GEBZE TECHNICAL UNIVERSITY
COMPUTER ENGINEERING

SYSTEM PROGRAMMING
CSE 344 - 2020 SPRING

FINAL PROJECT REPORT

SILA BENGİSU YÜKSEKLİ
1801042877

➢ *Explanation of the way of work and the aim of the program*

There are two processes , client and server. After compiling by typing make on terminal, user should give command line arguments like “-a 127.0.0.1 -p 8080 -s 0 -d 500” for client process and “-i inputFile -p 8080 -o outputFile -s 4 -x 25” for server process. Then it can be run by writing “./server command lines arguments” and “./client command lines arguments” from two separate terminals.

These arguments refer to the following:

Server :

- i : the input file that contains graph.
- o: the output file that will be used to write errors or normal outputs to in it.
- p: port number
- s: number of the threads at the beginning
- x: maximum number of threads that the thread pool can be extended

Client:

- a: ip address
- p: port number
- s: requested source vertex
- d: requested destination vertex

File must contain a graph structure. Each row should indicate a edge between a source(from node) and destination(to node). If there are n vertex in file, maximum identity can be n-1.

This program enables data to be exchanged between two processes on the same or different computers. In this scenario, there is/are client/s that request a path from server between two nodes. Server reads this request, calculate the requested path if it is available and send it to client. Two of them are represented as a process. Hence, there are two source files which are client.c and server.c . Also, I write a another source file called helper.c for structs and helper functions like reading file, printing messages, etc.

TCP-based socket structure is used to provide communication between processes.

➢ *Implementation of Graph*

To implement graph data structure, I wrote three structs as shown in below :

```
struct Edge
{
        int dest;
        int source;
};
```

-This struct refers to an edge between two nodes and keeps the destination and source vertex.

```
struct list
{
        int vertexID;
        struct Edge* adjacents;
        int size;
        int capacity;
};
```

-This struct refers to adjacency list of a vertex. The adjacency list is kept as an struct edge pointer and the id of source vertex. Also, size and the capacity of the list are kept.

```
struct Graph
{
        struct list* edges;
        int numOfVertices;
        int numOfEdges;
};
```

-This struct refers a graph implementation. It keeps a neighbor list for every vertex and number of vertices and edges that the graph has.

➢ *Finding a Path Using BFS Implementation*

A function is written for finding a path between requested nodes. If it exists, function returns one, otherwise it returns zero. If a path is found, path is copied into the parameter given in function.

There is a visited array with the size of vertices. I use this array for keeping track of the nodes that is visited before. Also, I have a array called parent. It indicates bfs tree. According to this array, requested path is found. When the current number reaches to desired destination, parent array is followed in reverse and the path is drawn. Also, queue data structure is implemented by using struct to store nodes that are waiting to be visited.

```
struct Queue
{
        int* array;
        int size;
        int capacity;
};
```

- It keeps a dynamic array. Also, it holds size and the capacity of the array.

➢ *Client Process*

This process queries whether there is a path between the two node it receives over the terminal.

It creates a TCP based socket and tries to connect server. It converts its query into a string such as "source id - destination id". If the connection is successful, it sends this string to the server and waits for the answer. Writing to the socket and reading from the socket is done by read and write system calls.

The timeval structure and the gettimeofday function are used to calculate the time it took to get the answer.

I create a two dimensional void pointer called addrs to hold addresses of the pointers that I created. Whenever I allocate space dynamically, I create a critical section by blocking and unblocking sigint signal and I save its address in this array. So, if signal is received during this time, this operation can not be interrupted. In case of receiving signal, I exit gracefully by freeing this addresses using addrs array.

➢ *Server Process*

This ensures that the query received by the client is answered through a TCP-based socket. For this purpose, a thread pool is created by creating the number of threads taken from the terminal(s). The thread pool is a dynamic structure and the maximum can be expanded to the number entered by the user from the terminal(x). For this purpose, a separate thread has been created that deals specifically with this operation. In this program, synchronization problems are solved with mutex and conditional variables.

The main thread is only obliged to take requests and forward them to an idle thread. Answering the queries is done by pool of the threads. The main thread takes the queries from the client and gives this task to an idle thread. If there is no thread in idle, it waits until a thread is finished. This part of the code as shown in below :

```
newSocket_fd = accept(socket_fd, (struct sockaddr*) &newAddr, &lenNewAddr);
pthread_mutex_lock(&mutex);
while(*busyCount == *s) {
        server_noThreadAvailable(fd2);
        pthread_cond_wait(&cond_busy,&mutex);
}

addElement(queue,newSocket_fd);
pthread_cond_signal(&cond);
pthread_mutex_unlock(&mutex);
```

There is a global variable called busyCount and it shows the number of the threads that work to handle a connection. It is protected by mutex, because it is a common variable. Main thread checks the value of this variable. If it reaches the number of threads that are created, it waits by using pthread_cond_wait. Whenever a thread handles a connection, value of this variable is decreased and the current thread sends signal by using pthread_cond_signal as shown in the code snippet below :

```
write(client_fd,ans,10000);
-- *busyCount;
pthread_cond_signal(&cond_busy);
...
```

When main thread receives a request from client, it accept the connection and push the file descriptor returned from accept function into the queue. Then it sends signal to inform threads. If the queue is empty, the threads are waiting until the queue has a element and the main thread informs them. This news is given by using conditional variable called cond. Waiting in thread function is shown here :

```
pthread_mutex_lock(&mutex);
while(isEmpty(queue) == 1) {
        pthread_cond_wait(&cond,&mutex);
}

client_fd = removeFront(queue);
++ *busyCount;
…
```

A cache structure was used to speed up the path calculation. To implement cache, a struct is written called cacheList :

```
struct cacheList
{
        struct Edge edge;
        char* graphPath;
};
```

It keeps an edge to indicate the source and the destination vertex and a string to hold path between these nodes. I create an array of this struct to define a cache structure.

When a path was calculated for the first time, it was written in cache. Thus, when the same query is made again, the process is accelerated and the result is read directly from the cache without recalculating. When threads takes the task from main thread, it reads the request from socket. Then it looks into the cache first and looks for the answer. If it cannot find it, it calculates and sends this path to the client and writes it in cache.

Threads both read from cache and write to cache. For this purpose, synchronization in the example of classical readers-writers was applied. The code written for this purpose is as follows :

For readers :

```
pthread_mutex_lock(&mutexCache);
while ((*AW + *WW) > 0) {
        *WR += 1;
        pthread_cond_wait(&okToRead,&mutexCache);
        *WR -= 1;
}
*AR += 1;
pthread_mutex_unlock(&mutexCache);
int check = checkPastCalculations(cache, cacheSize, ans, src, dst);
pthread_mutex_lock(&mutexCache);

*AR -= 1;
if (*AR == 0 && *WW > 0)
        pthread_cond_signal(&okToWrite);
pthread_mutex_unlock(&mutexCache);
```

For writers :

```
pthread_mutex_lock(&mutexCache);
while ((*AW + *AR) > 0) {
        *WW += 1;
        pthread_cond_wait(&okToWrite, &mutexCache);
        *WW -= 1;
}

*AW += 1;
pthread_mutex_unlock(&mutexCache);
int present = isPresent(cache, cacheSize, src, dst);
if (present == 0)
{
        cache[*cacheSize].edge = _edge;
        strcpy(cache[*cacheSize].graphPath,ans);
        ++ *cacheSize;
}

pthread_mutex_lock(&mutexCache);
*AW -= 1;
if (*WW > 0)
        pthread_cond_signal(&okToWrite);
else if (*WR > 0)
        pthread_cond_broadcast(&okToRead);
pthread_mutex_unlock(&mutexCache);
```

- AR : active readers
- AW : active writers
- WW : waiting writers
- WR : waiting readers

The threads trying to look for a path in the cache are the readers and the threads that are calculate a path and want to add this path into cache are writers. Writers have priority. When a reader finish its job, if active readers equals to zero and waiting writers are greater than zero, it sends a signal by using okToWrite conditional variable. When a writer finish its job,if waiting writers are more than zero, it sends signal by using okToWrite conditional variable. If waiting readers are more than zero, it sends to readers by using okToRead conditional variable. A mutex named mutexCache was also used to synchronize between the threads of the pool.

If the same query was made, on the two different thread side, since the program runs very fast, both of them will not find this query in cache and start calculating. So before I write, I check once again whether the same path is found in the cache.

Since only one writer can be found at the same time at that line, adding same path in cache is prevented. However, since the same request is made without writing the cache in the first query, that second thread also calculates the path. But it doesn't add it to cache.

➔ *Resizer Thread*

When the number of busy thread reaches seventy-five percent of the number of thread created, the number of threads in the system is increased by twenty-five percent of the threads so as not to exceed the maximum number of threads. A thread was created specifically for this job.
Whenever the number of busy threads is increased, the thread sends a signal using the cond_resize conditional variable.

Resizer thread is waiting with pthread_cond_wait as shown below, until this condition is met. To provide this, busyCount variable is checked and it is protected by using mutex, because resizer, main thread and threads of the pool use this variable.

```
pthread_mutex_lock(&mutex);
th_count = *s;
while(*busyCount <= th_count*0.75) {
        pthread_cond_wait(&cond_resize,&mutex);
}
```

If the number of threads is less than four, twenty-five percent will be less than 1, so I directly increased the number of threads by one.

```
if (*s < *x)
{
        if (th_count <= 4) {
        *s = th_count + 1;
}

else {
        *s = (int) th_count + th_count*0.25;
        if (*s > *x)
        {
                *s = *x;
        }
}
...
pthread_mutex_unlock(&mutex);
```

➢ *How to prevent two instantiation of server process*

To provide this, I create a shared memory with O_EXCL flag. So, if a second server process is run by terminal, it returns -1 and program exits. That's way it is not possible to have two instantiation of server process.

Note : if there is a problem and the program ends without shm_unlink, you should delete this file to try the program again. Otherwise, the program will not start. The file is saved under dev / shm path.