**Chapter 42: Crash Consistency: FSCK and Journaling**

The file system must **persist**, i.e., it must remain despite of power loss or system crash.

We will examine the **file system checker** and **journaling** (or **write-ahead logging**).

**42.1 A Detailed Example**

We’ll need to use a workload that updates on-disk structures in some way. Here, we simply append a single data block to an existing file. This is accomplished by opening the file and calling lseek() to move the file offset to the end, and then issue a single 4KB write.

Chart, box and whisker chart

Description automatically generated

The inode is denoted I[v1]. Inside of the inode, we have:

Table

Description automatically generated

The size of the file is 1, the first direct pointer points to the block 4 (The block Da) and all three other pointers are null (they are not used).

When we append a block to the file, we must perform three steps:

1. Update the inode to point to the new block and record the new larger size.
2. Update the new data block Db
3. Update the new version of the data bitmap (B[v2]) to indicate the new data block has been allocated.

Thus, the updated version of the inode would be:

Table

Description automatically generated

And the on-disk image of the system would be:

Chart, box and whisker chart

Description automatically generated

To do this transition, the file system must perform three separated writes to the disk: one each for the inode, bitmap and data block. However, the writes usually do not happened immediately when the user issues a write() system call. Yhe dirty inode, bitmap, and new data will sit in main memory (**page cache** or **buffer cache**) for some time first. After some delay, the file system decides to write them to disk. However, this is where a crash could happen and interfere the write requests to the disk.

**Crash Scenarios**

If there is only a single write succeeds, there are three possible outcomes:

1. Just the data block (Db) is written to disk: there is no inode that points to it and no bitmap to tell that the block is allocated (as if the write never occurs). This is not a problem from the perspective of file-system crash consistency.
2. Just the inode is written to disk. The inode points to the disk address where Db was about to be written, but Db has not yet been written there. Thus, we will likely to read **garbage** data from the disk.

This will introduce a problem, called **file-system inconsistency**.

1. Just the updated bitmap is written to disk. The bitmap indicates that block 5 is allocated, but there is no inode that points to it. Thus, the file system is inconsistence. This would result in a **space leak** as block 5 (for example) would never be used by the file system.

There are many other cases where only two out of three writes succeed:

1. The inode and bitmap are written, but not data block. In this case, the system metadata is completely consistent. However, the only problem is that 5 has garbage.
2. The inode and data block are written, but not the bitmap. In this case, we have an inconsistency between the inode and the old version of bitmap.
3. The bitmap and the data block are written, but not the inode. We have inconsistency between the inode and the data bitmap. Even though the block was written and the bitmap indicates its usage, we have no idea which file it belongs to, as no inode points to the file.

**The Crash Consistency Problem**

What we’d like to do ideally is move the file system from one consistent state (e.g., before the file got appended to) to another atomically (e.g., after the inode, bitmap, and new data block have been written to disk). However, we cannot do this because the disk only commits one write at a time, and crashes or power loss may occur between these updates. We call this the **crash-consistency problem**.

**42.2 Solution #1: The File System Checker**

Basically, the file systems decided to let inconsistencies happen and fix them later (when rebooting). Such lazy approach is found in **fsck**, which is a UNIX tool for finding such inconsistencies and repair them. However, this approach cannot fix all the problems (for example, the inode points to garbage data). The only real goal is to make sure the file system metadata is internally consistent.

The fsck is run before the file system is mounted and made available as fsck assumes that no other file-system activity is on-going while it runs. Once finished, the on-disk file system should be consistent and thus can be made accessible to users.

Fsck does the following:

1. **Superblock**: fsck first checks if the superblock looks reasonable, mostly doing sanity checks such as making sure the file system size is greater than the number of blocks that have been allocated. Usually the goal of these sanity checks is to find a suspect (corrupt) superblock; in this case, the system (or administrator) may decide to use an alternate copy of the superblock.
2. **Free blocks**: 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. The same type of check is performed for all the inodes, making sure that all inodes that look like they are in use are marked as such in the inode bitmaps.
3. **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 there are problems with the inode fields that are not easily fixed, the inode is considered suspect and cleared by fsck; the inode bitmap is correspondingly updated.
4. **Inode linkes**: fsck also verifies the link count of each allocated inode. To verify the link count, fsck scans through the entire directory tree, starting at the root directory, and builds its own link counts for every file and directory in the file system. If there is a mismatch between the newly-calculated count and that found within an inode, corrective action must be taken, 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.
5. **Duplicates**: fsck also checks for duplicate pointers, i.e., cases where two different inodes refer to the same block. If one inode is obviously bad, it may be cleared. Alternately, the pointed-to block could be copied, thus giving each inode its own copy as desired.
6. **Bad blocks**: A check for bad block pointers 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. In this case, it just removes the pointer from the inode or indirect block.
7. **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 has a big problem that it is **too slow**. With a very large disk volume, 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 just a tad irrational as it is expensive to scan the entire disk to fix a small problem.

**42.3 Solution #2: Journaling (or Write-Ahead Logging)**

The idea is that 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. By design, journaling thus adds a bit of work during updates to greatly reduce the amount of work required during recovery.

The **Linux ext2** file system looks like this:

A picture containing text

Description automatically generated

Assuming the journal is placed within the same file system image, the **Linux ext3** file system looks like this:

A picture containing shape

Description automatically generated

**Data Journaling**

**Data journaling** is available as a mode with the Linux ext3 file system, from which much of this discussion is based.

Let’s assume that we wish to write the inode, bitmap and data block to disk again. Before writing them to the final disk locations, we write them to the log (journal):

Diagram

Description automatically generated with low confidence

We have written 5 blocks. 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), and **transaction identifier (TID)**. The middle three blocks just contain the exact contents of the blocks themselves, or **physical logging** as we are putting the exact physical contents of the update in the journal. The final block (TxE) is a marker of the end of this transaction, and will also contain the TID.

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, 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 file system and are basically done. Thus, our initial sequence of operations:

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

However, things get trickier when a crash occurs during the writes to the journal as we are trying to write the set of blocks in the transaction to disk. One simple way to do this would be to issue each one at a time, waiting for each to complete and then issuing the next. However, this is slow. Ideally, we would want to write all five blocks at once, but this is unsafe. This is because internally, the disk may schedule and complete small pieces (TxB, I[v2], B[v2] and TxE) and then the larger one (Db). If we lose power between them, it will look like this:

A picture containing graphical user interface

Description automatically generated

If the system reboots and runs recovery, it will replay this transaction and ignorantly copy the contents of the garbage block to the location Db lives. 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.

To avoid this problem, 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 this:

A picture containing diagram

Description automatically generated

When these writes complete, the file system issues the write of the TxE block, ending the journal in the safe state:

Graphical user interface, text

Description automatically generated

An important aspect of this process is the atomicity guarantee provided by the disk. It turns out the the disk guarantees that **any 512-byte write** will either happen or not. Thus, out protocol to update the file system is:

Text

Description automatically generated

**Recovery**

A crash can occur at any time. If it happens before transaction is written, then we just skip the pending update.

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** 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.

Because recovery is a rare operation, a few redundant writes are nothing to worry about.

**Batching Log Updates**

Some file systems do not commit each update to disk one at a time. Rather, one can buffer all updates into a global transaction. For example, when the two files are created, the file system just marks the in-memory inode bitmap, inodes of the files, directory data, and directory inode as dirty, and adds them to the list of blocks that form the current transaction. When it is finally time to write these blocks to disk (say, after a timeout of 5 seconds), this single global transaction is committed containing all of the updates described above. Thus, by buffering updates, a file system can avoid excessive write traffic to disk in many cases.

**Making The Log Finite**

The log is finite. What would happen if we kept adding transactions to it before a batch update?

There are two problems:

1. The larger the log, the longer recovery will take, as the recovery process must replay all the transactions within the log (in order) to recover.
2. When the log is full (or nearly full), no further transactions can be committed to the disk, thus making the file system useless.

To address these problems, journaling file systems treat the log as a circular data structure (**circular log**). To do so, the file system must act sometime 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.

We can do this by simply mark the oldest and newest non-checkpointed transactions in the log in a **journal superblock** (all other blocks are free):

A picture containing table

Description automatically generated

In the superblock, the journaling system records enough information to know 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.

There is still a disadvantage: we are writing each data block to the disk twice, which is a heavy cost to pay, especially for something as rare as a system crash.

Text, letter

Description automatically generated

**Metadata Journaling**

Normal operation of the file system is slower than we might desire. For each write to disk, we are now also writing to the journal first, thus doubling write traffic. This doubling is especially painful during sequential write workloads, which now will proceed at half the peak write bandwidth of the drive.

Because of this high cost, people tried different approach. For example, the mode of journaling we described above is often called **data journaling** as it journals all user data. A simpler form is called **ordered journaling (metadata journaling)** where the user data is not written to the journal. For this approach, the following information would be written to the journal:

Chart

Description automatically generated

The data block Db, previously written to the log, would instead be written to the file system proper, avoiding the extra write. However, when should we write data blocks to disk?

The answer is that it should be written to the disk first, before metadata is written to disk:

A screenshot of a computer

Description automatically generated with medium confidence

By forcing the data write first, a file system can guarantee that a pointer will never point to garbage. Indeed, this rule of “write the pointed-to object before the object that points to it” is at the core of crash consistency and is exploited even further by other crash consistency schemes.

Note that it would be fine to concurrently issue writes to data, the transaction-begin block, and journaled metadata.

**Tricky Case: Block Reuse**

The issue arises when we delete a directory. This may makes the replay to overwrites user data to wrong location.

**Wrapping Up Journaling: A Timeline**

Table

Description automatically generated

Table

Description automatically generated

**42.4 Solution #3: Other Approaches**

Another approach is known as **copy-on-write** (**COW**). 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. Doing so makes keeping the file system consistent straightforward.

Another approach is **backpointer-based consistency** (or **BBC**). To achieve consistency, an additional back pointer is added to every block in the system. When accessing a file, the file system can determine if the file is consistent by checking if the forward pointer points to a block that refers back to it. If so, everything must have safely reached disk and thus the file is consistent; if not, the file is inconsistent, and an error is returned.

Finally, one approach is **optimistic crash consistency**. This new approach issues as many writes to disk as possible by using a generalized form of the **transaction checksum** and includes a few other techniques to detect inconsistencies should they arise.