McGill

Programming Assignment #2: Simple Key-Value Store

Due date: Check My Courses`

This assignment should be completed individually. You can collaborate in the design phase but the implementation should be an individual effort.

This assignment should be completed in a Linux machine. If you don't have Linux running on your personal machine, please use the ENGTR 3rd Floor Linux Lab machines.

1. Overview

As part of this assignment, you are expected to implement an in-memory key-value store. This key-value store will be setup in shared memory so that the data that is written to the store persists even after the process that writes it terminates. Because the shared memory is in RAM, the reads and writes are very fast. The amount of storage, however, is limited. Also, the store is not durable. If the machine reboots, the data is lost. Although it is possible to have a design of the memory-based key-value store, where the contents are stored to the disk for durability, we won't attempt it in this assignment.

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. Setting up the store involves creating a shared memory and attaching it to the current process.

From the lecture in processes you 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.

To setup the shared memory, you need to use the POSIX shared memory mechanisms. There are alternate mechanisms (e.g., System V), however, POSIX shared memory is what is required here. Also, POSIX shared memory has some known issues with Mac OS. Therefore, this assignment should be attempted in Linux (any Linux distribution with a recent kernel should be fine).

In POSIX shared memory a memory region that needs to be shared is referred to as the *memory object*. To create a memory object, you need to use the following library routine. The total size of all the shared memory objects that can be created in a single machine is limited by the size of the *tmpfs* file system setup by the system administrator. I don't think this limit would be a problem unless large number of students use the same machine for running their programs. You can check the **/dev/shm** directory to see shared memory objects in a machine. You need to do two steps to setup a shared memory object.

1. Use *shm_open()* function to open an object with the specified name. This is similar to the open() for files.
2. Pass the file descriptor obtained in the above step to a mmap() that specifies the MAP_SHARED in the flags argument.

The name argument of shm_open() identifies shared memory object to be created or opened (i.e., something already created). This function can include specific flag values and mode values to define the configuration required from the shared memory object.

```
#include <fcntl.h>           /* Defines O_* constants */
#include <sys/stat.h>        /* Defines mode constants */
#include <sys/mman.h>

int shm_open(const char *name, int oflag, mode_t mode);
```

Returns file descriptor on success, or –1 on error

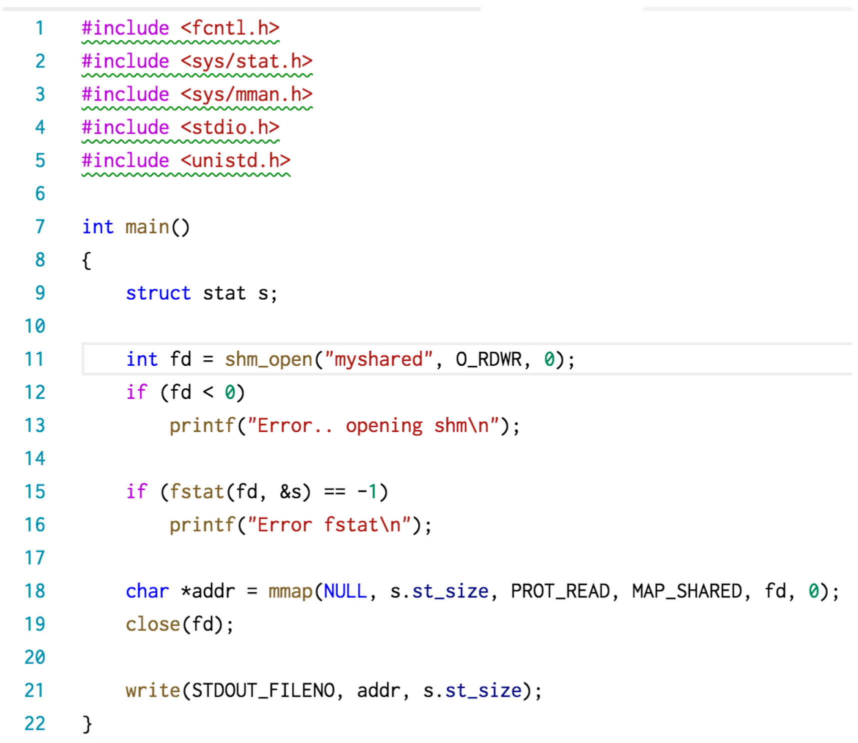
Flag	Description
O_CREAT	Create object if it doesn't already exist
O_EXCL	With O_CREAT, create object exclusively
O_RDONLY	Open for read-only access
O_RDWR	Open for read-write access
O_TRUNC	Truncate object to zero length

The program listing below shows a small example program where the string provided as the first argument is copied into the shared memory. The shared memory is created under the name "myshared." Obviously, you need to use a unique name that is related to your login name so that there are no name conflicts even if you happen to use a lab machine for running the program. The mmap() is used to map the shared memory object starting at an address. By specifying a NULL value for the first argument of the mmap(), we are letting the kernel pick the starting location for the shared memory object in the virtual address space.

```c
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>

int main(int argc, char **argv)
{
    char *str = argv[1];
    int fd = shm_open("myshared", O_CREAT|O_RDWR, S_IRWXU);

    char *addr = mmap(NULL, strlen(str), PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    ftruncate(fd, strlen(str));
    close(fd);

    memcpy(addr, str, strlen(str));
}
```

The ftruncate() call is used to resize the shared memory object to fit the string (first argument). As the last statement we copy the bytes into the shared memory region.

The example below shows a program for reading the contents in the shared memory object and writing it to the standard output. Obviously, the name of the shared memory object should match the one in the above program. The fstat() system call allows us to determine the length of the shared memory object. This length is used in the mmap() so that we can map only that portion into the virtual address space.



```c
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
    struct stat s;

    int fd = shm_open("myshared", O_RDWR, 0);
    if (fd < 0)
        printf("Error.. opening shm\n");

    if (fstat(fd, &s) == -1)
        printf("Error fstat\n");

    char *addr = mmap(NULL, s.st_size, PROT_READ, MAP_SHARED, fd, 0);
    close(fd);

    write(STDOUT_FILENO, addr, s.st_size);
}
```

The two example programs illustrate the three major activities we need to perform with the POSIX shared memory: setting up, writing, and reading.

Another problem to consider is synchronization. When multiple processes access the shared memory region, we will run into the synchronization problem. At the very least, we 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, you need to use the POSIX semaphores. POSIX semaphores can be created in two different ways: named and anonymous. You need to use the named semaphore in this project. With the named semaphores you don't need to put them in a shared memory. Different processes can share a semaphore by **simply using the same name**. (Because we have shared memory objects, we could have used the anonymous semaphores as well.)

To create a named semaphore, use the following library function.

```
#include <fcntl.h>           /* Defines O_* constants */
#include <sys/stat.h>        /* Defines mode constants */
#include <semaphore.h>

sem_t *sem_open(const char *name, int oflag, ...
                /* mode_t mode, unsigned int value */ );

                Returns pointer to semaphore on success, or SEM_FAILED on error
```

Once a semaphore is successfully created, two types of operations can be performed on them: *wait()* and *post()* (we referred to the post as signal in the lectures). The POSIX semaphore also provides additional APIs to obtain the current value and try waiting. The figure below shows the library function for waiting on a semaphore. It 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.

```
#include <semaphore.h>

int sem_wait(sem_t *sem);
                                    Returns 0 on success, or −1 on error
```

The post() operation shown below 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.

```
#include <semaphore.h>

int sem_post(sem_t *sem);
                                    Returns 0 on success, or −1 on error
```

2. Suggested Interface

You can think of the key-value store as a persistent hash map. There are two distinguishing attributes:

1. many reading and writing processes
2. large number of records that need the store to evict older or unused records to make room for new entries.

We could multiple writes with the same key value – that is duplicate writes. We want to save all duplicate values in the store. That is, there is no overwriting of values.

The following API is suggested for your implementation. You may implement a different API provided your implementation subsumes the functionality offered by this API. **The supplied testing routine could be affected if you change the API too much. You need to use the tester programs.**

```
int kv_store_create(char *name);
int kv_store_write(char *key, char *value);
char *kv_store_read(char *key);
char **kv_store_read_all(char *key);
```

Here are brief descriptions of each API routine. You need to take this into account while implementing them.

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 should have 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 should return -1. A successful creation should result in a 0 return value. The creation function could set a maximum size for the key-value store, where the size is measured in terms of the number of key-value pairs in the store.

The **`kv_store_write()`** function takes a key-value pair and writes them to the store. The key and value strings can be length limited. For instance, you can limit the key to 32 characters and value to 256 characters. If a longer string is provided as input, you need to truncate the strings to fit into the maximum placeholders. 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.

The **`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.

The **`kv_store_read_all()`** function takes a key and returns all the values in the store. A NULL is returned if there is no records for the key.

3. Suggested Implementation Approach

One of the important problems to solve here is synchronization: readers and writers (R&W) problem. See Section 2.5.2 of the textbook. You can fit the algorithm provided there to achieve the synchronization in your store. In the R&W problem, readers and overlap with other readers. However, there cannot be any overlap with writers. A writer needs mutually exclusive access to the database. Also, when a writer is updating the database, readers need to wait as well. Otherwise, readers would get invalid (partially updated) values.

The simplest implementation would be to just store the key-value pair into the shared memory object. You create a record that holds the bytes of key + value and just copy them into the shared memory. You keep track of the number of records in the store at any given time and just append to the end of the store and search all the store to find a record. This is not very efficient – at least for retrievals. We can use an indexing structure like a hash table to speed things up for the search. This could be quite complicated for the assignment because the hash table needs to be fully contained in the shared memory.

We believe the approach suggested below is a good compromise. It is not inefficient as the most naïve approach nor is it complicated as maintaining a separate index structure. Further, it could allow multiple updates to take place simultaneously. The idea is simple. Suppose the key-value store has size of *n* key-value pairs. Let's 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 the **`kv_store_write()`**, 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. It also depends on the arrival pattern of the key-value pairs. We will ignore the performance concerns here!

Your design needs to handle duplicate keys. With duplicate keys we can have multiple records with the same key but different values, so they are distinct key-value pairs. The write function should succeed in inserting the key-value pairs in the same pod (because the key is the same). When the read function is executed it should return one value at a time. To read all values, you just keep calling the read function. It cycles through the available values. So if there is only one record for a key, the same value is returned all the time.

The key-value store needs to use a first-come first-out (FIFO) discipline to evict the records when more space is required. That is, the record that is written the earliest is evicted when new records are written and all available space is exhausted. Because the store is divided into pods, the space eviction could happen in a pod even though there is space in the store.

4. Testing and Evaluation

We will provide you with a tester for the key-value store. You could write your own testers an submit with the assignment. If our testers are not working and you have substituted with your own testers, they should be equivalent in functionality to the one(s) we provide.

The programming assignment should be submitted via My Courses. Other submissions (including email) are not acceptable. **Your programming assignment should compile and run in Linux. Otherwise, the TAs can refuse to grade them**. Here the mark distribution for the different components of the assignment.

Simple store that can store and retrieve data	40%
Working FIFO	15%
Working with multiple readers & writers	20%
Correct multi-pod implementation (allowing multiple writers to the different pods)	10%
Memory leak problems	5%
Code quality and general documentation (make grading a pleasant exercise)	10%