Operating Systems
Spring 2018

Operating Systems Project

Professor : Philippe Cudre-Mauroux
Assistant : Ines Arous

Submitted by Groupe 4: Sylvain Julmy, Michael Papinutto, Sami Veillard

---

May 26, 2018

Introduction

For this project, we implemented a multi-threaded client-server system using TCP sockets.
One of the problem in designing such system is related to the database structure. Indeed, we can increase memory required but not time as this would impact User Experience. Hence, we tended to choose faster algorithm. Indeed, looking at applications usage revealed that sets are used in the following way: 90% of calls are contains(), 9% are add() and 1% are remove(). In this project, we selected a hash set for those reasons. Even though this system might require more memory resources and was rather complicated to implement, it was found to be efficient and powerful according to our test. Finally, our system has ability to write keys with values, read values providing a key, simultaneous safe access of the readers and the writers. This system was subsequently tested using an automated test in form of a bash script and text files of commands. This report mainly focus in explaining our approach as it was not part of the course and might be a strength of our project.

Chosen approach

We had to keep in mind various aspect when choosing the data structure to store the key-value entry : maximal number of entry stored in the database, extension of the data structure, number of simultaneous access on the server and how to synchronize the threads.
A lock-free hash-set data structure offers solutions to all of those challenges without blocking threads using mutex nor semaphores. We have implements our own data structure in C based on the one created by Herlihy in [Her06] in Java. Indeed, by using this implementation there is no need to prioritize neither read nor write action.
Such implementation to be non-blocking requires atomic operations. Atomic operations are unique or multiple operations that cannot be stopped before their ends. In C, atomic operations such as CompareAndSet can be done by "stealing" a bit from a pointer using bit-wise operators to extract the pointer and the mark from a single word [Her06]. More Specifically, we used a reversed split-ordering hash set. A bucket is linked to a stack and as the list grow supplemental buckets references are added so that no object is ever too far from the start of a bucket, *i.e.*, the bucket size is kept small. This implementation ensures that when an item is put in the stack, it will not be moved. To do so, items have to be put in the stack using a recursive-split order. Moreover, to avoid problems occurring when deleting a node referenced by a bucket, a sentinel node is added at the beginning of each bucket. This special type of buckets are never deleted and can be identified by the Most Sensibel Bit which is set to 0 whereas for usual bucket is set to 1(see Figure 1).

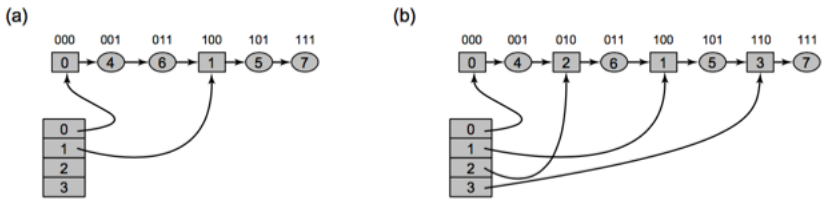

Figure 1: Scheme of the recursive nature of the split ordering. The split order can be seen in binary words above buckets. Sentinel buckets are represented by square whereas ovals are normal buckets. (a) Shows a split-ordering including two buckets. The buckets are linked to a stack. (b) Shows how buckets are split in two part after the capacity of the table grows from 2 to 4.

When inserting a new key in this data structure, the table is grown incrementally. As buckets

are in a linked list ordered using split-ordering, the table resizing mechanism is independent to the threshold that decides when to resize. As the sets grows, most of the array will be use. Hence, when adding new values to a not yet initialized bucket that should have beeen initialized according to the current table capacity, this bucket will be initialized without generating an error. (see Figure 2).

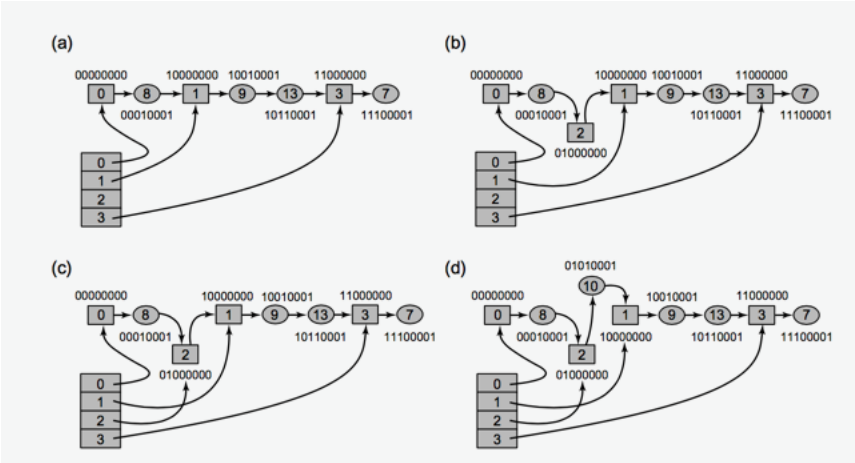

Figure 2: Scheme of the procedure that add the key 10 to the lock-free hash set. As above, split-order key values are expressed in binary above buckets (here 8 bits words). (a) Buckets 0, 1 and 3 but bucket 2 is not. (b) An object with hash value of 10 is inserted to the set. This cause bucket 2 to be initialized and a new sentinel is inserted with split-order key 2. (c) Bucket 2 is affiliated with a new sentinel. (d) Finally, the split-order ordinary key 10 is added to bucket 2.

In summary, our chosen approach implements a split-ordered hash set which is an array of buckets where each bucket is a reference to a lock-free list where nodes are sorted by their bit-reversed hash codes. The number of buckets grows dynamically, and each new bucket is initialized when accessed for the first time.

Challenges encountered

One of the challenge we encountered was to read or deleted entries from values. Indeed, looking for a value in a hash-table is not as straight forward as it seems. It requires navigating through all the keys until the corresponding value is found. Despite the high cost of this operation, we decided to use this method as we did not find any other way to provide such an operation.
Another challenge that we encountered was that we noticed using the implementation described above was not thread friendly. This problem was in fact due to the string tokenizer for commands was kept within an internal static variable. This issue was subsequently solved by reprogramming the tokenizer and further tested to ensure that thread kept their own tokenized command. This issue took us a lot of time and debugging to be solved.
Finally, as only one of us was familiar with non-blocking operation and well versed in the intricacies of C programming, the other two had to keep up and learn a new way to program and think as this material was not presented during the class.

Tests Results

We set up 3 test scenarii to evaluate our project: Collisions, No-Collisions ans Many Clients. In the Collisions scenario, we tested 11 clients and 28 commands (308 operations in total) using

every operations available. In the No-Collisions scenario we tested 8 clients and 2700 commands (21600 operations in total). In the Many-Clients we tested 32 clients and 300 commands (9600 operations in total). In the No-Collisions and the Many-Clients scenarii we only used add, read value from key and delete key from value.

Collision scenario			
	Add	Read	Delete
Number of errors	0	22	0
Percentage of errors	0%	7.14%	0%

No-Collision scenario			
	Add	Read	Delete
Number of errors	0	0	0
Percentage of errors	0%	0%	0%

Many-Clients scenario			
	Add	Read	Delete
Number of errors	0	0	0
Percentage of errors	0%	0%	0%

Interestingly, the Many-Clients scenario required more time than the previous one despite it has half less operations. This is probably due to the synchronisation of clients requests.

Conclusion

In conclusion, we addressed the challenges proposed in this project using a reversed split-ordering block-free hash-set data structure which offered us a way to cope with the prioritization of read and write operation. Moreover, as the usual operations use in a set was found to be mainly compose of contains and add, this implementation offered a fair trade off between between response time and memory pressure. However, such implementation does not prevent collisions but rather exhibits collision when entries were already deleted in the data structure also occuring in other implementations.

Bibliography

[Her06] Maurice Herlihy. "The art of multiprocessor programming". In: *Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing - PODC '06*. 2006. ISBN: 1595933840. DOI: 10.1145/1146381.1146382.

Documentation

Communication

The server and the clients communicate through socket.

Server

The server is composed of several binary files. The main file encompasses the socket setup for the server and dedicated files for communication and the shell graphical user interface. The server is multi-threaded, *i.e.*, after each connection of a client the server create a new thread. The data structure is described above.

Server usage

TCP Port	5000 (can be reset in the server main file)
.\server	server start

Client

The client is also composed of several binary files. The main file sets up client socket and dedicated files are used for execution of command and shell graphical interface. On the contrary of the server shell, the client shell is an interactive shell the usage thereof is described below. Moreover, to simplify benchmarking, the client can also accept files at launch.

Client usage

Client basic usage

.\client <server ip address>	client start
.\client -option <server ip address>	client start with options (see below)

Options at start

-? -h --help	client command help
-f <file> --file <file>	client start and execute command present in the file specified after this option
-F <file1> ... <fileN> --files <file1> ... <fileN>	client start and execute command present in the files specified after this option

Client command accepted in interactive GUI

add <value> or add <key> <value>	adds a value to the database with or without a generated key
ls	lists the content of the database (unordered)
read_v <key>	reads a value in the database from a key
read_k <value>	reads a key in the database from a value
rm_v <key>	deletes a value in the database from a key
rm_k <value>	deletes a value in the database from a key
update_kv <value> <newvalue>	updates an entry in the database