ext3 Journaling File System

"*absolute consistency of the filesystem in every respect after a reboot, with no loss of existing functionality*"

chadd williams

SHRUG

10/05/2001

Contents

- Design Goals
- File System reliability
- What is JFS?
- Why JFS?
- How?
- Details of ext3
- Detail of JFS

Design Goals

- No performance loss
  - Should have a performance gain
- Backwards compatible
- Reliable!

File System reliability

- Preservation
  - Stable data is not affected by a crash
- Predictability
  - Known failure modes
- Atomicity
  - Each operation either fully completes or is fully undone after recovery

What is a JFS?

- Atomically updated
- Old and new versions of data held on disk until the update commits
- Undo logging:
  - Copy old data to the log
  - Write new data to disk
  - If you crash during update, copy old data from log
- Redo logging:
  - Write new data to the log
  - Old data remains on disk
  - If you crash, copy new data from the log

Why JFS?

- Speed recovery time after a crash
  - fsck on a large disk can be very slow
  - ‘to eliminate enormously long filesystem recovery times after a crash’
- With JFS you just reread the journal after a crash, from the last checkpoint

Journal

- Contains three types of data blocks
  - Metadata: entire contents of a single block of filesystem metadata as updated by the transaction
  - Descriptor: describe other journal metadata blocks (where they really live on disk)
  - Header: contain head and tail of the journal, sequence number, the journal is a circular structure

Versus log-structured file system

- A log structured file system ONLY contains a log, everything is written to the end of this log
- LSFS dictates how the data is stored on disk
- JFS does not dictate how the data is stored on disk

How does it work?

- Each disk update is a Transaction (atomic update)
  - Write new data to the disk (journal)
  - The update is not final until a commit
- Only after the commit block is written is the update final
  - The commit block is a single block of data on the disk
  - Not necessarily flushed to disk yet!

How does data get out of the journal?

- After a commit the new data is in the journal – it needs to be written back to its home location on the disk
- Cannot reclaim that journal space until we resync the data to disk

To finish a Commit (checkpoint)

- Close the transaction
  - All subsequent filesystem operations will go into another transaction
- Flush transaction to disk (journal), pin the buffers
- After everything is flushed to the journal, update journal header blocks
- Unpin the buffers in the journal **only after** they have been synced to the disk
- Release space in the journal

How does this help crash recovery?

- Only completed updates have been committed
  - During reboot, the recovery mechanism reapplies the committed transactions in the journal
- The old and updated data are each stored separately, until the commit block is written

ext3 and JFS

- Two separate layers
  - /fs/ext3 – just the filesystem with transactions
  - /fs/jdb – just the journaling stuff (JFS)
- ext3 calls JFS as needed
  - Start/stop transaction
  - Ask for a journal recovery after unclean reboot
- Actually do compound transactions
  - Transactions with multiple updates

ext3 details

- This grew out of ext2
- Exact same code base
- Completely backwards compatible *(if you have a clean reboot)*
  - Set some option bits in the superblock, preventing ext2 mount
  - Unset during a clean unmount

JFS details

- Abstract API based on handles, not transactions
- Handle: “represents one single operation that marks a consistent set of updates on the disk”
  - Compound transactions

JFS details

- API provides start()/stop() pairing to define the operations within a handle
- Allows for nesting of transactions
- There must be enough space in the journal for each update before it starts
  - Lack of space can lead to deadlock
  - start/stop used to make reservations in journal

JFS details

- Also need to make VM reservations
  - Cannot free memory from an in process transaction
  - No transaction aborts – the transaction must complete
  - If you run out of VM you can deadlock
  - Provide call backs to VM so the VM can say, your using too much memory

Guarantees

- Write ordering within a transaction
  - All the updates in a transaction will hit the disk after the commit
  - Updates are still write-behind, so you don't know when the commit will be done
- Write ordering between transactions
  - No formal guarantees
  - Sequential implementation happens to preserve write ordering

Internals

- ext3 & JFS do redo logging
  - New data written to log
  - Old data left on disk
  - After commit new data is moved to disk
- Commit
  - Consists of writing one 512-byte sector to disk
  - Disks can guarantee the write of 1 sector

Internals

- Checkpointing
  - Writing the data from the log out to disk
  - Allows reuse of the log
  - Currently everything is written to disk twice
  - Uses a zero-copy to do the second write
    - new I/O request points to old data buffer

Internals

- What if we need to update data in a pending transaction?
  - Copy-on-write
    - Give a copy of the data buffer to the new transaction
  - Both transactions still get committed in full
    - In order