CSE 344 FINAL PROJECT REPORT

Furkan Kalabalık 161044001

I. Introduction

In this project we're asked to implement a server-client system that has thread pool structure for server-side that responds path in a graph if it is possible. Thread pool must be dynamic i.e it is expand when server load exceed some value of threshold(for project 75%). This resizing must not make busy server, therefore resizing must be a seperate thread. Also since searching a path takes much time, to reduce time of respond, a data structure must be created, that access and search is fast enough to respond as soon as possible. Server-side must be daemon process, this means it is a foreground process, that has no controlling terminal and no two server can run on same computer. Server process can be killed by *SIGINT* signal and this killing process must be graceful, There shouldn't be a memory leak.

Client-side is another process that request a path from given source node to destination node. Source and destination nodes must be unsigned integers. Client must connect to server by providing its ip and port number and then must send request. At the end server process request and respond with given path.

II. Server-Side

In server-side firstly I take arguments:

- -i: Input file path for graph. Graph readed from given this path.
- -p: Port number that server will serve to its clients.
- -o: This is the log file path. When server is become daemon, there will be no controlling terminal. To take output of server, a log file must be opened and all outputs must be write there.
- -s: This is the thread pool's initial pool size.
- -x: Since system has dynamic thread pool structure, when load exceed 75% new threads are created but this value bounds server's expand size.

After that I checked arguments, I tried to open input file and log file. If there is problem with files, I informed the user before creating daemon and detaching from terminal. If there is no problem with files, forking occurs and we exit parent process. By doing that we create daemon process that adopted by init process. After that I created a file under */tmp* folder called *server.pid*, that hold pid of server process. This is required for no two daemon process run concurrently at the same host. The reason of choosing */tmp* folder is, it is a common folder in linux systems and secondly we need a common place for put this file, since a new terminal tries to create second server it sees that there is already a *server.pid* file, so there is a process, then process print there is already a running server message. From there I setted session id and closed all (STDOUT, STDIN and STDERR).

Then signal mask is setted to prevent *SIGINT* until a some point of server. After blocking *SIGINT*, a signal handler *termHandler* setted to handle signal. After that a cleaner function, that called when process exit and clean all allocated resources and destroys all condition variables and mutexes.

After that I started to reading graph from file. To understand file properly I read two times. First one is needed to read comment line that gives how many nodes and edges in graph. Also it is needed to learn max label number. In graph there is missing nodes so we can't depend on comment line. After I learned max numbered label, I created a two dimensional integer array. This is adjacency matrix. In second read, I filled array with 1's for given edges.

Then I created socket with *socket()* syscall. Before creating socket I filled its sockaddr structure. Since we deal with TCP connection I filled family as AF_INET. I setted port with given argument at start and address is INADDR_ANY since we deal with any server running on current system. Also in PDF it is given loopback address and it is a ip running on every current system. I binded server to socket and started to listening. I setted BACKLOG as max thread num, also this value is queue size for requests.

Then I created thread pool. Thread pool is a structure that has all condition variables, mutex and thread structures of system. Also it holds request queue for server. In thread pool create function I filled queue-specific values, created request queue and allocate place for threads. Also I initialize all condition variables and mutex. Then I created threads in usual way, using *workerThread* function and I give id as argument.

After creating threads, I created database for cache structure. For cache structure, I used hash table with bucket list. I created an array of linked lists that has node amount index. When a source node given to table, table goes to given source node index and take linked list inside that place. From there this means, this is the place where source node's cache structure hold. This head of linked list, where searching will start. Every linked list entry includes a destination node that is need to find paths in cache and path to this destination node and there is a pointer that points next database entry. Searching is more efficient that a regular array or list since we know where we must start. But as linked list expands, it will take time, definitely. When a searched path doesn't found at cache, it must need to calculate explicitly and must be added to cache. This is easy for that data structure, we only need to take source index and go that index. After going index, we take head of linked list that resides inside array and

we explicitly point new entry's next entry as linked list head and we declare this new entry as head and put it to array. This can be said adding to database is $\mathcal{O}(1)$ and searching is $\mathcal{O}(n)$.

After database creating, I created a waiting mechanism for waiting main thread. The reason of that is, since not all threads are ready immediately, starting resizer thread causes creation of additional and redundant threads, even there is no load on server. So main thread will wait until ready thread number in thread pool structure is equal to starting thread number. This waiting done by a condition variable called *waitMain*. In worker thread function, there is a if statement that checks if ready thread number is equal to starting thread number or not. If it is equal, then current thread signals *waitMain* and main thread wake up. After that resizer thread can be created and since all structures is ready, signal mask can be canceled and system is ready to process or ready to shutdown.

In main server thread there is a while loop with condition quitFlag that returns status of quitFlag. If this flag setted, loop simply exits and if not it continues. Main thread first calls *accept()* syscall to accept connection request. After connection accepted, server checks if we reached max number of threads, if we reached, it simply look to ready thread number, if it is 0, this means there is no ready thread, and no new thread will created. So this request and whole main thread must wait. This waiting mechanism done by a condition variable called *waitMax*. Until a thread become ready, main thread will stop. When a worker thread become available, it checks if it is the first ready thread and if this is the case it signals *waitMax* and wake up main thread. After that connection socket file descriptor added to queue that resides inside thread pool structure, a simple integer array. Then dispatcher thread (i.e Main thread) signals *cond* condition variable. All threads waits this condition variable when there is no element in queue. This signal mechanism wake up one of threads in thread pool and any of waiting worker thread wakes up. After waking up, worker thread takes first element and moves head of queue one more. By that mechanism, workers waked up and requests processed.

In worker threads, there is a loop that checks value of quitFlag, to check exiting is occur or not. After that every thread take lock using *mutex*, increase ready thread number by one. Worker thread check if there is a pending request at the queue. If this is the case, thread will simply take connection file descriptor and move queue head by one. Otherwise, worker thread must wait and this waiting mechanism done by condition variable *cond*. As we said above, when a request putted on queue, main thread wakes up one of the waiting worker threads and worker that waiting exits its waiting loop. Worker decrease pending request queue and takes connection file descriptor. After taking descriptor first it reads 2 integer, that is send by client. It reads these two integer into a array. Then it checks, nodes if they are represented in graph, if it not presented in graph, thread write nodes not in graph message in log file. Then it sends path as a single value -1 to indicate no path possible. But first it sends 1 as a path size to allow client to create appropriate array for ongoing path.

If this is not the case, we can proceed to normal process. First we need to check if given source and destination is presented in database structure. For that we need to use reader-writer paradigm. Checking database means we will read database but writers have priority, so we need check any writer that is waiting or accessed to database. If this is the case, then as a reader we must wait until database become available to readers. For that I used condition variable called *readCond* that waits writers. Also for mutual exclusion of database I used a mutex called *dbMutex*. When writers done with database, they signals to this condition and readers can access to database. After reader accessed to database and looked to the path if it is cache, it exits database and if it is the last reader, it is inform waiting writers to access to database.

If result of database lookup is *NULL*, then this means we must perform Breath First Search and find path explicitly. BFS implementation is same as normal BFS implementation but when destination node encountered, the parent array that is constructed throughout search phase used to back tracing and path is constructed. After finding path explicitly, writing must done. For that we act as writer and we have priority of writing. We first waiting for if there is any active reader or writer, if this is the case, we simply wait by using condition variable *writeCond*. When one of readers is done or writer that accesed is done with database, it signals waiting writer and then writer access to database. After putting new entry to database, writer looks if there is a waiting writer, if this is the case it wakes up one of writer, otherwise it wakes up all readers. Then path is sent to client in this manner: First size of path array sent to client. Then client allocates space for that much size. After that server sends whole path and client reads that path socket. If there is no path server send a path that has only one integer -1. Then server closes connection socket and try to wait for next requests.

A. Resizer Thread

Resizer thread used to extend thread number in pool to decrease load of server. For that purpose, resizer thread is again in a while loop that checks quitFlag. Then resizer thread locks access with *mutex*. After locking resizer look load of server by calculating it from ready process number and if max number of thread is equal to total thread number. If these are the case, then resizer thread is wait with using condition variable *resizerCond*. When one of the worker thread is become busy and process a request, it check if the load is exceed 75% or not. If load exceed 75% then current worker thread signal *resizerCond* and wakes up resizer thread. After that resizer wake up and and exit waiting loop. After that from total number of thread, newly created number number calculated by multiplying it with 1/4. If this new thread number plus total thread number is exceed maximum number of thread, the a loop decrease by one until value equal to maximum number of thread. Then thread array inside thread pool struct is reallocated with

its new size. Then threads created. After that there is two variable called *willCreate* and *update*. These values here for bookkeeping. *willCreate* indicates number of threads will be created for that session and *updated* indicates updating of threads is ongoing. After that resizer thread is waiting with same condition variable *resizerCond*, to new threads become ready. Reason of this, even we create new threads, these threads don't become ready immediately and resizer works again since load is still greater or equal than 75%. By that resizer thread will wait until all newly created threads become ready. In every worker thread, there is if statements. One checks if *willCreate* equal to the created thread number and *update* is equal to 1. If this is the case, we can wake up resizer and end this creating session. If this not the case then we simply increase number of created thread.

B. Termination

Server is terminate when received *SIGINT*. First *SIGINT* is blocked, until all system is ready. While system is booting, we register a *SIGINT* handler to handle signal. This handler is very simple. First it close socket since main server can be blocked at *accept()* syscall. If this is the case error value is tolerable. After that handler set a flag and this flag indicate quitSignal received. Loops of threads check this quit flag and if quit is setted they are exit its loops. Also I registered *atexit* function called *cleaner* that cleans all allocated resources and destroys all condition variables and mutexes.

When signal arrived, main thread exit its loop and don't accept connections anymore. After that main thread waits all requests to process. When all requests processed, one of the workers wakeup main thread and main thread broadcast all workers that are stuck at waiting stage and also send signal to resizerThread if it is stuck at waiting stage. after that it simply wait other threads to return by joining them. Then exits. When exit called, cleaner function runs and all resources freed.

In threads, worker or resizer, there is a condition after condition variable wait. This is needed since even we put flag check at the while loop condition, threads may stuck at the waiting stage. By putting these conditions, when we wake up them, they check if quit is setted. If this is the case then we can exit threads by unlock mutexes and free resources.

III. CLIENT

Client is simply request a path between two nodes given as argument from given ip and port number. Client firstly check nodes is greater or equal than zero since nodes must be unsigned integers. Then creates socket as TCP socket. After that it fill sockaddr structure. Put port number to port field and put given id to address field using inet_addr() function. Then tries to connect. If it is connect, it send request to server while writing nodes array to connection. After that tries to read, one single integer that indicates size of path. Until server send respond to client, client is blocked. When server sends size of array, client takes this size value and creates an array of integer for path, then tries to read "size" much integer from server and when it is read, print the path calculated by server. If no path is possible, than server sends -1 and client prints No PATH!. After that client closes socket connection and exits by freeing its resources.