June 8, 2021

GTU Department of Computer Engineering
CSE344 - Spring 2021
Final Report

Akif Kartal
171044098

1 Problem Definition

The problem is to implement **a rudimentary database** with one server and multiple client using socket programming. Database operations will be handled by a thread pool.

2 Solution

The final project was finished as expected in homework pdf file.

2.1 Server Side

Server has some problems to be solved. Some of them are following;

2.1.1 Becoming a daemon

In order to make server daemon following code applied;

```
    switch (fork())
    {
        case -1:
            errExit("fork error!",1);
        case 0:
            break;
        default:
            exit(EXIT_SUCCESS);
    }

    if (setsid() == -1)
        errExit("setsid error!",1);

  //ignore some signals
    signal(SIGCHLD, SIG_IGN);
    signal(SIGHUP, SIG_IGN);

    switch (fork())
    {
        case -1:
            errExit( "fork error!",1);
        case 0:
            break;
        default:
            exit(EXIT_SUCCESS);
    }
    umask(0);
    //close all file descriptors
    for (int i = sysconf(_SC_OPEN_MAX); i >= 0; i--){
        if (i != givenParams.logFd && i != givenParams.datasetFd)
            close(i);
    }
```

2.1.2 Preventing Double Instantiation of Server

In order to solve this problem, I applied many solution but one of must be chosen. Followings are some possible solutions;

- Create a named semaphore = This leads memory leaks
- Use file locks = Not reliable
- Use .pid files = Clean way

I have used .pid file to solve this. When program starts, I am creating a unique name .pid file with my university id number by using **O_EXCL** flag in open function, then in second run if file already exist then program will not run. After server finishes, I am removing that file. My solution is following;

```
void openControlFile(){
    if(open(TEMPFILE, O_RDWR|O_CREAT|O_EXCL, 0666) == -1) {
        if (errno == EEXIST){
            printf("Only one instantiation can be created!\n");
            exit(EXIT_FAILURE);
        }
    }
}
void removeControlFile(){
    remove(TEMPFILE);
}
```

2.1.3 Loading the dataset into memory

This problem is the one of the most difficult problem. In order to make database operations we need to choose a good data structure. Some options for data structure are following;

- char ***data = Simple one
- Linked List = Difficult to implement but reliable

I have chose **Linked List** data structure to make things easier. Following is my Linked List node structure;

```
//column information node
 typedef struct node_s
 {
    int size; // size of full data(number of row)
    int capacity;
    char *columnName;
    char **data; // full data in a single column
    struct node_s *next;

 } node_t;
```

By using this structure finding a column is very easy, Also size and capacity information let us to control allocating memory.

Reading File

In order to read .csv file I used **read()** function. I read file one byte one and I checked newline character, in this way I can easily detect if there is an empty column. After reading a line I parse that line by using ,(comma) character and I filled the linked list. Check readFile function in sql_engine.c file.

2.1.4 Creating Thread Pool

My solution is following;

```
void createPool(){
    int n = givenParams.poolSize;
    pthread_mutex_lock(&runMutex);
    for (int i = 1; i <= n; ++i) {
        int *index = (int*) calloc(1,sizeof(int));
        *index = i;
        pthread_create(&tids[i-1], NULL, sqlEngine, index);
    }
    dprintf(givenParams.logFd, "[%s] A pool of %d threads has been created\n",
            getTime(), n);
    pthread_mutex_unlock(&runMutex);
}
```

In this solution, I only passed the index of thread as thread data because we don't need to any other thing.

Now, everything is ready we can delegate the queries to the threads.

2.1.5 Delegating a connection

When the new connection arrives, server main thread needs to delegate it to an available thread. In order to solve this problem, I have used **monitor(condition variable with mutex)**. My solution is following;

Main Thread

```
if ((newFd = accept(socketfd, (struct sockaddr *)&newAddr, &addr_size)) == -1)
  errExit("accept error",1);

pthread_mutex_lock(&busyMutex);
while (activeWorkers == givenParams.poolSize) { // if everyone is busy, wait
  dprintf(givenParams.logFd,"[%s] No thread is available! Waiting...\n",getTime());
  pthread_cond_wait(&okToDelegate,&busyMutex);
}
pthread_mutex_unlock(&busyMutex);
pthread_mutex_lock(&taskMutex);
addRear(queryQueue,newFd); //add new query to the queue
pthread_mutex_unlock(&taskMutex);

pthread_cond_signal(&okToExecute); //signal any avaliable thread
```

Worker Thread

```
pthread_mutex_lock(&taskMutex);
while (isQueueEmpty(queryQueue)) { // if no query just wait
  pthread_cond_wait(&okToExecute,&taskMutex);
}

pthread_mutex_lock(&busyMutex);
activeWorkers++;
pthread_mutex_unlock(&busyMutex);

dprintf(givenParams.logFd, "[%s] A connection has been delegated to thread id #%d\n", getTime(), *i);
int currentFd = removeFront(queryQueue);
pthread_mutex_unlock(&taskMutex);

//read and parse query and return result...

pthread_mutex_lock(&busyMutex);
activeWorkers--;
pthread_mutex_unlock(&busyMutex);
pthread_cond_signal(&okToDelegate); //signal to main thread
```

2.1.6 Readers-Writers Paradigm

My solution is following to this problem;

Worker Thread

```
while(safeRead(currentFd,queries[(*i)-1],MAX_READ,1)!=0){ //read query from client
  dprintf(givenParams.logFd, "[%s] Thread #%d: received query '%s'\n",
      getTime(), *i,queries[(*i)-1]);
  if(getQueryType(*i) == read){
    reader(*i,currentFd);
  }
  else if(getQueryType(*i) == write){
    writer(*i,currentFd);
  }
  //sleep for 0.5 seconds
  milSleep(500);
  free(queries[(*i)-1]);
  queries[(*i)-1] = (char*) calloc(MAX_READ+1,sizeof(char));
}
```

Reader function

```
void reader(int index,int fd){
    pthread_mutex_lock(&m);
    while ((AW + WW) > 0) { // if writers, wait
        WR++;
        pthread_cond_wait(&okToRead,&m);
        WR--;
    }
    AR++;
    pthread_mutex_unlock(&m);
    accessDB(index,fd);
    pthread_mutex_lock(&m);
```

```
    AR--;
    if (AR == 0 && WW > 0)
        pthread_cond_signal(&okToWrite);
    pthread_mutex_unlock(&m);
}
```

Writer function

```
void writer(int index,int fd){
    pthread_mutex_lock(&m);
    while ((AW + AR) > 0) {
        WW++;
        pthread_cond_wait(&okToWrite,&m);
        WW--;
    }
    AW++;
    pthread_mutex_unlock(&m);
    updateDB(index,fd);
    pthread_mutex_lock(&m);
    AW--;
    if (WW > 0) // give priority to other writers
        pthread_cond_signal(&okToWrite);
    else if (WR > 0)
        pthread_cond_broadcast(&okToRead);
    pthread_mutex_unlock(&m);
}
```

Access DB(Read)

```
void accessDB(int index,int fd){
    char *result = NULL;
    result = mySelect(queries[index-1]);
    dprintf(givenParams.logFd, "[%s] Thread #%d: query completed, %d records have been returned.\n",
            getTime(), index, getReturnSize(result));
    safeWrite(fd,result, strlen(result) + 1,1); //write result to the client
    free(result);
}
```

Update DB(Write)

```
void updateDB(int index,int fd){
    int affected = update(queries[index-1]);
    int len = 0;
    if (affected > 0)
        len = (int)((ceil(log10(affected))+1)*sizeof(char));
    else
        len = 2;
    dprintf(givenParams.logFd, "[%s] Thread #%d: query completed, %d records have been affected.\n",
            getTime(), index, affected);
    char result[len + 5];
    sprintf(result, "%d", affected);
    safeWrite(fd,result,strlen (result)+1,1); //write result to the client
}
```

2.1.7 Exit Gracefully

In order to terminate on **SIGINT** interrupt, I used **sigaction** function from **signal.h** library. But in order to do that we need to wait each thread return and then give resources back. See my solution;

Handler and Usage

```
volatile __sig_atomic_t exitSignal = 0;
void connectSignalHandler(){
    struct sigaction sa;
    memset(&sa, 0, sizeof(sa));
    sa.sa_handler = &exitHandler;
    sigaction(SIGINT, &sa, NULL);
}
//handler function
void exitHandler(int signal){
    if (signal == SIGINT){
        exitSignal = 1;
        dprintf(givenParams.logFd, "[%s] Termination signal received, waiting for ongoing threads to
    complete.\n", getTime());
        addRear(queryQueue,1);
        //send signal all waiting threads
        pthread_cond_broadcast(&okToDelegate);
        pthread_cond_broadcast(&okToExecute);
        //join all threads
        for (int i = 0; i < givenParams.poolSize; i++)
        {
            if (!pthread_equal(pthread_self(), tids[i]))
                pthread_join(tids[i], NULL);
        }
        //give resources back
        free(tids);
        for (int i = 0; i < givenParams.poolSize; ++i) {
            free(queries[i]);
        }
        free(queries);
        freeList(head);
        freeQueue(queryQueue);
        removeControlFile();
        dprintf(givenParams.logFd, "[%s] All threads have terminated, server shutting down.\n", getTime())
    ;
        close(givenParams.logFd);
        exit(EXIT_SUCCESS);
    }
}
```

2.2 Client Side

Client side is easy according to server. Most difficult thing in client, reading result because we don't know size of coming data.

My solution is following;

```
char *query1 = getLine(); //get next line of mine
while(query1 != NULL){
  printf("[%s] Client-%d connected and sending query %s\n", getTime(), givenParams.id, query1);
  clock_t start, end;
  int first = 1 ,isFinished = 0;
  int res;
  char *response = (char*) calloc(MAX + 1,sizeof(char));
  safeWrite(clientSocket, query1, strlen(query1),0); // write query to server
  start = clock();
  res = safeRead(clientSocket,response,MAX,0); //read first coming data
  end = clock();
  if(getQueryTypeEngine(query1) == read){
    do{

      //print result and contiune to read for remain part

    } while((res = safeRead(clientSocket,response,MAX,0)) == MAX);

    if(!isFinished) // if data is not finished print last part
      printData(response);
    printf("\n");
  }
  else{ //if query is write return just number of affected
    printf("[%s] Server's response to Client-%d is %d records affected, and arrived in %.5f seconds\n",
         getTime(), givenParams.id, atoi(response),(double)(end - start) / CLOCKS_PER_SEC);
  }
```

```
  free(response);
  count++;
  query1 = getLine(); //get next line of mine
}
```

3 References that was used

- Advanced Linux Programming Book Chapter 4 Threads.
- Week-10 slides Thread synchronization: Example: readers-writers
- Week-11 slides Socket Programming
- The Linux Programming Interface :Creating a daemon process

Simple Test Result

```
7 SELECT * FROM TABLE;
5 SELECT year FROM TABLE;
1 SELECT DISTINCT year FROM TABLE
3 UPDATE TABLE SET variable = System WHERE year='2005'
2 SELECT * FROM TABLE;
4 SELECT value, Unit FROM TABLE;
3 UPDATE TABLE SET year = 2021 WHERE region='SOUTH_ISLAND'
5 SELECT DISTINCT region, Source, Series FROM TABLE
2 SELECT * FROM TABLE;
4 UPDATE TABLE SET region = Turkey, Unit = Akif  WHERE year='2017'
7 SELECT DISTINCT variable FROM TABLE
1 UPDATE TABLE SET Series = Test, value = 123  WHERE Source='GNS'
6 UPDATE TABLE SET Source = Final WHERE variable='Abstraction by Hydrogeneration'
6 UPDATE TABLE SET variable = Try, year = 2020  WHERE region='NEW_ZEALAND'
```

Figure 1: Query File Before Test

```
akif@ubuntu:~/Desktop/system-programming/Final Project$ ./server -p 3456 -o logFile -l 5 -d water.csv
akif@ubuntu:~/Desktop/system-programming/Final Project$ kill -2 10525
akif@ubuntu:~/Desktop/system-programming/Final Project$
```

Figure 2: Running and Killing Server

```
akif@ubuntu:~/Desktop/system-programming/Final Project$ ./client -i 1 -a 127.0.0.1 -p 3456 -o example2
[Tue Jun  8 22:37:34 2021] Client-1 connecting to 127.0.0.1:3456
[Tue Jun  8 22:37:34 2021] Client-1 connected and sending query SELECT DISTINCT year FROM TABLE
[Tue Jun  8 22:37:34 2021] Server's response to Client-1 is 26 records, and arrived in 0.00014 seconds
        year
1       1995
2       1996
3       1997
4       1998
5       1999
6       2000
7       2001
8       2002
9       2003
10      2004
11      2005
12      2006
13      2007
14      2008
15      2009
16      2010
17      2011
18      2012
19      2013
20      2014
21      2015
22      2016
23      2017
24      2018
25      2019
26      2020

[Tue Jun  8 22:37:34 2021] Client-1 connected and sending query UPDATE TABLE SET Series = Test, value = 123  WHERE Source='GNS'
[Tue Jun  8 22:37:34 2021] Server's response to Client-1 is 494 records affected, and arrived in 0.00020 seconds
[Tue Jun  8 22:37:34 2021] A total of 2 queries were executed, client is terminating.
akif@ubuntu:~/Desktop/system-programming/Final Project$
```

Figure 3: Client Result

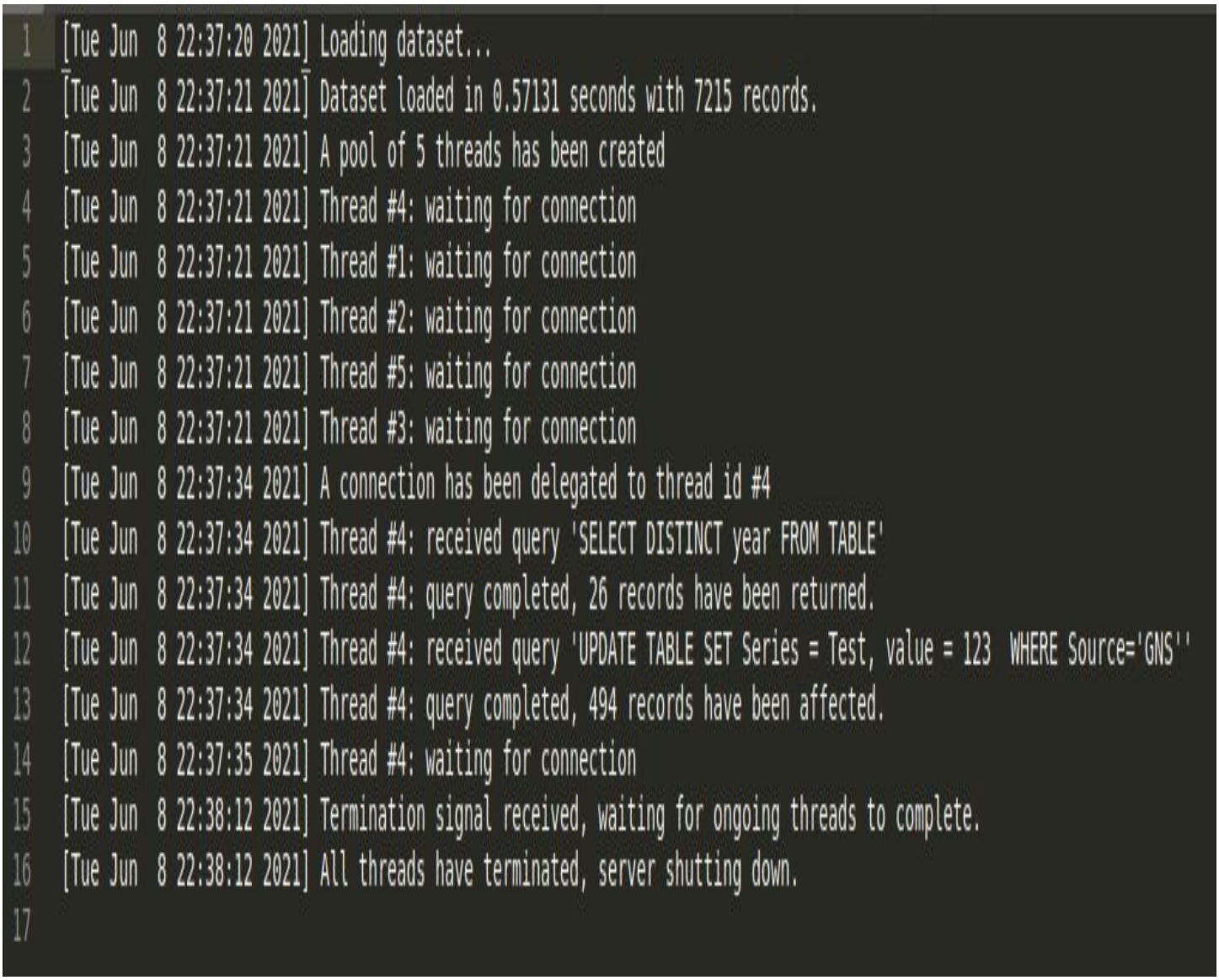

Figure 4: Server Log File