File System Reliability / Crash Consistency

ECE595
Nov 13

Y. Charlie Hu

Roadmap

- Functionality (API)
  - Basic functionality
    - Disk layout
    - File operations (open, read, write, close)
  - Directories
- Performance
  - Disk allocation
  - Buffer cache
    - Interactions with VM
  - File System Interface
  - Disk scheduling
- Reliability
  - FS level
  - Disk level: RAID

File system reliability

- Loss of data in a file system can have catastrophic effect
  - How does it compare to hardware (DRAM) failure?
  - Need to ensure safety against data loss
- Three threats:
  - Accidental or malicious deletion of data → backup
  - Media (disk) failure → disk mirroring (RAID)
  - System crash during file system modifications → consistency

1. Backup

- Copy entire file system onto low-cost media (tape), at regular intervals (e.g. once a day).
  - Implementation – do we need to copy the whole FS?
- In the event of a disk failure, replace disk and restore from backup media
- Amount of loss is limited to modifications occurred since last backup

2. Mirrored Disks

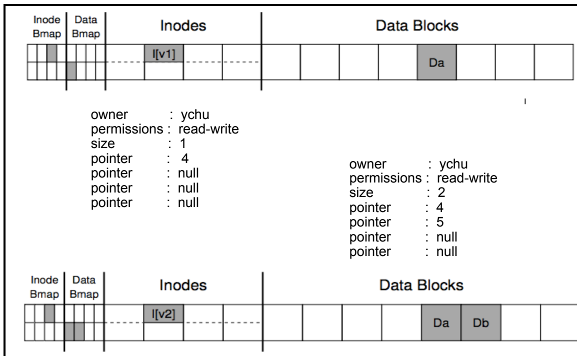
- Multiple copies of the file system are maintained on independent disks
- Disk writes update all redundant disks in parallel
- Used in applications that cannot tolerate any data loss (what applications?)

RAID Disks
(redundant array of independent disks)

- Use multiple parallel disk drives for higher throughput and increased reliability
- e.g. each bit of a data byte is stored on one of 8 disks, a 9th disk stores a *parity bit* for each data byte
- Can recover the data byte if 1 disk fails
- (more next week)

3. Crash Recovery

- After a system crash in the middle of a file system operation, file system metadata may be in an *inconsistent state*
  - Independent of buffer caching



3. Crash Recovery

- After a system crash in the middle of a file system operation, file system metadata may be in an *inconsistent state*
  - Independent of buffer caching
  - Examples:
    - "rm file1":
      - free disk blocks: put them back to the freelist
      - crash → free inode: put back to free inode list
      - crash → remove entry from parent dir data block
  - Can you avoid problems by reordering?

3. Crash Recovery

- After a system crash in the middle of a file system operation, file system metadata may be in an *inconsistent state*
  - Independent of buffer caching
  - Examples:
    - "rm file1":
      - free disk blocks: put them back to the freelist
      - crash → free inode: put back to free inode list
      - remove entry from parent dir data block
    - "rm file1":
      - remove entry from parent dir data block
      - crash → free disk blocks: put them back to the freelist
      - free inode: put back to free inode list

Deep thinking

- One file operation may involve modifying multiple disk blocks (and hence multiple disk I/Os)
- After crashing, do we know which blocks were involved at the moment of crashing?

Crash Recovery – Solution 1

- Run a program during system startup that examines the *entire* file system, detects inconsistencies, and restores the invariants
- In Unix, *fsck* checks and repairs
  - Correct i-node reference counts (hardlink)
  - Disconnected (unreachable) files and dirs
    - e.g., lost & found
  - Missing blocks in the freelist
  - Blocks that are both in the freelist and part of a file
  - Incorrect information in the superblock
    - free-block count and pointers…

- "rm file1":
  - remove from parent dir data block
  - free inode (put back to free inode list)
  - free disk blocks; put them back to the freelist

More deep thinking

- A file operation can involve changing 3 disk blocks (and hence 3 disk I/Os)
- Can we make the disk I/Os involved atomic?
- If not, can we at least narrow down suspects upon rebooting?

Crash Recovery – Solution 2

- Keep a separate log (e.g. non-volatile mem)
  - Write log entry describing operations about to be performed
  - Perform the operation on the file system
  - Delete the log entry after done
  - After a crash
  - Check the log
  - If non-empty, perform operations described in the log
- Example: "rm file1":
  - free disk blocks: put them back to the freelist
  - free inode: put back to free inode list
  - remove from parent dir data block

Yet more deep thinking

- A file operation can involve changing 3 disk blocks (and hence 3 disk I/Os)
- Can we make the disk I/Os involved atomic?
- If not, can we at least narrow down suspects upon rebooting?
- What if we do not have non-volatile mem?

Journaling file system (without non-volatile mem)

- Keep a small log (on disk), write each set of changes to it first
  - Always append to end of log
  - All synchronous writes go to this log
  - Each set of operations for a task is a transaction
  - Log entries replayed on the file system in the background
  - Upon reboot, if log not empty, know what to do or undo
- The changes are thus made to be atomic, in that they either
  - succeed (succeeded originally or are replayed completely during recovery), or
  - are not replayed at all (are skipped because they had not yet been completely written to the journal before the crash occurred).

Journaling file system
(without non-volatile mem)

- Keep a small log (on disk), write each set of changes to it first
  - Always append to end of log
  - All synchronous writes go to this log
  - Each set of operations for a task is a transaction
  - Log entries replayed on the file system in the background
  - Upon reboot, if log not empty, know what to do or undo
- Benefits
  - Fewer seeks for synchronous writes
    - Much faster metadata-oriented operations (open, delete)
  - Recovery time = scanning just the log
- Linux ext3, ReiserFS, UFS on Solaris 7 and above, NTFS, Veritas

Reading

- Chapters 11-12