CS323 Operating Systems

Filesystem journaling

Mathias Payer and Sanidhya Kashyap

EPFL, Fall 2021

Topics covered in this lecture

- Crash resistance
- Journaling

This slide deck covers chapters 42 in OSTEP.

Last two weeks: API, abstractions, disk layout

- Highlevel API and abstractions
- Filesystem API
- Different names for different use cases
    - Inodes and devices
    - Path
    - File descriptor
- Disk layout and inode/data block implementations

This week: Crash resistance and recovery

Recall atomic file update challenge

- Assume you want to update `important.txt` atomically

Recall atomic file update challenge

- Assume you want to update `important.txt` atomically
- If the application or the system crashes, the old version must remain
  - Write data to `./gener8 > important.txt.tmp`
  - Flush data to disk: `fsync important.txt.tmp`
  - Rename atomically: `mv important.txt.tmp important.txt`, replacing it
- What could still go wrong?

Recall atomic file update challenge

- Assume you want to update `important.txt` atomically
- If the application or the system crashes, the old version must remain
  - Write data to `./gener8 > important.txt.tmp`
  - Flush data to disk: `fsync important.txt.tmp`
  - Rename atomically: `mv important.txt.tmp important.txt`, replacing it
- What could still go wrong?
- File system metadata may not be written back to disk!

Crash resistance

- Power loss during writing
- Mechanical failure
- Magnetization failure
- Mechanical destruction (link)

Redundancy

Given A and B. If knowing A allows you to infer some or all values of B then there is redundancy between A and B.

- RAID1: mirrored disks (complete redundancy)
- RAID5 or 6: parity blocks (partial redundancy)

Redundancy in a filesystem

- Directory entries and inode table
- Directory entries and inode link count
- Data bitmap and inode pointers
- Data bitmap and group descriptor (for sets of blocks)
- Inode file size and inode/indirect pointers

Advantages of redundancy

- Can improve reliability (recover from failures)
- Can improve performance (easier to read file size from inode than parsing the full structure)
- Requires more storage (inefficient encoding)
- Requires consistency (all sides must agree)

Consistency

Redundant data must be consistent to ensure correctness. Otherwise functionality may break.

- Keeping redundant data consistent is challenging
- Filesystem may perform several writes to redundant blocks
- The sequence of writes is not atomic
- Interrupts due to power loss, kernel bugs, hardware failure

Consistency scenario (1/2)

- Filesystem appends to a file
- Must write to inode, data bitmap, data block
- What happens if only some writes succeed?
  - 001 Bitmap
  - 010 Data
  - 100 Inode
  - 011 Bitmap and data
  - 101 Bitmap and inode
  - 110 Data and inode

Consistency scenario (2/2)

- Filesystem appends to a file
- Must write to inode, data bitmap, data block
- What happens if only some writes succeed?
  - 001 Bitmap: lost block
  - 010 Data: lost data write (i.e., file is not updated)
  - 100 Inode: references garbage (another file may use)
  - 011 Bitmap and data: lost block
  - 101 Bitmap and inode: reference garbage (from previous usage)
  - 110 Data and inode: another file may grab the block
- How would you order the writes?

Consistency scenario (2/2)

- Filesystem appends to a file
- Must write to inode, data bitmap, data block
- What happens if only some writes succeed?
  - 001 Bitmap: lost block
  - 010 Data: lost data write (i.e., file is not updated)
  - 100 Inode: references garbage (another file may use)
  - 011 Bitmap and data: lost block
  - 101 Bitmap and inode: reference garbage (from previous usage)
  - 110 Data and inode: another file may grab the block
- How would you order the writes?

- Data (nothing bad happens), bitmap (lost block is detectable), then inode

Consistency through filesystem check (1/3)

- After a certain number of mount operations (remember the mount count in the super block?) or after a crash, check the consistency of the filesystem!
- Hundreds of consistency checks across different fields
  - Do superblocks match?
  - Are all '.' and '..' linked correctly?
  - Are link counts equal to number of directory entries?
  - Do different inodes point to the same block?

Consistency through filesystem check (2/3)

- Q: Two directory entries point to the same inode, link count is 1

Consistency through filesystem check (2/3)

- Q: Two directory entries point to the same inode, link count is 1
- A: Update the link count to 2

Consistency through filesystem check (2/3)

- Q: Two directory entries point to the same inode, link count is 1
- A: Update the link count to 2
- Q: Inode link count is 1 but no directory links this file

Consistency through filesystem check (2/3)

- Q: Two directory entries point to the same inode, link count is 1
- A: Update the link count to 2
- Q: Inode link count is 1 but no directory links this file
- A: Link the file in a `lost+found` directory

Consistency through filesystem check (2/3)

- Q: Two directory entries point to the same inode, link count is 1
- A: Update the link count to 2
- Q: Inode link count is 1 but no directory links this file
- A: Link the file in a `lost+found` directory
- Q: A referenced block is marked as free in the bitmap

Consistency through filesystem check (2/3)

- Q: Two directory entries point to the same inode, link count is 1
- A: Update the link count to 2
- Q: Inode link count is 1 but no directory links this file
- A: Link the file in a `lost+found` directory
- Q: A referenced block is marked as free in the bitmap
- A: Update the bitmap to 1

Consistency through filesystem check (2/3)

- Q: Two directory entries point to the same inode, link count is 1
- A: Update the link count to 2
- Q: Inode link count is 1 but no directory links this file
- A: Link the file in a `lost+found` directory
- Q: A referenced block is marked as free in the bitmap
- A: Update the bitmap to 1
- Q: Two inodes reference the same data block

Consistency through filesystem check (2/3)

- Q: Two directory entries point to the same inode, link count is 1
- A: Update the link count to 2
- Q: Inode link count is 1 but no directory links this file
- A: Link the file in a `lost+found` directory
- Q: A referenced block is marked as free in the bitmap
- A: Update the bitmap to 1
- Q: Two inodes reference the same data block
- A: Make a copy of the data block

Consistency through filesystem check (2/3)

- Q: Two directory entries point to the same inode, link count is 1
- A: Update the link count to 2
- Q: Inode link count is 1 but no directory links this file
- A: Link the file in a `lost+found` directory
- Q: A referenced block is marked as free in the bitmap
- A: Update the bitmap to 1
- Q: Two inodes reference the same data block
- A: Make a copy of the data block
- Q: An inode points to an inexistent block

Consistency through filesystem check (2/3)

- Q: Two directory entries point to the same inode, link count is 1
- A: Update the link count to 2
- Q: Inode link count is 1 but no directory links this file
- A: Link the file in a `lost+found` directory
- Q: A referenced block is marked as free in the bitmap
- A: Update the bitmap to 1
- Q: Two inodes reference the same data block
- A: Make a copy of the data block
- Q: An inode points to an inexistent block
- A: Remove the reference

Consistency through filesystem check (3/3)

- Are these operations correct?

Consistency through filesystem check (3/3)

- Are these operations correct?

- The file system is inconsistent, so all we do is best effort!
- It's not obvious how to fix filesystem corruption
- Correct state is unkown, just that it is inconsistent
- FSCK is slow and may take hours (must read full disk)

Consistency through filesystem check (3/3)

- Are these operations correct?

- The file system is inconsistent, so all we do is best effort!
- It's not obvious how to fix filesystem corruption
- Correct state is unkown, just that it is inconsistent
- FSCK is slow and may take hours (must read full disk)

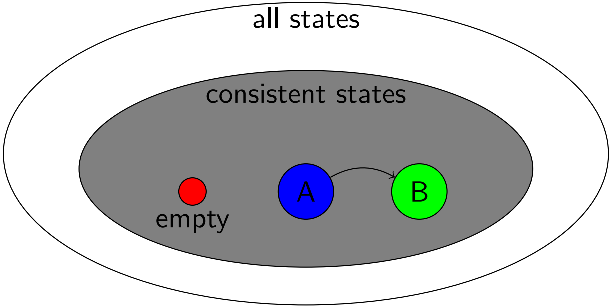
- Are there better approaches?

Consistency through journaling

- Goal: limit the amount of required work after crash
- Goal: get correct state, not just consistent state
- Strategy: atomicity
- Atomicity: being composed of indivisible units
  - *Concurrency*: operations in critical sections are not interrupted
  - *Persistence*: collections of writes are not interrupted by crashes (i.e., either all new or all old data is visible)

Consistency versus correctness

- Given: filesystem in state A, set of writes, resulting in state B
- Assume it crashes somewhere between the writes from A to B
  - Filesystem check (FSCK) gives consistency
  - Atomicity gives A or B

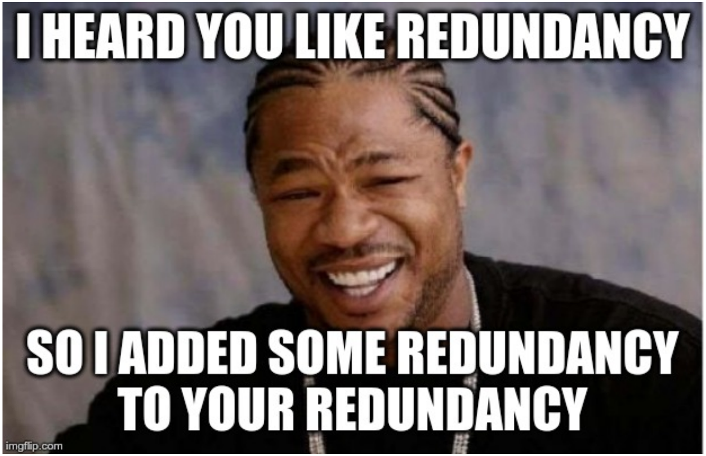


Journaling strategy

- Never delete (or overwrite) ANY old data until you have received confirmation that ALL new data is committed
  - Add redundancy to fix the problem with redundancy

Journaling strategy

- Never delete (or overwrite) ANY old data until you have received confirmation that ALL new data is committed
    - Add redundancy to fix the problem with redundancy


I HEARD YOU LIKE REDUNDANCY
SO I ADDED SOME REDUNDANCY TO YOUR REDUNDANCY
imgflip.com

Journaling strategy (1/4)

- Goal update file X with contents Y
  - Write Y, update metadata f(Y)

Journaling strategy (1/4)

- Goal update file X with contents Y
    - Write Y, update metadata f(Y)
- Classic strategy
    - Overwrite f(X) with f(Y), overwrite X with Y; or
    - Overwrite X with Y, overwrite f(X) with f(Y)
    - No matter the order, crash in the middle is bad!

Journaling strategy (1/4)

- Goal update file X with contents Y
  - Write Y, update metadata f(Y)
- Classic strategy
  - Overwrite f(X) with f(Y), overwrite X with Y; or
  - Overwrite X with Y, overwrite f(X) with f(Y)
  - No matter the order, crash in the middle is bad!
- Journaling strategy
  - Commit Y and f(Y) to journal
  - Update X with Y
  - Update f(X) with f(Y)
  - Delete journal entries
  - Resilient to crash in the middle, journal allows recovery

Journaling strategy (2/4)

- Goal: write 10 to block 0 and 5 to block 1 *atomically*

Time	Block 0	Block 1	Extra	Extra	Extra
0	12	3	0	0	0
1	10	3	0	0	0
2	10	5	0	0	0

Journaling strategy (2/4)

- Goal: write 10 to block 0 and 5 to block 1 *atomically*

Time	Block 0	Block 1	Extra	Extra	Extra
0	12	3	0	0	0
1	10	3	0	0	0
2	10	5	0	0	0

- This does not work! Must not crash between time 1 and 2!

Journaling strategy (3/4)

- Goal: write 10 to block 0 and 5 to block 1 *atomically*

Time	Block 0	Block 1	Block 0'	Block 1'	Valid?
0	12	3	0	0	0
1	12	3	10	0	0
2	12	3	10	5	0
3	12	3	10	5	1
4	10	3	10	5	1
5	10	5	10	5	1
6	10	5	10	5	0

Journaling strategy (3/4)

- Goal: write 10 to block 0 and 5 to block 1 *atomically*

Time	Block 0	Block 1	Block 0'	Block 1'	Valid?
0	12	3	0	0	0
1	12	3	10	0	0
2	12	3	10	5	0
3	12	3	10	5	1
4	10	3	10	5	1
5	10	5	10	5	1
6	10	5	10	5	0

- Crash before 3: old data
- Crash after 3: new data (need recovery)
- Crash after 6: new data

Journaling strategy (4/4)

```
// Pseudocode, assume we operate on blocks
void recovery() {
    if (*valid == 1) {
        *block0 = *block0p;
        *block1 = *block1p;
        *valid = 0;
        fsync();
    }
}
```

Journaling terminology

- Extra blocks are called ‘journal’
- Writes to the journal are a ‘journal transaction’
- The valid bit is a ‘journal commit block’

Journal optimizations

- Dedicated (small) journal area
- Write barriers
- Checksums
- Circular journal
- Logical journal
- Ordered journal

Journal optimization: small journal

- Allocating a shadow block per data block is wasteful
  - Recovery cost and lost storage

Journal optimization: small journal

- Allocating a shadow block per data block is wasteful
  - Recovery cost and lost storage
- Dedicate a small area of blocks to the journal
  - Store block number along with data
  - At the start of the transaction, mark which blocks are modified
  - Store the data blocks in the journal
  - Commit the transaction

Journal optimization: write barriers

- Enforcing total write order is costly (remember seek cost?)
- Idea: only wait until blocks of writes have completed
    - Wait before journal commit (journal data blocks were written)
    - Wait after journal commit (journal was committed)
    - Wait after data blocks are written (journal can be freed)

Journal optimization: checksums

- Can we get rid of the write barrier after journal commit?
- Idea: replace valid/invalid bit with checksum of written blocks
  - Checksum mismatch: one of the blocks was not written
  - Checksum match: all blocks were committed correctly
- We now only have two write barriers for each transaction
  - After writing the journal (make sure data ended up in journal)
  - Before clearing the journal entry (data was written to disk)

Journal optimization: circular buffer

- After data is written to journal, there is no rush to update/write back
    - Journaled data can be recovered
- Delay journaling for some time for better performance
    - Keep journal transactions in circular buffer
    - Flush when buffer space is used up

Journal optimization: logical journal

- Appending a block to the file causes writes to the data block, the inode, the data bitmap
  - Many small writes
  - Writing full blocks to journal is wasteful
- Idea: keep track how data changed (diff between old and new)
  - Logical journals record changes to bytes, not blocks
  - Save lots of journal space
  - Must read original block during recovery

Journal optimization: ordered journal

- How can we avoid writing all data twice?
- Idea: store only metadata in journal
  - Write data to new block
  - Store updates to metadata in logical journal
  - Commit journal (and new data blocks)
  - Update metadata
  - Free journal

Summary

- Crash resistance: filesystem check (FSCK)
- Journaling: keep track of metadata, enforce atomicity
  - All modern filesystems use journaling
  - FSCK still useful due to bitflips/bugs

Don't forget to get your learning feedback through the Moodle quiz!