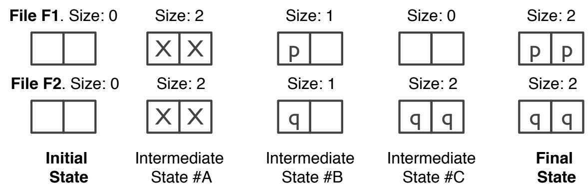
File System Consistency

1. Reviews:
    a. Starting next Tuesday, review format will get a lot simpler:
        i. Summary
        ii. Confusions
2. Topics for end of semester
    a. Security
    b. Reliability
    c. Power management
    d. Manageability
    e. GPUs
    f. Device drivers
3. Questions from reviews:
    a. Why is sequential overwrite bad?
        i. Has to journal write as doing in-place write
    b. More on FUA:
        i. Allow a single write to be forced to media; does not need to flush cache
        ii. Not guaranteed to work on SATA
            1. Disks lie to get better benchmarks
        iii. Used for journal writes to avoid full flushes
    c. Async notifications?
        i. Normally interrupt to signal acceptance of write by disk into cache
        ii. Why 2 notifications – one of accepting write?
            1. Allows OS to remove from queue to disk
    d. Why some applications more prone to probabilistic failures?
        i. Do they have ordering requirements? Call fsync() frequently?
    e. Industry use?
        i. Yes – Azure's block storage system, other cloud storage systems
    f. Relationship to GFS
        i. Are failures the norm?
            1. Scale: among 1000 machines, it is normal for one of them to fail
            2. For a single machine, failure is not the norm
4. Consistency problem:
    a. File systems are complex data structures
    b. Inconsistencies possible if updates partially complete
        i. Add data block to file + remove from free list
        ii. Add file to directory + write inode
5. What do applications have to know?
    a. How do applications enforce their own consistency rules?
        i. Use fsync() to make things durable before writing
            1. Write new file, fsync(), rename, fsync() to make rename
    b. What consistency guarantees do file systems make:

        i. Are operations delayed or not
            1. Ext4 story with delayed write; many apps depended on 30-second writeback
        ii. Example: write (f1, “pp”); write (f2, “qq”)
            1. 

            2. State A: length updated but not data
            3. State B: partial write (torn / not atomic)
            4. State C: second write persists first (out of order)
        iii. Does FS ensure operations written out in order or not – writes to different files are persisted in order?
            1. Not guaranteed to be true
        iv. Are writes atomic?
            1. Can you update multiple blocks but only have some of update show?
            2. Can the size of a file change (inode) without data showing up?
    c. What applications do this?
        i. Databases: write logs first, then data
        ii. Email servers: write email message, then fsync, before replying to client
6. Solutions:
    a. **So nothing**: FFS, FAT32, EXT2
        i. Run FSCK to fix things up afterwards
    b. **Pessimistic approaches** –
        i. Make sure ordering is enforced by disk
    c. Ordering every operation
        i. Write free block bitmap (BM)
        ii. write data (FB)
        iii. Write inode (IN)
        iv. Ordering: BM -> FB -> IN
    d. Shadow updates
        i. Write all new data
            1. New file blocks (FB)
            2. New file inode (IN)
            3. New free block bitmap (BM)
        ii. Swing pointer to new data
            1. Inode map (IM) pointing to inode
        iii. Ordering:
            1. (FB, IN, BM) -> IM
        iv. Note: uses copy-on-write (like LFS)
    e. (ordered) Journaling:

- i. Write data; make sure is durable (FB)
- ii. Write everything to a journal first
    1. New file inode (IN)
    2. New bitmap (BM)
- iii. Commit journal (JC)
    1. Why?
    2. Not all journal blocks may make it out; need to wait for them all to be durable before commit
- iv. Write metadata (checkpointing)
    1. File inode (IN)
    2. Bitmap (BM)
- v. Ordering:
    1. FB -> J(IN,BM) -> JC -> IN,BM -> journal clean
- vi. Note: when can you clean up the journal?
    1. After checkpointing
- vii. Note: when do you have to write back metadata?
    1. Any time you want
- viii. Recovery: if recover after JC, roll forward and write back metadata
    1. Else discard journal
- ix. Note: must hold metadata in memory until everything else is written
    1. Not safe to write early

f. All solutions require ordering:
- i. Need to know where you are in the steps so know what done/what not done
- ii. Need to know if operation is complete
    1. If too early, roll back (lack complete information)
    2. If past commit point, roll forward – fix up missing operations

g. How can you do this with a disk?
- i. Disks can reorder everything internally for reducing seek time / rotation time
- ii. Ordering primitive is *flush*
- iii. Flush cache and wait for it to complete
- iv. Guarantee: what is guarantee of a flush?
    1. Anything after a flush takes place after anything before a flush
    2. Nothing after a flush can hit disk before everything before a flush
- v. Note: no other way to know that write completed **except** do a flush after the write (excluding force-unit-access FUA operations)
    1. FUA writes a single operation out to disk bypassing cache
    2. Was often used for writing journal in NTFS, EXT4
    3. Problem: most disks now are SATA, work reliably in SCSI/SAS but not in SATA

h. Problems:
- i. Flushes are slow
- ii. Conflates ordering and durability

            1. After flush, everything before flush is durable
                a. Will survive power failure/system crash
            2. Sometimes, want ordering but don't need durability
                a. **QUESTION: Is this true?**
                b. **QUESTION: Examples of when?**
    i. **Probabilistic consistency**:
        i. Do everything above, but don't enforce at disk level
        ii. Issues writes in order, hope they complete in order
        iii. Window of vulnerability:
            1. Period when some of the blocks of a transaction have been written out
                a. E.g. new inode pointing to data block before data block
                b. E.g. journal transaction before data
            2. After all blocks out, inconsistency goes away
            3. Overall, fraction of time where a crash would cause inconsistency is probability of inconsistency.
        iv. When a good assumption?
            1. Writes are sequential
            2. Writes have large time gap
                a. Reordering is across a small span
            3. Writes have a large space gap (far apart on disk)
                a. Tends to cluster journal writes/data writes so they don't mix
        v. Else a bad assumption
        vi. Who does this?
            1. MacOS – doesn't actually wait for data to go to disk
        vii. Why some applications more vulnerable?
            1. More operations that require consistency
                a. Database, email server

7. Application to databases:
    a. Write a log for a transaction
        i. Commit to disk
    b. Write the data
    c. Truncate log after data written to disk
8. Techniques to reduce ordering
    a. Checksumming:
        i. Basic idea: if you want something atomic (all or nothing)
            1. Write the data + a checksum someplace **new**
            2. If checksum matches, all data was written, use it
            3. If checksum does not match, some data was not written, do not use it
            4. Note: cannot use for in-place updates
        ii. Where use:
            1. Journal commit: write journal entries + checksum instead

        2. Data append: write data checksum in journal; if checksum fails abort transaction
    b. Asynchronous durability notification:
        i. Notification that a previous write completed **without** a flush
        ii. O.k. to clean log, reuse a block that was previously used, etc.
9. Optimistic concurrency
    a. Goals:
        i. Want to write at full speed (no flushes)
        ii. Recovery consistently but not to latest transaction
            1. O.k. to keep a prefix of writes only
    b. Big idea:
        i. Write data out of order, using checksums for atomicity
        ii. On recovery, walk log and complete every fully formed transaction
        iii. For operations that require ordering (reusing blocks, cleaning logs) wait for disk to acknowledge data is durable rather than forcing data to be durable
    c. Techniques:
        i. Data checksumming: put data checksum in journal
        ii. Transaction checksumming: commit transaction by including checksum
            1. Net result: can tell from checksums if complete transaction was written or not; allows **atomicity**
        iii. Delay metadata checkpoint until preceding writes **durable**
            1. Use async. Durability notifications instead of flush
            2. May buffer writes for a long time
        iv. Ordering depends on preceding transactions
            1. Cannot write metadata for TX3 if TX1 and TX2 are not durable
                a. Journal/data for TX3 is not enough
    d. What ordering remains?
        i. (d,JM,JC) -> M
        ii. M -> clean J
        iii. Note: both off critical path!
    e. Cleaning TX
        i. Can only clean when metadata is durable
            1. QUESTION: WHY?
                a. Know that won't have to repeat journal
            2. Needs AND
    f. Recovery:
        i. Walk journal, re-execute TX that are complete
        ii. QUESTION: When stop?
            1. When get to first TX with failed checksum;
            2. Indicates incomplete TX or data didn't write
    g. Reusing blocks
        i. Problem: TXi frees block, TXj uses block
        ii. Data write for TXj completes before TXi's commit block, then crash

- iii. On recovery, TXi rolled back -> block still in old file; new data is there; wrong data in file
- iv. Solution: don’t reuse block until previous metadata write durable
  - 1. Big idea: wait don’t flush

h. In-place updates:
- i. Can use copy-on-write and allocate new block, but hurts locality for sequential files
- ii. Solution: **selective data journaling** where new data written to log first
  - 1. In-place update only happen after ADN for TX
  - 2. Benefit: makes data writes sequential; good for random write workloads
- iii. Why selective?
  - 1. For appends, no need to keep old value to abort transaction

10. Consistency vs durability
- a. Durability: after a crash data will be there
- b. Consistency: some prefix of data will be there
- c. Fsync() currently does both
- d. **QUESTION:** When want just consistency?
  - i. Multi-stage update; e.g. new files on a web server
    - 1. Use osync between stages; dsync at end to make sure all done
  - ii. Freshness not that important
    - 1. Logging, statistics
    - 2. Generated reports, intermediate files

11. Evaluation:
- a. QUESTION: How evaluate something like this?
  - i. Is it correct?
    - 1. Write a test that stresses correctness
      - a. Lots of dependent writes (same file, same directory)
      - b. Crash simulation: take possible reorderings of writes & try to boot FS
  - ii. Performance?
    - 1. Run applications using dsync instead of fsync
    - 2. Use osync instead of fsync
    - 3. Run applications that don’t call fsync
  - iii. Space
    - 1. Measure mem usage
    - 2. Measures CPU usage – may be blocked waiting on disk
- b. Techniques:
  - i. Disk simulators: to look at amount of reordering
  - ii. Reordering simulation:
    - 1. Look at order of blocks between flushes. Legally, they can be completely reordered. Try some
  - iii.