Ideas and Observations
Foundations for a versatile filesystem

Arkadiusz Bulski

Abstract

Cameleonica is a prototype of a highly versatile filesystem. This paper describes a wide range of concepts that could be incorporated into popular filesystem designs. Both potential benefits and reasons why these features were not popularized earlier are presented. Expectations of their performance is based on theoretic models.

[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. For example, Ext4 is an upgrade to Ext3 in sense that upgrade can be done in place.

[work in progress]

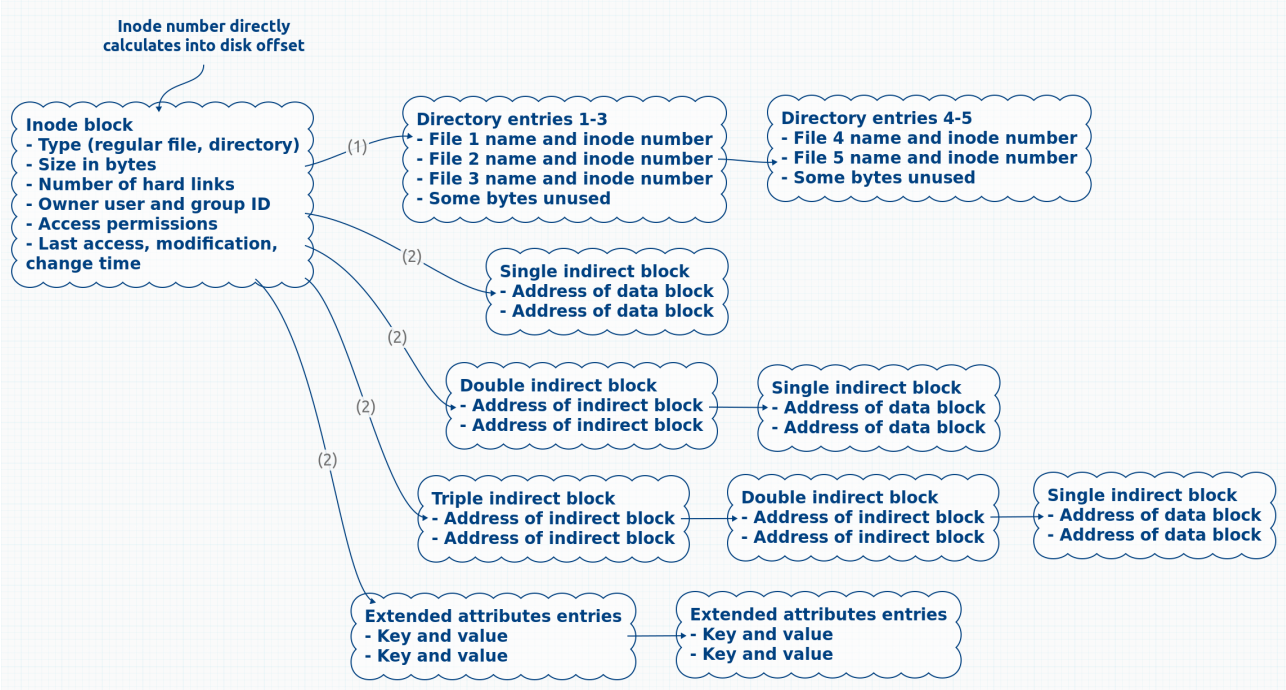
New ideas

Inodes are not efficient.

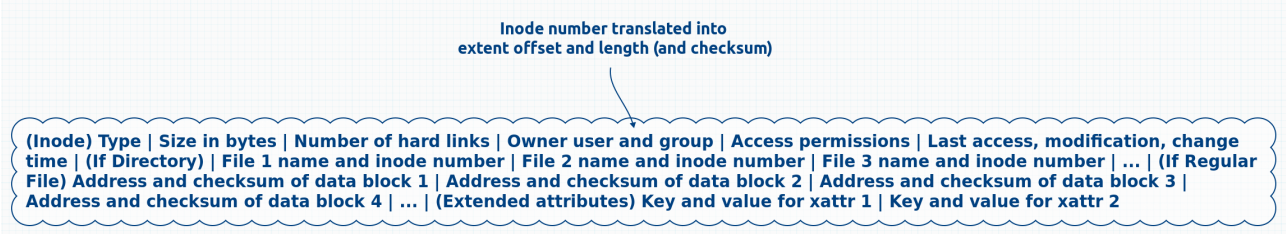
Since the early days, filesystems were designed under a principle that data structures have to be broken down into fixed size blocks. This seems to have been due to the fact that hard disks require read and write operations to be carried out on 512 byte long, discrete blocks of data. In traditional designs most operations can be done by processing one block at a time. Computers of the past often had little memory [McKusi84] 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], used for guaranteeing consistency after crash. 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 Ext2/3/4, 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.



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. 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 at once and after changes are made all blocks are stored to another continuous area at once. Last approach we shall call *block approach*. It is prevalent in modern filesystems. Space is allocated eagerly and non-preemptively, one block at a time 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. This kind of reasoning is flawed because it does not account for future operations. Further analysis considers amortization of performance over most of lifespan of a file, and not assumes 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 precluded due to reasons explained elsewhere. 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. Parent directory metadata is also excluded. At time zero file is assumed to already exist, with all content and metadata. 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.

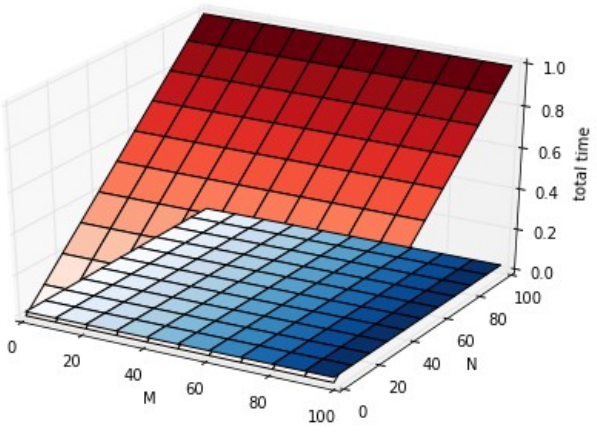
Reading or writing N blocks takes in total (assuming sequential and random pattern, referring to extent and block approach respectively):

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

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.

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 (non-preemptively) from entire disk (independently from uniform distribution). It would be possible to allocate blocks close to each other whenever possible, increasing so called *spatial locality*.

Secondly, disks can reorder outstanding operations to minimize total seek time. 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 read them all concurrently.

However, arguably neither issue would trump argument being made. Seek time and rotational delay are quite expensive, and considering hardware trends [Patterson], will

be even more expensive in the future. Constructing a theoretical model that includes clustered allocation and concurrent disk operations is outside of scope of this paper. Distribution parameters are not known.

Changes pending: Random seeks within a subset (maybe 1GB) would be more adequate. Manufacturers do not provide such data but it could be empirically measured.

B-trees are not efficient.

There is a trend among filesystem designs towards using both modified in place 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 is that memory needed to query and update trees is small, comparable to number of nodes from root to leaf. 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 phones constitute an exception but that does not negate benefits on desktops and servers.

Consider a hybrid or rather a transitional approach. Initially all entries are loaded from one extent, kept in memory in entirety and the whole time, and occasionally stored to disk as one extent. If at some point amount of entries grows over a certain threshold, a transition to a B-tree representation is commenced, all entries (already in memory) are relocated into tree nodes and stored to disk in one sweep. The threshold can be chosen low enough so storing entire dictionary in one sweep is faster than analogous B-tree operation (few disk seeks). After transition, operations are carried out on the B-tree representation. If changes are being accumulated over some period of time then comparison is even more favorable. Intents (patches) can be stored to disk to persist individual operations instead of storing the entire dictionary every time. Intents are described in a later chapter on fsync.

The threshold would actually be very high. We need to recognize that magnetic hard disks have quite skewed performance characteristics. Flash based storage is precluded due to reasons explained elsewhere. 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. Therefore a B-tree operation requiring 2 seeks or more takes on average more time than reading or writing an entire 1.2 MB in one sweep. A huge amount of entries can be stored in that amount of space.

Actually even more entries can be stored on disk than in memory. If for example entry

keys are sorted then their pairwise differences can be stored instead. Smaller numbers can be encoded more compactly using varint encoding. This topic is further explained in a later chapter on extent efficiency.

This argument is basically the same as for inode extent. If mapping entries are considered as metadata of some special file then same theoretical model can be applied.

Changes pending: Random seeks within a subset (maybe 1GB) would be more adequate. Manufacturers do not provide such data but it could be empirically measured.

Fsync is not efficient.

Files stored on computers were important since about the time when files started being stored on computers. Programs started using techniques that would ensure reasonable state after computer was interrupted in it's work 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 people started skipping on fsync. Later on ordering behavior was disabled to increase performance and non-standard code started corrupting files. This story has been described in LWN post POSIX vs Reality.

Another example is a property called *safe new file* that makes fsync on parent directory unnecessary when a new file is created. This behavior (and few others) is common in modern filesystems but it is not mandated by POSIX 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 standard (POSIX). Notice that all of the 6 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 remain to use the approach established by POSIX and filesystems will have to cooperate. Inventing new APIs that ensure safety while 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. Not used anywhere. 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. 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, 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 be priorities over others. 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 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 what a log-structured filesystem would contain.

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 writes need not to be neither atomic nor ordered. Disk writes can be reordered and interleaved with outstanding 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. Rather it provides a baseline upon which improvements will be made.

When 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 stored. 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 carries a signature that verifies both intent integrity (checksum), data it references elsewhere in the log, in case of file write operation (checksum) and that it represents actual intent, not stale bytes from whatever was stored on disk previously (generation number). 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 to being persistent, intents could be appended to log immediately after with each buffer. However this 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 getting rid of intents. If durability is a goal, this approach could be seriously taken into consideration.

Avoided fragmentation together with compactness can be achieved by assigning global incremental generation numbers to each operation. Inodes affected by given operation are loaded from disk (if not yet present), operation is applied to in-memory inodes, last generation number is updated in inodes. Before checkpointing, all in-memory inodes having last generation number less than some arbitrary generation are allocated in the log and sent to disk. Afterwards first block is atomically updated. This approach allows to implement delayed allocation, coalescing sequential file writes thus actively avoiding fragmentation and storing only one intent thus increasing compactness.

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. 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.

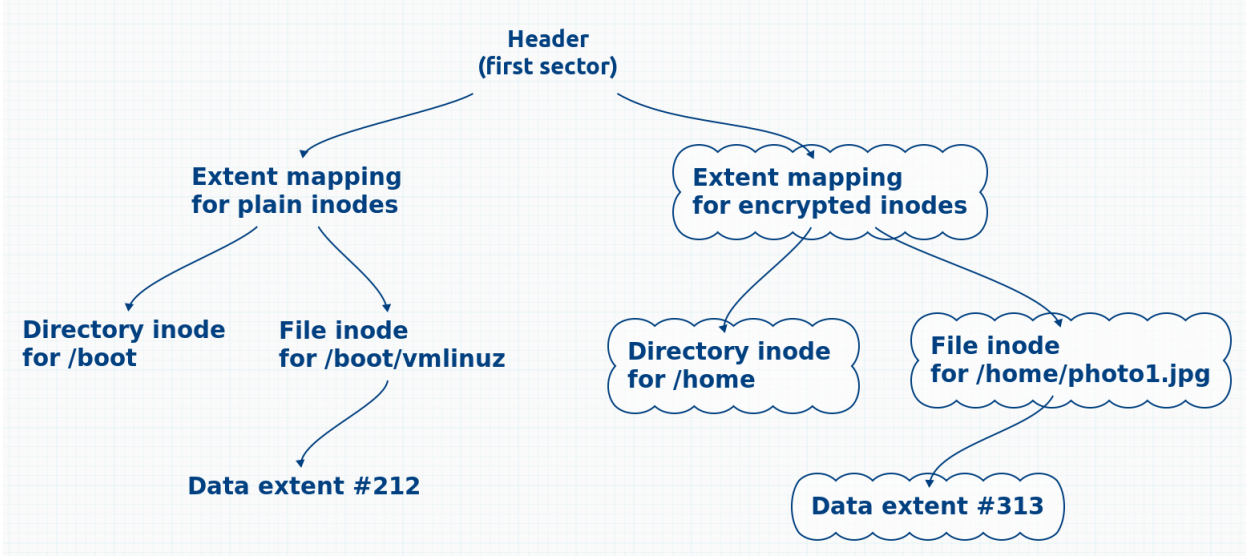
Only solution to booting from a (partially) encrypted filesystem seems to be encrypting filesystem itself and not the block storage device underneath.

Consider a filesystem where encryption is applied to data structures like inodes and not raw 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 be useful 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 it does not need to happen right away. 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 allow to determine that a progressively shrinking fraction of blocks are not in use. Content would not be revealed, only an upper bound on usage quota, and only for first few hours of operation.

Writing files in the clear before unlocking would be impossible since without password there is no way to distinguish 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.

[work in progress]

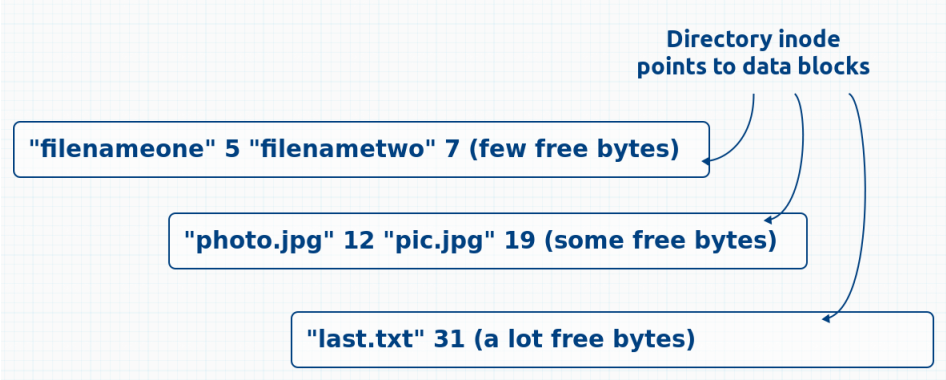
Blocks can store less than extents.

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 utilisation gain.

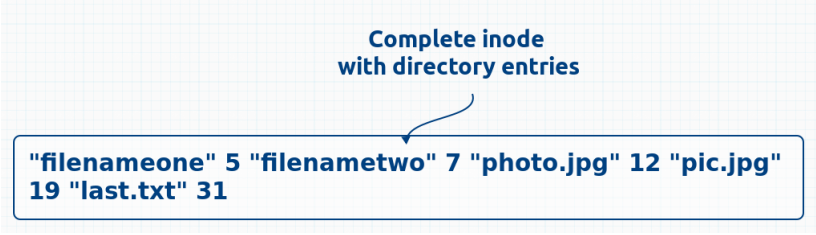
The main reason why extents are more efficient is that blocks impose rigid boundaries between data items stored inside. Items of variable size cannot be compactly laid out next to one another because boundaries between sets of items often not align with boundaries between blocks. When data is divided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

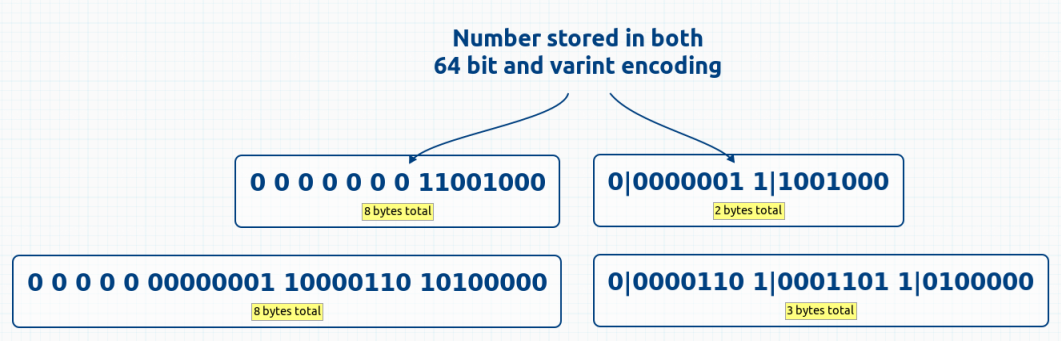


Another example, dictionary that maps integers into arbitrary values can benefit from having all keys available when dictionary is being serialized into a binary blob (an extent). When keys are kept in separate blocks only few keys may be processed at once. Loading and storing entire dictionary each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

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,
Rodeh12: Ohad Rodeh, BTRFS: The Linux B-tree Filesystem, 2012
ZFS: Jeff Bonwick, Bill Moore, ZFS: The Last Word in File Systems, ,
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
Pillai13: Thanumalayan Sankaranarayana Pillai, et al, Towards Efficient, Portable Application-Level Consistency, 2013

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.