Journaling File Systems and Solid State Drives

File system consistency

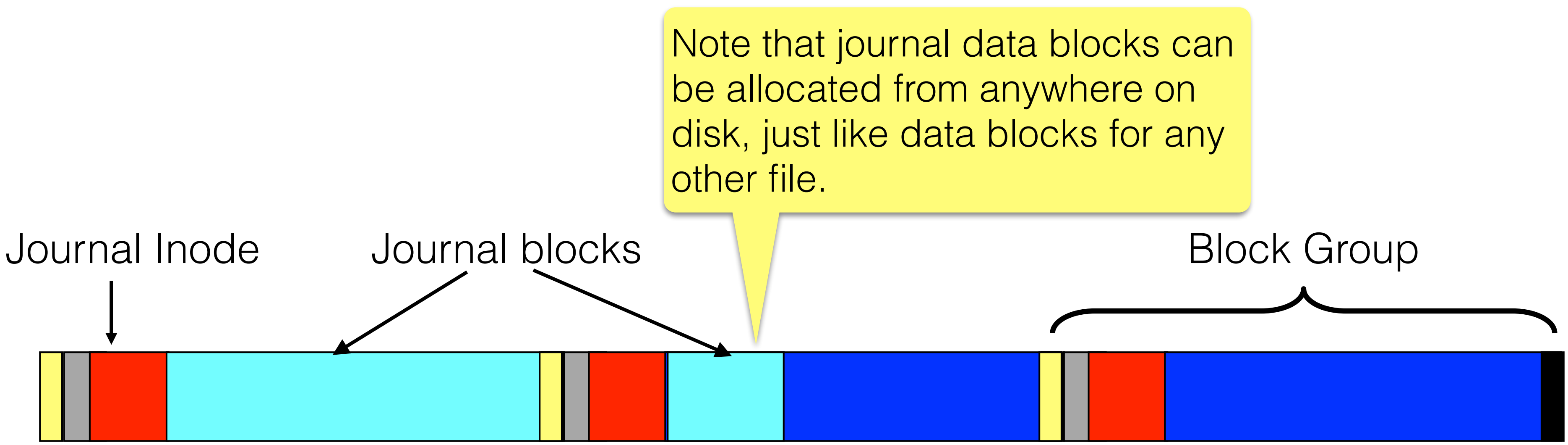
- Do nothing and try to recover in the event of a crash
  - Choose a good order for operations to minimize data loss
  - Most older file systems took this approach (ext2, ffs)
  - fsck
- Treat each file system operation as a transaction
  - roll-back if transaction didn't complete

Alternative solution: Journaling

- Aka Write-Ahead-Logging
- Basic idea:
  - Write a log on disk of the operation you are about to do, before making changes
- If a crash takes place during the *actual write* => go back to journal and retry the actual writes.
  - Don’t need to scan the entire disk, we know what to do!
  - Can recover data as well
- If a crash happens before *journal write* finishes, then it doesn’t matter since the actual write has NOT happened at all, so nothing is inconsistent.

Linux Ext3 File System

- Extends ext2 with journaling capabilities
  - Backwards and forwards compatible
    - Identical on-disk format
  - Journal can be just another large file (inode, indirect blocks, data blocks)



What goes in the “log”

- Transaction structure:
  - Starts with a “transaction begin” (TxBegin) block, containing a transaction ID
  - Followed by blocks with the content to be written
    - Physical logging: log exact physical content
    - Logical logging: log more compact logical representation
  - Ends with a “transaction end” (TxEnd) block, containing the corresponding TID

Journal Entry	TxBegin (TID=1)	Updated inode	Updated Bitmap	Updated Data block	TxEnd (TID=1)

Data Journaling Example

- Say we have a regular update – add 1 data block to a file:
  - Write inode (I[v2]), Bitmap (B[v2]), Data block (Db)
  - Markers for the log (transaction begin/end)

TxBegin	I[v2]	B[v2]	Db	TxEnd

Data Journaling Example

- Say we have a regular update – add 1 data block to a file:
  - Write inode (I[v2]), Bitmap (B[v2]), Data block (Db)
  - Markers for the log (transaction begin/end)

TxBegin	I[v2]	B[v2]	Db	TxEnd

- Sequence of operations

Crash!

1. Write the transaction (containing Iv2, Bv2, Db) to the log
2. Write the blocks (Iv2, Bv2, Db) to the file system
3. Mark the transaction free in the journal

Redo the transaction

Data Journaling Example

- Say we have a regular update – add 1 data block to a file:
  - Write inode (I[v2]), Bitmap (B[v2]), Data block (Db)
  - Markers for the log (transaction begin/end)

TxBegin	I[v2]	B[v2]	Db	TxEnd

- Sequence of operations

1. Write the transaction (containing Iv2, Bv2, Db) to the log
2. Write the blocks (Iv2, Bv2, Db) to the file system
3. Mark the transaction free in the journal

Crash!

Redo the transaction

Data Journaling Example

- Say we have a regular update – add 1 data block to a file:
  - Write inode (I[v2]), Bitmap (B[v2]), Data block (Db)
  - Markers for the log (transaction begin/end)

TxBegin	I[v2]	B[v2]	Db	TxEnd

- Sequence of operations

  1. Write the transaction (containing Iv2, Bv2, Db) to the log

  2. Write the blocks (Iv2, Bv2, Db) to the file system

  3. Mark the transaction free in the journal

Crash!

No problem
Nothing to fix

Data Journaling Example

- Say we have a regular update – add 1 data block to a file:
  - Write inode (I[v2]), Bitmap (B[v2]), Data block (Db)
  - Markers for the log (transaction begin/end)

TxBegin	I[v2]	B[v2]	Db	TxEnd

- Sequence of operations

Crash!

1. Write the transaction (containing Iv2, Bv2, Db) to the log
2. Write the blocks (Iv2, Bv2, Db) to the file system
3. Mark the transaction free in the journal

This one is tricky.
What if only part of the log was written?

Data Journaling Example

- One solution: write each block at a time => Too slow!

- Ideally issue multiple blocks at once.
  - Unsafe though! What could happen?
  - Normal operation: Blocks get written in order, power cuts off before TxEnd gets written => We know transaction is not valid, no problem.

TxBegin	I[v2]	B[v2]	Db	???

  - However, Internal disk scheduling: TxBegin, Iv2, Bv2, TxEnd, Db
  - Disk may lose power before Db written

TxBegin	I[v2]	B[v2]	???	TxEnd

- Problem: Looks like a valid transaction!

Data Journaling Example

To avoid this, split into 2 steps

1. Write all except TxEnd to journal (Journal Write step)

TxBegin	I[v2]	B[v2]	Db

2. Write TxEnd (only once 1. completes) (Journal Commit step)
   => final state is safe!

TxBegin	I[v2]	B[v2]	Db	TxEnd

3. Finally, now that journal entry is safe, write the actual data and metadata to their right locations on the FS (Checkpoint step)

4. Mark transaction as free in journal (Free step)

Journaling: Recovery Summary

- If crash happens before the transaction is committed to the journal
  - Just skip the pending update
- If crash happens during the checkpoint step
  - After reboot, scan the journal and look for committed transactions
  - Replay these transactions
  - After replay, the FS is guaranteed to be consistent
  - Called redo logging

Journal Space Requirements

- How much space do we need for the journal?
  - For every update, we log to the journal => sounds like it's huge!

- After "checkpoint" step, the transaction is not needed anymore because metadata and data made it safely to disk
  - So the space can be freed (free step).

- In practice: circular log

Metadata Journaling

- Recovery is much faster with journaling
  - Replay only a few transactions instead of checking the whole disk

- However, normal operations are slower
  - Every update must write to the journal first, then do the update
    - Writing time is at least doubled
  - Journal writing may break sequential writing. Why?
    - Jump back-and-forth between writes to journal and writes to main region
  - Metadata journaling is similar, except we only write FS metadata (no actual data) to the journal:

Journal Entry

TxBegin	I[v2]	B[v2]	TxEnd

Metadata Journaling

- What can happen now?
  - Say we write data after checkpointing metadata
  - If crash occurs before all data is written, inodes will point to garbage data!
  - How do we take care of this?

- Write data BEFORE writing metadata to journal!
  1. Write data, wait until it completes
  2. Metadata journal write
  3. Metadata journal commit
  4. 4. Checkpoint metadata
  5. Free

- If write data fails => as if nothing happened, sort of (from the FS's point of view)!

- If write metadata fails => same!

Summary: Journaling

- Journaling ensures file system consistency
- Complexity is in the size of the journal, not the size of the disk!
- Is fsck useless then?
- Metadata journaling is the most commonly used
  - Reduces the amount of traffic to the journal, and provides reasonable consistency guarantees at the same time.
- Widely adopted in most modern file systems (ext3, ext4, ReiserFS, JFS, XFS, NTFS, etc.)

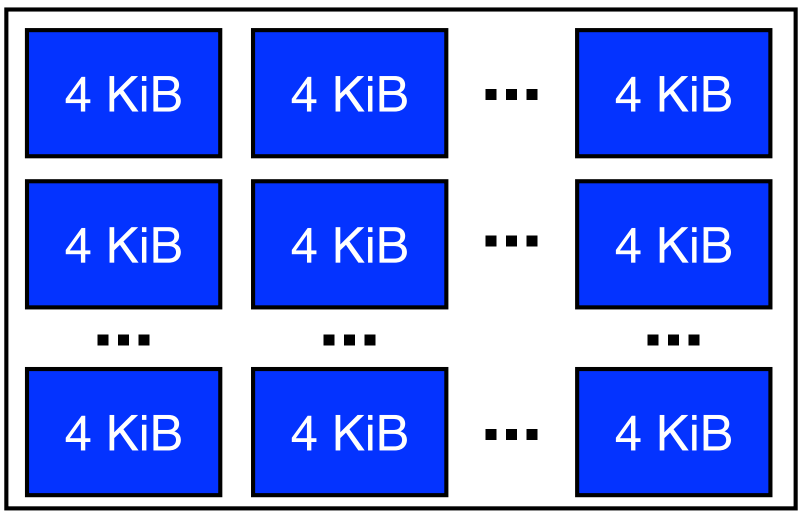
Solid State Disks (SSD)

- Replace rotating mechanical disks with non-volatile memory
  - Battery-backed RAM
  - NAND flash
- Advantages: faster
- Disadvantages:
  - Expensive
  - Wear-out (flash-based)
- NAND flash storage technology
  - Read / write / erase! operations

SSD Characteristics

- Data cannot be modified “in place”
  - No overwrite without erase

- Terminology:
  - Page (unit of read/write), block (unit of erase operation)



Data written in 4KB pages

Data erased in blocks of typically >= 128 pages

- Uniform random access performance!
  - Disks typically have multiple channels so data can be split (striped) across blocks, speeding access time

Writing

- Consider updating a file system block (e.g. a bitmap allocation block in ext2 file system)
    - Find the block containing the target page
    - Read all active pages in the block into controller memory
    - Update target page with new data in controller memory
    - Erase the block (high voltage to set all bits to 1)
    - Write entire block to drive

- Some FS blocks are frequently updated
    - And SSD blocks wear out (limited erase cycles)

SSD Algorithms

- Wear levelling
  - Always write to new location
  - Keep a map from logical FS block number to current SSD block and page location
  - Where does it store the map?
  - Old versions of logically overwritten pages are “stale”
- Garbage collection
  - Reclaiming stale pages and creating empty erased blocks
- RAID 5 (with parity checking) striping across I/O channels to multiple NAND chips

File Systems and SSDs

- Typically, same FSs as for hard disk drives
  - ext4, Btrfs, XFS, JFS and F2FS support SSDs
- No need for the FS to take care of wear-leveling
  - Done internally by the SSD
  - But the TRIM operation is used to tell the SSD which blocks are no longer in use. (Otherwise a delete operation doesn't go to disk)
- Some flash file systems (F2FS, JFFS2) help reduce write amplification (esp. for small updates – e.g., FS metadata)
- Other typical HDD features – do we want these?
  - Defragmentation
  - Disk scheduling algorithms

Summary: File System Goals

- Efficiently translate file name into file number using a directory
- Sequential file access performance
- Efficient random access to any file block
- Efficient support for small files (overhead in terms of space and access time)
- Support large files
- Efficient metadata storage and lookup
- Crash recovery

Summary: File System Components

- Index structure to locate each block of a file

- Free space management

- Locality heuristics

- Crash recovery