Crash consistency

The file system data structures must persist, i.e., they must survive over the long haul, stored on devices that retain data despite power loss.

- The system may crash or lose power between any two writes, and thus the on-disk state may only partially get updated and leads to **crash-consistency problem**.
- After the crash, the system boots and wishes to mount the file system again (in order to access files and such).
- Given that crashes can occur at arbitrary points in time, how do we ensure the file system keeps the on-disk image in a reasonable state

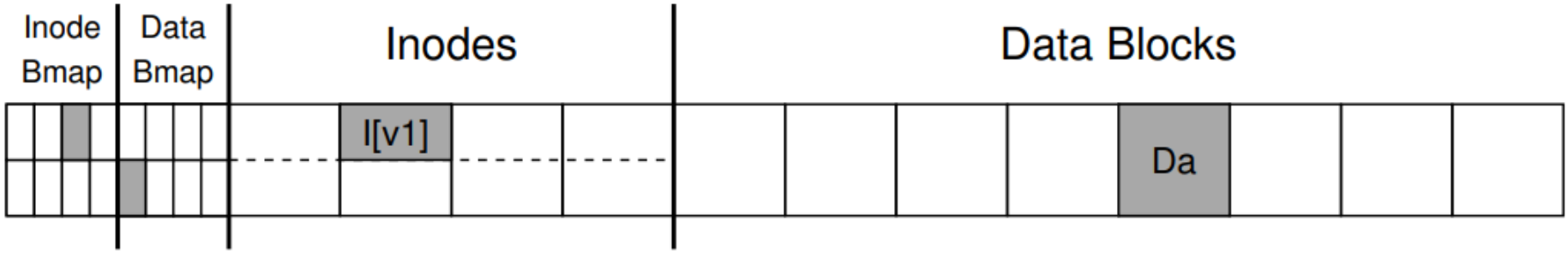
We will see the file system checker (*fsck* used in old system and also delve in detail with the *Journalling system; a.k.a wriet-ahead logging* used in recent FSs (Linux ext3).

Updation of data structure for a simple task

Let us consider the task of appending a single data block to an existing file. The steps primarily are

- Open the file; use lssek() to get the file offset to the end of the file.
- issue 1 block write and close the file

For a hypothetical system say we have 1 byte i-node (1 bit per i-node) and 1-byte data bitmap for 8-i-node and 8 data blocks. We start with i-node version 1 and a data block Da as shown in the figure. Note that i-node bitmap says i-node 2 is used while data bitmap indicates block 4 is used. Inside the i-node we may have information like
owner : CST; permission: RO; size: 1; ptr :4; ptr : NULL: ptr : NULL

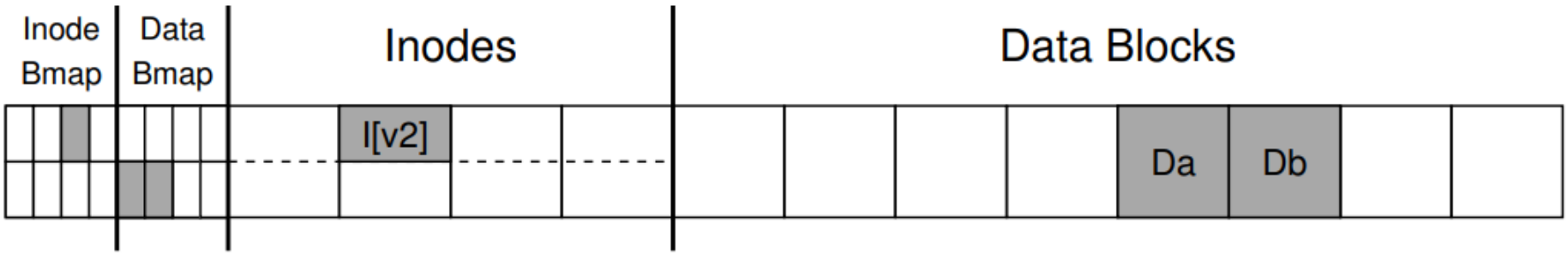


Updation of data structure for a simple task...contd.

When we append to the file, we are adding a new data block to it, and thus must update three on-disk structures:

- the inode (which must point to the new block as well as have a bigger size due to the append)
- , the new data block Db,
- and a new version of the data bitmap (call it B[v2]) to indicate that the new data block has been allocated.

Thus, in the memory of the system, we have three blocks which we must write to disk. The updated inode (inode version 2, or I[v2] for short) now looks like this: owner : CST; permissions : RO; size : 2; ptr: 4; ptr: 5; ptr : null pointer : null



Inconsistency: Pending write(s) due to crash

To achieve this transition, the file system must perform three separate writes to the disk, one each for the inode (I[v2]), data bitmap (B[v2]), and data block (Db).

- these writes usually don't happen immediately when the user issues a write() system call; rather, the dirty inode, bitmap, and new data will sit in main memory (in the page cache or buffer cache) for some time first;
- when the file system finally decides to write them to disk (after say 5 seconds or 30 seconds), the file system will issue the requisite write requests to the disk.
- Unfortunately, a crash may occur and thus interfere with these updates to the disk.
- if a crash happens after one or two of these writes have taken place, but not all three, the file system could be left in an inconsistent state.

Crash scenario

Let us see what happens for the completion of single write before the crash

- Data block (Db) is written to disk. There is no inode that points to it and no bitmap that even says the block is allocated. Thus, it is as if the write never occurred. So, no problem at all.
- I[v2]) is written to disk that points to the disk address (5). So, we have garbage in disk address (5) instead of Db. Moreover, Data bitmap didn't say disk address (5) is allocated but i-node bitmap indicates it is allocated. FS inconsistency.
- B[v2] is written to disk indicating that block 5 is allocated, but there is no inode (and no data either) that points to it. This inconsistency would lead to space leak if it is not repaired.

Crash scenario ... contd

Let us see what happens for the completion of two writes before the crash

- The inode (I[v2]) and bitmap (B[v2]) are written to disk, but not data (Db). Here, metadata wise everything is perfect except there is garbage as Db is not written
- The inode (I[v2]) and the data block (Db) are written, but not the bitmap (B[v2]). In this case, we have the inode pointing to the Db but data-bitmap says it has not allocated the block. Thus, inconsistency once again
- The bitmap (B[v2]) and data block (Db) are written, but not the inode (I[v2]). However, even though the block we have no idea which file it belongs to, as no inode points to the file.

So, out of 6 possible events 5 lead to inconsistency in the file system and that must be addressed.

Solution : FSCK

Early file systems took a simple, but lazy, approach (File system check) to crash consistency – let inconsistencies happen and then fix them later (when rebooting). Note that such an approach can't fix all problems; consider, for example, the case above where the file system looks consistent but the inode points to garbage data. The only real goal is to make sure the file system metadata is internally consistent. FSCK runs in phases to check consistency –

- Superblock is checked first to find inconsistency like mismatch in the file system size.
- Free blocks: Next, fsck scans the inodes, indirect blocks, double indirect blocks, etc., to build an understanding of which blocks are currently allocated within the file system. It uses this knowledge to produce a correct version of the allocation bitmaps; thus, if there is any inconsistency between bitmaps and inodes, it is resolved by trusting the information within the inodes.
- Inode state: Each inode is checked for corruption or other problems. fsck makes sure that each allocated inode has a valid type field (e.g., regular file, directory, symbolic link, etc.). If it cannot be fixed the corrupt inode is deleted and inode bit map is updated.

FSCK ...contd.

- Inode links: fsck also verifies the link count of each allocated inode. As you may recall, the link count indicates the number of different directories that contain a reference (i.e., a link) to this particular file. To verify the link count, fsck scans through the entire directory tree. If there is a mismatch it is corrected usually by fixing the count within the inode. If an allocated inode is discovered but no directory refers to it, it is moved to the lost+found directory.
- Duplicates: checks for duplicate pointers, i.e., cases where two different inodes refer to the same block.
- Bad blocks: bad block pointers check is also performed while scanning through the list of all pointers. A pointer is considered "bad" if it obviously points to something outside its valid range, Here, fsck just removes (clears) the pointer from the inode or indirect block.
- Directory checks: fsck performs additional integrity checks on the contents of each directory, making sure that "." and ".." are the first entries, that each inode referred to in a directory entry is allocated, and ensuring that no directory is linked to more than once in the entire hierarchy

fsck – summary and pitfalls

As you can see, building a working fsck requires intricate knowledge of the file system; making sure such a piece of code works correctly in all cases can be challenging.
fsck worked well though they cannot entirely detect and correct all sorts of anomalies in older system with limited disk capacity.
However, fsck (and similar approaches) has problems

- they are too slow as scanning the entire disk to find all the allocated blocks and read the entire directory tree may take many minutes or hours.
- Performance of fsck, as disks grew in capacity and RAIDs grew in popularity, became prohibitive

At a higher level, the basic premise of fsck seems a bit irrational. Consider our example above, where just three blocks are written to the disk; it is incredibly expensive to scan the entire disk to fix problems that occurred during an update of just three blocks.

Another Solution: Journaling or write ahead logging

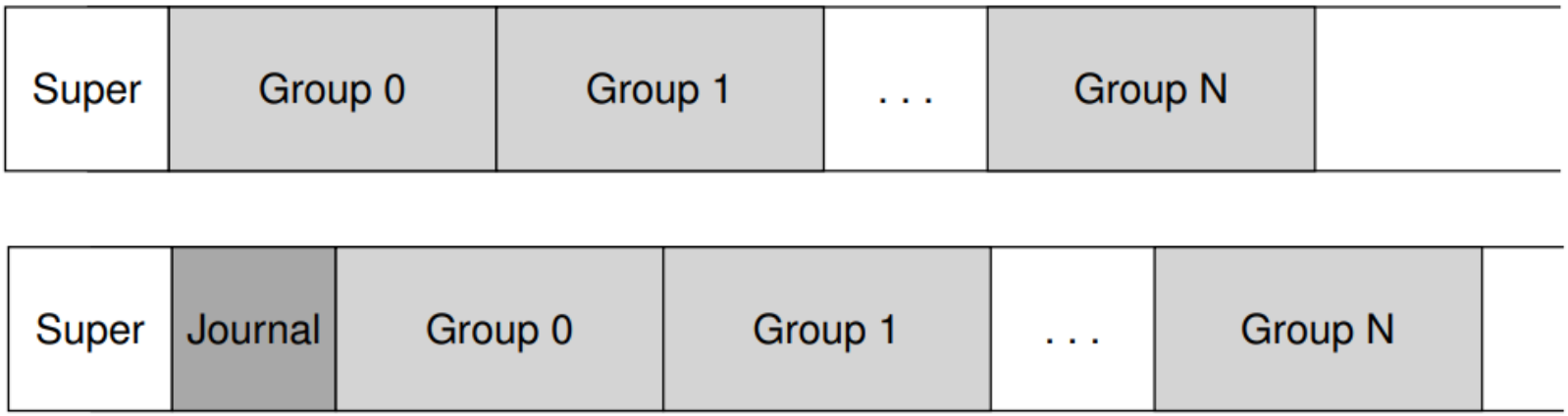
The most popular solution to the consistent update problem is known as write-ahead logging, was invented to address exactly this type of problem in DBMS. Many modern file systems use the idea, including Linux ext3 and ext4, and Windows NTFS.
The basic idea is as follows.

- When updating the disk, before overwriting the structures in place, first write down a little note (somewhere else on the disk, in a well-known location) describing what you are about to do.
- Writing this note is the "write ahead" part, and we write it to a structure that we organize as a "log"; hence, write-ahead logging.

By writing the note to disk, you are guaranteeing that if a crash takes places during the update (overwrite) of the structures you are updating, you can go back and look at the note you made and try again; thus, you will know exactly what to fix (and how to fix it) after a crash, instead of having to scan the entire disk. By design, journaling thus adds a bit of work during updates to greatly reduce the amount of work required during recovery.

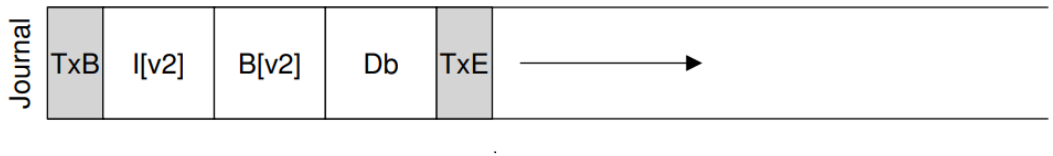
Journaling in LINUX EXT3 FS

Linux ext3 incorporates journaling into the file system. The new key structure is the journal itself, which occupies some small amount of space within the partition or on another device. Thus, an ext3 file system is similar to ext2 (see the first figure) with additional jornalling information (here in the figure we have assumed the journaling data within the same FS).
The most important aspect of journalling is the strategy – how are we going to use the journal data to recover the FS mantaining consistency.



Data Journalling

Say we take the previous example where we wish to write the 'inode (I[v2]), bitmap (B[v2]), and data block (Db) to disk again. Before writing them to their final disk locations, we are now first going to write them to the log (a.k.a. journal). This is what this will look like in the log:
We have written five blocks here. The transaction begin (TxB) tells us about this update, including information about the pending update to the file system (e.g., the final addresses of the blocks I[v2], B[v2], and Db), as well as some kind of transaction identifier (TID). The middle 3 blocks just contain the exact contents of the blocks themselves known as physical logging as we are putting the exact physical contents of the update in the journal (an alternate idea, logical logging, puts a more compact logical representation of the update in the journal, e.g., "this update wishes to append data block Db to file X", which is a little more complex but can save space in the log and perhaps improve performance). The final block (TxE) is a marker of the end of this transaction, and will also contain the TID.



Data Journalling ... contd

Once this transaction is safely on disk, we are ready to overwrite the old structures in the file system; this process is called checkpointing. Thus, to checkpoint the file system (i.e., bring it up to date with the pending update in the journal), we issue the writes I[v2], B[v2], and Db to their disk locations as seen above; if these writes complete successfully, we have successfully checkpointed the the file system and are basically done. Thus, our initial sequence of operations:

- Journal write: Write the transaction, including a transaction-begin block, all pending data and metadata updates, and a transaction end block, to the log; wait for these writes to complete.
- Checkpoint: Write the pending metadata and data updates to their final locations in the file system.

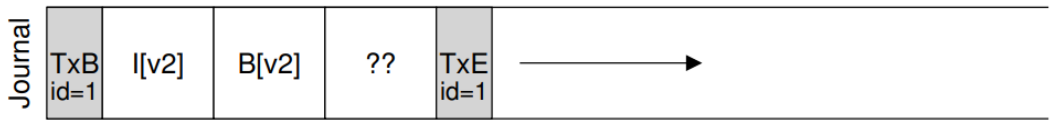
In our example, we would write TxB, I[v2], B[v2], Db, and TxE to the journal first. When these writes complete, we would complete the update by checkpointing I[v2], B[v2], and Db, to their final locations on disk.

Data Journalling ... contd

What happened if a crash occurs during the writes to the journal Here, we are trying to write the set of blocks in the transaction (e.g., TxB, I[v2], B[v2], Db, TxE) to disk.

- We may issue each write one at a time, waiting for each to complete, and then issuing the next.
- However, this is slow. Ideally, we'd like to issue all five block writes at once, as this would turn five writes into a single sequential write and thus be faster.
- However, this is unsafe, given such a big write, the disk internally may perform scheduling and complete small pieces of the big write in any order.

Thus, the disk internally may (1) write TxB, I[v2], B[v2], and TxE and only later (2) write Db. Unfortunately, if the disk loses power between (1) and (2), this is what ends up on disk:
This is bad for arbitrary user data in a file; it is much worse if it happens to a critical piece of file system, such as the superblock, which could render the file system unmountable.

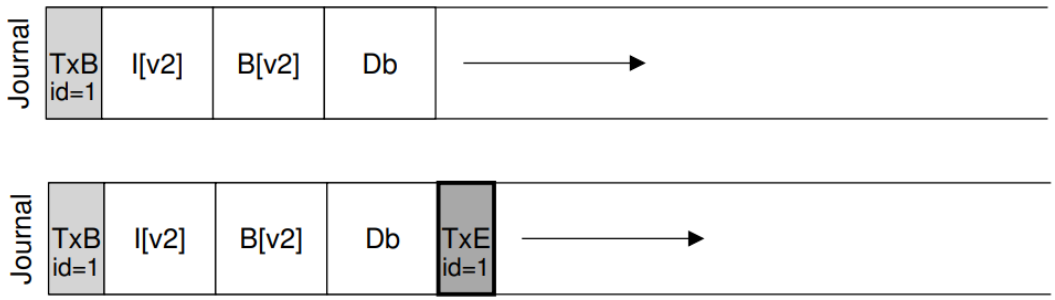


Data Journalling ... contd

To avoid problem cited above the file system issues the transactional write in two steps.

- First, it writes all blocks except the TxE block to the journal, issuing these writes all at once. When these writes complete, the journal will look something like the first figure (assuming our append workload again):
- When those writes complete, the file system issues the write of the TxE block, thus leaving the journal in this final, safe state (see second figure)

OPTIMIZING LOG WRITES: You may have noticed a particular inefficiency of writing to the log. Namely, the file system first has to write out the transaction-begin block and contents of the transaction; only after these writes complete can the file system send the transaction-end block to disk. The performance impact is clear, if you think about how a disk works: usually an extra rotation is incurred (think about why).



Data Journalling ... contd

An important aspect of this process is the atomicity guarantee provided by the disk. It turns out that the disk guarantees that any 512-byte (sector size) write will either happen or not (and never be half-written); thus, to make sure the write of TxE is atomic, one should make it a single 512-byte block.
Thus, our current protocol to update the file system, with each of its three phases labeled:

- 1. Journal write: Write the contents of the transaction (including TxB, metadata, and data) to the log; wait for these writes to complete.
- 2. Journal commit: Write the transaction commit block (containing TxE) to the log; wait for write to complete; transaction is said to be committed.
- 3. Checkpoint: Write the contents of the update (metadata and data) to their final on-disk locations. First, it writes all blocks except the TxE block to the journal, issuing these writes all at once. When these writes complete, the journal will look something like the first figure (assuming our append workload again):

Recovery

A crash may happen at any time during this sequence of updates; if it happens

- before the transaction is written safely to the log (i.e., before Step 2 above completes), then the pending update is simply skipped.
- If the crash happens after the transaction has committed to the log, but before the checkpoint is complete, the file system can recover the update as follows.

  When the system boots, the file system recovery process will scan the log and look for transactions that have committed to the disk; these transactions are thus replayed (in order), with the file system again attempting to write out the blocks in the transaction to their final on-disk locations. This form of logging is one of the simplest forms there is, and is called redo logging.

By recovering the committed transactions in the journal, the file system ensures that the on-disk structures are consistent, and thus can proceed by mounting the file system and readying itself for new requests.

LOG update batching

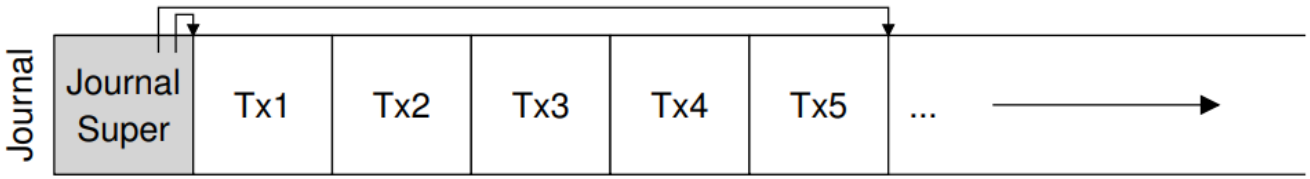
Journaling increases disk I/O in general and if we are not careful we have to come to a block with which we done writing just a couple of moments ago. Consider the case of creating two files *file1* and *file2*. Note that we need to modify minimally a number of structures for file creation; i) allocate i-node (update i-node bitmap); ii) write i-node; iii) data block of the parent directory to enlist this new directory entry; and iv) the updation of the parent directory i-node. For journaling we logically commit all of this information to the journal for each of our two file creations; because the files are in the same directory, and let's assume even have inodes within the same inode block, this means that if we're not careful, we'll end up writing these same blocks over and over.
FS like ext3 marks the relevant portion to be written as dirty (on the memory) and buffers all transaction to a single global transaction and when time comes (after 5 or 30 sec) it write the blocks to the disk to avoid accessing the same blocks repeatedly. Thus by buffering updates the FS can avoid excessive writing on the disk.

The LOG becomes infinite ... stop it

Journaling, as we see, increases the log continuously. So, the growth should be managed. In practice, this is achieved by having a circular log. To do so, the file system must take action some time after a checkpoint. Specifically, once a transaction has been checkpointed, the file system should free the space it was occupying within the journal, allowing the log space to be reused.
There are many ways to achieve this end; for example, you could

- simply mark the oldest and newest transactions in the log in a journal superblock; all other space is free. Here is a graphical depiction of such a mechanism.

The journal superblock (not the FS superblock) kept enough information to identify which transactions have not yet been checkpointed, and thus reduces recovery time as well as enables re-use of the log in a circular fashion. And thus we add another step to our basic protocol: 1. Journal write: 2. Journal commit: 3. Checkpoint: and 4. Free: i.e., after some time mark the transaction free in the journal by updating the journal superblock.



Metadata Journalling

Journaling increases dik I/O – by continuously updating the log; moreover it write the data block twice. As the datablock is much bigger than the metadata the cost of extra write leads to slower trnasactions.
One way to reduce transaction cost is obviously the metadata journaling (or ordered journaling) where the administrative data structure like the i-node(s) are journaled and not the data block. Thus, when performing the same update shown earlier the following information would be written to the journal (see image).
The data block Db, previously written to the log, would instead be written to the file system proper, avoiding the extra write; given that most I/O traffic to the disk is data, not writing data twice substantially reduces the I/O load of journaling.
The modification, however, does raise an interesting question, though: when should we write data blocks to disk? It has been found that this could be done as the first step (done in LINUX ext3) to avoid i-node pointing to a garbage data block. So, now the steps are i) Data write; ii) Journal metadata write; iii) Journal commit; iv) Checkpoint metadata; and finally v) Free: mark using journal superblock

Other methods

Other than fsck and journalling mechanism we have seen soft update technique to ensure consistency. One such approach, known as Soft Updates. This approach carefully orders all writes to the file system to ensure that the on-disk structures are never left in an inconsistent state. For example, by writing a pointed-to data block to disk before the inode that points to it, we can ensure that the inode never points to garbage; similar rules can be derived for all the structures of the file system. Implementing Soft Updates can be a challenge, however; whereas the journaling layer described above can be implemented with relatively little knowledge of the exact file system structures, Soft Updates requires intricate knowledge of each file system data structure and thus adds a fair amount of complexity to the system.
Another approach is known as copy-on-write (COW), and is used in a number of popular file systems. This technique never overwrites files or directories in place; rather, it places new updates to previously unused locations on disk. After a number of updates are completed, COW file systems flip the root structure of the file system to include pointers to the newly updated structures.

Log Structured File System

Observing that the i) Memory is growing; ii) random I/O performance is much poorer than sequential I/O; iii) Existing FS does a lot of write even for a simple workoload (like creating a file) etc. a new file system was developed and known as Log Structured FS.

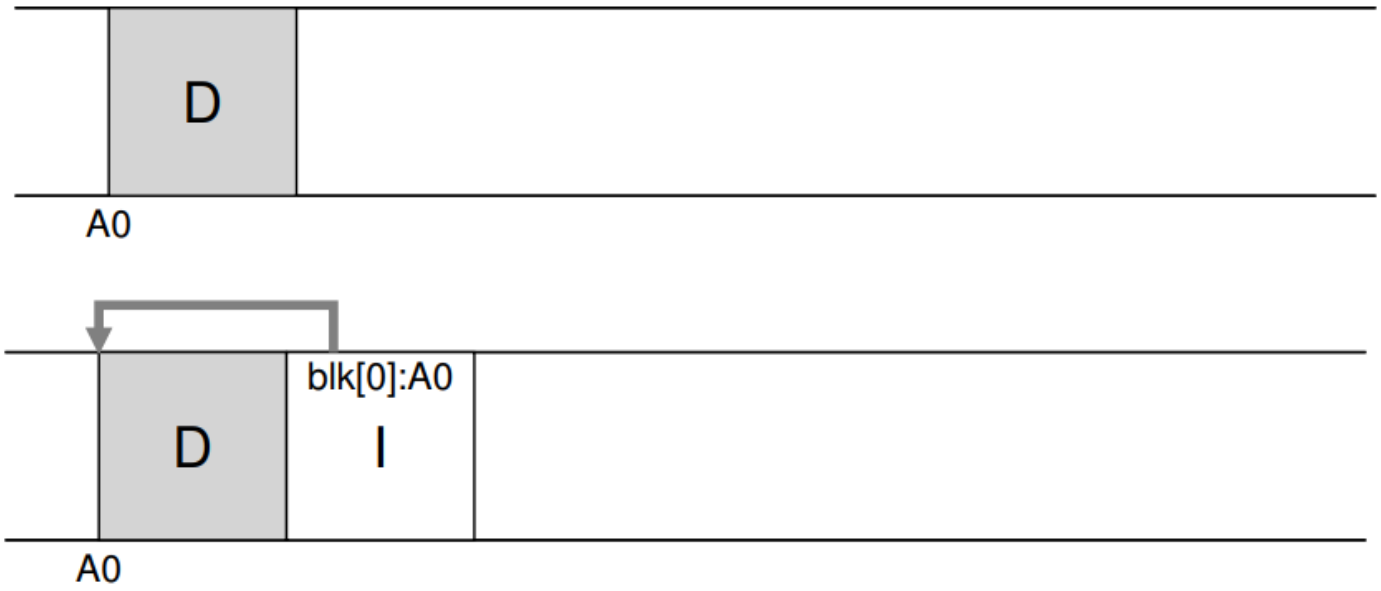
- When writing to disk, LFS first buffers all updates (including metadata!) in an in-memory segment; when the segment is full, it is written to disk in one long, sequential transfer to an unused part of the disk,
- So, LFS never overwrites existing data, but rather always writes segments to free locations. Because segments are large, the disk is used efficiently, and performance of the file system approaches its zenith

An ideal file system would focus on write performance and to do so it would try writing things sequentially possibly by buffering all the write requests together and issuing a long and single sequence of writes.

Writing disk sequentially

The first task in LFS is how we transform all updates to file-system state into a series of sequential writes to disk? Imagine we are writing a data block D to a file. Writing the data block to disk might result in the following on-disk layout, with D written at disk address A0.

Along with the data we also write the inode (I: 128 bytes) of the file to disk, and have it point to the data block D (4KB). When written to disk, the data block and inode would look something like this.

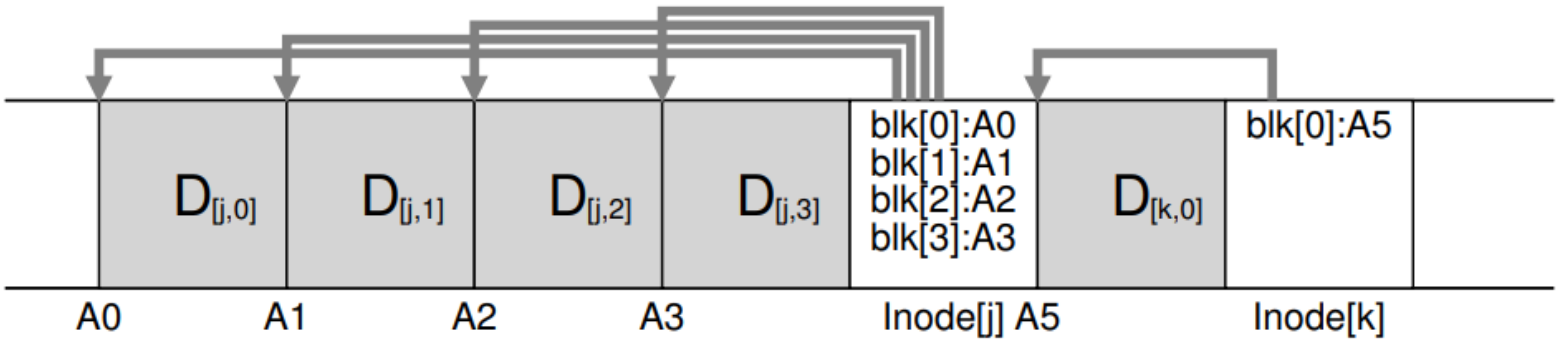


Sequentially and Effectively

Effective Sequential write : Unfortunately, writing to disk sequentially is not (alone) enough to guarantee efficient writes. For example, imagine if we wrote a single block to address A, at time T. We then wait a little while, and write to the disk at address A + 1 (the next block address in sequential order), but at time $T + \delta$. In-between the first and second writes, unfortunately, the disk has rotated; when you issue the second write, it will thus wait for most of a rotation before being committed (specifically, if the rotation takes time $T_{rotation}$, the disk will wait $T_{rotation}$ - δ before it can commit the second write to the disk surface). And thus you can hopefully see that simply writing to disk in sequential order is not enough to achieve peak performance; rather, you must issue a large number of contiguous writes (or one large write) to the drive in order to achieve good write performance.

Write a large chunk together

Before writing to the disk, LFS keeps track of updates in memory; when it has received a sufficient number of updates, it writes them to disk all at once, thus ensuring efficient use of the disk.
Here is an example, in which LFS buffers two sets updates into a small segment; actual segments are larger (a few MB). The first update is of four block writes to file j; the second is one block being added to file k. LFS then commits the entire segment of seven blocks to disk at once. The resulting on-disk layout of these blocks is as follows:



How much to buffer

Buffering optimization is needed to reach the pek transfer rate; i.e., how much buffering is optimal. This indeed depends on the postioning time and the transfer rate for the disk. Assume that we are writing D MB at a time. So, the write time would be

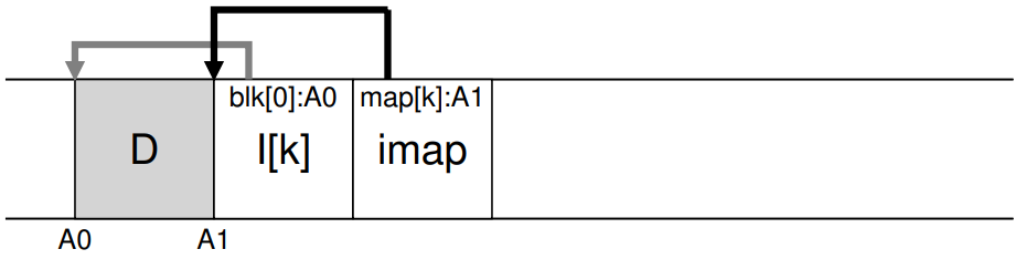
$$T_{write} = T_{position} + D/R_{peak}$$

and the effective rate of writing would be $R_{effective} = D/T_{write} =$ Technically we would try to achieve the effective rate as some factor (near to 1, say 0.9) of the peak rate. So, $R_{effective} = F \times R_{peak} = \frac{D}{T_{position}+D/R_{peak}}$
Now, solving for D we get $D = \frac{F}{1-F} \times R_{peak} \times T_{position}$ For example, if we take F (the effective bandwidth) =0.9 and $T_{position} = 10$ ms for a peak transfer rate, $R_{peak} = 100$ MB/s we get D = 9 MB.

Finding I-nodes

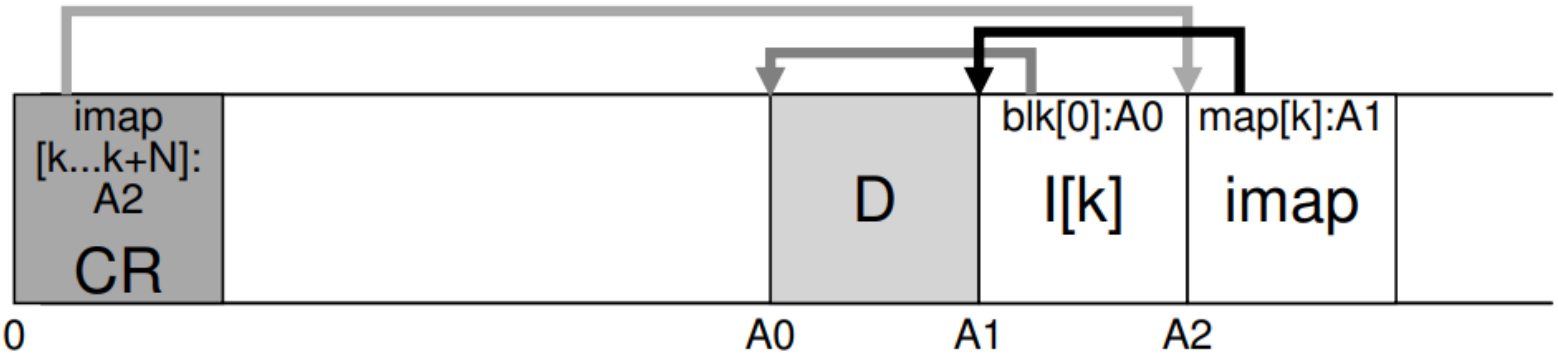
In LFS finding i-node is not easy as the i-node is scattered all through the disk. This problem is solved by using an imap; a structure that takes an inode number as input and produces the disk address of the most recent version of the inode– implemented as a simple array, with 4 bytes (a disk pointer) per entry. Any time an inode is written to disk, the imap is updated with its new location.
The i-map however should be made persistent (written to the disk). If it is stored in a fixed location on the disk then there would be more seeks with each update . LFS places chunks of the inode map right next to where it is writing all of the other new information. Thus, when appending a data block to a file k, LFS actually writes the new data block, its inode, and a piece of the inode map all together onto the disk, as follows:

In this picture, the piece of the imap array stored in the block marked imap tells LFS that the inode k is at disk address A1; this inode, in turn, tells LFS that its data block D is at address A0.



Checkpoint Region

Finding i-map is still a problem as they are scattered all over the disk but to reduce seek they cannot be kept in a known location. So, a region on a fixed location of the disk containing i-map pointers is the solution.
LFS has just such a fixed place on disk for this, known as the checkpoint region (CR). The CR contains pointers to (i.e., addresses of) the latest pieces of the inode map, and thus the inode map pieces can be found by reading the CR first. Note the checkpoint region is only updated periodically (say every 30 seconds or so), and thus performance is not ill-affected.
So, searching the checkpoint region we get the i-node map that contain addresses of the inodes; the inodes point to files (and directories) just like typical UNIX file systems.
[Note: in a real CR we have much more i-map chunks.]



Reading A File From Disk in LFS: A Recap

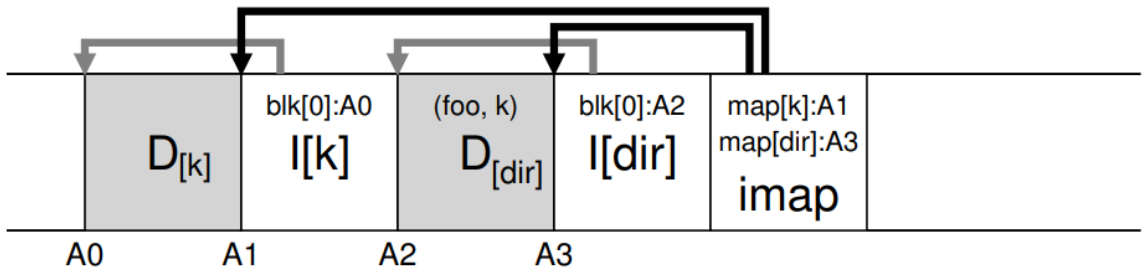
Assume we have nothing in memory to begin.

- The first on-disk data structure we must read is the checkpoint region. The checkpoint region contains pointers (i.e., disk addresses) to the entire inode map, and thus LFS then reads in the entire inode map and caches it in memory.
- After this point, when given an inode number of a file, LFS simply looks up the inode-number to inode-disk address mapping in the imap, and reads in the most recent version of the inode.
- To read a block from the file, at this point, LFS proceeds exactly as a typical UNIX file system, by using direct pointers or indirect pointers or doubly-indirect pointers as need be.

In the common case, LFS should perform the same number of I/Os as a typical file system when reading a file from disk; the entire imap is cached and thus the extra work LFS does during a read is to look up the inode's address in the imap.

Handling Directories

Fortunately, directory structure is basically identical to classic UNIX FS; just a collection of (name, inode number) mappings. When creating a file on disk, LFS writes both a new inode, some data, as well as the directory data and its inode that refer to this file. Remember that LFS will do so sequentially on the disk (after buffering the updates for some time). Thus, creating a file (say, foo) in a directory would lead to the following new structures on disk (see diagram)
The piece of the inode map contains the information for the location of both the directory file dir as well as the newly-created file f. Thus, when accessing file foo (with inode number k), you would first look in the inode map (usually cached in memory) to find the location of the inode of directory dir (A3); you then read the directory inode, which gives you the location of the directory data (A2); reading this data block gives you the name-to-inode-number mapping of (foo, k). You then consult the inode map again to find the location of inode number k (A1), and finally read the desired data block at address A0.



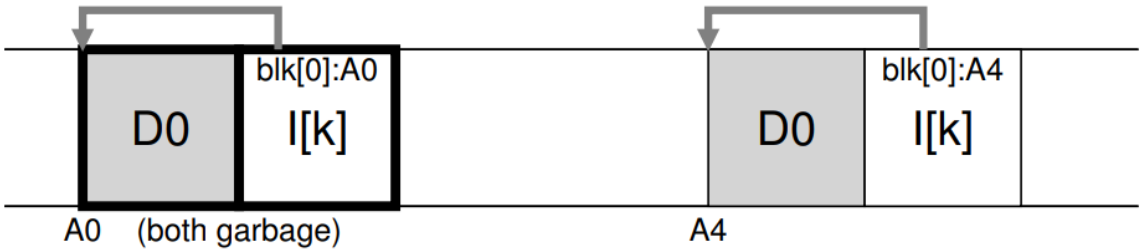
Recursive update problem

There is one other serious problem in LFS that the inode map solves, known as the recursive update problem. The problem arises in any file system that never updates in place (such as LFS), but rather moves updates to new locations on the disk.
Specifically, whenever an inode is updated, its location on disk changes. This would lead to an update to the directory that points to this file, which then would have mandated a change to the parent of that directory, and so on, all the way up the file system tree.
LFS cleverly avoids this problem with the inode map. Even though the location of an inode may change, the change is never reflected in the directory itself; rather, the imap structure is updated while the directory holds the same name-to-inumber mapping. Thus, through indirection, LFS avoids the recursive update problem.

Garbage Collection problem

LFS keeps writing newer version of a file, its inode, and in fact all data to new parts of the disk. This process, while keeping writes efficient, implies that LFS leaves older versions of file structures all over the disk, scattered throughout the disk. This is logically a garbage and we need to clean it up to make room.
For example, let's imagine the case where we have an existing file referred to by inode number k, which points to a single data block D0. We now overwrite that block, generating both a new inode and a new data block. The resulting on-disk layout of LFS would look something like this (note we omit the imap and other structures for simplicity; a new chunk of imap would also have to be written to disk to point to the new inode)

We see both the inode and data block have two versions on disk, one old (left) and one current and thus live (right). By the simple act of overwriting a data block, a number of new structures must be persisted by LFS, thus leaving old versions of said blocks on the disk.



Garbage Collection problem ... contd.

As another example, take append a block to that original file k. So, a new version of the inode is generated, but the old data block is still pointed to by the inode. Thus, it is still live and very much apart of the current file system:
One could keep those older versions around and allow users to restore old file versions (for example, when they accidentally overwrite or delete a file, it could be quite handy to do so); such a file system is known as a versioning file system because it keeps track of the different versions of a file. However, LFS instead keeps only the latest live version of a file; thus (in the background), LFS must periodically find these old dead versions of file data, inodes, and other structures, and clean them; cleaning should thus make blocks on disk free again for use in a subsequent writes. [Note cleaning is a form of garbage collection, a technique similar to automatically freeing unused memory for programs.]

Garbage Collection problem ... contd.

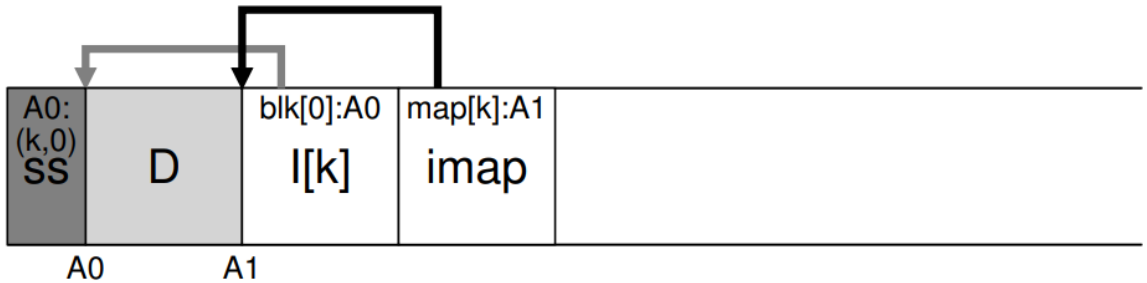
LFS only keep the latest live version of a file unlike the *versioning file system* . So, LFS must periodically find these old dead versions of file data, inodes, and other structures, and clean (a form of grabage collection) them; cleaning should thus make blocks on disk free again for use in a subsequent writes.
We know segments that enables large writes to disk in LFS. As it turns out, they are also quite integral to effective cleaning. Otherwise, cleaning would leave a lot of small holes mixed between allocated disk blocks and finding contiguous block that leads to poor performance.
Instead, the LFS cleaner works on a segment-by-segment basis, thus clearing up large chunks of space for subsequent writing. The basic cleaning process works as follows. Periodically, the LFS cleaner reads in a number of old (partially-used) segments, determines which blocks are live within these segments, and then write out a new set of segments with just the live blocks within them, freeing up the old ones for writing. Specifically, we expect the cleaner to read in M existing segments, compact their contents into N new segments (where $N < M$), and then write the N segments to disk in new locations. The old M segments are then freed and can be used by the file system for subsequent writes.

Garbage Collection problem ... contd.

We have two issues now i) How to find out live blocks and ii) How often the cleaner runs and the policy to pick up a segment for cleaning.

Determining block liveness: Given a data block D within an ondisk segment S, whether D is live (or Block liveness) is determined by adding little extra information to each segment (the *segment summary block*) For a block D located on disk at address A, look in the segment summary block and find its inode number N and offset T. Next, look in the imap to find where N lives and read N from disk (perhaps it is already in memory, which is even better). Finally, using the offset T, look in the inode (or some indirect block) to see where the inode thinks the Tth block of this file is on disk. For a match it is live else dead.

Here is a diagram depicting the mechanism, in which the segment summary block (marked SS) records that the data block at address A0 is actually a part of file k at offset 0. By checking the imap for k, you can find the inode, and see that it does indeed point to that location.



Garbage Collection problem ... contd.

Which block to clean and when: When we need cleaning is a simple policy – i) when system is idle or ii) when the disk is full. On the other hand picking which block is to be cleaned is not that starighforward.

An approach which tries to segregate hot and cold segment. A hot segment is one in which the contents are being frequently over-written; thus, for such a segment, the best policy is to wait a long time before cleaning it, as more and more blocks are getting over-written (in new segments) and thus being freed for use. A cold segment, in contrast, may have a few dead blocks but the rest of its contents are relatively stable. Thus, it is concluded that one should clean cold segments sooner and hot segments later, and develop a heuristic that does exactly that.

However, as with most policies, this is just one approach, and by definition is not "the best" approach; we have observed later approaches in literature that show how to do it better.

Crash recovery : LFS

(i) During normal operation, LFS buffers writes in a segment, and then writes the segment to disk. The log, i.e., the checkpoint region points to a head and tail segment, and each segment points to the next segment to be written. (ii) LFS also periodically updates the checkpoint region.

Crashes could clearly happen during either of these operations (write to a segment, write to the CR). So how does LFS handle crashes during writes to these structures? Let's cover the second case first. To ensure that the CR update happens atomically, LFS actually keeps two CRs, one at either end of the disk, and writes to them alternately. LFS also implements a careful protocol when updating the CR with the latest pointers to the inode map and other information; specifically, it first writes out a header (with timestamp), then the body of the CR, and then finally one last block (also with a timestamp). If the system crashes during a CR update, LFS can detect this by seeing an inconsistent pair of timestamps. LFS will always choose to use the most recent CR that has consistent timestamps, and thus consistent update of the CR is achieved.

Crash recovery : LFS ... contd.

Let's now address the first case.

- Because LFS writes the CR every 30 seconds or so, the last consistent snapshot of the file system may be quite old.
- Thus, upon reboot, LFS can easily recover by simply reading in the checkpoint region, the imap pieces it points to, and subsequent files and directories; however, the last many seconds of updates would be lost.

To improve upon this, LFS tries to rebuild many of those segments through a technique known as roll forward in the database community.

- The basic idea is to start with the last checkpoint region, find the end of the log (which is included in the CR), and then use that to read through the next segments and see if there are any valid updates within it.
- If there are, LFS updates the file system accordingly and thus recovers much of the data and metadata written since the last checkpoint.