Arkadiusz Bulski
arek.bulski@gmail.com

safe cryptographic steganographic advanced filesystem

Design specification

Features include integrity, versioning, snapshots, atomicity, transactions, confidentiality, authenticity, permanent deletion, plausible deniability, file and directory cloning, internal compression, integration of compression and hashing, serializability, file-level replication and tiering.

Internal design includes complete inodes, complete dictionaries, segmented log-structured disk layout, atomic and ordered operations, diskless fsync, versioning metadata representation, rings and chains atomic abstract data structures, partial and hierarchical encryption, parallel encryption and decryption.

Project sites

https://github.com/arekbulski/Cameleonica

Printed 2022-01-12

Table of contents

Mission Statement
Safe cryptographic steganographic advanced filesystem

Introduction

This document describes an experimental filesystem called “Cameleonica”. Its properties are explained through following definitions and scenarios:

Filesystem

is a data structure that maps a hierarchy of regular files and directories into flat storage of a block device. Access to the files is governed by rules that make up so called semantics. Most semantics are already well defined by POSIX standard and LINUX defacto standard.

The fact that open files can be removed and replaced while still being open is a notable example of those. Linux man-pages define even more fancy semantics.

For example, *fallocate* syscall, in first mode, allows to preallocate some amount of disk space to be used by the file. That space can then be subsequently written to. Preallocated space that was not overwritten remains full of zeros. If the allocation succeed, then subsequent writes cannot fail due to lack of disk space. In second mode, it allows to zero-out a contiguous segment of the content, but without changing file size. In third mode, it allows to collapse a contiguous segment without leaving a hole (zeroed bytes), stitching together both sides. Fourth mode is essentially same as the first mode. In fifth mode, it allows to insert a hole (zero bytes) within the file, while pushing left and right side of it aside. In that respect, fifth mode is similar to third mode.

Another example, *open* syscall can create temporary anonymous files that are not vulnerable to race conditions because they do not allow for outside access, since they do not have a name to be addressed with. Temporary files are guaranteed to disappear after interruption, or after process is ended (if the file was not “named into” existence). Replacing files is an atomic action, which is often used in modern software as a means of ensuring that some consistent version remains on disk in case of a failure.

Another feature would be extended attributes, which are a key-value store associated with a file or directory. Extended attributes can only contain a limited amount of data.

All these semantics are interesting but are not the selling point. They only build a foundation for a useful general purpose filesystem.

Safe filesystem

is a filesystem that refuses to become unusable or lose access to already existing files in event of abrupt interruption, such as system crash or power outage. Modern filesystems are able to reliably recover from interruption encountered at any point in time. This safety guarantee usually 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, a single 1 MB file write is not necessarily going to be stored atomically, but be divided into 4 KB chunks. There are also other failure scenarios, in fact there are too many to count them.

Integrity is a guarantee, that currently read data is what was written down earlier on. In that respect, integrity is very similar to authenticity, except that integrity assumes mere mistakes done by a hard drive and not a sophisticated attack by a formidable adversary. Currently only

Btrfs and ZFS use checksums to achieve data integrity.

Versioning is a 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.

Snapshots are essentially the same as versioning, except snapshots are made more rarely and usually cover entire filesystem. Given some further ideas on how to achieve this, snapshots could be made on directory trees and not entire filesystem. Currently only Btrfs and ZFS seem to support snapshots.

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 LINUX 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 from using it, ZFS has support but is not popular on Linux, and other filesystems to my knowledge have no support for transactions.

First scenario, 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.

Second 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.

Third scenario is when a program modifies a series of related files, that user could perceive as a single batch 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 they usually cover a whole volume at once. This is clearly not the right way to go. Either, versioning infrastructure should retain the state of file and directory structure after every significant change, or a transaction be initiated and then committed for a whole set of files. Or both.

Usable recovery scenario should be easily discoverable for any of mentioned failure scenarios. User should be able to easily recover from sets of changes made to sets of files, assuming that those changes were somehow grouped, for example by a transaction or a snapshot.

Cryptographic filesystem

is a filesystem that provides at least 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 directory structure, file names, file 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 that were written earlier. That is, a valid password must be provided before any meaningful changes can be made to the file structure. No file can be moved or truncated without a valid password. Random or malicious changes to files are not allowed to remain undetected. Unauthorized changes must lead to errors to be reported. In such a situation, a read cannot just return garbage bytes.

Permanent deletion is a property that guarantees that if a file was wiped, that is removed in a specific manner, its content or metadata cannot be meaningfully recovered anymore. This feature is another area where modern filesystems are lacking. Modern filesystems depend on a method of overwriting entire files, sometimes more than once, to achieve an effective wipe. This is wrong on more than one count. Firstly, user has to manually take the specific action and if he fails to do it right, unwanted evidence remains on disk basically forever. Secondly, performance of an overwrite operation is comparable to writing few times (up to 35 times) the size of the file to be wiped, and with current fragmentation. This makes wiping huge files very problematic. Thirdly, reliability is also a problem as filesystems do not always make guarantees whether overwriting is done in-place or copy-on-write. And finally fourth point, also concerning reliability, is that truncated file cannot be securely wiped as some data blocks are no longer reachable. Filesystems do not ever track data blocks that were used by a file in the past. More so, to get rid of those blocks as well, the entire free space would need to be overwritten as well. In summary, overwrite approach is just wrong. On cryptographic filesystems, permanent deletion can be easily achieved through cascaded use of encryption keys. Overwriting just the key itself is sufficient to wipe any data that was ever related to a file, which is both quick and reliable. Note that there are experimental ways of recovering overwritten data using magnetic force microscopy, however this paper assumes that this method is infeasible, see [Gutmann96].

Steganographic filesystem

is a filesystem that provides steganographic confidentiality.

A steganographic filesystem stores more than one file structure using only one storage. Every file structure is encrypted with an independent valid password. We can assume that passwords create a linear progression, meaning every password reveals those earlier ones.

Steganographic confidentiality (also called just steganography, also called plausible deniability) is a property that if one valid password is revealed, it does not aid in discovery of any other valid password further in the chain of passwords, nor even imply the existence thereof. In other words, existence of further 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, more passwords may exist and even their existence is a secret in it's own right.

The only known example to me of such a filesystem is Rubberhose, created by Julian Assange et al which never gained traction, and I do not know if it was ever completed, see [Dreyfus2000].

This feature is probably the most difficult in the entire project. Further ideas are needed.

Advanced filesystem

is a filesystem with outstanding usability. Usability can be essentially whatever else that was not covered in previous sections, and can be meant for example as performance (cloning files and directories), as utilisation (lower disk usage), and as integration (speeding up other utilities). Let us briefly look at all these features.

Cloning files is also related to performance, and is currently found only on Btrfs. A long time ago, 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 single file as atomic transactions. This operation is so beneficial that Linux has now syscalls for cloning files.

Cloning directories is not found on any existing filesystem. Given a suitable copy-on-write architecture this could be achievable. Further ideas are needed.

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 could be allowed to explicitly disable compression on selected files, although the compressor should notice at some point that gain is negligible and stop trying to compress incompressible files. Btrfs and ZFS both implement transparent compression, although it may be turned off by default.

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.

Integration of compression: compressed representation of a 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 an 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 even 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 and not CPU bound like it is now.

Integration of hashing: 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 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 hashing requests. User space sharing of hashes would not be safe because programs could forget to change checksums, or change them out of sync with file itself, or purposefully corrupt it.

Serializability (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 backup a disk that was already used for a long 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 and ZFS.

Replication is a feature of multiple-device setups. In the past, cloning and stripping happened on block level using RAID modes 1 and 0. Today, modern filesystems like ZFS implement these on file level. This means that small but important files can be cloned across devices while large and unimportant files can be striped across devices with no regard for backups, all happening within same filesystem. On RAID this would not be possible, where either everything would be cloned or everything would be stripped. 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 level rather than block 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 devices. 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 all 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. 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 and saving 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 the entire directory was quickly recovered.

Human rights activist

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

Admittedly this usage scenario was more or less taken from Rubberhose documentation.

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 due to cloning. This takes away some time from compilation time. After a day worth of work, he sends a big disk image with upgrades. Afterwards he needs to obtain sha1 checksums. Filesystem computed checksums already during a 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. 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

Introduction

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 would not have any effect. Ext4 still uses journaling which essentially halves disk speed making metadata operations quite slow and data operations unacceptably slow. This paper does postulate that radical new designs can be successful but only when used in new designs from scratch rather than when incrementally integrated into existing designs.

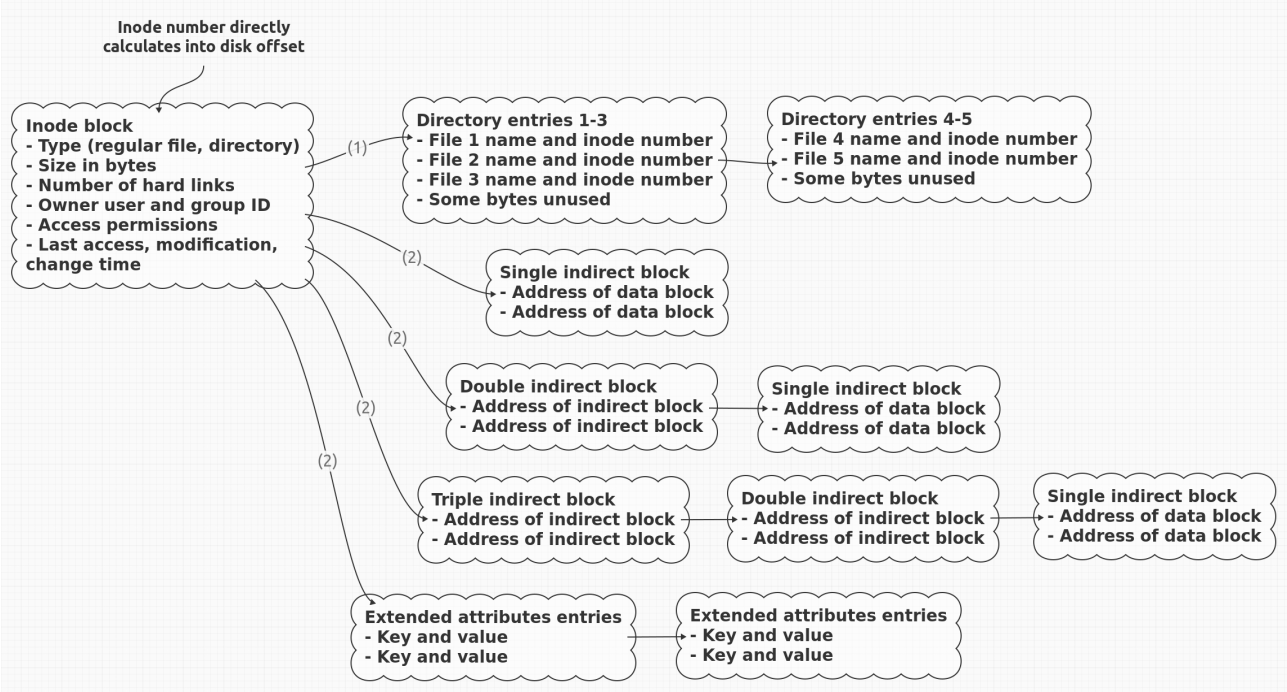
Inodes are not efficient.

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 of RAM 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, see [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, meaning they have more throughput compared to seek delays, 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 at least, and memory pages at most) and the only possible 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] [Overstreet21] 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 at all, it only changed shape and switched data structures.

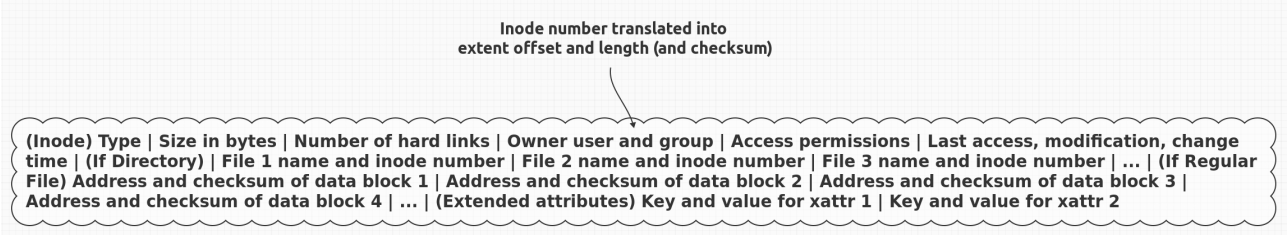
Historically, as started in Fast File-System [McKusi84] and carried on in Ext2/Ext3, main block (called inode) would hold most important data and point to further blocks, which would point to actual content (indirect blocks). For a large 1 GB file, assuming 4 KB blocks and 64 bit pointers, at

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 just the same.



The indirect block approach seems to be optimal when computation and memory costs far outweigh the disk seeking and transfer costs, meaning that file operations are CPU or memory bound and not disk bound. On HDDs at least, the opposite is true. Please note that this might not hold true for SSDs. This paper is concerned with HDDs and only those.

Instead of pointer structure, a complete inode, variable size data 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.



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 amortisation 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 of 1~6 TB. 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 and throughputs measured on SAMSUNG HD154UI (1.5 TB) were as follows. Disk performance measuring code can be found in the repository.

Area size:	Seek time sequential:	Seek time concurrent:
1 MB	0.00 ms	1.87 ms
4 MB	0.00 ms	0.44 ms
16 MB	0.00 ms	1.64 ms
64 MB	**5.13 ms**	7.57 ms
256 MB	8.03 ms	9.38 ms
1 GB	8.96 ms	9.89 ms
4 GB	9.94 ms	10.97 ms
16 GB	11.20 ms	14.26 ms
64 GB	11.66 ms	14.01 ms
256 GB	14.14 ms	17.07 ms
1 TB	19.34 ms	21.18 ms

Buffer size:	Random throughput:	Sequential throughput:
1 MB	24.92 MB/s	101.41 MB/s
2 MB	27.79 MB/s	102.51 MB/s
4 MB	43.98 MB/s	95.53 MB/s
8 MB	56.98 MB/s	98.13 MB/s
16 MB	64.96 MB/s	93.92 MB/s
32 MB	67.09 MB/s	99.17 MB/s
64 MB	**78.04 MB/s**	98.40 MB/s
128 MB	66.75 MB/s	100.35 MB/s
256 MB	88.24 MB/s	100.73 MB/s

Above seek times are averages either over 5 seconds or 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 in read-only mode then extent approach requires only 1 of the 2 passes, which overestimates it in favor of the block approach by a lot.

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

$$R_e(N) = 0.00513 + N * 4\,KB/78.04\,MBps = 0.00513 + N * 0.00005$$
$$R_b(N) = N * (0.00513 + 4\,KB/78.04\,MBps) = N * 0.00518$$

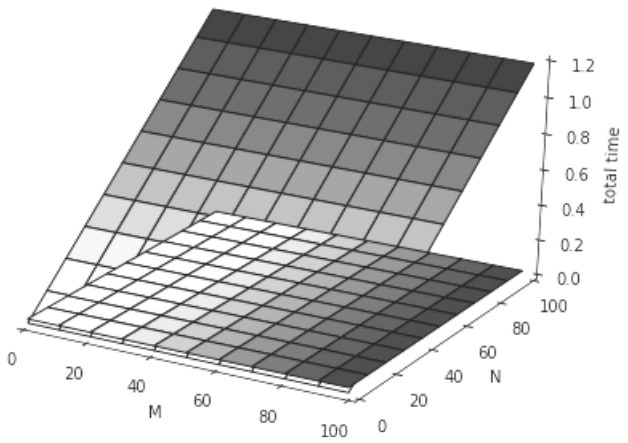
Simulation of entire lifespan is based on three variables, N is number of blocks accessed in each cycle, M is number of blocks total, and B is number of cycles. Only a subset of blocks is accessed during a cycle so N<M. Therefore total access time is (extent and block approach respectively):

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

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

$$T_b < T_e$$
$$B * R_b(N) < B * 2 * R_e(M)$$
$$N * 0.00518 < 2 * 0.00513 + M * 0.00005$$
$$M > N * 103.6 - 205.2$$
$$N < 1.98 + M * 0.00965$$

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 103 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, deleting files, reading and writing files 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.

B-trees are not efficient.

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, going 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.

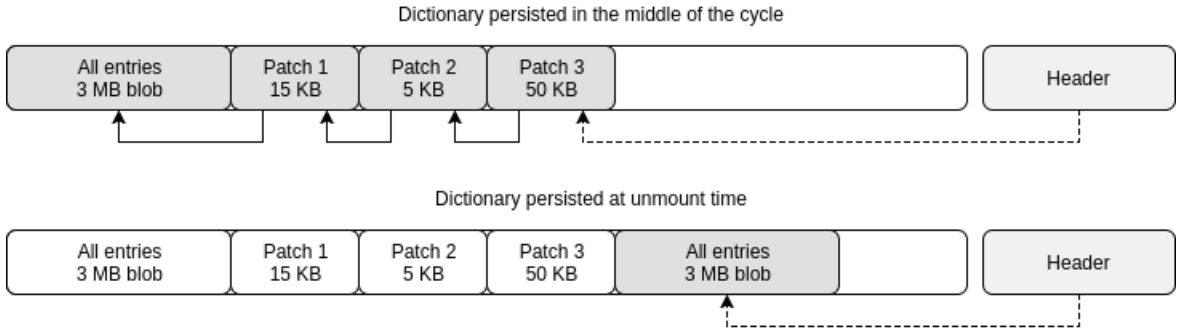
Secondly, computers of the past had little memory and B-trees need to hold only a few blocks in memory during query and update operations. [McKusi84] had less than 8 MB of memory. However, modern desktops have gigabytes and servers have tens of gigabytes of memory. Conserving memory is not justified 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 neither desktops nor server 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] or [Douceur99] which admittedly are desktop-only studies.

Consider two approaches described below, in the context of how many entries we expect to hold and process. As in the previous chapter, we assume hard disks can sustain ~78.04 MB/s throughput and ~5.13 ms seek time on average across 64 MB disk area.

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. If varint encoding was employed, the amount to be written on each cycle and super-cycle would be much less, this

underestimation benefits the B-tree approach. This amount of keys should be enough to store 200'000 files. Furthermore, instead of storing entire dictionary during each checkpoint, small changesets (diffs) are stored to disk every five seconds (a checkpoint) and entire dictionary gets stored only every five minutes (a super-cycle). 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. Above that mounting from interrupted state happens so rarely that it does not really matter anyway. 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 is shown to be better in every practical usage scenario by considering total time spent on all query/update operations over a longer time period (a super-cycle). Both representations are presumed to exist before experiment. Some 60 checkpoints are considered, during which N entries are either queried or updated. This comes from an assumption that after each 5 seconds 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):

$$R_d(N) = 0.00513 + N * 24/78.04\,MBps = 0.00513 + N * 0.000000293$$
$$R_b(N) = N * (0.00513 + 64\,KB/78.04\,MBps) = N * 0.00593$$

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 (dictionary and B-tree approach respectively):

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

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

$$T_b < T_d$$
$$60 * R_b(N) < 59 * R_d(N) + R_d(1000000)$$
$$N < 1.68867$$

Results show that B-tree has advantage only if we access less than 1.6 keys per five second checkpoint which is 0.33 keys per second. This kind of workload seems totally unrealistic and useless. One would think that if we expect to keep billions of entries then we would also expect to keep processing them a lot.

This entire argument is basically the same as for complete inodes. This includes the fact that SSDs performance characteristics are not being considered.

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

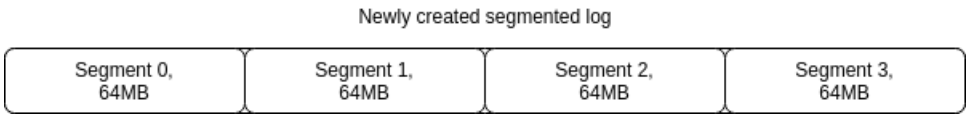
Block allocation is not efficient.

Since the times of Fast File System and ending at Ext4, disk space was divided into fixed sized blocks but grouped into clusters. There are few heuristics that were used when assigning blocks to files that improved performance. Firstly, same file can have blocks allocated sequentially or otherwise in close proximity. However, seek time over small distances is still quite significant, mostly due to settling phase [Ruemmler94]. Secondly, files that are supposed to be accessed together, like when they reside in same directory, can be allocated in same cluster. Regardless of these improvements, block allocation in entirety seems to be suboptimal. 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 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 free space at same time. This approach has its own problems, namely that already continuous large files still need to be moved. Although this design does not seem entirely practical, it has laid out grounds for segmented log-structure below and notably SSD internal layouts.

Segmented log-structure as described here aims to achieve the best of both worlds, to support large allocations and avoid large relocations during garbage collection. Entire disk gets partitioned into segments, which are later used to satisfy extent allocations. Segment size should be chosen experimentally but probably 16 MB or 64 MB.



Extent allocation is done in batches. Filesystem awaits until either more than a segment in total is requested or more than 5 seconds have passed, whichever happens first. If possible, the entire batch gets allocated a single joint extent from the smallest adequate free extent possible. This approach maintains as many entirely free segments as possible. If that is not doable, the batch is decomposed and each extent is allocated individually. If needed, the reserved space can be used for that purpose. If a file gets preallocated a lot of space at once, then all but the tail are given entire segments. The tail can be allocated on a partially occupied segment if possible. All extents

and segments should be either 512 byte sector or 4096 byte block aligned for performance and safety reasons.

In the background, there are 4 processes that can happen at the same time: Firstly, files can be defragmented by moving all data to (1) entirely empty segments, for the multiple of a segment size amount of data and to (2) a partially empty segment, for the tail and metadata. Secondly, free space can be defragmented by moving existing extents away from segments that have most count of empty extents in them. Thirdly, compression can be applied to data that was initially written without or with a lesser compression method. Fourthly, scrubbing can be applied to both data and metadata checksums. If a soft error is detected, the available good replicas are copied over and wrong data gets deallocated. If a hard error is detected, then wrong data gets deallocated and a warning message gets displayed.

Same as in previous chapters, SSD performance characteristics are not considered.

Admittedly, this chapter has no proofs about its performance. It requires further study. Lacking any theoretical arguments, one could implement the filesystem and then simulate identical workload on several filesystems and measure comparative performance. Something to maybe look forward to.

Fsync is not efficient or effective.

POSIX standard implies (not defines) means of ensuring consistency of files. Windows systems use an identical approach despite not recognizing POSIX in its entirety. The overall approach is to write a new entire version into a second file, then atomically replace one file with the other. The problem is that, as more or less explained in rename and fsync man pages, rename is an atomic operation only with respect to parent directory entry, not to writing file buffers. This is why there must be an fsync before the rename. This has an unfortunate side-effect that rename must wait until the data was persisted on disk, which takes a considerable amount of time. Notice that fsync is used by developers not to persist data immediately but to ensure the order of persistence, that data is on disk before related metadata.

This approach has been the expected way of achieving atomic changes to files and caused a lot of pain and confusion when developers were depending on different guarantees than those of POSIX. For example, Ext3 implemented fsync in a way that flushed all files to disk and not just the file of interest which caused fsync to be much slower than necessary. At same time, Ext3 implemented non-POSIX behavior of persisting data blocks before persisting related metadata which made 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 developers stopped using it. Later, ordering behavior was disabled in Ext3 to increase performance, and some applications started corrupting files. This story has been described in LWN post POSIX vs Reality.

On side note, heavy reliance on fsync led to a famous major performance bug in Firefox. This has lead developers to a twisted dilemma, give up on correctness (on power-down) or give up on performance.

Some filesystems possess this behavior (data gets persisted before metadata) but since this is not mandated by POSIX, developers should be very careful and not depend on this behavior. Refer to [Pillai13] for modern filesystem semantics that are common but not part of POSIX standard. Notice that all behaviors they describe are met by a filesystem where all operations are both atomic and ordered, as proposed below. Also the article shows bugs found in LevelDB, a predecessor to SQLite, that corrupted files because the same ordering 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.

Whatever filesystems of the future will employ to ensure consistency after power-down, it will have to both provide correctness and performance to applications, and be compatible with code using (or overusing) fsync. History shows that developers are not keen to use new APIs even for that purpose, and therefore it is the filesystems that must meet the demand, not applications. For example, Btrfs provides an ioctl interface to clone a file in zero-time, that would allow to apply sets of changes to single files and also small changes to large files in an atomic manner. No frameworks or applications, as far as I am aware, take advantage of this feature in any way. Another example, Btrfs provides an ioctl interface to manage transactions spanning across files. This feature is even less used, if that expression even makes sense considering the first feature is essentially not used by anyone. Btrfs own documentation outright discourages from using the transactions feature with warnings of kernel deadlocks.

Below is described a proposed solution, that keeps files correct even when applications do not use fsync, and where performance is close to not using disk syncs at all.

Consider a filesystem where all operations are both atomic and ordered, that is with respect to all other operations, and fsync is no-op. Both applications using and skipping fsync 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, or more precisely to make rename atomic and ordered with respect to all previous operations.

Interestingly, Linux man page defines fsync as the method to persist data immediately but surprisingly this is not a strict requirement but rather a mainstream interpretation. Man page first says that fsync should flush buffers to the device, but then it explains in the notes, and explicitly, that if filesystem can guarantee safety in some other way, other than flushing buffers, then that is also an acceptable implementation. 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. System 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

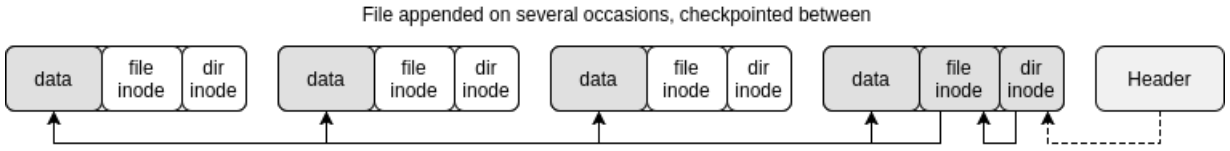
locations on disk. Unfortunately, system was in no position to determine current orientation of the platter or current position of the head. The system made decisions based on some principle like elevator scheduling, not on the exact state of the disk. Everything changed again when disks started to reorder requests themselves, with a technology called NCQ. System was again no longer aware of what was the status of each write request. System did not even know which write request will go to the platter first. The solution to ensure some 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 operations demanding that previously scheduled writes are completed before future writes. Previous requests can still be reordered between themselves, and so can future requests. Flushes only prohibit the disk from reordering writes across the boundary. Force Unit Access (FUA) are disk writes that impose no ordering whatsoever, but the 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 Barriers, Caches, Filesystems.

Consider an implementation of a filesystem where all operations are atomic and ordered:

First sector contains the address of last persisted root inode, further sectors hold both data and metadata in any layout desired, only that copy-on-write writes and atomic sector overwrites are possible. If the atomic data structures from a later chapter are used, the atomic sector writes are not needed either. 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 content.

For example, a file write stores data in the cache buffer. During checkpointing operation, the data from the write buffer gets allocated an extent and pushed to disk. File inode gets updated with the location of the data extent, then gets pushed to disk. Root directory inode then gets updated with new location of the file inode and itself gets pushed to disk. After these structures were sent do the disk, the disk get synced and then header gets updated to point to the location of the root directory inode. Note that only the header overwrite needs to be atomic and ordered, and the header can be a single sector. Other disk writes are neither atomic nor ordered. If the disk can implement a barrier between storing data structures and updating header, using explicit flushes for example, that is without withholding ongoing operations, then checkpointing does not affect filesystem performance in any significant way.

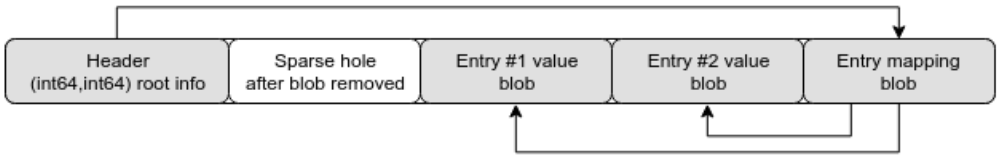


Scheme can be upgraded through adding indirection. A dictionary that maps extent ids into disk offsets can be referenced from the header. Directories then associate file names with extent ids rather than disk offsets. Changing an inode does not affect the inodes above in a path to the root anymore, instead only a single dictionary entry gets updated. See complete dictionaries chapter.

Recovery after a power-down can surprisingly be zero-time. Regardless of any interruptions, the header always points to data structures that were ordered before it and were fully persisted.

Simple safe key-value store can be implemented easily on an atomic ordered filesystem. First 128 bits (the header) hold offset and length of a dictionary blob. The dictionary maps keys into offset-lengths of value blobs inside the file. When the store gets committed, value blobs are appended copy-on-write to the file, then dictionary blob also gets appended copy-on-write, and

then the header gets overwritten atomically. This way, existing value and dictionary blobs cannot be damaged due to copy-on-write appends, header cannot be damaged due to atomicity, and header is always consistent due to ordering. Reference implementation is in the repository.



Cumulative representation is not effective.

In the past, before the feature of snapshots was invented, filesystems only stored the current (final) state of files. To implement snapshots, filesystems had to choose between two mutually exclusive designs. First approach is to store entire state of the file for each snapshot. This has a detrimental effect when a huge file has a lot of snapshots but very little gets changed between each snapshot. On the other hand, if a file gets overwritten entirely between every snapshot then the filesystem makes good use of this representation. Second approach would be to store differences (diffs) between each snapshot.

Proposed approach would be to take the second approach to an extreme. The complete inode would keep on record metadata of each particular file operation performed on the file, including checksums that would allow to check each data integrity, since its creation up to now. This would allow effectuating two important features of the filesystem, both versioning and snaphots. After a recovery period has passed, the complete inode would get compacted and some operations would get removed from the record. In fact, in the example below all items can be removed on compaction. Example of that is shown below.

An example from the diagram below, the file gets appended several times, then truncated, then overwritten several times, then truncated again.

Journaling representation of a file:

- created new file
- write at (0..49)
- write at (50..99)
- write at (100..149)
- write at (150..199)
- truncated to 0
- write at (0..49)
- write at (0..49)
- write at (0..49)
- truncated to 0

Compacting can use several rules for combining or discarding redundant items:

- subsequent writes that sum to a single write without holes, replaces items 2-5
- subsequent writes that overlap, replaces items 7-9
- write before truncate that leaves shorter file than before write, drops items 2-5 and 7-9
- truncate without predating writes, drops items 6 and 10

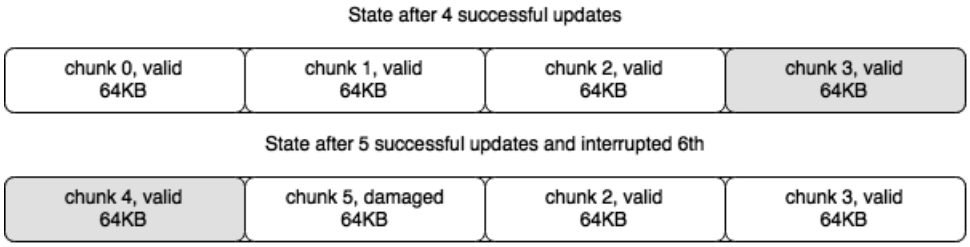
On a side note, Martin Fowler made a lecture about Event driven architectures that mentions this kind of approach, although his talk was about web services rather than local filesystems.

Atomic sector overwrite is not needed or guaranteed.

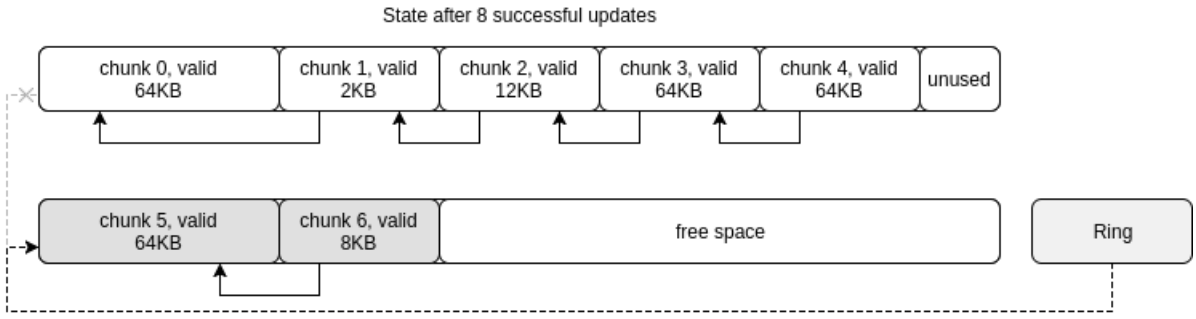
Many filesystems but also frameworks like SQLite guarantee consistency by depending on the underlying hard drive to overwrite disk sectors atomically. There are some issues with this approach, although experimentally so far it seems to have worked just fine. Firstly, this simply moves responsibility down the chain. It is generally a good idea to implement a trait yourself than to 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 this hardware trait should be taken into consideration.

There are at least two new ways to implement atomic changes on disk through software. Both use ids and hashes and no longer require the disk to support atomic sector writes but they do require that any interrupted writes do not damage other sectors. This is 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 attached data are hashed, then the chunk gets stored to disk in one sweep. Updating requires last id and its slot to be known, making this mechanism stateful. When reloading, the entire extent is read from disk in one sweep, and the chunk with highest id and valid checksum is considered current value. Note that 2 chunks are enough for most uses but more chunks can also make sense depending on a particular application. Rings can also be used across multiple devices to synchronize newest value, such as in RAID 1 alike setups.



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 (diffs) but that depends on a particular 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. Updating requires last id and offset to be known, making this mechanism stateful. When reloading, entire extent gets read from disk in one sweep. Each chunk gets parsed only if previous chunks were parsed successfully, that is they had a valid id and checksum. Valid chunks are merged (changesets are applied to base value) or simply last value is taken, again depending on a particular application. Note that a chain is defacto *append only*, meaning it can be updated only until becomes full, and then a new chain must be allocated. 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 the example below, the first chain can be dropped from the ring because the chunk no 5 was fully persisted onto disk.

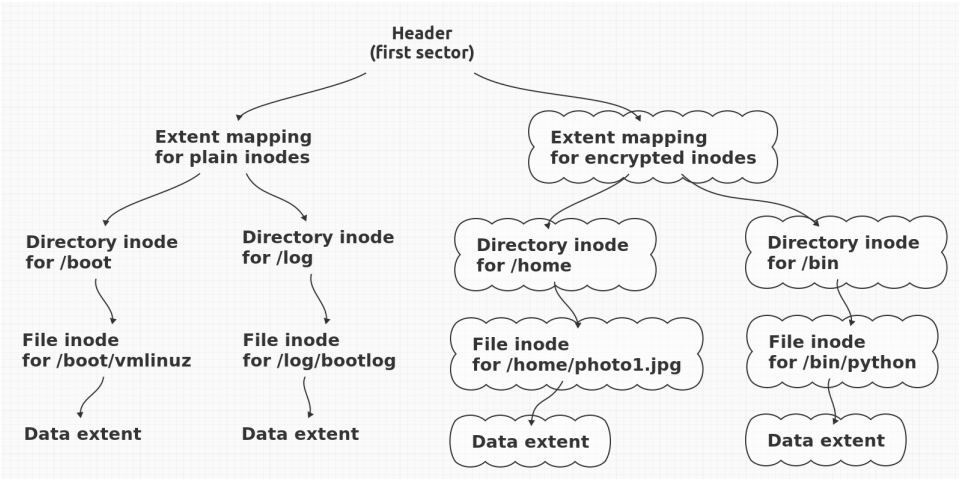


Encrypting disk blocks is not effective.

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 different partitions, unencrypted /boot and encrypted / LUKS partition. There is a reason why one partition setup cannot work, at least not with how things are at the moment. LUKS partitions are encrypted block-wise and provide no means to keep a subset of files unencrypted. A filesystem stacked on top of LUKS is only aware of a block device underneath and have no clue how encryption is applied.

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



Admittedly, there is already a different solution available, by encrypting /home on the fly. Few implementations exist like Ecryptfs and EncFS. Main problems with those are that encryption is not applied outside /home, performance is quite often low, and also their security was called into question by [Hornby14a] [Hornby14b].

Overwriting files is not safe or efficient.

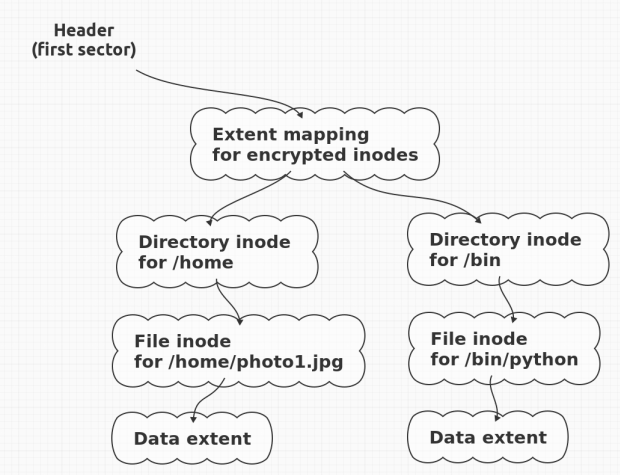
Wiping is an operation that somehow renders a file permanently unreadable and unrecoverable, even under scrutiny in laboratory conditions. Historically and still today, this was achieved by overwriting file content with zero or random bytes. This approach is definitely not safe or efficient.

Regarding safety: files are being overwritten through an interface that provides no means of checking or changing how the process of overwriting files is effectuated. Modern filesystems may use copy-on-write strategy so overwriting files has no effect on already allocated blocks. This means an overwrite may or may not have intended effect. 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, other than wiping entire free disk space.

Regarding performance: 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 of 1~6 TB. For large or highly fragmented files, an entire overwrite takes a long time. And to be really sure, data would have to be overwritten up to 35 times, due to possibility of a forensic use of magnetic force microscopy, see [Gutmann96].

An encrypted LUKS partition is only a partial solution, because it allows to wipe the entire filesystem 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, as described below.

Proposed solution is to encrypt the data structures within the 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 uses. Consider a filesystem where encryption is applied individually and hierarchically to data structures. Each structure is encrypted by a key stored in a structure above it. Wiping any structure in the tree makes the entire branch unrecoverable.

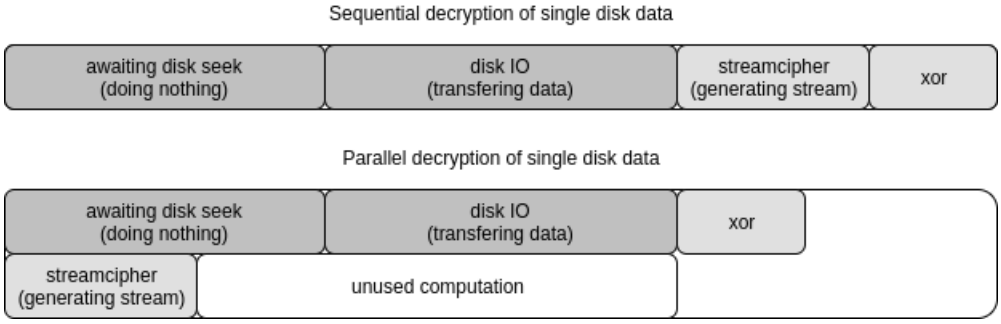


Decryption after reading from disk is not efficient.

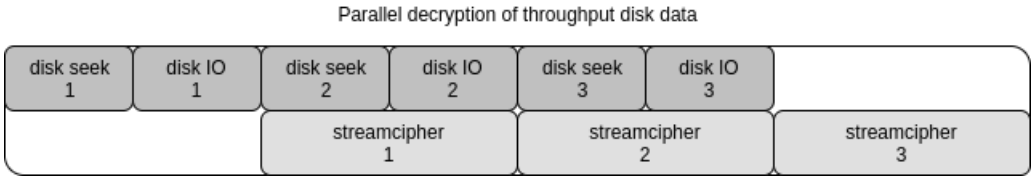
Disk encryption schemes usually employ block ciphers instead of stream ciphers. Perhaps the reasoning is that since usually there is no authentication employed, block ciphers are at least less malleable than stream ciphers. However, in terms of performance, stream ciphers have a certain advantage. Stream 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.

With small reads, since data is small, a second buffer is not costly in terms of memory space, but then not much gain is achieved either.

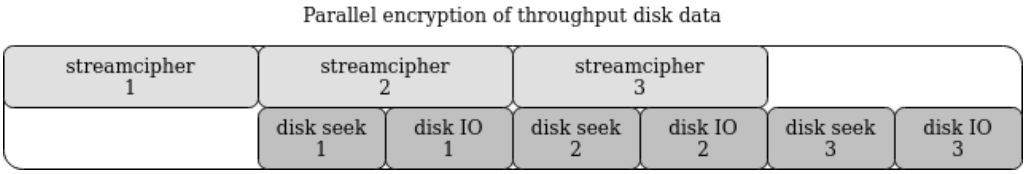


With large reads, almost entire decryption can be done in parallel to disk reads. This approach is already well known, in particular in CUDA community.



Implementation-wise, this scheme despite being parallel can still be effectuated using a single thread because the disk read operation can be done asynchronously, and off-CPU using DMA.

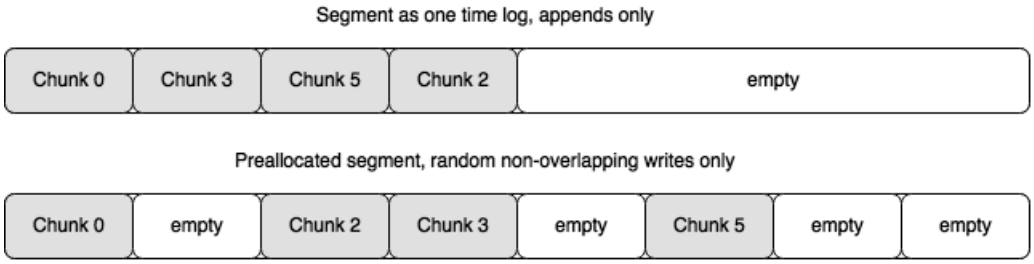
Encryption before writing can also be done almost entirely in parallel.



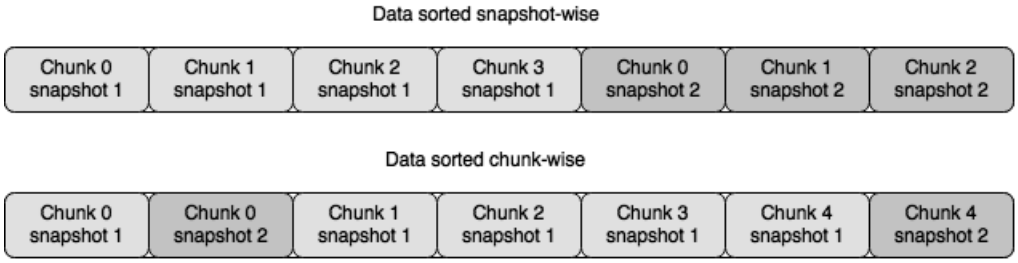
Strategies for data allocation are mutually exclusive.

One issue is how files are written initially versus read subsequently. Consider the difference between how web browsers and torrent downloaders write files to disk. Web browsers always download files sequentially. Therefore it is reasonable to layout chunks on disk in same (arrival) order. However, torrents are downloaded in a different way. Files are downloaded chunk-wise in random order. That is, entire file gets partitioned into chunks and each chunk is written to disk exactly once. In this case, a better strategy is to preallocate an entire segment for this file and then store chunks with random writes. There already exists a system interface which applications use to advise the filesystem on expected file size, POSIX *fallocate*. These two strategies have different performance goals. First approach has high throughput initially when storing chunks to disk, but low throughput on subsequent sequential file reads which are effectuated with random disk reads. To the contrary, second approach initially suffers low throughput due to random disk writes, but then benefits high throughput on all subsequent sequential file reads. It is reasonable to assume that files are usually written once but read several times, but there are known exceptions to that rule. There does not exist a strategy that has performance benefits of

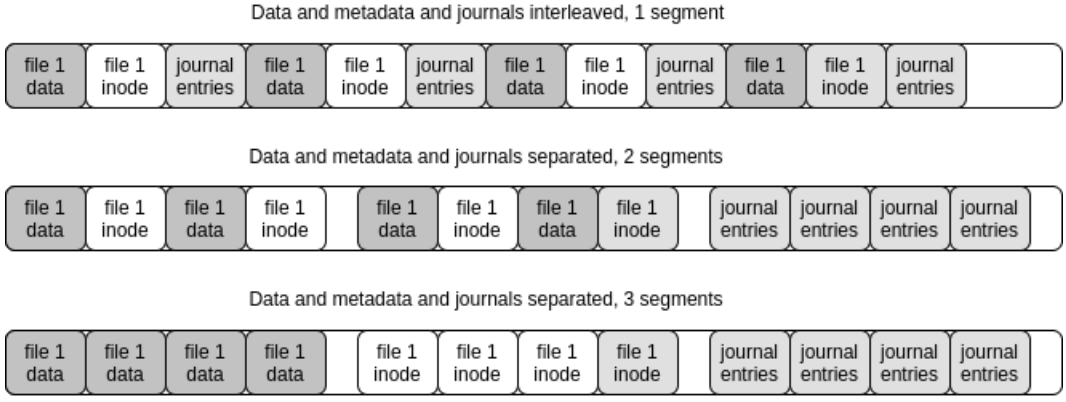
both, it depends entirely on expected access pattern. Layouting strategy should perhaps be selected per file. When *fallocate* is called, it may be reasonable to assume that entire file will be written once, either sequentially or randomly, so preallocating dedicated space of the requested amount should be beneficial in either case.



Another issue is how chunks from across different snapshots but of same file are laid out on disk. First strategy is to store file chunks grouped by snapshot, second is to group by location within file content. Assume the file was modified several times between many snapshots. If the file is to be read only on newest snapshot and those previous snapshots are only kept just in case, then first strategy is better. If the file is to be read randomly across snapshots, it might be better to use second strategy. Again, the strategy should be chosen per file, but there seems no easy way to predict the access pattern. Applications do not report to the system how they are going to access the file, neither spatially not temporally. There does exist a POSIX syscall for that *fadvise*, however no applications known to me use it. The filesystem might gather statistics and predict access pattern from past access patterns. The topic was also investigated by [Rodeh12].



Yet another issue is how data and metadata are stored. There are at least 3 approaches where data could be either laid out together, or partially separated, or entirely separated. Those are file content, file inodes, and journal entries. If journal entries are stored together with data and inodes they refer to, then checkpointing gains speed as each checkpointing requires only 1 seek but both reading files and reading journal entries becomes slower due to fragmentation. On the other hand, if all three are stored in separate segments, then each checkpointing operation needs 3 seeks but both files and journals become less fragmented. First approach may be not as horrible as it looks, if the garbage collector is going to move those extents soon enough. Third approach may happen as a result of explicit preallocation.



Conclusion is that there is always a performance trade-off between initial writing throughput and subsequent reading throughput, due to internal fragmentation. Actually, fragmentation just becomes a synonym for layouting and access patterns, with some performance implications.

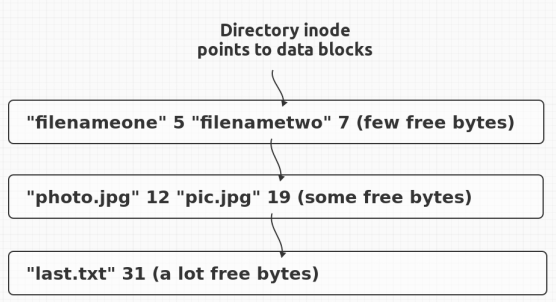
Block based formats are not updateable.

Specifications often define fixed data formats and leave little to 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. Conversion usually requires taking the entire filesystem offline to rewrite all metadata. For example, even filesystems like Ext4 and Btrfs still use fixed size blocks for metadata despite using variable length extents for file content.

Proposed solution is to use variable length extents for everything and to use a data format that stores a version number in each complete inode. This approach allows to update format of each inode one at a time and in real time, without taking the entire filesystem offline.

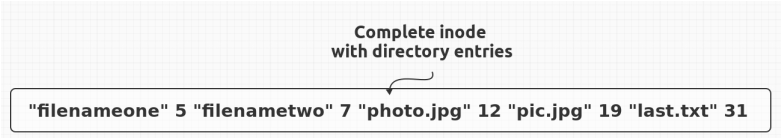
Block based formats are not space efficient.

In Fast File System, directory entries (that map string file names to integer inode ids) 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. Computers of the past often had little memory, [McKusi84] had maybe 8 MB of RAM. Iterating, querying, adding, and removing entries required holding in memory only one block at once.

Consider using an extent instead, to be used as a serialized version of a complete inode. Items can be laid out next to each other without any blanks in between. The extent could have a tail but that takes less than several tails. However, if the filesystem allows extents to start and end at byte resolution, then literally no byte would go to waste. Btrfs supports byte resolution extents, although it still does use fixed sized structures for metadata.



Secondly, suppose we have a dictionary that maps integers onto integers. The application does not matter. When keys are kept in separate blocks, only few keys may be processed together.

A serialized dictionary can benefit from having all keys and values available when dictionary is being serialized into a blob. If all keys are available then keys can be arbitrarily reordered within the blob because deserialisation 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. 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.

First few bytes of file should be easier to read.

When a directory is opened in Nautilus, all files within it are scanned with *file* command that recognizes their mime type. This is a significant 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 the file content, these bytes could be readable more cheaply. Most filesystems would require at least one disk seek (and often several seeks) to start reading first data block or extent.

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

Related files are treated independently.

Often enough, applications access files in sets. For example, Chromium and Firefox are known to open a lot of SQLite files during startup. When one file from a set gets opened, then the entire set could at that point be preloaded concurrently from disk into the cache. Grouping files into sets would probably be done manually.

FUSE overhead is negligible.

This chapter requires some hard experimental statistical data which, at the time of this writing, is not available to me. However, if you want some theoretical arguments, then consider this: FUSE adds mostly some additional context switches on top of disk seek and transfer times. Context switches are very very cheap in comparison.

Preliminary design
Layout and Methods for a versatile filesystem

Introduction

(This chapter is a work in progress, and is therefore inconsistent and incomplete.)

This design phase is difficult. It was attempted more than once, and every time the details got too complicated for the entire thing to make any sense anymore. Therefore a decision was made that the design will be done incrementally, one step at a time.

Version 1: Something that can create, stat, read and write, and delete files.

Purpose of this version:

- Learn how Rust/FUSE framework works.
- Provide a usable persisted on disk filesystem.
- Gather performance statistics.

Introduced features:

- Entire inodes and entire dictionaries
- Atomic and ordered operations (even across files)

Known restrictions:

- No segments, no defragmentation, basic extent allocation strategy
- No atomic data structures, no checksums
- Explicit checkpointing on sync only
- No LRU so inodes get cached permanently, data does not

On disk layout and in memory representation:

Block is hereby defined as 4096 bytes. This document explicitly avoids using the term sector.

Extent is a contiguous area on disk that is block aligned (although it may be split into chunks that are not necessarily block aligned). All disk writes are both copy on write and block aligned, both for safety and performance reasons.

Offset Length is a struct

- Offset on disk, integer u64 that is byte aligned, serialized as varint

- Length on disk, integer u64 that is byte aligned, serialized as varint

Extent Info (extended) is a struct

- Offset on disk, integer u64 that is byte aligned, serialized as varint
- Length on disk, integer u64 that is byte aligned, serialized as varint
- Inode Type, an enum that discerns between inodes, data, regular files, directories, etc.

Note: both Offset Length and Extent Info can be null, that is zero length. Any code processing those must potentially take care of that.

Note: The Offset Length struct used to be called a simplified Extent Info, but not to confuse the readers, it was decided to use two different names for these two struct.

FsHeader (first block on disk):

- "CAMELEONICA\0" magic string, 12 bytes constant
- FormatVersion: integer set to 1, u64, implementation should panic on any other value
- ExtentMap: offset-length of the extent containing the entire Extent Map
- FreeMap: offset-length of the extent containing the entire Free Map
- NextOperationID: an integer u64 that gets incremented on every modifying operation (sometimes also called the Revision ID)

Note: FsHeader always fits in a single atomically overwritten block. In reality, the atomicity of disk writes is questionable and might not exist but for the time being let us assume it does. After all, all other filesystems seem to rely on this anyway.

Note: NextOperationID field serves two purposes. First, it gives globally unique IDs (be it Inode IDs or Extent IDs or Revision IDs). The IDs are never ever reused. Secondly, the IDs establish a strong order, meaning the operation carrying the higher ID must have happened after all the other operations carrying lesser IDs.

Extent Map: in memory it is a dictionary (a hashtable) that maps an integer u64 Extent ID (sometimes also called an Inode ID) into an Extent Info. On disk, the contiguous blob is a list of keys and values as varints.

Free Map: in memory it is a list that keeps track of offset-length tuples. Optionally, it might also use a dictionary to allow for merging of deallocated extents. On disk, the contiguous blob is a list of offset-length tuples as varints.

Note: the Free Map blob cannot contain itself, so after it was serialized it is used to allocate an extent for its own blob. Therefore, after it gets read from disk and parsed, its own extent needs to be removed from the in memory representation.

Buffer cache: only in memory, a dictionary from integer u64 inode IDs into Inode structs. This data structure holds only past and presently used inodes, and indirectly some dirty data. After sync, the dirty data is removed from buffer cache, however the non-dirty inodes remain.

Inode (for all inode types): a struct

- ID: an integer u64 copied from the Extent Map dictionary
- Inode Type: an enum copied from the Extent Map dictionary

    - This enum discerns between regular files, directories, etc.
- Chunk Map: a list of past operations done on this file or directory
    - This includes dirty data that was not sent to disk yet.
- NewestSize: an integer u64 that is consistent with the chunk map
- IsDirty: a bool that signifies this inode was modified (or has dirty data)

Inode chunk (for regular files): an enum

- File write
    - Revision ID of the file operation
        - This ID comes from the NextOperationID field.
    - Extent ID of the data extent where the chunk is stored, an integer u64
        - This ID is translated by the Extent Map into an Extent Info.
    - Offset inside the data extent, an integer u64
    - Length inside the data extent, an integer u64
    - Offset inside the file content space, an integer u64
- Dirty file write
    - Revision ID of the file operation
        - This ID comes from the NextOperationID field.
    - Offset inside the file content space, an integer u64
    - Dirty data buffer, as byte list

Inode chunk (for directories): an enum

- File addition
    - Revision ID of the file operation
        - This ID comes from the NextOperationID field.
    - Name of the new file
    - Inode ID of the new file, as per the Extent Map
- File deletion
    - Revision ID of the file operation
        - This ID comes from the NextOperationID field.
    - Name of the deleted file

Deallocated extents: only in memory. This list contains extents that will not be reachable from the new metadata after next checkpointing but can still be read, for example if next checkpoint is interrupted and is not persisted on disk. Those extents get moved into the Free Map during the checkpointing (sync) operation.

Methods presented through FUSE:

Creating and mounting the filesystem:

- Method is given a disk pathname as a string.
- Set Extent Map to an empty dictionary.
- Set Free Map to entire disk, minus first block for FsHeader.
- Create a dirty empty directory inode, for the root directory.
- Set inode 0 in buffer cache as that inode.
- Set the FsHeader entry for Extent Map as null (zero length).
- Set the FsHeader entry for Free Map as null (zero length).

- Set the FsHeader entry for NextOperationID as 1 (since 0 is used by root dir inode).
- Call the sync method.

Mounting the filesystem:

- Method is given a disk pathname as a string.
- Read and parse FsHeader.
- Read and parse the Extent Map, using FsHeader.
- Read and parse the Free Map, using FsHeader.
  - Note the Free Map extent itself needs to be subtracted from free space.
- Read and parse inode 0, which is the root directory inode, using the Extent Map, populate the buffer cache with it.

Internal stat function:

- Method is given a pathname string, made up of path components.
- Assume the inode ID=0 (which is the root directory inode) is a directory inode, that it is already in buffer cache, and that it is the last inode so far.
- Assume the list of inode IDs is empty, add 0 entry to it.
- In loop, for every path component
  - Check if the path component is an empty string, then panic.
  - Check if the path component is "." or "..", then panic.
  - Check if last inode ID was null, then add a null to the list of IDs and skip this loop.
  - Check if last inode so far is a directory, otherwise set last inode ID to null and add a null to the list of IDs and skip this loop.
  - Get inode ID from last directory inode, use path component as key.
  - Check if that inode is in buffer cache, then skip this loop.
  - Get extent info for next inode from the Extent Map, use the inode ID as key.
  - Read and parse next inode, using the extent info, populate the buffer cache.
  - Assume this is the last inode so far.
  - Add the inode ID to the list of inode IDs.
- Return the list of inode IDs.
  - Note this list always contains at least one non-null (the ID of 0).
  - Note that one null is an expected result for some purposes (like file creation).

Stat method:

- Method is given a pathname as a string.
- Call internal stat on given pathname.
- Check that none of inode IDs is null, otherwise return an error.
- Compute stat structure from the last inode.
- Details like permissions or datetimes can be faked.

Create directory method:

- Method is given a pathname as a string.
- Call internal stat on given pathname.
- Check that exactly last inode ID is null, otherwise return an error.
- Get and increment the global NextOperationID counter.
- Create a new empty directory inode under the ID, populate buffer cache.
- Mark the directory inode as dirty.

- Modify last directory inode, add an entry for the filename to map to the ID.
- Mark the last directory inode as dirty.

Delete directory method:

- Method is given a pathname as a string.
- Call internal stat on given pathname.
- Check that none of inode IDs is null, otherwise return an error.
- Check that last inode is a directory file, otherwise return an error.
- Check that last inode is empty, otherwise return an error.
- Check if the inode ID is listed in the Extent Map, then
  - Deallocate directory inode itself (move it to deallocated extents).
  - Drop the inode ID from the Extent Map.
- Drop the directory inode from the buffer cache.
- Modify the upper directory inode, add a Chunk Map entry for the directory removed.
- Mark the upper directory inode as dirty.

Create file method:

- Method is given a pathname as a string.
- Method is given a mode, assumed to be the default.
- Call internal stat on given pathname.
- Check that exactly last inode ID is null, otherwise return an error.
- Get and increment the global NextOperationID counter.
- Create a new empty regular file inode under the ID, populate buffer cache.
- Mark the regular file inode as dirty.
- Modify last directory inode, add an entry for the filename to map to the ID.
- Mark the last directory inode as dirty.

Delete file method:

- Method is given a pathname as a string.
- Call internal stat on given pathname.
- Check that none of inode IDs is null, otherwise return an error.
- Check that last inode is a regular file, otherwise return an error.
- Get and increment the global NextOperationID counter.
- Deallocate all data extents from inode chunk map (move them to deallocated extents).
- Check if the inode ID is listed in the Extent Map, then
  - Deallocate file inode itself (move it to deallocated extents).
  - Drop the inode ID from the Extent Map.
- Drop the file inode from the buffer cache.
- Modify the directory inode, add a Chunk Map entry for the filename removed.
- Mark the directory inode as dirty.

Note: the delete file method does not respect opened files or hard links.

List directory method:

- Method is given a pathname as a string.
- Call internal stat on given pathname.

- Check that none of inode IDs is null, otherwise return an error.
- Check if last inode is a directory, otherwise return an error.
- Compute the filename list from the last inode.

Open file method:
- Method is given a pathname as a string.
- Method is given a mode, assumed to be read-write mode.
- Call internal stat on given pathname.
- Check that none of inode IDs is null, otherwise return an error.
- Return the file inode ID as a file handle result.

Close file method:
- Method is given a file inode ID as a file handle.
- Method does nothing.

Read file method:
- Method is given a file inode ID as a file handle.
- Method is given an offset-length in file content space.
- Get the file inode, using the ID as a key to the buffer cache.
- Search the chunk map for extents that overlap with the offset-length.
- Read those chunks from disk (can be less than the entire data extents).
- Merge those chunks into an appropriate buffer, and return that buffer.

Write file method:
- Method is given a file inode ID as a file handle.
- Method is given an offset and a buffer.
- Get the file inode, using the ID as a key to the buffer cache.
- Get and increment the global NextOperationID counter.
- Add an entry to the chunk map, including the revision ID, offset and buffer.
- Mark the inode as dirty.

Internal extent allocation function:
- Function is given an amount of bytes (an extent size).
- If the size is zero, panic.
- Round the size up to a multiple of a block size.
- Search the Free Map list for a smallest extent at least the demanded size.
- If the extent is exatly the size, then
  - Remove it from the Extent Map.
  - Return it.
- If the extent is larger than demanded, then
  - Remove it from the Extent Map.
  - Split the removed extent into 2.
  - Add the tail back to the Extent Map.
  - Return it.
- If there is no sufficient extent available, then
  - Panic.

Internal extent deallocation function:

- Function is given an offset-length of an unused extent.
- If the size is zero, panic.
- Add the extent to the Deallocated Extents list.

Note: Deallocated extents cannot be reused until the filesystem metadata on disk is updated.

Fsync method:

- Method is given an inode ID as a file handle.
- Method does nothing.

Sync method:

- (… files …)
- Loop over inodes in the buffer cache.
- If the inode is dirty, then
  - If the inode is a regular file,
    - (… the data …)
    - Search the inode Chunk Map for dirty buffers.
    - If there are no dirty writes, skip this step.
    - Serialize and concatenate all those dirty buffers into a single data extent.
    - Get and increment the global NextOperationID counter.
    - Update the Chunk Map, so that dirty write entries are replaced with file write entries and point to the new data extent.
      - This effectively removes data from buffer cache, leaving only the inode.
    - Allocate enough space for the data extent, using the Free Map.
    - Update the Extent Map, so that the ID points to the data extent.
    - Send the data extent to disk.
  - (… the inode …)
  - Serialize the inode into a blob.
  - Check that the inode has an allocated extent in the Extent Map, then
    - Deallocate the current inode extent, using Extent Map (move it to deallocated).
  - Allocate enough space for the inode blob using the Free Map.
  - Send the inode blob to disk.
  - Mark the inode as non-dirty.
- (… filesystem metadata …)
- Check if the Extent Map entry is non-null (has non zero length), then
  - Deallocate the Extent Map, using the FsHeader (move it to deallocated).
- Serialize the Extent Map into a blob, allocate it space from Free Map, and send it to disk.
- Update the Extent Map entry in the FsHeader to point to the new blob.
- Check if the Free Map entry is non-null (has non zero length), then
  - Deallocate the Free Map, using the FsHeader (move it to deallocated).
- Move all extents from the deallocated list into the Free Map.
- Serialize the Free Map into a blob, allocate it space from Free Map, and send it to disk.
- Update the Free Map entry in the FsHeader to point to the new blob.
- Sync the disk.

- Send the FsHeader to disk.
- Sync the disk.

Performance measuring scripts:

All scripts should be tested on a 15 GB partition (out of which 12.2 GB goes for data and 390 MB goes for metadata) with either Cameleonica, Ext4, Btrfs, or ZFS.

In order for the measurements to be fair, other filesystems (those implemented in the kernel) should go through a single-thread pass-through Rust/FUSE filesystem. Also, they all should use noatime flag.

File creation:

- Create a new filesystem on the device.
- Create 100'000 files, each being 128 KB in size, each having an 8 levels deep random path where each intermediate directory has a name between 0 and 255.
  - Some paths will be chosen more than once, but that is alright as long as it happens the deterministically same way for every test.
  - Call a sync after every 1000 files.
- Call a sync at the end.
- Measure total and average time for all operations.

File stat:

- Assume the filesystem was created using the first script.
- Unmount the filesystem and drop the entire buffer cache.
- Mount the filesystem again.
- Call stat syscall on every directory and file exactly once.
- Call a sync at the end.
- Measure total and average time for all operations.

File removal:

- Assume the filesystem was created using the first script.
- Unmount the filesystem and drop the entire buffer cache.
- Mount the filesystem again.
- Call unlink or rmdir syscall on every file and directory exactly once.
- Call a sync at the end.
- Measure total and average time for all operations.

Questions and Answers
Topics that reviewers found unclear

asked by Daniel Arndt
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. First is reversible and second 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 150 MB/s assuming no seeks due to fragmentation. Overriding data 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 from inode, due to truncates for example, cannot be overridden. In general, it is not possible to find out wether a file used more blocks before that were deallocated, nor to reallocate those specific blocks back. Once a block is deallocated, it is gone and not trackable anymore. Third problem, 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 to 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 (here called deletion) is an action that can be taken by non-privileged users. User processes are able to take this action without any user interaction. Therefore this kind of action should be inspectable and undoable. Browsing revision history reveals regular deletions and allows recovery, at least as long as there exists a version prior to deletion, available either through versioning or snapshot mechanism. Permanent deletion (here called wiping) is a different action that should require root privileges. After a file was wiped using the filesystem command-line tool, its content including historical revisions is no longer accessible, regardless of versioning and snapshots.

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 in a version control software, 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 a file currently requires same level of access as browsing current version. When user deletes a file, it will be recoverable as long as it is browsable in revision history or there was a snapshot made before it was deleted. When user permanently deletes a file (wipes it), it will not be recoverable even through past versions of the file or snapshots containing it.

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 close to zero time. This also erases data blocks no longer assigned to a file, due to truncate operations. Versioning is an independent feature. Same file is encrypted with same encryption keys, regardless of versioning and snapshots, allowing swift and 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 cryptographic products
- required to disclose cryptographic keys to authorities in advance or on demand
- asked to enter password and hand over your laptop for inspection at checkpoints
- asked by authorities if you have any other undisclosed encrypted partitions
- subpoenaed by courts to produce keys or decrypted data itself
- jailed for long time or until you produce encryption keys or decrypted data itself

Lesson to learn here is that national law can put you in a position where technical solutions do not give you an easy or legal way out. Consider crossing US border with child pornography on your laptop (there was a famous real case like that). 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 enable you to lie your way out effectively but it will not make it any more legal. This makes the staganographic feature the most controversial one, and for that reason alone it might not make it into the final product.

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 many other advanced features, some are specialized and clearly not many people will have a use for them (non transparent hashing, for example), while some are general purpose and should be useful to a lot of people (transparent compression, for example).

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 significantly 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 compressible data, while a hard disk can sustain only about ~150 MB/s transfer. Fragmentation lowers disk side of inequality even further. Direct Memory Access (DMA) allows to transfer data to disk in parallel to computation so compression can be done in parallel. The minimum of two throughputs then becomes a bottleneck. Compression is therefore likely to be disk bound, and not CPU bound. Also, adaptive mechanism 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 gets written in plain form during initial write, at full speeds, and then 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 ~150 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. Revision history (continuous versioning) is different than snapshots in that every file operation creates a revision. As time progresses, old revisions get automatically compacted. So in summary, snapshots are manually created while versioning is automatic, and snapshots are kept forever while versions are only available for a short period of time.

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. Even later filesystems like Btrfs and ZFS started using copy on write and checksums, which is probably the best solution atm.

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 no-op since header always references extents that were completely synced to disk. Interruption 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? This was explained in SSDs chapter.

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 (garbage collection) is supposed to be delayed into the future, but what about those servers running continuously 24/7?

Some servers actually do experience peek hours, especially those hosting non-international websites. Then less disk IO rate would be used for garbage collection during peek hours than during non-peek hours. On servers running continuously, compacting can run continuously but at small IO rate. Also note that preallocating huge files (gigabyte sized) can be used to avoid some compacting, as well as disabling versioning feature and not using snapshots.

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

As explained in Entire dictionary chapter, only up to 200'000 files are expected, both on desktops and servers. Under such assumptions, the dictionary approach is better.

What about the fact that common compression tools are not aware of non-transparent compression functionality?

System tools are open source and patches could be submitted to their repositories. And also, the filesystem command-line tool itself could be used for creating archives, using a proprietary format. Cooperation from the side of standard compression tools is not required.

Bibliography

Gutmann96: Peter Gutmann, Secure Deletion of Data from Magnetic and Solid-State Memory, 1996
Dreyfus2000: Suelette Dreyfus, The Idiot Savants' Guide to Rubberhose, circa 2000
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/
Overstreet21: Kent Overstreet, bcachefs: Principles of Operation, 2021
Patterson: David A. Patterson, Kimberly K. Keeton, Hardware Technology Trends and Database Opportunities,
Agraw07: Nitin Agrawal, William J. Bolosky, John R. Douceur, Jacob R. Lorch, A Five-Year Study of File-System Metadata, 2007
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
Pillai13: Thanumalayan Sankaranarayana Pillai, et al, Towards Efficient, Portable Application-Level Consistency, 2013
Tweedie15: Stephen Tweedie, Stephen Tweedie Email.txt, 2015
Hornby14a: Taylor Hornby, eCryptfs Security Audit, 2014, https://defuse.ca/audits/ecryptfs.htm
Hornby14b: Taylor Hornby, EncFS Security Audit, 2014, https://defuse.ca/audits/encfs.htm

Please note that all cited documents (both references and hyperlinks) are saved as copies in the repository and you do not need to search the web to read them.