COMP 321: Introduction to Computer Systems

Project 5: Key-Value Store
Assigned: 4/14/25, **Due: 4/25/25, 11:55 PM**

Overview

You will write a persistent key-value store that will map keys to values and allow you to set, get, and delete key-value pairs. Such a key-value store is often used to cache values that are expensive to recompute. In general, a key-value store has a fixed total size and rejects new values when the store is full until old values are deleted to make space. The program exposes the key-value store as an HTTP service that can be accessed over the network using well-known HTTP methods.

The goals of this project are as follows:

- Understand how a key-value store operates.
- Understand file system persistence.

General Overview of The Key-Value Store

A key-value store is a system that associates keys with values. Such systems are often deployed in the cloud and accessible via HTTP requests. They are often used as a cache for a larger, more expensive system. For example, you may have a web application that dynamically constructs web pages using multiple database accesses and template computations. The result could be stored in a key-value store (with the key being the URL and the value being the page's HTML) to allow subsequent accesses to that page to be much quicker. When the underlying data changes, the key-value pair would be deleted to force the page to be reconstructed for future accesses. The key-value store should persistently store all key-value pairs that are stored in it until the user of the store explicitly deletes them.

The typical API of a key-value store includes three operations: `get`, `set`, and `delete`. As the names imply `get` takes a key as input and returns the corresponding value, if any, `set` takes a key and a value and stores that pair (either by creating a new pair or overwriting an existing one with the same key), and `delete` takes a key and deletes any existing key-value pair with that key.

You will implement a key-value store that has 64 character keys and values that are abitrary data. Your system will implement the following interface:

```
#define KEYSIZE 64

enum resp get_entry(const char key[KEYSIZE], uint32_t *len, uint8_t **value);
enum resp set_entry(const char key[KEYSIZE], uint32_t len, uint8_t *value);
enum resp delete_entry(const char key[KEYSIZE]);
```

Each of these functions returns one of the following results:

- SUCCESS - the get, set, or delete succeeded

- SERVER_ERROR - there was an internal error processing the request
- NOT_FOUND - in a get or delete request, the key was not found
- NOT_ENOUGH_SPACE - in a set request, the value was too large to fit in the remaining storage

Note that keys must *always* be 64 characters. It is up to the application that uses the key-value store to decide how to use such keys. The values must be at least a single byte and can be as large as desired. As with the keys, it is up to the application using the key-value store to decide how to format and interpret the values. The key-value store itself never interprets the meanings of the keys or the values, it simply associates them so that it can perform the get, set, and delete operations.

Append-only Logging

In order to enable the key-value store to be persistent, the key-value pairs must be stored on disk. Your key-value store will use a single log file for this purpose. The log file is "append only", meaning that as new keys are added or old keys are deleted, you only add a new entry to the *end* of the log file. This ensures consistency of the data at all times. If you were to overwrite a key-value pair in the middle of the file, for instance, and the system crashed during that operation, then you would neither have the old key-value pair nor the new one stored on disk. With an append-only log, the old key-value pair remains available while you write the new one. When the write of the new key-value pair is complete, the the old one is obsolete and the new one should be used for all future requests.

Even if you did try to overwrite key-value pairs in place in the file, this would not work because values are not a fixed size. There is no reason to believe the new value would fit into the same space as the old value. This would require you to use a maximum value size for every value, no matter how small it actually is. This is obviously an inefficient use of disk space and would allow you to store fewer overall key-value pairs in the same amount of space.

Furthermore, when you delete a key-value pair, you also need to append to the log file information that the key has been deleted. Again, you do not want to try to delete the key-value pair in the middle of the file, in case the operation fails before it fully completes.

By using an append-only log in this way, the key-value store is always in a consistent state where the set of key-value pairs always matches the set that existed at the time of completion of the last successful operation. If the system ever crashes (or the machine on which it is running reboots), then you can "replay" the log file to restore the state of the key-value store as it was before the error.

Note that there are several issues that you must deal with when using this approach:

1. As the key-value pairs are stored on disk, you must ensure they are actually written to disk (and are not just in the buffer cache) before you can declare the successful completion of an operation.
2. You must ensure that the key-value store is always in a consistent state. This means when you write a key-value pair to disk, if the store crashes during the write, you must be able to recreate the key-value store *without* the new key-value pair from the information stored on disk (which potentially includes a partial key-value pair that must be ignored).
3. It will be very inefficient to satisfy a `get` request, as you must search the log to find the latest value that was associated with the given key (or determine it was deleted).

To solve the first problem, you must force the data to be written to disk before returning that the operation was successful. You must also make sure you are careful how your order the synchronization operations to solve the second problem. The solution in this assignment is to use an "end of storage" (EOS) operation that signals the end of the log. When you write a new key-value pair, you should not move the EOS marker until *after* the new key-value pair has been written to disk. You should think about why this is necessary.

To solve the third problem, you need an in-memory index to make lookups faster. This index is simply a performance optimization, so you should be able to recreate this index from the log file. For this purpose, we will use a hash table. The hash table will associate keys with locations in the file that hold the corresponding value. The best (and easiest) way to do this is to use `mmap` to map the log file into the address space, and then the keys and values in the hash table can simply be pointers to the appropriate locations in the mapped memory. The operating system will ensure that data is read from disk, when necessary. Note that it does not matter if the file is mapped into different addresses on different executions of the program, as you will recreate the hash table from the log on each launch of the program. In other words, the hash table is not persistent, only the log file on disk.

Double Buffering

Your system will not have infinite space available to it. Therefore, the log file will have a finite size. Rather than allowing the file to grow until the file system runs out of space, we will explicitly bound the size of the log file at the start. The log file can not be further increased in size.

As the log file is append-only, when it fills up, that does not necessarily mean there is no more space. For instance, keys could have been deleted or overwritten with new values. When that happens, there are old key-value pairs in the log that are no longer needed.

To deal with the situation when the log file files up, instead of treating the file as a single append-only log, it should be divided into three parts: a header, a first half, and a second half. These halves will be managed similarly to a copying garbage collector, where the active half is the "new space" and the inactive half is the "old space".

At any given time, only one half is actively an append only log and all new pairs will be stored in the active half. The other half is "inactive", meaning that no new pairs are stored there, but it may contain pairs that are still valid. Each half must have its own hash table that refers to all of the live items in that half. A "live" item is the most recently set key-value pair stored in the log for a particular key that has not been deleted. Note that there may be "old" key-value pairs in the log file that have since been overwritten or deleted. When the active half fills, the halves will be switched. The file header contains information about which half is the currently active half.

In more detail, this works as follows:

- On a `get` operation, you must search both the inactive and active hash tables to determine if the key is stored in the key-value store. If you find it in either half, the value can be returned. If you find it in neither half, then it is not contained in the key-value store. The most recent key-value pair associated with any key can only be contained in one half at any given time, so it should not be possible to find the key in both hash tables.

- On a `set` or `delete` operation, the key-value pair associated with the given key must *always* be deleted from the inactive half's hash table if it is there, as the set or delete entry will be added to the log (and therefore hash table) in the active half. If there is a key-value pair associated with the given key in the active half, a delete operation must be logged and the key-value pair must be removed from the active half's hash table on a delete operation, or a set operation must be logged and the active half's hash table updated on a set operation. If the key is not found in either hash table, it should be added to the active half's log and hash table.

When the active half is "full", you must switch the active half and the inactive half. Note however, that this must happen *before* the active half is actually full. Before switching, all of the remaining "live" pairs in the inactive half must first be written to the log in the active half, to ensure that they do not get lost. So, you must keep a running total of the total amount of space in the inactive half that is live. When adding a new entry (`set` or `delete`) would reduce the unused space in the active half such that there is no longer enough space for all of the live items from the inactive half (minus potentially the key-value pair that is being updated or deleted if it currently resides in the inactive half) to be logged in the active half, then you must switch the halves.

Switching the halves requires the following steps:

1. All of the live items in the inactive half must be logged in the active half and added to the active half's hash table. Then, the hash table for the inactive half should be destroyed.
2. The halves should be flipped by making the active hash table the new inactive hash table, recomputing the count of "used space" in the new inactive half, writing an `OP_EOS` marker at the start of the new active space, creating a new hash table for the active half, and flipping the active half in the file header.
3. Finally, the operation (`set` or `delete`) can be written to the beginning of the new active half (which was empty except for the `OP_EOS` marker after the switch) and the key-value pair can be added to the active hash table on a `set` or removed from the inactive hash table on a `delete`.

If, after switching halves, there is still not enough room for the new operation, then the key-value store must reject the operation because it is full. Note that this can only happen on a `set` operation, never on `delete`. You should think about why.

Log File Format

The log file starts with a `struct file_header`, defined as follows::

```
#define MAGIC_NUM 0xabcddcba // 32 bit number files must have in header

typedef uint32_t half_type;

#define FIRST  0
#define SECOND 1

struct file_header {
        uint32_t  magic;        // Magic number to check if file is valid
        half_type active_half;  // Which half of the file is currently active
        uint64_t  storage_size; // How much storage space in each half
};
```

The two halves, each of size `storage_size` immediately follow this header.

Each half consists of a sequence of log entries. Each log entry should consist of a `struct file_entry`, defined as follows:

```
#define OP_SET     0  // Set operation
#define OP_DELETE 1  // Delete operation
#define OP_EOS     2  // End of storage marker
```

```
typedef uint32_t op_type;

struct file_entry {
        op_type  op;            // Type of operation
        uint32_t padding;       // To make this 80 bytes, unused
        char     key[KEYSIZE];  // Key of the entry
        uint32_t len;           // Length of the value
        uint32_t padded_len;    // length for 8-byte padding
        uint8_t  value[];       // Value of the entry
};
```

The `op` must be one of `OP_SET`, `OP_DELETE`, or `OP_EOS`. The padding is unused in order to make everything consistent and aligned. The `len` is the actual length of the value stored in the `value` field. The `padded_len` is `len` rounded up to the nearest 8 bytes, which should be used to determine the starting location of the next log entry. The last item in the active half must *always* have an `OP_EOS` value for its `op` to indicate the end of the active storage. This means that an "empty" active half must have a single entry with an `op` of `OP_EOS`.

The `OP_EOS` entry is used to find the end of the append-only log when reconstructing the key-value store when opening an existing log file at the start of the program. It is the only way to know that the rest of the active half has not yet been used. Note that an `OP_EOS` operation does not need to consist of a complete `struct file_entry`. All other fields are meaningless (and do not really exist) for the final `OP_EOS` entry in the log.

When reconstructing the key-value store from a log file, each entry must be "replayed" in order first from the inactive half up to its `OP_EOS` marker and then from the active half up to its `OP_EOS` marker. An initially completely empty log file would have `OP_EOS` markers at the beginning of both halves.

Persistence

To achieve persistence with your key-value store, you will need to carefully think about both ordering and synchronization. Your key value store should operate such that at *any* time if the program crashes (because someone killed it, because of an error, or because the machine shut down, etc.), then it will be in a consistent state when you start it up again with the same log file. This means that when you reconstruct the key-value store, the current state of the store should reflect the effects of *all* operations before the last one and either the last operation should have completely taken effect or not taken effect at all.

Note that for operations that would trigger a half switch, it is okay if the switch doesn't start, fully complete, or is in progress, as long as the observable state of the key-value pairs in the store meets the criteria above. In other words, as you copy log entries from one half to the other, you need to ensure each key-value pair is fully correct in one or the other half of the log file on disk at all times and that neither half becomes corrupted at any time.

In general, in order for your key-value store to be persistent and always remain consistent, you must be careful about the ordering and synchronization you use *every* time you update *anything* in the log file. In particular, you must write the log in the correct order with the correct synchronization on a delete operation, a set operation, and when switching halves. No other operations should modify the log file.

Getting Started

Please visit the web page at `https://classroom.github.com/a/jzcOCVaE`. (If you are copying this URL, do not include the period at the end of the sentence, as it is not part of the URL.) This page should say "RICE-COMP321-S25-Classroom" and "Accept the assignment — KVStore". Moreover, it should have a green button labeled "Accept this assignment"[1]. Please accept the assignment.

Upon accepting the assignment, you will be redirected to another web page. This page will confirm that you have accepted the assignment, and it will eventually (after you click refresh) provide you with a link to your personal repository for the assignment. Click this link to go to your personal repository.

The web page for your personal repository has a green button labeled "Code". Click this button. You should now see a text field with a URL. Copy or remember this URL.

Login to the `CLEAR9` system if you have not already done so. Type the following:

`git clone` [Copy the URL for your repo here]

You will be prompted for your `github` username and password.

Once the clone operation is complete, you will have a directory named

`kvstore-`[YOUR `github` ID]

Please `cd` into this directory, and do the following:

- Type a header comment at the top of `kvstore.c` with your name and NetID.
- Type the command `make` to compile and link a skeleton kvstore executable and some simple programs you can run to test it.

The provided code includes the following source files for the `kvstore` program:

1. `kvstore.c/h`
2. `hash_table.c/h`
3. `tiny.c`
4. `rio.c/h`
5. `file_initializer.c/h`

The `kvstore.c` file is the heart of the key-value store. It implements all of the key-value store functionality. You will need to complete this file to create a functioning key-value store.

The `hash_table.c` file is a complete hash table implementation from COMP 222.

The `tiny.c` and `rio.c` provide an implementation of the tiny web server from the text book and threads assignment. It is common for cloud-based services to integrate a web server directly into the program. By doing so, the key-value store is accessible over the network using the well understood HTTP protocol that includes "GET", "PUT", and "DELETE" methods that correspond nicely to our key-value stores `get`, `set`, and `delete` operations.

You will not need to generate HTTP requests. Instead, you can use the provided `kvtester` program to test your key-value store.

Finally, the `file_initializer.c` file enables the creation of new, empty log files for use with your `kvstore` program.

[1] You may have to login to GitHub to see this page. If so, you will be prompted for your GitHub username and password.

Checking Your Work

You can run your program as follows:

```
$ ./kvstore <port_num> <filename> [-c size]
```

This allows you to run your server listening on port `<port_num>` opening `<filename>` as the log file. If `-c size` is specified, it will create a new log file named `<filename>` of the specified size. It is an error to specify the create option (`-c`) with the filename of a file that already exists. You would then need to send it HTTP requests to exercise your program.

However, you likely will not run your program directly. Instead, to simplify running tests, we have provided a testing program here:

```
$ /clear/www/htdocs/comp321/assignments/kvstore/kvtester
```

To make it easier to run, you can link it into your kvstore directory as follows:

```
$ ln -s /clear/www/htdocs/comp321/assignments/kvstore/kvtester kvtester
```

Then, in your kvstore directory, you can run the provided test program as follows:

```
$ ./kvtester -p <port_num> -t <test_name>.json
```

Where `port_num` is the port on which you want to run your kvstore on and `test_name` is the name of one of the tests described below. This will run your `kvstore` program with the appropriate arguments and then send it the HTTP requests from the test and check the results. If the test passes, the program will print a line:

```
Test passed name=<test_name>
```

There are two parts to each test. First, the tester runs a series of requests against the kvstore, and checks that the kvstore returns the expected statuses and values for each request. If one of these requests fails, the test fails and will print a line "Trace Failed", which will be immediately preceded by a line describing the failure.

Second, the tests compare the log file contents at the end of the test to their expected values to ensure that the entries have been properly logged and managed. If the test fails on this part, it will print a line

```
Test failed on file check name=<test_name>
```

This will be immediately preceded by a line describing the failure.

The following tests have been provided:

1. `test_get_delete_nonexistent`: Tests that the KVStore correctly returns 404 for items never added, and continues to do so after ”deleting” that element.

2. `test_get_delete`: Tests that the KVStore correctly returns 404 for items never added, gets the element successfully after it has been set, and returns 404 after it has been deleted.

3. `test_overwrite`: Tests that KVStore correctly replaces a key which has been overwritten, returning the newest value.

4. `test_replay`: Tests that the KVStore can correctly replay a basic log file, with gets on elements which were added, then removed, then readded; an element added, then removed; an element added; and an element never added.

5. `test_second_replay`: Tests that the KVStore can correctly replay a log file with the active half being the second half.

6. `test_switch`: Tests that the KVStore correctly switches halves when the active half has been sufficiently filled. Validates that, during the switch, no entries were erroneously removed or re-added.

7. `test_random_X`: `X` can be one of 1, 2, 3, or 4. These traces are longer traces that test that the KVStore can handle long sequences of operations.

Notes

- Be sure to familiarize yourself with the provided code before starting.
- Do *not* modify the provided code. This includes the structure definitions, file layout, etc.
- Develop your code incrementally. Do not try to make everything work all at once without testing anything. This is a recipe for failure.
- One particular issue you should pay attention to is that you *cannot* update the `op` field that contains `OP_EOS` until the next new entry has been completely written and synchronized to disk. Since the existing `OP_EOS` and the `op` field of the new entry occupy the same memory location, this means that a new entry needs to initially have the value of `OP_EOS` in the `op` field. Furthermore, a new `OP_EOS` entry also needs to be written after the new entry and synchronized to disk. Only after both the new entry (with the `OP_EOS` op) and the new `OP_EOS` entry have been written and synchronized, is it safe to change the `op` value of the new item from `OP_EOS` to the actual value (`OP_DELETE` or `OP_SET`). Performing the updates in this order ensures that either the (potentially only partially written) new log entry will be after the `OP_EOS` when restarting (so it has effectively not been written to the log) or the completely written new log entry will be in the log before the `OP_EOS`.

  Note that the `OP_EOS` entry is very similar to the epilogue header in the malloc assignment and is managed in a similar fashion (though it must be persistent).
- Whenever you delete a key-value pair from the inactive hash table, don't forget to reduce `inactive_space_used` by the overall size of the key-value pair.
- When you switch halves, you will need to copy each "live" entry in the inactive half to the active half. This needs to be performed in the same way that you would write any other new entry to the active half in order to ensure that either the entry is completely written to the active half with the new `OP_EOS` marker after it, or the original `OP_EOS` marker is not changed and the entry in the inactive half is still the latest version.
- When you switch sides, all copies need to be completed and synchronized before you can update and synchronize the active half in the log file header.

Turning in Your Assignment

To turn in your code, you ***must*** use `git push` to copy your work to the GitHub remote repository. We will *only* look at the last version that you `push`ed before the deadline. As a precaution against accidental loss of your code, we encourage you to `push` periodically. Please note, the *only* file that you need to turn in is `kvstore.c`. **You should not modify or add any other files in your repository.** In other words, this is the only file on which you should ever perform `git add`. **Do not ignore this instruction.**

For grading your submission, we will use the `Makefile` that was originally provided to compile your code. Therefore, your code should not rely on any modifications to the `Makefile` for correct compilation.

As a sanity check, you should use your web browser to visit your assignment repo on GitHub. Make sure that what you see in the browser is consistent what you think you have pushed.

Evaluation

The project will be graded as follows:

- Correctness: 100%

Your solution will be tested for correctness on a CLEAR9 machine.