Journaling

1 Introduction

Hello, and welcome to this lesson on journaled file systems.

Journaled file systems are a somewhat newer development, with IBM's JFS for "Journaled File System" being the first commercial product at around 1990. The Windows NTFS file system implemented journaling around 2000, and and Linux's EXT3 a bit later.

We will start by observing what can go wrong if the system crashes during a write operation to disk. Because write file system write operations require multiple writes to disk, to update the data block, the inode block, and maybe the a free list block, it can happen that only some of these writes can make it to disk before then the system crashes. This can cause difficulties after the system reboots.

The are two approaches to deal with disk crashes. One is to not worry about crashes until after they happen. If a crash occurs, and the system reboots, a special program checks the file system for correctness and recovers any missing data.

An alternative is to pro-actively organize the write operations to prepare them for a possible recovery. This approach uses a write transaction log, which is called "journal". We will look at how journals crash resistant consistent file systems.

We will then look at an optimization of journaling, which tries to reduce the number of additional writes that need to be written to the log.

We will wrap up with a pesky problem that one has to deal with in journaling, which occurs when files or directory can be deleted and their blocks re-used. Similarly to re-use of memory in lock-free data structures re-use of blocks in journaling can lead to inconsistencies. We will briefly discuss how this can be avoided.

After this lesson you will understand how journaling works in modern file systems such as NTFS in Windows and EXT3 in Linux and Mac OS. The discussion of journaling is also a preparation for the upcoming discussion of so-called "log-based" file systems in a next lesson.

Let's begin.

2 What can Happen when a System Crashes?

Let"s first get an understanding what can go wrong when the system crashes during a file system write operation.

For this, let's assume that we have a simple file system, which consists of:

- data blocks, here 8 of them
- a list of 8 inodes stored in a four blocks,
- a data block bitmap to keep track of free blocks,

- and an inode bitmap to keep track of free inodes in the inode list.

A marked entry in the inode bitmap, here entry 3, indicates the the corresponding inode is used.

Similarly, a marked entry in the data block bitmap indicates that the corresponding block is used.

Let's look at the steps that the file system needs to do to append a block to the file.

1. First, we need to write the data block to disk.
2. Then we update the data block bitmap to indicate that the new block is used.
3. Finally we need to update the inode block with the updated inode for the file.

These are three separate write operations. Let's look at what can go wrong when a crash happens while these operations are being executed. Fortunately, we can safely assume that block writes to disk are atomic: a write either happens or it does not happen. We can assume that if a write gets interrupted by a system crash, this is detected during restart, and this partial write is rolled back.

We start with the first of the three steps to append the block to the file, and we write the new data block Db.

Unfortunately, now the system crashes, before we can complete the other two steps.

What is the effect of the crash?

Well, the data block has been written to disk.

This change has not been recorded in the inode, so the file does not appear as having changed. It is also not recorded in the data bitmap. As a result, the blocks still appears free to the file system.

After the system boots the file system is again completely consistent. The data stored in one of the unused blocks has changed, but this does not affect the file system.

For the user, of course, things look a bit different, as he or she lost an update. Depending on the application, this may not be a problem, as the user can simply re-issue the operation that led to the addition of the block.

Let's look at another scenario.

Here the write the disk block, and update the data block bitmap to indicate that the block is used.

Before we can update the inode, the system crashes.

The effect of this crash on the append operation are:

The data block is written to disk. (5)

The change has been recorded in the data block bitmap. The block is therefore marked as used.

The change has not been recorded in the inode yet, meaning that the file appears as not having changed. The user looses the update.

In addition, the system just lost a block. This block is marked as used but does not belong to any file. As a result, it cannot be freed as part of a file deletion operation. We say that the block has been "leaked" from the file system.

Let's look at a third scenario.

Sometimes, disk scheduling may re-order the write operations, and we manage to update the data block bitmap and the inode, but the data block write is delayed, and the system crashes before we can write the data block to disk.

The effects now are more severe.

The change is recorded in the data block bitmap and in the inode. As a result, the file appears to have been modified.

The new block has not been written, however.

The allocation data in the updated inode points to a block that has not been updated, and therefore contains garbage data.

The file system is clearly inconsistent.

Let's look at an overview of the possible crash scenarios.

In this table the top-three rows indicate whether

the write to the data block, the bitmap block, and the inode were successful.

The last row indicates the severity of the effect of the crash on the file system.

We have 7 different cases.

The first case, where all writes were successful clearly has no negative effect on the file system. We mark this with green. The next two cases we discussed earlier.

If the system crashes after writing the block to disk, there is no serious effect other than the user having to re-issue the operation.

If the data block and the data block bitmap can be written, then we have a block leak.

If the data block cannot be written, then the file system becomes inconsistent.

If only the data bitmap can be written, then we have a block leak.

In the remaining two cases, where the inode can be updated, the file system again becomes inconsistent.

Given that crashes are a fact of live, how can we recover from them?

3 Post-Mortem Recovery: File System Checkers

One way is to recover post mortem, after the fact.

After the system reboots, a file-system checker checks the file system and attempts to detect and recover from errors.

It does this in a sequence of steps. First it checks whether the superblock is ok. If it is corrupted, it is critical that a copy can be recovered from somewhere else, as the superblock contains data that is absolutely critical for the system to even understand what is on the disk.

If the superblock is ok, the checker scans the allocation information in the inodes and in the index blocks. All the blocks that are not allocated to inodes are free, and the bitmap of free blocks can be recovered.

As we scan through the inodes we also get a list and location of valid inodes, which allows us to update the inode bitmap.

Now the inodes are checked in detail. If an inode has errors that cannot be recovered, the inode, and its corresponding blocks are deleted.

We now check the link counters in the inodes. Starting from the root directory, we traverse all the inodes and check whether the link count in the inodes is consistent with the directory graph.

The allocation tables are checked so see whether multiple entries point to the same block. In some cases it makes sense to copy the block to have both inodes get a copy.

We check for out-of-range block pointers, and finally check that the content of the directory files.

Let's look at the pros and cons of post-mortem style recover.

The advantage, as opposed to the pro-active approaches that we explore next, is that there is no overhead during normal operation. A file checker is only invoked when a crash occurred, or some file system inconsistency is suspected.

It has a number of disadvantages, however.

One is that it requires very detailed information about the operation of the file system. A minor change in the file system implementation may require a re-implementation of parts of the file system checker. This is a minor issue, compared to the following ones.

An important problem is that file system checkers often cannot recover from all types of errors. Sometimes, files cannot be reconciled with their directories, for example, which requires expensive post-processing by the user.

Also, once a file system checker runs, it may take a very long time for it to complete all the steps, as inodes, allocation tables, and directories may have to be scanned multiple times to attempt to identify and try to recover from errors. This is particularly irritating of one considers that crashes may affect at most a few uncommitted write operations.

4 Pro-Active "Recovery": Journaling

An alternative to post-mortem recovery using a file system checker is to pro-actively handle possible errors. The idea is to log the intended operations as transactions in a separate portion of the disk, which we call the "journal".

If we want to append a block to a file, for example, we first log a "begin transaction" record, followed by the three write operations that we need to perform as part of the append, and finally by an "end-of-transaction" record.

Once these logs are committed to disk, we can push to disk the updates to the data block, the data bitmap, and the inode.

Once the updates are committed. the transaction can be deleted from the log.

Let's assume that we have the transaction committed to the log, and a crash occurs when we want to push the writes to disk.

We push the update to the data block, and the update to the data bitmap.

While we update the inode, the system crashes and has to reboot.

This is no problem. When the system reboots, the transaction is still in the log, and the file system retries again..

If it succeeds this time around, it deletes the transaction from the log.

What can happen when the system crashes while we write to the journal?

Let's see. The transaction is logged in the journal, but not all the operations have been committed to disk yet.

If the system crashes now, during recovery it will find garbage in the journal.

This can be easily avoided by what is called "journal commit".

When we log a transaction we write the "begin transaction" record, and the records for the operations. Before writing the "end-of-transaction" record, we wait until all previous records in the log have been committed to disk.

Once they have, we can write the "end-of-transaction" record to the journal.

During recovery after a crash the system pushes only complete transactions to disk.

5 Meta-Data Journaling

One thing that we have not talked about is that.

Write operations with journaling are slow.

This is because for each write operation, we have to write to the journal as well, which basically doubles the number of write operations.

The solution is to reduce the number of writes to the journal by journaling the meta data only. Data blocks are not written to the journal. The idea is to ensure file system consistency, at first, and then have the user applications deal with data inconsistencies caused by crashes. Because we only journal the meta-data, this is called ?meta-data journaling?.

In our example, with meta-data journaling we only log the update to the data bitmap and to the inode.

Once the transaction is committed to the journal,

We may write the data block directly to disk, push the transaction to disk, and then delete the transaction from the log. Often, transactions write multiple data blocks to disk, so meta-data journaling saves quite a lot of writes.

We have to be careful, however, how we order the writes to data blocks to avoid inconsistencies after crashes.

For example, if we commit to the journal, and then write the data block to disk, but cannot commit it yet, we may have a problem when a crash occurs.

When the system boots up after the crash, it finds the transaction that has been committed to the log.

We see that the data block has not been written, however.

The transaction gets pushed to the disk. and the transaction is deleted from the journal.

Because the meta-data has been updated but the data block has not, the content of the data block is garbage.

How can we avoid this situation? By careful ordering of the write operations.

We first write the data block, and make sure that it is committed.

After that we can write the transaction to the journal, and commit it. At this point we can be sure that the file system will remain consistent despite crashes.

This business with journaling only the meta-data has a few more problems however.

6 Dealing with Block Reuse

For example, when files or directories are deleted and their blocks re-used..

In this example, we add a file to directory X. This changes the inode of the directory file and the content of the directory file as well.

We log that we change the inode from IX1 to IX2, and the data block of the directory file from DX1 to DX2. We log this data block because directories are part of the meta data.

Sometimes later we delete directory X.

We log that we delete the inode of directory X [5] as well as the content of the directory file blocks. Shortly thereafter we create a file Y and write some data to it. Following the ordering that we defined when we talked about the importance of ordering of writes in meta-data journaling, we write the data block for file Y to disk and wait until it is committed. Then we log the write of the meta-data to in the journal. In particular, we log the write of the update of the new file. Note that until now we have not pushed the transactions to disk yet, because we want to batch multiple transactions into a single push, for performance reasons.

What happens if the system crashes now?

The system reboots and finds the three transactions in the journal.

It pushes the first transaction to disk, which updates the inode for directory X, and the directory file data block of directory X. We notice here that we are overwriting the data block of file Y.

This takes care of the first transaction,

and we start pushing the second, which deletes the inode for directory X.

We delete the second transaction, and start the last one. This one writes the inode for the newly created file Y.

At the end, we have an inconsistent state because the inode of file Y is pointing to the data block which was originally correct but has been overwritten as part of the recovery with a data block for the directory file for directory X.

This inconsistency due to file and directory deletion and re-use of blocks is a real problem in existing systems, and there are several solutions to address this.

One approach could be to not re-use blocks as long as there are "delete" operations in the journal.

The approach in the Linux EXT3 system is uses special "revoke" records, which prevents recovery from redoing operations for files or directories that have been deleted, such as Directory X in the previous example.

7 Journaled File Systems: Summary

With this we have come to the conclusion of our exploration of Journaled File Systems.

We have investigated what can happen when systems crash while the file system is in the middle to writing to disk. Because some of these write operations are actually sequences of writes, crashes can lead to all kind of errors, ranging from the user missing writes, to the file system leaking blocks, to the file system becoming inconsistent.

There are two ways to deal with this. The first is to now worry about it and recover when the system restarts after a crash. The problem with this approach is that it is expensive and that it may not find all the errors.

The alternative is to ensure atomicity of the write instructions by adding a journal, where file system operations are logged as transactions, which are simply replayed when the system has to recover.

The main problem with naïve journaling is that it doubles the number of writes because operations need to be written to the journal and then pushed to the disk again. Meta-data journaling improves performance significantly, because it only journals meta data.

Originally, one thought that journaling is easy. One simply adds a transaction log to the basic file systems and that's it. Meta data journaling and its problems with file deletion and block reuse illustrate that journaled file systems are very tricky, and that their design has to be approached very carefully.

We truly hope that you enjoyed this exploration in to the benefits and complications of Journaled File Systems. Journaled file systems play a very important role in today's systems, and they provide a very good initial step towards understanding log-based file systems, which we will study in one of the next lectures.