Implementation in Python/FUSE
First attempt at coding

Arkadiusz Bulski

Introduction

(This chapter is still being written and is therefore incomplete.)

This document (chapter rather) describes a first attempted implementation of the filesystem. The exact implementation guide is the main specification, and this document only serves as a work breakdown structure. There will be two separate campaigns, and difficult features were moved towards the second half. Python and FUSE framework were chosen as the purpose of this project is **correctness and readability, not performance**. Hopefully, this implementation will be able to reach or surpass competitive benchmarks with Ext4 and Btrfs. This is quite plausible because Python and FUSE overhead should be negligible compared to the disk seeks that this design avoids, and heavy methods can be rewritten in native speed Cython. Even if benchmark results are not beaten, it will be a useful filesystem where features outweigh performance, and as it gains more popularity and a large user base, it will serve as a reference implementation to speed up development of an in-kernel high performance implementation. Implementation effort will result in both the code itself and an extensive documentation of how it works, to be used as a teaching material, both released to open source community to benefit all.

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 add/remove/rename and read/write/truncate/setcontents and stat/statfs. Code can run on single thread with no concurrency, at least until later. Primarily the purpose is to become acquainted with FUSE performance quirks and establish how to obtain best throughput and lowest delays, and to document these findings for other developers as a guide. On public forums there are many complains as to its performance and many posts to the opposite, but no one has made it known what works. Secondly this code will serve as basis for further phases.

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 previous views. This phase adds a chunk map to inode structure, different for regular files and directories and symlinks. Inodes must only append new entries on every operation. No compaction happens at this point.

3. 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. Reverting a parent directory closes sub files for safety reasons.

4. Snapshots

Objective is to extend previous code to support snapshots. Both creating removing and enumerating should be enabled. 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.

5. 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.

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. Multiple views can be open at the same time, 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 work in small batches, as normal operations will be interleaved with compacting operations. Open inodes can be withheld from compaction entirely or partially. Obviously, regular files and directories inodes are processed by different algorithms.

8. Free map, and space allocation

Objective is to extend previous code with an internal structure, 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. Map 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 an internal structure, the extent map. An extent map is a dictionary mapping integer ids to byte ranges (offset pairs). Map 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 an 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 adds rings and chains abstract types.

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 quota.

14. Defragmentation

Objective is to extend previous code to support reclaiming disk space over time and reorganizing data into sequential-like layout. Deframenting process must work in small batches, as normal operations will be interleaved with compacting operations. Primarily, this process actively counteracts external fragmentation. As data extents keep dropping chunks, they become sparse. More importantly, as extents are re-written into new locations their previous copies get unallocated thus leaving holes in their segments. Those holes can be used for sporadic allocations but should be removed by compacting segments. Algorithm used for selecting extents to be moved and their destinations remains to be invented. Secondly, this process counteracts internal 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 defrag and continuously ask for progress report.

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 ioctl. 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. Python class tools

Objective is to create a Python os2 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 can fallback to deep copy. Requesting structure of a sparse file should fallback to reading entire content and finding zeroed areas. Requesting detailed information, like physical offsets on disk or fragmentation status, and other that are not obtainable through the standard stat/statfs operations. This is a programmer interface around a conduit.

18. Command-line tools

Objective is to create a CLI application to support special operations from a terminal. This is a terminal interface around os2 module. Mainly this should produce argparse and pretty printing.

19. Graphical interface tools

Objective is to create a GUI applications to support special operations from a desktop. Nautilus extensions over standalone applications are preferred. This is a GUI interface around the os2 module. Mainly this should produce meaningful information to the end users.

20. Benchmarks

Objective is to create a benchmarking methodology or incorporate existing one. Competitors like Ext4 and Btrfs will go through a passthrough FUSE layer to unbias results affected by context switching and FUSE internals. Benchmarking framework will work by doing same sets of operations on separate filesystems, interleaved, and measuring times of each phase. This does simulate usage over long periods of time, and showing how fragmentation affects long term performance. Comprehensive measurements should be made, with plots and detailed conclusions. Once the benchmarking code has been set up, it can be used to do A/B testing when later adding improvements to the codebase. If competitors were beaten, it will also bring more attention to the entire effort. As a side effect, this also produces a unit testing framework that allows to check if entire filesystem code is correct, by comparing entire state across filesystems.

Introduction cd

This chapter describes the second half of Python/FUSE implementation. Up to this point, the filesystem should be highly useable and stable. The more interesting but also more difficult features are now going to be added to the codebase.