Jin-Soo Kim
(jinsoo.kim@snu.ac.kr)

Systems Software &
Architecture Lab.

Seoul National University

Spring 2019

# File Systems

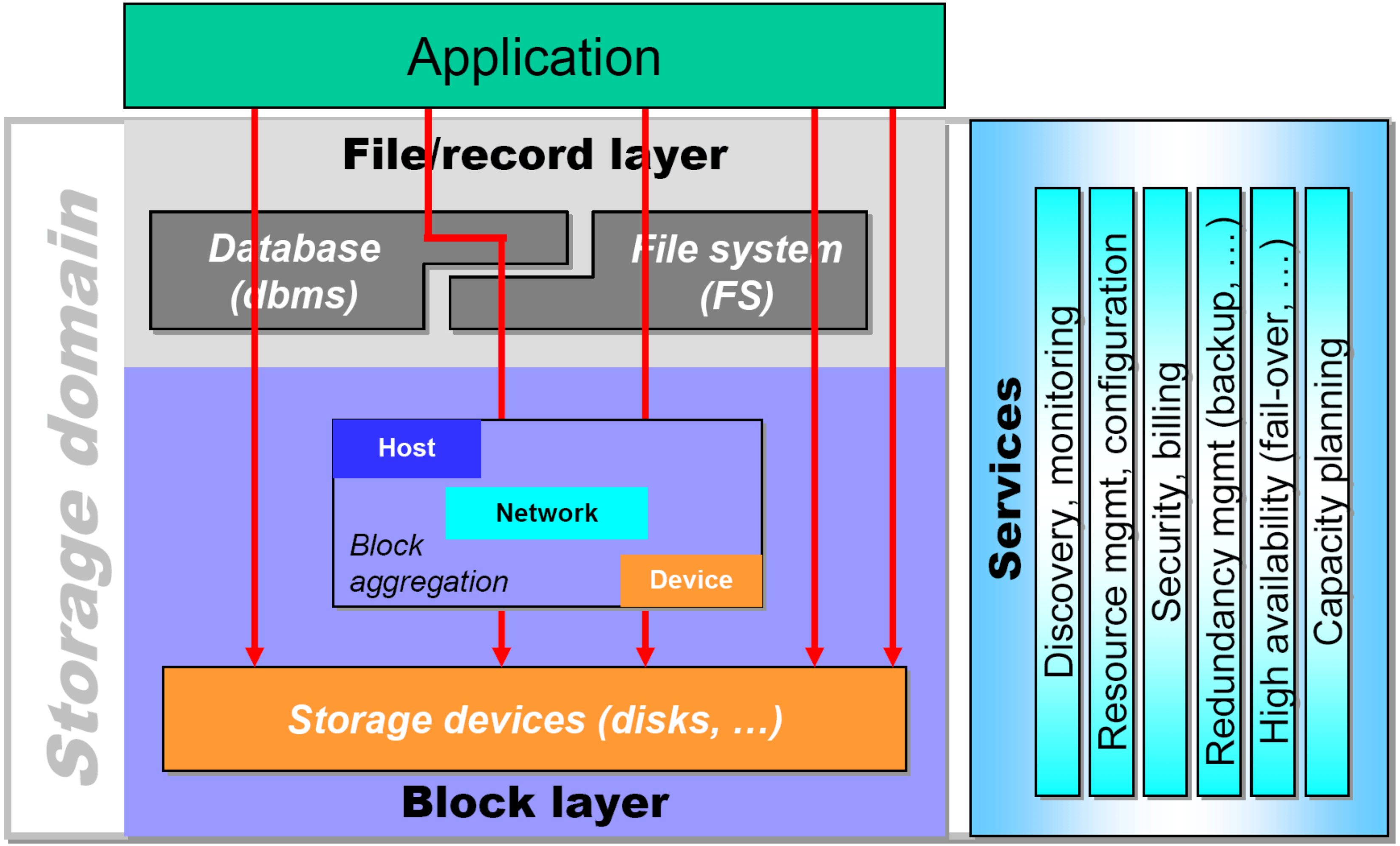
# Storage Abstraction (Revisited)

- Abstraction given by block device drivers:


- Operations
  - Identify(): returns N
  - Read (start sector #, # of sectors, buffer address)
  - Write (start sector #, # of sectors, buffer address)

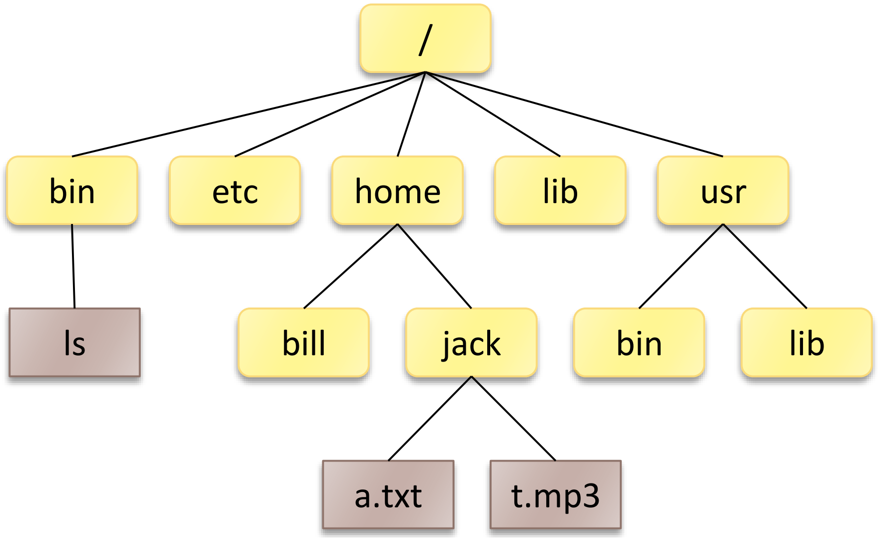
*Source: Sang Lyul Min (Seoul National University)*

# SNIA Shared Storage Model



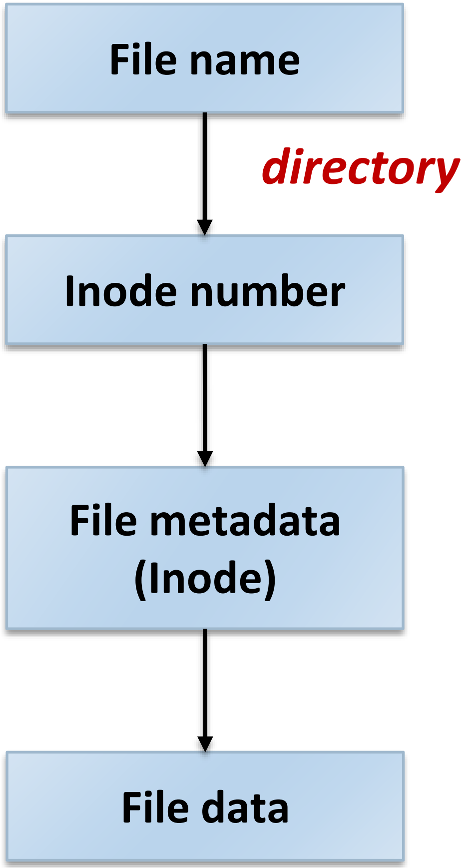
# Abstractions for Storage

- File
  - A named collection of related information that is recorded on persistent storage
  - Each file has an associated inode number (internal file ID)
  - Inodes are unique (at a given time) within a file system
- Directory
  - A logical group of files
  - Hierarchical directory tree: directories can be placed within other directories
  - Implemented as a special file: a list of <file name, inode number>



# Representing Files

- For each file, we have
  - File contents (data)
    - A sequence of bytes
    - File systems normally do not care what they are
  - File attributes (metadata or inode)
    - File size
    - Owner
    - Access control lists
    - Timestamps
    - Block locations, …
  - File name
    - The full pathname from the root specifies a file
    - e.g. `open("/etc/passwd", O_RDONLY);`



# File Interfaces

- POSIX operations

| | |
|---|---|
| `open` | Create a file or open an existing file |
| `close` | Close a file |
| `read` | Read data from a file |
| `write` | Write data to a file |
| `lseek` | Reposition read/write file offset |
| `stat` | Get file status |
| `fsync` | Synchronize a file's in-core state with storage device |
| `link` | Make a new name for a file |
| `unlink` | Delete a name and possibly the file it refers to |
| `rename` | Change the name or location of a file |
| `chown` | Change ownership of a file |
| `chmod` | Change permissions of a file |
| `flock` | Apply or remove an advisory lock on an open file |

# POSIX Inode

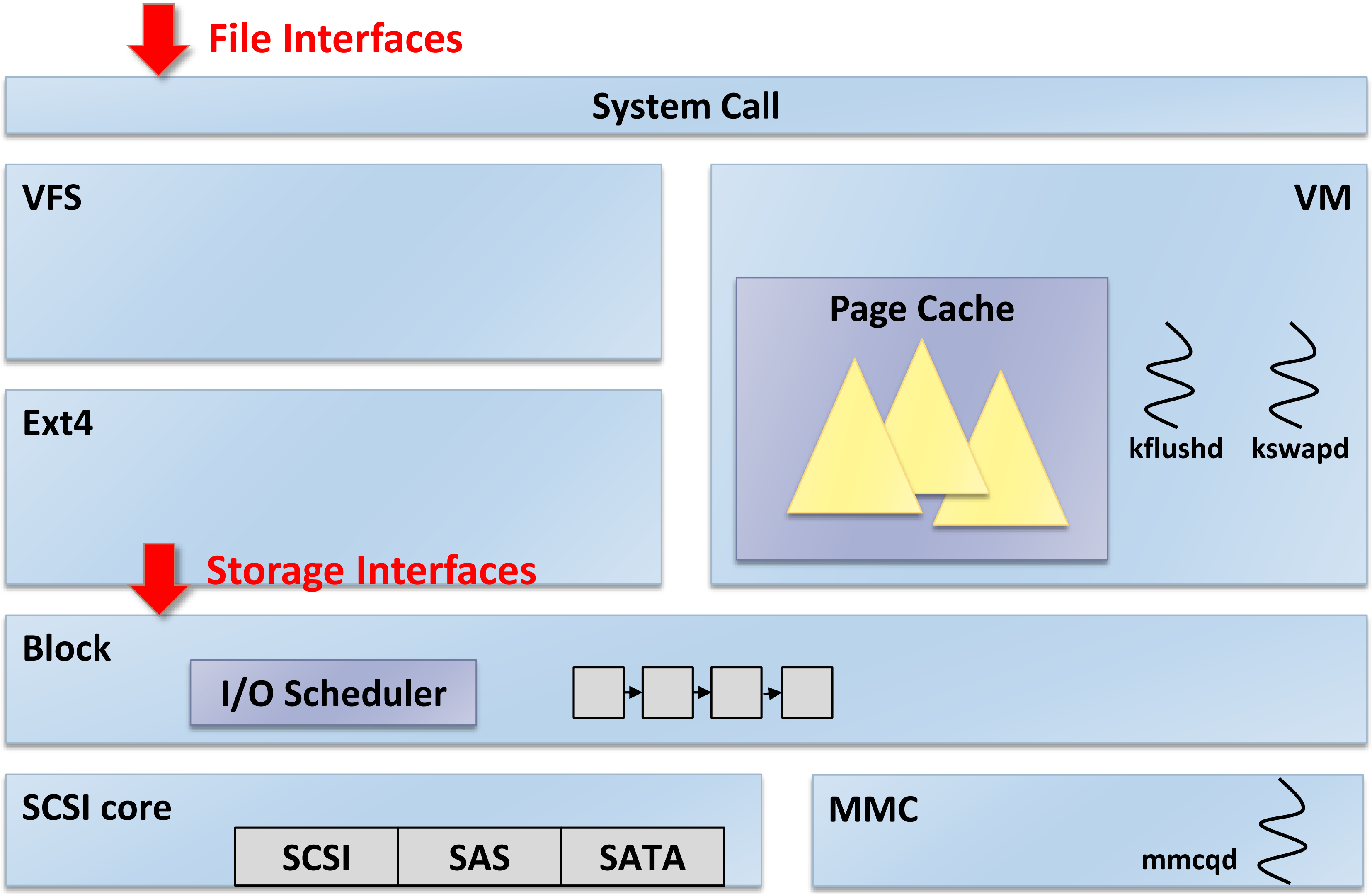
- Inode number
- File type: regular, directory, char/block dev, fifo, symbolic link, …
- Device ID containing the file
- User ID and group ID of the owner
- Access permission: *rwx* for owner(*u*), group(*g*), and others(*o*)
- Number of hard links
- File size in bytes
- Number of 512B blocks allocated
- Timestamps: time of last access (*atime*), time of last modification (*mtime*), time of last status change (*ctime*)

# Ensuring Persistence

- File system buffers writes into memory ("page cache")
  - Write buffering improves performance
  - Up to 30 seconds in Linux
  - `sync()` causes all pending modifications to filesystem metadata and cached file data to be written to the underlying filesystem
  - `fsync()` flushes all dirty data to disk, and metadata associated with the file and tells disk to flush its write cache to the media too
  - `fdatasync()` does not flush modified metadata

```
int fd = open("foo", O_CREAT | O_WRONLY | O_TRUNC);
int rc = write(fd, buffer, size);
rc = fsync(fd);
close(fd);
```

# The Big Picture

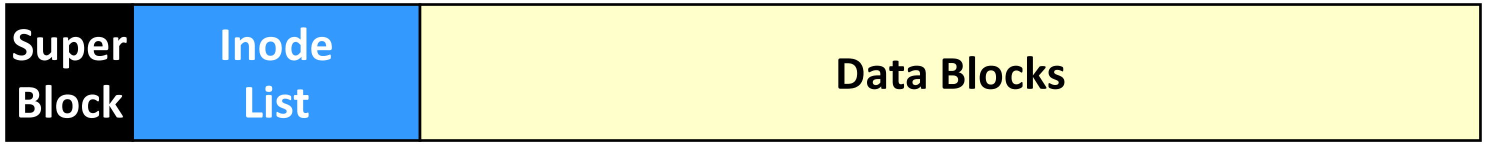


# A Fast File System for UNIX

(M. McKusick et al., ACM TOCS, 1984)

# The Original Unix FS

- First Unix file system developed by Ken Thompson

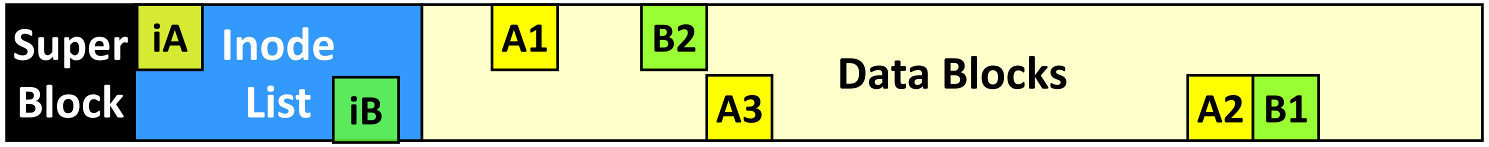



- Super block
  - Basic information of the file system
  - Head of freelists of Inodes and data blocks
- Inode list
  - Referenced by index into the inode list
  - All inodes are the same size
- Data blocks
  - A data block belongs to only one file

# FFS

- The original Unix file system (70's) was very simple and straightforwardly implemented
  - But, achieved only 2% of the maximum disk bandwidth
- BSD Unix folks redesigned file system called FFS
  - McKusick, Joy, Leffler, and Fabry (80's)
  - Keep the same interface, but change the internal implementation
- The basic idea is disk-awareness
  - Place related things on nearby cylinders to reduce seeks
  - Improved disk utilization, decreased response time

# Unix FS: Problems




- Files are fragmented as the file system "ages"
  - Blocks are allocated randomly over the disk
- Inodes are allocated far from blocks
  - Traversing pathnames or manipulating files and directories requires long seeks between inodes and data blocks
- Files in a directory are typically not allocated in consecutive inode slots
- The small block size: 512 bytes

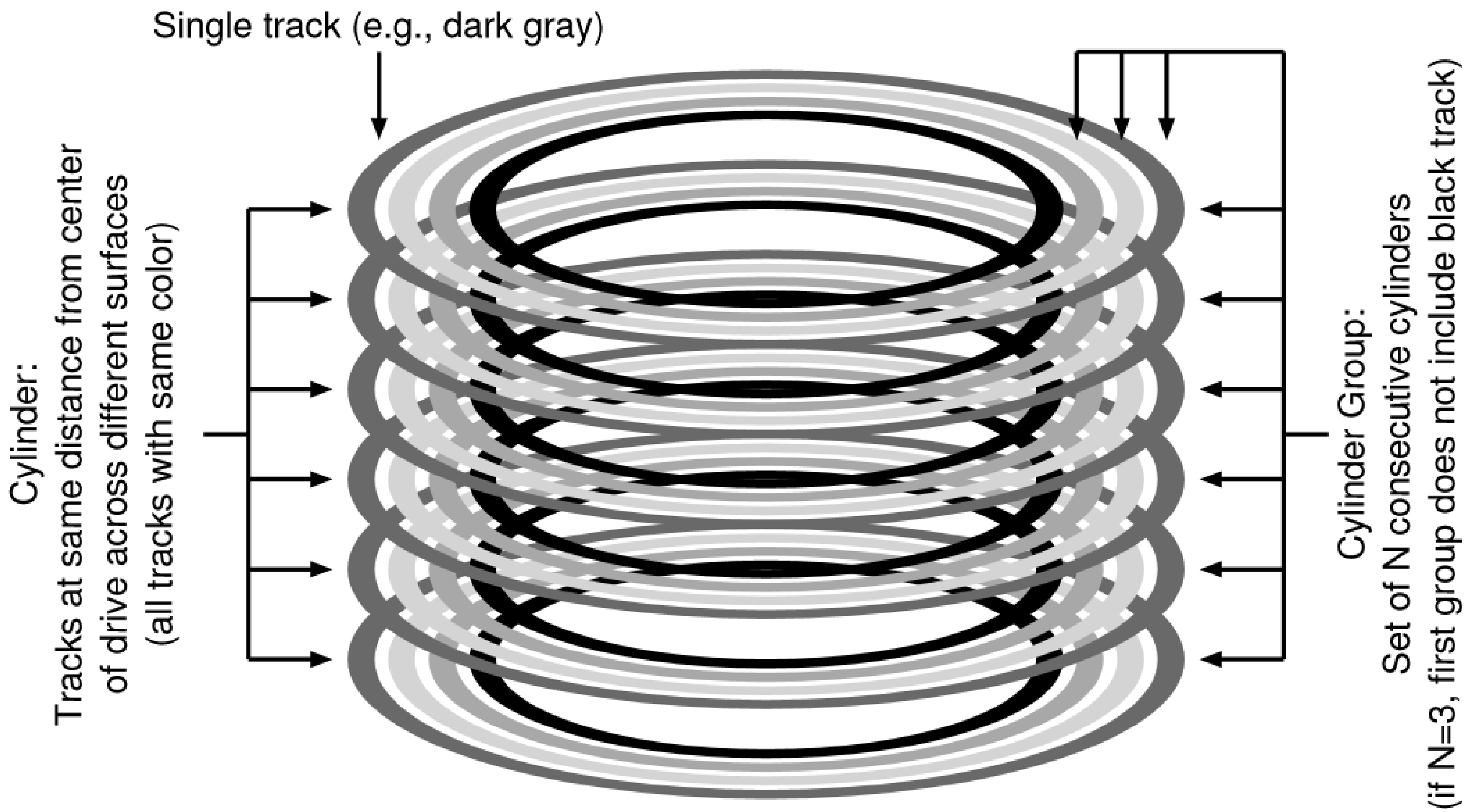
# Bitmaps

- Use bitmaps instead of free lists


  - Each bit represents whether the corresponding inode (or data block) is free or in use
  - Provides better speed, with more global view
  - Faster to find contiguous free blocks
  - Helps to reduce file fragmentation

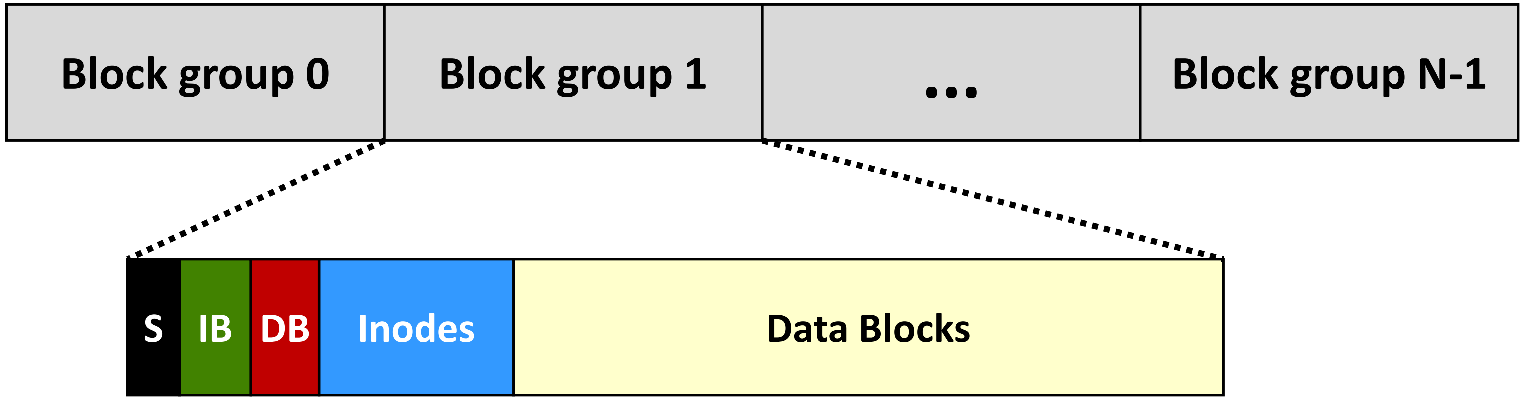
# Cylinder Groups

- Divides the disk into a number of cylinder groups



# On-Disk Layout

- Put all the structures within each cylinder group
  - Modern drives do not export disk geometry information
  - Modern file systems organize the drive into "block groups" (e.g. Linux Ext2/3/4)
  - Block size is increased to 4KB to improve throughput
  - Super block (S) is replicated for reliability reasons

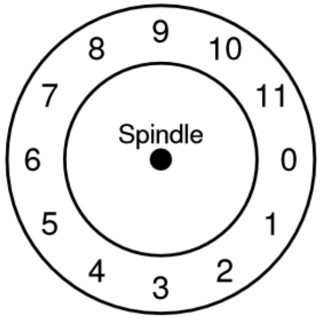


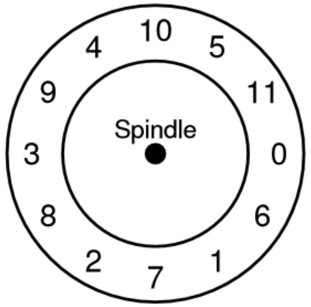
# Allocation Policies

- Keep related stuff together
- Balance directories across groups
  - Allocate directory blocks and its inode in the cylinder group with a low number of allocated directories and a high number of free inodes
- Files in a directory are often accessed together
  - Place all files that are in the same directory in the cylinder group of the directory
  - Allocate data blocks of a file in the same group as its inode
  - Data blocks of a large file are partitioned into chunks and distributed over multiple cylinder groups

# Other Features

- Fragments to reduce internal fragmentation
  - Each block can be broken optionally into 2, 4, or 8 fragments
  - The block map manages the space at the fragment level
- File system parameterization
  - Make the next block come into position under the disk head by skipping some blocks




- Free space reserve
- Long file names
- Atomic rename
- Symbolic links

# Summary

- First disk-aware file system
  - Cylinder groups
  - Bitmaps
  - Replicated superblocks
  - Large blocks
  - Smart allocation policies
- FFS achieves 14% ~ 47% of the disk bandwidth
  - The throughput deteriorates to about half when the file systems are full
- FFS inspired modern file systems including Ext2/3/4

# Ext4 File System

# Ext2/3/4

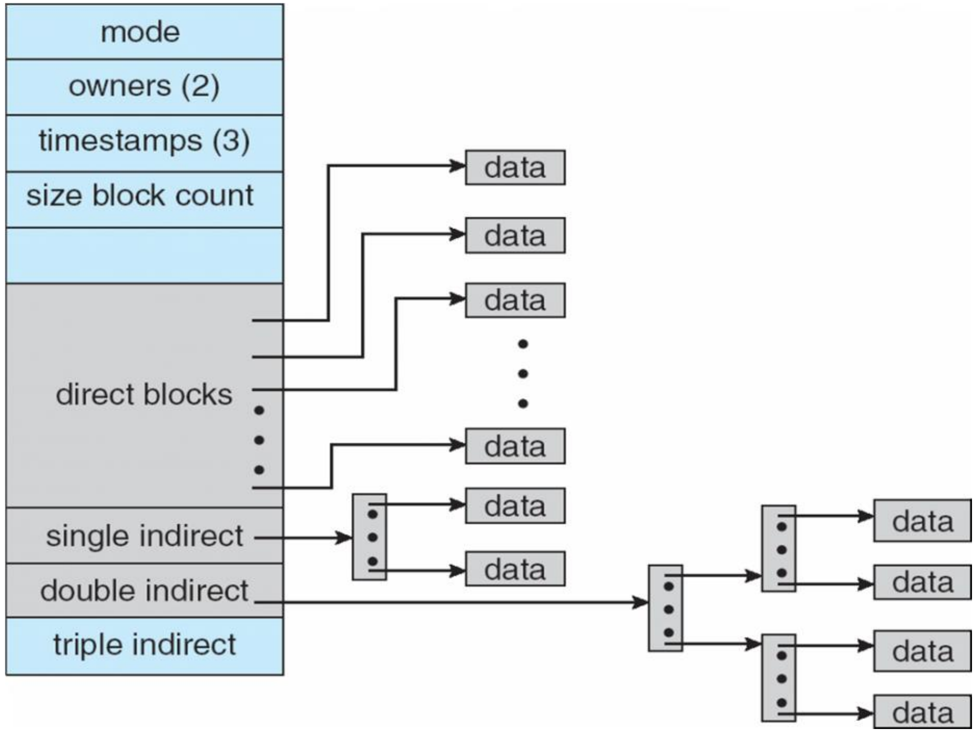
- Evolved from Minix filesystem
  - Maximum file size: 64MB (16-bit block addresses)
  - Directory: fixed-size entries, file name up to 14 chars
- Virtual file system (VFS) added
- Extended filesystem (Ext), Linux 0.96c, 1992
- Ext2, Linux 0.99.7, 1993
- Ext3, Linux 2.4.15, 2001
- Ext4, Linux 2.6.19, 2006

# Ext4 Features

- Scalability
  - Support volume sizes up to 1EB
  - Support file sizes up to 16TB
- Extents-based mapping
- Flex block group
- Delayed allocation
- Multi-block allocator
- Directory indexing with Htree (since Ext3)
- Journaling for file system consistency (since Ext3)

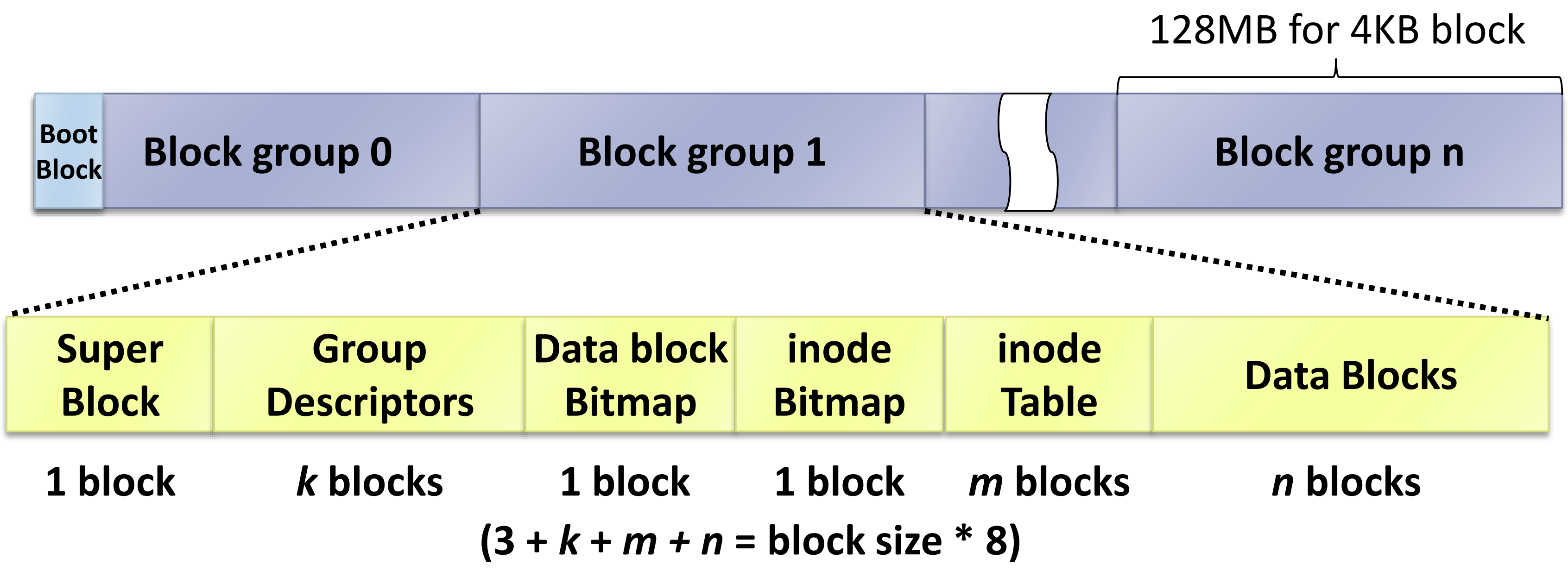
# Ext4 Inode

- File metadata (256 bytes/inode by default)
- Pointers for data blocks or extents



# Ext4 On-disk Layout

- Block group
  - Similar to the cylinder group in FFS
  - All the block groups have the same size and are stored sequentially



# Ext4 Block Group

- Superblock: file system metadata
  - Total number of inodes
  - File system size in blocks
  - Free blocks / inodes counter
  - Number of blocks / inodes per group
  - Block size, ...
- Group descriptor
  - Number of free blocks / inodes / directores
  - Block number of block / inode bitmap, etc.
- Both superblock and group descriptor are duplicated in other block groups

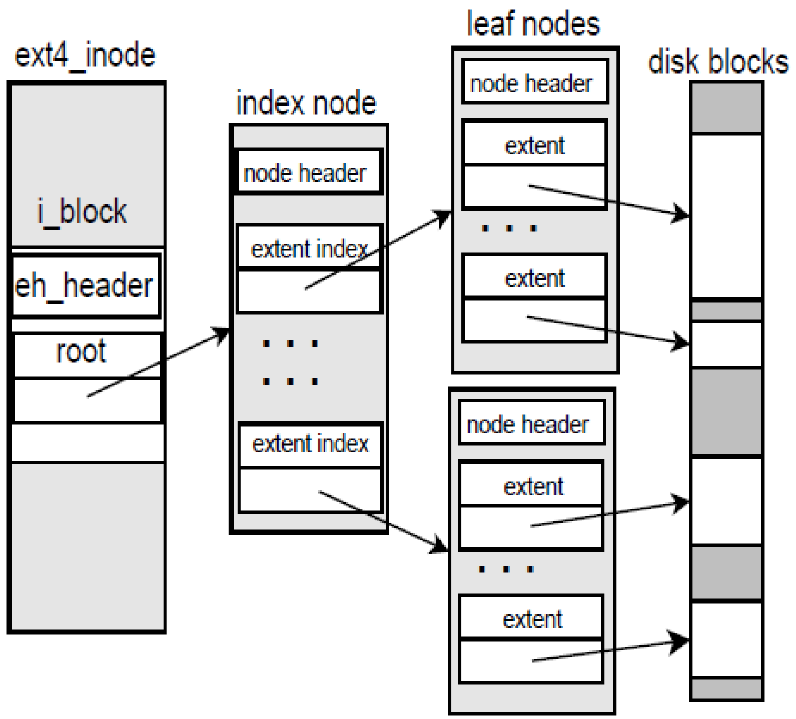
# Ext4 Extents

- Extent <offset, length, physical block>:
  A single descriptor for a range of contiguous blocks
  - 32-bit logical block number: file size up to 16TB
  - 48-bit physical block number: up to 1EB filesystem
  - 15-bit length: Max 128MB contiguous blocks
- An efficient way to represent large files
- Prevent file fragmentation
- Less metadata information to change on file deletion

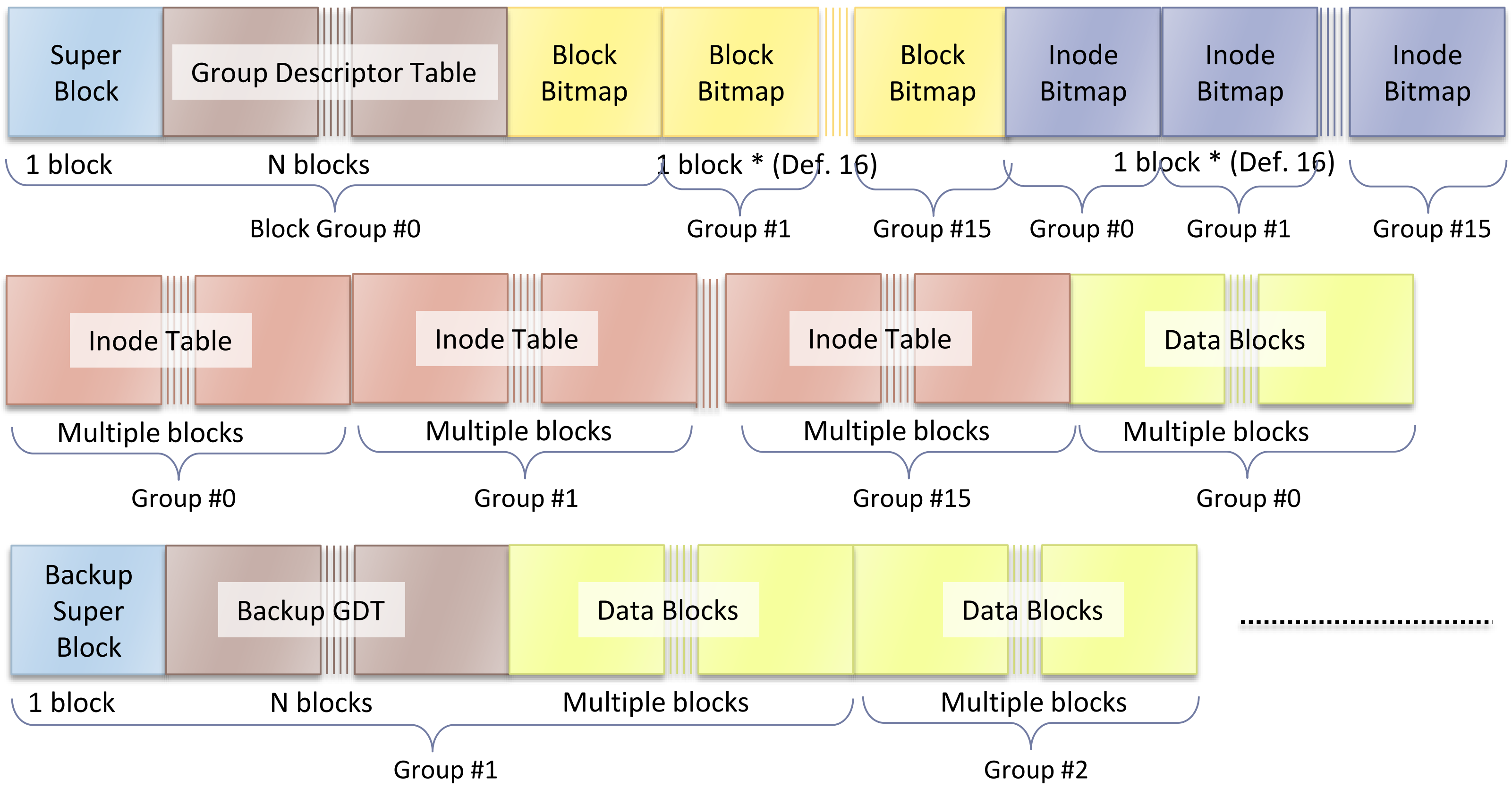
# Ext4 Extents Tree

- Up to four extents in the inode. Otherwise, extents tree is used.

- Extent header
  - # valid entries
  - # entries / node
  - Tree depth
  - Magic number



# Ext4 Flex Block Groups



# Ext4 Delayed Allocation

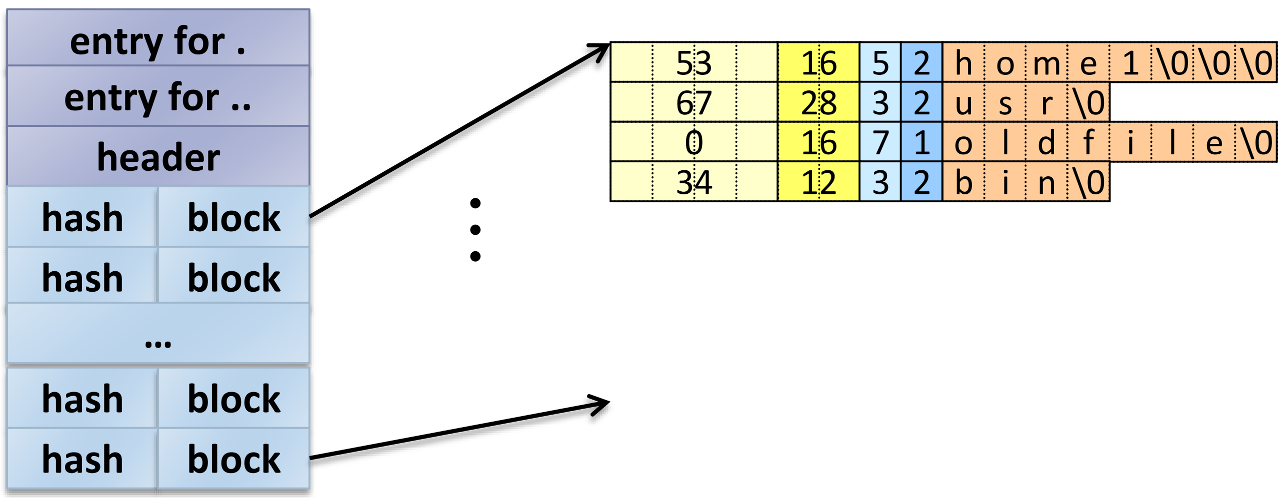
- Blocks allocations postponed to page flush time, rather than during the `write()` operation
  - Provides the opportunity to combine many block allocation requests into a single request
  - Reduce possible fragmentation and save CPU cycles
  - Avoid unnecessary block allocation for short-lived files

# Ext4 Multi-block Allocator

- Ext3 allocates one block at a time
  → Inefficient for larger I/Os
- An entire extent, containing multiple contiguous blocks, is allocated at once
  - Reduce fragmentation
  - Reduce extent metadata
  - Eliminate multiple calls and reduce CPU utilization
- Stripe size aligned allocations
- Pack small files together and avoid fragmentation of free space ("per-cpu locality group")

# Ext4 Directory Indexing

- Htree-based directory
  - 32-bit hashes for keys
  - Each key refers to a range of entries in a leaf block
  - High fanout factor (over 500 for 4KB block)
  - Constant depth (one or two levels)
  - Leaf blocks are identical to old-style directory blocks



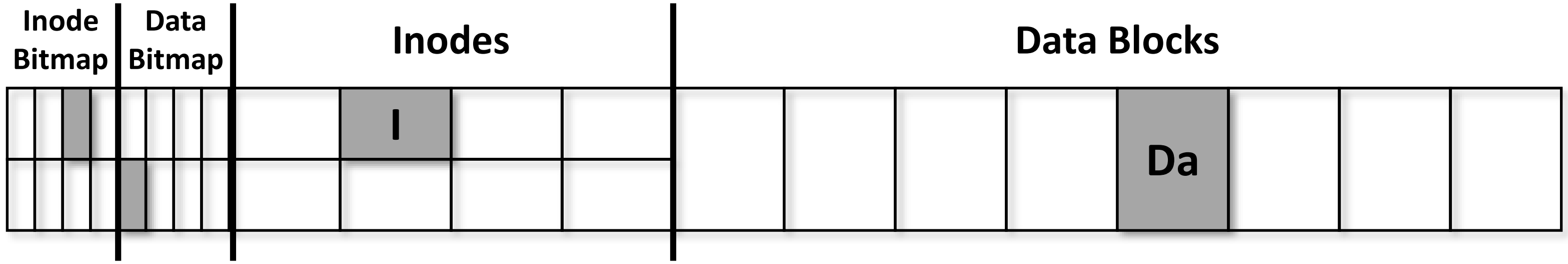
# Crash Consistency

# Crash Consistency

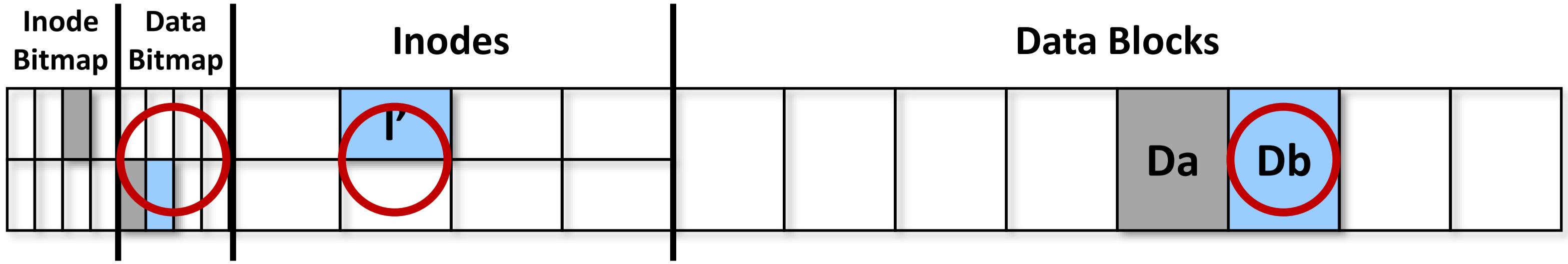
- File system may perform several disk writes to complete a single system call
  - e.g. `creat()`, `write()`, `unlink()`, `rename()`, ...
  - But, disk only guarantees atomicity of a single sector write
- If file system is interrupted between writes, the on-disk structure may be left in an inconsistent state
  - Power loss
  - System crash (kernel panic)
  - Transient hardware malfunctioning
- We want to move file system from one consistent state to another atomically

# Example: Appending Data

- Initial state

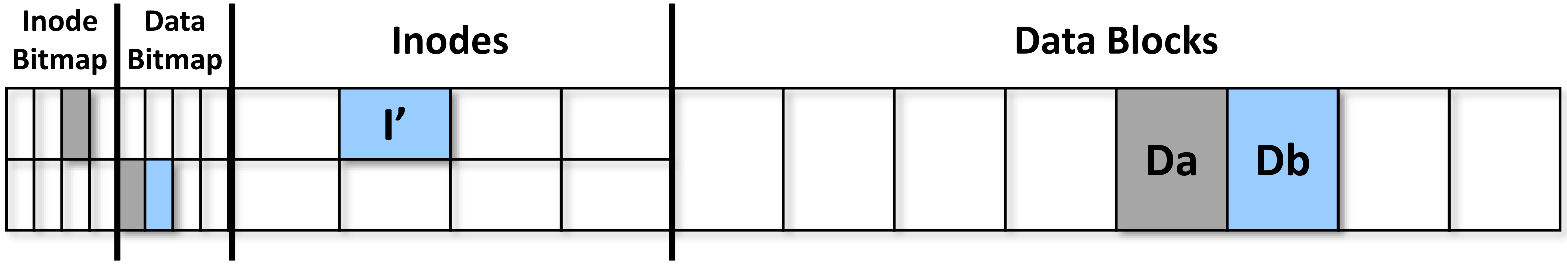



- Appending a data block Db

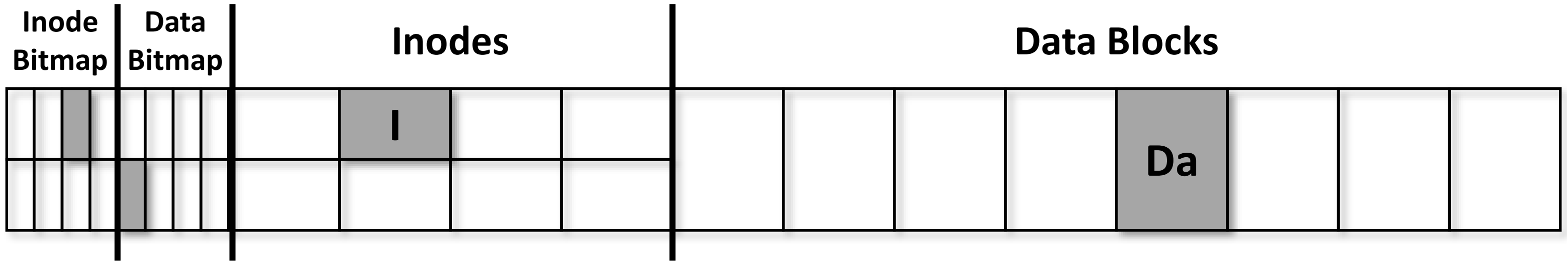


# Example: Crash Scenarios (1)

- Everything touched media: No problem

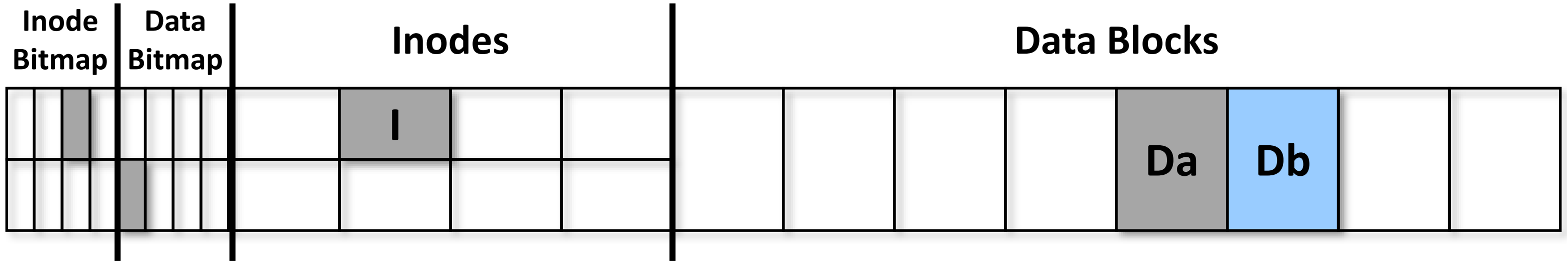



- Nothing touched media: No problem
  - Due to page cache or internal disk write buffer



# Example: Crash Scenarios (2)

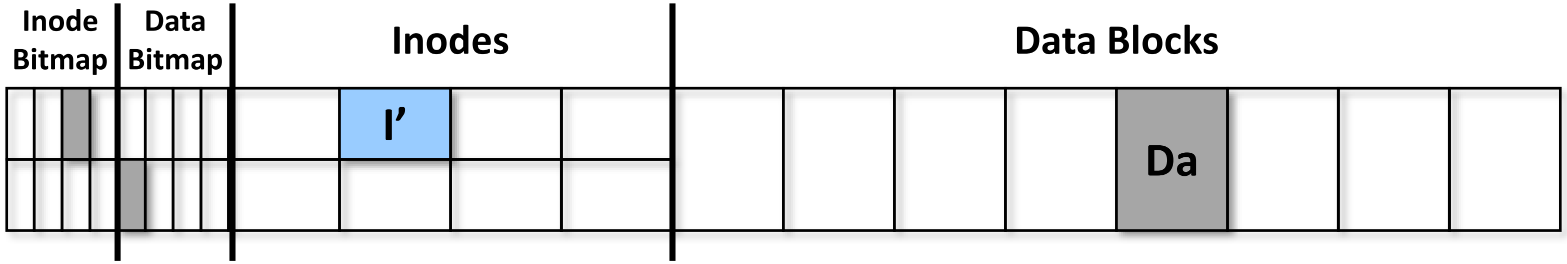
- Only data block (Db) is written: OK




  - No inode points to data block 5 (Db)
  - Data bitmap says data block 5 is free

# Example: Crash Scenarios (3)

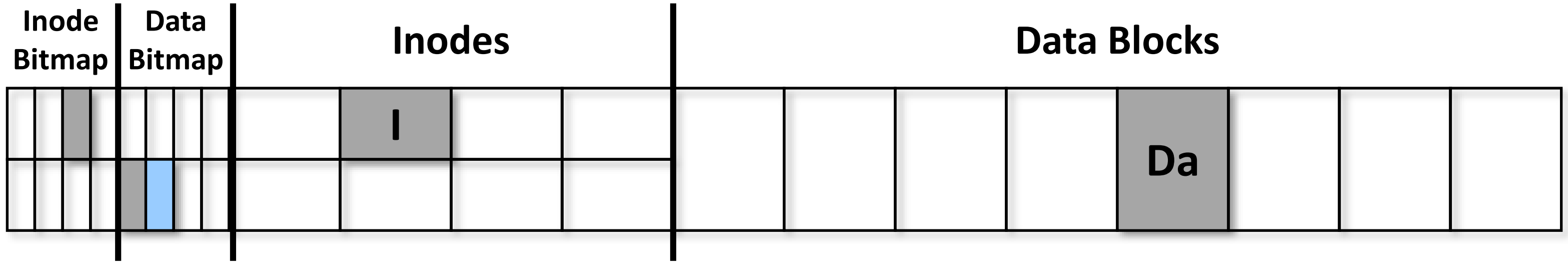
- Only updated inode (I') is written: Inconsistency




- Inode I' points to data block 5, but data bitmap says it's free
- Read will get garbage data (old contents of data block 5)
- If data block 5 is allocated to another file later, the same block will be used by two inodes

# Example: Crash Scenarios (4)

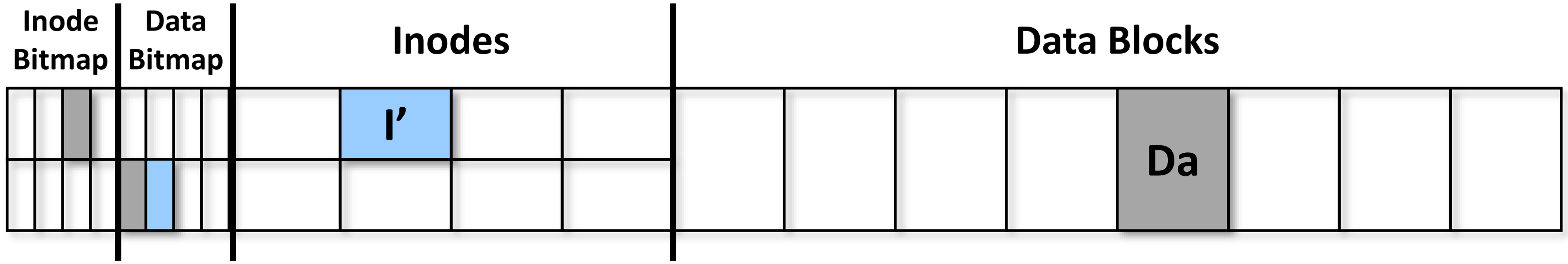
- Only updated data bitmap is written: Inconsistency




- Data bitmap indicates data block 5 is allocated, but no inode points to it
- Data block 5 will never be used by the file system
- Lost data block (space leak)

# Example: Crash Scenarios (5)

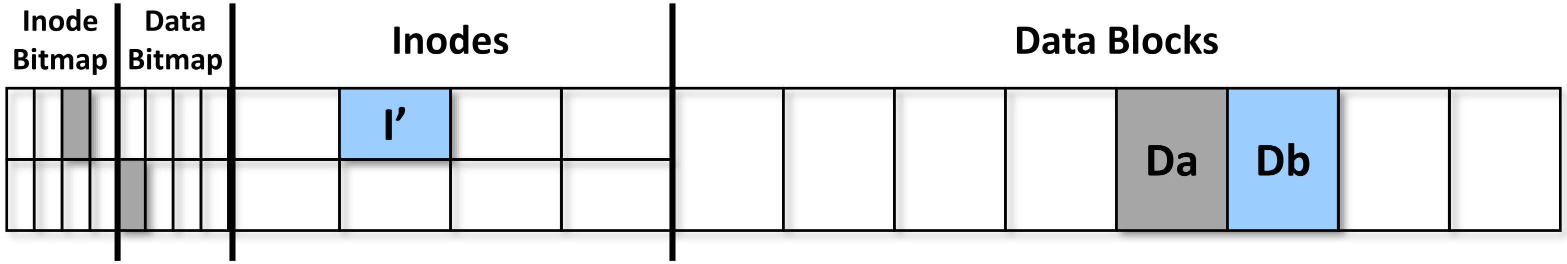
- Only inode and bitmap are written: OK




  - File system metadata is completely consistent
  - Inode I' has a pointer to data block 5 and data bitmap indicates it is in use
  - Read will get garbage data (old contents of data block 5)

# Example: Crash Scenarios (6)

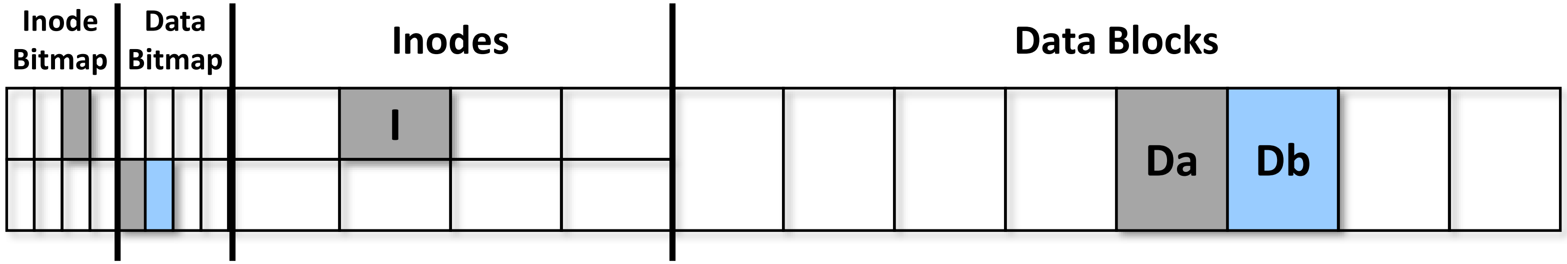
- Only inode and data block are written: Inconsistency




  - Inode I’ has a pointer to data block 5, but data bitmap indicates it is free
  - Data block 5 can be reallocated to another inode

# Example: Crash Scenarios (7)

- Only bitmap and data block are written: Inconsistency




- Data bitmap indicates data block 5 is in use, but no inode points to it
- Data block 5 will never be used by the file system
- Lost data block (space leak)

# FSCK

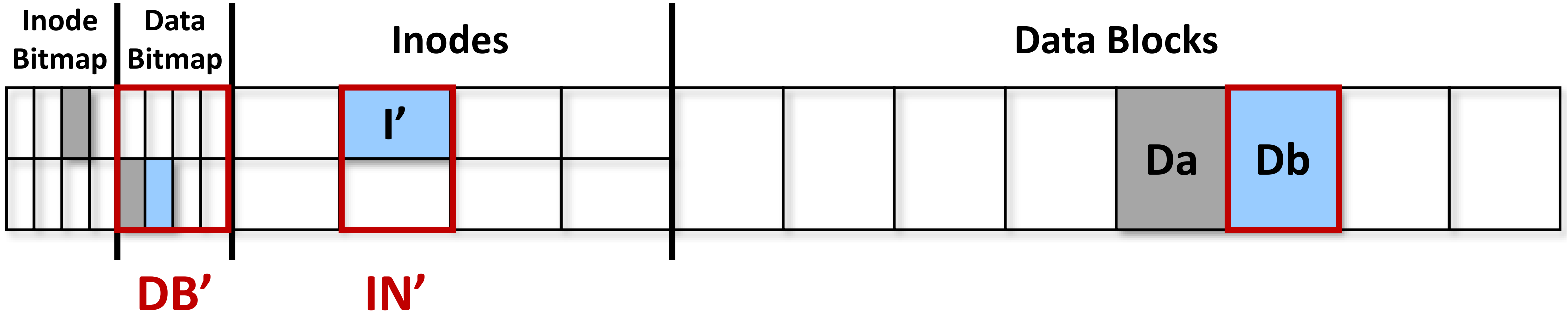
- File System Checker
  - A Unix tool for finding inconsistencies in a file system and repairing them (cf. Scandisk in Windows)
  - Run before the file system is mounted and made available
- After crash, scan whole file system for contradictions and “fix” it if needed
  - Inode bitmap consistency
  - Data bitmap consistency
  - Inode link count
  - Duplicated/invalid data block pointers
  - Other integrity checks for superblock, inode, and directories

# Journaling

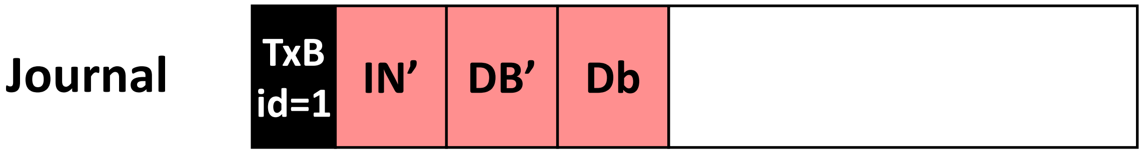
- Write-ahead logging
  - A well-known technique for database transactions
  - Record a log, or journal, of changes made to on-disk data structures to a separate location ("journaling area")
  - Write updates to their final locations ("checkpointing") only after the journal is safely written to disk
  - If a crash occurs:
    - Discard the journal if the journal write is not committed
    - Otherwise, redo the updates based on the journal data
  - Fast as it requires to scan only the journaling area
  - Used in modern file systems:
    Linux Ext3/4, ReiserFS, IBM JFS, SGI XFS, Windows NTFS, …

# Example: Appending Data (1)

- Appending a data block Db

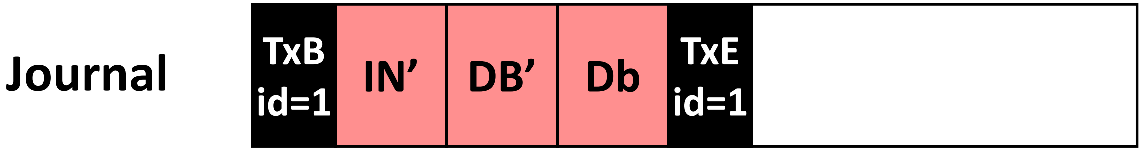



- Step 1: Journal write
  - Write journal header block (TxB), inode block (IN'), data bitmap block (DB') and data block (Db)

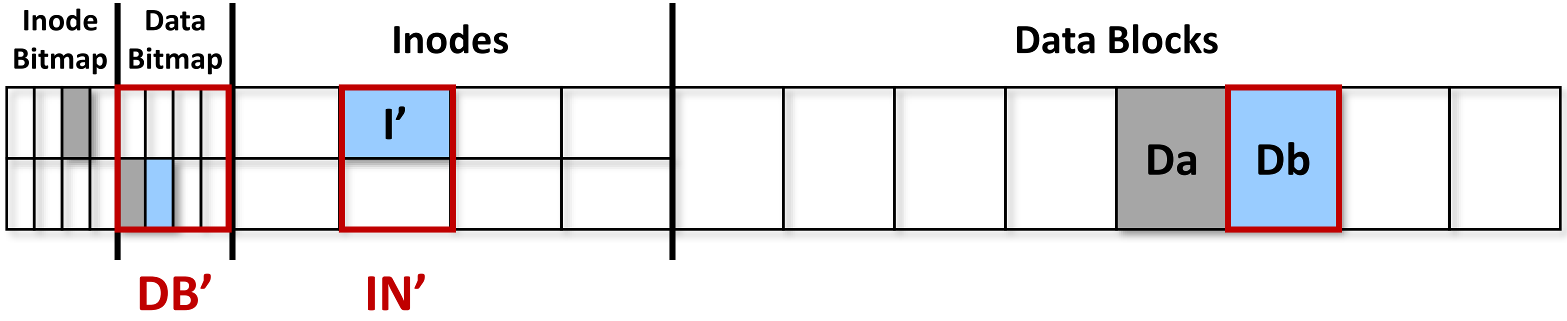


# Example: Appending Data (2)

- Step 2: Journal commit
  - Write journal commit block (TxE)

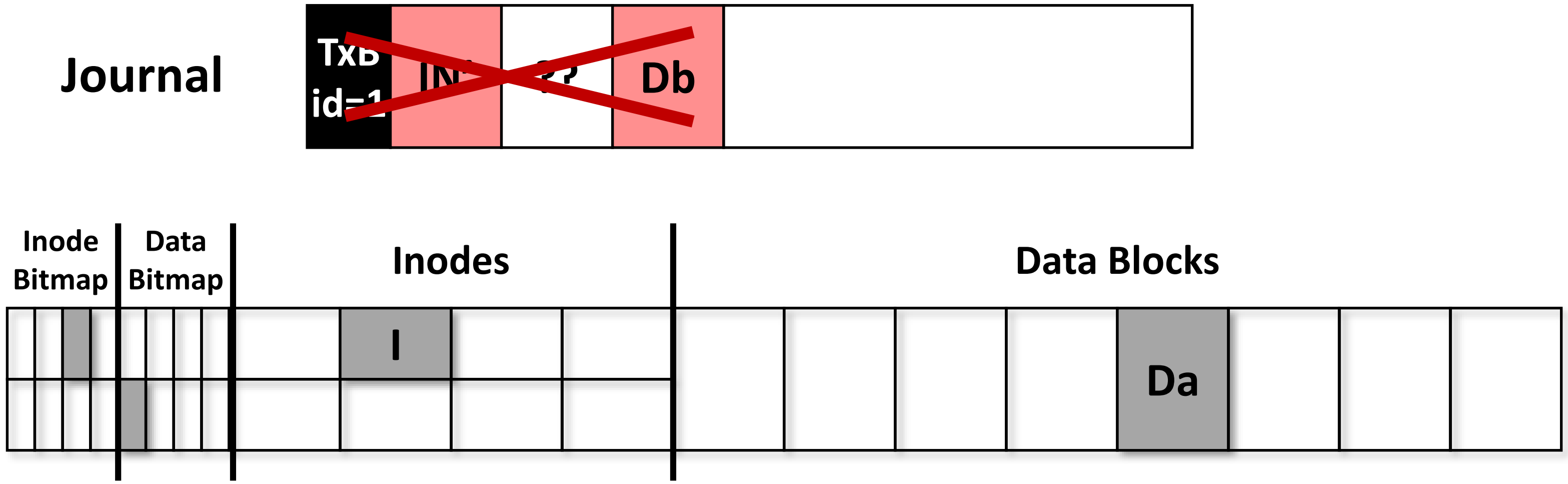



- Step 3: Checkpoint
  - Write updates to their final on-disk locations (IN', DB', Db)



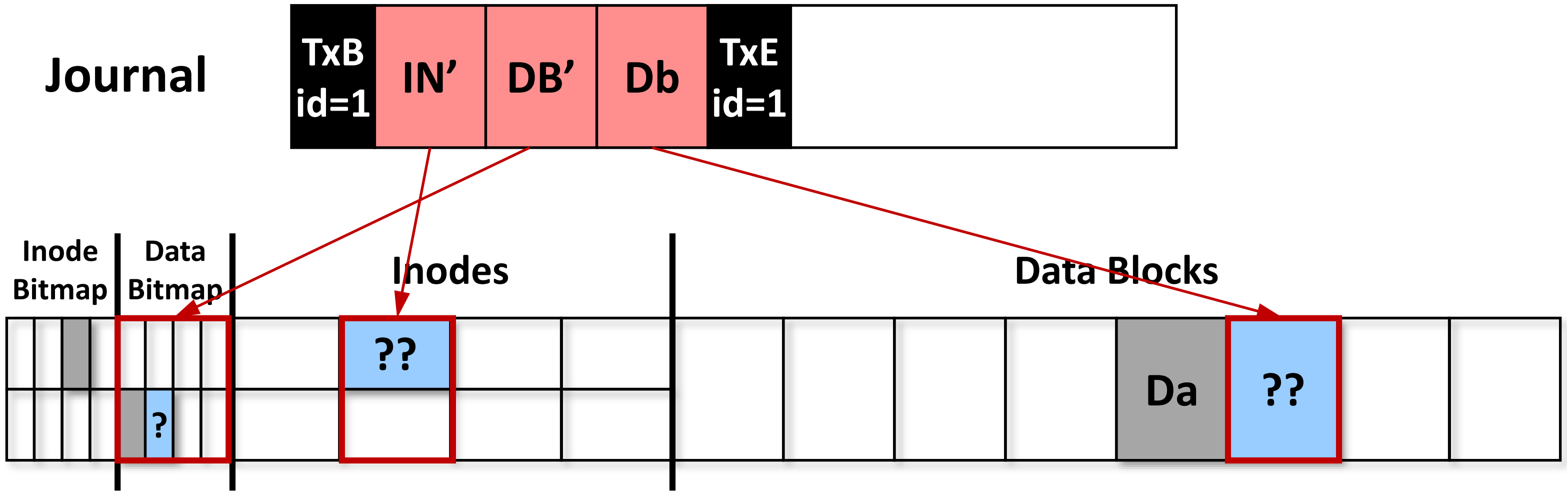
# Example: Recovery (1)

- Crash between step 1 & 2
  - Journal write has not been committed
  - Simply discard the journal
  - File system is rolled back to the state before data block Db is appended



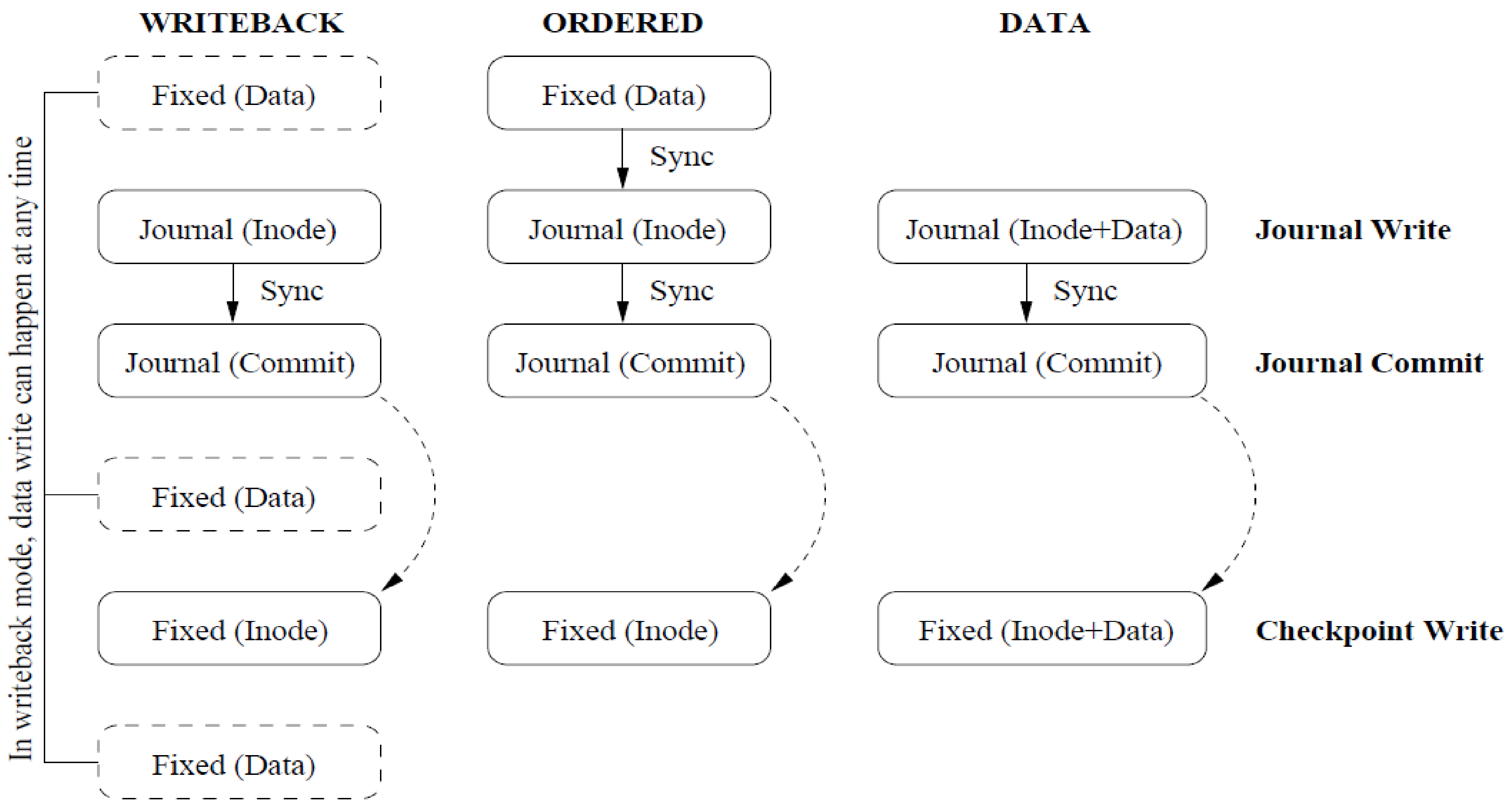
# Example: Recovery (2)

- Crash between step 2 & 3
  - Doesn't matter which metadata/data blocks were actually updated
  - Roll-forward recovery (redo logging): overwrite their final on-disk locations using the journal data



# Ext4 Journaling

- Journaling modes



# Optimizing Journaling

- Circular log
  - Mark the transaction free and reuse the journal space
- Batching log updates
  - Buffer all updates into a global transaction
  - e.g. 5 seconds in Ext3/4
- Journal checksums
  - Eliminate write barrier between journal write & commit
- Metadata journaling
  - Only guarantees metadata consistency
  - Ordered journaling in Ext3/4: force the data write before the journal is committed so as not to point to garbage