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 had to implement a multi-threaded client-server system using TCP sockets. This system had abilities 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 testing in form of a bash script and text files of commands. 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 tend 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 ressources and was rather complicated to implement, it was found to be efficient and powerful according to our test. 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 take care of 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, there is no need to prioritize neither read nor write action.
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 keep small. This implementation ensures that when an item is put in the stack, it will not be move. However, to do so, items have to be put in the stack using a recursive-split order. Moreover, to avoid problems occuring when deleting node referenced by a bucket, a sentinel node is added at the beginning of each bucket. This special type of buckets are never deleted (see 1).

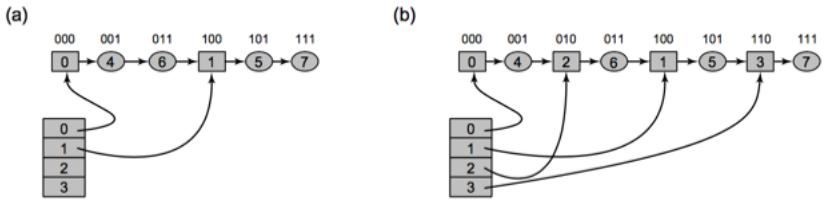


Figure 1: Scheme of the recursive nature of the split ordering. The split order can be seen in binary words above buckets. Sentinel buckets are represented by square whereas ovals are normal buckets. (a) Shows a split-ordering including two buckets. The buckets are linked to a stack. (b) Shows how buckets 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 independant to the threshold that decide when resizing. As the sets grows more of the array will be use. Hence, when adding new values to a not yet initialized bucket that should have be initialized according to the current table capacity, it will be initialized (see 2).
In summary, our chosen approach implements a split-ordered hash set which is an array of

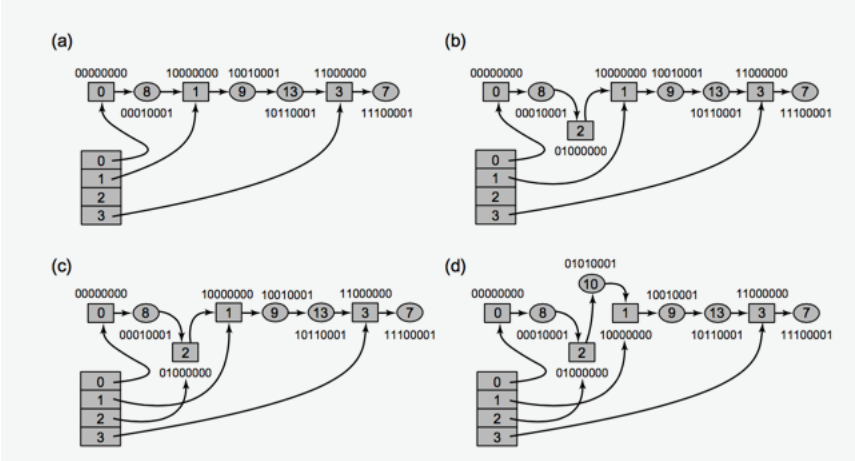


Figure 2: Scheme of the procedure that add the key 10 to the lock-free hash set. As above, split-order key values are expressed in binary above buckets (here 8 bits words). (a) Buckets 0, 1 and 3 but bucket 2 is not. (b) An object with hash value of 10 is inserted to the set. This cause bucket 2 to be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

buckets where each bucket is a reference into 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 to. It required to navigate through all the keys until finding the corresponding value. Despite the high cost of this operation, we decided to use this method as we did not find any other way top 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 fact that tokenizing commands string was keeping an internal static variable. This issue took us a lot of time and rethinking to be solved.
Finally, as only one of us was familiar with non-blocking operation and well experimented in c programming, the two others had to keep up and learn a new way to program and think as this material was not presented during the class.

## Conclusion

In conclusion, we addressed the challenges proposed in this project using a lock-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 do not prevent collisions but rather exhibits collision when entries were already deleted in the data structure which also occurs 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 from several binary files. The main file set 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 which usage is described below. Moreover, to simplify benchmark 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**

| | |
|---|---|
| -?<br>-h<br>--help | client command help |
| -f <file><br>--file <file> | client start and execute command present in the file specified after this option |
| -F <file1> ... <fileN><br>--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> | add a value to the database with or without generated key |
| ls | list the content of the database (unordered) |
| read_v <key> | read a value in the database from a key |
| read_k <value> | read a key in the database from a value |
| rm_v <key> | delete a value in the database from a key |
| rm_k <value> | delete a value in the database from a key |
| update_kv <value> <newvalue> | update an entry in the database |