FFS's Consistency

Berkeley FFS (Fast File System) formalized rules for file system consistency

FFS acceptable failures:

- May lose some data on crash
- May see someone else's previously deleted data
  - Applications must zero data out if they wish to avoid this + fsync
- May have to spend time to reconstruct free list
- May find unattached inodes → lost+found

Unacceptable failures:

- After crash, get active access to someone else's data
  - Either by pointing at reused inode or reused blocks

FFS uses 2 synchronous writes on each metadata operation that creates/destroy inodes or directory entries, e.g., creat(), unlink(), mkdir(), rmdir()

- Updates proceed at disk speed rather than CPU/memory speed

Write Ordering & Logging

Problem: as disk sizes grew, fsck becomes infeasible

- Complexity proportional to used portion of disk
- takes several hours to check GB-sized modern disks

In the early 90s, approaches were developed that

- Avoided need for fsck after crash
- Reduced the need for synchronous writes

Two classes of approaches:

- Write-ordering (aka Soft Updates)
  - BSD – the elegant approach
- Journaling (aka Logging)
  - Used in VxFS, NTFS, JFS, HFS+, ext3, reiserfs

Write Ordering

Instead of synchronously writing, record dependency in buffer cache

- On eviction, write out dependent blocks before evicted block: disk will always have a consistent or repairable image
- Repairs can be done in parallel – don't require delay on system reboot

Example:

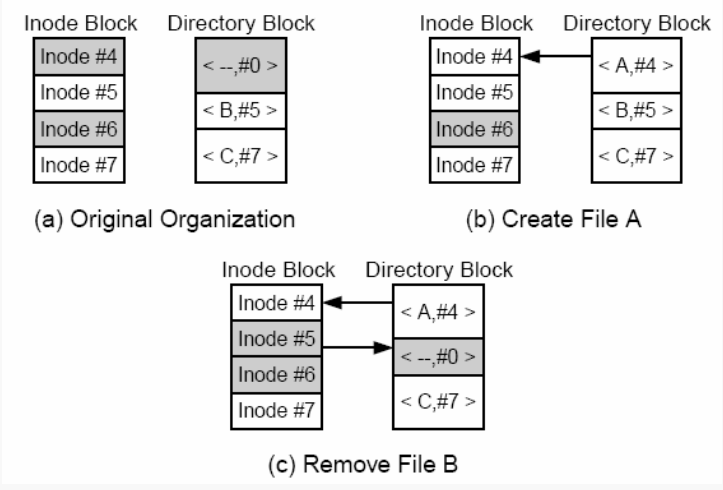
- Must write block containing new inode before block containing changed directory pointing at inode

Can completely eliminate need for synchronous writes

Can do deletes in background after zeroing out directory entry & noting dependency

Can provide additional consistency guarantees: e.g., make data blocks dependent on metadata blocks

Write Ordering: Cyclic Dependencies



Tricky case: A should be written before B, but B should be written before A? … must unravel

Logging Filesystems

Idea from databases: keep track of changes

- "write-ahead log" or "journaling": modifications are first written to log before they are written to actually changed locations
- reads bypass log

After crash, trace through log and

- redo completed metadata changes (e.g., created an inode & updated directory)
- undo partially completed metadata changes (e.g., created an inode, but didn't update directory)

Log must be written to persistent storage

Logging Issues

How much does logging slow normal operation down?

Log writes are sequential

- Can be fast, especially if separate disk is used
- Subtlety: log actually does not have to be written synchronously, just in-order & before the data to which it refers!
  - Can trade performance for consistency – write log synchronously if strong consistency is desired

Need to recycle log

- After "sync()", can restart log since disk is known to be consistent

Physical vs Logical Logging

What & how should be logged?

Physical logging:

- Store physical state that's affected
  - before or after block (or both)
- Choice: easier to redo (if after) or undo (if before)

Logical logging:

- Store operation itself as log entry (rename("a", "b"))
- More space-efficient, but can be tricky to implement

Summary

Filesystem consistency is important

Any filesystem design implies metadata dependency rules

Designer needs to reason about state of filesystem after crash & avoid unacceptable failures

- Needs to take worst-case scenario into account – crash after every sector write

Most current filesystems use logging

- Various degrees of data/metadata consistency guarantees