Assignment 2: Simple Key-Value Store Report
ECSE 427: Operating Systems

Muhammad Anza Khan
ID: 260618490
muhammad.anza.khan@mail.mcgill.ca

November 5, 2018

1 Introduction

I am writing this report to outline all the work I did regarding the second assignment for ECSE 427 this semester. The main goal of this assignment was to implement an in-memory key value store. This key-value store had to be set up in shared memory so that the data that is written to the store persists even after the process that performs the write is terminated. The implementation of the simple key-value store can be broken into three parts: **setup of the store**, **writing to the store**, and **reading from the store**.

From the lecture in processes we know that each process has its own memory space. We use the inter-process communication (IPC) mechanism of the OS to setup a shared memory space that can be used as the store. So the idea is pretty simple: create a shared memory space and store all the records there. The records stored in this space can be accessed by all processes.

With regards to the structure of the store itself, the reader can imagine it as a persistent hash map. This has two distinguishing attributes **many reading and writing processes** and **large number of records that need the store to evict older or unused records to make room for new entries.** This is what I ended up implementing for the design of the store. Suppose the key-value store has size of n key-value pairs. We can split the store it into k pods (k could be multiple of 16). Each pod can hold n/k entries or records. Now, when a key-value pair is written by a writing process, we hash the key to obtain a pod index. We insert the key-value pair in the pod. Similarly, when we want to search, we compute the pod index using the given key and look for the entry in the pod. Depending on how good the hash function is in distributing the entries among the pods, we can have some pods overflowing before the others.

I used a total of **256** pods, each pod holds 256 key-value pairs, thus we can see that the total number of bytes the store can hold is 256 x 256 = **65536** bytes. With regards to the size of the key-value pair itself, it comprises a maximum length of **32** bytes for the key and a maximum number of **256** bytes for the value, thus the size, in bytes of a key-value pair is 32 + 256 = **288** bytes. Now the natural question to ask is what is the relationship between our pods and the key-value pair? A bit of thinking yields us the thought of asking, well how many key-value pairs can be stored in a pod? Well, reader it is simple math, really each pod can hold 256 x 288 = **73728** bytes or around 72 MB of data. Now if we scale this to accommodate for all the pods, this number is further multiplied by 256 for all the pods resulting in 256 x 73728 = **18** GB of data.

A major issue we had to tackle was **synchronization**. When multiple processes access the shared memory region, we run into the synchronization problem. As such to avoid this issue need to take care of the mutually exclusive access needed for race free update of the shared structures. The UNIX operating system (Linux in our case), provides different variations of semaphore: System V, POSIX, etc. In this assignment, we had to use the POSIX semaphores. POSIX semaphores can be created in two different ways: named and anonymous. We used the named semaphore in this project. With the named semaphores we do not need to put them in a shared memory. Different processes can share a semaphore by simply using the same name. Once a semaphore is successfully created, two types of operations can be performed on them: **wait()** and **post()**. The wait() function decrements the semaphore and if the value is greater than or equal to 0, it returns immediately. Otherwise, it gets blocked. Whether the value of the semaphore goes to negative or not is implementation dependent. In Linux, the value does not go negative. The process is blocked until the value becomes greater than 0 and then the value is decremented. The **post()** operation increments the semaphore value. If a process is waiting on the semaphore, it will be woken and allowed to decrement the semaphore value. If multiple processes are waiting on the semaphore, one process is arbitrarily woken up and allowed to decrement the semaphore value. That is, only one process is let go when a post() operation happens among the waiting processes, however, the order in which the waiting processes are let go is undefined.

With this introduction done, I will now go into detail about the various APIs I wrote, their function and how I went about implementing them.

2 int kv_store _create(char *name)

The kv_store _create() function creates a store if it is not yet created or opens the store if it is already created. After a successful call to the function, the calling process has access to the store. This function

could fail, if the system does not enough memory to create another store, or the user does not have proper permissions. In that case, the function returns -1. A successful creation results in a 0 return value.

3 int kv_store _write(char *key, char *value)

The kv_store _write() function takes a key-value pair and writes them to the store. If the store is already full, the store needs to evict an existing entry to make room for the new one. In addition to storing the key-value pair in the memory, this function needs to update an index that is also maintained in the store so that the reads looking for an key-value pair can be completed as fast as possible.

In my implementation, firstly I call a semaphore to get exclusive access as we are about to enter the critical section. This semaphore is called **db**. Next we generate a hash index ranging from 0 - 255 which gives us the pod the value will be written to. Than I multiply this index by the size of a pod to accommodate for the address of the pod in memory where the value will be written. I keep track of the positions already written to using a write counter in my **Data** struct which the TA told us can be used for book keeping information. If a pod is full, I simply wrap it around by 256 to start at the beginning, again 256 because the number of pods we have.

4 int kv_store _read(char *key)

The int kv_store _read() function takes a key and searches the store for the key-value pair. If found, it returns a copy of the value. It duplicates the string found in the store and returns a pointer to the string. It is the responsibility of the calling function to free the memory allocated for the string. If no key-value pair is found, a NULL value is returned.

In my implementation, again we firstly wait on our semaphore **mutex**, this provides us with exclusive access to the critical region. To accommodate for additional reader, we increment the a reader counter which is again part of my **Data** data structure which is used for book keeping purposes. Once we get access, we cycle through all possible keys stored in the key-value store and check it against our current key, if found, we duplicate it and return it, if not ultimately we return NULL.

5 int kv_store _read _all(char *key)

The function int kv_store _read _all() takes a key and returns all the values in the store. A NULL is returned if there is no records for the key.

6 int kv_delete _db(char *key)

This is the final function I wrote, it simply deletes the store.