CS 111

Lecture 13 Scribe Notes (Fall 2013)

by David Kang

File System Robustness

Goals

We can decompose FS robustness into subgoals:

Durability - Survives (limited) hardware failures. E.g. can survive power loss, but cannot survive computer melting.

Atomicity - Changes are done or not done.

Performance - Throughput and latency.

What can go wrong with hardware?

- lose power
- run out of disk space
- heavily loaded system
- sudden disconnect
- bad sectors/tracks
- lose disk (disk crashes)

We focus on the "lose power" aspect since it is a simple problem. We need to deal with losing contents of RAM, CPU/controller caches, and registers. We only keep what was written to the disk.

Suppose you're writing to a disk, you only get a few microseconds to:
have the CPU execute a few instructions and
make the disk controller park its head.
We'd like to assume that sectors writes are atomic for the sake of this example.

Exceptions to atomicity

However, it is important to remember that sector writes are not always atomic. For example, according to Zheng et al. FAST '13 "Understanding the robustness of SSD's under power fault," out of 15 SSD's, only 2 worked after power loss during operation. The other 13 suffered from:

- bit corruption - random bit errors
- flying writes - bits get written to the wrong location when the SSD is powered back on due to the controller's wear leveling becoming confused upon power up
- shorn writes - write operations are only partially done

- metadata corruption - metadata in FTL is corrupted
- dead device - devices is dead or mostly non-functional
- unserializability - some requests were written, but not in an understandable order

Implementing save in emacs

Suppose we're trying to implement ^X^S (save) in emacs.

```
write(fd, buf, sector size)
```

We don't have an operating system for now. If we lose power here, we can lose data. To try circumvent this issue, suppose we have 2 devices.

Golden Rule of Atomicity: Never overwrite your only copy.

We have two blocks, A and B. We write A first with B's data, then overwrite B's data.

old to new picture

One problem with this is that we bloat data by 2x. The second, more significant problem is that if we lose power right before we overwrite B's data, we do not know whether A or B's data is correct:

Don't know which data is right.

The solution to this is to do three writes. Then if we have a power loss, we take the 2/3 majority that are the same. If A, B, C are all different, we take A as the correct bit.

Doing three writes solves bit corruption. Writes are not atomic, but they eventually finish since we know which bit is correct.

Don't know which data is right.

Lampson-Sturg's assumptions

- storage writes may fail or corrupt another piece of storage
- a late read can detect the bad sector
- storage may spontaneously decay
- errors are rare
- repairs can be done in time (before another failure occurs

Eggert Assumption (overly rosy)

- block writes are atomic

We use Eggert's Assumption to simply things for us.

An example

```
if (lseek(fd, 0, SEEKSET) % 8192 == 0)
        write(fd, buf, 8192);  // assume already allocated
        // assume BSD style file system
```

Picture of drive layout.

We update the time stamp first, then the data. This will cause 'make' to do extra work, but it will still succeed.

A common optimization is to do the write, dally, and then timestamp the file. This, however, has the issue of breaking 'make' since it depends on updated timestamps. The solution to this is to use a cleanup procedure following a reboot, i.e. 'make clean'.

Another example

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

a being crossed out to b

For the first operation, we find d's inode, then we find d's data and change the "a" to "b".

The second operation can be more difficult because it requires dealing with block allocation, as the file names are of different sizes.

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

showing directory change.

A problem that arises here is that when we write first, the link count is 1, although there are now two links pointing to the inode.

To fix this:

1. increment link count
2. write e's data
3. write d's data
4. decrement link count

This way, the link count never under underestimates the true value. It can, however, overestimate, but this is better. This way we may lose disk space but cannot lose data.

Invariants

It is important to have a model for invariants for your file system. An invariant is a condition that should always be true. Otherwise there can be serious consequences, like data loss.

Example of invariants for a BSD-style file system include:

1. Every block of f.s. is used for exactly one purpose. Otherwise, there is "disaster" (data loss).
2. All referenced blocks are initialized to data appropriate for that type. Otherwise, disaster.
3. All referenced blocks are marked used in the bitmap. Otherwise, disaster.
4. All unreferenced blocks are mareked free in the bitmap. Otherwise, there can be storage loss.

The fourth invariant is the least important and not really an invariant since its consequence isn't too serious. Other invariants exist as well and vary for different file systems.

Performance improvement ideas

1) Commit Record

1. write "data" outputs into a preliminary area, like a temp file
2. wait for all blocks to hit disk (reordering is okay)
3. copy temporary data back to the original, wait for it to hit the disk
4. write "done" to the commit record

```
BEGIN
        pre-commit phase        // work is being done, but you can back out
COMMIT or ABORT                 // atomic at lower level
        post-commit phase       // work is being done, but cannot back out
END
```

This process appears to be atomic to the user.

2) Journal

Showing what journal looks like.

It is possible to have only a journal and no cell blocks. This allows very fast writes and slow reads. Some cell storage can be cached to RAM

When the journal runs out, it can start over at the beginning of the jornal. This requires garbage collecting the journal as only some of the early parts of the journal can be used.

ext4 is an example of a journaled file system.

Logging protocols

Writeahead logs

- log planned writes
- implement them later

Writebehind logs

- log old data about to be written
- write new data into cell
- mark as done

When replaying logs after a crash, for writeahead logs replay left to right. For writebehind logs, replay right to left.