Final Project CSE344
System Programming
Hamza YOĞURTCUOĞLU - 171044086

➜ **server.c :** I put the **Usage** at first. I seperated command line argument with **getopt(). I did extra point please write like following command:**

./server -i pathToFile -p PORT -o pathToLogFile -s 4 -x 24 -r 1

As stated in the homework pdf file, the input file is thought to be given in the proper format (**Internet peer-to-peer networks datasets** (https://snap.stanford.edu/data/)).

Any comments can be written with a # at the beginning of the file. Not all line with # character was created while creating Graph. You can write following example :

```
1  # Directed graph (each unordered pair of nodes is saved once): p2p-Gnutella08.txt
2  # Directed Gnutella P2P network from August 8 2002
3  # Nodes: 63ASDAS01 Edges: 2ASDASD0777
4  # FromNDASDodeId    ToNodeIASDAd
5  # Hamza YOĞURTCUOĞLU
6  0   1
7  0   2
8  0   3
9  0   4
```

...

...

```
24  3   1898
25  # GTU
26  # Hamza YOĞURTCUOĞLU
27  3   1899
28  4   144
29  4   258
30  4   491
31  4   1021
```

Anywhere you can put comments with #

NOTE: There must be newline/s end of the file.

IS SERVER PROCESS DAEMON ?

I did following issues in order to do daemon process.

No Double Instantiation → I kept a shared array in shared memory. If a process is run for the first time, it assigns **Y and E** characters to this shared array. If the server process is run again, it will understand that the server process is running and write the error message on the screen and terminate itself.

When the running program ends, it assigns **N and O** characters to the shared arrays and ends.

Error Message : The server is already running. You can't run it again.

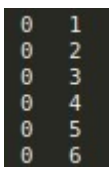
No Controlling Terminal → Opening a new process and exiting from the main process. New process has no controlling terminal through **setsid**().

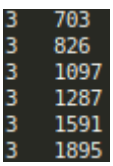
Close All of Inherited Open Files → The main thread just closed the file(that is log file) it had opened before it terminated.

HOW TO CREATE A GRAPH AND THIS GRAPH IS LOADED ?

Firstly, how many **nodes** and **edges** are found by reading the file. Then, by returning to the beginning of file, all the individual elements assign to the vector as follows. Each node (Example 98) holds a vector in itself. Each node is filled into its own **vector** with its own edges.

For Example :





Node Node's Vector

(0)	1	2	3	4	5	6	…

(3)	703	826	1097	1287	1591	1895	…

…

…

Note : Very fast access can be provided by vector.

After the loading of the graph, the socket is created. **Setsockopt()** is used in order to allocate control timeout. Then all threads are created.

HOW THE MAIN THREAD ACCEPTED THE SOCKET TO WORKER THREADS?

After creating the main thread graph loading processes, worker threads wait only the request message from the client in the **accept function**. When the client sends a request, this thread connection is assigned into the socket point of an empty thread that is awakened by the conditional variable.

HOW IS THREAD POOL EXTENDED ?

The main thread only transfers the incoming connection links to the worker threads. Therefore, an **extended thread** has been implemented. The main thread awakens the extender thread whenever a connection link is received. All thread information is in the form of **linklist**. If the thread pool operates 75 percent, the extend thread is expanded by 25 percent (according to integer). When thread pool size reaches **x**, no expansion is done.

If the thread pool cannot be expanded yet and **all worker threads are running**, **the main thread is waiting** in the conditional variable. This conditional variable is sent a signal whenever the thread finishes its job.

HOW DOES CACHE WORK?

The worker thread first reads the socket. NODE {source and destination} comes from this socket. The worker thread checks the cache before going to the main graph.

Note: Cache is approximately 10-15 times faster than graph searching. Check output section.

→ CACHE STRUCTURE

```
struct cache {
    struct vector datas;
    struct cache * next;
};
```

Cache is a **linklist of struct cache**. This data structure contains a path-holding vector and a struct cache pointer is pointing the other next path.

```
struct workerThread{
    int id;
    int  socketFd;
    struct Nodes * node;
    pthread_cond_t conditionalV;
    pthread_mutex_t mutex;
    int full;
    int * logFd;
    struct vector * v;
    int nodeNumber;
    int * currentWorkThread;
    struct workerThread * next;
    struct cache * c;
    int r;
};
```

"**cache head**" exists in the object of all threads. In this way, all threads have the same cache head. If the desired source and destination path are available in the cache, the client socket is sent immediately to the desired path. **Struct vector datas** keeps path.

HOW TO ADJUST THE CACHE WRITING THREAD AND READ THREAD PRIORITY ?

(BONUS PART)

I have reviewed how to implement this bonus part from the book below.

Allen B. Downey,The Little Book of Semaphores 65-80 (reader priority)

BIL 344 System Programming 8 week slide 30-31 → (writer priority)

-r 0 : reader prioritization : While a any reader can read the cache, all other readers can read the cache at the same time. But cache mutex locked by readers. Writer cannot write in cache until no reader is left.

-r 1 : writer prioritization : If any writer waiting in order to get in critical section any reader cannot be got in. When the last reader leaves the critical section, the writer enters directly. Whether or not any reader is waiting. So I keep entering reader and waiting writer in critic section. There are difference mutexes for writerMutex and readerMutex. But reader prioritization has just 1 common mutex that is roomEmptyMutex.

-r 2 : equal priorities to readers and writers : There is only one mutex. Any reader or writer comes up and processes and unlocks mutex. So only one thread can be in the critical section.

IF THERE IS NO PATH IN CACHE, HOW DOES BREADTH FIRST SEARCH WORK IN THE GRAPH ?

(https://www.guru99.com/breadth-first-search-bfs-graph-example.html)

I implemented as above link bfs expression. What I did. I explained briefly;

First of all, we add the source point to the queue as the first data. Then, as long as the queue array is equal's counter, we do the following.

I) If the queue is empty, end the program. If not, take the node at the top of the queue.

II) Travel all the children of this node with the help of the loop.

III) Add all your children who haven't been traversed before to the queue.

IV) Delete current node from queue since traversing is finished.

V) Go back step I.

```
if ((*v)[front].edges[i] == destination)
    thereIsPath = 1;
```

Front node and destination are equal. We can say that there is at least one path. If we cannot find this equation anywhere, client empty path is sent. But we keep current shortest distance between source node and destination node in BFS. Each step I check if I can improve the less distance.

IF ALL THREADS ARE WORKING,HOW IS THE MAIN THREAD UNDERSTAND TO WAIT?

The **currentWorkThread** variable is a common variable for all threads. If this variable is equal to the maximum working variable, the main thread conditional variable is also waiting. (After each thread task is finished, it sends a signal to this conditional variable.)

client.c : The client process only sends the **struct node {destination, source}** to the server. In response, the path length and path are taken from the server. If the **incoming path length is 0**, it means that there is no path. The results are displayed on the screen.

```
Client (3123) connecting to 127.0.0.1:6006
Client (3123) connected and requesting a path from node 0 to 999
Server's response to (3123): 0->7->5577->4611->1792->3755->22506->273->276->999, arrived in 0.022934seconds.
```

or

```
Client (13595) connecting to 127.0.0.1:6005
Client (13595) connected and requesting a path from node 0 to 99999
Server's response (13595): NO PATH, arrived in 0.021924seconds, shutting down
```

→ HOW DOES DAEMON SERVER PROCESS BE KILLED AND RELEASE ALL RESOURCES? ?

If the server receives the SIGINT signal (through the `kill` command) it must wait for its ongoing threads to complete, return all allocated resources, and then shutdown gracefully.

Command : kill -2 [processId] #SIGINT COMMAND

NOTE : Please just send this SIGINT command. Do not try to close the server with other ways and other kill commands. This is a problem because I use shared memory. Just use SIGINT to kill. All resources are released and there are no error or warning messages.

Due to the **memory processes** you run with valgrind, the entire (loadGraph) process and response time to requests are prolonged. It takes 1 million edge 30 seconds with valgrind. 4 seconds without valgrind load 1 million edge.

```
==5174== HEAP SUMMARY:
==5174==     in use at exit: 0 bytes in 0 blocks
==5174==   total heap usage: 506,842 allocs, 506,842 frees, 11,380,968 bytes all
ocated
==5174==
==5174== All heap blocks were freed -- no leaks are possible
==5174==
==5174== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==5174== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figure → There is no leak memory or errors in server program.

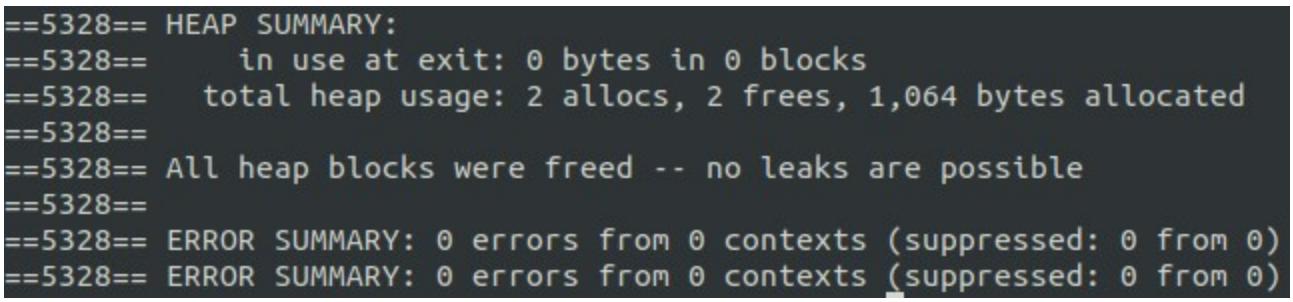


Figure → There is no leak memory or errors in client program.

In SIGINT handler function, terminate is converted to 0, so that the program ends **after the last worker sent the message to the client.**

Then main thread releases all allocated source and no zombies.

Test Program Examples: The following commands write how you can run process.

You can execute with Absolute path or relative path for files.

Execute Server

./server -i p2p-Gnutella31.txt -p 6005 -o logFile.txt -s 4 -x 8 -r 2

./server -p 6005 -i p2p-Gnutella31.txt -s 4 -r 2 -o logFile.txt -x 8

./server -i (AbsolutePath)/p2p-Gnutella31.txt -p 6005 -o (AbsolutePath)/logFile.txt -s 4 -x 8 -r 2

Execute Client

./client -a 127.0.0.1 -p 6005 -s 0 -d 999

./client -p 6005 -a 127.0.0.1 -s 0 -d 999

Kill Command

kill -2 [processID] #SIGINT COMMAND

NOTE : Please just send this SIGINT command. Do not try to close the server with other ways and other kill commands.

→ Input File Examples: The file format should be as described on page 1. the input file is thought to be given in the proper format(**Internet peer-to-peer networks datasets** (https://snap.stanford.edu/data/)).

→ Output Example : All output is printed to the desired log file in **real time**.

./server -i p2p-Gnutella31.txt -p 6005 -o log.txt -s 4 -x 8 -r 2

```
Executing with parameters:
-i p2p-Gnutella31.txt
-p 6005
-o log.txt
-s 4
-x 6
-r 2
Loading graph...
Graph loaded in 3.7 seconds with 62586 nodes and 147892 edges.
A pool of 4 threads has been created
Thread #0: waiting for connection
Thread #1: waiting for connection
Thread #2: waiting for connection
Thread #3: waiting for connection
```

Server: First output when running Server. 1 mllion edge takes 3.7 second to load graph

I write script that sends 500 client request to server. You can see outputs.

```
A connection has been delegated to thread id #0, system load 25.0%
Thread #0: searching database for a path from node 0 to node 999
Thread #0: no path in database, calculating 0->999
Thread #0: path calculated: 0->7->5577->4611->1792->3755->22506->273->276->999
Thread #0: responding to client and adding path to database
```

Server: There is no path in cache 0→999 But there is path in graph.

```
Client (5908) connecting to 127.0.0.1:6005
Client (5908) connected and requesting a path from node 0 to 999
Server's response to (5908): 0->7->5577->4611->1792->3755->22506->273->276->999, arrived in 0.031967seconds.
```

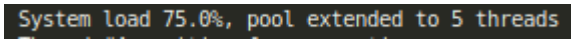
Client: takes 0.031967 seconds.

```
A connection has been delegated to thread id #1, system load 50.0%
Thread #1: searching database for a path from node 0 to node 999
Thread #1: path found in database: 0->7->5577->4611->1792->3755->22506->273->276->999
```

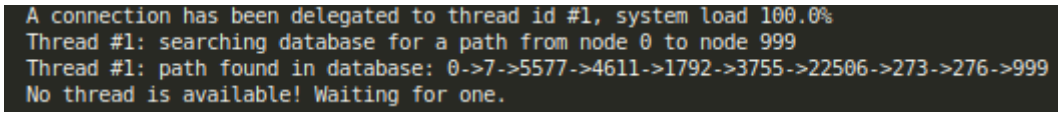
Server: Other client send same source and destination 0→999 there is path in cache.

```
Client (5909) connecting to 127.0.0.1:6005
Client (5909) connected and requesting a path from node 0 to 999
Server's response to (5909): 0->7->5577->4611->1792->3755->22506->273->276->999, arrived in 0.000665seconds.
```

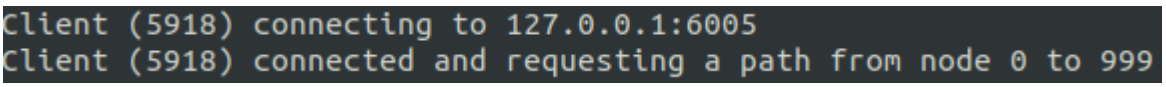
Client: takes 0.000665 seconds. Because there is path in cache. Vector so fact :)



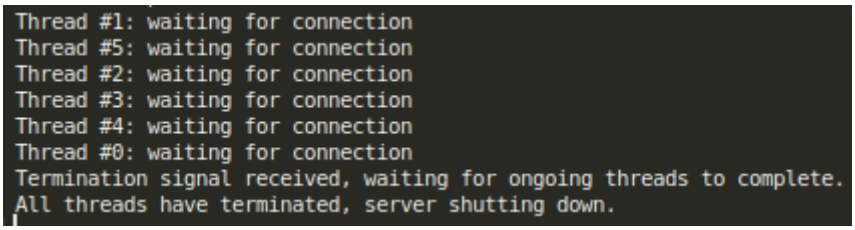
Server: If the thread load is 75.0% and above, the extender thread extends thread pool25.0%.



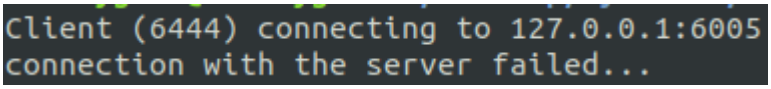
Server:If all threads are full and the maximum thread count has been reached, the main thread is waiting.



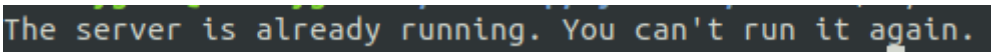
Client: If all threads are full, the client will send the connection anyway and wait until it is answered.



Server: waiting for its ongoing threads to complete, returning all allocated resources, and then shutdown gracefully.



Client: You send request. But server is not running. You will get above error message:)



Server: If a server is running and trying to open a server again, you will get the above error message:)

Note: All Internet peer-to-peer networks datasets files have been tried. Please do not forget shut down server with (kill -2 processID) not other kill commands.

→ Compiler and Run :

Makefiles Commands:

```
make  #that command compiler
make clean #that command cleans all object files
```

To run :

```
./server -i p2p-Gnutella31.txt -p 6005 -o logFile.txt -s 4 -x 8 -r 2
./client -a 127.0.0.1 -p 6005  -s 0 -d 999
```

If you want to see outputs. Check above **outputs.**

→ Note : 1) There is a newline(‘\n’) end of inputFile.
2) Please do not forget shut down server with (kill -2 processID) not other kill commands.
3) I did bonus part. Do not forget -r parameter.
4) I tried 2 million nodes and 5 million edges. Program work perfect but load operation takes 120 seconds.

Thanks :)