Assignment 4: Key-value store

In this assignment you will implement your own **remote in-memory key-value store**. It must support multiple concurrent clients accessing and modifyng the data store. Similar key-value stores, such as Redis and Memcached, are very popular and often used for caching. During this assignment you will learn about writing a server application, using multiple threads to support parallel clients, and how to protect data structures for concurrent accesses.

A key-value store allows users to store and retrieve data from a server over the network. The data (value) is addressed by a string (the key). Conceptually they are very simple, and only have 3 operations: SET, GET, and DEL. A SET inserts or modifies data into the store, whereas GET retrieves previously stored data. Finally, DEL simply removes a stored entry. For a detailed description of these commands, and the exact protocol that is used, see section Protocol specification.

High-level overview

A client application interacts with the key-value store via a TCP server. The client and the server establish a connection using sockets and they exchange messages. The client sends a message to the server via its socket in order to modify an item from the store. The server parses the message, handle it, and replies with a message containing a status code. For example, the client's requet may succeeed or fail.

Besides the *check.py* script you are already familiar with, in this assignment we also provide a client *test_client.py* that you can use to interact with the server by sending request one at a time. In section Testing and debugging we describe how to use this script.

The server stores items in a data structure as described in Hashtable.

Getting started

You should add your code to *kvstore.c* and *kvstore.h*, already included in the framework. This file already contains some boilerplate code to get you started. In particular, it already sets up the server socket, and contains some (single-threaded) code to accept connections and requests. You should not modify other existing files. The key-value store must be implemented by means of a hashable.

Hashtable

Due to some differences in terminology, we here summarize what a hashtable is. A hashtable is a data structure with a fixed amount of entries called *buckets*. An *hash* function is applied to the key-value pair key. Each bucket contains entries with the same hash.

The hashtable struct is already defined in *hash.h*.

```
typedef struct hash_item_t {
    struct hash_item_t *next, *prev;    // Next and previous item in the bucket
    char *key;                          // Item's key
    char *value;                        // Item's value
    size_t value_size;                  // Item's value length
    struct user_item *user;
} hash_item_t;

typedef struct {
    unsigned int capacity; // Number of buckets
    hash_item_t **items;   // Buckets
    struct user_ht *user;
} hashtable_t;
```

`hashtable_t` is your hashtable structure.

`capacity` specifies how many buckets the hashtable contains. **The hashtable must be initialized with 256 buckets** (i.e., *capacity* must be 256).

`items` is a dynamic list of pointers. These pointers point to elements of type `hash_item_t`. At index `i` of `items` (`items[i]`) you store the `ith` *bucket*. The bucket contains several `hash_item_t` elements, as a double linked list (see `next` and `prev` field).

The the diagram below depict the hashtable structure. To determine the bucket of a specific key, use `hash(key) % capacity` (where the *hash* function is provided in *hash.c*). New items should be inserted at the **head** of the list. Each item contains a null-terminated *key*. The key has a *value* string of *value_size* chars associated with it.

Your first task it to initialize an empty hashtable. Once you have a functional hashtable, you should extend these structures to include any additional data required for locking/synchronization code. Indeed, during the assignment you need to ensure the hashtable can be accessed and modified in a concurrent manner.

Both `hashable_t` and `hash_item_t` contains an **user** field. Both are defined in *kvstore.h* but their implementation is empty. *You can add other fields to the data structure!*

The `user` field inside `hash_item_t` can be used to access user defined variables per item (`user_item`). The user field inside `hashable_t` can be used to access user defined variables inside the hashtable of type `struct user_ht`.

You can add fields into `user_item` and `user_ht` to keep your locks, mutex, synchronization variables etc.

```
              hashtable_t               hash_item_t               hash_item_t
              items                   key   ...                 key   ...
              +-----+                +-----+-----+  --next-->  +-----+-----+
hash(caw) --> |  0  | --items[0]-->  | bsf | ... |             | caw | ... |
              +-----+                +-----+-----+  <--prev--  +-----+-----+
              |  1  |
              +-----+
              |  2  |
              +-----+       +-----+-----+
hash(waf) --> |  3  |  -->  | waf | ... |
              +-----+       +-----+-----+
              |  4  |
              +-----+       +-----+-----+  -->  +-----+-----+  -->
hash(hoi) --> |  5  |  -->  | hoi | ... |       | sal | ... |       ...
              +-----+       +-----+-----+  <--  +-----+-----+  <--
              |  6  |
              +-----+
              |  .  |
              |  .  |
              |  .  |
              |     |
```

Next steps

You should then implement a (single-threaded) version of the SET request. You have to use the provided data structures, as the framework uses these internally to check your results. Once your single-threaded SET works, you can also implement single-threadedversions of GET and DEL, or move on to implement concurrency.

To implement concurrency, first start by creating separate threads for each connection. Use the `pthread_create` function to create threads.

After your server supports multiple concurrent connections, you should modify your hashtable data structure to be thread-safe. In particular, you should any required locks (see `pthread_mutex`) to the provided structs. Ensure that concurrent operations are allowed on different items (even in the same bucket), but not on the same item. Make sure to also protect modifications of the hashtable and bucket linked-list.

NOTE For this assignment you only can add all your code to kvstore.c and kvstore.h. **You are not allowed to modify any of the other framework files.** You can add your additional sources. Do not forget to add them in the Makefile under:

```
# Add additional .c files here if you added any yourself.
ADDITIONAL_SOURCES =

# Add additional .h files here if you added any yourself.
ADDITIONAL_HEADERS =
```

Protocol specification

Requests

Our key-value store uses a simple text-based protocol. After opening a connection, a user can issue any of the following commands. Once a command is completed, another command can be issued over the same connection. All commands and responses are terminated by a newline (`\n`).

Keys can only consist of printable ascii characters except for whitespace. Values can contain any character (including `\n` and `\0`), and must therefore always be accompanied by a size field.

The general format of every command is as follows:

```
<command> [<key>] [<payload_len>]\n
[<payload>\n]
```

Where <command> is one of SET, GET, DEL or RESET. Terms in brackets are optional depending on the command. When <payload_len> is omitted or 0, <payload> *and* its accompanying `\n` are not sent.

SET

```
SET <key> <payload_len>\n
<payload>\n
```

Description Inserts a new key with a provided value, or overwrites the value of an existing key-value pair. Until the server has received the full payload and written it as value, the value must remain locked, and any other operation on this key should return an error. **Return codes**: *OK* If the data was successfully and completely inserted. *KEY_ERROR* If the key could not be inserted because it is already locked by an operation of another user. *STORE_ERROR* If data could not be stored, e.g., when no memory could be allocated for it.

Note: the server must consume the whole payload even if an error is detected, to prevent the connection from desyncing.

GET

```
GET <key>\n
```

Description Retrieves the value of a previously inserted key. Until the entire value has been sent successfully, the key must be locked for write operations (SET and DEL). Concurrent reads (GET) are allowed.

Return codes: *OK* Followed by the value of <payload_len> bytes. *KEY_ERROR* If the key does not exist, or if the key is locked by another operation.

DEL

```
DEL <key>\n
```

Description Removes a key-value pair from the store.

Return codes: *OK* If the key was successfully deleted. *KEY_ERROR* If the key does not exist, or if the key is locked by another operation.

STAT

```
STAT <stat>\n
```

Description Retrieve general statistics. Possible values of <stat> are:

- REQ: number of requests served at that time
- THG: server throughput (i.e., number of requests server per seconds)

Return codes: *OK* Followed by an integer representing the value of the <stat> statistic.

RESET

```
RESET\n
```

Description Reset the entire store, so that no key-value pairs remain.

Return codes: *OK* On successful clearing of all items.

Note: supporting this command is optional, but may be required for using some of the debugging modes mentioned later in this document.

Responses

For every command the server will respond with a response in the following format:

```
<status> <code> <payload_len>\n
[<payload>\n]
```

If <payload_len> is 0, payload (and its accompanying `\n`) is omitted. Possible responses are:

status	code
0	OK
1	KEY_ERROR
2	PARSING_ERROR
3	STORE_ERROR

Note on PARSING_ERROR: Upon arrival of a SET request, you should read the data from the stream in any case (i.e., even if the key is already locked by another thread). Another request, which finds the stream into an invalid state, cannot be parsed and PARSING_ERROR is returned.

Framework

The provided framework aims to abstract away most code related to networking, such as sockets and the protocol.

This section provides a more detailed description about our framework. Not everything here will be directly useful to you. However the following description aims to provide you with a deeper knowledge about the underlying framework workings. For your assignment you are asked to implement the server main function and functions used to handle client requests.

- **common.h** defines some basic data structures such as *request_t*.

```c
struct request {
    enum method method;
    char *key;
    size_t key_len;
    size_t msg_len;
    int connection_close;
};
```

This request_t data structure describes an incoming request. The `method` field can contain one of the following operations: `
`enum method {UNK, SET, GET, DEL, PING, DUMP, RST, EXIT, SETOPT};` `

Note that you should only handle SET,GET,DEL and RST requests since the framework handles the remaining methods for internal operations and they are never visible to your code

The other fields are filled with the key, key length and, if applicable the payload length. You should properly handle the flag *connection_close*. You can read and write the flag to specify how to treat the connection at the end of the request. For example, during the request handling you may decide to set the flag *connection_close*. At the end of the request you should check its value and accordingly call *close_connection(int)*.

- **server_utils.h** contains interface functions to interact with the framework:

API	Description
allocate_request()	It allocate a *struct request* and return a pointer to it that can be passed to the functions below.
recv_request(int, struct request*)	Check if the socket in ready (see connection_ready) and read the command header from the socket. It populates the data structure pointed by *request*
accept_new_connection(int, struct conn_info*)	Accept a new incoming connection from the listening socket and return connection info in the *conn_info* struct
read_payload(int, struct request*, size_t, char*)	Read *expected_len* bytes from the socket and put the data into the buffer passed as argument
check_payload(int, struct request*, size_t)	Check if payload's length is *expected_len* and read the last byte from the socket (which should be 'n')
close_connection(int)	Close the connection with the client on the given socket
send_response(int, int,int,char*)	Send a response to the client.

- **hash.h** contains the hashtable definition.

Server design

In this assignment we ask for a thread-based server providing key-value store features. Duplicating the process by means of `fork()` is forbidden. You must use Pthreads APIs instead. A typical design includes the main thread (dispatcher or listener) and several other threads dubbed helpers.

In a relatively simple design threads are spawned on demand. For each new incoming connection (accepted by `accept_new_connection()`), you should carry out the following operations:

- Spawning of a new thread, the dispatcher.
- You should allocate a new request object (i.e.,`struct request`).
- You should populate the pre-allocted request by calling *recv_request*.
- In accordace with the *method* field you should handle the request.
- You may decide to set the request's field *connection_close* to 1. On return from the request handling routine you should read *connection_close* to check if you should close the connection (e.g, an error occured).

A big downside of this design is that on incoming connections the server adds overhead to create a new thread. A more efficient design instead creates threads once at startup, and uses a so-called **job-queue**.

- The server creates a pool of threads when it starts. The main thread is the dispatcher, while the others are called "helpers"
- The dispatcher listens for incoming connections and it passes connection info (i.e.,struct conn_info) to the helpers by a shared data structure such as a queue.
- The helpers retrieve connection infos from the queue and they handle incoming requests.

Mutual exclusion and synchronization is required when accessing the shared connection infos Queue in a Producer/consumer approach. You should handle the case in which the queue is empty:

You should use conditional variables for communication.

A simple approach is as the following:

- Upon a new incoming request the dispatcher inserts a new object into the queue.
- It notifies the workers that the queue is not empty by calling (*pthread_cond_signal* or *pthread_cond_broadcast*).
- The workers wait for the queue to not be empty (i.e., they should block on *pthread_cond_wait*) and when they wake up they pop an element from the shared queue.

Testing and debugging

Concurrency issues are notoriously hard to debug. The included tests try to detect any potential issues, and can be invoked via `make check` (or `make docker-check`). Additionally, you can manually test your code using the included *test_client.py*.

For debugging deadlocks, it may be useful to add prints around every lock and unlock operation, to try to determine where a program gets stuck. The framework provides functions to print output, including the thread-id, which are enabled via server flags: - `pr_info` prints if the `-v` (or `--verbose`) flag is passed to the server. - `pr_debug` prints if the `-d` (or `--debug`) flag is passed to the server.

The included tests will, by default, restart your server for every individual test. Additionally, any output produced by the server is not shown. To aid in testing and development, the tests support some additional flags. These flags should be passed directly to *check.py* (e.g., `python3 check.py -d --debug-print-server-out get del`).

Any arguments passed after the flags refer to test groups that should be run. By default, *check.py* runs all tests. The names of test groups are printed during execution (between parenthesis, in grey).

The `-d` flag will halt tests after the first error that occurs, and show a backtrace inside the python code. You may read through the particular test case in *check.py* to see what the test is expecting, and what could be going wrong.

The `--debug-print-server-out` tries to print any output produced by your server, after each test. Note that the server is not started with either `-d` or `-v` flag, so any `pr_info` or `pr_debug` prints are not enabled.

The `--debug-server-pid` is probably the most useful, as it allows you to run the server manually in a separate terminal while the tests run. The flag expect the PID of the server as argument. Make sure to implement the *RST* command when using this feature. To automatically pass the PID of a running server, use:

```
$ python3 check.py --debug-server-pid `pgrep kvstore`
```

For manual testing, you can issue your own commands to a running server using *test_client.py*. It can be used to issue a single command, such as:

```
$ python3 test_client.py set mykey somevalue
```

You can also issue multiple commands over the same connection in interactive mode:

```
$ python3 test_client.py --interactive
> set mykey somevalue
OK
> set
Key: otherkey
Value: foobar
OK
> get mykey
OK: somevalue
> get otherkey
OK: foobar
```

The assignment and grading

This assignment is individual; you are not allowed to work in teams. Submissions should be made to the submission system before the deadline. Multiple submissions are encouraged to evaluate your submission on our system. Our system may differ from your local system (e.g., compiler version); points are only given for features that work on our system.

Your grade will be 1 if you did not submit your work on time, has an invalid format, or has errors during compilation.

The following tests determine your tentative grade (in no particular order):

- +1.0pt if you made a valid submission that compiles.
- +0.5pt for implementing the SET command. **Required with at least 0.3pts**
- +0.5pt for implementing the GET command. **Required with at least 0.3pts**
- +0.5pt for implementing the DEL command. **Required with at least 0.3pts**
- +0.5pt for implementing basic parallelism (supporting concurrent connections via different threads). **Required**
- +1.0pt for supporting concurrent SET commands correctly (prevent concurrent accesses to the same key but allowing concurrent accesses to different keys).
- +1.0pt for supporting concurrent GET commands (e.g., allowing different GET requests to different keys, disallowing GET requests on keys that are being modified by SET).
- +2.0pt for implementing a readers-writer lock (i.e., supporting concurrent read accesses through GET to the same key, but only allowing a single writer via SET/DEL).
- +2.0pt for implementing a thread pool instead of spawning new threads for every connection. You should use conditional variables.
- +1.0pt for surviving the parallel stress tests for the various commands.
- +1.0pt for supporting the STAT method. You should use atomics to modify statistics.
- -1.0pt if `gcc -Wall -Wextra` reports warnings when compiling your code.

If you do not implement an item marked with **Required** you cannot obtain any further points.

The grade will be capped at 10, so you do not need to implement all features to get a top grade.

To get an indication of the grade you might get, you can run the automated tests using the command `make check`. You may also use `make docker-check`.

Note: Your key-value store will be evaluated largely automatically. This means features only get points if they pass the particular tests, and there will be no half grade for "effort". Most features (and thus points) are split up into multiple separate tests however.

Note2: The test framework may not properly detect incorrect locking and data races. Indeed, automatically verifing the correctness of concurrency programs is challenging. Incorrect uses of locking privitives will be checked by manual inspection.

Run with Docker

You can use *make docker-check* to run all the tests inside a docker container. To enter the container you can run

` docker run --privileged -it --rm -v "`pwd`:/code" -w /code vusec/vu-os-kvstore -check /bin/bash `

Please note that you need to omit the *--read-only* flag in order to properly run the server.

Pthreads

The man pages for pthreads may not be present on your system. To install these, use the following command on Ubuntu/Debian:

```
sudo apt install manpages-posix manpages-posix-dev
```

Pthreads is the standard C interface for threading, adhering to the POSIX standard. It provides APIs to create and manage threads in your program. See `man pthreads` for general info.

Here we list the most common APIs which will be likely useful to you. Please always refer to the man page for an exhaustive description.

API	Descrption
pthread_create, pthread_exit, pthread_attr_init	Create a new thread, manage attributes and exit a thread.
pthread_self	Get id for the running thread
pthread_join	Wait for a specific thread to terminate

Mutual exclusion

API	Descrption
pthread_mutex_init,pthread_mutex_destroy	Initialize and destroy mutex
pthread_mutex_lock, pthread_mutex_trylock, pthread_mutex_unlock	Locking/Unlocking a mutex with a blocking/non-blocking behavior.
pthread_rwlock_init, pthread_rwlock_destroy	Init and destroy and read/write mutex
pthread_rwlock_rdlock, pthread_rwlock_tryrdlock, pthread_rwlock_wrlock, pthread_rwlock_trywrlock	Locking/Unlocking the read/write mutex with a blocking/non-blocking behavior

Synchronization

Conditional variables are built on top of mutual exclusion and they provide synchronization on shared resources.

Conditional variables

API	Description
pthread_cond_wait	Block on a conditional variable

pthread_cond_signal,pthread_cond_broadcast	Unblock on or more threads currently blocked on a conditional variable