When rwlock is locked for reading only (rdlock), any other thread can lock this variable for reading and return imediately, but anyone which wants to write should wait until all rdlocks become unlocked. So if only reading from database during SELECT we are not holding another threads for reading, but when updating something in database, updater thread will acquaire exclusive lock and every other thread should wait. The mutex object referenced by mutex shall be locked by calling pthread_mutex_lock(). If the mutex is already locked, the calling thread shall block until the mutex becomes available, this means calling thread should wait until mutex becomes unlocked. This operation shall return with the mutex object referenced by mutex in the locked state with the calling thread as its owner.

Another problem was to provide communication between programs, I used sockets to provide the connection here. As for the sockets this is just simple tcp socket implemetation, nothing special, just using socket(), listen(), accept(), bind(), read() and write() functions. Those function names are descriptions itself.

socket() function creates new file descriptor associated with socket
listen() function listens this socket file to receive incomming connections or data
accept() function accepts new incomming connections
write() sends data to socket endpoint and read() reads data from socket endpoint

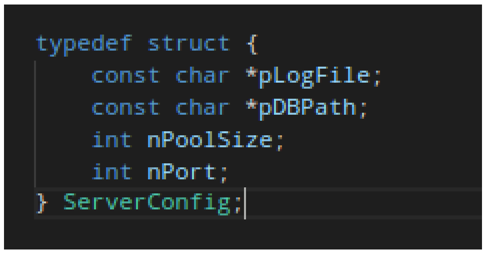
I created dynamic struct structures to hold and process the data. These helped me both initiate, destroy and implement. I worked on SQL and analyzed query jobs after parse operations.

3)Design Decisions and Algorithm:

After running the server application, at first global variable flags are initialized, we need this flags for termination to check which objects are allocated and destroy them properly. After signal handler is initialized to catch the interrupt signal for graceful termination. Then program is parsing command line arguments and initializes logger with path provided from command line argument. The next step is running in background, for that daemon() function is called which forks the process and and exits from main to detach process from terminal. If everything above is successful program creates listener socket and starts listening to it with listen() function. Then server initializes input file and loads it into RAM. For that server opens file for reading only and parsing it line by line with getline() function. Every line from the file is stored in the database structure variable RowData, which is dynamically allocated array to support all sizes of input file. Later this array is iterated to select matching recordings while executing SQL query. After loading the csv file program initializes mutex and condition variables for synchronization with threads and running threads as many as thread pool argument from command line. When threads are run server enters in the endless while loop where it is waiting and accepting incomming connections with accept() function. When new connection arrives program will iterate all the threads from thread pool to check which thread is available for the given moment. If there are no any thread available, program will block and waith with waiting on the pthread condition variable. If any thread gets available, it will signal to this variable so main will wake from waiting and assign the client to the available thread. Then main is signaling the selected threads condition variable to notify the thread about new connection. When worker threads are run they also enter in the endless while loop, where they are waiting for new connection by waiting on the pthread condition variable. When new connection arrives in the thread, thread will receive sql query from client socket, parse it and try to execute. For that worker thread locks the database with mutex variable and iterating database rows and columns to select the requested data. If column/row is matching to the query criteria the data of that row will be added in the response buffer. After finishing operations in the buffer the number of recordings is sent to client to notify client how many recordings are found for its query. Then the queries are sent to the client from response buffer. After finishing the query execution server is sleeping for 0.5 seconds to simulate intensive data execution as it was requested in the job description and then thread is starting again the while loop.

The struct structures I use on the server

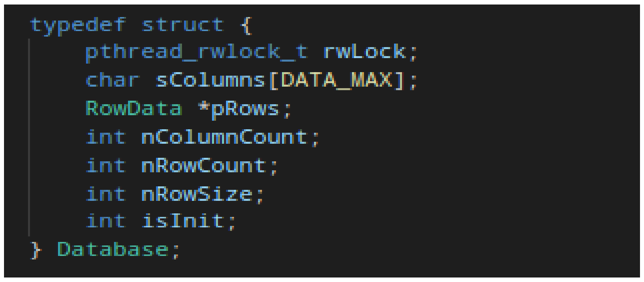
*** ServerConfig**



```
typedef struct {
    const char *pLogFile;
    const char *pDBPath;
    int nPoolSize;
    int nPort;
} ServerConfig;
```

To get command line arguments and use them.

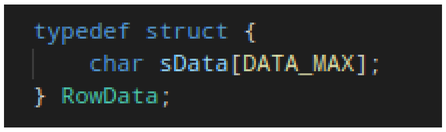
*** Database**



```
typedef struct {
    pthread_rwlock_t rwLock;
    char sColumns[DATA_MAX];
    RowData *pRows;
    int nColumnCount;
    int nRowCount;
    int nRowSize;
    int isInit;
} Database;
```

I used it for database structure to hold information that is in a database table. Here I used the "pthread_rwlock_t rwLock" lock while reading or writing from the database. I would like to explain why I am using rwlock instead of mutex. Mutex can have only one reader or writer a time but rwLock can have one writer or multiple reader at a time. So I used rwLock to make it safe to perform a lot of reads. I use isInit when making a flag allocated and destroy. It is necessary for me to be able to hold the other fields data and to make SQL queries.

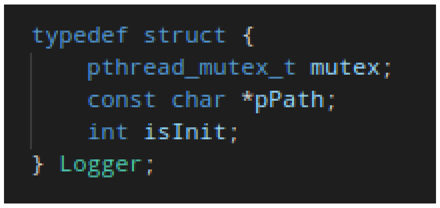
***RowData**



```
typedef struct {
    char sData[DATA_MAX];
} RowData;
```

This struct helps me when parsing to keep the data in a row in my database.

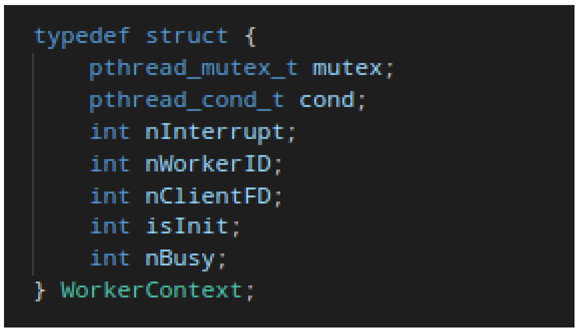
***Logger**



```
typedef struct {
    pthread_mutex_t mutex;
    const char *pPath;
    int isInit;
} Logger;
```

I use this structure in my log file preparation. I used Mutex to sync from the log file.

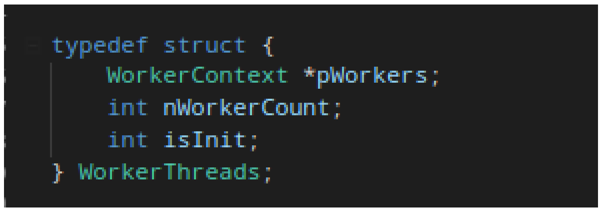
***WorkerContext**



```
typedef struct {
    pthread_mutex_t mutex;
    pthread_cond_t cond;
    int nInterrupt;
    int nWorkerID;
    int nClientFD;
    int isInit;
    int nBusy;
} WorkerContext;
```

I keep the fields I need inside the worker threads. I keep one mutex and one condition variable to ensure synchronization. I keep a flag to check if there is an interrupt. And I have client id and worked id. If it's nBusy, I'm checking whether worketThreading is working now.

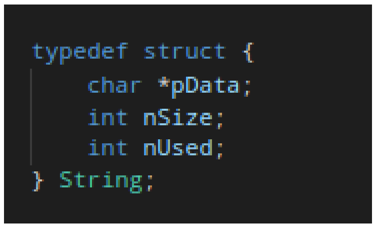
***WorkerThreads**



```
typedef struct {
    WorkerContext *pWorkers;
    int nWorkerCount;
    int isInit;
} WorkerThreads;
```

It contains the information I need for the Worker thread.

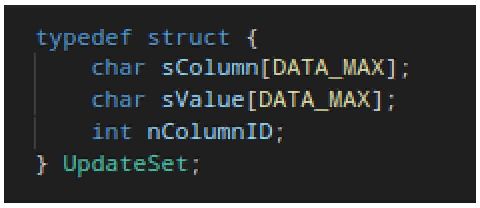
***String**



```
typedef struct {
    char *pData;
    int nSize;
    int nUsed;
} String;
```

I keep string information so that I can manipulate strings.

***UpdateSet**

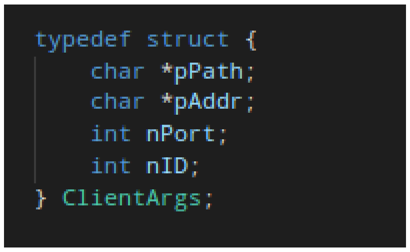


```
typedef struct {
    char sColumn[DATA_MAX];
    char sValue[DATA_MAX];
    int nColumnID;
} UpdateSet;
```

I use it to keep the update information when the database SQL query is updated.

The struct structures I use on the client

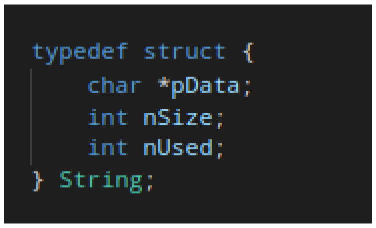
***ClientArgs**



```
typedef struct {
    char *pPath;
    char *pAddr;
    int nPort;
    int nID;
} ClientArgs;
```

To get command line arguments and use them.

***String**



```
typedef struct {
    char *pData;
    int nSize;
    int nUsed;
} String;
```

I keep string information so that I can manipulate strings

The functions I use are on the server

DYNAMIC STRINGS:

void removeCharacter(char *pDst, int nSize, const char *pStr, char nChar) : This function removes specified character from back, we need this function to remove new line characters (\n) from the line while parsing input csv file.

void stringInit(String *pStr, size_t nSize) : This function initializes dynamically allocated string. We need this string to assemble responses with various size.

void stringClear(String *pStr) : This function cleans string and frees its allocated memory.

size_t stringAppend(String *pStr, char *pData, size_t nSize) : This function reallocates string size and appends new data to the string.

SIMPLE UTILS:

uint32_t timeStamp() : This function returns timestamps in uses.

void logToFile(int nType, char *pStr, ...) : This function receives various arguments, opens file and writes input into file.

EXIT RELATED STUFF:

void globalDestroy() : This function destroys absolutely all allocated memory by server.

void exitFailure(const char *pMessage) : This function prints error message, cleanups allocated memory and exits with failure.

void exitHandler(int sig) : SIGINT signal callback.

CONDITION VARIABLES:

void waitCondition(pthread_cond_t *pCond, pthread_mutex_t *pMutex) : This function just calls pthread_cond_wait() and exits if call is not successful, nothing more.

void signalCondition(pthread_cond_t *pCond) : This function just calls pthread_cond_signal() and exits if call is not successful, nothing more.

MUTEX:

void initMutex(pthread_mutex_t *pMutex) : This function mutex attribute is set and mutex is initialized.

void lockMutex(pthread_mutex_t *pMutex) : This function just calls pthread_mutex_lock() and exits if call is not successful, nothing more.

void unlockMutex(pthread_mutex_t *pMutex) : This function just calls pthread_mutex_unlock() and exits if call is not successful, nothing more.

void lockWrite(pthread_rwlock_t *pLock) : This function just calls pthread_rwlock_wrlock() and exits if call is not successful, nothing more.

void lockRead(pthread_rwlock_t *pLock) : This function just calls pthread_rwlock_rdlock() and exits if call is not successful, nothing more.

void unlockRW(pthread_rwlock_t *pLock): This function just calls pthread_rwlock_unlock() and exits if call is not successful, nothing more.

SOCKETS:

int createServerSocket(int nPort) : This function creates server socket, binds the port and listens to the newely created socket.

DATABASE:

void initDatabase(Database *pDB): This function initializes database structure.

void destroyDatabase(Database *pDB): This function destroys database structure and all associated variables.

void appendDatabase(Database *pDB, const char *pRowData): This function appends the row data into the database.

void loadDatabase(const char *pPath, Database *pDB): This function opens database file, line-by-line reads it and saves recordings in the Database structure.

int selectColumnID(Database *pDB, char *pColumnName, int *pIDS, int nCount): This function searches column id with column name from database.

int selectWithID(const char *pFrom, int nID, int nFound, String *pResponse): This function selects recordings from database with column id and appends those recordings in the pResponse variable.

int selectFromIDS(Database *pDB, int *pIDS, int nCount, String *pResponse): This function selects recordings from database with column id array and appends those recordings in the pResponse variable.

int executeSelectQuery(Database *pDB, char *pQuery, String *pResponse): This function parses SQL queries and executing them according the query type.

int parseEquality(Database *pDB, UpdateSet *pSet, char *pData): This function parses equality from the UPDATE sql query and saves parsed values into UpdateSet variable.

int updateDatabase(Database *pDB, UpdateSet *pSet, int nCount, UpdateSet *pCond): This function updates database recordings according to UpdateSet and UpdateSet condition.

int executeUpdateQuery(Database *pDB, char *pQuery, String *pResponse): This function executes UPDATE query and appends response in the string.

void initWorker(WorkerContext *pCtx, int nID): This function initializes worker thread associated variables.

void destroyWorker(WorkerContext *pCtx): This function destrois worker thread associated variables.

void* werkerThread(void *pArg): This is the worker thread function.

MAIN STUFF

void parseArgs(int argc, char *argv[], ServerConfig *pConf): This function parses command line arguments.

The functions I use are on the client

void stringInit(String *pStr, size_t nSize) : This function initializes dynamically allocated string. We need this string to assemble responses with various size.

void stringClear(String *pStr) : This function cleans string and frees its allocated memory.

size_t stringAppend(String *pStr, char *pData, size_t nSize) : This function reallocates string size and appends new data to the string.

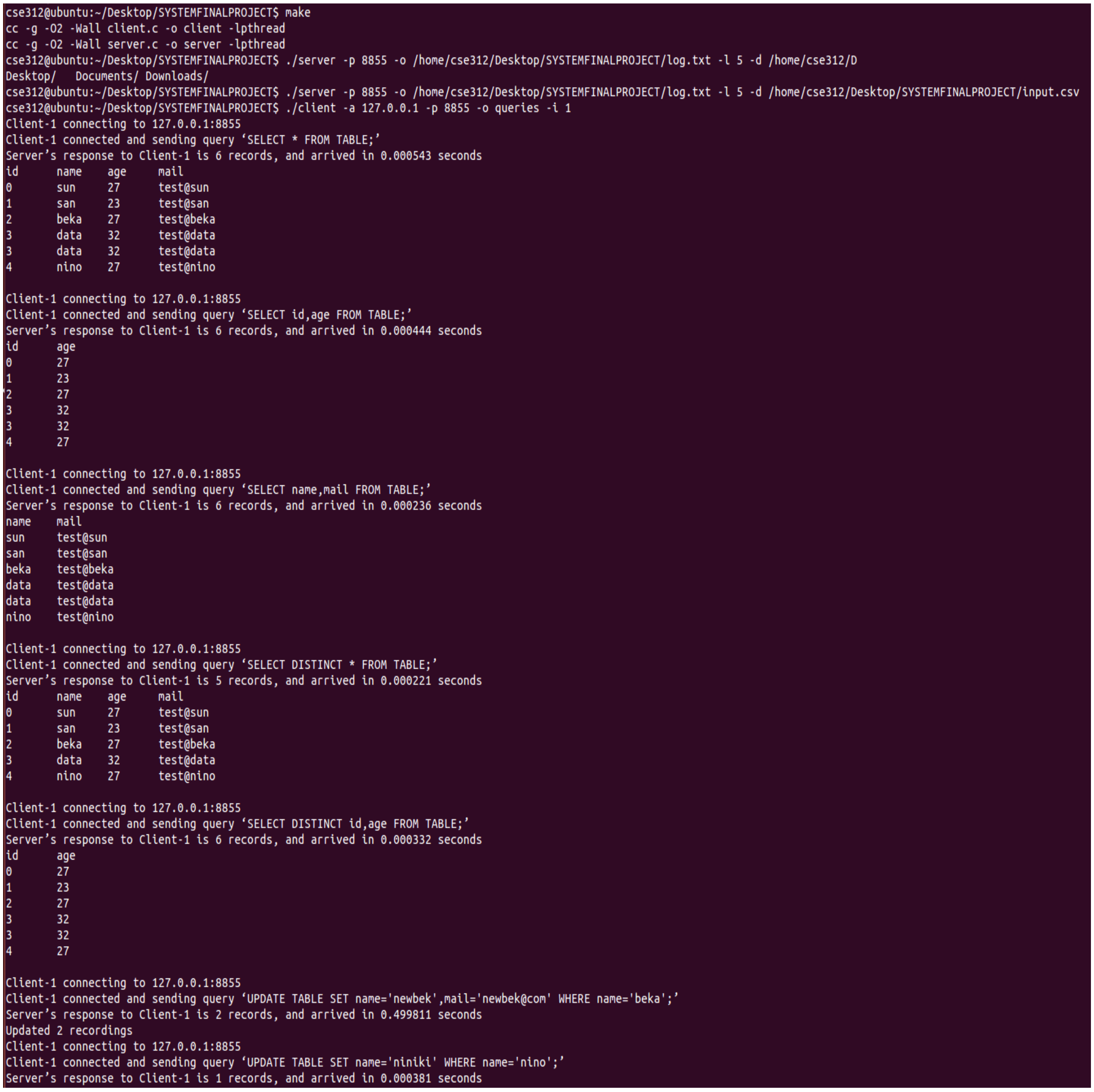
uint32_t timeStamp() : This function returns timestamps in uses.

int createClientSocket(const char *pAddr, uint16_t nPort): This fucntion creates client socket and connects to newely created socket.

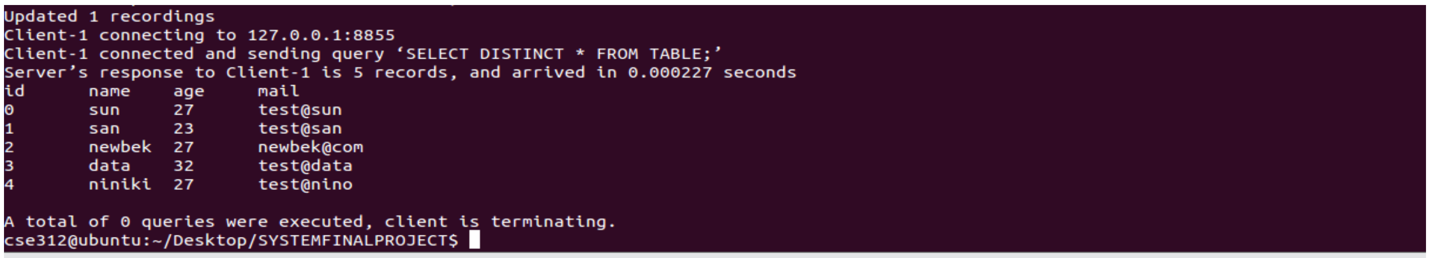
void parseArgs(int argc, char *argv[], ClientArgs *pConf): This function parses command line arguments.

int sendQueries(ClientArgs *pArgs): This function line by line reads sql queries from input file and sends to server, receives responses and prints them in the terminal.

4)Sample Running ScreenShots:

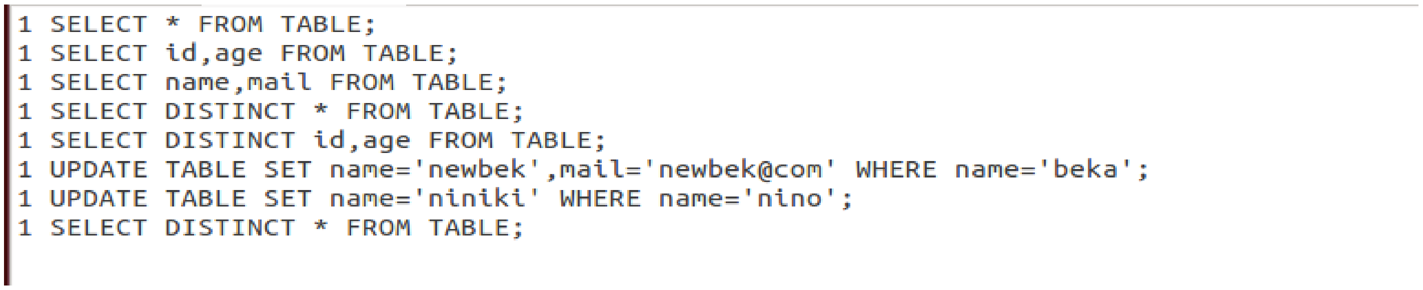


```
cse312@ubuntu:~/Desktop/SYSTEMFINALPROJECT$ make
cc -g -O2 -Wall client.c -o client -lpthread
cc -g -O2 -Wall server.c -o server -lpthread
cse312@ubuntu:~/Desktop/SYSTEMFINALPROJECT$ ./server -p 8855 -o /home/cse312/Desktop/SYSTEMFINALPROJECT/log.txt -l 5 -d /home/cse312/D
Desktop/   Documents/ Downloads/
cse312@ubuntu:~/Desktop/SYSTEMFINALPROJECT$ ./server -p 8855 -o /home/cse312/Desktop/SYSTEMFINALPROJECT/log.txt -l 5 -d /home/cse312/Desktop/SYSTEMFINALPROJECT/input.csv
cse312@ubuntu:~/Desktop/SYSTEMFINALPROJECT$ ./client -a 127.0.0.1 -p 8855 -o queries -i 1
Client-1 connecting to 127.0.0.1:8855
Client-1 connected and sending query ‘SELECT * FROM TABLE;’
Server’s response to Client-1 is 6 records, and arrived in 0.000543 seconds
id      name    age     mail
0       sun     27      test@sun
1       san     23      test@san
2       beka    27      test@beka
3       data    32      test@data
3       data    32      test@data
4       nino    27      test@nino

Client-1 connecting to 127.0.0.1:8855
Client-1 connected and sending query ‘SELECT id,age FROM TABLE;’
Server’s response to Client-1 is 6 records, and arrived in 0.000444 seconds
id      age
0       27
1       23
2       27
3       32
3       32
4       27

Client-1 connecting to 127.0.0.1:8855
Client-1 connected and sending query ‘SELECT name,mail FROM TABLE;’
Server’s response to Client-1 is 6 records, and arrived in 0.000236 seconds
name    mail
sun     test@sun
san     test@san
beka    test@beka
data    test@data
data    test@data
nino    test@nino

Client-1 connecting to 127.0.0.1:8855
Client-1 connected and sending query ‘SELECT DISTINCT * FROM TABLE;’
Server’s response to Client-1 is 5 records, and arrived in 0.000221 seconds
id      name    age     mail
0       sun     27      test@sun
1       san     23      test@san
2       beka    27      test@beka
3       data    32      test@data
4       nino    27      test@nino

Client-1 connecting to 127.0.0.1:8855
Client-1 connected and sending query ‘SELECT DISTINCT id,age FROM TABLE;’
Server’s response to Client-1 is 6 records, and arrived in 0.000332 seconds
id      age
0       27
1       23
2       27
3       32
3       32
4       27

Client-1 connecting to 127.0.0.1:8855
Client-1 connected and sending query ‘UPDATE TABLE SET name='newbek',mail='newbek@com' WHERE name='beka';’
Server’s response to Client-1 is 2 records, and arrived in 0.499811 seconds
Updated 2 recordings
Client-1 connecting to 127.0.0.1:8855
Client-1 connected and sending query ‘UPDATE TABLE SET name='niniki' WHERE name='nino';’
Server’s response to Client-1 is 1 records, and arrived in 0.000381 seconds
```



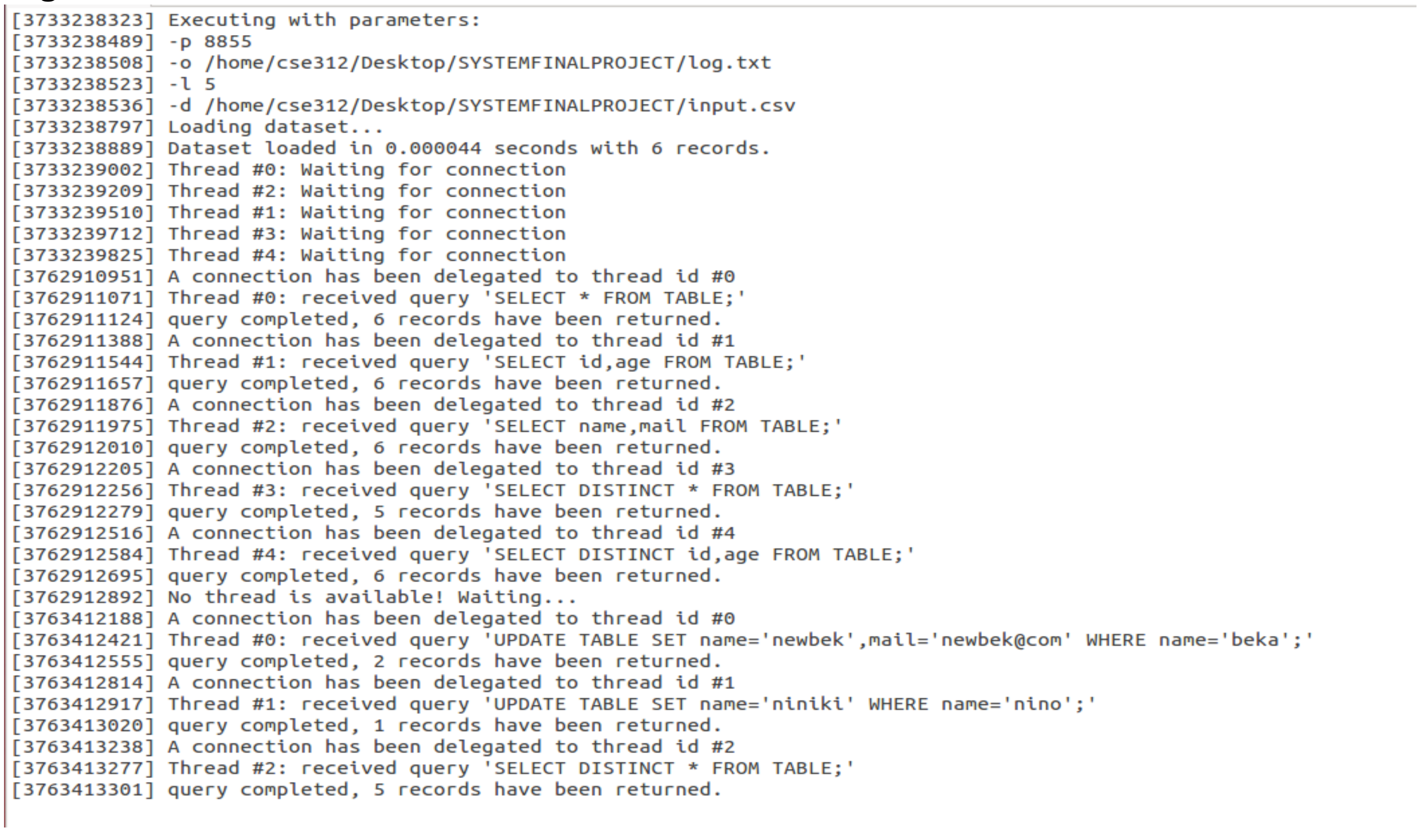
```
Updated 1 recordings
Client-1 connecting to 127.0.0.1:8855
Client-1 connected and sending query ‘SELECT DISTINCT * FROM TABLE;’
Server’s response to Client-1 is 5 records, and arrived in 0.000227 seconds
id      name    age     mail
0       sun     27      test@sun
1       san     23      test@san
2       newbek  27      newbek@com
3       data    32      test@data
4       niniki  27      test@nino

A total of 0 queries were executed, client is terminating.
cse312@ubuntu:~/Desktop/SYSTEMFINALPROJECT$
```

Queries:



```
SELECT * FROM TABLE;
SELECT id,age FROM TABLE;
SELECT name,mail FROM TABLE;
SELECT DISTINCT * FROM TABLE;
SELECT DISTINCT id,age FROM TABLE;
UPDATE TABLE SET name='newbek',mail='newbek@com' WHERE name='beka';
UPDATE TABLE SET name='niniki' WHERE name='nino';
SELECT DISTINCT * FROM TABLE;
```

Log.txt:



```
[3733238323] Executing with parameters:
[3733238489] -p 8855
[3733238508] -o /home/cse312/Desktop/SYSTEMFINALPROJECT/log.txt
[3733238523] -l 5
[3733238536] -d /home/cse312/Desktop/SYSTEMFINALPROJECT/input.csv
[3733238797] Loading dataset...
[3733238889] Dataset loaded in 0.000044 seconds with 6 records.
[3733239002] Thread #0: Waiting for connection
[3733239209] Thread #2: Waiting for connection
[3733239510] Thread #1: Waiting for connection
[3733239712] Thread #3: Waiting for connection
[3733239825] Thread #4: Waiting for connection
[3762910951] A connection has been delegated to thread id #0
[3762911071] Thread #0: received query 'SELECT * FROM TABLE;'
[3762911124] query completed, 6 records have been returned.
[3762911388] A connection has been delegated to thread id #1
[3762911544] Thread #1: received query 'SELECT id,age FROM TABLE;'
[3762911657] query completed, 6 records have been returned.
[3762911876] A connection has been delegated to thread id #2
[3762911975] Thread #2: received query 'SELECT name,mail FROM TABLE;'
[3762912010] query completed, 6 records have been returned.
[3762912205] A connection has been delegated to thread id #3
[3762912256] Thread #3: received query 'SELECT DISTINCT * FROM TABLE;'
[3762912279] query completed, 5 records have been returned.
[3762912516] A connection has been delegated to thread id #4
[3762912584] Thread #4: received query 'SELECT DISTINCT id,age FROM TABLE;'
[3762912695] query completed, 6 records have been returned.
[3762912892] No thread is available! Waiting...
[3763412188] A connection has been delegated to thread id #0
[3763412421] Thread #0: received query 'UPDATE TABLE SET name='newbek',mail='newbek@com' WHERE name='beka';'
[3763412555] query completed, 2 records have been returned.
[3763412814] A connection has been delegated to thread id #1
[3763412917] Thread #1: received query 'UPDATE TABLE SET name='niniki' WHERE name='nino';'
[3763413020] query completed, 1 records have been returned.
[3763413238] A connection has been delegated to thread id #2
[3763413277] Thread #2: received query 'SELECT DISTINCT * FROM TABLE;'
[3763413301] query completed, 5 records have been returned.
```