June 12th, 2020

CSE344 – System Programming – Final project - v2
Processes, Sockets, Threads, IPC, Synchronization

Context and objective

Overall, you are expected to develop 2 programs. A threaded server and a client. The server will load a graph from a text file, and use a dynamic pool of threads to handle incoming connections. The client(s) will connect to the server and request a path between two arbitrary nodes, and the server will provide this service. However, things are not as simple as they seem.

*****Server*****

The server process will be a **daemon**, this means that you should at least:

- take measures against double instantiation (it should not be possible to start 2 instances of the server process)
- make sure the server process has no controlling terminal
- close all of its inherited open files

The server will receive the following command line arguments (more to come later):

```
./server –i pathToFile –p PORT –o pathToLogFile
```

`pathToFile`: denotes the relative or absolute path to an input file containing a directed unweighted graph from the Stanford Large Network Dataset Collection (https://snap.stanford.edu/data/), an example is included in this archive.

`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).

Main thread: The server process will load the graph to memory from its input file (file structure is straightforward and explained in it). It is up to you to decide how to model the graph. The server process will communicate with the client process(es) through stream/TCP socket based connections.

The server will 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 (i.e. not occupied) thread of the pool, and immediately continue waiting for a new connection. If no thread is available, then it will wait in a blocked status until a thread becomes available. Be careful how you implement this waiting procedure.

Thread pool: The pool of threads will be created and initialized at the daemon's startup, and each thread (executing of course the same function) will execute 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.

Handling a connection: Once a connection is established, the client will simply send two indexes i1 and i2, as two non negative integers, representing each one of the nodes of the graph, to the server, and wait for the server to reply with the path from i1 to i2. Use breadth-first search (BFS) to

find a path from i1 to i2 (yes, this approach is suboptimal but mandatory all the same). The indexes of the nodes will be those contained in the input file (so don't fool around naming nodes arbitrarily). The exact format of the stream messages is up to you to decide.

This was the easy part.

Efficiency: The graphs that you load from the files, can be potentially very large with millions of nodes and edges. Make sure your graphs and your BFS implementation are such that requests don't take "forever".

That is why, you will avoid recalculating paths if the same request arrives again. 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.

Update: assuming we have a path in the cache calculated from 0->3->5->1->2 and now the client requests a path from 3 to 2. Is it to be considered as known or should it be calculated explicitly? It will be considered as unknown, and calculated explicitly. I know it's subtoptimal, this is not a data structures/algorithms course, I want to keep it simple, so that you'll focus on system programming aspects.

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 will respond accordingly.

The cache data structure will be common to all threads of the pool and can be thought of as a "database". It must not contain duplicate entries, and it must support fast access/search. You decide the data structure type and justify your decision in your report. It can be anything (e.g. array, list, tree, etc but not a file).

This database poses multiple and serious synchronization risks. Implement it following the readers/writers paradigm, by prioritizing writers. Readers are the threads attempting to search the data structure to find out whether a certain path is in it or not (will happen often, and will take longer when the database grows). Writers are the threads that have calculated a certain path and now want to add it into the data structure (very frequent in the beginning).

Start BONUS
For extra +30 points, enable the `-r` commandline argument of the server,

`-r 0` : reader prioritization
`-r 1` : writer prioritization
`-r 2` : equal priorities to readers and writers
End BONUS

Dynamic Pool: Furthermore, it was mentioned earlier that the pool would be "dynamic". This means it will not have a fixed number of threads, instead the number of threads will increase depending on how busy the server is. Specifically, 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%. Use an additional thread to keep

track of the pool's size, server load and pool resizing, we don't want to occupy the server/main thread with anything else besides connection waiting. Detail in your report how you implement this additional resizer thread.

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

`-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.

Output: The server will write all of its output and errors to the log file provided at startup. If it fails to access it for any reason then make sure the user is notified through the terminal before the server detaches from the associated terminal.

Example output for the server (**include a timestamp at the beginning of every line**):

```
Executing with parameters:
-i /home/erhan/sysprog/graph.dat
-p 34567
-o /home/erhan/sysprog/logfile
-s 8
-x 24
Loading graph…
Graph loaded in 1.4 seconds with 123456 nodes and 786889 edges.
A pool of 8 threads has been created
Thread #5: waiting for connection
Thread #4: waiting for connection
Thread #6: waiting for connection
Thread #7: waiting for connection
Thread #3: waiting for connection
Thread #0: waiting for connection
Thread #2: waiting for connection
Thread #1: waiting for connection
A connection has been delegated to thread id #5, system load 12.5%
Thread #5: searching database for a path from node 657 to node 9879
Thread #5: no path in database, calculating 657->9879
Thread #5: path calculated: 657->786->87->234->9879
Thread #5: responding to client and adding path to database
...
Thread #4: path not possible from node 5243 to 98
…
Thread #6: path found in database: 445->456->75->6787->7897
…
System load 75%, pool extended to 10 threads
…
No thread is available! Waiting for one.
…
Termination signal received, waiting for ongoing threads to complete.
All threads have terminated, server shutting down.
```

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.

*****Client*****

The client is simple when compared to the server. Given the server's address and port number, it will simply request a path from node i1 to i2 and wait for a response, while printing its output on the terminal.

```
./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

Example output for the client (**include a timestamp at the beginning of every line**):

```
Client (4567) connecting to 127.0.0.1:45647
Client (4567) connected and requesting a path from node 768 to 979
Server's response to (4567): 768->2536->34783->12->979, arrived in 0.3seconds.

or

Client (4569) connected and requesting a path from node 578 to 87234
Server's response (4569): NO PATH, arrived in 0.19seconds, shutting down
```

where 4567, 4569 are the client process PIDs

Report: elaborate in your report which synchronization issues you encountered and how you solved them.

Rules:

- Implement your own data structures.
- Don't use sleep, goto, non blocking operations, timed waits or busy waiting of any kind
- No late submissions for any reason.

What to submit:

- source files, makefile and your report.

Evaluation: your program will be tested with multiple graph files (small, large, complicated, simple, etc).

- Compilation error -99
- Warnings, -1 per warning, -20 if more than 10
- No makefile -20
- No (pdf) report -20, insufficient report -19
- Well prepared report, in latex +10 (include tex source along with pdf)
- Not testing for missing parameters, not testing for wrong/invalid or unacceptable parameters, or not printing usage information: -10 if any of the above takes place
- Memory leak: -30
- Violating the project rules: -100
- Plagiarism (from online sources or other students): failing the course and disciplinary action for all involved parties.
- If your program fails with **any** of the test input files, it will considered as unsuccessful and will be penalized by -80;
  failing with input file = calculating wrong paths, not implementing the dynamic pool, failing to exit gracefully, crashing, freezing, not using the cache data structure.

Good luck.