File System Robustness

By Ryosuke Takeuchi

File system robustness is the most difficult part of file system implementation; it is where most of the problems tend to occur.

What can go wrong?

-There may be bugs in the system code
-What is an inexpensive way to find these bugs?
-We can implement invariants in our file system code, and make sure they are true.

Invariants are boolean expressions over variables which must always stay true.

FFS Invariants

1. Every block in the file system is used for exactly one purpose.

   - For example: bool block, super block, bitmap block, inode block, data block
   - Invariant: A block should not be used for more than one purpose
   - This is an obvious one, but we want to check in case someone violates this rule.

2. All referenced blocks are initialized with data that is apprpriate for their block type.

   - For the bitmap block, all blocks should be initialized at 1 since they must all be free.
   - For the inode block, only the blocks being used by the super block should be correctly initialized.
   - For data blocks, all blocks can have any data except the root directory, which should be empty.
   - We make sure these invariants stay true as the file system grows and shrinks.

3. All referenced data blocks are marked as used in the bitmap.
4. All unreferenced blocks are marked as free.

What happens if any of these are violated?

1. Bitmap may overwrite important data -> data loss.
2. Data is incorrectly initialized.
3. Data may be incorrectly allocated, and may cause security problems.
4. Memory leak (not a critical problem).

Cache and Performance, Coherence Issues

Because reading from RAM is faster than reading from disk, we want to keep commonly used items cached in RAM.
However, this means we must deal with the problem of **cache coherence**: What do we do if the cache

is not the same as the original data?
This may occur due to a performance technique known as **dallying**.
Dallying is a process in which some tasks are prioritized before working on the task which was originally instructed.

Caches and System Crashes

When we call write() with a buffer, we may write the buffer to cache instead of disk and return immediately.
If we need to read this data later, we may look at cache instead of disk, leading to speed up.
However, if the system crashes we do not have the data saved to disk!

What can cause crashes?

Assuming we do not have system bugs, crashes may be cause by power shortages or disk disconnections.
We may prevent this by having an uninterruptable power supply, or by saving the contents of the cache before the power goes out.
Alternatively, we may change our invariants to address the power outage problem.

sync()

- Schedule so that all of the cache is saved to disk, preventing the system from dallying until the entire cache is written.
- This means that everything written before sync is called is saved.
- However, this writes out the entire cache when we may only need part of it.
- This function returns immediately.

fsync(fd)

- Write all of cached data in fd to disk, and wait until finished.
- This takes more real time than sync because it waits.
- This function is slow because even for a write of 1 byte, we must work on at least 1 data block and 1 inode block.

fdatasync(fd)

- Only guaranteed to save the data blocks; not guaranteed for the metadata in inode blocks.
- This is twice as fast because it only uses one block.

Using Cache for rename

1. rename("d/a", "e/b")
2. read d inode
3. read d data
4. read e inode
5. write inode for files (+1 link count)
6. read e data
7. write e data
8. write d data

9. write d inode
10. write e inode

1-4 can be done in any order as long as they are done before the others.
9 and 10 can be done in any order.

Commit Records

A drive may contain 2 copies of a file - the commit record tells us which copy of the file we should use.

```
Keep 3 copies of the commit record
1. A ? B B B B B
2. A A A ? B B B
3. A A A A A ? B
Use the majority
---A A A B B B B
```

If all 3 records disagree, use the first record.
This method is usually used for small data such as metadata. (2 copies of data and 3 copies of record)

Flash Memory

Blocks are used in flash - However, eventually the blocks wear out from usage.
The FTL is used as an image of the drive which does not match what is stored on the drive.
The FTL is part of the controller of the flash drive, spreading out the usage of blocks.
If power is lost, the contents of the FTL is lost - this is similar to a file system.

Problems of Flash Drives:

- Bit corruption
- Shorn writes: Loss of trailing data when writing a block
- Unserializable writes: If power is lost while writing multiple blocks, blocks written in a different order due to the FTL may not be saved
- Metadata corruption
- Total drive failure

To build a reliable file system, we must make assumptions about underlying drives (Lampson/Sturgis assumptions)

- Storage writes can fail, or corrupt another piece of storage. A later read can detect these failures.
- Storage my spontaneously decay.
- Assume errors are rare enough so that repair is feasible.
- Make similar assumptions about process failures.

If we write a block and lose power, we do not know what is written.
With these assumptions we can detect the bad block.
We can also narrow down the file system design - e.g. only need 2 records.

Using these assumptions to design a robust file system

1. Commit Record

- If we want to do a large action atomically, commit to the record.
- Tells the system if the large object was written or not.

2. Journal
Two Areas: Cell data and Journal

- Cell Data: Data blocks
- Journal: Record of actions and changes made to the data.
- Commit after the changes are made.
- If the system crashes, redo the actions in the journal which are not written to data.
- Can work with a complicated set of instructions such as rename.
- Written sequentially