CS 3204
Operating Systems

Lecture 23

Godmar Back

Announcements

- Project 4 Help Session Monday 6-8pm room TBD

Filesystems

Consistency & Logging

Filesystems & Managing Faults

- Filesystem's promise:
  - Persistence
  - Defined behavior in the presence of faults
- Need Failure Model
  - Define acceptable failures (disk head hits dust particle, scratches disk – you will lose some data)
  - Define which failure outcomes are unacceptable
- Need a strategy to avoid unacceptable failure outcomes
  - Proactive (avoid them)
  - Reactive (recover from them)

Proactive Methods

- Use atomic changes
  - construct larger atomic changes from the small atomic units available (i.e., single sector writes)
- Idea: use state duplication

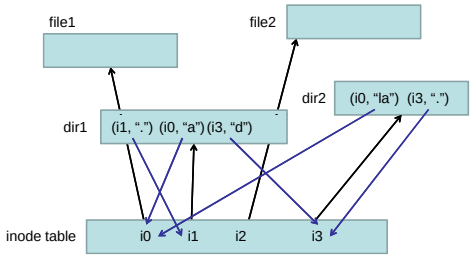
```
write(data)
  version++
  write_atomic(versionloc1, version)
  write_multiple(dataloc1, data)
  write_atomic(versionloc2, version)
  write_multiple(dataloc2, data)
```

```
read()
  v1 = read(versionloc1)
  v2 = read(versionloc2)
  data1 = read(dataloc1)
  data2 = read(dataloc2)
  return v1 == v2 ? data1 : data2;
```


Reactive Methods

- Option 1: On failure, retry entire computation
  - not a good model for persistent filesystems
- Option 2: Define recovery procedure to deal with unacceptable failures:
  - Recovery moves from an incorrect state A to correct state B
  - Must understand possible incorrect states A after crash!
  - A is like "snapshot of the past"
  - Anticipating all states A is difficult
- For file systems, option 2 means to ensure that on-disk changes are ordered such that if crash occurs after any step, a recovery program can either undo change or complete it

Unix Filesystem Overview



Sensible Invariants

- In a Unix-style file system, *ideally*, we would want that:
  - File & directory names are unique within parent directory
  - Free list/map accounts for all free objects
    - all objects on free list are really free
  - All data blocks belong to exactly one file (only one pointer to them)
  - Inode's ref count reflects exact number of directory entries pointing to it
  - Don't show previously deleted data to applications
- Some of these invariants
  - can be reestablished via fsck after crash
  - must be maintained proactively before crash (either because fsck couldn't fix them or because it would lead to unacceptable state if user skipped fsck)
  - are not maintained because of cost

Crash Recovery (fsck)

- After crash, fsck runs and performs the equivalent of mark-and-sweep garbage collection
- Follow, from root directory, directory entries
  - Count how many entries point to inode, adjust ref count
- Recover unreferenced inodes:
  - Scan inode array and check that all inodes marked as used are referenced by dir entry
  - Move others to /lost+found
- Recomputes free list:
  - Follow direct blocks+single+double+triple indirect blocks, mark all blocks so reached as used – free list/map is the complement
- In following discussion, keep in mind what fsck could and could not fix!

Example 1: file create

- On create("foo"), have to
  1. Scan current working dir for entry "foo" (fail if found); else find empty slot in directory for new entry
  2. Allocate an inode #in
  3. Insert pointer to #in in directory: (#in, "foo")
  4. Write a) inode & b) directory back
- What happens if crash after 1, 2, 3, or 4a), 4b)?
- Does order of inode vs directory write back matter?
- Rule: never write persistent pointer to object that's not (yet) persistent

Example 2: file unlink

- To unlink("foo"), must
  1. Find entry "foo" in directory
  2. Remove entry "foo" in directory
  3. Find inode #in corresponding to it, decrement #ref count
  4. If #ref count == 0, free all blocks of file
  5. Write back inode & directory
- Q.: what's the correct order in which to write back inode & directory?
- Q.: what can happen if free blocks are reused before inode's written back?
- Rule: first persistently nullify pointer to any object before freeing it (object=freed blocks & inode)

Example 3: file rename

- To rename("foo", "bar"), must
  1. Find entry (#in, "foo") in directory
  2. Check that "bar" doesn't already exist
  3. Remove entry (#in, "foo")
  4. Add entry (#in, "bar")
- Suppose crash after directory block containing old "foo" was written back, but before "bar" entry was written back

Example 3a: file rename

- To rename("foo", "bar"), conservatively
  1. Find entry (#i, "foo") in directory
  2. Check that "bar" doesn't already exist
  3. Increment ref count of #i
  4. Add entry (#i, "bar") to directory
  5. Remove entry (#i, "foo") from directory
  6. Decrement ref count of #i
- Worst case: have old & new names to refer to file
- Rule: never nullify pointer before setting a new pointer

Example 4: file growth

- Suppose file_write() is called.
  - First, find block at offset
- Case 1: metadata already exists for block (file is not grown)
  - Simply write data block
- Case 2: must allocate block, must update metadata (direct block pointer, or indirect block pointer)
  - Must write changed metadata (inode or index block) & data
- Both writeback orders can lead to acceptable failures:
  - File data first, metadata next – may lose some data on crash
  - Metadata first, file data next – may see previous user's deleted data after crash (very expensive to avoid – would require writing all data synchronously)

FFS's Consistency Model

- 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:
  - Inability to bring filesystem back into consistent state via fsck
  - 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/destroys inodes or directory entries, e.g., creat(), unlink(), mkdir(), rmdir()
  - Note: assumption here is that eviction order in underlying buffer cache is not controllable, hence need for synchronous writes

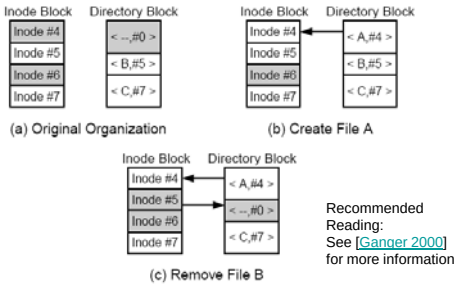
Write Ordering & Logging

- Problem (1) with using synchronous writes
  - Updates proceed at disk speed rather than CPU/memory speed
- Problem (2) with using fsck
  - complexity proportional to used portion of disk
  - takes several hours to check GB-sized modern disks
- In the early 90s, approaches were developed that
  - Reduced the need for synchronous writes
  - Avoided need for fsck after crash
- 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 to using the filesystem – 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



Recommended Reading: See [Ganger 2000] for more information

- Tricky case: A should be written before B, but B should be written before A? … must unravel by introducing intermediate versions

Logging File Systems

- 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 kept in 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 by a change
    - before or after block (or both)
  - Choice: easier to redo (if after) or undo (if before)
  - Advantage: can retrofit logging independent of logic that manipulates metadata representation
- Logical logging:
  - Store operation itself as log entry (rename("a", "b"))
  - More space-efficient, but can be tricky to implement since metadata operation logic is involved in log replay

Summary

- File system consistency is important
- Any file system design implies metadata dependency rules
- Designer needs to reason about state of file system after crash & avoid unacceptable failures
  - Needs to take worst-case scenario into account – crash after every sector write
- Most current file systems use logging
  - Various degrees of data/metadata consistency guarantees