A User-Space File-system based on Key-Value stores

Afshin Sabahi Khosroshahi

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2019

Abstract

This is an interim report of my project progress so far. I have included a draft of chapter describing the design and implementation of a POSIX compliant file system using a key-value store as base storage. The key-value store used for my development is RocksDB but the design can allow for other similar key-value stores such as LevelDB to be used as back storage.

Table of Contents

Chapter 1

The file system

The filesystem based on key-value store which I will refer to as KVFS in this document, provides an API to represent and perform operations on directories, inodes and files. This API is a `C++ shared library` which provides `POSIX` compliant file system operations and it is developed using `C++ STL(standard template library)`.

1.1 Local object store

KVFS relies on the key-value store to manage it's space and uses this key-value store to allocate and store objects. A key-value store such as RocksDB is cross platform and is able to operate on different `POSIX` file systems. KVFS packs directories, inodes and files into tables, in this case RocksDB stores sorted logs(SSTables), the local file system sees many fewer larger objects. I use `ext4` as the object store for KVFS in development and test.

The block size threshold is chosen as 4KB, which is the median size of files in desktop workloads.

1.2 Table Schema

KVFS's metadata store aggregates directory entries, inodes attributes and files into one Big table (RocksDB in this case) with a row for each file. To link together the hierarchical structure of the user's namespace. the rows of the table are ordered by a 64/96 bit key consisting of inode number(64 or 32bit depending on the host Operating system) of a file's parent directory and a 32bit hash value of its fielname string(final component of its pathname). The value of a row contains the file's full name and inode attributes, such as inode number, ownership access mode, file size and timestamps(struct `stat` in Linux). For each file, the file's row also contains an inline 4KB of the file's data. To write files with a size bigger than the threshold the files data are aggregated into the same table with a row for each block. The rows are ordered by a 128/160 bit key consisting inode, hash value of file name and a 64bit block number. All enteries in the same directory have rows that share the same prefix inode number of their key. For /tt

readdir() operations, once the inode number of the target directory has been retrieved, a scan sequentially lists all the entries having the direcory's inode number as the first 32/64 bits of their table key. To resolve a single pathname, KVFS starts searching from the root inode, which has a well-known inode number (0). Traversing the user's directory tree involves constructing a search key by concatenating the inode number of current directory with the hash of next component name in the pathname.

1.3 Hard Links

Hard links, as usual, are a special case because two or more rows must have the same inode attributes and data. Whenever KVFS creates the second hard link to a file, it creates a separate row for the file itself, with a null name, and its own inode number as its parent's inode number in the row key. Creating a hard link also modifies the directory entry such that each row naming the file has an attribute indicating the directory entry is a hard link to the file object's inode row.

1.4 Scan Operations Optimisation

KVFS utilises the scan operation provided by RocksDB to implement `readdir()` system call. The scan operation in RocksDB is designed to support iterations over arbitrary key ranges, which may require searching SSTables at each level. In such a case, Bloom filters cannot help to reduce the number of SSTables to search. However in KVFS, `readdir()` only scans keys sharing the common prefix – the inode number of the search directory. For each SSTable, an additional Bloom filter can be maintained, to keep track of all inode numbers that appear as the first 64bit row of keys in SSTable. Before starting an iterator in a SSTable for readdir(), KVFS can first check its Bloom filter to find out it contains any of the desired directory entries, Therefore, unnecessary iterations over SSTables that do not contain any of the requested directory entries can be avoided.

1.5 Inode number Allocation

KVFS uses a global counter for allocating inode numbers. The counter increments when creating a new file or new directory. This global counter is saved in a row with a key called `superblock` to save file system state. Since in my development I use 64bit inode numbers, it will soon not be necessary to recycle the inode number of deleted entries. Coping with operating systems that use 32bit inode numbers may require frequent inode number recycling, a problem beyond the scope of this project and addressed by many file systems. One idea for 32bit systems is to save each freed inode in a array and during allocation an entry can be used from the array. The array can then be saved in a row in the table.

1.6 Locking and Consistency

RocksDB provides atomic insertion of a batch of writes, atomic deletion of a range of range of keys and transactions. The atomic batch write grantees that a sequence of updates to the database are applied in order, and committed to write ahead log automatically. The delete range operation is designed in RocksDB to replace a pattern where user wants to delete a range of keys `[start,end)`. This has advantage of being atomic and is more suitable for performance-sensitive write path. Transactions allow data to be modified concurrently while letting RocksDB handle the conflict checking. The `rename()` operation can be implemented as a batch of two operations: insert the new directory entry and delete the stale entry. For operations like `chmod` and `utime`, since all of an inode's attribute are stored in a single key-value pair, KVFS must read-modify-write attributes atomically. A light weight locking mechanism implemented in the KVFS core layer and together with the RocksDB transactions API ensures correctness under concurrent access.

1.7 Journalling

KVFS relies on the key-value store to achieve journalling. In this case RocksDB has its own write-ahead log that journals all updates to the table. RocksDB can be set to commit the log to disk synchronously or asynchronously. To achieve a consistency guarantee similar to "ordered mode" in Ext4, KVFS forces the key-value store to commit the write-ahead log to disk periodically.

Chapter 2

API structure

In this chapter I will go over how the file system API is implemented and explain the design decisions that are made. As mentioned before this project is a `c++` shared library providing `POSIX` file system operations to the user. A user can either directly use this library to develop their application or use it as a userspace mounted filesystem.

2.0.1 Design and architecture

This software is developed adhering Google's C++ Style Guidelines. The code targets C++11 features. The project uses CMake for builds.

KVFS core is implemented with factory design pattern, this allows to create object without exposing the creation logic to the client and refer to newly created object using a common interface. There is an abstract class called Store which can be used to implement key-value store operations. I have implemented RocksDB operations using this interface. Similar approaches to implement these methods from the interface can be taken for other Key-value stores such as LevelDB. Apart from direct Store operations, there are two in memory LRU caches implemented in KVFS for performance gain purposes. One cache is used for directory entries and another for caching Store's key value pairs. The latter is used instead of directly using the Store to perform operations on inodes.

2.0.2 KVFS library

I have implemented the most commonly used POSIX file system operations, they are listed in the following table.

Table 2.1: Provided API operations

POSIX operation	**KVFS operation**
fopen() — opendir()	Open()
readdir()	ReadDir()
closedir()	CloseDir()
remove() — rmdir()	RemoveDir()
unlink()	Unlink()
mkdir()	MakeDir()
rename()	Rename()
stat() — stat64()	GetStat()
chown()	Chown()
chmod()	Chmod()
access()	Access()
utime()	UpdateTimes()
truncate()	Truncate()
mknod()	MakeNode()
readlink()	ReadLink()
symlink()	SymLink()
link()	Link()