Arkadiusz Bulski

safe cryptographic steganographic advanced filesystem

It aims to guarantee confidentiality, authenticity, plausible deniability, permanent deletion, versioning, snapshots, atomicity, transactions, file and directory cloning, high throughputs, low delays, internal compression, non-transparent compression and hashing, deduplication, serializability, multiple-device replication and tiering.

Internal design includes atomic and ordered operations, diskless fsync, extents, complete inodes, copy on write, segmented log structured disk layout, append only operations, copying garbage collector, complete dictionary and alternative B-tree, rings and chains abstract data structures, file level RAID modes.

https://github.com/arekbulski/Cameleonica

Printed 2017-09-04

Table of contents

Mission Statement
Safe cryptographic steganographic advanced filesystem

Arkadiusz Bulski

Introduction

Cameleonica is a prototype safe cryptographic steganographic advanced filesystem.

This short description expands into a list of features:

It aims to guarantee confidentiality, authenticity, plausible deniability, permanent deletion, versioning, snapshots, atomicity, transactions, file and directory cloning, high throughputs, low delays, internal compression, non-transparent compression and hashing, deduplication, serializability, multiple-device replication and tiering.

These properties are more elaborately explained through definitions and examples:

Filesystem

is a data structure that maps a hierarchy of regular files and directories into a flat storage. Access to the files is governed by rules that make up so called semantics. Most semantics are already well defined by POSIX standard. The fact that open files can be removed and replaced while still being open is a notable example. There are also other, de-facto standards that are built upon POSIX. Linux man-pages define even more fancy semantics. For example, fallocate allows to remove a range of bytes without leaving a hole, stitching together both sides. Another example, creat/open can create anonymous temporary files that are not vulnerable to outside access since they do not have a name to be addressed with. Also temporary files are guaranteed to disappear after interruption. Replacing files is an atomic action, which is often used in modern software as means of ensuring that some consistent version remains on disk in case of a failure. All these semantics are interesting but are not the main selling point. They only build a foundation for a useful general purpose filesystem that is protected from at most interruption.

Cryptographic filesystem

is a filesystem that provides two attributes: confidentiality and authenticity. Confidentiality is a property that easily translates to filesystems. If a filesystem is encrypted, it should not reveal any information about it's file structure, names, sizes, content or usage quota until a valid password is provided. Authenticity is a property that guarantees that the files read now are same as the files written earlier. That is, a valid password must be provided before any changes can be made to the file structure. No file can be moved, truncated, or it's content changed without a valid password. Random or malicious changes to files are not allowed to remain undetected. If bytes being read were modified without a valid password, an error must be returned upon read. It is not acceptable for a read operation to return garbage bytes in this situation.

Steganographic filesystem

is a filesystem that provides more than one file structure using only one backend storage. Every file structure is encrypted with an independent valid password. Steganographic confidentiality is a property that demands that if one valid password is revealed, it does not aid in discovery of any other valid password or even existence thereof. In other words, existence of valid passwords remain concealed on top of their corresponding file structures. Steganographic confidentiality is a stronger notion than cryptographic confidentiality in this respect. Multiple independent valid passwords, or equivalently independent file structures, can exist at same time within a given filesystem. As a matter of contrast, cryptography usually deals with only one password at a time. More importantly, the mere existence of that one password is hardly a secret. In steganography, zero or one or more passwords may exist and their existence is a secret in it's own right.

Safe filesystem

is a filesystem that refuses to become unusable or lose access to already existing files in event of abrupt interruption. Modern filesystems are able to reliably recover from an unexpected power loss or system crash encountered at any point in time. This safety guarantee comes without any extra settings turned on. It is commonly expected from any popular filesystem to handle interruptions gracefully and reliably. However, commonly established semantics of what state is returned to after recovery are far from being acceptable. For example, once a file was opened for writing, usually a set of discrete changes is applied. An interrupted write can be partially successful, leaving content neither in the state before opening, neither in the state expected after a complete write. Often even individual file writes are not atomic. This outcome is unacceptable. Content from before changes started should be available for recovery. Another scenario is when a user copies a file overwriting the destination file. Old destination file gets truncated to zero before new content is written into it, which may take several minutes. After interruption, user can expect old content to be gone while new content is only partially present, ending at undefined position. Even worse, eventual file size does not mean that all bytes up to that offset were persisted. Filesystems can persist file size before the content. This outcome is also unacceptable. Old content from before truncation should be available for recovery. Another scenario is when a program modifies a series of related files as supposedly one operation. Consider for example rotating pictures in a photo album. User could not only expect a rotation of a single image to be atomic, but could also expect the whole album to be processed seemingly at once. After interruption, user could demand the whole album to be reverted to original state. Mentioned problems are not new and can be solved using existing mechanisms. Snapshots are becoming quite popular lately, however they are too cumbersome to be of practical use. They have to be taken manually, often enough, and usually cover a whole volume at once. This is clearly not the right way to go. Versioning infrastructure should retain the state of file structure after every significant change, making save points automatically and transparently. Usable recovery scenario should be easily discoverable for any of mentioned scenarios. User should be able to recover from sets of changes made to sets of files.

Advanced filesystem

is a filesystem with outstanding usability. Usability can be meant as performance (sequential disk speeds, and instant copying), as security (deletions equivalent to permanent wiping), as introspectability and revertability (versioning, snapshots), as utilisation (lower disk usage), as integration (speeding up utilities). Let us briefly look at all these features.

Performance of modern filesystems is getting close to physical limitations of underlying hard-drives. Processing of big files would be expected to happen at high throughputs due to preallocation and runtime detection of a pattern. Processing of small files would be expected to happen at high rates due to log structured layout. To achieve this in the long term, file and free space fragmentation must be actively avoided by both adequate disk layout and regular defragmentation in the background.

Cloning files is found only on Btrfs. Hard-links were invented as means of instantaneous forking of files. This however only works in a shared-state fashion, where subsequent changes are also

shared. It is possible to achieve similar performance characteristics for copying operation (cloning files), resulting in independently writable forks sharing an immutable copy of common data. Cloning files may be used to apply sets of changes to a file as atomic transactions. This operation is so beneficial that Linux even added a new syscall for cloning files.

Cloning directories is not found on any existing filesystem. Given a suitable architecture this should be achievable.

Secure deletion is another area where modern filesystems are lacking. Wiping files is not practical. User has to manually take action, and if he fails to do it right, unwanted evidence remains on disk basically forever. Performance of a wipe operation is usually comparable to writing few times more than the amount of data to be wiped, with current fragmentation. This makes wiping huge files problematic. Reliability is also a problem as filesystems do not always make guarantees whether overwriting is done in-place or copy-on-write. Above that, truncated file cannot be securely wiped as some data blocks are no longer reachable. Filesystems do not track data blocks that were used by a file. File removal could be made quick and reliable using basic cryptography.

Versioning is another functionality that is commonly missing. Users do often enough start modifying their documents without considering that they may later want to revert their changes, merely as a matter of changing their mind. Also, programs do not always apply changes in a safe, atomic manner while the user is not necessarily aware of it. This poses a risk of losing data, one way or another. Risk could be minimized by underlying filesystem by automatically creating a continuous history of changes, an ongoing list of save-points that keep being added in front of the list and being scrubbed away from the end of the list after some time.

Atomicity means all operations are individually atomic, that is preserved entirely or none. Modern filesystems like Ext4 and Btrfs surprisingly do not make most operations atomic, even file writes are usually not atomic. The only exception is rename, which by both POSIX and defacto standards provides some means of atomic replacing of files. Software developers are expected to deliver applications that never fail but the filesystem infrastructure hardly provides any means to that effect. This project aims to make all operations both atomic and ordered, which allows for very simple schemes to provide reliable persistence of both simple and structured data.

Transactions are just atomic sets of atomic operations. There are many file formats like SQLIte that would benefit from this feature. However, despite that modern databases supported transactions for years, modern filesystems do not support transactions at all. Btrfs has experimental support but outright discourages using it, ZFS has support but is not popular on Linux, and other filesytems to my knowledge have no support for transactions. This project aims to provide atomicity above transactions, treating first feature as necessity and second feature as convenience.

Disk utilisation can be increased through transparent compression. Compression can happen transparently, with the user only noticing that he can store files with more total size than hard-disk capacity. User can explicitly disable compression on selected files, although the compressor should notice at some point that gain is negligible and stop trying. Btrfs notably implements transparent compression albeit without runtime skipping on incompressible files.

Integration is a feature where system can process files on behalf of a user space process, but having more capabilities. Filesystem is in a position where it can do more or more efficiently and reliably because it has access to internal information that processes do not have.

Compressed representation of file content can be directly accessible to compressing software making re-compression unnecessary. Current filesystems use compression only internally and do not expose encoded representation to user processes. When a utility compresses a file, a compressed representation is read from disk, decompressed internally by the filesystem, only to be then handed over to the program that will re-compress it again, with perhaps the same algorithm. It would be beneficial for the filesystem to just hand over compressed representation. This could make compression software significantly faster, making it only disk bound.

Hashing can also be done on behalf of common utilities. Filesystem can compute checksums on file content and provide system utilities with result checksums instead of all the data necessary to compute it. Computation can both be done in advance when system is idle or doing scrubbing

or defragmenting, on the fly during a sequential write, or on demand when a user process requests it. Once computed, results can be cached on disk and kept indefinitely for future use. System utilities can be modified to take advantage of filesystem provided computation when available and revert to usual method of reading file otherwise. Filesystem guarantees that cached checksum is correct even if other processes also contributed to it's computation, by concurrent requests. User space sharing of checksums would not be safe because programs could forget to change checksums, or change them out of sync with file itself, or purposefully corrupt it. Furthermore, checksum request can be computed on a particular revision, as if a temporary snapshot was created just for the purpose of checksumming, making sure that checksum is not affected by any writing being done while checksumming is in progress. Concurrent checksumming of a same file would not be possible on user space side, and correctness would not be possible at all.

Disk backup is an important thing in system administration, however making copies of entire partitions is completely unacceptable. First, entire partition has to be read even if only a small fraction contains allocated data. Secondly, if the partition was not zeroed in advance then there is not much gained from compressing the backup image. This cannot be done if we back up a disk that was already used for a longer time. Thirdly, backup image can be prepared in a way that preserves shared structures, without duplicating clones, and also possibly defragments or compresses the data on the fly. For example, a fragmented file can end up in a contiguous extent in the backup image. This functionality is known as send/receive commands on Btrfs.

Deduplication is a feature not found in popular filesystems, most probably because not many people would benefit from it. Given a suitable architecture this could be easily added.

Replication is a feature of multiple-device setups. In the past, cloning and stripping happened on block level using RAID. Today, modern filesystems like ZFS implement these on file level. This means small but important files can be cloned across devices and large but unimportant files can be striped across devices with no regard for backups, all happening within same filesystem. On RAID this would not be possible. Each file has a replication factor property that determines where it gets stored. There are also other issues with RAID that [ZFS] explains, that are solvable when feature is implemented on filesystem rather than block device level.

Tiering is a second feature of multiple-device setups, where some devices are much faster than others. For example, when a filesystem spans both SSDs and HDDs. Some files get accessed much more frequently, or are more important to overall responsiveness, that they should be promoted to SSDs while other files get demoted and stored on slower backends. Even files in same directory could end up on different devices, which was not possible with RAID. Each file has a priority property that determines where it gets stored. Thanks go to Bcachefs author for pointing out the difference between tiering and replication.

Usage scenarios

People with different needs can benefit from using a versatile filesystem. There are several reasons why someone might want to use any given feature. Provided below are possible scenarios of people benefiting from different subsets of functionality:

Graphic designer

He works at an advertising company. Everyday he edits images and photographs provided by his agency. At regular meetings he exchanges materials with his coworkers, where everybody copy source materials from a shared drive and also copy their produced work to their superior's drive. Processing of medium sized files at high rate is beneficial, as it regularly saves time. After morning meeting, he started working on his last assignment. He would like to browse history of his changes to see what remains to be done. Regular snapshots and individual file versioning makes reviewing done work easy. He can now resume his work. While editing an image, his editor crashed. He would like to revert to a state few changes back but the image was saved after they were made and there is no undo possible after editor was closed. Even worse, editor crashed before saving was done, corrupting the only copy of the file. Filesystem keeps a continuous history of recent changes and fine grained undo is possible. He can revert to any state after a successful close operation. After a day full of work, he copies his files onto another drive, overwriting several files. Right after he started copying files power went off. After power was restored, some files were already partially overwritten and thus not usable. Replacing operation triggered a save-point however, and old version of entire file structure was quickly recovered.

Human rights activist

She works for a local newspaper. Country in which they operate is ruled by an oppressive government. Her daily job activities revolve around gathering incoming reports and writing articles to be printed. She drives a lot around the country and encounters random checkpoints. She must protect her sources and cannot allow her notes to be seized by an opportunist militia soldier. If randomly searched, her laptop is encrypted and does not provide access to unauthorized persons. She refuses to decrypt it until a warrant is presented and her agency lawyer arrives. She is briefly questioned about her business and let go. After getting back home she gets detained by a security force and questioned about any involvement with anti-government organizations. She admits having no involvement. She is presented with a warrant to search her computers. She agrees to comply and provides a valid password that decrypts some documents on her laptop. To the auditors it is clear that the provided password is indeed valid and she complied with the order to decrypt her laptop. Her laptop is throughly checked and only expected agency documents are discovered. Her interrogators do not stop accusing her of being a suspected member and keep searching her laptop for incriminating evidence. Indeed, she was regularly in contact with rebel forces and her laptop contains illegally obtained documents. If these documents were found, or even a hint of their existence was found, she would likely be taken to jail. Secret documents are kept on a separate file structure which is unlocked by a different password that she did not disclose or even mention. Filesystem itself does not reveal how many more file structures exist on the hard disk, if any. Ultimately, no evidence is found on her computer showing that she hides any documents and she is cleared of suspicion.

Medical center maintenance staff

He works at a major hospital that processes dozens of patients every day. His job is to maintain a database of medical records of current patients. Regularly he has to delete old records of former patients. Government regulations demand that these records are permanently purged when no longer needed. Medical center also has an obligation to guard privacy of it's patients

and keep their records confidential. If these records would resurface later the center would get fined for breaking regulations or sued by patients for not providing privacy. He can rest assured that deleted files are permanently gone, as the filesystem guarantees it by design. Aside of regular file purging, there is a need to retire some hard-drives that were used for years. He needs to remove any remaining files from them before he can send them for disposal. Again, data needs to be purged from disks permanently or the center would be liable. He can rest assured that quick formatting done on the hard drives destroyed master keys permanently as the filesystem guarantees it. Also strong pass-phrase can be stored on a separate device like a pendrive, so even quick formatting is not needed.

Linux distro maintainer

He works for a company maintaining a linux distro repository. Everyday he compiles and archives whole collections of source code and binaries. When he compiles, scripts often copy files which actually takes no space and no time. This takes away some time from compilation time. After a day worth of work, he sends a big disk image with upgrades. Big files are being copied at high throughput, which again saves time. Afterwards he needs to obtain sha1 checksums. Filesystem computed checksums already during copy operation. When he uses a standard command-line utility to obtain a checksum it gets results immediately from cache.

Computer forensics expert

He works for FBI as a consultant. He is often being sent to crime-scenes to secure evidence. When a computer gets seized his job is to make disk images of confiscated hard-drives. Disk images take a huge amount of space. High throughput is necessary to make copies in acceptable amount of time. Also a fraction of the disk image is full of zero bytes allowing the filesystem to compress some of the data. Hash of the image gets calculated on the fly during the long process of sequential writing. Hash of the image then gets digitally signed and handed over to the court. After this is done, he needs to keep a copy in his possession until further notice. Court later demands evidence to be presented and jury needs to be assured that these copies were not modified after being obtained. The expert can remain calm as he was the only user with a password and the filesystem can guarantee that no one else could modify the files in his possession. If defense asks for a proof that the image presented in court is the same as the image obtained during a search, expert can present a hash computed by the filesystem or the image itself.

Software engineer

He works for a major software vendor. His company has a policy that employees can bring work home only on encrypted devices. After work he took a copy of current project on a company laptop and drove back home. When shopping, his car got burglarized and the company laptop was stolen. Filesystem was encrypted and he can be sure that no company secrets fell into wrong hands. Company executives are relieved that there was no major loss and a new laptop was issued to the employee.

Conceptual Design
Ideas and Observations for a versatile filesystem

Arkadiusz Bulski

Introduction

(This chapter is being expanded and edited, but is entirely readable.)

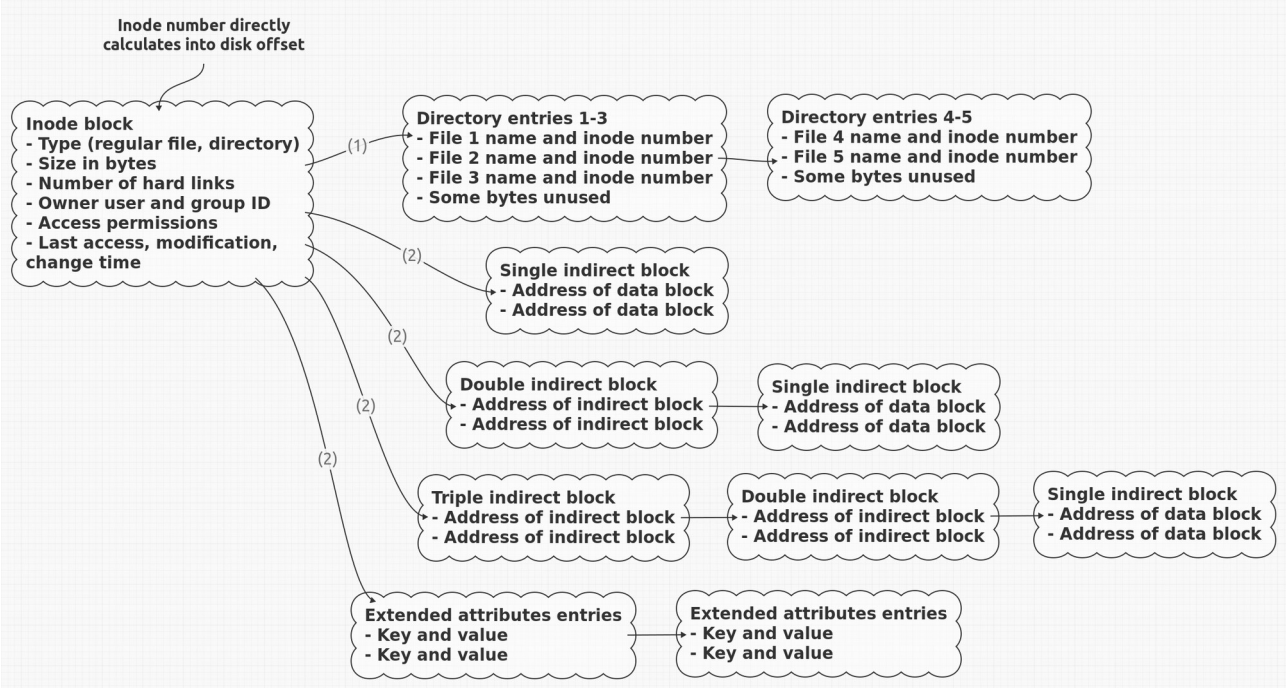
When reading papers that describe filesystem designs, from the golden historical times of Unix development to contemporary times of Linux, one might come to a conclusion that evolution of filesystem designs was very incremental in nature and much effort was put into not making too many changes at once. Ext4 transitioned from blocks to extents, but without different layout and garbage collection the extents are not going to be larger, so fragmentation remains. Ext4 widened offsets to 48 bits, but it is already expected that they will be widened to 64 bits in foreseeable future. Variable length encoding like varint would be better, but without variable sized inodes it does not have any effect. Ext4 still uses journaling which essentially halves disk speed making metadata operations quite slow and data operations unacceptably slow. Copy on write schemes are better, and probably contributed to Btrfs and ZFS success. Track record seems to be showing that radical new designs can be successful but rather when used in new designs from scratch and not when integrated into old existing previous generation designs.

Inodes are not efficient. (OK)

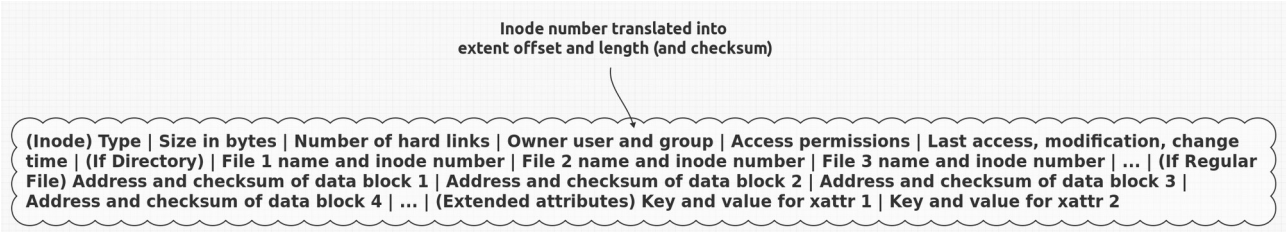
Since the early days, filesystems were designed under a principle that data structures have to be broken down into fixed size blocks. This seems to have been due to the fact that hard disks require read and write operations to be carried out on 512 byte discrete blocks of data (sectors). In traditional designs most operations could be done by keeping only one block in memory at a time. Computers of the past often had little memory, [McKusi84] had maybe 8 MB and so caching complete metadata was not feasible. Furthermore, changing metadata often ended up in writing just one block which might have been reasonable on hard disks that back then had much less throughput and much higher latencies. Furthermore, some hard disks supposedly guarantee that overwriting sectors is atomic [Twee00] which was used for guaranteeing consistency after crash, in Ext2 and its predecessors. All these reasons seem to have contributed to a defacto assumption that data structures need to be broken down into fixed sized blocks. However, hard disks today have much better performance characteristics which invalidates this reasoning.

Filesystem designs so far seem to be based on the assumption that data must be processed in smallest chunks possible (disk sectors or at most memory pages) and the only improvement is to process as small amount of them. It seems that the opposite approach of processing entire datasets either did not gain traction or was not even considered. In traditional designs [OSTEP] like the Inode pointer structure, directories put entries into individual data blocks, and files put pointers into indirect blocks. In modern designs like [Mathur07] [Rodeh12] [ZFS] and [Bcachefs] B-trees are replacing indirect blocks with leaf nodes but data is still being divided into page sized chunks. The issue being described here has not gone away, only changed shape and switched data structures.

Historically, as started in Fast File-System [McKusi84] and carried on in Ext2/Ext3, main block (called inode) would hold most important data and point to further blocks, which would point to actual content (indirect blocks). For a large 1 GB file, assuming 4 KB blocks and 64 bit pointers, at least 512 blocks are needed. This is the so called Inode pointer structure. Only files are being described here for simplicity sake, but the argument is valid for directories as well.



Instead of pointer structure, a *complete inode*, variable size structure could contain all metadata associated with a file. In memory representation may remain decomposed into a tree but on disk format would be a single continuous blob holding entirety of information. Complete inodes are always loaded and stored in one sweep, reallocated on every change. This unfortunately requires an adequate extent allocator and garbage collector, because even large complete inodes always must be able to allocate enough space, but those are discussed in other chapters.



To compare performance, consider both extremes of a spectrum of layouts, where all metadata blocks are either always allocated in one continuous range (called an extent) or are divided into individual blocks. First approach we shall call the *extent approach* and it requires that all blocks are read from a continuous area on disk in one sweep, and after any changes all blocks are stored to disk in one sweep. Last approach we shall call the *block approach*, and it is representative of Orlov allocator in Ext3.

Block approach seems efficient because changing one field requires writing only one block. To the contrary, extent approach requires writing all blocks regardless of amount of changes. **This reasoning is flawed because it is focused only on one update operation itself** and does not account for any subsequent reading, nor accounts for multiple updates in a batch. Following analysis considers amortization of performance over a longer time period. It could be argued

that extent approach is better in every practical usage scenario. To show that, we need to recognize that hard disks (platters) have quite skewed performance characteristics. Representative hard disk is capable of ~150 MB/s of sustained throughput and ~12 ms seek time to random location on average and has capacity 1~6 TB. Charts support these numbers. Simple calculation shows that **reading or writing an extent smaller than 1.8 MB is faster than literally one seek** on average. This observation may seem counter intuitive and perhaps even unbelievable but this math is easy to confirm.

To accommodate for clustered allocation (see Orlov allocator), we will use seek time measured over some small segment (64 MB) instead of entire disk. Experimentally, average seek times measured on SAMSUNG HD154UI were as follows. Code can be found in the repository.

Area size:	Seek time sequential:	Seek time concurrent:
1 MB	0.20 ms	0.01 ms
4 MB	0.44 ms	0.36 ms
16 MB	1.76 ms	1.30 ms
64 MB	**4.53 ms**	3.87 ms
256 MB	6.76 ms	7.59 ms
1 GB	8.61 ms	8.50 ms
4 GB	9.41 ms	9.20 ms
16 GB	10.93 ms	10.78 ms
64 GB	10.99 ms	10.85 ms
256 GB	12.19 ms	12.73 ms
1 TB	16.30 ms	15.72 ms

Above seek times are averages over 100 samples. Formulas used for comparison between block and extent approach can be updated with any seek time from table above. However, large files are going to need more metadata blocks and those will have to come from a larger area, so longer seek times apply. This current value underestimate is accurate only for small files and benefits the block approach.

Extent approach is shown to be better in every practical usage scenario by considering total time spent on all file read/write operations throughout its lifespan instead of just updates. Only metadata blocks are taken into account. File content is excluded from consideration. The file is presumed to exist in final size before experiment for simplicity. In each cycle, file gets opened, some random subset of metadata blocks is accessed, either read or written, then file gets closed and cache dropped. Blocks are assumed to be 4 KB in size. If file is opened read-only then extent approach requires only 1 of the 2 passes, which overestimates it in favor of the block approach.

Reading or writing N blocks takes total (assuming sequential and random pattern, referring to extent and block approach respectively):

$$R_e(N) = 0.00453 + N * 4\,KB/150\,MB = 0.00453 + N * 0.000026$$
$$R_b(N) = N * (0.00453 + 4\,KB/150\,MB) = N * 0.004556$$

Simulation of entire lifespan is based on three variables, N is number of blocks accessed in each cycle, M is number of blocks total, and B is number of cycles. Only a subset of blocks is accessed during a cycle so N<M. Therefore total access time is (extent and block approach respectively):

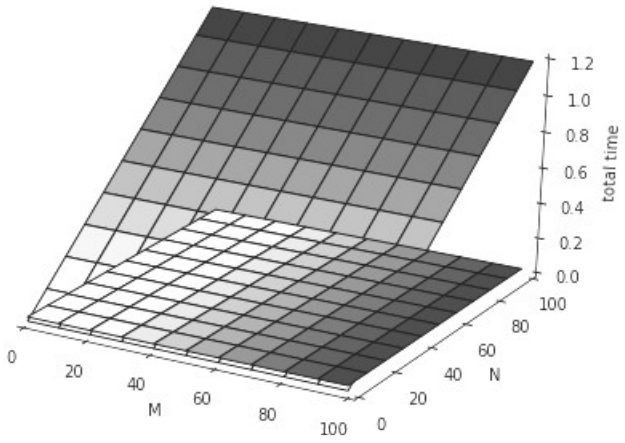
$$T_e(N, M, B) = \sum_{i=1}^{B} R_e(M) + R_e(M)$$

$$T_b(N, M, B) = \sum_{i=1}^{B} R_b(N)$$

There are cases where extent approach loses to block approach in comparison. Total times are compared through an inequality. Number of cycles cancels out.

$$T_b < T_e$$
$$B * R_b(N) < B * 2 * R_e(M)$$
$$M > 87 * N - 174$$
$$N < 2 + 0.0114 * M$$

Results are easily explainable. When individual blocks are stored, time is gained on updates but then more time is lost to seeking when blocks are read during next cycle. In case of R_b we have smaller arguments going into a faster growing function. Plot shows total time of the extent approach (lower) and block approach (upper) for least and most accessed files (N), and least and most sizable files (M).



Last two inequalities show limits on how much metadata can be accessed in each cycle for the block approach to have advantage. Asymptotically, for largest files less than 1 in 87 blocks could be accessed. For smallest files the limit is 2 blocks. Is this really the kind of workload we are aiming to support? Remember that 1 GB file would require at least 512 metadata blocks. It is also important to notice that certain operations such as adding copying moving and deleting files, reading and writing them entirely, and browsing directories necessarily requires all metadata to be read. This was also noted in [Mathur07]. In those cases, the extent approach is a clear winner.

Finally it should be admitted that model above assumes that blocks are allocated individually (no deferred allocation). Probabilistic distribution parameters of such a model are not known. Secondly, disks can reorder outstanding operations to minimize total seek time (NCQ). Model assumes that blocks are accessed without concurrency, one block after another. Some usage scenarios allow to anticipate which blocks will be subsequently accessed. However, concurrent random disk operations will never be faster than one sequential disk operation. That is simply the obtainable minimum. Thirdly, hardware trends suggest that delays and throughputs of HDDs will become more asymmetric in the future [Patterson], benefiting the extent approach.

SSD disks are not being considered in this design. The extent approach is aimed to exploit the asymmetric performance characteristics of hard drives, namely high throughputs but also high delays. This design is counter productive on SSDs.

B-trees are not efficient. (OK)

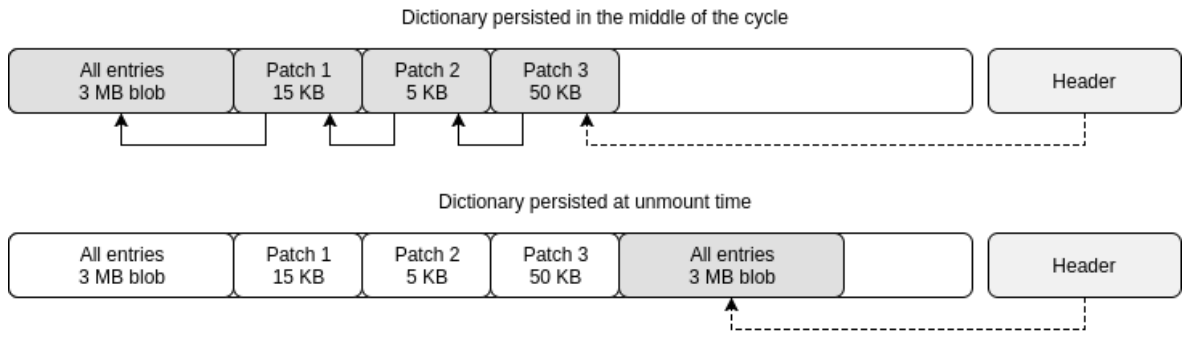
There is a trend among filesystem designs towards using both modified in-place [Mathur07] and copy on write B-trees [Rodeh12] [ZFS] [Bcachefs]. This trend might be easily explained by common expectation that filesystems should be able to handle huge numbers of files, in the billions. B-trees are a good approach if huge amounts of keys are expected. Trees scale asymptotically with logarithmic complexity which means even billions of entries can be stored with only a few disk accesses needed to reach any given entry. However, the operative word is asymptotically. **This asymptotic behavior seems to have mislead everyone**. It indeed would be justified to use B-trees if actual amounts of keys were in the billions but that is not the case in general case. Most desktops hold less than 100'000 [Agraw07] or 200’000 files [Douceur99]. That amount of entries can be stored and accessed more efficiently, thus making B-trees suboptimal.

Secondly, computers of the past had little memory and B-trees need to hold only a few blocks in memory during query and update operations. [McKusi84] had less than 8 MB of memory. However, modern desktops have gigabytes and servers have tens of gigabytes of memory. Conserving memory is not justified anymore. Memory constrained devices like smartphones are not being considered, since they use flash based storage which is also not under consideration.

Thirdly, companies that hold onto billions of files usually make use of distributed filesystems and databases which store data in aggregates. Google database uses SSTables [Bigtable06] and Google filesystem uses something similar [Googlefs03]. **There is no reasonable expectation for cloud nodes to contain so many files**. In fact, given how large SSTables are, those servers might keep even less files than desktops. If this assumption is wrong then that is yet to be confirmed by a publication alike [Agraw07] [Douceur99] which additedly are desktop-only studies.

Consider two approaches described below, in the context of how many entries we expect to hold. As previously, we assume hard disks can sustain ~150 MB/s throughput and ~12 ms seek time on average across disk (and ~4.5 ms across a 64 MB cluster).

First approach, entire dictionary is stored in a single extent. At mount time, entire dictionary is loaded from disk in one sweep, kept in memory in entirety and entire time and regularly stored in entirety to disk as copy on write in one sweep. 1’000’000 entries dictionary would take 23 MB of disk space at most, assuming 64 bit keys and 128 bit values and much less space if varint encoding was employed. This amount of keys should be enough to store 200’000 files. Instead of storing the entire dictionary during each checkpoint, small changesets (diffs) are stored to disk every few seconds and entire dictionary gets stored only every few minutes. If filesystem was unmounted successfully then only final blob containing entire dictionary is loaded at next mount (in one sweep), and if unmount was interrupted then about 60 diffs are loaded. Diffs are stored in same disk segment so they can be loaded in one sweep as well, and above that mounting from interrupted state happens so rarely that it does not really matter. Entire dictionary remains in memory entire time so all key lookups are diskless, and key updates are effectuated during checkpointing. Following diagram shows deallocated space in white.



Second approach, B-tree as the contemporary alternative. Tree height is equivalent to amount

of disk seeks. Assuming the root node is always in memory and nodes have 1480 entries (64 KB with 36 byte values and 64 bit pointers) then 1 seek gives a capacity of 2.2×10^6 entries, 2 seeks give 3.2×10^9 entries, 3 seeks gives 4.7×10^{12} entries, 4 seeks gives 7×10^{15} entries, and so on.

Dictionary approach can be shown to be better in most practical usage scenarios by considering total time spent on all query/update operations over a longer time period. Both representations are presumed to exist before experiment. Some 60 cycles are considered, during which N entries are either queried or updated. This comes from an assumption that after each 5 secs changes are pushed to disk, and in case of the dictionary every 5 mins entire blob is pushed instead of a diff. B-tree nodes are 64 KB in size, and dictionary diffs contain 24 bytes per item.

Reading or writing N entries takes total (assuming sequential and random pattern, referring to dictionary and B-tree approach respectively):

$$R_d(N) = 0.012 + N * 24/150\,MB = 0.012 + N * 0.000000152$$
$$R_b(N) = N * (0.00453 + 64\,KB/150\,MB) = N * 0.00494$$

Simulation of entire super-cycle is based on only one variable, N is number of entries accessed in each cycle, 60 is number of cycles, and 1M is total entries. Therefore total access time is:

$$T_d(N, B) = [\sum_{i=1}^{59} R_d(N)] + R_d(1000000)$$
$$T_b(N, B) = \sum_{i=1}^{60} R_b(N)$$

There are cases where dictionary approach loses to B-tree approach in comparison. Total times are compared through an inequality.

$$T_b < T_d$$
$$60 * R_b(N) < 59 * R_d(N) + R_d(1000000)$$
$$N < 7.5599$$

If amount of entries is less than 1M or varint encoding is used then overestimation would benefit the dictionary in comparison. Results show that B-tree has advantage only if we access less than 7.5 keys per cycle which is 1.5 keys per second. This kind of workload seems unrealistic, and also one would think that if we expect to keep billions of entries then we would also expect to keep processing them a lot. It makes little sense to store so many keys and process so few of them.

This entire argument is basically the same as for complete inodes.

Also it is important to note that the two approaches are not mutually exclusive. When entire dictionary grows over a threshold then entries can be easily migrated into a B-tree, and when tree becomes sparse then entries can be easily migrated to a dictionary. Filesystem can switch between these two representations back and forth over time.

SSD disks are not being considered in this design. Entire dictionary is aimed to exploit asymmetric performance characteristics of hard drives, namely high throughputs but also high delays. This design is counter productive on SSDs.

Block allocation is not efficient.

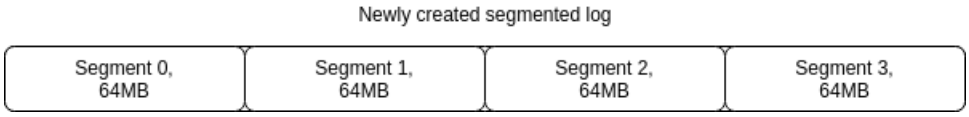
Since the times of Fast File-System and ending at Ext4, disk space was divided into blocks but grouped into clusters. There are few heuristics that were used when assigning blocks to files

that improve performance. Firstly, same file can have blocks allocated sequentially or otherwise in close proximity. However, seeking time over small distances is still quite significant, mostly due to settling phase [Ruemmler94]. Secondly, files that are supposed to be accessed together, when they reside in same directory, can be allocated next to each other. Regardless of these improvements, block allocation entirely seems to be sub-optimal. Orlov allocator in Ext4 is not actively counteracting file or space fragmentation [Aneesh08].

XFS documentation [Sweeney96] points out that fragmentation does not exist on filesystems containing only small files and on filesystems containing only large files. It only exists on filesystems containing both small and large files, because small deallocations must also satisfy large subsequent allocations. Eventually free space becomes fragmented and it becomes very difficult to find large contiguous areas while there are many small free chunks instead.

Log-structured filesystem [Rosenblum91] solves the problem of external fragmentation by allocating disk space from one end of a huge circular log, while regularly moving files from the other end, both defragmenting existing files and compacting deallocated space at same time. This approach has its own problems, that already continuous large files still need to be moved. Although this design does not seem entirely practical, it has laid out grounds for new ideas like segments below or SSDs internal logs.

Segmented log-structure as described here aims to achieve the best of both worlds, to support large allocations and avoid large relocations at garbage collection. Entire disk gets partitioned into segments, which are later used to satisfy extent allocations. Segment size should about ~0.5 second disk time, which is 64 MB. This size is large enough that seek component is negligible, and other processes experience 0.5 second delays at maximum.



Extent allocation is satisfied by appending given amount (in byte resolution) to existing data in a selected segment (selection algorithm is described later). This scheme has obvious benefits of supporting large allocations (extents larger than a segment do not improve performance as seek time component is already negligible), little metadata is kept (one byte-count per segment, plus a set of deallocated extents per segment), and no space is wasted (no tails). Due to deferred allocation, extents are allocated in batches which influences the selection algorithm. To keep many segments empty, algorithm performs greedy bin-packing, putting extents in arbitrary order into half full segments, while minimizing count of segments used (to decrease amount of seeks). Empty segments are only used when otherwise necessary. Few segments would need to be kept in reserve for garbage collection and critical allocations. MFFD bin packing algorithm [Johnson85] would probably be employed.

Extent deallocation is done by marking it as unused and putting it into the appropriate set. During garbage collection, up to few segments are selected (those that contain most unused bytes or unused areas). Existing extents are relocated to other segments, using batch allocation as previously described, and then source segments are marked entirely empty. This process can be combined with defragmentation (files are relocated entirely when their subset gets selected), recompression (files data extents are recompressed using higher ratio algorithm than what was used during initial writing), and scrubbing (checksums are verified and errors are either fixed immediately by pulling from other devices or reported to user).

Fsync is not efficient or effective.

Files stored on computers become increasingly important as businesses and governments store

ever more files of evermore importance on automated systems. At some point in the past, programs started using techniques that would ensure reasonable state after computer was interrupted which did and still does occur quite often. POSIX standard implies an approach that is being used to this day. Windows systems use an identical approach despite not recognizing POSIX. The well established approach replaces a file using following template:

```
int fd = creat("file.new", S_IRUSR|S_IWUSR);
write(fd, buf, len);
fsync(fd);
close(fd);
rename("file.new", "file");
```

This approach has been the expected way of achieving atomic changes to files and caused a lot of pain when people were depending on different guarantees than that of fsync. For example, Ext3 implemented fsync in a naive way that flushed all files to disk and not just the file of interest, which caused it to be slower than perhaps it should have been. At same time, Ext3 implemented non-POSIX behavior of persisting data blocks before related metadata which made the rename safe even without fsync. These two facts made fsync both slow and useless. As a result fsync was no longer needed for correctness and so people started skipping it. Later, ordering behavior was disabled to increase performance and fsync skipping code started corrupting files. This story has been described in LWN post POSIX vs Reality. Also, heavy reliance on fsync led to a famous major performance bug in Firefox.

Described behavior (and few others) is common in modern filesystems but it is not mandated by POSIX standard so applications should not rely on this property being always met. Refer to [Pillai13] for modern filesystem semantics that are common but are not part of the published POSIX standard. Notice that all of the behaviors they describe are met by a filesystem where all operations are both atomic and ordered, as proposed below. Article also describes how SQLite can increase performance by assuming specific filesystem behavior although user has to enable it manually. Also article shows bugs found in LevelDB, a predecessor to SQLite, that corrupted files because the same behavior was assumed but never verified. Both SQLite and LevelDB would be safe on a filesystem where all operations are atomic and ordered. How such a filesystem could be constructed is described below.

Following code is not safe on all filesystems but is representative of applications:

```
int fd = creat("file.new", S_IRUSR|S_IWUSR);
write(fd, buf, len);
fsync(fd);
close(fd);
rename("file.new", "file");
```

Whatever improvements to filesystems will be made in the future, it seems clear that applications will continue to use the approach established by POSIX and filesystems will have to cooperate. Inventing new APIs that ensure safety that also break existing code will not gain traction, no matter how fancy they seem. For example, Btrfs provides an interface to clone a file in zero-time, that allows to apply sets of changes and also changes to huge files in an atomic manner. This was integrated into *cp* command but databases like SQLite do not take advantage of this feature. Another example, Btrfs also provides a way to manage transactions spanning across files. This feature would seem useful if its own documentation have not outright discouraged using it, with warnings of deadlocks.

Below is described a new solution to the problem of ensuring consistency that keeps files correct whether the POSIX compliant approach is used or not while performance is approaching no use of fsync at all.

Consider a filesystem where all operations are both atomic and ordered. Above that fsync calls

are implemented as no-op. Both codes above remain to keep files consistent. This is due to the fact that application developers (usually) use fsync not to persist data immediately but to persist data before metadata, which is what ordering behavior implies. Linux man pages define fsync as a means to persist data immediately but surprisingly this is not what POSIX strictly requires, rather it is just the mainstream interpretation. POSIX first defines fsync (in vague terms) as flushing buffers to a device but then explains in the notes, and explicitly, that if a filesystem can guarantee safety in a different way then that also counts as a valid implementation of fsync. Flushing buffers may be an obvious way to do it but are not the only way. Excerpt from POSIX documentation:

> *(DESCRIPTION) The fsync() function shall request that all data for the open file descriptor named by fildes is to be transferred to the storage device associated with the file described by fildes. The nature of the transfer is implementation-defined. The fsync() function shall not return until the system has completed that action or until an error is detected.*
>
> *(RATIONALE) The fsync() function is intended to force a physical write of data from the buffer cache, and to assure that after a system crash or other failure that all data up to the time of the fsync() call is recorded on the disk. […] It is **explicitly intended that a null implementation is permitted**. This could be valid in the case where the system cannot assure non-volatile storage under any circumstances or **when the system is highly fault-tolerant and the functionality is not required**. In the middle ground between these extremes, fsync() might or might not actually cause data to be written where it is safe from a power failure.*

Here is some historical background:

Persisting ordered operations used to be implemented though ordered persistence. Long time ago blocks were written one at a time (synchronous writes). Operating system buffered changes in memory (in the buffer cache) and was the only party that buffered. When hard disks started buffering writes themselves things got broken. System was no longer in a position to tell which blocks were already persisted on the platter and which were still pending. Immediate solution was to implement *flushing buffers* through so called *queue draining*. System withheld all future writes until disk reported that all outstanding writes completed, that write queue was empty. When flushing buffers occurred quite often this approach basically defeated the purpose of installing buffers in hard disks in the first place.

Later another major overhaul of disks took place. Up to this moment systems have reordered requests themselves. Buffer cache accumulated blocks to be written and system decided, without much help from the disk, in which order should the disk write them. It was more beneficial to process blocks in the order that minimized total time spent on jumping between locations on disk. Unfortunately, system was in no position to determine current orientation of the platter or current position of the head. The system made decisions based on some principle like elevator scheduling, not on exact state of the disk. Everything changed again when disks started to reorder requests themselves, with a technology called NCQ. System was again no longer aware of what is the status of each request. System did not even know which of them will go to the platter first. The solution to ensuring order of writes was again the same, through withholding further writes.

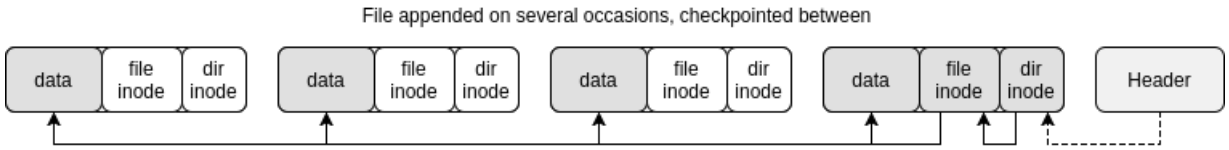
Modern solution comes in form of two related mechanisms: explicit flushes and FUA requests. Flushes are writes that are weakly ordered, demanding that previously scheduled writes are completed before current writes. Previous requests can still be reordered between themselves, and so can future requests. Force Unit Access (FUA) requests can be reordered with any other request, imposing no ordering whatsoever, but disk reports immediately after they landed on the platter. FUA requests are also said to have priority over normal requests. System is no longer responsible for ensuring ordering. Instead the disk is being issued requests marked with these two flags. Topic is discussed on LWN The end of block barriers and Monolight Barriers, Caches, Filesystems.

Consider a filesystem where all operations are atomic and ordered:

(Needs checking language and entire scheme.)

First sector contains the address of last persisted root inode, further sectors hold both data and metadata in any layout desired, be it log-structured or segments or blocks. Each operation puts new extents onto the disk, referencing previous extents. Everything that is being changed is stored copy on write, nothing is overwritten in place for safety reasons. Header points to a complete inode of root directory which points to complete inodes of files within the directory. File inodes point to data extents containing file content.

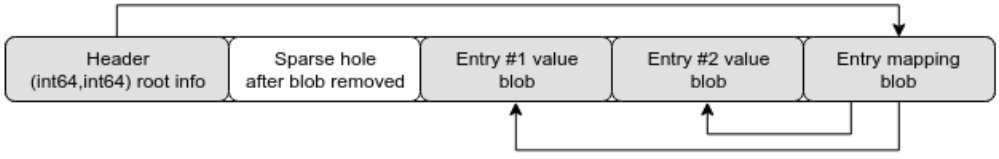
For example, a file write stores the data in a write buffer. After a short period the write buffer gets pushed to disk. File inode is updated with the location of the data extent. File inode then gets pushed to disk. Root directory then gets updated with new location of the file inode and itself gets pushed to disk. After these structures were sent do disk, disk operations get synced and the header is updated to point to latest location of the root directory inode. Note that only the header overwrite needs to be atomic. Other disk writes are neither atomic nor ordered. If the disk can implement a sync without withholding ongoing operations then checkpointing does not affect filesystem operations in any way. In summary, changes become durable in groups between checkpoints. Data can be sent continuously to disk for a long time before a checkpoint, even up to a minute if desired. Only one disk sync is triggered per checkpoint, regardless of fsync calls. No journaling is being employed so no data needs to be written twice. One might argue that inodes are written more than once but that is justified by shortening future reads to one seek anyway. Each inode gets serialized to a continuous extent and all dirty inodes can be laid out next to each other making entire checkpoint into a single sequential disk write followed by a single sync. Following diagram shows a log structured version, each after few operations:



Upgrades can be added. A dictionary that maps ids into disk offsets can be referenced from the header. Directories then associate file names and files associate data with extent ids instead of disk offsets. Changing an inode does not affect the inodes above in a path to the root anymore, instead only a dictionary entry translating the id gets updated. Furthermore, some structures are updated so frequently that patches (diffs) might be stored instead of full structures. That would make checkpointing more lightweight and could lead to more frequent checkpointing.

(Add a diagram for dictionary and inodes, more compact than just inodes.)

Simple key-value store can be implemented easily on an atomic ordered filesystem. First 128 bits hold extent info (which means offset and length) of a dictionary blob. The dictionary maps keys into extent infos of value blobs. Both the dictionary and the values are serialized into binary blobs and stored as contiguous extents. Value blobs are appended atomically to the file, and when database gets committed, dictionary blob also gets appended atomically and then header is overwritten atomically. This way, existing blobs cannot be damaged due to copy on write, header cannot be damaged due to atomicity, and current header is always consistent due to ordering. Reference implementation can be found in the repository.



Even simpler document store can be implemented when setcontents method is available. Only one entry can be stored at once, but it can be an arbitrarily complex data structure. Setcontents is a filesystem operation that truncates the file to zero and writes to it in one combined atomic operation. There is no reference implementation yet.

Cumulative representation is less useful.

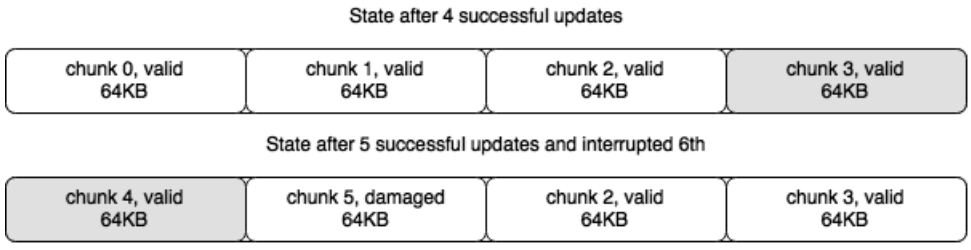
(This needs to be added entirely.)
mention Martin Fowler talk about Event driven architecture

Atomic sector overwrite is not necessary for consistency. (OK)

Many filesystems but also frameworks like SQLite guarantee consistency by depending on the underlying hard drive to overwrite disk sectors atomically. There are some issues with this approach, although so far it seems to have worked just fine. Firstly, this simply moves responsibility down the chain. It is generally a good idea to implement a trait yourself than make a dependency. Secondly, disk manufacturers are known for breaking specifications. Hard disks regularly lie to operating systems about flushing write buffers. This is common knowledge. Thirdly, conversation [Tweedie15] with Stephen Tweedie, coauthor of Ext3, suggests that each particular hardware system would have to be examined carefully for this trait to exist or not, and not just the drive but all components up to the power supply. Therefore, a software solution breaking dependency on hardware should be taken into consideration.

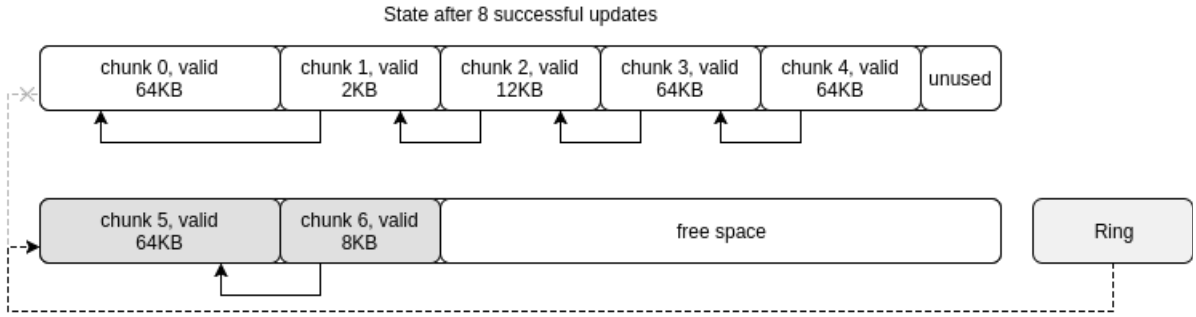
There are at least two new ways to implement atomic changes on disk through software. Both no longer require the disk to support atomic sector writes but they do require that interrupted write does not damage other sectors. This is also how SQLite operates, see [Pillai13]. Note that other problems like phantom writes, corrupted writes, and misplaced writes are not taken into account.

Ring is an extent that is sector aligned and divided into fixed sized chunks that are also sector aligned. Each chunk has a format [checksum|id|length|data]. When updating, an incremented id and data are hashed, then chunk gets stored to disk in one sweep. When reloading, entire extent is read from disk in one sweep, and the chunk with highest id and valid checksum is considered current value. Updating requires last id and its slot to be known, making this mechanism stateful. Note that 2 chunks are enough for most uses but more chunks can also make sense depending on application. Rings can also be used across multiple devices to synchronize newest value, such as in RAID.



Chain is an extent that is sector aligned and contains a concatenation of variable sized chunks that are also sector aligned. Each chunk has a format [checksum|id|length|data]. Often the first chunk contains a base value while further chunks contain changesets but that depends on the application. When updating, an incremented id and other fields are hashed, then stored to disk in

one sweep. Padding can be computed from the length field. When reloading, entire extent gets read from disk in one sweep. Each chunk gets parsed only if previous chunks were parsed successfully, that is had valid id and checksum. Valid chunks are merged (changesets are applied to base value) or simply last value is taken, again depending on application. Updating requires last id and offset to be known, making this mechanism stateful. Note that a chain is defacto *append only*, meaning it can be updated only until becomes full. Then it must be reallocated. Removing chunks is not recommended practice, but they could be removed (sliced off end) by overwriting chunk id/hash fields with zeros. However, I do not foresee any useful scenario for it. Rings can be used to alternate between chains or to keep a list of chains. Chains can also be used across multiple devices to synchronize newest value. In example below, first chain can be dropped from the ring, because patch 5 was superseded by a base blob. Simple alternative would be to always start a chain with base value, so the ring only keeps one chain at any moment.



Multi-view steganographic filesystems are problematic.

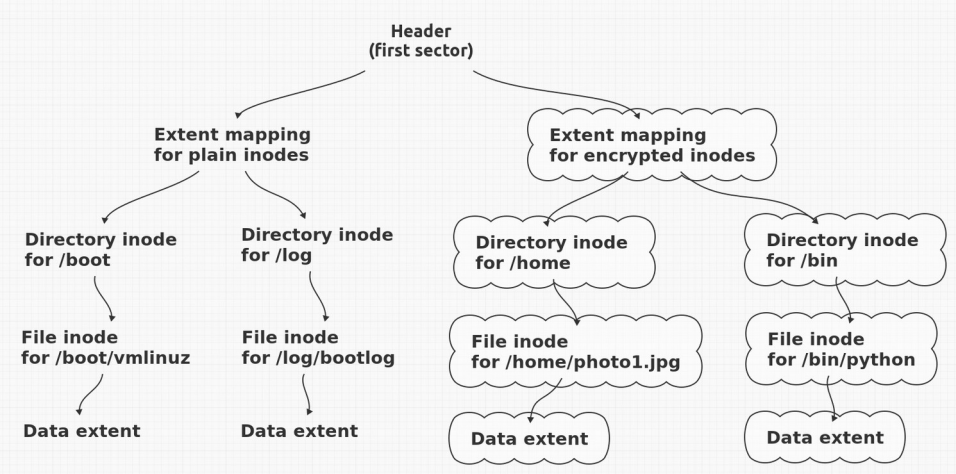
(This needs to be added entirely.)

Encrypting disk blocks is not useful. (OK)

Operating systems today are expected to offer encryption out of the box. In particular, systems are often installed on encrypted partitions and are supposed to boot off them. The problem is that initial code (a bootloader and several files) must be loaded before any decryption can take place. Therefore some amount of files need to be stored in the clear, somewhere.

Currently Linux desktops use at least 2 partitions, unencrypted /boot and encrypted / LUKS partition. There is a reason why one partition setup cannot work. LUKS partitions are encrypted block-wise and provide no means to keep a subset of files unencrypted. Filesystems stacked on top of LUKS are only aware of a block device and have no clue how encryption is added. Only solution to allow a subset of files to remain unencrypted on an otherwise encrypted filesystem is to encrypt selected data structures within a filesystem and not entire block device underneath. Ergo, encryption must happen on data structures level and not on block level.

Consider a partially-encrypted filesystem where encryption is applied to data structures. Each data structure is encrypted individually and data structures form a tree hierarchy where each structure is encrypted by keys stored in a structure above it. Headers contains two pointers, following one pointer allows discovery of all files stored in the clear, and following the second allows discovery of all encrypted files provided the password. Writing to files stored in the clear before providing password might also be possible. Then files like boot logs could also be written. Following diagram shows example data structures on disk (clouds mean extents are encrypted):



Admittedly, there is already a different solution available. Ecryptfs allows to encrypt /home on the fly. Only problem is that encryption is not applied outside /home and performance is low.

Overwriting files is not safe or efficient. (OK)

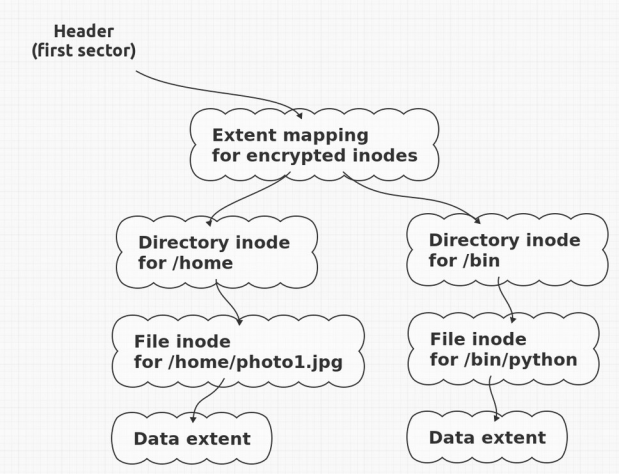
Wiping is an operation that renders a file permanently unreadable and unrecoverable even under scrutiny in lab conditions. Historically and still today, this was achieved by overwriting file content with zeros or random bytes. This approach is definitely not fast or safe.

Regarding performance, throughput of a modern hard drive is ~150 MB/s, and due to fragmentation ~12 ms of seek time per fragment gets added. For large or highly fragmented files an entire overwrite takes a long time. Also magnetic force microscopy [Gutmann96] allows to recover data from disk platters in lab conditions, at least partially, but even after few overwrites. To be really sure data would have to be overwritten up to 35 times.

Regarding safety, files are being overwritten through an interface that provides no means of checking or changing how overwriting files is effectuated. Modern filesystems may use copy on write strategy so overwriting files has no effect on already allocated blocks. This means overwrite may or may not have intended effect depending on circumstances. Also when a file gets truncated the filesystem loses track of deallocated blocks. This means there is no way to find out if there are any remnants left on disk or to wipe them.

Encrypted LUKS partitions are only a partial solution, because they allow to wipe the entire filesystems but not individual files. However, the idea of keeping cryptographic keys in some sort of header for quick and reliable wiping is sound and was investigated before. This approach can be taken further by storing cryptographic keys in several data structures and not only header.

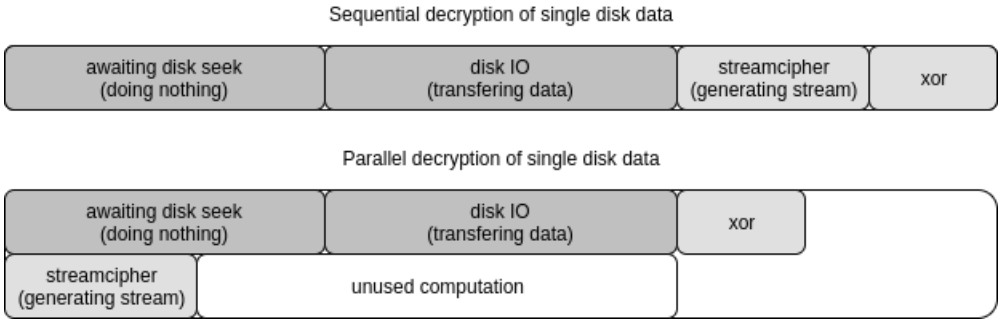
Proposed solution is to encrypt data structures within a filesystem instead of blocks on a block device, to efficiently and safely wipe files, and for that matter any other data structures that the filesystem is divided into, from data extents that compose a file to inodes to entire directories. Consider a filesystem where encryption is applied to data structures. Each structure is encrypted individually and data structures form a tree hierarchy where each structure is encrypted by keys stored in structure above it. Wiping any structure in the tree makes entire branch unrecoverable.



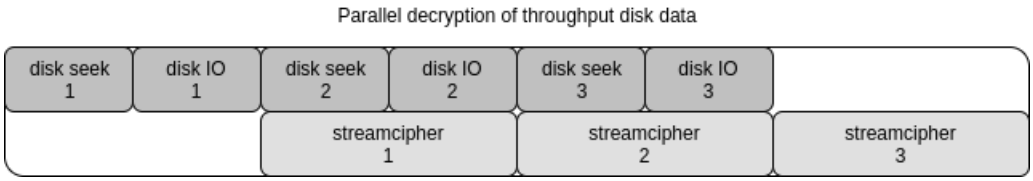
Decryption after reading from disk is slower. (OK)

Disk encryption schemes usually employ block ciphers instead of stream ciphers. Perhaps the reasoning is that since usually there is no authentication employed, block ciphers are at least less malleable than stream ciphers. However, in terms of performance stream ciphers have a certain advantage. Stream decryption can be separated into 2 phases, stream generation and xoring. Perhaps it is counter intuitive but stream generation does not require the data to be present and therefore can happen before device transfers data to memory. This allows for certain amount of parallelism. In this approach, cipher stream fills a second buffer in parallel to disk read, and after both finish (usually disk read takes much longer) those two buffers get xored.

Admittedly, this approach is only applicable to small reads. Since data is small, a second buffer is not costly in terms of memory space, but then not much gain is achieved.



With large reads, entire decryption (both phases) can be done in parallel to disk reads. This approach is already well known, in particular in CUDA community.

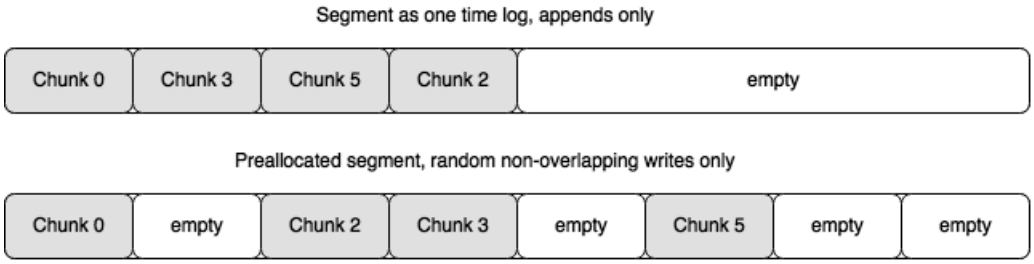


Strategies for data allocation are mutually exclusive.

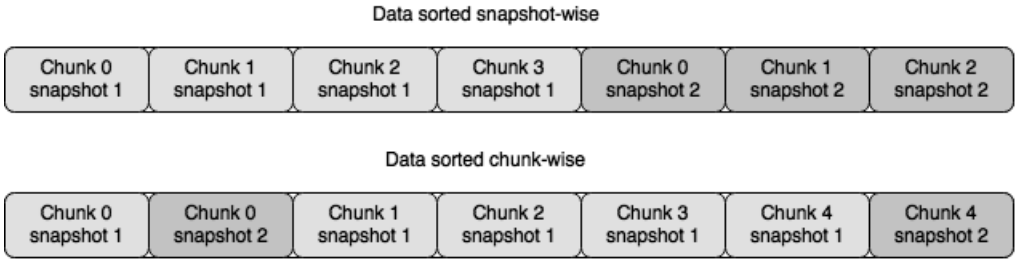
Segmented log structured filesystem is a layout where the entire disk is divided into segments, which are themselves one time logs. Filesystem selects an empty segment and appends data to

it until that segment is full, and then selects a new segment. Concurrently, outdated segments can be read, compacted and defragmented internally, checksummed for integrity, recompressed, and finally stored to another location. Also, compacted segment could be split apart to fit into previously compacted leftover free space. There are however some mutually exclusive strategies how to allocate data within the segments, and whether is it always a good idea to append only to one segment.

Consider the difference between how web browsers and torrent clients download files. Web browsers always download files sequentially. Therefore it is reasonable to lay buffers in same order. However, torrents are downloaded in a different way. Files are downloaded chunk wise in unpredictable order. That is, entire file gets divided into chunks, and each chunk is stored to disk exactly once. In this case, a better strategy is to preallocate the entire segment for this file and store chunks with random writes. There already exists a system interface which applications use to advise the filesystem on expected file size (fallocate). These two strategies have different performance goals. First approach has high throughput initially when storing chunks to disk, but all subsequent sequential file reads will cause random reads from disk. To the contrary, second approach is initially costly due to random writes to disk, but then reading the file is sequential on disk. There does not exist a strategy that has benefits of both. Strategy should be chosen per file. When *fallocate* is called on an empty file, it may be reasonable to assume that entire file will be written once, either sequentially or in random chunks, so allocating dedicated space of the requested size should be beneficial in both cases.

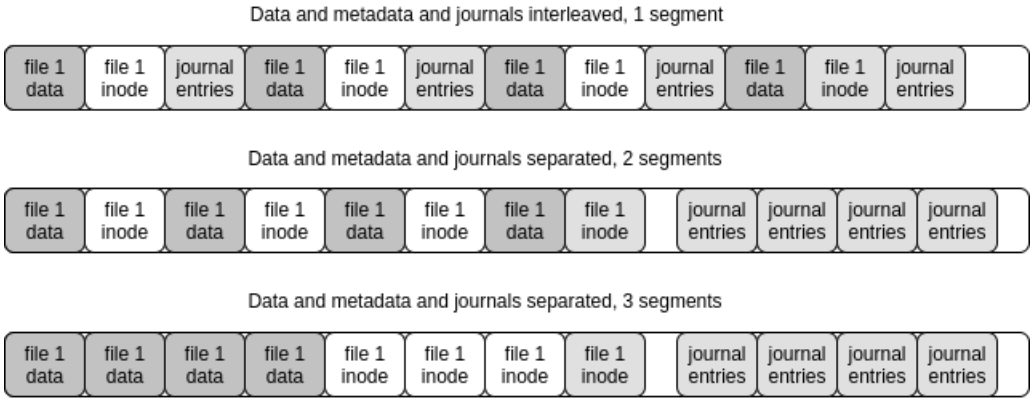


Another issue is how chunks from across different snapshots are laid out within the segment. If file get snapshotted, truncated to zero and overwritten in entirety, then first layout is more beneficial. However, if a file gets snapshotted and overwritten in only a small subset of chunks, then second layout is more beneficial. Which layout is better depends on how much a particular file is going to be overwritten. Applications do not report this to the filesystem, neither there seems to be an easy way to predict it for a particular file. During defragmentation this information is available, so layout can be optimized during defragmentation in anticipation of a read pattern. This topic was investigated by [Rodeh12].



Yet another issue is how journal entries are stored. If groups of journal entries are stored together with data and inodes they refer to, then each checkpointing operation requires only one seek but subsequent reading of journal entries then becomes slower due to caused fragmentation. On the other hand, if journal entries are stored in separate segments and data and perhaps even metadata are stored in another segments, then each checkpointing operation needs 2-3 seeks but both reading both journal and data, if files were large enough, becomes faster due to less fragmentation. First approach may be not as horrible as it looks, if the garbage

collector is going to move these extents around soon enough. Third approach may happen as a result of explicit preallocation.



Conclusion is that there is always a tradeoff between performance during initial writing and performance during subsequent reading, due to fragmentation.

Design decisions for HDDs and SSDs are mutually exclusive.

(This needs to be added entirely.)

FUSE overhead is negligible.

(This needs to be added entirely.)

First few bytes of file should be easier to read. (OK)

When a directory is opened in Nautilus, all files within it are scanned with *file* command that recognizes their mime type. This is a noticeable difference from Windows where filename extensions are used to determine mime type instead. Since *file* command is used often and uses only first few bytes of content, these bytes should be readable more cheaply. Most filesystems would require at least one disk seek (often several) to start reading first data block or extent.

These bytes could be stored within the inode itself. If file gets queried with *stat* first, which happens with Nautilus, then the inode will get cached to serve a subsequent read operation.

Related files are treated independently.

(This needs to be added entirely.)

Behavior is based on assumptions, not history.

Filesystems can gather internal statistics about the files and other data structures, specifically the patters of how they were accessed in the past. For example, filesystem can easily find out which files are accessed immediately and frequently after boot. Then it can move these files and

group them in some segment. At boot time, this segment would be read ahead of time to speed up boot process. If a file is no longer accessed at boot time, then filesystem can decide to move it away to make room for another file. This gain in speed used to be achieved by running some additional application to log accessed files and another application to move files around Ext4 partition, and had to be run again after major changes to system files. There is no reason why this could not be handled by filesystem itself. Modern filesystems like Btrfs and ZFS provide very sophisticated features so a stance that filesystems need to be as simple as possible is no longer valid. Especially that Btrfs source reached ~120K sloc. Nowadays fast boot is mostly achieved by using solid state drives. While SSDs are bringing good performance characteristics, the problem of fast booting from hard drives will remain valid in foreseeable future.

Some patterns can be recognized in real time. For example, there is opportunity to use past patters during file reading and writing. Sequential pattern can be easily detected and lead to written buffers being laid out sequentially on disk or space being reserved in advance for the file in anticipation of further growth. Reading throughput can be increased by steady increase of prefetching, which is already implemented in ZFS. If previously written buffers were compressed with close to no gain, further compression for the file can be temporarily disabled to relieve the CPU.

Also when entire directory tree is being read, filesystem may read ahead other files and also do it concurrently in anticipation of incoming operations. For example, calling stat on entire directory entries list leads to this kind of pattern. Once few of directory entries were accessed, this might trigger an immediate read ahead of few or all of remaining entries.

When a file gets accessed more frequently than others it can be selected for defragmentation or compaction more often than other files or before other files.

Also since filesystem needs to do background scrubbing at some point, statistics can be built to predict peeks and valleys of user initiated disk activity. This way background operations may minimize impact on the foreground, user initiated activity.

Applications and operating systems often use internal statistics to find patterns like UX design efficiency. There is no reason why this could not be applied to filesystems as well.

Behavior is based on assumptions, not requests.

Future access patterns can also be anticipated by following explicit advice from the user or the application. For example, Chromium and Firefox executables always load a bunch of Sqlite files. User could send an explicit advice that defines a set of files (let us call it a load set) to be kept together in one segment or to preload all of them at once whenever one of them is about to be loaded. This would certainly mitigate the well known Firefox performance bug.

Compressing split data is not efficient.

(This needs to be added entirely.)

Block based formats are not flexible. (OK)

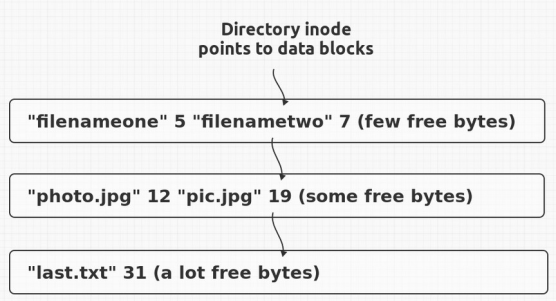
Specifications often define fixed data formats and leave no allowance for changing on-disk layout post release. When a new feature gets added, usually a new but also fixed data format is created in revisioned specs. Conversion usually requires taking filesystem offline for hours to

rewrite all metadata. Even filesystems like Ext4 and Btrfs use fixed size blocks for metadata despite using variable length extents for file content. Solution is to use variable length extents and a data format that prefixes version number in each inode. This approach allows to update format of each inode independently of others, in real time and without taking filesystem offline or redoing all other inodes.

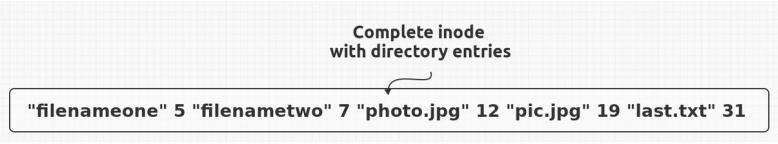
Blocks can store less than extents. (OK)

While previous chapters were considering performance of disk devices, this chapter is focused on compactness. Less bits are needed to store same data in variable length extent than in many individual blocks, leading to gain in disk utilization. The reason why extents are more efficient is that blocks impose rigid boundaries between items. When data is divided into blocks, usually items should not be split between blocks.

First example, in the old days of FFS, directory entries used to be grouped in blocks. File names are much shorter than a single block so several names are put into each block. However, since file names are variable length, then each block has some unpredictable amount of space left unused (a tail).



No doubt this approach was beneficial when memory was very limited. Iterating, querying, adding and removing entries required holding in memory only one block at once. Consider using an extent instead. Items can be laid out next to each other without any blanks in between. The extent could have a tail but that is less than several tails. However if a filesystem allows extents to start and end at byte resolution, then literally no byte could go to waste. Btrfs supports byte resolution extents, although it does use fixed structures for metadata.



Second example, a dictionary that maps integers into arbitrary values can benefit from having all keys available when dictionary is being serialized into a blob. When keys are kept in separate blocks, only few keys may be processed together. Keys can be arbitrarily reordered within the blob because deserialization process does not care. If keys are sorted in increasing order then instead of keys themselves, only their pairwise differences need to be stored. Differences are much smaller than entire keys. Small numbers can be encoded much more efficiently using varint encoding which is explained below. Values may also be varint encoded.

Normally a 64 bit number occupies exactly that much space, 64 bits. Varint encoding divides a number into 7 bit chunks, where every 8th bit is a flag (signaling which chunk is final). Note that even petabyte addresses (<57 bit addresses) still serialize to 64 bits or less.

(expected more paragraphs…)

Bibliography

ZFS: Jeff Bonwick, Bill Moore, ZFS: The Last Word in File Systems, ,
McKusi84: Marshall Kirk Mckusick, William N. Joy, Samuel J. Leffler, Robert S. Fabry, A Fast File System for UNIX, 1984
Twee00: Steven Tweedie, EXT3, Journaling Filesystem, 2000, http://olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html
OSTEP: Remzi H. Arpaci-Dusseau, Andrea C. Arpaci-Dusseau, Operating Systems: Three Easy Pieces, 2015, http://pages.cs.wisc.edu/~remzi/OSTEP/
Mathur07: Avantika Mathur et al., The new ext4 filesystem: current status and future plans, 2007, https://www.kernel.org/doc/ols/2007/ols2007v2-pages-21-34.pdf
Rodeh12: Ohad Rodeh, BTRFS: The Linux B-tree Filesystem, 2012
Bcachefs: Kent Overstreet, Bcachefs guide, 2016, https://bcache.evilpiepirate.org/BcacheGuide/
Agraw07: Nitin Agrawal, William J. Bolosky, John R. Douceur, Jacob R. Lorch, A Five-Year Study of File-System Metadata, 2007
Patterson: David A. Patterson, Kimberly K. Keeton, Hardware Technology Trends and Database Opportunities,
Douceur99: John R. Douceur and William J. Bolosky, A Large-Scale Study of File-System Contents, 1999
Bigtable06: Fay Chang, Bigtable: A Distributed Storage System for Structured Data, 2006
Googlefs03: Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, The Google File System, 2003
Ruemmler94: Chris Ruemmler, John Wilkes, An introduction to disk drive, 1994
Aneesh08: Aneesh Kumar, Mingming Cao, et al, Ext4 block and inode allocator improvements, 2008
Sweeney96: Adam Sweeney, Doug Doucette, Wei Hu, et al, Scalability in the XFS File System, 1996, http://oss.sgi.com/projects/xfs/papers/xfs_usenix/index.html
Rosenblum91: Mendel Rosenblum and John K. Ousterhout, The Design and Implementation of a Log-Structured File System, 1991
Johnson85: David Johnson, Michael Garey, A 71/60 Theorem for Bin Packing, 1985
Pillai13: Thanumalayan Sankaranarayana Pillai, et al, Towards Efficient, Portable Application-Level Consistency, 2013
Tweedie15: Stephen Tweedie, Stephen Tweedie Email.txt, 2015
Gutmann96: Peter Gutmann, Secure Deletion of Data from Magnetic andSolid-State Memory, 1996

Please note that all cited documents are saved in the repository and you do not need to search the web to read them. This includes cited web pages.

Preliminary design
Layout and Methods for a versatile filesystem

Arkadiusz Bulski

Introduction

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

Data structure hierarchy

Storage pool is the highest data structure in hierarchy. Both partitions (also called backends) and filesystems (also called volumes) are each assigned to at most one pool. Each partition is divided into large, fixed size, sector aligned contiguous areas called segments. Default segment size is from 16 to 128 MB. Each segment has a unique id within a pool. When a new backend is added to a pool, it gets divided into segments and the segments are marked as uninitialized. Background process overwrites uninitialized segments with random bytes and only then marks them as unallocated. Optionally, the segments may also remain uninitialized. Whenever a volume requires more segments (new volume has none), the volume requests a free segment from the pool. Volumes can request either kind and be given either kind of segment depending on the pool. When a volume has too many segments or when a volume is destroyed, the segments are returned to the pool and marked as uninitialized.

Volume is the second data structure in hierarchy. A volume keeps track of allocated space within the segments allocated by the pool and locations of extents. Each file, directory, symlink, and groups of data chunks are stored as extents that are assigned unique incremental ids that are translated by the volume into physical disk locations and sizes.

File inode and directory inode are third in data structure hierarchy. Complete file and directory metadata are always contained in one continuous extent and referenced by a parent directory by id. Root directory inode has an id=1 by definition. When a file has many hard links, their parent directories reference the same file inode by id, and the file inode contains a list of parent ids. Symlink inodes are analogous.

Data extents are fourth in data hierarchy. Each file write creates data chunks that are first stored in a write buffer. After a few second window, chunks awaiting persistence are grouped in one continuous data extent and the extent is written to disk at sequential speed.

Extent map is a data structure that maps integer ids into physical locations and sizes on disk. Each extent gets assigned a unique incremental id. When an extent is modified or compacted, id remains the same but the location and size are updated to point to the new copy. Extents are always modified copy on write and the extent map needs to update references atomically.

Free extent map is a different data structure that keeps track of unused space within segments allocated by the pool. When a new extent needs to be persisted, it gets assigned free space from the free map. An empty segment can be split to satisfy an allocation request (and the remainder gets returned to the free map) or a partially filled segment can allocate more space (splitting it again) or a new segment can be requested from the pool (which should be avoided). Sequential allocation from the same segment should be preferred over allocating across entire disk. This leads to both metadata changes and random file writes being persisted at sequential disk speed.

Journal is a data structure that keeps track of changes made to files within a volume with fine granularity down to individual file writes. When user changes a file or directory, the operation gets assigned a volume-wide unique incremental revision number and an entry is added at the end of the journal mentioning what was done and to which extents and at which revision. When the user wants to browse the revision history, the journal is scanned from the end backwards and presented to the user in a well formatted format. The user can select a particular revision to browse the volume as read only or revert the entire volume or just a selected file or directory. Journal is kept as a back-linked list of extents which are chains. Earliest extents can be removed to allow compacting process to reclaim space, while newest extents are used for adding more entries.

Data structure fields

Storage pool

is the top most structure. Every backend (disk) in the pool has the first segment used for 2 rings. First ring keeps a serialized blob of *diskdict* while the second ring contains a serialized blob of *pooldict* structure. Only one partition is needed to mount the pool. Its pooldict has a list of all other partitions. All rings are loaded at pool mount time and checked which one has the newest valid entry. The current pooldict is kept in memory until unmount. Other rings are also updated with the current value to ensure consistency. Whenever pooldict gets persisted, ring in each backend is updated with the new value. All segments in the pool are concatenated into a unified disk space with byte resolution. Pooldict translates disk space addresses into an open descriptor and offset within. New backends are assigned new byte ranges (ranges never being reused) and removed backends make byte ranges permanently inaccessible.

Diskdict contains:

- Checksum covering further fields, not keyed.
- Unique long random id of the backend. This is to ensure that disks are recognized properly when their system paths have changed.

Pooldict contains:

- Checksum covering further fields, not keyed.
- Unique long random id of the pool. This can be used to differentiate between pools.
- List of backends in the pool: their random ids, non unique labels for convenience, disk sizes, segment sizes, byte ranges in unified disk space, and bitmaps of unallocated and uninitialized segments.
- End of existing disk space ranges. This gets incremented when a new backend is added and is never decremented, even when a backend is removed.
- List of volumes in the pool: their random ids, unique names, non unique labels for convenience, extent infos (locations and sizes in unified disk space) of fsdict rings, bitmaps of allocated segments, encryption and authentication keys to fsdict if used, salted hash of an external key if these keys are also encrypted, selection of cryptographic algorithms.

Volume

always stores its extents within the segments allocated by the pool. Main responsibility of a volume is to allocate free space for new extents and to translate extent ids into physical locations on disk. Also it manages a revision history (fine grained record of user made changes) and a list of snapshots (coarse grained history).

Fsdict contains:

- Checksum covering further fields, keyed from pooldict.
- Unique long random id of the volume. This can be attached to backup images and reused when restoring a volume from an image.
- Extent map, either an extent info of the chain containing an entire dictionary with diffs or a list of extent infos with their internal allocation bitmaps for B-tree fixed sized nodes, and a pointer to the root B-tree chunk, and the size of B-tree chunks.
- Root directory extent id. Alternatively it could be defined constant.
- Free extent set, as a list of extent infos. These can be used to satisfy allocation requests before the pool needs to allocate new segments. In memory representation can be more like a buddy allocator but on disk representation is a list.
- Unwiped extent set, as a list of extent infos. On encrypted volumes, inode extents end up on this list before they are passed to the free extent map. On unencrypted volumes, this list can be empty.
- Uncompacted extent set, as a list of extent ids. When the earliest journal entries are discarded, inodes mentioned in those entries are put onto this list. Duplicate ids are discarded.
- Journal, as a list of extent infos of chains containing journal entries. Earliest chain is used for compacting and newest chain is used for adding new entries.
- Earliest and newest revision number of the journal entries. The earliest revision is used to constrain compacting to revisions before the earliest remembered revision. The newest revision plus one is used for next operation.
- Snapshots, as a list of revision numbers and associated non-unique text labels. Version history feature uses the journal while the snapshots feature uses this one.
- Settings that were passed during volume creation or set online, for example, selection of encryption and authentication algorithms, RAID mode for new files, should fsync be ignored, is revision history enabled, amount of bytes for preallocation, amount of bytes to have always allocated from the pool, sector size, etc.
- Encryption and authentication keys and schemes to all mentioned data structures, independently for each external data structure.

Under volume

There are few data structures that are not contained but referenced, such as the journal, and the extent map (which switches between two representations).

Extent map chain chunk content:

- List of entries, each as variable size tuple containing: the key (extent id) and the value (extent info in unified disk space, and extent type). All fields are as varints. First chunk holds entire dictionary, and further chunks hold diffs. Removed entries are represented in diffs as null values assigned to their keys. Extent type can be either file inode, directory inode, symlink inode, or data.

Extent map B-tree chunk content:

- Concatenation of fixed size tuples each containing the key (extent id) and the value (extent info in unified disk space, and extent type). All fields are as uint64. Empty slots are represented by null keys. Extent type can be either file inode, directory inode, symlink inode, or data.
- Concatenation of fixed size pointers to other chunks, each in unified disk space.

Journal chunk content:

- List of variable length tuples, each having one of following formats. First field (revision number as varint) and second field (entry type as varint) are always varints and suggest how further fields are to be parsed. Value of of the second field is in parentheses.
  - (Initialized volume) No fields. This is used for informing the user.
  - (Mounted volume) No fields. This is used for informing the user.
  - (Opened file) Inode id of parent directory inode as varint, file inode id as varint, length of the path as varint, path as UTF8 encoded string.
  - (Created file) Inode id of parent directory inode as varint, length of the path as varint, path as UTF8 encoded string, owner and group as varints, and permissions as int. Since new inode carries same id as the journal entry there is no need to reference it.
  - (Written to file) Inode id as varint, buffer length as varint, offset within content space as varint. Since chunks carry same id as the journal entry there is no need to reference it. Other fields are merely for pretty printing.
  - (Appended to file) (same fields as file write)
  - (Truncated file) Inode id as varint. File's chunk map represents the truncation as one of the chunks but using a different type of entry does not require reading the inode to differentiate between chunks representing writes and truncations.
  - (Moved file) Inode id of source parent directory as varint, inode id of destination parent directory as varint, source path length as varint, source path as UTF8 encoded string, destination path length as varint, destination path as UTF8 encoded string. This also represents a renamed file.
  - (Removed file) File inode id as varint, parent directory inode id as varint, path length as varint, path as UTF8 encoded string.
  - (Created directory) Inode id of parent directory inode as varint, length of the path as varint, path as UTF8 encoded string in that many bytes, owner and group as varints, and permissions as int. Since new inode carries same id as the journal entry there is no need to reference it.
  - (Removed directory) Directory inode id as varint, parent directory inode id as varint, path length as varint, path as UTF8 encoded string, forced removal as byte boolean. Non empty directories can be removed, but for safety it requires root privileges.
  - (Unmounted volume) No fields. This is used for informing the user.

Extents

can be either of the few types: file inode, directory inode, symlink inode, data extent, or a special extent used for journals and extent maps. Depending on the type, the data structure is like the following:

File inode:

- Checksum covering further fields, keyed from any parent directory, parametrized by id.
- Unique volume wide id, as varint. This is the same id that is used by the extent map to translate into byte range. Also it is the revision number of the file creation operation.
- Hard links, as a set of extent ids as varints. These are ids of parent directory inodes that reference this file. Analogously directory inodes keep id of the file assigned to filename.
- Chunk map, as a list of tuples containing: revision number, byte range in content space, extent id of the data extent in which chunk is stored, byte range within that data extent (can be shorter due to compression, equal means no compression, zero means it was a truncation). Some chunks represent file truncation and holes, and do not refer to any extent. Also some chunks refer to ownership or permission changes, or atime mtime ctime changes, or extended attributes changes, and therefore are interpreted differently. Adjacent entries can have same revision number if they represent different information but same operation.

- Ignored revisions, as a list of revision ranges. This is only used when reverting. Journal provides explanations of each revision and snapshots point to particular revisions.
- File size, owner and permissions, first 64 bytes of content, datetimes for atime mtime ctime for the newest revision. Those can be computed from the chunk map for any revision but newest values are cached here.
- Encryption and authentication keys and schemes to be used on data chunks.
- Padding bytes, needed for sector alignment and discarded during parsing.

In memory only:

- Is dirty, as a boolean flag. When set this means the inode needs to be persisted, even if there are no dirty data chunks.
- Dirty chunks, as a dict from chunk ids (which are also revision numbers) to byte buffers holding the data to be persisted. Newest entries in the chunk map refer to these buffers, they do not have on disk locations yet. This collection becomes empty after a checkpoint. Global cache may still keep these chunks for further read operations.
- Frozen at revisions, as a list of integers. When a file is opened at historical revision or is opened for hashing or compressed reading, the inode is withheld from compacting of enumerated revisions forward. Inode can be independently frozen at different revisions.

Directory inode:

- Checksum covering further fields, keyed from any parent directory, parametrized by id. Root directory is keyed from fsdict.
- Unique volume wide id. This is the same id that is used by the extent map to translate into extent info. Also it is the revision number of the dir creation operation.
- Entries map, as a list of tuples containing: revision number, filename, extent id, and type (regular file, dir, symlink). Some entries represent unlinking by associating filenames with null ids. Also some entries refer to ownership or permission change and therefore are interpreted differently.
- Ignored revisions, as a list of revision ranges. This is only used when reverting. Journal provides explanations of each revision and snapshots point to particular revisions.
- Link count (number of entries) and owner and permissions for the newest revision. Those can be computed from entries map for any revision but newest values are cached here.
- Datetimes for atime, mtime, ctime. These are not affected by revisioning or reverting.
- Encryption and authentication keys and schemes to be used on inodes.
- Padding bytes. These bytes are discarded during parsing.

In memory only:

- Is dirty, as a boolean flag. When set this means the inode needs to be persisted, because some fields were changed.
- Frozen at revisions, as a list of integers. When a file is opened at historical revision or is opened for hashing or compressed reading, the inode is withheld from compacting of enumerated revisions forward. Inode can be independently frozen at different revisions.

Symlink inode:

- Checksum covering further fields, keyed from any parent directory, parametrized by id.
- Unique volume wide id. This is the same id that is used by the extent map to translate into extent info. Also it is the revision number of the link creation operation.
- Entries map, as a list of tuples containing: revision number, and target filename. Some entries represent unlinking by assigning null filenames. Also some entries refer to ownership or permission change and therefore are interpreted differently.
- Ignored revisions, as a list of revision ranges. This is only used when reverting. Journal provides explanations of each revision and snapshots point to particular revisions.
- Owner and permissions, and target for the newest revision. Those can be computed from

the chunk map for any revision but newest values are cached here.

- Datetimes for atime, mtime, ctime. These are not affected by revisioning or reverting.
- Padding bytes. These bytes are discarded during parsing.

In memory only:

- Is dirty, as a boolean flag. When set this means the inode needs to be persisted, because some fields were changed.
- Frozen at revisions, as a set of integers. When a file is opened at historical revision or is opened for hashing or compressed reading, the inode is withheld from compacting of enumerated revisions forward. Inode can be independently frozen at different revisions.

Data extent:

- Algorithm selection, as varint. Implies length of checksum field.
- Checksum covering further fields, keyed from the file inode, parametrized by id.
- Unique incremental volume wide id, as varint. This is the same id that is used by the extent map to translate into byte range. Also it is the revision number of the fle write operation.
- Concatenation of variable size data chunks.
- Padding bytes. Extents must be sector aligned. Since the file inode keeps track of internal chunk structure, it does not matter how much bytes were added. Extent map keeps track of entire extent size. Chunks do not need padding.

Data chunk:

- Cheksum algorithm selection, as varint, implying length of checksum field.
- Checksum covering data field, keyed from the file inode, parametrized by id.
- Compression algorithm selection, as varint.
- Data, as bytes. Chunk map keeps track of the entire chunk size.

Some data structures

like diskdict, pooldict, and fsdict are stored in rings and the extent map, and journal entries as stored in chains. These two generic data structures are described below.

Ring extent:

- Concatenation of fixed size chunks.
- Padding bytes. The extent must be sector aligned. Sector size is externally provided.

Ring chunk:

- Checksum covering further fields, can be keyed or parametrized.
- Revision number. This is also the revision of checkpointing operation. When a ring holds few valid chunks (valid checksums) then revision number dictates which value is current. When a value is duplicated across few different rings, this number also dictates which value is current.
- Length of data, in bytes.
- Data blob.

Chain extent:

- Concatenation of variable size chunks.

Chain chunk:

- Checksum covering further fields, can be keyed or parametrized.
- Revision number. This is also the revision of checkpointing operation. All valid chunks (that have valid checksum) from the beginning of the chain are considered part of the current value. When a value is duplicated across few different chains, this number also dictates which chain is most up to date.
- Length of data, in bytes.
- Length of padding, in bytes.
- Data blob.
- Padding bytes. Chunks must be sector aligned. Sector size is provided in fsdict.

User exposed methods

Regardless of implementation, in kernel, through FUSE or Dokan, a GUI application, or a base class in some programming language, the set of methods for manipulating files and directories is the same. Only one of the interfaces above needs to be truly implemented, as other can be mere wrappers.

Mounting a pool: User provides a list of paths to backends. Diskdict and pooldict is read from each listed backend. Newest valid value from all pooldict rings is taken as current. Each diskdict is checked against the random ids contained in the pooldict. This both sorts out which id goes to which path, and checks if all backends are present. If any backened mentioned in the pooldict is missing, mount fails, unless the user explicitly selected an override. Also some fields are checked for consistency, like if any segment was allocated more than once or if byte ranges are not overlapping. This is because this structure cannot be authenticated and malicious changes could be made when offline. Checksumming protects against accidental inconsistency. Pooldict is updated on each backend to current value, in case some rings were stale due to earlier interruption, unless mounted as read only. At this point mount is successful. File descriptors to backends are kept open, and current value of pooldict is kept in memory, until unmount.

Mounting a volume: Fsdict is loaded from all rings, and newest valid value is taken as current value. If the volume has access to many disks, then fsdict is duplicated in several rings, which are discoverable through pooldict. Each fsdict ring is updated to current value, in case some rings were stale due to earlier interruption, unless mounted as read only. If extent map is stored as entire dictionary, its chain is loaded and prepared for appending, if it is stored as B-tree, the root node is loaded. Last journal chain is loaded and prepared for appending. Mount is successful at this point. Current value of fsdict is kept in memory, until unmount.

Creating a file: Absolute path is computed. Root directory inode is loaded. Using its entry map, the inode for second path component is loaded. All inodes for intermediate path components are loaded. If any intermediate component does not exist in its parent directory or its inode has the wrong type, method fails. If the parent directory, which is the last intermediate inode, already contains an entry with the name, method fails. Revision number is recorded and incremented. New file inode is created, the revision number is assigned as the id. Journal entry is added. Inode is retained in memory and considered dirty.

Opening a file: Absolute path is computed. All inodes for path components are loaded. If any component does not exist in its parent directory or its inode has the wrong type, method fails. If atime is to be updated, the inode field is changed and inode is considered dirty. A new handle is assigned, perhaps a new revision number, and the handle is associated with the inode. Journal entry is added. Inode is retained in memory until closed and persisted on disk.

If path is like “/dirname/filename?rev=21” then file is opened as read only, chunk map is filtered so only chunks up to specified revision are visible, stat fields are computed from the chunk map

rather than taken from inode fields directly, and inode becomes frozen at specified revision or completely.

Writing to a file: Open handle is used to find the inode in memory. Revision number is recorded and incremented. New chunk is added to the chunk map, with id set to revision number, buffer size set as the buffer provided, and a pointer to the buffer. Data buffer is stored in memory for further persistence and read operations, and is considered dirty. Overwrites do not discard previous buffers. Other inode fields such as newest file size and mtime are updated.

Closing a file: Open handle is used to find the inode in memory. If file was opened for particular revision, then inode is thawed for that revision. Handle is invalidated. Dirty inode and data are not persisted immediately and remain in memory.

Removing a file: Absolute path is computed. All inodes for path components are loaded. If any component does not exist in its parent directory or its inode has the wrong type, method fails. Revision number is recorded and incremented. Parent directory inode entry map is added a new entry assigning the filename to null id. File inode is not changed. Extent map is not changed. Journal entry is added. Parent directory inode is marked dirty and remains in memory.

Checkpointing: This is an internal operation called whenever few seconds have passed or enough dirty inodes or data accumulated in memory. For each dirty file inode, all dirty data chunks are processed by compressor. Fsdict active options dictate which compressor is used, if any. If chunks are too big, they can be either split before being compressed or let the compressor split the compressed stream. Second alternative is preferred but requires a specialized compressor implementation. Compressed chunks are concatenated into a data extent or few extents depending on available free space, each chunk followed by its checksum. Inode chunk map is updated with new extent id (new revision number for each data extent) and offsets/lengths within extent. Inode is serialized and also allocated disk space by fsdict, preferably right next to the last allocation. Extent map is updated for both extents. Extent info of previous inode blob is returned back to fsdict as unwiped. Dirty journal entries are allocated chunks within last journal chain. If latest journal chain is almost full, entries can be split apart. Both data extents, inode extents, extent map (chain or B-tree) chunks, and journal chunks are sent to disk. Disk is synced, and then fsdict is serialized and stored to its rings. There is no second sync.

Background process: This is an internal operation called every few seconds during peek hours and called continuously during night hours. End of the journal is scanned. If the latest entry is older than active options dictate, this entry is removed and the affected inode gets loaded and inode chunk map has entries dropped accordingly. If that removes one or all chunks from a data extent, the inode fields are updated to remember that. If data extent had all chunks dropped then the extent can be deallocated immediately. Inode id is put into the awaiting compaction dict, with the number of bytes expected to be freed as value. After the journal was finished, a second phase is commenced. Inodes from the awaiting compaction are loaded, and their data extents are moved and compacted, if they contain any dropped chunks. Extents can also be split and joined as part of defragmentation. Chunks can be recompressed if needed. Moved extents can be split or merged if needed. Extents are moved to a new segment. After all inodes were processed, third phase is commenced. Segments having most empty space or most small spaces are selected, loaded from disk, compacted and stored into new continuous segments. Extents can be split or merged if needed.

Conduits: Since FUSE does properly implement ioctl operations, not allowing data structures to be passed as parameters, alternative mechanism is going to be used for special operations. Each open() call having a path “?ioctl?” is going to return a descriptor connected to nothing. Writing to it represents a request with parameters passed as buffer, and blocks until that request

completes. Reading from it returns the result of last operation. Several conduits can be used concurrently.

[awaiting: creating snapshots, browsing revision history, reverting to a revision, transactions, anonymous files, truncation, cloning files, setcontents]

Implementation in Python/FUSE
Work breakdown structure and Schedule

Arkadiusz Bulski

Introduction

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 **features and readability, over 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 still be negligible compared to 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.

It should be noted that descriptions below seem overly simple when in fact they are not. Design is highly experimental and work described includes implicitly much time spent on experimenting.

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.

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.

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.

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.

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.

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.

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 although similar algorithms.

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.

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.

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.

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 extents can be allocated. This phase is easier than previous three, as inodes are mostly done and this is just ensuring their serialization into blobs. Note that

data is not stored on disk yet.

Disk storage

Objective is to extend previous code to store extents to disk over time (checkpointing), and to load extents from disk and parse them on demand. Persisted objects can be immediately dropped in memory, without caching. This phase adds rings and chains abstract types.

LRU cache

Objective is to extend previous code to support caching of inodes and data. In case of metadata, either parsed inodes can be cached or their blobs. In case of data, either entire data extents can be cached or individual data chunks. There is a trade-off here, as blobs are more compact so more can be stored but parsing them takes a non-trivial amount of computation. It would be even possible to make a hybrid model, that caches both types depending on setting or runtime condition. 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. In that respect the cache is also the write-back buffer.

Defragmentation

Objective is to extend previous code to support reclaiming disk space over time and reorganizing data into sequential-like layout. Defragmenting 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.

B-trees

Objective is to add copy on write B-trees as alternative to entire dictionary. This includes both transitions when amount of entries becomes too large for a dictionary or too small for a B-tree. Complete inodes cannot have B-trees alternative due to the fact that all entries are used to rebuild a current state. This phase allows the filesystem to grow infinite amount of files, but not entries in each directory.

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.

Dump utility

Objective is to add a dumping utility that clones a partition to a qcow2 alike container, and includes only metadata. Metadata includes the extent map, allocation map, journal, all inodes.

The purpose is to enable end users to share an actual filesystem for manual debugging after they suffered some catastrophic error, both during and after development. Note that dump is considered a different utility than send. Dump is a low-level sparse copy that should work no matter what happened, and send allows to very efficiently store a backup but assumes no errors on disk. Internally and effectively, those are very different tools.

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.

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. Defragmentation of selected files can be both demanded and its progress monitored. This is a programmer interface around a conduit.

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. Also this should implement proper mkfs utility.

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.

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. This simulates usage over longer periods , and shows 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 also be used to do A/B testing when 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.

Man pages

Objective is to create proper man pages and bash completion for end users. This is going to be the last step before second public announcement and pushing non-beta apt packages.

Public beta release

Objective is to push into the apt repository with beta status. Since core features already exist and testing framework does not signal any problems, it is time to announce beta availability to a broader audience and start hands on testing on a larger user base.

Questions and Answers
Topics that reviewers found unclear

Arkadiusz Bulski

asked by Thistle (pseudonym)
recorded on 2015-04-10

What control will a user have over wiping of files? Wiping of files is not practical? You mean manual wiping?

There are two different actions, regular deletion and permanent deletion. One is reversible and other is permanent. Practicality is meant as both performance and assurance. Wiping in general is a synonym to permanent removal but in this context it means a specific implementation, through overwriting of data in place. Historically, this is how it was done and is still being done. There are two major deficiencies with this approach. First problem, wiping is limited by hard disk performance and file sizes and fragmentation. Hard disks operate usually at maximum of about 120 MB/s assuming no seeks due to fragmentation. Overriding cannot proceed faster than that. And to be really sure, you should do that 35 times (Gutmann method). Second problem, this method is not safe. Data blocks that are no longer reachable by inode, due to truncates for example, cannot be overridden. In general it is not possible to find out which blocks used to belong to a selected file, much less allocate these blocks back to enable overwrite. Filesystems do not keep history of which blocks used to belong to a file. Once a block is deallocated, a track is gone. Also, copy on write mechanism can be applied to data writes, and if so, overwriting has no effect anyway. There is no common interface to find out whether underlying filesystem uses copy on write or not, nor to disable copy on write for a specific file before overwriting. Please also refer next question.

Will there be an option for non-permanent wiping in case a user accidentally wipes a file? What safety measures will be there to prevent accidental complete wiping of content? Or is that a risk users have to take?

There are two distinct actions. Regular deletion (called deletion) is an action that can be taken by non-privileged user. User processes are able to take this action without any user interaction. Therefore this kind of action is inspectable and undoable. Browsing history reveals any deletion and allows for recovery. Permanent deletion (called wiping) is a different action that requires root privileges. This kind of action is partially inspectable, as revision history reveals that some wiping has taken place but file content is not accessible anymore.

History of changes to files means that all versions of the file remain stored, or at least a history that can be viewed. Isn't that impractical in some cases of confidential files? An option to not save a file history?

History of changes is like a list of commits, states saved at some points in time, ordered chronologically. History can be browsed and every revision can be inspected for details but also every revision can be restored back. Confidentiality is not compromised because browsing past versions of files requires same level of access as browsing current version. Anyone having access to past history would necessarily have had access to current file in the first place. When user deletes a file, it will be recoverable as long as it is browsable in history. When user permanently deletes a file (as opposed to regularly deleting), it will not be recoverable even through past versions. Versioning as a feature can also be disabled. If so, regular deletion would make a file immediately inaccessible.

Deleted files are permanently gone, how would that be done? In comparison to saving file history...

Technically there are two ways of ensuring data is permanently gone, either through overriding data itself or overriding encryption keys that were used to encrypt said data. Second approach is much better since keys occupy only few bytes and overriding few bytes takes almost no time. This also erases data blocks no longer assigned to a file, due to truncate operations. Versioning is an independent feature. Different versions are encrypted with same encryption keys, allowing a swift permanent removal of entire history of a given file.

Legal questions, may use of the program encounter difficulties with existing laws in some countries?

Depending on your country of residence, you may be:

- forbidden from using cryptographic products in general
- required to get license or send notification before importing
- required to disclose cryptographic keys to authorities in advance
- asked to enter password and hand over your laptop
- asked if you have any other undisclosed encrypted partitions
- subpoenaed to produce keys or decrypted data itself
- jailed for long time or until your produce encryption keys

Lesson to learn here is that national law can put you in a position where technical solutions do not give you an easy and legal way out. Consider crossing US border with child pornography on your laptop (there was a famous real case). Officer asks you to mount all partitions and then asks if you have any hidden partitions. Note that lying to a federal agent is a criminal offense in US. If you mount the hidden partition, jail, if you lie it does not exist, jail if caught, if you deny to answer, perhaps denial of entry into the country. Plausible deniability feature may let you lie your way out of it but it will not make it any more legal.

asked by Steven Balderrama
recorded on 2015-08-14

On mission document, page 3, I found top 3 things what lacks today in a filesystem: versioning, secure deletion and disk utilisation.

There are also other features but some are so specialized that clearly there will not be many people that will have a use for them. This project aims to deliver a versatile filesystem that will be useful to everybody.

Compression is great, but some files should not be compressed for performance.

Some files are almost impossible to compress, audio video formats are a good example. However, performance does not necessarily have to be negatively impacted. Methods like Gzip are meant to achieve high utilisation with little regard for performance but there are also other, less efficient, methods that are many times faster than a hard disk. Snappy developed by Google can compress ~250 MB/s and decompress ~500 MB/s per core at it's slowest, with worst case input data, and much more with more compressible data, while a hard disk can sustain only about ~120 MB/s transfer. Fragmentation lowers disk side of inequality even further. Direct Memory Access (DMA) allows to transfer data to disk in parallel to code execution so compression and disk operations can be done in parallel. The minimum of two throughputs then becomes a bottleneck. Compression is therefore CPU bound, and not disk bound. Processor load should be fractional and not impact user experience. Also, adaptive compression can be used. If during sequential writing compression gain is close to none, then further writes can skip compression in anticipation of no gain. Also, compression can be postponed till defragmentation so data is written raw during initial write and compressed in the background afterwards.

We sorta talked about performance. Since there is compression and encryption, how will that not hinder performance?

Compare throughput of modern hard disks to throughput of modern processors. Hard disks can sustain ~120 MB/s. Compression like Snappy can handle ~250 MB/s per core at least. Encryption using Salsa20 can handle ~400 MB/s per core. Lightweight hashing using SipHash can handle ~530 MB/s per core. All of the above can be achieved at a fraction of CPU load, and you can always disable some features to lower the CPU load. DMA allows to transfer data to disk in parallel to computation. Therefore, throughput becomes a function of the minimum of the two, which is equal to disk throughput. Refer to previous question.

And yes, versioning. How many times I have seen this, especially programmers who do not utilize software versioning. I have seen it even at my work now. Version control, is that what you mean?

There are two features that may be used to revert changes. Snapshots is a feature that saves state at explicitly chosen points in time, allowing to revert only to specific states that have been prepared in advance. This is the model of how version control software works. Commits are then the equivalent of snapshots. So this was the first feature. Continuous versioning is different than snapshots in that every operation creates a revision. As time progresses, old revisions become automatically compacted. So in summary, snapshots are manually created while versioning is automatic, and snapshots are kept forever while versions are only available for some period of time unless manually retained.

Lastly, you said about having it's own type of recovery. Explain in detail how it will recover or roll back.

Early filesystems like Fast File System and it's descendant Ext2 depended on running a program called *fsck* that scanned all inodes after an interruption. Later filesystems like Ext3 were using journaling to reapply a set of changed blocks. Later filesystems like Btrfs and ZFS started using copy on write and checksums. ZFS also uses intents. The state of the art solution would perhaps comprise of copy on write only. Initial idea of using intents was discarded.

Solution is described in Fsync chapter. Overwriting a file puts the data and the modified inode into new extents, doing copy on write. Their previous copies remain locked until a checkpoint. After all dirty data and inodes up to a specified revision were synced to disk, header is atomically updated to point to new extents, and then old locked extents get finally unallocated. Recovery is a noop since header always references extents that were completely synced to disk. Interrupting before or during a checkpoint only affects space that is considered unallocated anyway.

asked by Robert Węcławski
recorded on 2017-03-03

You wrote that this is a general purpose filesystem, but then that it only works on hard drives (not SSDs). Such a narrow application makes it not useful.

General purpose means that it meets minimum requirements for everyone and surpasses them for some, not that it beats every competitor on every benchmark in every configuration. If I were designing a filesystem for SSDs then I would have made different design decisions, in fact I would have made opposite design decisions. Then the design would be inferior on hard drives. Would you then complain that it does not work on HDDs?

Extents are not beneficial on SSDs, and are shortening their lifespan.

True but SSDs are not being considered, for reasons explained in SSDs chapter.

Extents compaction is supposed to be delayed into the future, but what about servers running 24/7?

Some servers actually do experience peek hours, especially those hosting non-international websites. Then more disk IO will be used on compaction during these non-peek hours. On servers running continuously compaction also runs continuously but at small IO rate. Also, note that preallocation can be used to avoid defragmentation entirely. Then only inodes remain to be compacted, which adds negligible overhead.

I think B-trees would cache better because the entire dictionary will be slow with large amount of files.

The dictionary representation switches to a B-tree representation when the amount of entries grows too large for that representation to be effective. It is used only until it is beneficial, and not longer.