CS 111 Spring 2014

Lecture 13: File System Robustness

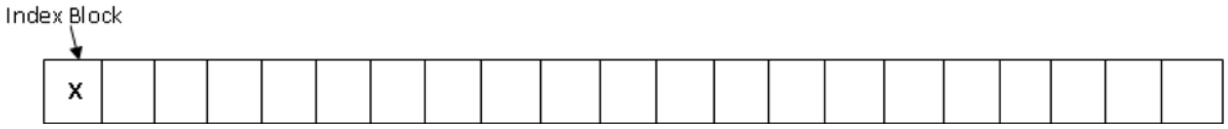
by Weichen Huang, Yuanzhi Gao, Pengcheng Pan, Yifang Liu

Problems in real SSD

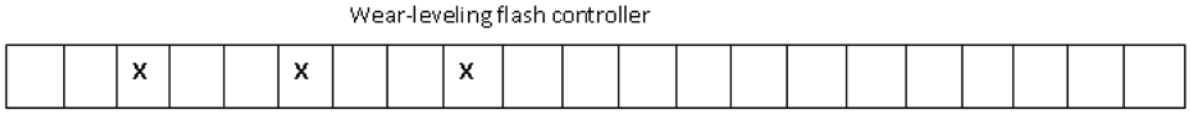
In Understanding the Robustness of SSDs under Power Fault by Zheng et al. at FAST '13, out of 15 SSDs, only 2 worked when they cut the power. The 13 SSDs that failed to work suffered from :

1. **bit corruption :** random bit errors on subsequent reads
2. **flying writes :** Block is written in wrong place when SSD restore.

   Further explanation of flying writes : Flash drive is not as reliable as hard drive. Flash may wear out after writing to the flash drive 10 billion times.

   What flash controller does to overcome this problem is they use wear-leveling.

   You think you are reading from block zero, but you are actually reading wherever block zero happens to be today because the blocks are moving around. So when you lose power when writing to flash, you may lose the table that maps the virtual and physical location, which will give inconsistancy.
3. **shorn writes :** Writes truncated and there is garbage sitting there in the rest of the block when you go back and read it.
4. **metadata corruption :** Drive loses track of much of data location (more serious problem). You can lose large trunk of your original data because it's out there in the physical disk, you just don't know where it is.
5. **dead device :** Fried data. After restore the power, absolutely nothing works. The entire device is dead.
6. **unserializability :** Actual writes are out of order compared to what you and this matters.

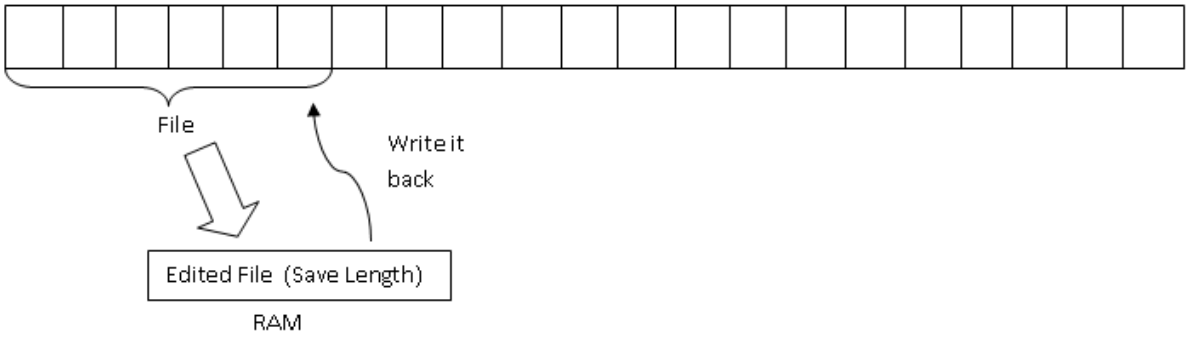
File System Robustness Goals

- **Durability** - survive limited failure in underlying hardware
- **Atomicity** - Changes are either done or not done
- **Performance** - throughput + latency

Implementing save in Emacs

Question : How should ^x^s (SAVE FILE) work on a raw device?

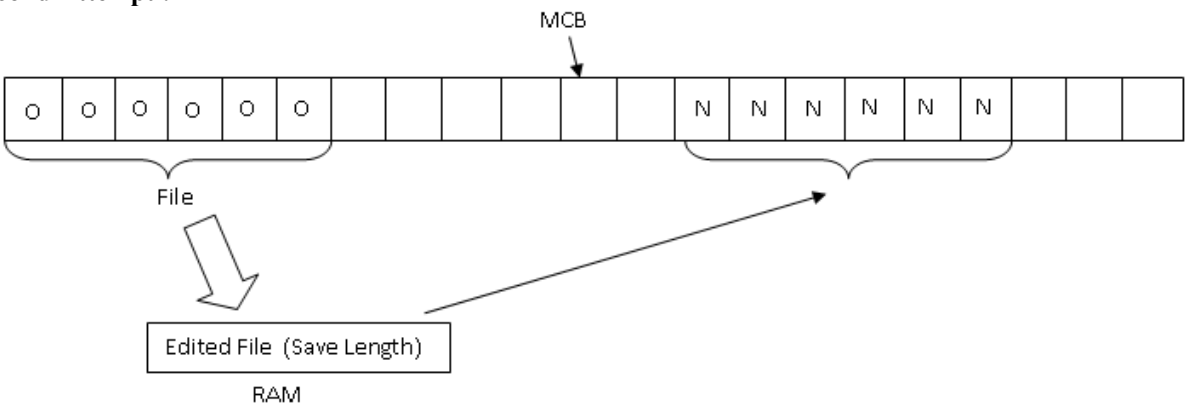
First Attempt :



Problem : What if unplugged at the middle of the write back?

GOLDEN RULE OF ATOMICITY : Never overwrite your only copy, i.e. write into a different location.

Second Attempt :



Instead of overwriting your original data, we will overwrite this area so that if the system crashes in the middle, you can continue to refer to your old copy it was not changed and once you finished writing it, you will have the new copy saved on disk as well. So in this way, we obey the golden rule of atomicity.

Problem : How do you know which copy you are using after rebooting?

Timestamp Approach :

What we can do at application level is that we can put a timestamp in each file. In operating system level is to put timestamp into every block that you write on the disk, which might help you to achieve the reliability here. But that can waste space in each block. Timestamp approach will work but it got its own problem(occupy too much space in block).

Master Control Block :

MCB is a block that we keep separate from files. We just store one bit into the master control block and if this bit is zero that means this is the old version of the file. if it's one that means this is the new version of the file.

But this approach does not survive all of the problems. For example, it won't survive the flying write, bit corruption: It actually can't survive any of the problems! But the master control block approach is better if we assume we are one of those 2 of 15 SSD's that actually work without having those problems (bit corruption, flying write).

Problem with using only one MCB :

Block size in file system is large. Our problem is that a write we have at file system level can’t be implemented atomically at the lower level because the device can’t simply write 8 KB block atomically a time. Let’s suppose writes

can be partial at the lower level. Let's say we pick an 8 KB file system block and the device is 512 bytes sectors. The way you can model it is: if you issue a high level write of 8KB, at the low level, you can issue 8 writes. The way to do it is to issue a bunch of writes to the disk controller and we are going to wait to get 16 responses back.

When you have a block A and you issue a write so that the new contents in the block is B instead of A, in between, the block has indeterminate content (some mixture of A and B that we don't care). Our model is in the middle of a write, if you lose power, the contents are indeterminate. We need an algorithm that helps us reliably tell whether we store a zero or one with its data.

One idea : we have two copy of MCB instead of just having one.

They both start up with the same content

MCB1 A ?
MCB2 A A

Then we start to write in MCB1

MCB1 A ? B
MCB2 A A A

As time goes on

MCB1 A ? B B B
MCB2 A A A ? B

Notice that at all time during execution, we have either A or B stored, so we will always have reliable data.

Problem : we can't tell which one is garbage and which one is correct after you reboot.

Another idea : use three MCBs

They start up having the same value

MCB1 A
MCB2 A
MCB3 A

Then we start to write in MCB1

MCB1 A ? B
MCB2 A A A
MCB3 A A A

As time goes on

MCB1 A ? B B B B B
MCB2 A A A ? B B B
MCB3 A A A A A ? B

After reboot, you look at all 3 :

1. If they all agree or If two out of three agree, then use that.
2. If all three disagree, the use MCB1.

Lampson-Sturgis Assumptions

- Storage writes may fail
  - a later read can detect the bad sector

- Storage may spontaneously decay
- Errors are rare
- Repairs can be done in time

Professor Eggert's Assumptions

- The only failure is a power failure
- Writes are atomic

Rename Problem

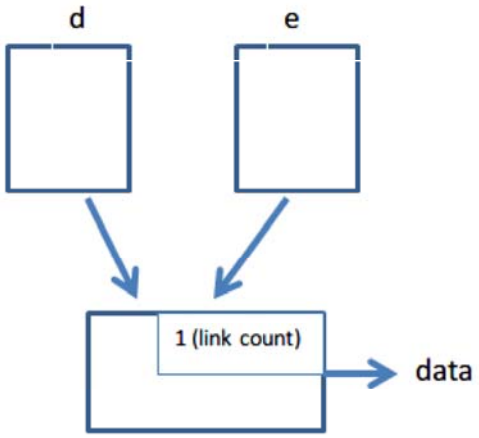
rename("a","aa");

Procedure to complete this request:

1. allocate a new block
2. write two blocks

Because directory entry varies in length, it can be longer. Rename remains atomic as long as the entry fits into block.

rename("d/a", "e/b"); - (assume e has space and has no "b")



Two hard links link to the same file and its link count is 1. When you unlink ("d/a"), link count will go to 0, and thus you will get a dangling link. So what is a safe procedure?

1. increment file's link count (write inode)
2. write e's new block
3. write d's new block
4. decrement file's link count (write inode)
5. write d's new timestamp into its inode
6. write e's new timestamp into its inode

File system correctness invariants

1. Every block in file system's partition is used for exactly 1 purpose: superblock, bitmap, inode, data.
2. All referenced blocks are properly initialized for their type
3. All referenced data blocks are marked as used in bitmap
4. All unreferenced data blocks are marked as free in bitmap

Consequence of :

- Violating property 1: multiple purposes at the same time -> you lose data -> disaster
- Violating property 2: allocating data twice -> you lose data -> disaster
- Violating property 3: you lose data -> disaster
- Violating property 4: disk space leak, though fsck would probably fix this

fsck

- inspects file system and look for storage leaks (while file system is idle)
- updates bitmap accordingly

If you lose power, assume fsck is idempotent.

File system Robustness Ideas: surviving power failures

Commit Record:

A commit is the single, low-level write operation that matters. It is the operation that actually determines whether or not a file has been written.

A general outline for Commit Record process:

BEGIN
  Pre-commit phase (can back out, leaving no trace)
Commit
  Post-commit phase (does not affect doneness, may be used for performance reasons such as clean-up)
END