Implementation in Python/FUSE
First attempt at coding

Arkadiusz Bulski

Introduction

This document (or chapter) describes a first attempted implementation of the filesystem. Several features were dropped for the sake of simplicity. Primary goals are to include versioning and snapshots, and reverting. All operations must be atomic and ordered. This internally requires adding space allocation and an extent map, a journal, garbage collection and defragmentation, storing and loading to disk, and LRU cache. Python and FUSE framework were chosen as the purpose of this project is **correctness and readability, not performance**. This implementation may serve as a reference to a future in-kernel high performance implementation.

This document is part of a larger specification, currently edited outside of it. Please refer to https://github.com/arekbulski/Cameleonica/blob/master/documentation/combined.pdf.

1. In-memory filesystem

Objective is a memory only filesystem that supports basic operations like file add/remove/move and read/write/truncate/setcontents and stat/statfs. Partially the purpose is to become acquainted with FUSE and establish how it works. Also, this can be used later for unit testing purposes, as same operation performed on few filesystems results in same state. Also, this code can serve as basis for further phases. Code can run on single thread, no concurrency.

2. Revision history

Objective is to extend previous code to preserve high resolution history of each operation. Only newest views are supported at this point, no reverting or opening previous views. Inodes must only append new entries on every operation. No compaction happens at this point.

3. Cloning

Objective is to extend previous code to support file and directory cloning. Filesystem must support instantaneous clone operation on both files and directories. After cloning, both clones must not share further changes and be independently modifiable and revertible. Multiple views can be open at the same time to different clones.

4. Reverting

Objective is to extend previous code to support reverting files and directories to previous states. Backward and forward reverting must be effectuated by only appending entries. Forward reverting means undoing an undo, back to original revision. No compaction happens at this point as well. Multiple views can be open at the same time to different clones.

5. Snapshots

Objective is to extend previous code to support creating and removing snapshots. Reverting to a snapshot is just the same as reverting to a revision it references. This complicates garbage collection but that is not implemented yet.

6. Read-only views

Objective is to extend previous code to support opening files and directories at earlier revisions in read only mode, also called previous views. A file can be opened concurrently multiple times, as both editable current view and read only previous views.

7. Garbage collection

Objective is to extend previous code with compacting inodes. Since disk space is not yet used, objects can be merely dropped. Compacting process must however work in small batches, as normal operations will be interleaved with compacting operations. Currently open inodes can be withheld from compaction or only partially locked.

8. Free map, and space allocation

Objective is to extend previous code with the free map. A free map is like a buddy allocator, keeping track of free space as extents. Extents must be sector aligned. Adjacent extents must be merged upon deallocation, and free extents must be split on allocation. Should allow for multiple allocations by coalescing requests. Extents must be grouped and divided by segments. The map is to be serializable to and from blobs and diff blobs.

9. Extent map, and inode allocation

Objective is to extend previous code with the extent map. An extent map is a dictionary mapping integer ids to byte ranges (pair offsets). Must support allocation and deallocation and reassignment of ids. The map is to be serializable to and from blobs and diff blobs.

10. Journal, and entry allocation

Objective is to extend previous code with another internal structure, the journal. The journal is a data structure containing same entries as inodes but with additional fields, namely affected inode ids. Must support appending and removal of latest entries, and forward iteration. The journal is to be serializable to and from blobs and diff blobs.

11. Inodes and data

Objective is to extend previous code to enable serialization of inodes and data to and from blobs. Inodes are always made into a single blob, however data chunks can be coalesced into data extents depending on how large or small extents the free map can provide. Note that data is not stored on disk yet.

12. Disk storage

Objective is to extend previous code to store extents to disk over time after their inodes were modified and serialized, or to load extents from disk and parse them when needed. Non-dirty inodes can be dropped in memory immediately after being persisted, without caching. This phase requires adding rings and chains as abstract types, as well as using existing components.

13. LRU cache

Objective is to extend previous code to support caching of inodes and data. In case of metadata, either parsed inodes should be cached or their blobs. In case of data, either entire data extents can be cached or individual data chunks. Cache should occupy up to specified amount of bytes and drop objects when overfilling. Explicit emptying should also be allowed. Objects still in use should also be counted into the cache quota.

14. Defragmentation

Objective is to extend previous code to support reclaiming disk space over time and reorganizing data into sequential-like. This process should run in the background and be combined with the compaction process. Primarily, this process actively counteracts external fragmentation. As inodes keep dropping entries and data extents keep dropping chunks, their blobs get smaller. More importantly, as extents are re-written into new locations their previous copies get unallocated thus leaving holes. Those holes can be used for sporadic allocations but should be removed by moving and splitting other extents. Algorithm used for selecting extents to be moved and their destinations remains to be invented. Secondly, this process counteracts file fragmentation. Files that have been rewritten or written too slowly or non-sequentially will eventually be selected to be reorganized on disk. Algorithm for laying out data chunks have been specified in main document, however selecting files remains to be invented. Also the user can explicitly select files for immediate processing and continuous progress reporting.

15. Preallocation

Objective is to extend previous code to support fallocate operation. If the user choses to grow a file in advance, this space remains unusable for other purposes. Each write that fits and does not overwrite any data is stored explicitly there instead of depending on the allocator. If a segment contains both unused space and chunks overwriting each other, they can all be merged and re-written to a new segment location.

16. Conduit special files

Objective is to extend previous code to support special operations without using ioctl, since FUSE does not properly support those. Opening a special filename opens a virtual file, where writing to it sends a blocking request that returns an error code, and subsequent reading from it returns structured response. For example, online defragmentation of selected files can be both demanded and its progress monitored through special operations.

17. Command-line tools

Objective is to create a CLI application to support special operations from a terminal. Operations can be requested through conduits. This is merely a terminal interface.

18. Python class tools

Objective is to create a Python module extending the os module with file, directory, and special operations. For example, setcontents can ask if this operations is supported and fallback to open truncate write close. File and directory cloning separately can either fallback to deep copy or raise error. Operations can be requested through conduits. This is merely an interface.