INFO0940-1: Operating Systems (Groups of 2)
Project 3: Adding System Calls

Prof. L. Mathy - G. Gain - K. Yasukata

Submission before Wednesday, April 15, 2020

ATTENTION: In this assignment, the number of submission is limited. Students can submit their programs to the submission platform up to **15** times. To avoid running out all challenges for failure, use the testing environment. Usage of the test environment is described in the slide.

1 Introduction

In this assignment, you will implement a Key-Value-Store (KVS) functionality in the Linux kernel. The kernel-space KVS offers three system calls for insert, search, and delete operations.

2 Key-Value-Store (KVS) Primer

KVS is a type of data store software used to store specific information. Users can register a pair of key and value. Typically, we can perform three types of operations, insert, search, and delete.

For instance, let's think about a KVS storing pairs of web browsers and vendors. Suppose the key is the name of a browser, and the value is the name of a vendor.

For storing data, we use the `insert` operation as follows[1]

```
insert("Internet Explorer", "Microsoft")
insert("Edge", "Microsoft")
insert("Chrome", "Google")
insert("Firefox", "Mozilla")
```

Now, if we want to know which vendor provides Internet Explorer, we can check it by using the `search` operation.

```
vendor = search("Internet Explorer")
# here "vendor" has a pointer to string "Micorsoft"
```

[1] The following example is simplified. Detailed interface specification follows below.

Oh, by the way, Microsoft has stopped developing Internet Explorer. If we do not want such old information, we can remove an entry using the `delete` operation.

```
delete("Internet Explorer")
vendor = search("Internet Explorer")
# here "vendor" should not have any value,
# that means not found
```

As you see, KVS is quite simple. Nevertheless, it is quite popular, and many production systems use it. `Memcached` and `Redis` are representative implementations.

3 Assignment

For this project, you will develop the KVS functionality within the Linux kernel.

The KVS functionality implements the following system calls as an interface for user-space applications.

```
long kvs_insert(const char *key, size_t keylen,
                const char *val, size_t vallen);

long kvs_search(const char *key, size_t keylen,
                char *val, size_t max_vallen);

long kvs_delete(const char *key, size_t keylen);
```

3.1 kvs_insert

Arguments specification:

- `key`: pointer to a key string to be stored.
- `keylen`: length of the key string.
- `val`: pointer to a value string to be stored.
- `vallen`: length of the value string.

Return value: On success, it returns 0. On failure, it returns -1.

If there is already an entry whose key is the same as the currently requested one, please replace the old value with the new value.

3.2 kvs_search

Arguments specification:

- `key`: pointer to a key string.
- `keylen`: length of the key string.
- `val`: pointer to a buffer that will receive the result value.
- `max_vallen`: size of the `val` buffer.

Return value: If it finds a value for a specified key, it returns 1. If no value is found, it returns 0. If other problems occur, it returns -1.

You need to copy the found value onto `val` specified as the argument. The buffer size of `val` will be set in max_valllen.

3.3 kvs_delete

Arguments specification:

- `key`: pointer to a key string.
- `keylen`: length of the key string.

Return value: On success, it returns 0. On failure, it returns -1.

Even if the `delete` operation does not find the corresponding entry, and does not delete any entry, return 0.

3.4 System Call Numbers

The system call number assignment is as follows:

- `kvs_insert`: 385
- `kvs_search`: 386
- `kvs_delete`: 387

3.5 Other Specification

- Your implementation should be able to store key and values whose lengths are kmalloc's maximum allocatable size;
- Your implementation should be able to store reasonably large number of entries. This means two things, your implementation should be reasonably memory efficient, and you can add entries until available memory space is almost fully utilized;

- We do not consider the case to store datasets that do not fit into memory;
- We do not consider concurrency of the KVS operations. In other words, we assume that we execute the operations one by one;
- You can use any type of data structures such as list, tree, and hash tables;
- You must only consider x86 architecture (32bits).

3.6 Generic Requirements

The following requirements also have to be satisfied:

- Your code must be readable. Use common naming conventions for variable names and comments.
- Your code must be robust and must not crash. In addition, errors handling and cleaning must be managed in a clean way.

4 Evaluation and tests

Your program can be tested on the submission platform. A set of automatic tests will allow you to check if your program satisfies the requirements. Depending on the tests, a **temporary** mark will be attributed to your work. Note that this mark does not represent the final mark. Indeed, another criteria such as the structure of your code, the memory management. You are however **reminded** that the platform is a **submission** platform, not a test platform.

5 Submission

Projects must be submitted before the deadline. After this time, a penalty will be applied to late submissions. This penalty is calculated as a deduction of 2^{N-1} marks (where N is the number of started days after the deadline).

Students will submit patches that contain diffs from linux-4.15. The patch file(s) has/have to be generated by the `git format-patch` command. Your submission will include your patch files in a directory named `patch`.

Submissions will be made as a `patch.tar.gz` archive, on the submission system. Failure to compile will result in an awarded mark of 0. Students can submit their files up to 15 times.
Bon travail...