CSE 511: Operating Systems Design

Lecture 16

Journaling File Systems

File System Consistency

- Historically, file systems were unsafe in the event of failures (e.g., FAT/FAT32)
  - Could lose your data or even metadata due to power loss
  - Need to run a file system checker to recover
- Several approaches can address the problem
  - Soft updates (e.g., BSD's FFS and UFS)
  - **Journaling**
  - Log-structured and copy-on-write (COW) file systems (LFS, ZFS, BtrFS, etc)

Journaling

- A special technique used in database management systems and file systems (FS)
- Every operation (transaction) is written into a ***journal***
  - A special on-disk place (log) containing information allowing during failures to completely negate the effect of the transaction (i.e., ***roll*** it ***back***)

Journaling Guarantees

- ***Atomicity*** of operations, file system ***consistency***, and ***durability*** of changes in the event of failure
- ***Consistency*** is a property of a ***file system*** to remain in correct state (even during failures) after an operation, i.e., FS data structures and *metadata* are not damaged and in unison with each other
- ***Atomicity*** is a property of a file system ***operation*** to either be completed entirely or not completed at all (even during failures)

Journaling Approaches

- What is being logged
  - Metadata only (XFS, JFS, ext4's data=writeback) vs. metadata + data (ext4's data=journal)
  - Complementary methods to guarantee data consistency (ext4's data=ordered)
    - Data is written prior to its metadata being committed to the journal

Journaling Approaches

- How is being logged
  - ***Logical*** journaling (XFS, NTFS): transactions are represented as series of trivial high-level modifications (e.g., changing the name of a file)
  - ***Physical*** journaling (ext4): transactions are represented as series of disk block modifications
    - Very low level
    - Journaling handling procedures are often unaware of what exactly they are doing

Write-Ahead Logging (WAL)

- ***All*** modifications are written into a journal buffer (log) ***prior*** to making ***any*** modifications in the target disk locations
- If failure happens, two scenarios are possible:
  - The journal buffer contains ***all*** modifications made during the last transaction; thus, the transaction can simply be ***replayed*** again
  - The last incomplete transaction is ***discarded*** since the changes were not yet made in the target disk location yet

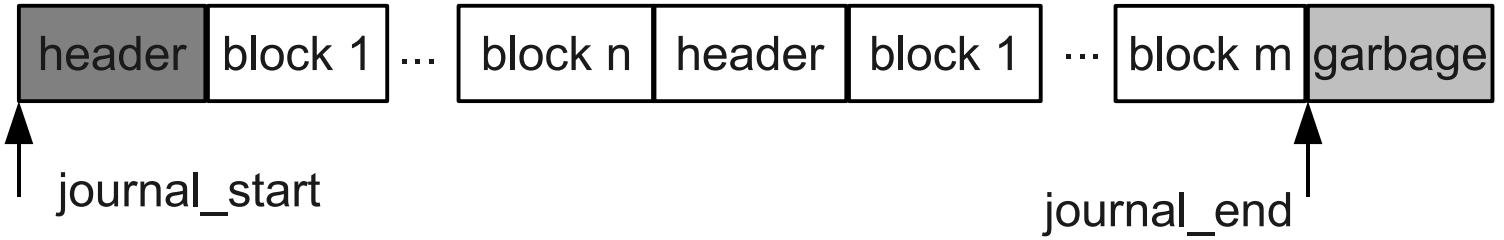
Write-Ahead Logging (WAL)

- It is possible to keep both previous and new versions in the buffer
  - Can completely negate both *successful* and *unsuccessful* transactions
- In practice, only new versions are needed
  - Rarely need to negate *successful* transactions
  - *Batching* is simple, where only the latest version of commonly modified blocks is logged

Example: Circular Journal Buffer

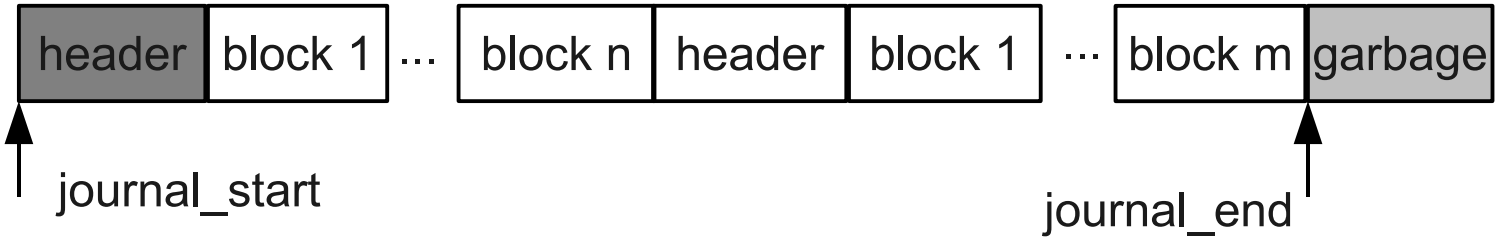
- Physical journaling, wherein FS reserves contiguous space on an FS partition for a *circular journal buffer*
- Each transaction is represented on a disk as a ***header*** followed by a contiguous set of blocks modified during the transaction
  - **journal_start** points to the beginning of the first transaction in the buffer
  - **journal_end** points to the end of the last transaction

Example: Circular Journal Buffer



- Each time a new transaction is ***completely*** written into the buffer, **journal_end** is updated
- Only after journal_end is updated, the FS can modify the blocks at their actual locations
- At some point, the buffer cannot hold additional transactions, and the FS is forced to **flush** all the blocks modified in the very first transaction

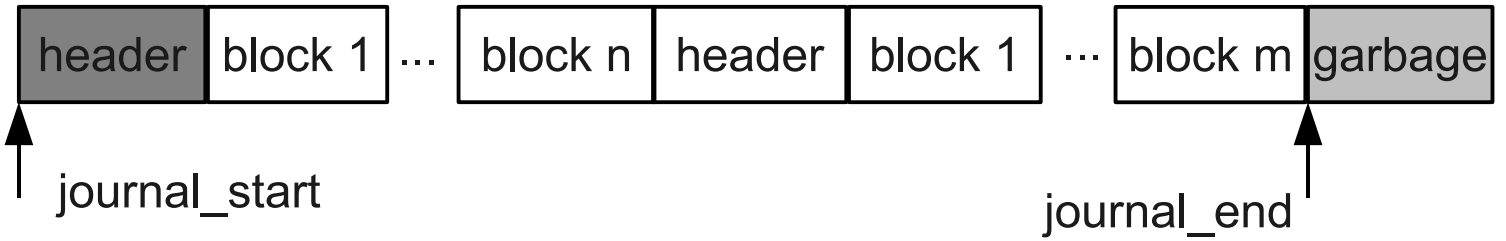
Example: Circular Journal Buffer



- After flushing, the first transaction is deleted from the buffer by incrementing the **journal_start** marker
  - Journal space is reclaimed allowing new transactions to proceed
  - Repeat the same for any following transaction as necessary

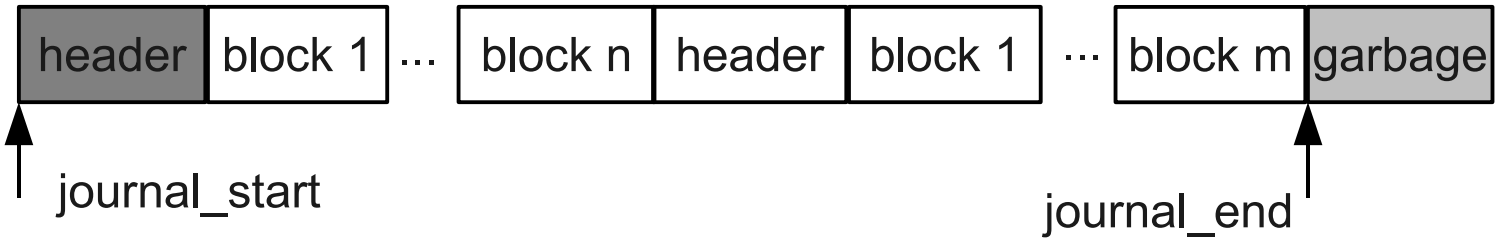
Example: Failure

- The FS super block has the “dirty” flag set
  - Replay the journal when mounting the FS
- Suppose the FS was in the process of writing last transaction, and then the failure occurred
  - Since not all blocks were written into the buffer (marked as “garbage”), **journal_end** has not yet been updated
  - No blocks were written to their actual locations



Example: Failure

- The last transaction is incomplete => discarded
- No need to look at anything prior to **journal_start**
  - All modified blocks from transaction(s) prior to **journal_start** were already flushed
- All or some blocks written between **journal_start** and **journal_end** might not yet be written to their actual locations



Example: Failure

- The FS recovery procedure has to take a block from the log and write it to its actual location
  - Transactions must be processed in order from the earliest to the latest one
    - Very important!
  - Different transactions may modify the same blocks

Example: Failure

- The FS recovery procedure itself can fail
  - No problem since the log is not erased until after ***all*** blocks are written to their target locations
  - Can be replayed as many times as necessary
- While replaying, an earlier transaction T1 may overwrite blocks of a later transaction T2
  - No problem since the journal ***must*** be fully replayed before mounting is even permitted

Transaction Batching

- Allows to merge several small transactions into a bigger one
  - Reduces disk I/O overhead (e.g., the same bitmap block is updated by several transactions)
  - Naturally extends the new-value-only logging
  - Several last transactions can be lost => which is still OK per our consistency and atomicity definitions

Other Challenges

- Disks can still reorder operations internally
  - Was a big deal for ext3!
  - Need to insert I/O “barriers”
- Ext4 enables write I/O barriers by default
  - Their total number can be reduced by storing hash sums