# CSC 369

Week 11:

Example File Systems

Reliability and Write Optimizations

University of Toronto, Department of Computer Science

# Overview

- Last time:
  - Optimizations: caching, read-ahead
  - Disk characteristics, optimizations
  - Disk-aware allocation, I/O scheduling

- This week:
  - Example file systems
  - Crash consistency and recovery
  - Optimizing writes
  - Log-structured file systems
  - VFS
  - Solid State Drives (SSDs)

# Example File Systems

# FS Comparison

| | FAT | FFS | NTFS |
|---|---|---|---|
| Index structure | Linked list | Tree (inodes) | Tree (extents) |
| Index structure granularity | Block | Block | Extent |
| Free space management | FAT array | Bitmap | Bitmap |
| Locality heuristics | Defragmentation | Block groups, reserved space | Best fit Defragmentation |

*Table Source: Operating Systems, Anderson & Dahlin, p 554*

# FAT

- File Allocation Table
  - Late 70s
  - Blocks allocated in a linked index structure

FAT

| | |
|---|---|
| **0** | |
| 1 | |
| 2 | |
| 3 | nil |
| 4 | |
| 5 | 6 |
| 6 | 7 |
| 7 | 3 |

Directory

| | |
|---|---|
| **afile** | **5** |
| | |
| | |
| | |
| | |

# FAT

- Directories map file name to first block of file
- FAT stores both the linked list of blocks belonging to files and the free blocks

- Limitations
  - Poor random access
  - Poor locality
  - Limited file metadata and access control

# Ext2, Ext3, Ext4

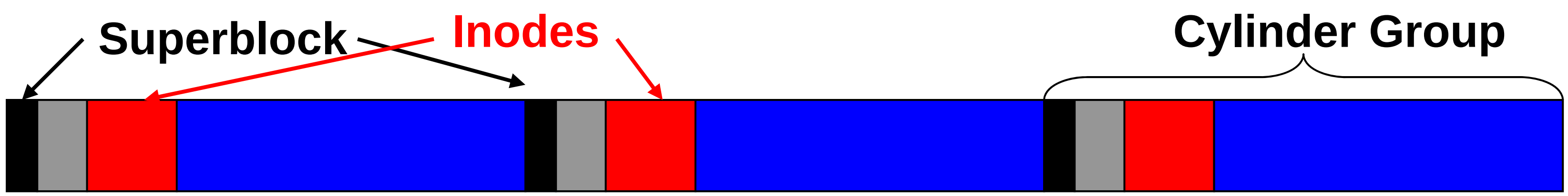
- Linux file system evolution
- Ext2 originally borrowed heavily from FFS
- Recall: Reduce seeks for faster reads, etc.
- More details on reliability and optimizations for writes

# Reliability and Write Optimizations

- How do we guarantee consistency of on-disk storage?
- How do we handle OS crashes and disk errors?
- How do we optimize writes?

# FFS: Consistency Issues - Overview

- Inodes: fixed size structure stored in cylinder groups



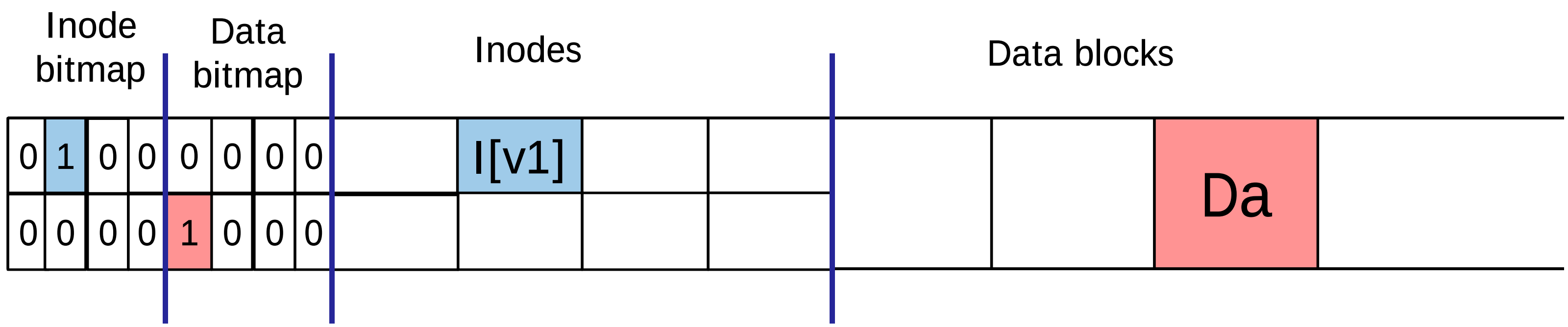

- Metadata updates must be synchronous operations. Why?
- File system operations affect multiple metadata blocks
  - Write newly allocated inode to disk before its name is entered in a directory.
  - Remove a directory name before the inode is deallocated
  - Deallocate an inode (mark as free in bitmap) before that file's data blocks are placed into the cylinder group free list.

# FFS Observation 1: Crash recovery

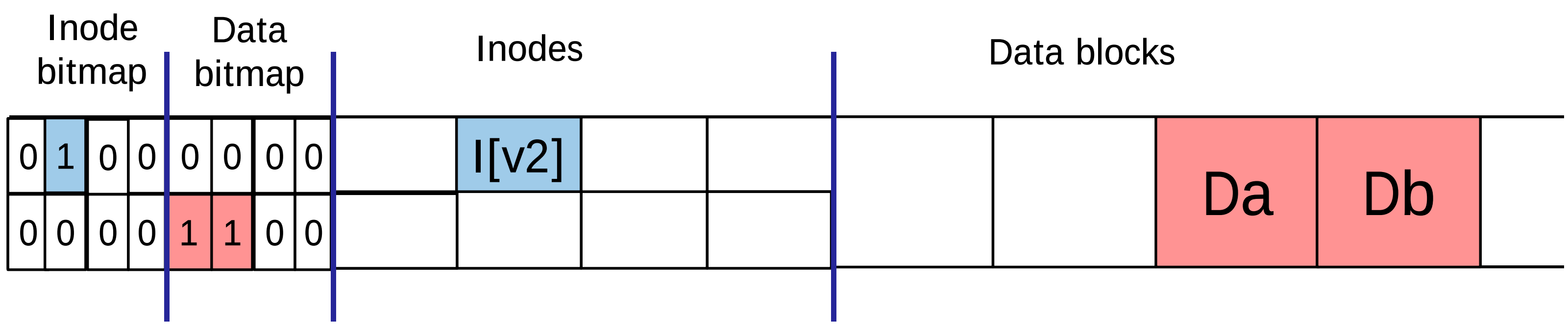
- If the OS crashes in between any of these synchronous operations, then the file system is in an inconsistent state.
- Solutions (overview):
  - *fsck* – post-crash recovery process to scan file system structure and restore consistency
    - All data blocks pointed to by inodes (and indirect blocks) must be marked allocated in the data bitmap
    - All allocated inodes must be in some dir entries
    - Inode link count must match
  - Log updates to enable roll-back or roll-forward

# Example: update

- Consider a simple update: append 1 data block to a file
- Assume a similar FS structure as seen before:




- Add a data block: Db .. What changes?




- Three writes: I[v2], B[v2], Db

# Crash consistency

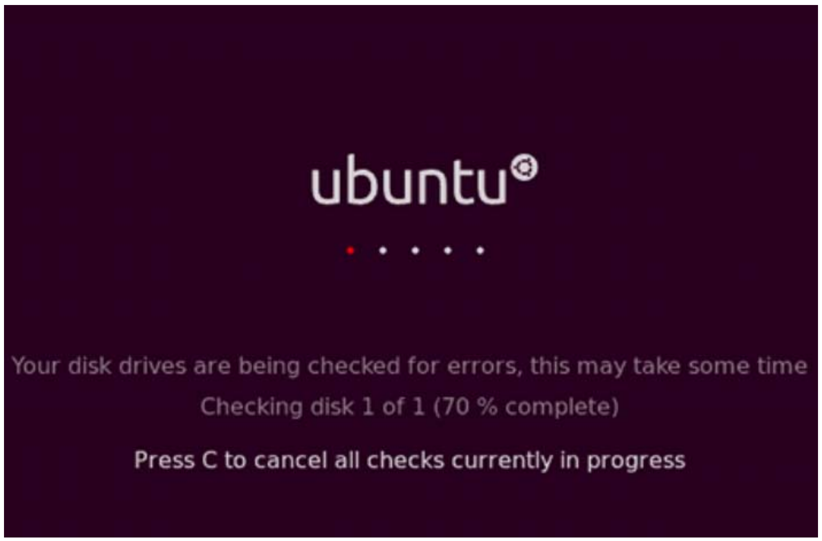
- What if only one write succeeds before a crash?
  - 1. Just Db write succeeds.
    - No inode, no bitmap => as if the write did not occur
    - FS not inconsistent, but data is lost!
  - 2. Just I[v2] write succeeds.
    - No data block => will read garbage data from disk.
    - No bitmap entry, but inode has a pointer to Db => FS inconsistency!
  - 3. Just B[v2] write succeeds.
    - Bitmap says Db is allocated, inode has no pointer to it => again, FS inconsistent + Db can never be used again

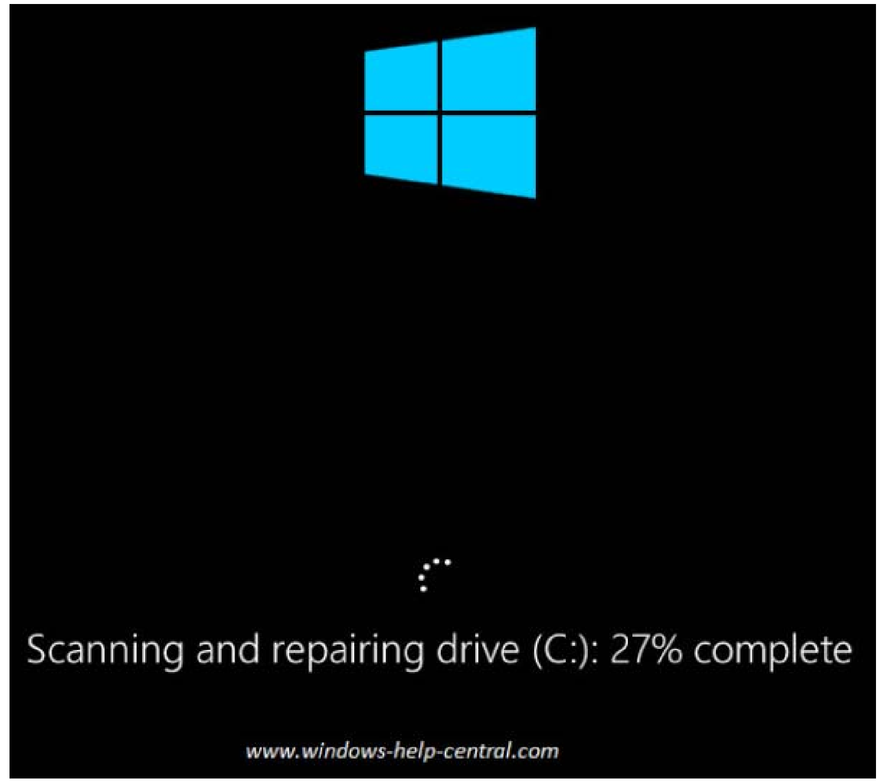
# Other crash scenarios

- What if only two writes succeed before a crash?
  - 1. Only I[v2] and B[v2] writes succeed.
    - Inode and bitmap agree => FS metadata is consistent
    - However, Db contains garbage.
  - 2. Only I[v2] and Db writes succeed.
    - Inode points to correct data, but clashes with B[v1] => FS inconsistency, must fix this!
  - 3. Only B[v2] and Db writes succeed.
    - Again, inode and bitmap info does not match
    - Even though Db was written, no inode points to it (which file is it part of? No idea!)

# Crash consistency problem

Solution #1: fsck




Similar tools exist on various systems

# Solution: fsck

- fsck: UNIX tool for finding inconsistencies and repairing them
- Cannot fix all problems!
  - When Db is garbage – cannot know that's the case
  - Only cares that FS metadata is consistent!

# Solution: fsck

- What does it check?
  - 1. Superblock: sanity checks
    - Use another superblock copy if suspected corruption
  - 2. Free blocks: scan inodes (incl. all indirect blocks), build bitmap
    - inodes / data bitmaps inconsistency => resolve by trusting inodes
    - Ensure inodes in use are marked in inode bitmaps
  - 3. Inode state: check inode fields for possible corruption
    - e.g., must have a valid “mode” field (file, dir, link, etc.)
    - If cannot fix => remove inode and update inode bitmap
  - 4. Inode links: verify links# for each inode
    - Traverse directory tree, compute expected links#, fix if needed
    - If inode discovered, but no dir refers to it => move to “lost+found”

# Solution: fsck

- 5. Duplicates: check if two different inodes refer to same block
  - Clear one if obviously bad, or, give each inode its own copy of block
- 6. Bad blocks: bad pointers (outside of valid range)
  - Just remove the pointer from the inode or indirect block
- 7. Directory checks: integrity of directory structure
  - E.g., make sure that “.” and “..” are the first entries, each inode in a directory entry is allocated, no directory is linked more than once

# fsck limitations

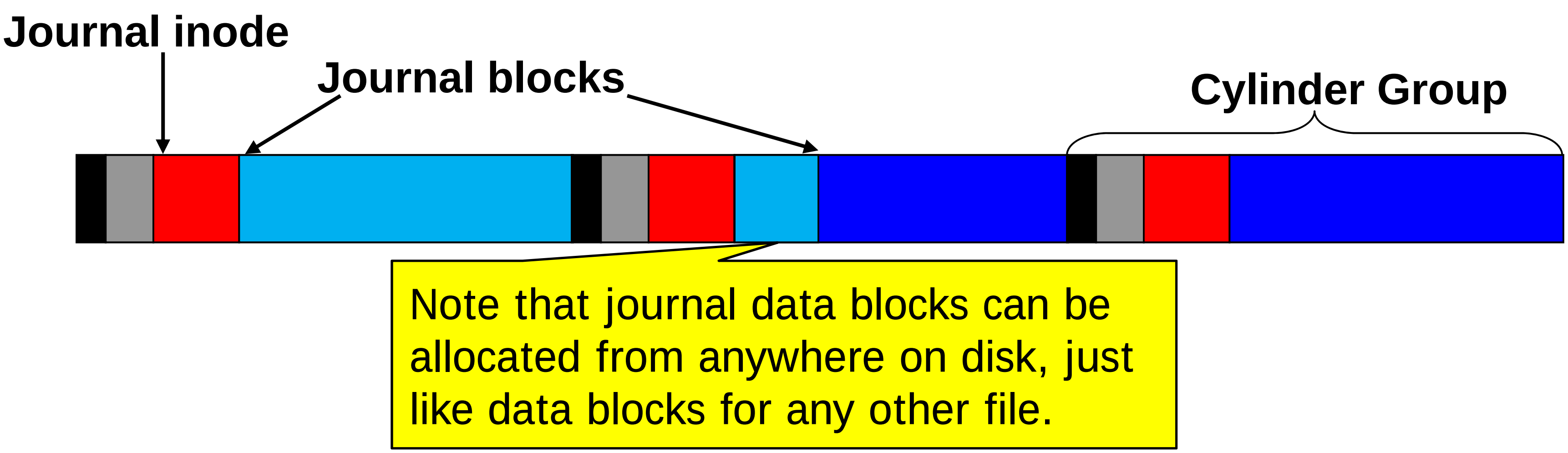
- So, fsck helps ensure integrity
- Only FS integrity, cannot do anything about lost data!
- Bigger problem: too slow!
  - Disks are very large nowadays – scanning all this could take hours!
  - Even for small inconsistency, must scan whole disk!

# Alternative solution: Journaling

- Aka Write-Ahead-Logging
- Basic idea:
  - When doing an update, before overwriting structures, first write down a little note (elsewhere on disk) saying what you plan to do.
  - i.e., “Log” the operations you are about to do.
- 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 that “note”

- 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
  - 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 may happen at any point!
  - If between 1 and 2 => on reboot, replay non-free transactions (called redo logging)
  - If during writes to the journal (step 1) => tricky!

# Data Journaling Example

- One solution: write each block at a time
  - 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 (TID=1) | Updated inode | Updated Bitmap | TxEnd (TID=1) |
|---|---|---|---|---|

# 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. 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.)

# Ext3 final notes

- Lacks modern FS features (e.g., extents)
  - For recoverability, this may actually be an advantage
  - FS metadata is in fixed, well-known locations, and data structures have redundancy
  - When faced with significant data corruption, ext2/3 may be recoverable when a tree-based file-system may not

- Next up: Log-structured file systems