# JOURNALING FILE SYSTEMS

CS124 – Operating Systems

Spring 2024, Lecture 24

# File System Robustness

- The operating system keeps a cache of filesystem data
  - Secondary storage devices are much slower than main memory
  - Caching frequently-used disk blocks in memory yields significant performance improvements by avoiding disk-IO operations
- Problem 1:  Operating systems crash.  Hardware fails.
- Problem 2:  Many filesystem operations involve multiple steps
- Example:  deleting a file minimally involves removing a directory entry, and updating the free map
  - May involve several other steps depending on filesystem design
- If only some of these steps are successfully written to disk, filesystem corruption is highly likely

# File System Robustness (2)

- The OS should try to maintain the filesystem's correctness
  - …at least, in some minimal way…
- Example: ext2 filesystems maintain a "mount state" in the filesystem's superblock on disk
  - When filesystem is mounted, this value is set to indicate how the filesystem was mounted (e.g. read-only, etc.)
  - When the filesystem is cleanly unmounted, the mount-state is set to **`EXT2_VALID_FS`** to record that the filesystem is trustworthy
- When OS starts: if it sees an ext2 drive mount-state as not **`EXT2_VALID_FS`**, it knows something happened
  - The OS can take steps to verify the filesystem, and fix it if needed
- Typically, this involves running the **`fsck`** system utility
  - "File System Consistency checK"
  - (Frequently, OSes also run scheduled filesystem checks too)

# The `fsck` Utility

- To verify the filesystem, must perform various exhaustive checks of the entire filesystem layout and data structures
- E.g. for ext2 filesystems, must check these things:
  - Verify that inode metadata (specifically, file size) matches the number of blocks referenced (directly and indirectly) by the inode
  - Verify that all directory entries reference inodes (and that active inodes are referenced by directory entries)
  - Verify that all directory entries are reachable from the device root
  - Verify that inode reference-counts match how many directory entries reference them
  - Verify that the set of blocks referenced by inodes actually matches up with the state of the free-space map
- Any errors along the way are fixed as best as **`fsck`** can

# Improving File System Recovery

- Of course, all these exhaustive checks are very slow…
- As storage device sizes grew over the years, file-system consistency checks became *extremely* slow
  - Would often take hours to complete
- Needed to find a way to ensure filesystem robustness, without having to spend so much time on verification
- Solution: record [some] filesystem operations in a **journal** on disk, before writing to the filesystem data structures
- When system crash occurs, perform recovery from journal
  - Should restore the system to a known-good state, without requiring exhaustive verification of the entire filesystem
  - Recovering from the journal will be *much* faster – only need to consider logged operations, not the entire filesystem structure

# Filesystem Journaling

- Certain operations must be performed atomically on the filesystem
  - Either all of the operations are applied, or none are applied
  - Examples:  extending a file, deleting a file, moving a file, etc.
  - All of these are comprised of multiple lower-level operations
- The filesystem journal logs **transactions** against the filesystem
  - Transactions can either include one atomic operation, or multiple atomic operations, depending on filesystem design
- Note:  Not as sophisticated as database transactions!
  - No ACID properties, no concurrency control (not actually needed)
  - The filesystem simply attempts to maintain consistency by ensuring that transactions are applied atomically

# Filesystem Journaling (2)

- Like the free-map, the filesystem journal is a separate region of the disk volume, devoted to journaling
  - Often implemented as a circular queue large enough to hold multiple transactions
- **What should be logged in a journal transaction?**
  - Filesystems differ in the actual details that are logged…
- Many filesystems only journal changes to metadata
  - i.e. changes to directory structures, file inode information, free space map, any other structures the filesystem maintains on storage devices
  - **Changes to file data are not journaled!** (This is mostly OK.)
- After a crash, a given file's contents might become corrupt, but the overall filesystem structure will stay correct
  - Reason: writes to data and metadata might be interleaved
  - Metadata-changes can hit the disk before data-changes do
  - If a crash occurs between the two, the file will likely contain garbage

# Filesystem Journaling (3)

- This issue can occur with operations that affect both a file's data and metadata
  - Primary scenario: file extension
  - If file's metadata was updated to indicate that it is extended, but the actual data wasn't written, the file will become corrupt
- Can improve robustness by following an ordering rule:
  - All data-changes must be written to disk before any metadata-changes are logged to the journal
  - Note: changes to file data are still not journaled
- This primarily improves the robustness of file-extension operations (which occur very frequently)
- Places an overhead on the filesystem implementation:
  - Before journal records may be written to disk, the OS must make sure that all corresponding data blocks have been written out

# Filesystem Journaling (4)

- Finally, filesystems can log <u>all</u> data and metadata changes to the journal
  - Imposes a <u>significant</u> space overhead on the journal, as well as a time overhead
  - All data ends up being written twice – once to journal, once to file
  - Also is the best way to ensure that files cannot become corrupt
- Modern journaling filesystems often support multiple levels of operation
- Example:  ext3/ext4 supports three journaling modes
  - "Writeback" only records metadata changes to the journal
  - "Ordered" (default) records metadata changes to the journal, after the corresponding data changes have been written to the device
  - "Journal" records both data and metadata changes into the journal

# Atomic Operations

- Atomic operations generally correspond to the system calls that operate on the filesystem
  - Could be from many different processes, on behalf of various users
- An atomic operation could be comprised of several writes to the filesystem
- Example: append data to a file
  - Modify free-space map to allocate data blocks for the new data
  - Update file's inode index (possibly including indirect blocks) to reference new data blocks
  - Write the data to the new data blocks
  - Update file's inode metadata with new file size, modification time
- All of these writes must be performed, or none of them
  - (with the possible exception of the data write, depending on the journaling filesystem implementation and configuration)

# Atomic Operations and Transactions

- Since atomic operations correspond to system calls, will likely have a huge number of them…
- For efficiency, Linux groups multiple atomic operations together into a single transaction
- The entire transaction is treated as an atomic unit in the filesystem journal
  - All atomic operations in the transaction are applied, or none are
- The filesystem only maintains one “active” transaction at a time
  - The transaction that the filesystem is adding atomic operations to

- (This is why concurrency control and isolation aren’t needed; there is only one active transaction at a time.)

# Atomic Operations and Transactions (2)

- As atomic operations are performed, they are added to the current transaction, until one of the following occurs:
  - A fixed amount of time passes, e.g. 5 seconds
  - The journal doesn't have room to record another atomic operation
- At this point, the filesystem will "lock" the transaction
  - The transaction is closed
  - Any new atomic operations are logged in the next "active" transaction
- Of course, the transaction is still far from complete…
  - The transaction's logs may not yet be in the filesystem journal
  - Changes recorded in logs may not be applied to the filesystem
  - (In "ordered" mode, data changes may not yet be flushed to disk)

# Atomic Operations and Transactions (3)

- If a transaction's logs haven't been fully written to journal, it is in "flush" state
  - A crash during this state means the txn is aborted during recovery
- Once transaction logs are fully written to the journal, it enters "commit" state
  - All the logs are in the journal on disk, but the actual filesystem changes recorded in those logs haven't been completed
- Once all changes specified in the transaction have been written to filesystem, it is "finished"
  - The filesystem itself reflects all changes recorded in the txn logs…
  - Don't need to keep the transaction in the journal anymore!
  - It is removed from the circular queue that holds the journal

# Recovery

- The ext3 filesystem recovery mechanism only requires replaying the logs of transactions in the “commit” state
  - If a transaction is in the “finished” state, the filesystem already reflects the changes in the journal
- If a transaction is in the “commit” state:
  - All changes to filesystem metadata have been written to journal, but some changes may not be recorded to the filesystem itself
- All other transactions that haven’t reached the “commit” state are incomplete
  - The journal may not actually contain all parts of one or more atomic operations
  - Therefore, the filesystem recovery mechanism simply ignores these incomplete transactions

# Recovery: Requirements

- Note: for this to work, the filesystem must follow a rule:
  - No changes may be made to the filesystem metadata itself until the journal on disk reflects all changes being made in the transaction
  - i.e. the filesystem itself cannot be updated until the corresponding transaction enters the "commit" state
- Otherwise, the filesystem itself will include changes from an incomplete transaction…
  - The transaction could be aborted by a system crash…
  - In that case, those changes would need to be rolled back somehow
- To simplify recovery, ext3 imposes this requirement
  - (Computers have plenty of memory to hold modified data by now)
- Otherwise, the filesystem would also require some kind of undo-processing during recovery
  - (Note: just an ext3/ext4 design choice; undo-processing isn't "bad")

# Filesystem Journaling: Limitations

- Filesystem journaling is generally nowhere near as sophisticated as database transaction logging
- Journaling mechanism has a few simple goals:
  - Maintain the integrity of the filesystem
  - Avoid extremely costly, exhaustive consistency checks
- Frequently, other constraints are imposed to simplify recovery processing
  - Again, databases often are much more sophisticated in this area, and require complex redo/undo processing during recovery
- Typically, filesystems offer facilities more than sufficient to build very sophisticated transaction logging systems

# Filesystem Journaling: Benefits

- Obvious benefit: filesystems become significantly more robust, without costly exhaustive consistency checks!
- Another unexpected benefit: I/O performance (!!)
  - Produces benefits on both magnetic HDDs and SSDs
- The filesystem must write logs to the journal
  - These writes are basically all sequential
- The filesystem must batch up (frequently random) writes to data and metadata, to perform at specific times
  - The OS can often reorder these writes to minimize seek overhead
  - (Of course, OS must make sure not to violate ordering constraints)
- Journaling allows the operating system to interact with hard disks in a significantly higher-performance way

# Alternatives to Filesystem Journaling

- Several other interesting alternatives to incorporating a journal into the filesystem

**Soft Updates**

- The OS can carefully order changes to the file system to ensure that it never becomes corrupt, even in a crash
- Typically, the only issue that occurs in a crash is that free space is leaked
  - i.e. the filesystem thinks that space is unavailable, but no file is actually using it
  - This isn’t a corruption issue, so OS can resolve it in the background during normal operation
  - (Approach: scan through all file-system inodes; if a block isn’t referenced by any inode, reclaim it)

# Alternatives to Filesystem Journaling (2)

**Soft Updates** (cont.)

- Benefit: Can mount a filesystem immediately after a crash; no log to replay
  - (not that journal logs ever take that long to replay…)
- Difficulty: Requires *very careful* design and implementation of the file system
  - (implementers have to be <u>much</u> more careful)

- The Unix File System (UFS) uses soft updates
  - In BSD, this is also called the Fast File System (FFS)

# Alternatives to Filesystem Journaling (3)

**Log-Structured File Systems**

- Instead of having separate file-system disk structures and journal area, just use the journal as the file-system
  - The journal is still maintained as a (now very large) circular queue
- Reads against files and directories are resolved against the most recently logged journal entries
- Writes against files and directories are implemented as new records logged into the journal
- Rationale:
  - Sequential writes are much faster and easier to batch up than random writes
  - Also facilitates filesystem snapshotting, by looking at the journal at a specific point in time

# Alternatives to Filesystem Journaling (4)

**Log-Structured File Systems** (cont.)

- Recovery is very simple:  just identify the last point in time where the journal is consistent
- Main challenge:  when can journal entries be reclaimed?
- All log entries record writes…
  - If a later log entry records a write to the same data as an earlier log entry, the earlier entry can be discarded
  - If an early log entry has no later log entries, it can simply be moved forward in the log (typically to the head of the log)
- The OS implements basic garbage-collection mechanism to free up space in the journal using these techniques

# Alternatives to Filesystem Journaling (5)

**Log-Structured File Systems** (cont.)

- On HDDs, log-structured file systems generally don't work very well
  - Significant performance decreases on disk reads (seek overhead)
- But, a very appealing approach for flash-based storage…
  - Seeks are free, so no performance overhead on reads
- Recall:  flash devices really can't perform in-place writes
  - Can only write to empty cells; erase cells in larger erase-blocks
  - Also, each cell can only endure so many write-erase cycles
- Log-structured file systems satisfy these constraints well
  - File system doesn't perform in-place writes to filesystem data
  - Journal traverses the entire storage device in sequence, giving very even wear

# Alternatives to Filesystem Journaling (6)

**Log-Structured File Systems** (cont.)

- A number of log-structured file systems in use
- Universal Disk Format (UDF) is used on DVDs and many optical disks (has largely replaced use of ISO 9660)
- Numerous filesystem implementations for flash-based devices are being created (primarily on Linux)
  - Journalling Flash File System (JFFS, JFFS2)
  - Several intended replacements for JFFS/JFFS2, including:
  - LogFS
  - Unsorted Block Image File System (UBIFS)
  - Flash-Friendly File System (F2FS)
  - Yet Another Flash File System (YAFFS)

# Alternatives to Filesystem Journaling (7)

**Copy-On-Write File Systems**

- Copy-on-write technique can also be applied to filesystems to avoid corruption
- Premise:
  - Never modify old data in-place! Always make a copy of it and then change the copy.
  - Perform a single atomic update that changes from using the old data to using the new data
- The filesystem is always moving between valid states using atomic operations
  - After a crash, the filesystem will either be in old state or new state, but nothing in between
- B-tree File System (Btrfs) implemented by Oracle uses copy-on-write
  - The basic B-tree data structure doesn't easily support copy-on-write, so the filesystem uses a modified B-tree implementation
- ZFS (also Oracle) also uses copy-on-write