ASSIGNMENT 2: A Key-value Store Server

SHUAI KE - 1609628

April 27, 2018

1 Implementation

1.1 Overall

Firstly, the main thread does the initialization. Then, it polls the two sockets until the server is closed. For a new data request, the main thread stores the file descriptor into the queue and broadcasts all worker threads, while for a new control request, the program will handle it immediately. When the server receives the shutdown command, the main thread will set the shutdown flag to true, broadcast all workers and wait they finished their work. After that, the main thread deallocates the memory and releases the semaphores.

The thread pool consists four worker threads, each thread has a loop for serving the data request. When there is a new data request, the program ensures that only one worker thread gets the file descriptor from the queue and this thread will handle the request until user exit.

1.2 Global variables

There are a queue and five semaphores [lines 27-30].

- Que: store the file descriptor of data requests.
- s_shutdown: indicate the server is closed or not.
- s_work_avail: the number of available request in queue.
- s_space_avail: the number of available space in queue.
- s_data_lock: prevent race conditions in the kv-store.
- s_queue_lock: prevent race conditions in the queue.

1.3 Threads and Functions

The key-value store server has two kinds of threads, main thread and worker thread. The program accepts two kinds of requests, data and control, it handles them by the function *data_work* and *control_work* respectively.

The **main thread** checks the arguments is enough or not firstly [lines 789-792] and calls the *initialise* function to initialise the global variables, including five semaphores and the queue[lines 795, 506-576]. Then, it will create two sockets, bind them to the control and the data port respectively [lines 798-803, 462-499], and create the worker threads to handle the data requests [lines 806-813]. After that, the main thread starts to listen to the two ports and waits for the new requests until the server is closed[lines 816, 662-780]:

- Poll the two sockets to wait for the new requests [lines 674-678, 684].
- For the data request, the main thread accepts it and stores the file descriptor into the queue when the queue is not full, then decrease 1 of available space and increase 1 of available work of queue [lines 710-778].
- For the control request, the main thread accepts it and calls *control_work* function to handle immediately. It will check the server should close or not after handle each control request [lines 687-707, 234-343]

Finally, it closes the sockets and releases the global variables [lines 819-833, 583-654].

For each **worker thread**, there is a loop for handling the data request until the server is closed [line 364]. In each iteration, the thread will block when there is no available work [line 367], otherwise, it gets the file descriptor of the data request from queue[line 381], then decreases 1 of available work and increase 1 of available space of queue. Before handling the data request, the worker thread will check the s_shutdown flag, if the server is closed, it will exit directly [lines 405-426]. Furthermore, when there is no new data request and the server is shutting down, the thread will still check the s_shutdown signal even it cannot pop a file descriptor from queue [lines 435-449].

The **control_work** and **data_work** function have a loop for handling the commands, and they will read the command from and write the response to the file descriptor.

1.4 Error checking and Debugging

Error checking: Server checks the operation in heap memory, semaphore, thread, IO, kv-store and queue. The invalid data commands also be handled.

Debugging: When the program catches an error, it uses the *perror* function to print the information in stderr for tracing the bug.

2 Correctness

- Each thread needs to acquire the data and queue lockers before operating the kv-store and the queue, it will release the lockers after the operation, which ensures only one thread operates the kv-store or queue at the same time and no more than one thread has the same file descriptor getting from the queue.
- The main thread waits for all worker threads to finish their work when it receives a shutdown command.
- The server will exit for a system call error, but it not aborts for any user commands.
- There is no deadlock, two semaphores work cooperation to ensure the file descriptor pass from the main thread to worker thread by the queue. And when the server should close, the main thread will wake up all worker threads.

2.1 Testing and Bug Fixes

Most tests are correct, but some problems appear in the testing section. There are the bugs and solutions as follows:

- **Blocking problem**: NTHREAD=1, QUEUE_SIZE=1, three data requests. In this case, one request is han-

dled by the worker thread and one request is stored in the queue, but the remaining one will block the main thread since it waits for the semaphore s_avail_space and the server cannot accept any request, including data request, during this time. The solution is to set the initial value of *s_space_avail* to QUEUE_SIZE + 1, the new data request will get a warning message, exit directly and release a space avail when the queue is full [lines 539, 740-770].

- **Exiting problem**: NTHREAD=1, QUEUE_SIZE=2, three data requests and one control request. In this case, one data request is handled by the worker thread, two data requests are stored in the queue. During this time, send a shutdown command by the control request and exit the first data request. The worker thread finishes the first data request and detects the server should close before it handles the second data request, so it sends a warning message to the second user, but for the third user, the link just close, there is no any warning message and the file descriptor also not close. The solution is the worker thread will not exit directly when it detects the server should close unless it does not get any data request [line 425].

3 Introduction for extension

The extension part implements the skip list [1] in *skiplist.h* to fix the memory problems in *kv.h* and improve the complexity in operating key-value store from $o(n)$ to $o(logn)$.

The implementation of kv-store in *kv.h* has some tiny memory problems, **memory leak**. There are three points which are fixed in *skiplist.c* as follows:

- In the update function (updateItem), it overrides the pointer of the value and does not release the memory of the previous value [line 64 in kv.c].
- In the delete function (deleteItem), it releases the memory of the value [line 73 in kv.c], but it does not release the memory of the key.
- There is no function for release the memory of the whole kv-store after the server is closed.

Skip list [1] provides an efficient strategy for the key-value store. The complexity of the operation in the skip list, like search, insertion, and deletion, is $o(logn)$.

4 Implementation for extension

4.1 Overall

The extensional code is using the APIs in *skiplist.h* to operate the kv-store, and the memory management for the key and value is doing in the skip list, which replaces doing in the out of kv-store. The memory of the whole kv-store also is deallocated [lines 617-634].

4.2 Global variables

There is a added global variable Items which is the skip list instance for key-value store.

4.3 The structure of skip list

A skip list is a variant linked list which the node has different levels. The bottom layer is an ordinary ordered linked list. Furthermore, the level of each node is depending on the *random_level* function [lines 22-28 in skiplist.c] during the insertion. The *Figure 1* shows the structure of skip list:

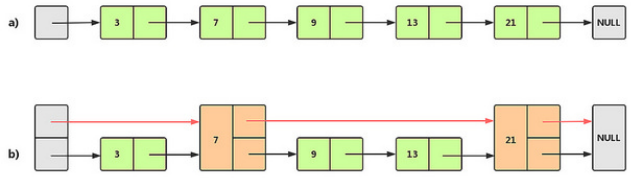

Figure 1: The Structure of Skip List

$a)$ is an ordinary ordered linked list and $b)$ shows the skip list with maximum level is 2. The nodes of level 2 form a new linked list, which as the quick path. When we need to search $Key = 13$, it searches the nodes of level 2 firstly to determine the scope, then finds it in the level 1. The *Figure 2* shows the procedure.

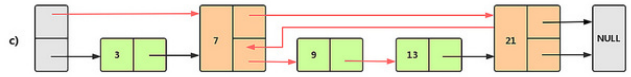

Figure 2: The search path

4.4 APIs

The implementation of the key APIs in the skip list is shown as follows (the code is in skiplist.c):

- init_skiplist: initialize the skip list instance, the return value is a pointer of skip list or NULL [lines 30-43].
- search_skiplist: search for the value stored under the key [lines 46-68].
- insert_skiplist: insert a new item under the given key, the memory for storing the key and value are allocated in this function [lines 94-139].
- update_skiplist: update a new item under the given key, the old value will be overridden by the new value [lines 142-168].
- delete_skiplist: delete an item, including to free memory of the key and value [lines 171-203].
- free_skiplist: free the memory of the whole skip list instance [lines 212-225].

5 The reasons for using skip list

The skip list is a realizable and efficient data structure. The average complexity is $o(logn)$, the worst situation is $o(n)$ [1]. Redis, an open source, in-memory data structure store, used as a database, cache and message broker, which also use the skip list to implement the sorted set.

6 References

1. W. Pugh. Skip lists: a probabilistic alternative to balanced trees. In *WADS*, 1989.