Ideas and Observations
Foundations for a versatile filesystem

Arkadiusz Bulski

Abstract

Cameleonica is a prototype of a highly versatile filesystem. This filesystem is to cover a wide range of safety and security and usability features.

Mission statement explains what the project is all about, as it is the first phase in systems engineering, it sets a direction in which further development will tend to. Purposefully omitted are any technical details on how to achieve it. Features are mostly described through basic definitions and elaborate examples, not technical implementations. Usage scenarios showing obvious benefits are also presented. Scenarios show how different people benefit from different subsets of features.

Ideas and Observations describe a wide range of theoretic concepts that could be incorporated into a new filesystem design. Potential benefits are presented, and also reasons why these features were not popularized earlier. Expectations of their performance are based on theoretic models and measured empirical evidence.

[work in progress]

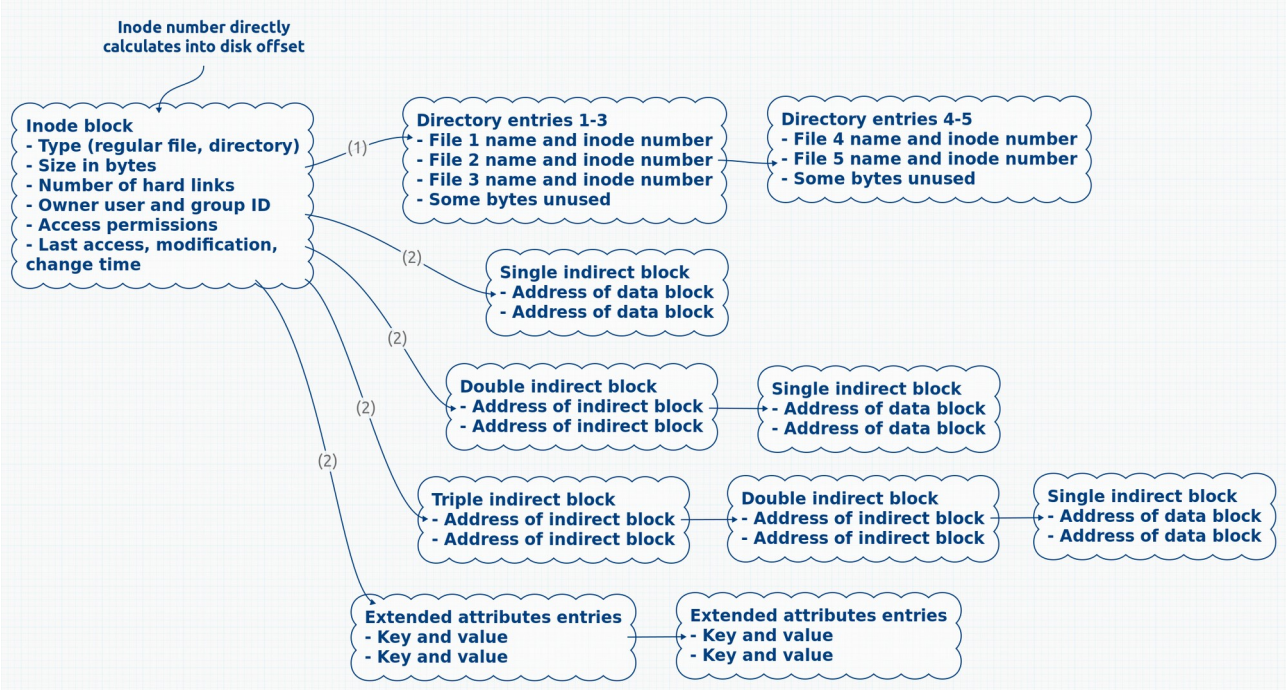
Introduction

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. Some designs like log-structured filesystem [Rosenblum91] seem to have been too radically different from mainstream designs that they have not gained much traction. For example, Ext4 [Mathur07] is an upgrade of Ext3 but it is somehow constrained by previous data formats. Switching from block to extent allocation is not that easy. B-trees were introduced per-directory but not filesystem wide. Offset fields were widened to 48 bits but they will have to be widened again at some point, instead of using variable length encoding. Ext3 was designed directly upon Ext2 with only one component added, a special file used for journaling. Journaling is slow as everything needs to be written to disk twice, and fragmentation is not actively avoided. Ext2 was built as analog to Fast File-System. Btrfs and ZFS seem to be examples of quite radical redesign from scratch rather than gradual evolution. It is possible that they did not gain wider adoption because of overly complex implementation (Btrfs has 120K sloc) and non portability (ZFS is on Solaris). This paper presents even more radical design ideas and hopefully will bring more features to the open source community.

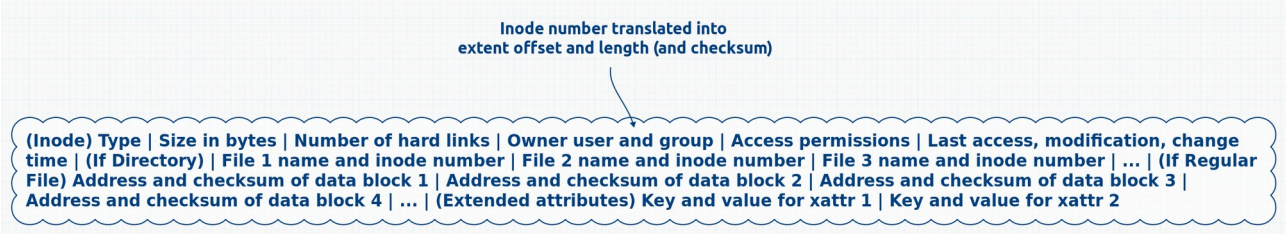
Inodes are not efficient.

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 long, discrete blocks of data. In traditional designs most file operations can be done by processing one block at a time. Computers of the past often had little memory, [McKusi84] had maybe 8 MB, and caching complete sets of metadata was not feasible. Also changing metadata often ended up in writing just one block, which might have been seen as a compelling reason to use this approach. Furthermore hard disks guarantee that any given block is written atomically [Twee00], which was used for guaranteeing consistency after crash, for example in Ext2. All these reasons contributed to a decision that data structures had to be broken down into blocks.

To fully describe a single file, an amount of metadata needed almost always takes space of more than one block. Traditionally, as started in Fast File-System [McKusi84] and carried on in Ext2/Ext3, one main block (called inode) would hold most important data and point to further blocks, several blocks would point to where actual content is located (indirect blocks), some blocks would keep directory entries in case of a directory, and so on. For a large 1 GB file for example, assuming 4 KB blocks and 64 bit pointers, at least 512 blocks is needed. Certain operations such as copying or deleting file, or browsing directory necessarily requires all metadata blocks to be read. This was also noted in [Mathur07].



Instead, a *complete inode*, variable size structure could contain all metadata associated with a given file. In memory representation may remain similar to the one above but on disk format would be like one below. On disk structure always remains continuous. Size of entire metadata could easily go into hundreds of kilobytes for gigabyte sized files, depending on whether per-block checksums are included. Below are presented arguments showing expected performance gains of complete inodes over classic inodes and indirect blocks.



Consider a range of possibilities, where all metadata blocks are either allocated in one continuous range (called an extent), are divided into subsets, 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 changes are made all blocks are stored to another continuous area in one sweep. Last approach we shall call *block approach.* It is prevalent in modern filesystems. Space is allocated even one block at a time if allocation happens with long enough intervals, eventually leading to fragmentation of metadata.

Block approach seems efficient at first glance because changing one field requires only one block to be written to disk. To the contrary, extent approach may require rewriting the whole extent which takes many blocks. This kind of reasoning is flawed because it is focused on time of update operation but does not account for future read operations. Further analysis considers amortization of performance over most of lifespan of a file, and does not assume that any one operation must be carried out as fast as possible independently of other further operations.

It could be argued that extent approach, where all metadata is always loaded and stored in one sweep is better in every practical usage scenario. To show that, we need to recognize that magnetic hard disks have quite skewed performance characteristics. Flash based storage is considered separately. Representative hard disk is capable of ~120 MB/s of sustained throughput and ~10 ms seek time to random location on average. Quick calculation shows that reading or writing an extent smaller than 1.2 MB is faster than literally one seek, on average.

This observation may seem counter intuitive. Filesystem designs so far seem to be based on the assumption that data needs to be processed in smallest chunks possible and only way to go is to process as small amount of them. It seems that the opposite approach of processing bigger chunks never gained traction. In traditional designs [OSTEP] directories put entries into individual data blocks, files put data locations into individual indirect blocks, and so on. In modern designs [Rodeh12] [ZFS] copy on write B-trees are replacing inodes and indirect blocks with leaf nodes but data is still being divided into small chunks. The issue being described here has not gone away.

Extent approach is shown to be better by considering total time spent on all operations throughout most of the lifetime of a file. Only metadata is counted towards the time spent on file operations. Both file content and parent directory metadata are excluded. At time zero file is assumed to already exist, with all content and metadata. It is assumed that file grew slowly enough for metadata blocks to be allocated individually, not being affected by deferred allocation. Blocks are allocated from entire disk, although smaller areas will also be investigated. A certain number of cycles is considered, where file is opened, several metadata blocks are accessed and then file gets closed. In each cycle some subset of metadata blocks is accessed, either read or written. Cached blocks are forgotten between cycles due to assumed long time intervals between cycles. Blocks are assumed to be 4 KB size. Suffixes like K M refer to thousands and millions of bytes.

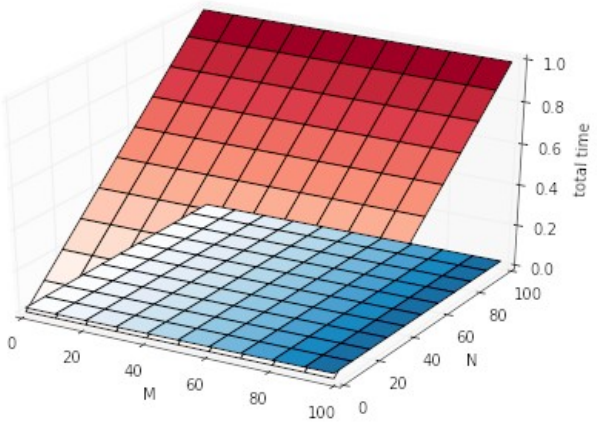
Reading or writing N blocks takes in total (assuming sequential and random pattern, referring to extent and block approach respectively):

$$R_1(N) = 0.010 + N \cdot 4\,K/120\,M = 0.010 + N \cdot 0.00003$$
$$R_N(N) = N \cdot (0.010 + 4\,K/120\,M) = N \cdot 0.01003$$

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):

$$T_1(N, M, B) = \sum_{i=1}^{B} R_1(M) + R_1(M)$$
$$T_N(N, M, B) = \sum_{i=1}^{B} R_N(N)$$

Plots show total time of extent approach (blue) and block approach (red) for least and most accessed files (N), and least and most sizable files (M).



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

$$T_N < T_1$$
$$B \cdot R_N(N) < B \cdot 2 \cdot R_1(M)$$
$$0.01003 \cdot N < 2 \cdot (0.01 + 0.00003 \cdot M)$$

$$N < 1.994 + 0.006 \cdot M$$
$$M > 166 \cdot N - 332$$

Last two inequalities show limits on how much metadata can be accessed during one cycle before block approach starts to lose advantage. Asymptotically, less than 1 in 166 metadata blocks could be accessed. For smallest files the limit is 2 blocks. Remember when earlier it was mentioned that a 1 GB file would require at least 512 metadata blocks? According to the limits, after opening at most 5 metadata blocks could be accessed before block approach would lose advantage and this includes the inode block. Is this really the kind of workload we are aiming to support?

When individual blocks are stored, time is gained on transfer time due to small block size but then more time is lost to seek time when blocks are read at next cycle. Disk characteristics make this exchange totally unfair. One seek takes as much time as transferring 1.2 MB which is a lot. For many files, complete metadata could be fit within that amount of space. Also, on desktops 90% of directories have at most 20 entries [Agraw07]. Additionally the extent approach requires one checksum to verify integrity while the block approach requires as many checksums as there are blocks.

Finally it should be admitted that the theoretical model above assumes most naive

implementation of block approach where blocks are allocated individually (not grouped, as if allocations were separated by long time intervals) and non-preemptively (no space reserved in advance) from entire disk (space not divided into segments). It would be possible to allocate blocks in groups (deferred allocation), allocate space in advance, and also select blocks from small areas or segments (clustered allocation), increasing so called *spatial locality*. However, constructing a theoretical model that includes deferred allocation is outside of scope of this paper. Probabilistic distribution parameters are not known.

Secondly, disks can reorder outstanding operations to minimize total seek time, with technology called NCQ/TCQ. Model assumes that blocks are accessed without concurrency, one block after another. Some usage scenarios allow to anticipate which blocks will be subsequently accessed and disks would be able to take advantage of it. However, reordered random disk operations will never be as fast as one contiguous disk operation.

Thirdly, seek times and rotational delays are quite expensive and the hardware trends are diverging. Delays and throughputs will be even more asymmetric in the future [Patterson].

Experimentally, average seek times measured on SAMSUNG HD154UI are as follows. Seek time is a function of area size over which seeks are made with uniform distribution. Code can be found in the repository.

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

Above seek times are averages for single seeks. Formulas used for comparison between block and extent approach can be updated with any seek time from table above if clustered allocation is to be included. Sequential times mean that blocks are accessed one at a time, not that they are laid out in a contiguous area. For small files, metadata blocks could be clustered in small areas. For example, for 1 MB and 16 MB areas block approach works for at most 1 block access. For larger files, many blocks are needed so they have to be spread over larger areas, which leads to longer seek times. Concurrent times converge with sequential times as disk area gets larger. Reordered random disk operations will never be as fast as one contiguous disk operation.

Changes pending: SSD devices may be subject to the same argument, due to write amplification and switching between segments. Quantifying it is another matter.

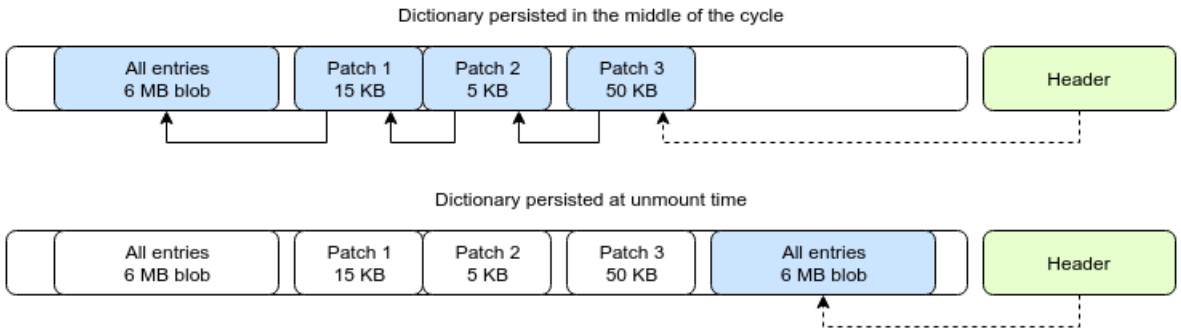
B-trees are not efficient.

There is a trend among filesystem designs towards using both modified in-place [Mathur07] and copy on write B-trees [Rodeh12] [ZFS]. This trend might be easily explained by common expectation that filesystems should be able to handle huge numbers of files. Popular filesystems even explicitly advertise their ability to scale among main points why to choose them over the competition. B-trees are a good approach if a huge amount of keys is 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. This asymptotic behavior seems to have mislead everyone. It would be justified to use B-trees if billions of files were expected to be stored but that is a false assumption, at least in general case. Most computers hold on the order of 100'000 files [Agraw07]. That amount of dictionary entries can be stored and accessed more efficiently.

Second reason to use B-trees in the past is that memory needed to query and update trees is small, comparable to number of nodes from root to leaf. Computers of the past often had little memory, [McKusi84] had maybe 8 MB of memory. However, modern desktop computers have several gigabytes of memory and servers have tens of gigabytes of memory. Conserving memory is not justified anymore. Memory constrained devices like smartphones constitute an exception but that does not negate benefits on desktops and servers.

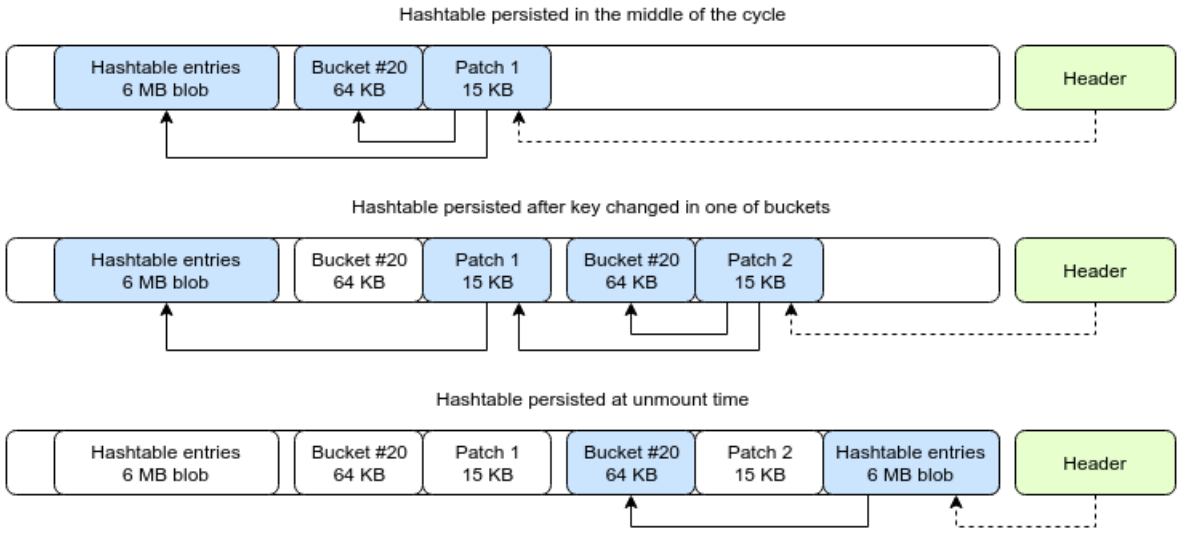
Consider the three approaches described below, depending on how many entries we expect to hold at any given time. Respective capacities will be tweaked in order to make performance comparable. As in previous section, we assume hard disks can sustain ~120 MB/s throughput and ~10 ms seek time on average.

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. If these disk operations are going to take time equivalent of five disk seeks, the dictionary can store about 260K entries (6 MB or less) assuming 64 bit keys and 128 bit values. This amount of extents should be enough to store the files found on a typical desktop, which is 100K files of mean size 190 KB as found by [Agraw07]. Even better, small patches containing sets of changed entries can be stored to disk every few seconds and entire dictionary be stored only every few minutes. If filesystem gets unmounted successfully, only entire dictionary is going to be loaded at mount time, otherwise some few dozen patches need to be loaded and combined. On disk extents would actually take less space due to varint coding. These modifications push performance of this approach even further. Entire dictionary is being cached so all key lookups are instantaneous. Write buffers can gather key changes across few second periods, amortizing time spent on disk seeks. Concurrency is a non issue since operations are always instantaneous. Amount of extents can be minimized if files store sets of chunks in single extents.



Second approach, hashtable dictionary is stored on disk in single extent, pointing to bucket dictionaries also stored on disk in single extents. When a key is accessed, key is hashed modulo number of buckets, selected bucket gets loaded from disk in one sweep, bucket is modified and

stored back to disk as copy on write in one sweep. Hashtable is assigned a new extent for the modified bucket. Every few minutes the hashtable gets stored to disk as copy on write just as in previous approach, with patches every few seconds as well. Suggested capacity would be 260K entries in the hashtable (6 MB or less) and 2730 entries per bucket (64 KB or less), giving a total ~700K entries. On disk extents would actually take less space due to varint coding. Loading and storing 64 KB from disk is approximate to a disk seek itself. Patches can again be used for the hashtable, but buckets should be stored to disk as soon as a new bucket is about to be modified. Buckets can be cached but only one bucket should be held in a write buffer at a time. Each key access takes at most 1 disk seek, unless cached or buffered. Concurrency is possible as buckets can be processed independently with proper locking.



Third approach, B-trees as the contemporary alternative. If we assume five seeks are allowed per key access and nodes hold 4096 keys (64 KB), then entire tree has a capacity of about 10^{18} keys. This assumes no data structures that keep track of allocated space, so compacting disk space requires reading all nodes and through them finding which blocks are used. Each key access takes up to 5 disk seeks, with any node being possibly served from cache. Capacity of trees is indeed impressive and memory requirements are very low, only few nodes are needed at any given time, but performance is worse than those of previous approaches.

This argument is basically the same as for inode extent. If extent dictionary is considered as a metadata of some special file then same theoretical argument can be applied.

Fsync is not efficient.

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

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

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

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

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

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

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

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

Consider a filesystem where all operations are both atomic and ordered, with respect to crash. Above that fsync calls are implemented as no-op.

The file replacing codes above remain to keep files consistent. This is due to the fact that application developers (usually) use fsync not to persist data immediately but to persist data before metadata, which is what ordering behavior implies. Linux man pages define fsync as a means to persist data immediately but surprisingly this is not what POSIX strictly requires, rather it is just a mainstream interpretation. POSIX first defines fsync (in vague terms) as flushing buffers to a device but then explains in the notes, and explicitly, that if a filesystem can guarantee safety in a different way then that also counts as a valid implementation of fsync. Flushing buffers may be an obvious way to do it but not the only way. Excerpt from POSIX documentation:

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

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 (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 queue was empty. When flushing buffers occurred quite often this approach basically defeated the purpose of installing buffers in disks in the first place.

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

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

Here is a plausible implementation of a filesystem where all operations are atomic and ordered (but not yet durable) with respect to crash.

Consider a copy on write scheme where first sector contains address of last persisted block and address of root inode, further sectors contain a continuous cyclic stream of data and metadata similar to a log-structured filesystem.

Each operation puts more blocks into the stream in a sequential but asynchronous way. Ordered operations require that space on disk is allocated sequentially, so spatial ordering corresponds to chronology of operations, but actual disk writes need not be atomic nor ordered. These disk writes can be reordered and interleaved with previous and future disk operations. In most naive implementation, each operation appends full metadata of affected inodes to the log (storing complete inodes on each operation). However, considering that complete inodes are

quite bulky, this approach is not good enough. Rather it provides a baseline upon which improvements will be made.

After the filesystem is sure that blocks up to a given point were persisted, it issues an atomic but also unordered write of the first sector (checkpointing), moving the address of last used block. Recovery is not strictly necessary so far because first sector reliably tells us how much data was persisted. Operations that happened between checkpoints are always lost. The window of changes that can get lost can be quite significant, depending on how many changes (or for how long) are accumulated between atomic writes of header. Modern filesystems make this sacrifice on regular basis. Losing few seconds worth of operations is commonly accepted.

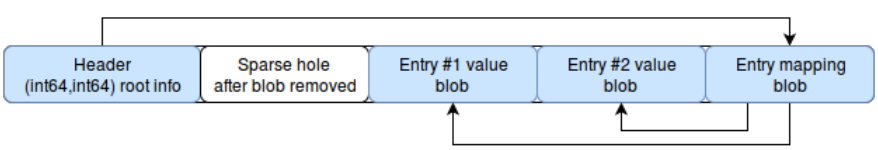
Compactness can be achieved by storing intents instead of full structures. For example, in log structured filesystem a single file write could end up storing the data buffer, indirect blocks, file inode, parent directory data and inode, and so on up to root inode. Instead only the data buffer and an intent describing to which file this data belongs to would be enough. Inode has to be stored eventually but not before file write operation was persistent, neither between open-close cycles. Recovery involves doing a so called *roll forward*, scanning the log from last known end forward, looking for intents. Each intent references the extent holding data (), a checksum that verifies said data, another checksum that verifies intent integrity, and some information that confirms the chain of intents (so stale bytes from whatever was stored on disk previously can be distinguished from actual intent. On recovery, if complete inode was not saved properly then last complete copy is loaded, changes defined in intents have to be merged, incorporated into in-memory inode. Final complete copy is then stored. Complexity is added to recovery code but performance and utilisation gain may well be worth it.

Robustness can also be achieved in the sense that operations can become persistent as soon as their intents hit the disk. To maximize durability, that is shortening the time from operation returning success to being persistent, intents could be appended to log immediately after each operation, with each buffer in case of files writes. However, this approach also maximizes fragmentation because even between successive sequential file writes there is always an intent allocated in middle. On the other hand, garbage collection will eventually move the data, both defragmenting (joining buffers) and compacting it (getting rid of intents). If durability is a goal, this approach could be taken into consideration.

Fragmentation can be actively avoided and compactness preserved by associating each operation with a global incremental generation number. Inodes affected by given operation are loaded from disk to memory (if needed), having their last generation field updated to current counter value. Before checkpointing, all in-memory inodes having last generation field less than some arbitrary generation are allocated in the log and sent to disk. When these disk writes were persisted, first block is atomically updated. This approach allows to implement delayed allocation, coalescing sequential file writes and storing only one intent in total thus actively avoiding fragmentation and increasing compactness.

[scheme needs a complete rework]

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



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

Encrypting disk blocks is not useful.

Encryption became a common feature that is not used only by geeks and criminals anymore. While in the past user would use some custom program to password-protect archives and store them away with other files on otherwise unencrypted media, today operating systems 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 must be loaded before any decryption can be done. Some files need to be available in the clear before rest of them can be decrypted and booting sequence can proceed.

Linux uses a separate unencrypted /boot partition to start a booting process. Then a screen pops up asking the user to provide a password which decrypts a LUKS partition. There is a reason why one partition cannot be enough to boot a system. LUKS partitions are encrypted block-wise. There is no facility to keep selected blocks in the clear. Filesystems are mounted on arbitrary block devices and are unaware of how (or whether) underlying blocks are being encrypted.

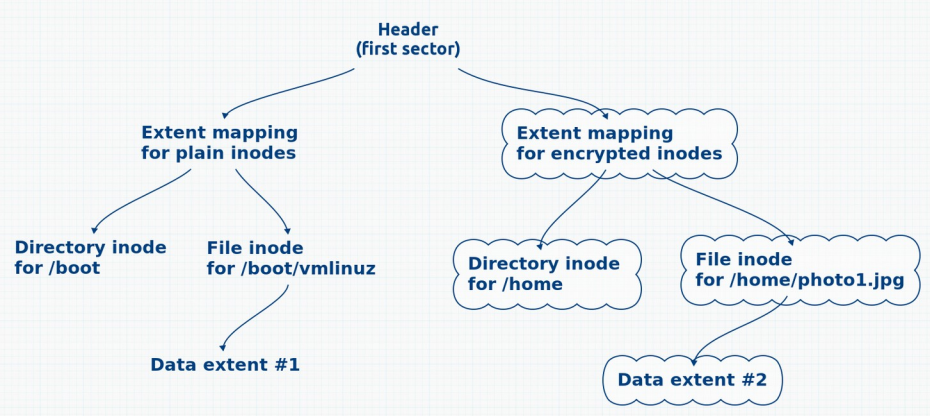
Only solution to booting from a (partially) encrypted filesystem seems to be encrypting selected data structures within filesystem itself and not the block storage device underneath.

Consider a filesystem where encryption is applied to data structures like inodes and data extents, and not raw disk blocks. Each data structure is encrypted individually and initially free space is filled with random bytes. Header is stored at a known location and keeps two addresses in the clear: one pointing to unencrypted data structure and one pointing to encrypted data structure. Following one address would allow discovery of all files stored in the clear, while second address would allow discovery of encrypted files but only after password was provided.

Encrypted structures would be indistinguishable from free space so there is no way to tell how much data is stored aside of files stored in the clear. When encrypted blocks are freed they just become part of the garbage in the background. Freeing blocks from files stored in the clear should be followed by overwrite with random garbage, although erasing can be postponed, resulting in space being unavailable for time being. Biggest disadvantage is that entire block device should be filled with random bytes before being used, although that also can be postponed. Filesystem creation would happen instantaneously but several hours would pass before entire device is filled and full security is guaranteed. Until then, an examination would reveal that certain segments are not encrypted with high probability. Content would not be revealed, only whether it is encrypted or not and an upper bound on usage quota, and only for first few hours of operation. Eventually entire disk would become filled with random bytes mixed with encrypted bytes.

Writing files in the clear before unlocking would be impossible since without password there is no way to distinguish used space from free space to allocate from, unless space was reserved in advance. Then files like boot logs could be stored in the clear even before unlocking.

Following diagram shows an example of data structures on disk (clouds mean extents are encrypted, locations on disk are not represented):

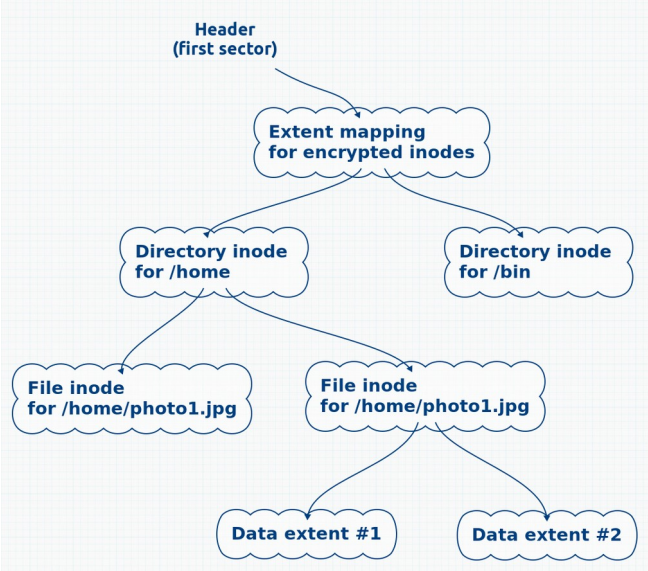


Overwriting files is not safe or efficient.

Wiping is an operation that renders a file unreadable, so as it is not possible to recover the content. Historically, and even to this day, this is achieved by overwriting content with zero or random bytes. This approach is definitely not fast or safe. Throughputs of modern hard disks is about 120 MB/s, and due to file fragmentation 8 ms of seek time per fragment will have to be added. For large or highly fragmented files this is going to take a long time. Also, magnetic force microscopy [Gutmann96] allows to recover overwritten data from disk platters, at least to some degree. To be sure, data should be overwritten up to 35 times. As for safety, files are being overwritten through the standard abstracted interface that provides no means of checking or changing how overwriting is effectuated. Modern filesystems like Btrfs tend to use copy on write strategy so overwriting files has no effect on already allocated blocks. Also when a file gets truncated, filesystem loses track of deallocated blocks. User space tools do not have an interface that would allow to query where these blocks are, or to allocate them specifically. Usually filesystems do not keep track of those blocks anyway.

Encrypted LUKS partitions are a partial solution, because they allow only to wipe entire filesystems and not individual files. However, the idea of keeping cryptographic keys in a header, that are used to encrypt subsequent blocks is a good idea that can be taken further.

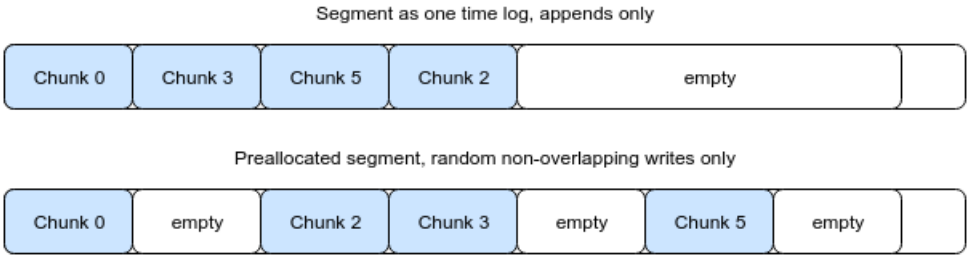
Encrypting data structures within a filesystem, instead of data blocks within the block device underneath, is an approach that allows to efficiently and safely wipe files, and for that matter any other data structures that the filesystem is divided into, from individual data extents that compose file content to entire directories instead of directory trees. Principle is the same as in previous section, and also shown on diagram below. Header contains a key to a data structure that contains keys to further data structures which also contain keys to further data structures. This creates a tree of both cryptographic keys and data structures, and wiping of any key in the tree breaks decryptability of all data structures further down the branch.



Strategies for data allocation are mutually exclusive.

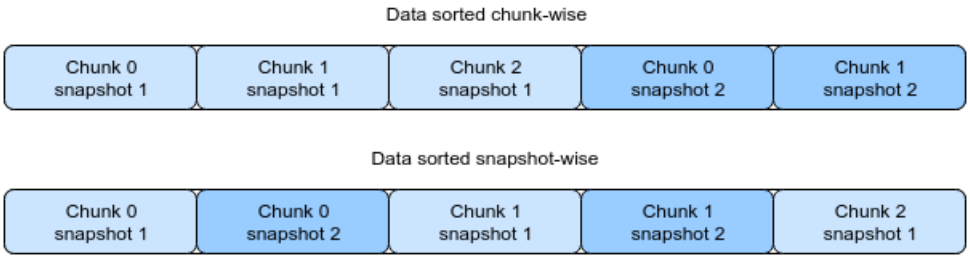
Segmented log structured filesystem is a layout where the entire disk is divided into segments, which are themselves one time logs. Filesystem selects an empty segment and appends data to it until that segment is full, and then selects a new segment. Concurrently, outdated segments can be read, compacted and defragmented internally, checksummed for integrity, recompressed, and finally stored to another location. Also, compacted segment could be split apart to fit into previously compacted leftover free space. There are however some mutually exclusive strategies how to allocate data within the segments, and whether is it always a good idea to append only to one segment.

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



Another issue is how chunks from across different snapshots are laid out within the segment. If file get snapshotted, truncated to zero and overwritten in entirety, then first layout is more beneficial. However, if a file gets snapshotted and overwritten in only a small subset of chunks, then second layout is more beneficial. Which layout is better depends on how much a particular

file is going to be overwritten. Applications do not report this to the filesystem, neither there seems to be an easy way to predict it for a particular file. During defragmentation this information is available, so layout can be optimised at least during defragmentation.

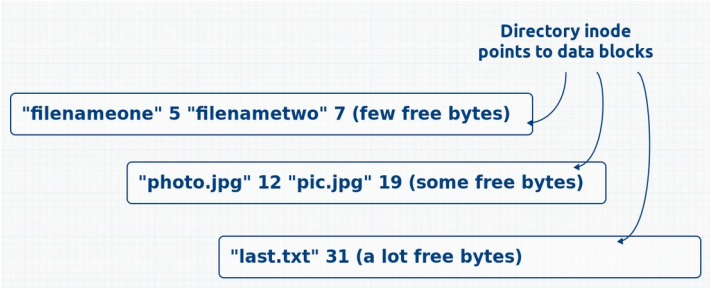


Blocks can store less than extents.

Earlier chapters discussed the issue of storing data structures like dictionaries and inodes in single extents. While previous chapters were considering performance of disk devices this chapter is focused on compactness. Less bits are needed to store data in extent form than in individual blocks, leading to performance and space utilization gain.

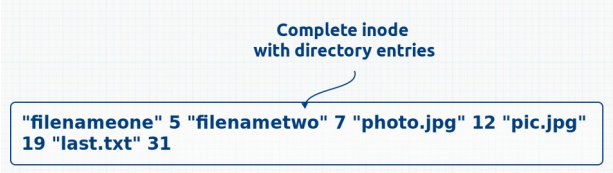
The main reason why extents are more efficient is that blocks impose rigid boundaries between data items stored inside. Items of variable size cannot be compactly laid out next to one another because boundaries between sets of items often not align with boundaries between blocks. When data is divided into blocks, usually items should not be split between blocks.

For example, directory entries used to be stored in data blocks like on a diagram below. File names are much shorter than a single block so they are being grouped to occupy as much space in each block as possible. However, since file names are variable length each block has some unpredictable amount of space left unused.



No doubt this approach was beneficial when memory was heavily constrained. Listing and querying entries requires processing only one block at a time. Removing or changing an entry requires writing only one block to disk. New entries can be added to an existing block in place of already removed entries, or can allocate a new block. All of these operations can be carried out with only one block in memory at a time, although keeping them in the buffer cache would be beneficial.

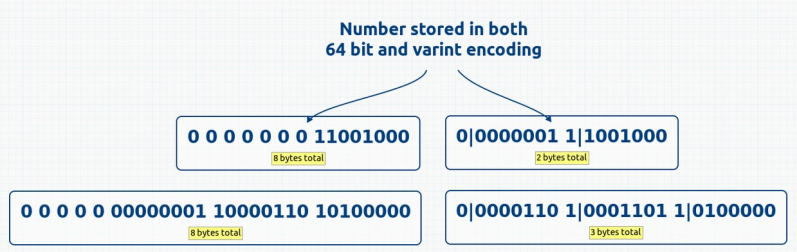
Consider using an extent instead. Items can be laid out next to each other with not even a byte wasted. At most, extent could be aligned to block boundaries wasting only some fragment of the last block, often called a *tail*. However if filesystem would allow extents to start and end at byte precise addresses then literally no byte would go to waste. This is the case with Btrfs filesystem.



Another example, dictionary that maps integers into arbitrary values can benefit from having all keys available when dictionary is being serialized into a binary blob (an extent). When keys are kept in separate blocks only few keys may be processed at once. Loading and storing entire dictionary each time becomes necessary when dictionary is supposed to always occupy a single extent.

Keys can be arbitrarily reordered within the blob because deserialization process does not care. If keys are sorted in increasing order then instead of keys themselves only their pairwise differences need to be stored. Differences are smaller than entire keys, on average and however else we choose to look at them. Small numbers can be encoded more efficiently using varint encoding which is explained right below. Values may be stored after each corresponding key and be also encoded.

Normally a 64 bit number occupies exactly that much space, 8 bytes. Varint encoding divides a number into 7 bit chunks starting from least significant bits. 8th bit is used to signal whether more chunks follow. For example, numbers 200 and 100'000 are encoded as below. Each byte is separated with a space and bits are concatenated with a pipe. Chunks are shown in binary system.



Varint encoding was developed by Google and used in their serialization framework Protocol Buffers.

Bibliography

Rosenblum91: Mendel Rosenblum and John K. Ousterhout, The Design and Implementation of a Log-Structured File System, 1991
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
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/
Rodeh12: Ohad Rodeh, BTRFS: The Linux B-tree Filesystem, 2012
ZFS: Jeff Bonwick, Bill Moore, ZFS: The Last Word in File Systems, ,
Agraw07: Nitin Agrawal, William J. Bolosky, John R. Douceur, Jacob R. Lorch, A Five-Year Study of File-System Metadata, 2007
Patterson: David A. Patterson, Kimberly K. Keeton, Hardware Technology Trends and Database Opportunities,
Pillai13: Thanumalayan Sankaranarayana Pillai, et al, Towards Efficient, Portable Application-Level Consistency, 2013
Gutmann96: Peter Gutmann, Secure Deletion of Data from Magnetic andSolid-State Memory, 1996

Please know that all cited documents are saved in project's repository and you do not have to search the web to read them. This includes cited web pages.