CS 111

Scribe Notes for 11/19/14

by Christian Thorenfeldt

File System Robustness

Motivation & Invariants

When a file system is built, there are many possible mistakes that can occur. Simply put, you may have bugs in your system code, and we need a way to find those bugs really fast. To alleviate this issue, we introduce the **invariants**, which are boolean expressions that are always true. We have to check our code to keep these invariants true, and as long as they stay true, we know our code passes our basic rules that we've set up.

Berkeley Fast File System Invariants:

1. Every block in the file system is used for exactly one purpose. For example, we would look at every block in the system, check that it's a boot block, a superblock, a bitmap block, inode block, or a data block. We want to make sure that the system isn't using a single block for two different purposes.
2. All referenced blocks are initialized with data that is appropriate for their type. For example, a bit map block should be initialized to all 1's (not random values), referenced inode blocks should have a size field set up, and also set up to indicate whether or not the inode is in use.
3. All referenced data blocks are marked used in the bitmap.
4. All unreferenced blocks are marked free.

Increasing Performance

What happens when we want to increase performance of our system? In order to get increase our performance, the system will have to play around with the cache. We can think of the problem as a **cache coherence problem**. We will have to cache a lot of the file system in RAM, maybe 1% of the system, and we hope that the system will talk to the cache.

For example, let's look at the following: /usr/bin/sh

The cursor will first look at the root directory, look at the inodes, and maybe accumulate up to 10 disk accesses for the entire file. That's quite a lot of accesses and will take a significant amount of time if each access takes 20 milliseconds.
Solution: We can cache things that we need often such as inode information and commonly used data. Because /usr/bin is so commonly used, most UNIX systems keep this easily accessible in RAM.

However, we now run into the aforementioned problem: cache coherence. If we look at the contents of the cache, it says that, for example, we have a certain copy of a block on disk, but in reality, it's not.

Moreover, to increase performance even more, we use the technique called **Dallying** . The idea is that we have some work that we need to do, but we postpone it a bit to do some work on the way. For example, let's look at the system call: `write (fd, buffer, buf_size)` . The system will place a copy of the buffer into the system cache and immediately return without actually writing the data. If there is a subsequent read in another process, the system will look into the cache to check if the buffer is there and read it. However, this is incredibly vulnerable to crashes since the data isn't saved to disk, which leads us to our next section.

System Crashes

There are many reasons why our system might crash. Maybe our flash drive that we are using is abruptly unplugged (a storage disconnect), or perhaps our power runs out and the system shuts down.

At the end of the day, we want to make sure that the disk represents what people write. We have several options to address this problem:

1. `sync()` , which schedules all of the cache to get written to the disk (in effect, stop dallying).
2. `fsync (fd)` , which writes all of fd's cache data to disk and waits for it to finish (but can be quite slow and use extra blocks)
3. `fdatasync (fd)` , which is only guaranteed to save data blocks (not metadata and costs only 1 block).

We are telling the users about cache coherence and complicating user programs by giving the user the responsibility to issue these calls. It would be best if this was invisible to the user.

Implementing Rename System Call

```
rename ("d/a", "e/b")
```

The rename is changing the two directories "d" and "e" (file doesn't change). First directory will have directory entry removed and the second will have a directory entry added to it.

1. We can try to update the directory name, and then update the file name. If we encounter a power failure here, we could lose the file and consequently lose all of our data.
2. If we try to update the file name, and then the directory name, we may end up having two hard links to the same file.

Example Application… Emacs

Suppose emacs operates on a flash drive. Let's say we have a file that takes 10 blocks in the flash drive. We edit the file, and save the file. However, we want to save the file in a reliable way. When we hit save, we want to know that losing power in the middle of a save means we don't lose any of our data. It is okay though to lose the new data, as long as we still have the old version of the file.

A solution to this problem would be to keep a set of blocks for the old file, and a set of blocks for the new file. That way if we lose power when editing the new file, we still can access the old file's data. However, it might be difficult to differentiate between the two. This is where the **commit record** comes in. We could keep a sort of "directory" block and it tells us which set of block is our old file and which set of blocks is our new file that we are editing. Any changes only occur when we commit a change in the "directory" block.

One more problem though until we can be at peace. What if we lose power during a commit? There could be garbage values. We have to worry about the atomicity of the commit record. To solve this, we could have three copies of the commit record. If two records disagree with one another, choose the majority. This is important if one of the three records has undefined data due to power loss.

Problems Concerning Flash Drives

Writing to a flash drive over time can wear out a block. A solution to this is FTL which is a flash file system that in a way distributes block usage evenly so as to not wear out commonly used ones, which is completely unbeknownst to the user. For example, if the user wants to use block 3, FTL may assign it to block 7 or some other block.

Moreover, we also have a couple more real world problems concerning flash drives to consider…

1. Bit corruption - maybe we lose power and our bit that was A turned into B
2. Shorn writes - write out a block, lose power, and read the block back in.. and we only have part of the write
3. Unserializable writes
4. Meta data corruption - if we lose metadata, how will the FTL map block 3 to block 7. Everything can be scrambled and this is very bad.
5. Total drive failure - cut the power, turn it on, and nothing works.

We could write a file system that survives these kind of issues, but it's close to impossible.

We want a file system that is reliable, so we have to make some assumptions about our underlying storage…

Lampson-Sturgis Assumptions (developed for hard disk)

1. storage writes can fail and it can corrupt another piece of storage
2. storage may spontaneously decay - assume if a block decays, can get a read error.
3. assume errors are rare enough so that repair is feasible.
4. similar assumptions about process failures

With these assumptions, we know there is no such thing as undefined data in a block. If we read a bad block, we will know that it's undefined as per the assumptions. This will help us narrow down file system design, and perhaps allow us to have two copies of commit records rather than three.