42

Crash Consistency: FSCK and Journaling

As we've seen thus far, the file system manages a set of data structures to implement the expected abstractions: files, directories, and all of the other metadata needed to support the basic abstraction that we expect from a file system. Unlike most data structures (for example, those found in memory of a running program), file system data structures must **persist**, i.e., they must survive over the long haul, stored on devices that retain data despite power loss (such as hard disks or flash-based SSDs).

One major challenge faced by a file system is how to update persistent data structures despite the presence of a **power loss** or **system crash**. Specifically, what happens if, right in the middle of updating on-disk structures, someone trips over the power cord and the machine loses power? Or the operating system encounters a bug and crashes? Because of power losses and crashes, updating a persistent data structure can be quite tricky, and leads to a new and interesting problem in file system implementation, known as the **crash-consistency problem**.

This problem is quite simple to understand. Imagine you have to update two on-disk structures, A and B, in order to complete a particular operation. Because the disk only services a single request at a time, one of these requests will reach the disk first (either A or B). If the system crashes or loses power after one write completes, the on-disk structure will be left in an **inconsistent** state. And thus, we have a problem that all file systems need to solve:

> THE CRUX: HOW TO UPDATE THE DISK DESPITE CRASHES
> The system may crash or lose power between any two writes, and thus the on-disk state may only partially get updated. 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?

In this chapter, we'll describe this problem in more detail, and look at some methods file systems have used to overcome it. We'll begin by examining the approach taken by older file systems, known as **fsck** or the **file system checker**. We'll then turn our attention to another approach, known as **journaling** (also known as **write-ahead logging**), a technique which adds a little bit of overhead to each write but recovers more quickly from crashes or power losses. We will discuss the basic machinery of journaling, including a few different flavors of journaling that Linux ext3 [T98,PAA05] (a relatively modern journaling file system) implements.

42.1 A Detailed Example

To kick off our investigation of journaling, let's look at an example. We'll need to use a **workload** that updates on-disk structures in some way. Assume here that the workload is simple: the append of a single data block to an existing file. The append is accomplished by opening the file, calling `lseek()` to move the file offset to the end of the file, and then issuing a single 4KB write to the file before closing it.

Let's also assume we are using standard simple file system structures on the disk, similar to file systems we have seen before. This tiny example includes an **inode bitmap** (with just 8 bits, one per inode), a **data bitmap** (also 8 bits, one per data block), inodes (8 total, numbered 0 to 7, and spread across four blocks), and data blocks (8 total, numbered 0 to 7). Here is a diagram of this file system:
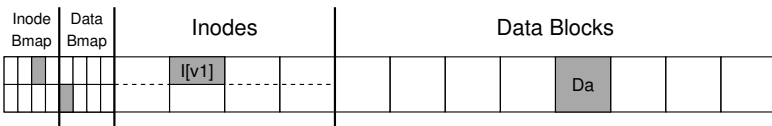
If you look at the structures in the picture, you can see that a single inode is allocated (inode number 2), which is marked in the inode bitmap, and a single allocated data block (data block 4), also marked in the data bitmap. The inode is denoted I[v1], as it is the first version of this inode; it will soon be updated (due to the workload described above).

Let's peek inside this simplified inode too. Inside of I[v1], we see:

```
owner       : remzi
permissions : read-write
size        : 1
pointer     : 4
pointer     : null
pointer     : null
pointer     : null
```

In this simplified inode, the `size` of the file is `1` (it has one block allocated), the first direct pointer points to block `4` (the first data block of the file, Da), and all three other direct pointers are set to `null` (indicating

that they are not used). Of course, real inodes have many more fields; see previous chapters for more information.

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        : remzi
permissions  : read-write
size         : 2
pointer      : 4
pointer      : 5
pointer      : null
pointer      : null
```

The updated data bitmap (B[v2]) now looks like this: 00001100. Finally, there is the data block (Db), which is just filled with whatever it is users put into files. Stolen music perhaps?

What we would like is for the final on-disk image of the file system to look like this:
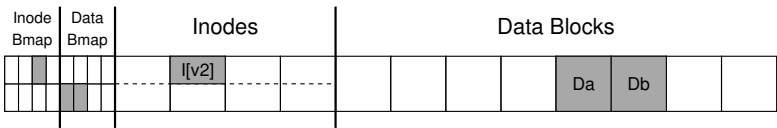
To achieve this transition, the file system must perform three separate writes to the disk, one each for the inode (I[v2]), bitmap (B[v2]), and data block (Db). Note that these writes usually don't happen immediately when the user issues a `write()` system call; rather, the dirty inode, bitmap, and new data will sit in main memory (in the **page cache** or **buffer cache**) for some time first; then, 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. In particular, 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 a funny state.

Crash Scenarios

To understand the problem better, let's look at some example crash scenarios. Imagine only a single write succeeds; there are thus three possible outcomes, which we list here:

- **Just the data block (Db) is written to disk.** In this case, the data is on disk, but 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. This case is not a problem at all, from the perspective of file-system crash consistency[1].

- **Just the updated inode (I[v2]) is written to disk.** In this case, the inode points to the disk address (5) where Db was about to be written, but Db has not yet been written there. Thus, if we trust that pointer, we will read **garbage** data from the disk (the old contents of disk address 5).

  Further, we have a new problem, which we call a **file-system inconsistency**. The on-disk bitmap is telling us that data block 5 has not been allocated, but the inode is saying that it has. This disagreement in the file system data structures is an inconsistency in the data structures of the file system; to use the file system, we must somehow resolve this problem (more on that below).

- **Just the updated bitmap (B[v2]) is written to disk.** In this case, the bitmap indicates that block 5 is allocated, but there is no inode that points to it. Thus the file system is inconsistent again; if left unresolved, this write would result in a **space leak**, as block 5 would never be used by the file system.

There are also three more crash scenarios in this attempt to write three blocks to disk. In these cases, two writes succeed and the last one fails:

- **The inode (I[v2]) and bitmap (B[v2]) are written to disk, but not data (Db).** In this case, the file system metadata is completely consistent: the inode has a pointer to block 5, the bitmap indicates that 5 is in use, and thus everything looks OK from the perspective of the file system's metadata. But there is one problem: 5 has garbage in it again.

- **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 correct data on disk, but again have an inconsistency between the inode and the old version of the bitmap (B1). Thus, we once again need to resolve the problem before using the file system.

- **The bitmap (B[v2]) and data block (Db) are written, but not the inode (I[v2]).** In this case, we again have an inconsistency between the inode and the data bitmap. However, 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.

---

[1]However, it might be a problem for the user, who just lost some data!

The Crash Consistency Problem

Hopefully, from these crash scenarios, you can see the many problems that can occur to our on-disk file system image because of crashes: we can have inconsistency in file system data structures; we can have space leaks; we can return garbage data to a user; and so forth. 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). Unfortunately, we can't do this easily because the disk only commits one write at a time, and crashes or power loss may occur between these updates. We call this general problem the **crash-consistency problem** (we could also call it the **consistent-update problem**).

42.2 Solution #1: The File System Checker

Early file systems took a simple approach to crash consistency. Basically, they decided to let inconsistencies happen and then fix them later (when rebooting). A classic example of this lazy approach is found in a tool that does this: **fsck**[2]. fsck is a UNIX tool for finding such inconsistencies and repairing them [M86]; similar tools to check and repair a disk partition exist on different systems. 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.

The tool fsck operates in a number of phases, as summarized in McKusick and Kowalski's paper [MK96]. It is run *before* the file system is mounted and made available (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.

Here is a basic summary of what fsck does:

- **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 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.
- **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. 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.

---

[2]Pronounced either "eff-ess-see-kay", "eff-ess-check", or, if you don't like the tool, "eff-suck". Yes, serious professional people use this term.

- **Inode state:** Each inode is checked for corruption or other problems. For example, `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.
- **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, 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.
- **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.
- **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, e.g., it has an address that refers to a block greater than the partition size. In this case, `fsck` can't do anything too intelligent; it just removes (clears) the pointer from the inode or indirect block.
- **Directory checks:** `fsck` does not understand the contents of user files; however, directories hold specifically formatted information created by the file system itself. Thus, `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.

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 [G+08]. However, fsck (and similar approaches) have a bigger and perhaps more fundamental problem: they are *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 (despite recent advances [M+13]).

At a higher level, the basic premise of `fsck` seems just a tad 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. This situation is akin to dropping your keys on the floor in your bedroom, and then com-

mencing a *search-the-entire-house-for-keys* recovery algorithm, starting in the basement and working your way through every room. It works but is wasteful. Thus, as disks (and RAIDs) grew, researchers and practitioners started to look for other solutions.

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

Probably the most popular solution to the consistent update problem is to steal an idea from the world of database management systems. That idea, known as **write-ahead logging**, was invented to address exactly this type of problem. In file systems, we usually call write-ahead logging **journaling** for historical reasons. The first file system to do this was Cedar [H87], though many modern file systems use the idea, including Linux ext3 and ext4, reiserfs, IBM's JFS, SGI's XFS, 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.

We'll now describe how **Linux ext3**, a popular journaling file system, incorporates journaling into the file system. Most of the on-disk structures are identical to **Linux ext2**, e.g., the disk is divided into block groups, and each block group has an inode and data bitmap as well as inodes and data blocks. The new key structure is the journal itself, which occupies some small amount of space within the partition or on another device. Thus, an ext2 file system (without journaling) looks like this:

| Super | Group 0 | Group 1 | . . . | Group N | |
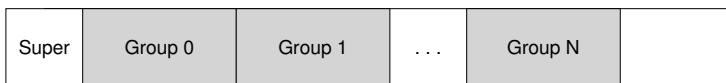
Assuming the journal is placed within the same file system image (though sometimes it is placed on a separate device, or as a file within the file system), an ext3 file system with a journal looks like this:

| Super | Journal | Group 0 | Group 1 | . . . | Group N | |

The real difference is just the presence of the journal, and of course, how it is used.

Data Journaling

Let's look at a simple example to understand how **data journaling** works. Data journaling is available as a mode with the Linux ext3 file system, from which much of this discussion is based.

Say we have our canonical update again, 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:

Journal	TxB	I[v2]	B[v2]	Db	TxE	⟶
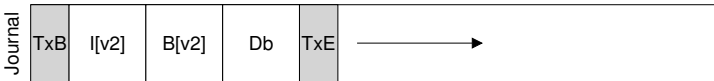
You can see we have written five blocks here. The transaction begin (TxB) tells us about this update, including information about the pending update to the file system (e.g., the final addresses of the blocks I[v2], B[v2], and Db), as well as some kind of **transaction identifier** (**TID**). The middle three blocks just contain the exact contents of the blocks themselves; this is 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.

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 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 transaction-end 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.

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.

Things get a little trickier when 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. 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'd like to issue

ASIDE: **FORCING WRITES TO DISK**

To enforce ordering between two disk writes, modern file systems have to take a few extra precautions. In olden times, forcing ordering between two writes, A and B, was easy: just issue the write of A to the disk, wait for the disk to interrupt the OS when the write is complete, and then issue the write of B.

Things got slightly more complex due to the increased use of write caches within disks. With write buffering enabled (sometimes called **immediate reporting**), a disk will inform the OS the write is complete when it simply has been placed in the disk's memory cache, and has not yet reached disk. If the OS then issues a subsequent write, it is not guaranteed to reach the disk after previous writes; thus ordering between writes is not preserved. One solution is to disable write buffering. However, more modern systems take extra precautions and issue explicit **write barriers**; such a barrier, when it completes, guarantees that all writes issued before the barrier will reach disk before any writes issued after the barrier.

All of this machinery requires a great deal of trust in the correct operation of the disk. Unfortunately, recent research shows that some disk manufacturers, in an effort to deliver "higher performing" disks, explicitly ignore write-barrier requests, thus making the disks seemingly run faster but at the risk of incorrect operation [C+13, R+11]. As Kahan said, the fast almost always beats out the slow, even if the fast is wrong.

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, for the following reason: 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:
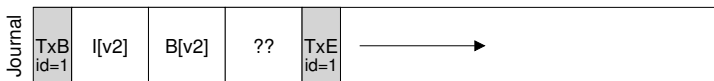
Why is this a problem? Well, the transaction looks like a valid transaction (it has a begin and an end with matching sequence numbers). Further, the file system can't look at that fourth block and know it is wrong; after all, it is arbitrary user data. Thus, if the system now reboots and runs recovery, it will replay this transaction, and ignorantly copy the contents of the garbage block '??' to the location where Db is supposed to live. 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.

ASIDE: **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 im-
pact is clear, if you think about how a disk works: usually an extra rota-
tion is incurred (think about why).

One of our former graduate students, Vijayan Prabhakaran, had a simple
idea to fix this problem [P+05]. When writing a transaction to the journal,
include a checksum of the contents of the journal in the begin and end
blocks. Doing so enables the file system to write the entire transaction at
once, without incurring a wait; if, during recovery, the file system sees
a mismatch in the computed checksum versus the stored checksum in
the transaction, it can conclude that a crash occurred during the write
of the transaction and thus discard the file-system update. Thus, with a
small tweak in the write protocol and recovery system, a file system can
achieve faster common-case performance; on top of that, the system is
slightly more reliable, as any reads from the journal are now protected by
a checksum.

This simple fix was attractive enough to gain the notice of Linux file sys-
tem developers, who then incorporated it into the next generation Linux
file system, called (you guessed it!) **Linux ext4**. It now ships on mil-
lions of machines worldwide, including the Android handheld platform.
Thus, every time you write to disk on many Linux-based systems, a little
code developed at Wisconsin makes your system a little faster and more
reliable.

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 (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:
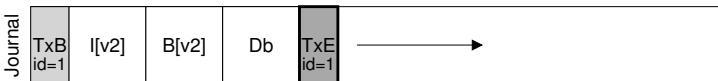
An important aspect of this process is the atomicity guarantee pro-
vided by the disk. It turns out that the disk guarantees that any 512-byte

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.

Recovery

Let's now understand how a file system can use the contents of the journal to **recover** from a crash. A crash may happen at any time during this sequence of updates. If the crash happens before the transaction is written safely to the log (i.e., before Step 2 above completes), then our job is easy: 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.

Note that it is fine for a crash to happen at any point during checkpointing, even after some of the updates to the final locations of the blocks have completed. In the worst case, some of these updates are simply performed again during recovery. Because recovery is a rare operation (only taking place after an unexpected system crash), a few redundant writes are nothing to worry about[3].

Batching Log Updates

You might have noticed that the basic protocol could add a lot of extra disk traffic. For example, imagine we create two files in a row, called `file1` and `file2`, in the same directory. To create one file, one has to update a number of on-disk structures, minimally including: the inode bitmap (to allocated a new inode), the newly-created inode of the file, the

---

[3]Unless you worry about everything, in which case we can't help you. Stop worrying so much, it is unhealthy! But now you're probably worried about over-worrying.

data block of the parent directory containing the new directory entry, as well as the parent directory inode (which now has a new modification time). With 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.

To remedy this problem, some file systems do not commit each update to disk one at a time (e.g., Linux ext3); rather, one can buffer all updates into a global transaction. In our example above, 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

We thus have arrived at a basic protocol for updating file-system on-disk structures. The file system buffers updates in memory for some time; when it is finally time to write to disk, the file system first carefully writes out the details of the transaction to the journal (a.k.a. write-ahead log); after the transaction is complete, the file system checkpoints those blocks to their final locations on disk.

However, the log is of a finite size. If we keep adding transactions to it (as in this figure), it will soon fill. What do you think happens then?
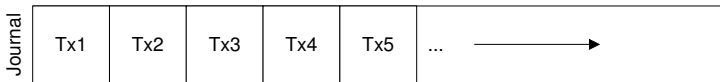
Two problems arise when the log becomes full. The first is simpler, but less critical: the larger the log, the longer recovery will take, as the recovery process must replay all the transactions within the log (in order) to recover. The second is more of an issue: when the log is full (or nearly full), no further transactions can be committed to the disk, thus making the file system "less than useful" (i.e., useless).

To address these problems, journaling file systems treat the log as a circular data structure, re-using it over and over; this is why the journal is sometimes referred to as 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:
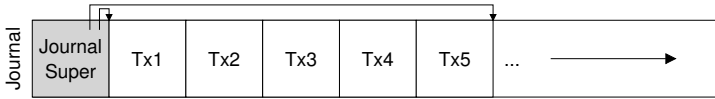
In the journal superblock (not to be confused with the main file system superblock), the journaling system records enough information to know which transactions have not yet been checkpointed, and thus reduces recovery time as well as enables re-use of the log in a circular fashion. And thus we add another step to our basic protocol:

1. **Journal write:** Write the contents of the transaction (containing TxB and the contents of the update) to the log; wait for these writes to complete.
2. **Journal commit:** Write the transaction commit block (containing TxE) to the log; wait for the write to complete; the transaction is now **committed**.
3. **Checkpoint:** Write the contents of the update to their final locations within the file system.
4. **Free:** Some time later, mark the transaction free in the journal by updating the journal superblock.

Thus we have our final data journaling protocol. But there is still a problem: 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. Can you figure out a way to retain consistency without writing data twice?

Metadata Journaling

Although recovery is now fast (scanning the journal and replaying a few transactions as opposed to scanning the entire disk), normal operation of the file system is slower than we might desire. In particular, 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. Further, between writes to the journal and writes to the main file system, there is a costly seek, which adds noticeable overhead for some workloads.

Because of the high cost of writing every data block to disk twice, people have tried a few different things in order to speed up performance. For example, the mode of journaling we described above is often called **data journaling** (as in Linux ext3), as it journals all user data (in addition to the metadata of the file system). A simpler (and more common) form of journaling is sometimes called **ordered journaling** (or just **metadata**

journaling), and it is nearly the same, except that user data is *not* written to the journal. Thus, when performing the same update as above, the following information would be written to the journal:
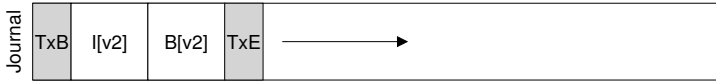
The data block Db, previously written to the log, would instead be written to the file system proper, avoiding the extra write; given that most I/O traffic to the disk is data, not writing data twice substantially reduces the I/O load of journaling. The modification does raise an interesting question, though: when should we write data blocks to disk?

Let's again consider our example append of a file to understand the problem better. The update consists of three blocks: I[v2], B[v2], and Db. The first two are both metadata and will be logged and then checkpointed; the latter will only be written once to the file system. When should we write Db to disk? Does it matter?

As it turns out, the ordering of the data write does matter for metadata-only journaling. For example, what if we write Db to disk *after* the transaction (containing I[v2] and B[v2]) completes? Unfortunately, this approach has a problem: the file system is consistent but I[v2] may end up pointing to garbage data. Specifically, consider the case where I[v2] and B[v2] are written but Db did not make it to disk. The file system will then try to recover. Because Db is *not* in the log, the file system will replay writes to I[v2] and B[v2], and produce a consistent file system (from the perspective of file-system metadata). However, I[v2] will be pointing to garbage data, i.e., at whatever was in the slot where Db was headed.

To ensure this situation does not arise, some file systems (e.g., Linux ext3) write data blocks (of regular files) to the disk *first*, before related metadata is written to disk. Specifically, the protocol is as follows:

1. **Data write:** Write data to final location; wait for completion (the wait is optional; see below for details).
2. **Journal metadata write:** Write the begin block and metadata to the log; wait for writes to complete.
3. **Journal commit:** Write the transaction commit block (containing TxE) to the log; wait for the write to complete; the transaction (including data) is now **committed**.
4. **Checkpoint metadata:** Write the contents of the metadata update to their final locations within the file system.
5. **Free:** Later, mark the transaction free in journal superblock.

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 with the pointer to it" is at the core of crash consistency, and is exploited even further by other crash consistency schemes [GP94] (see below for details).

In most systems, metadata journaling (akin to ordered journaling of ext3) is more popular than full data journaling. For example, Windows NTFS and SGI's XFS both use non-ordered metadata journaling. Linux ext3 gives you the option of choosing either data, ordered, or unordered modes (in unordered mode, data can be written at any time). All of these modes keep metadata consistent; they vary in their semantics for data.

Finally, note that forcing the data write to complete (Step 1) before issuing writes to the journal (Step 2) is not required for correctness, as indicated in the protocol above. Specifically, it would be fine to issue data writes as well as the transaction-begin block and metadata to the journal; the only real requirement is that Steps 1 and 2 complete before the issuing of the journal commit block (Step 3).
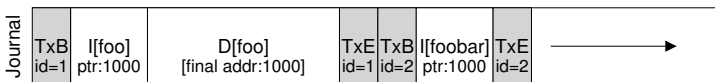
Tricky Case: Block Reuse

There are some interesting corner cases that make journaling more tricky, and thus are worth discussing. A number of them revolve around block reuse; as Stephen Tweedie (one of the main forces behind ext3) said:

> "What's the hideous part of the entire system? ... It's deleting files. Everything to do with delete is hairy. Everything to do with delete... you have nightmares around what happens if blocks get deleted and then reallocated." [T00]

The particular example Tweedie gives is as follows. Suppose you are using some form of metadata journaling (and thus data blocks for files are *not* journaled). Let's say you have a directory called foo. The user adds an entry to foo (say by creating a file), and thus the contents of foo (because directories are considered metadata) are written to the log; assume the location of the foo directory data is block 1000. The log thus contains something like this:
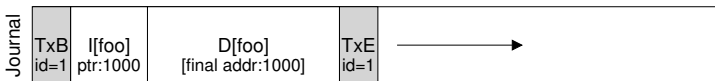
At this point, the user deletes everything in the directory as well as the directory itself, freeing up block 1000 for reuse. Finally, the user creates a new file (say foobar), which ends up reusing the same block (1000) that used to belong to foo. The inode of foobar is committed to disk, as is its data; note, however, because metadata journaling is in use, only the inode of foobar is committed to the journal; the newly-written data in block 1000 in the file foobar is *not* journaled.
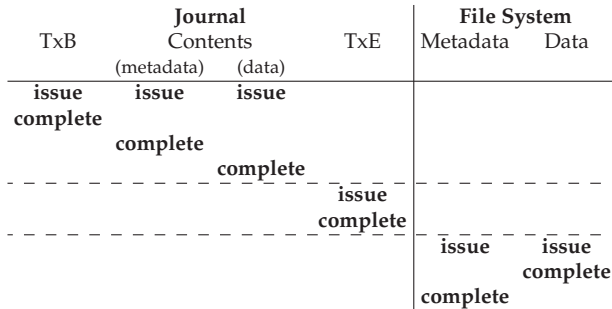
	Journal			File System	
TxB	Contents		TxE	Metadata	Data
	(metadata)	(data)			
issue	**issue**	**issue**			
complete					
	complete				
		complete			
			issue		
			complete		
				issue	**issue**
					complete
				complete	

Figure 42.1: **Data Journaling Timeline**

Now assume a crash occurs and all of this information is still in the log. During replay, the recovery process simply replays everything in the log, including the write of directory data in block 1000; the replay thus overwrites the user data of current file `foobar` with old directory contents! Clearly this is not a correct recovery action, and certainly it will be a surprise to the user when reading the file `foobar`.

There are a number of solutions to this problem. One could, for example, never reuse blocks until the delete of said blocks is checkpointed out of the journal. What Linux ext3 does instead is to add a new type of record to the journal, known as a **revoke** record. In the case above, deleting the directory would cause a revoke record to be written to the journal. When replaying the journal, the system first scans for such revoke records; any such revoked data is never replayed, thus avoiding the problem mentioned above.

Wrapping Up Journaling: A Timeline

Before ending our discussion of journaling, we summarize the protocols we have discussed with timelines depicting each of them. Figure 42.1 shows the protocol when journaling data as well as metadata, whereas Figure 42.2 shows the protocol when journaling only metadata.

In each figure, time increases in the downward direction, and each row in the figure shows the logical time that a write can be issued or might complete. For example, in the data journaling protocol (Figure 42.1), the writes of the transaction begin block (TxB) and the contents of the transaction can logically be issued at the same time, and thus can be completed in any order; however, the write to the transaction end block (TxE) must not be issued until said previous writes complete. Similarly, the checkpointing writes to data and metadata blocks cannot begin until the transaction end block has committed. Horizontal dashed lines show where write-ordering requirements must be obeyed.

A similar timeline is shown for the metadata journaling protocol. Note that the data write can logically be issued at the same time as the writes
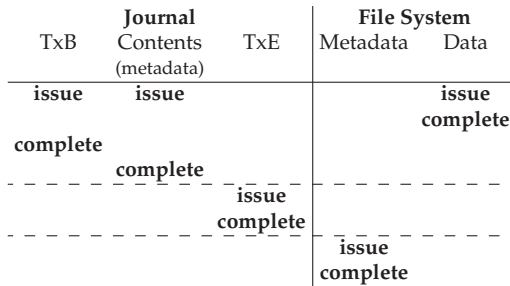
	Journal		File System	
TxB	Contents (metadata)	TxE	Metadata	Data
issue	**issue**			**issue**
				complete
complete				
	complete			
		issue		
		complete		
			issue	
			complete	

Figure 42.2: **Metadata Journaling Timeline**

to the transaction begin and the contents of the journal; however, it must be issued and complete before the transaction end has been issued.

Finally, note that the time of completion marked for each write in the timelines is arbitrary. In a real system, completion time is determined by the I/O subsystem, which may reorder writes to improve performance. The only guarantees about ordering that we have are those that must be enforced for protocol correctness (and are shown via the horizontal dashed lines in the figures).

42.4 Solution #3: Other Approaches

We've thus far described two options in keeping file system metadata consistent: a lazy approach based on fsck, and a more active approach known as journaling. However, these are not the only two approaches. One such approach, known as Soft Updates [GP94], was introduced by Ganger and Patt. 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** (yes, **COW**), and is used in a number of popular file systems, including Sun's ZFS [B07]. 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. We'll be learning more about this technique when we discuss the log-structured file system (LFS) in a future chapter; LFS is an early example of a COW.

Another approach is one we just developed here at Wisconsin. In this technique, entitled **backpointer-based consistency** (or **BBC**), no ordering is enforced between writes. To achieve consistency, an additional **back pointer** is added to every block in the system; for example, each data block has a reference to the inode to which it belongs. When accessing a file, the file system can determine if the file is consistent by checking if the forward pointer (e.g., the address in the inode or direct block) 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. By adding back pointers to the file system, a new form of lazy crash consistency can be attained [C+12].

Finally, we also have explored techniques to reduce the number of times a journal protocol has to wait for disk writes to complete. Entitled **optimistic crash consistency** [C+13], this new approach issues as many writes to disk as possible and uses a generalized form of the **transaction checksum** [P+05], as well as a few other techniques, to detect inconsistencies should they arise. For some workloads, these optimistic techniques can improve performance by an order of magnitude. However, to truly function well, a slightly different disk interface is required [C+13].

42.5 Summary

We have introduced the problem of crash consistency, and discussed various approaches to attacking this problem. The older approach of building a file system checker works but is likely too slow to recover on modern systems. Thus, many file systems now use journaling. Journaling reduces recovery time from O(size-of-the-disk-volume) to O(size-of-the-log), thus speeding recovery substantially after a crash and restart. For this reason, many modern file systems use journaling. We have also seen that journaling can come in many different forms; the most commonly used is ordered metadata journaling, which reduces the amount of traffic to the journal while still preserving reasonable consistency guarantees for both file system metadata as well as user data.

References

[B07] "ZFS: The Last Word in File Systems"
Jeff Bonwick and Bill Moore
Available: http://opensolaris.org/os/community/zfs/docs/zfs_last.pdf
ZFS uses copy-on-write and journaling, actually, as in some cases, logging writes to disk will perform better.

[C+12] "Consistency Without Ordering"
Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
FAST '12, San Jose, California
A recent paper of ours about a new form of crash consistency based on back pointers. Read it for the exciting details!

[C+13] "Optimistic Crash Consistency"
Vijay Chidambaram, Thanu S. Pillai, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
SOSP '13, Nemacolin Woodlands Resort, PA, November 2013
Our work on a more optimistic and higher performance journaling protocol. For workloads that call `fsync()` *a lot, performance can be greatly improved.*

[GP94] "Metadata Update Performance in File Systems"
Gregory R. Ganger and Yale N. Patt
OSDI '94
A clever paper about using careful ordering of writes as the main way to achieve consistency. Implemented later in BSD-based systems.

[G+08] "SQCK: A Declarative File System Checker"
Haryadi S. Gunawi, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
OSDI '08, San Diego, California
Our own paper on a new and better way to build a file system checker using SQL queries. We also show some problems with the existing checker, finding numerous bugs and odd behaviors, a direct result of the complexity of `fsck`*.*

[H87] "Reimplementing the Cedar File System Using Logging and Group Commit"
Robert Hagmann
SOSP '87, Austin, Texas, November 1987
The first work (that we know of) that applied write-ahead logging (a.k.a. journaling) to a file system.

[M+13] "ffsck: The Fast File System Checker"
Ao Ma, Chris Dragga, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
FAST '13, San Jose, California, February 2013
A recent paper of ours detailing how to make fsck an order of magnitude faster. Some of the ideas have already been incorporated into the BSD file system checker [MK96] and are deployed today.

[MK96] "Fsck - The UNIX File System Check Program"
Marshall Kirk McKusick and T. J. Kowalski
Revised in 1996
Describes the first comprehensive file-system checking tool, the eponymous `fsck`*. Written by some of the same people who brought you FFS.*

[MJLF84] "A Fast File System for UNIX"
Marshall K. McKusick, William N. Joy, Sam J. Leffler, Robert S. Fabry
ACM Transactions on Computing Systems.
August 1984, Volume 2:3
You already know enough about FFS, right? But yeah, it is OK to reference papers like this more than once in a book.

[P+05] "IRON File Systems"
Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
SOSP '05, Brighton, England, October 2005
A paper mostly focused on studying how file systems react to disk failures. Towards the end, we introduce a transaction checksum to speed up logging, which was eventually adopted into Linux ext4.

[PAA05] "Analysis and Evolution of Journaling File Systems"
Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
USENIX '05, Anaheim, California, April 2005
An early paper we wrote analyzing how journaling file systems work.

[R+11] "Coerced Cache Eviction and Discreet-Mode Journaling"
Abhishek Rajimwale, Vijay Chidambaram, Deepak Ramamurthi,
Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
DSN '11, Hong Kong, China, June 2011
Our own paper on the problem of disks that buffer writes in a memory cache instead of forcing them to disk, even when explicitly told not to do that! Our solution to overcome this problem: if you want A to be written to disk before B, first write A, then send a lot of "dummy" writes to disk, hopefully causing A to be forced to disk to make room for them in the cache. A neat if impractical solution.

[T98] "Journaling the Linux ext2fs File System"
Stephen C. Tweedie
The Fourth Annual Linux Expo, May 1998
Tweedie did much of the heavy lifting in adding journaling to the Linux ext2 file system; the result, not surprisingly, is called ext3. Some nice design decisions include the strong focus on backwards compatibility, e.g., you can just add a journaling file to an existing ext2 file system and then mount it as an ext3 file system.

[T00] "EXT3, Journaling Filesystem"
Stephen Tweedie
Talk at the Ottawa Linux Symposium, July 2000
olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html
A transcript of a talk given by Tweedie on ext3.

[T01] "The Linux ext2 File System"
Theodore Ts'o, June, 2001.
Available: http://e2fsprogs.sourceforge.net/ext2.html
A simple Linux file system based on the ideas found in FFS. For a while it was quite heavily used; now it is really just in the kernel as an example of a simple file system.