Final Project

Elif Akgün
1801042251

1. Problem Definition

This project includes two programs: server and client. The server which is daemon loads a graph from text file, handles incoming connections from client by using dynamic pool. Every time that the load of the server reaches 75%, i.e. 75% of the threads become busy, then the pool size will be enlarged by 25%. The client(s) connect to the server and request a path between two arbitrary nodes, and the server will provide this service.

1.1 Server

The server receives the following command line arguments:

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

pathToFile: denotes the relative or absolute path to an input file containing a directed unweighted graph (from https://snap.stanford.edu/data/)

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).

s: this is the number of threads in the pool at startup (at least 2)

x: this is the maximum allowed number of threads, the pool must not grow beyond this number.

BONUS
Enable the -r commandline argument of the server:
r = 0 : reader prioritization
r = 1 : writer prioritization
r = 2 : equal priorities to readers and writers

The server process communicates with the client processes through stream/TCP socket based connections. The server possess a pool of POSIX threads, and in the form of an

endless loop, as soon as a new connection arrives, it will forward that new connection to an available thread of the pool, and immediately continue waiting for a new connection. If no thread is available, then it waits in a blocked status until a thread becomes available.

The pool of threads created and initialized at the daemon's startup, and each thread executes in an endless loop, first waiting for a connection to be handed to them by the server, then handling it, and then waiting again, and so on. Once a connection is established, the client sends two indexes i1 and i2, as two non negative integers, representing each one of the nodes of the graph, to the server, and waits for the server to reply with the path from i1 to i2. Server uses BFS to find a path form i1 to 12. If the requested path from i1 to i2 has been already calculated during a past request (by the same or other thread), the thread handling this request should first check a "cache data structure", containing past calculations, and if the requested path is present in it, the thread should simply use it to respond to the client, instead of recalculating it. If it is not present in the "cache data structure" then it must inevitably calculate it, respond to the client, and add the newly calculated path into the data structure, so as to accelerate future requests. If no path is possible between the requested nodes, then the server responds accordingly. The cache data structure is common to all threads of the pool and can be thought of as a "database". It does not contain duplicate entries.

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

1.2 Client

Given the server's address and port number, client requests a path from node i1 to i2 and wait for a response, while printing its output on the terminal.

The client receives the following command line arguments:

./client -a 127.0.0.1 -p PORT -s 768 -d 979

a: IP address of the machine running the server
p: port number at which the server waits for connections
s: source node of the requested path
d: destination node of the requested path

2. Implementation

2.1 Server

In server process, I made also bonus part. According to r's values, threads created related pool. If r is equal to 0, then threadPoolR function is used; If r is equal to 1 or -1(when r is not included in the command line), then threadPoolW function is used; and if r is equal to 2, then threadPoolRW function is used.

2.1.1 Main

Server become daemon with *daemon_* function and fork, setsid, umask, chrdir system calls used for this purpose.After fork parent process terminates and the child process runs in the background. To prevent double instantiation, I used *createLockFile* function. To prevent double instantiation, I create a file (*lockFile*), then I locked this file. So if another instantiation of the program wants to run it, file lock prints error message.
Input file was read then the graph was loaded to memory with *loadGraph* function. Then the socket and threads were created.
In while loop, server waits for new connection. When client makes a request, server signals to thread pool to notify that a request is made. If SIGINT comes, then server break the loop and waits for threads.

2.1.2 threadPoolW

This pool implemented by readers/writers paradigm, by prioritizing writers. First of all, all threads wait signal(*condVar*) from main thread. When a signal came, one of threads take mutex and goes on. Take accept system call's file descriptor, then sends signal (*condVar2*) to main again to notify that signal is tooken. Thus, main thread waits for pool threads until one of threads takes request. So, if there is no available thread, main thread waits. Then, thread increases *busyThrNum*, which keeps number of busy threads, then sends signal(*condVar3*) to tracker thread to notify busy thread number was increased. Then, reads the source and destination nodes. It checks the cache, if it includes this path, thread responses to client. If it does not, it finds path with *BFS* function. Actually, this function finds visited array from source node to destination node. Then *absPath* function finds path from visited array. After path was found, thread responses to client and writes path to cahce. If graph does not include this path, then server sends to client -1. It means that, this path is not possible.

According to readers/writers paradigm, reader threads wait if there is any writer in cache or waiting. If there no waiting writer thread or active writer thread, then reader thread can read the database. On the other hands, Writers wait if there are active reader or active writer. For readers/writers paradigm, I used *m* mutex, *okToRead* and *okToWrite* conditional variables and *AR*, *AW*, *WW*, *WR* variables.

For other synchronization problem, I used *mutex*,*mtx*, *condVar*, *condvar2*, and *condVar3*. Main thread also used *mutex* so they run synchronously.

2.1.3 threadPoolR

This function same as *threadPoolR*. There is one difference: this pool implemented by readers/writers paradigm, by prioritizing readers. Reader threads wait if there is any writer in cache. If there is no active writer thread, then reader threads can read the database. On the other hands, writers wait if there are active reader, active writer or waiting reader.

2.1.4 threadPoolRW

This function same as *threadPoolW* and *threadPoolR*. There is one difference: this pool implemented by readers/writers paradigm, by priorities equally. When reader thread wants read, it waits for mutex, takes mutex, read cache, then unlocks the mutex. As the same, when writer thread wants write to cache, it waits for mutex, takes mutex, wirtes to cache, then unlocks the mutex.

2.1.5 tracker

This thread tracks the load of pool, if it increase 75% then enlarges it by 25%. If load less than 75%, it waits until load increases. If thread numbers reache maximum thread number, it does not enlarge the pool. When SIGINT came, breaks the infinite loop and waits for threads that it created.
For this synchronization problem, I used *mtx* and *condVar3*. When busy thread number is increase, threads send signal to tracker with *condvar3*.

2.1.6 handler

When SIGINT came, handler handles this signal and assign 0 to *isRuning* to notify other threads teminating signal has come.

2.1.7 cache data structure

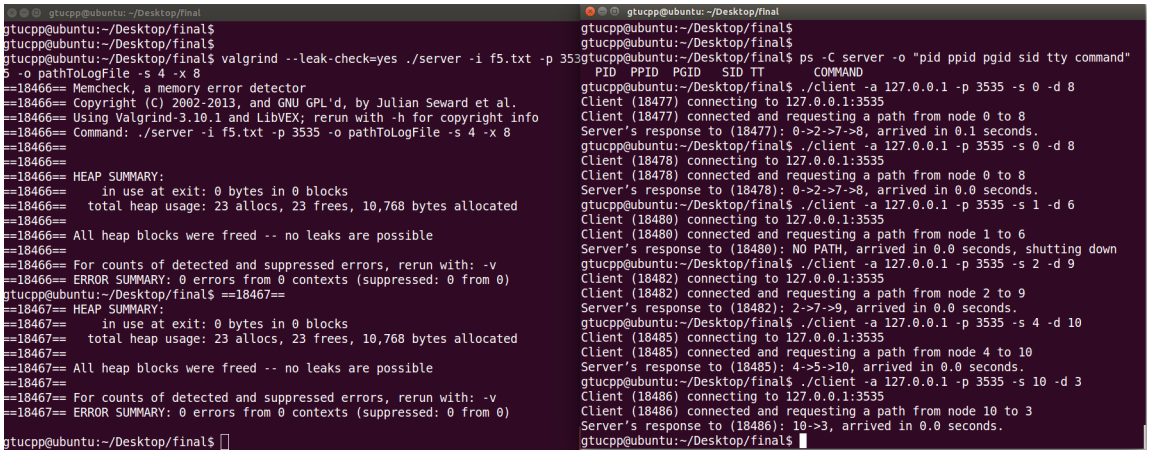
I used 2D dynamic array for cache. Firstly its size is 1*1. As the request calculates and writes to cache, its row enlarged one by one. If new path's size is larger than cache's column size, then column resizes.

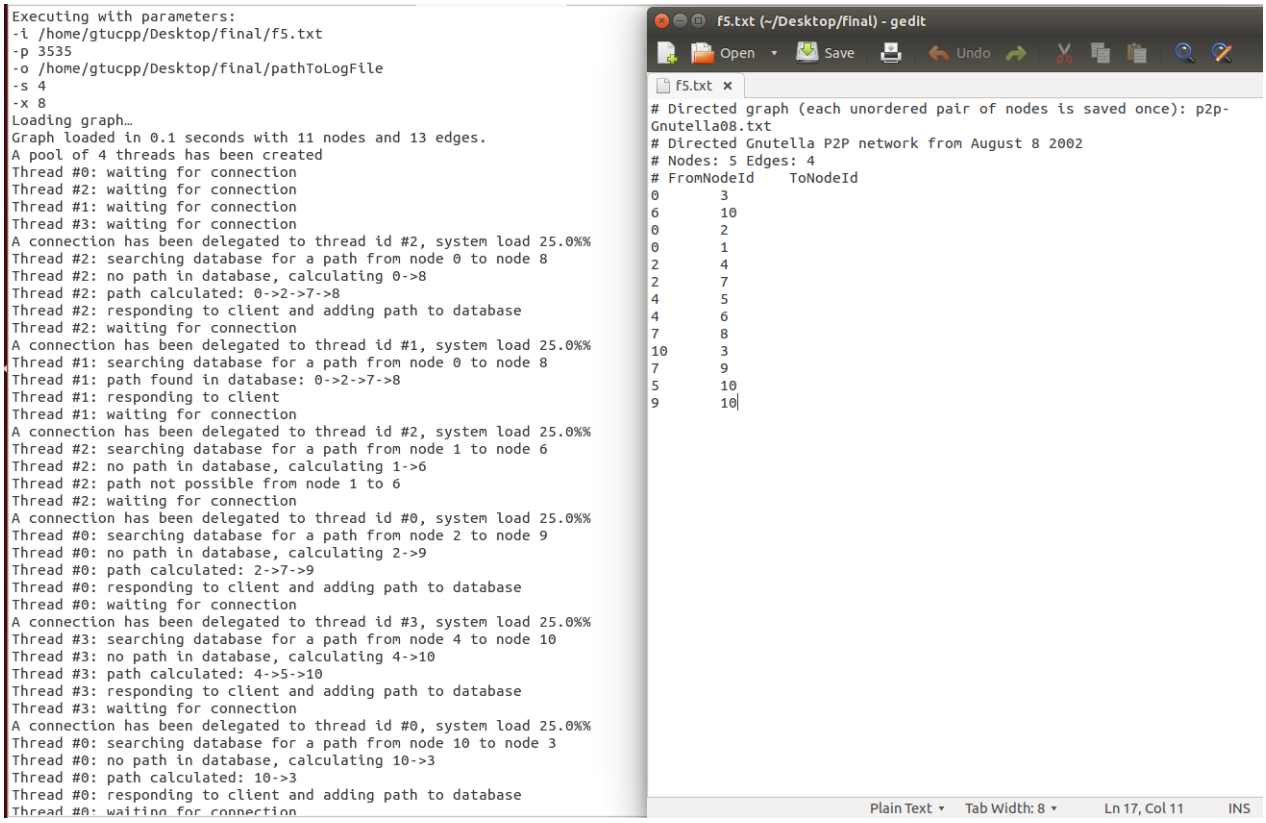
2.2 Client

Client process more simple than server process. It creates socket then sends to source and destination nodes, which are readed from command line. It reads server's respons and wirtes path to terminal.

3. Tests

3.1 Sample Input/Output





3.2 Testing Command Line Arguments

```
gtucpp@ubuntu:~/Desktop/final$ ./server -i f5.txt -p 3535 -o pathToLogFile -s 4 -x 8 -r 1 k
Usage: ./server -i <pathToFile> -p <port_num> -o <pathToLogFile> -s <thread_num> -x <max_thread_num>
-i: denotes the relative or absolute path to an input file containing a directed unweighted
graph from the Stanford Large Network Dataset Collection
-p: this is the port number the server will use for incoming connections.
-o: is the relative or absolute path of the log file to which the server daemon_ will
write all of its output (normal output & errors).
-s: this is the number of threads in the pool at startup (at least 2)
-x : this is the maximum allowed number of threads, the pool must not grow beyond this number
**BONUS**
-r 0 : reader prioritization
-r 1 : writer prioritization
-r 2 : equal priorities to readers and writers
gtucpp@ubuntu:~/Desktop/final$ ./server -i f5.txt -p 3535 -o pathToLogFile -s 4 -x 8 -r
Usage: ./server -i <pathToFile> -p <port_num> -o <pathToLogFile> -s <thread_num> -x <max_thread_num>
-i: denotes the relative or absolute path to an input file containing a directed unweighted
graph from the Stanford Large Network Dataset Collection
-p: this is the port number the server will use for incoming connections.
-o: is the relative or absolute path of the log file to which the server daemon_ will
write all of its output (normal output & errors).
-s: this is the number of threads in the pool at startup (at least 2)
-x : this is the maximum allowed number of threads, the pool must not grow beyond this number
**BONUS**
-r 0 : reader prioritization
-r 1 : writer prioritization
-r 2 : equal priorities to readers and writers
gtucpp@ubuntu:~/Desktop/final$ ./server -i f5.txt -p 3535 -o pathToLogFile -s 4 -x
Usage: ./server -i <pathToFile> -p <port_num> -o <pathToLogFile> -s <thread_num> -x <max_thread_num>
-i: denotes the relative or absolute path to an input file containing a directed unweighted
graph from the Stanford Large Network Dataset Collection
-p: this is the port number the server will use for incoming connections.
-o: is the relative or absolute path of the log file to which the server daemon_ will
write all of its output (normal output & errors).
-s: this is the number of threads in the pool at startup (at least 2)
-x : this is the maximum allowed number of threads, the pool must not grow beyond this number
**BONUS**
-r 0 : reader prioritization
-r 1 : writer prioritization
-r 2 : equal priorities to readers and writers
gtucpp@ubuntu:~/Desktop/final$
```

```
gtucpp@ubuntu:~/Desktop/final$ ./client
Usage: ./client -a <ip_addr> -p <port_num> -s <src_node> -d <dest_node>
-a: IP address of the machine running the server
-p: port number at which the server waits for connections
-s: source node of the requested path
-d: destination node of the requested path
gtucpp@ubuntu:~/Desktop/final$ ./ client -a 192.168.1.1 -p 3535 -s 4 -d
bash: ./: Is a directory
gtucpp@ubuntu:~/Desktop/final$ ./client -a 192.168.1.1 -p 3535 -s 4 -d
Usage: ./client -a <ip_addr> -p <port_num> -s <src_node> -d <dest_node>
-a: IP address of the machine running the server
-p: port number at which the server waits for connections
-s: source node of the requested path
-d: destination node of the requested path
gtucpp@ubuntu:~/Desktop/final$ ./client -a 192.168.1.1 -p 3535 -s 4 -d 6 p
Usage: ./client -a <ip_addr> -p <port_num> -s <src_node> -d <dest_node>
-a: IP address of the machine running the server
-p: port number at which the server waits for connections
-s: source node of the requested path
-d: destination node of the requested path
gtucpp@ubuntu:~/Desktop/final$ ./client -a 192.168.1.1 -p 3535 -s
Usage: ./client -a <ip_addr> -p <port_num> -s <src_node> -d <dest_node>
-a: IP address of the machine running the server
-p: port number at which the server waits for connections
-s: source node of the requested path
-d: destination node of the requested path
gtucpp@ubuntu:~/Desktop/final$ ./client -a 192.168.1.1 -p 3535
Usage: ./client -a <ip_addr> -p <port_num> -s <src_node> -d <dest_node>
-a: IP address of the machine running the server
-p: port number at which the server waits for connections
-s: source node of the requested path
-d: destination node of the requested path
gtucpp@ubuntu:~/Desktop/final$
```

3.3 Testing Double Instantiation

```
p@ubuntu:~/Desktop/final$
p@ubuntu:~/Desktop/final$
p@ubuntu:~/Desktop/final$ ./server -i f5.txt -p 3535 -o pathToLogFile -s 4 -x 8
p@ubuntu:~/Desktop/final$
gtucpp@ubuntu:~/Desktop/final$
gtucpp@ubuntu:~/Desktop/final$ ./server -i f5.txt -p 3556 -o pathToLogFile -s 4
-x 8 -r 1
gtucpp@ubuntu:~/Desktop/final$ It is not possible double instantiation! Exiting.
```

3.4 Testing SIGINT Signal

```
A connection has been delegated to thread id #0, system load 20.0%%
Thread #0: searching database for a path from node 0 to node 1
Thread #0: path found in database: 0->1
Thread #0: responding to client
Thread #0: waiting for connection
A connection has been delegated to thread id #1, system load 20.0%%
Thread #1: searching database for a path from node 0 to node 2
Thread #1: path found in database: 0->2
Thread #1: responding to client
Thread #1: waiting for connection
Termination signal received, waiting for ongoing threads to complete.
All threads have terminated, server shutting down.
```