KVFS - A Key-Value Store Filesystem Driver for Linux

Jacob Garby [j4cobgarby@gmail.com]

December 2022

1 Introduction

KVFS (Key-Value Store Filesystem) is a loadable kernel module (LKM) which provides a key-value store (KVS) in the form of an in-memory virtual filesystem. There may be any number of KVS's present on a system at any given time (often mounted at `/dev/kvs1`, `/dev/kvs2`, etc., unless the user desires more descriptive naming). *KVFS* handles synchronisation issues that may arise if multiple threads are trying to access it simultaneously. In addition to simple setting and getting of keys' values, *KVFS* also provides a mechanism for incrementing and decrementing numeric values.
KVFS provides a valuable service to userspace applications, with various uses. It can be used to provide a mechanism for system-wide configuration, for example, that can be accessed from any application that needs to use it. Some existing methods of global configuration on Linux can be complex and difficult for new users, so this alternative could be useful.

2 Filesystem Structure

When a *KVFS* filesystem is mounted somewhere within the Linux Virtual Filesystem, four files are present in its root directory. These are `_mk`, `_del`, `_inc`, and `_dec`.
`_mk` and `_del` can be used to make or delete keys from the system. Writing the name of a key to `_mk` (e.g. `echo -n keyname > /dev/kvs1/_mk`) will, from the users point of view, create a new file in the filesystem's root, with the name of their new key. From the driver's point of view, a new *inode* has been created for that key. An *inode* in Linux contains a field for "private data", which *KVFS* utilises to store the associated value. When a key is first initialised, its value is NULL.
Writing the name of a key to `_del` will delete the file representation from the directory, and free up the associated memory in kernel space.
`_inc` and `_dec` can be used in a similar fashion to increase or decrease the value associated with a given key by 1. This will only work if the current value of that key can be interpreted as an integer (in base 2, 8, 10, or 16).
As mentioned, keys are also reflected as files in the root directory. Key files can be written to and read from like a regular file, except that the offset is ignored when writing. This decision is due to the fact that the write operation is thought of more as a *SET* command in other key-value stores.

3 Usage

Here are several examples of interacting with a *KVFS* filesystem. We'll assume that a KVS is mounted at `/dev/kvs1`[1].

[1] This mounting can be achieved by first creating the directory, and then running a command such as `mount -t kvfs none /dev/kvs1`

The first example is using the *bash* shell:

```
# Create a new empty key called 'my_key'
$ echo -n my_key > /dev/kvs1/_mk

# Set the value of 'my_key' to 'Hello, world!'
$ echo -n "Hello, world!" > /dev/kvs1/my_key

# Read the value of 'my_key'
$ cat /dev/kvs1/my_key

# Delete the key 'my_key'
$ echo -n my_key > /dev/kvs1/_del
```

Note that we use the `-n` flag when calling `echo` to prevent a newline. If the newline is written too, then our key name will end with a newline character.
The next example uses Python:

```
mkf = open("/dev/kvs1/_mk", "wb", buffering=0)
incf = open("/dev/kvs1/_inc", "wb", buffering=0)

# Create a new key. You may need to mkf.flush().
mkf.write(b"newkey")

k1 = open("/dev/kvs1/newkey", "r+b", buffering=0)
k1.write(b"324")
print(k1.read())
incf.write(b"newkey")
print(k1.read())

mkf.close()
incf.close()
k1.close()
```

The above example creates a key named *"newkey"*, and increments it once. The files are opened as binary files, so that Python doesn't throw an error when it realises it's not able to seek in them. Buffering is turned off so that the actions are carried out as soon as the reads and writes are executed.
kvfslib in *KVFS*'s Github repository contains some userspace C code to do this, too. *kvfslib* is a userspace library which provides functions for easier usage of *KVFS*.

4 Synchronisation

In *KVFS*, there are several places that synchronisation issues could occur if not specifically dealt with. These are:

- If multiple threads to create a key of the same name, at the same time.
- If multiple threads try to delete the same key at the same time. One of them may end up attempting to remove an inode/dentry that no longer exists.
- If multiple threads try to access the value of a key, be it reading, writing, incrementing or decrementing, at the same time.
- If a thread is trying to delete a key at the same time as another thread trying to access its value.

This will be resolved as following. First, each *KVS* will maintain a mutex to lock key *creation*, and one to lock key *deletion*. Additionally, each key will be associated with a mutex to lock access to its value.

In this way, any thread creating a key will need to hold the *creation* mutex, any thread trying to write or read the value of a key will need to hold the keys's *access* mutex, and any thread trying to delete a key will need to hold both the *deletion* mutex as well as the specific key's *access* mutex. Deletion needing both of these mutexes is to ensure that a key cannot be deleting while its value is being read.

5 Performance

KVFS takes advantage of Linux's virtual filesystem mechanisms to implement a very fast key-value store. Internally, the kernel maintains a hash table of *dentries*. A *dentry* is, in short, a structure which relates *inodes* and together with filenames. Using this hash table allows kernel code to find a file's *inode* with an average time complexity of $\Theta(1)$. The time complexity is also the same for insertion and deletion of keys. The worst case time complexity is, like any other hash table, $O(n)$. Each *inode* within *KVFS* is packaged together with its associated value (where the file's name is the key).

6 Userspace library (kvfslib)

To accompany *KVFS*, we present the userspace library *kvfslib*. We chose to implement this as a C library, for maximum compatibility with other languages; it's easy to call C code from the majority of modern programming languages.
This library provides a thin wrapper around *KVFS*'s filesystem interface. Firstly, it defines a `struct kv_mnt` which stores paths to all of the special files within a specific *KVFS* mount. As a future extension of this library, this struct could also store data to aid in caching, if desired. It could also be used to store `FILE*` objects to keep all the control files for *KVFS* open. This would prevent the need for the files to be opened and close every time the *KVS* is accessed.
As well as providing wrappers around all of the existing *KVFS* functionality, *kvfslib* defines a function to check if a given key exists. This is implemented simply by checking if a file of a given name exists in the root directory of the given mount.
kvfslib can be used in the following way, from C:

```c
#include <stdio.h>
#include <kvfslib.h>

int main() {
    struct kv_mnt kv1 = kvfs_attach("/dev/kvs1");

    key_create(&kv1, "key_1");
    key_create(&kv1, "key_2");

    key_set_s(&kv1, "key_1", "Some data! Very nice.");
    key_set_i(&kv1, "key_2", 32);

    key_inc(&kv1, "key_2");

    kvfs_free(&kv1);
}
```

7 Future Developments

Inspired by existing key-value storage systems such as Redis, it would be nice to implement some new commands in *KVFS*. These could include commands to duplicate a key, set an expiry time for a key, and rename a key.

Furthermore, *KVFS* itself could support more data types. At the moment, values are strings, but in some cases are interpreted as integers. It would be useful to allow list data types as well. If we had lists, we should then also support commands to push, pop, insert, and append to them, as well as sorting them.
A lot of commands that Redis has are not explicitly implemented by *KVFS*, but are easy to achieve anyway. For example, listing available keys is simply iterating the root directory.