Assignment 2 - Key-Value Store
COMP 310 - Operating Systems
Prof. Muthucumaru Maheswaran
Fall 2018

LE, Nhat Hung
McGill ID: 260793376
Date: November 6 2018
Due date: November 5 2018

Store specifications i.e. tester settings

Bigger sizes for the data structures would only sometime work, and often give segmentation faults on my personal laptop. The chosen settings are the highest that gave no segmentation faults. The tester (tester 1 at least) ran with no errors.

a2_lib.h:

```
#define STORE_NAME "hle30_260793376"
#define SEM_READ_NAME "hle30_read_lock"
#define SEM_WRITE_NAME "hle30_write_lock"
#define POD_CAPACITY 128
#define NUM_PODS 32
#define KEY_SIZE 31
#define VALUE_SIZE 256
```

Modifications made in tester header:

comp310_a2_test.h:

```
#define __TEST_MAX_POD_ENTRY__ 128
#define __TEST_MAX_KEY__  64
#define __TEST_SHARED_MEM_NAME__ "hle30_260793376"
```

Data structures

The store:

- `struct store_t` is an array of pods `struct pod_t`

The pod:

- A `struct pod_t` adopts the general data structure of a circular array/queue. It contains an array of entries struct entry_t, which is the queue, and additional data as follow:
  - `int size`: number of entries in the pod
  - `int head, tail`: circular queue implementation
  - `int read_count`: implementation of unique semaphore for each pod
  - `int next[]`: array, each index corresponding to the index of an entry in the pod. Emulates a linked list linking entries of the same key
  - `int read[]`: array, each index representing a the ID of a unique key in the pod
    - E.g. three identical keys in the pod will be represented by the same ID and be represented by the same index
    - The read function looks into the cell of the key's ID to see which entry to read, then updates the array for the next time it's called

The entry:

- A `struct entry_t` contains two string buffers. One for the key, the other for the value.

- It also contains an ID number to identify unique keys present in the pod. The ID is thus an identifier for keys. The source files' comments refer to it as the entry ID, but only because that's how it is defined in the structure.

  The data structures:

  a2_lib.h:

```
struct entry_t {
  int id;
  char key[KEY_SIZE];
  char value[VALUE_SIZE];
};

struct pod_t {
  int size;
  int head;
  int tail;
  int read_count;
  int num_unique_keys;
  int read[POD_CAPACITY];
  int next[POD_CAPACITY];
  struct entry_t entries[POD_CAPACITY];
};

struct store_t {
  struct pod_t pods[NUM_PODS];
};
```

Implementation details

The biggest challenge lied in read function's requirement to loop through all values of the same key in a pod. This required the implementation of the linked list (`next[]` array) and of the `read[]` array.

The general flow is as follows, expressed in pseudocode:

```
write(key, value) {
      Check if entry isn't already in pod. If it isn't then proceed.

      *entry = get entry address from hashed key

      if pod is full {
            ... // we will see later
      }

      Write key and value into entry

      entry.next = entry  // assume new entry is the only one with its key
      nextEntry = most recently added entry with the same key
```

```
        if no other entry with same key is found {
                entry.id = new id different from all others in pod
        }
        else {
                entry.id = nextEntry.id

                Here there are two cases:
                1. The new entry doesn't overwrite anything or overwrites an entry of
                a different key
                2. The new entry overwrites an entry of the same key

                if case 1 {
                        // point entry.next to the oldest entry of the same key
                        entry.next = nextEntry.next

                        // point nextEntry.next to the new entry
                        nextEntry.next = entry
                }
                else case 2 {
                        If the new entry overwrites an entry of the same key, then
                        nextEntry.next is already pointing to the new entry. We need
                        only then to point entry.next to nextEntry

                        entry.next = nextEntry
                }
        }

        // the newly added entry will be the first to be read by the read function
        read[entry.id] = entry.index
}
```

Now that we have seen how the linked list is built and updated, we can go back to the snippet of code in bold.

We check if the pod is full BEFORE writing the new entry, because of the pod is full, then we know the new entry will overwrite an old one.

The problem is that the old entry might already be a part of a populated linked list with other entries sharing its key. Therefore, before overwriting the old entry, must disconnect it from the linked list so the read function can correctly loop as desired.

```
        if pod is full {
                entry behind overwrittenEntry = overwrittenEntry.next
        }
```

With the linked list and read[] array updated every time an entry is written, the idea of the read() function is to use the linked list and update the read[] array for the next times it's called.

```
read(key) {
	// get an arbitrary entry of the desired key in the pod
	*entry = get entry address from hashed key

	// read the entry's id, and use the read[] array to get the desired entry
	entry = read[entry.id]

	// update read array for the next read() call
	read[entry.id] = entry.next

	return clone of entry.value
}
```

With this, every call of read() will return a different value with the same key (or only one value if it's the only one with the desired key).

Additional information

Among the submitted source files, `a2_helpers.c` is not a modified version of the file posted by the TA's, but my own.