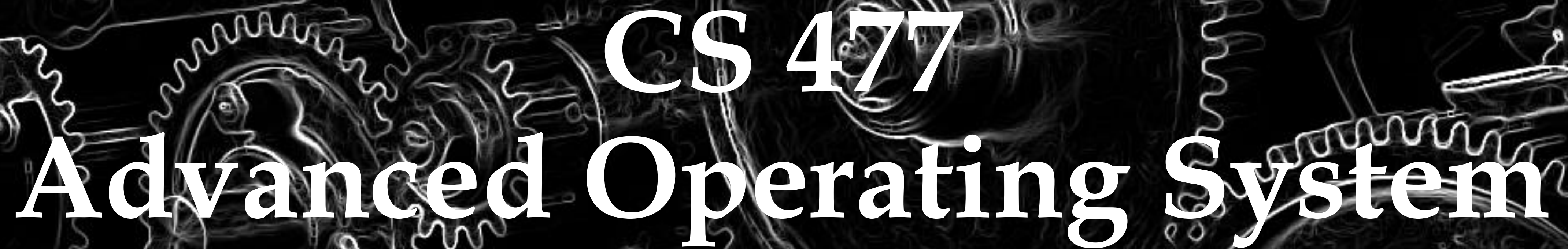


CS 477
Advanced Operating System

Lecture 13: Crash Consistency and Block IO

EPFL

Today's agenda

- Importance of crash consistency
  - Journaling
  - LFS
  - COW
- Block layer
  - bio structure
  - Request scheduling
  - IO scheduler

File system core problem

- Single FS operation → multiple disk writes
- System can crash at **any** time

→Inconsistent file system state

Three solutions:

1. Journaling (ext3/ext4, XFS)
2. Log-structured FS (LFS, F2FS)
3. Copy-on-write (btrfs, ZFS, WAFL)

EPFL

Problem: File append example

```
// User code:
int fd = open("log.txt", O_WRONLY | O_APPEND);
write(fd, "hello\n", 6);
close(fd);
```

Block bitmap (1 bit) 0 → 1	Inode (pointers + mtime)	Data block "hello\n"

On-disk structures that change

// What the file system must do:

1. Allocate new data block → Update block bitmap
2. Write data to block → Write "`hello\n`" to block
3. Update inode → Add block pointer
4. Update inode timestamp → `mtime = now()`

Problem: 3 separate disk writes, and the system can crash between ANY of them!

Crash scenarios: What can go wrong

1. Only bitmap written

Block Bitmap: ✓ (block marked allocated)

Inode: ✗ (no pointer to block)

Data: ✗ (not written)

Result: **space leak**

→ Block marked used but unreachable

→ Lost disk space until fsck

Crash scenarios: What can go wrong

2. Only inode written

Block Bitmap: ✗ (block still marked free)

Inode: ✓ (points to block N)

Data: ✗ (not written)

Result: **corruption**

→ Inode points to unallocated block

→ May be allocated to another file

→ Two files share the same block!

Crash scenarios: What can go wrong

3. Only data written

Block Bitmap: ✗ (block still marked free)

Inode: ✗ (no pointer)

Data: ✓ ("hello\n" written)

Result: **data loss**

→ Data written, but unreachable

→ Will be overwritten when block allocated

Root cause of the problem?

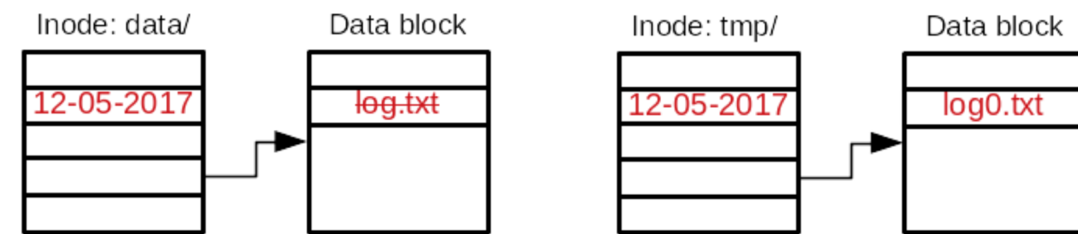
Question: What is the root cause we have inconsistent state?

- Disk guarantees only single-block **atomicity**
- But FS operations need **multi-block atomicity!**

EPFL

Exercise

- File rename: **`mv /data/log.txt /tmp/log0.txt`**
- Required updates:
  A. Remove "log.txt" from /data directory
  B. Update /data inode timestamp
  C. Add "log0.txt" to /tmp directory
  D. Update /tmp inode timestamp



- Questions: What happens if crash occurs:
  1. After A, before C — File disappears entirely (worst case!)
  2. After A and C, before B and D — File moved, but wrong timestamp (acceptable)
  3. In the middle of writing /data directory block? — Directory may be corrupted with partial entry

Journaling: Two philosophical approaches

Approach 1: **Lazy/optimistic**

- *Update disk directly, fix inconsistency later*
- On crash, run fsck (file system check)
  - Scans entire disk
  - Checks all pointers
  - Fixes inconsistencies
- Example: ext2 (old default Linux)
- **Problem: fsck on the whole disk: O(disk)**
  - 1-3 hours for full check for 1 TB disk

Journaling: Two philosophical approaches

Approach 2: **Eager/pessimistic**

- *Maintain copy/log of updates*
- On crash, replay log (seconds)
  - Only check logged operations
  - Replay journal: O(uncommitted transactions)
- Examples: ext3, ext4, XFS, JFS

- **Pro: Fast recovery (seconds)**
- **Cons: Write overhead during normal operations**

Logging: UNDO vs. REDO

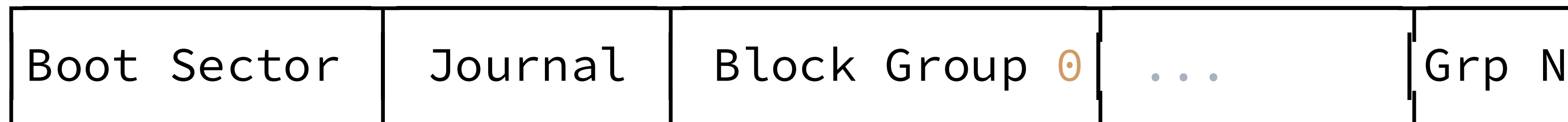
	UNDO logging	REDO logging
Also know as	Rollback journaling	Write-ahead logging (WAL) journaling
Logging	How to undo a change (old copy)	How to reproduce a change (new copy)
Read	Read from original location	Read from log if there is a new copy

- Logs allow for stashing file operations information in a scratch pad
  - Later this information is replayed in case crash occurs
- **Turns multiple writes into atomic action!**

Journaling fundamentals

- Journal = Write-ahead log
- Rule: Before modifying disk structures in place, write description of change to journal

// Journal structure in ext3/ext4:



↑
Fixed location, circular buffer

// Journal contents (logical view):

↑ Transaction begin

↑ Transaction End

Basic journaling protocol (buggy: don't use)

- Write protocol
  1. Write transaction starts
  2. Write all blocks to the journal
  3. Write transaction commit
  4. Checkpoint: write to the final location
- Recovery protocol:
  - Replay committed transactions
- Read protocol:
  - Design 1: If a block is in the log, read the log; else, read content from the original location
  - Design 2: Prevent eviction of journaled page until checkpointing
    - Read content from the page cache or from the original location

Q. Why is this approach buggy?

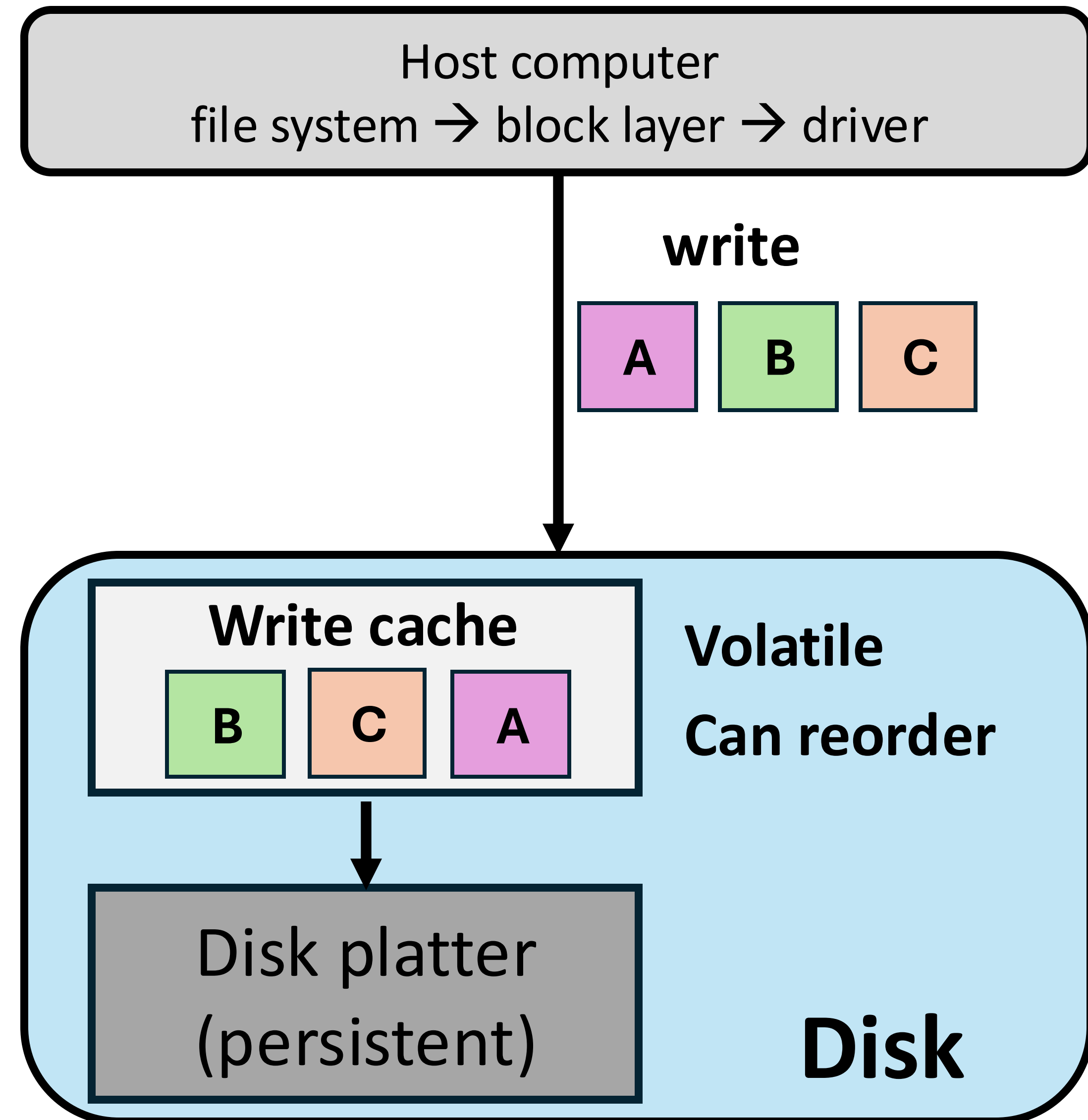
EPFL

Modern disks have caches

- Disk caches: critical for performance
  - Cache tracks on reads
  - Buffer writes before committing to physical memory
  - Example: SMR, SSD, HDD, etc.
- Caches improve performance up to 2x for some workloads
- Writes can be re-ordered!

Q. Is reordering really an issue?

Q. How to avoid reordering of writes?



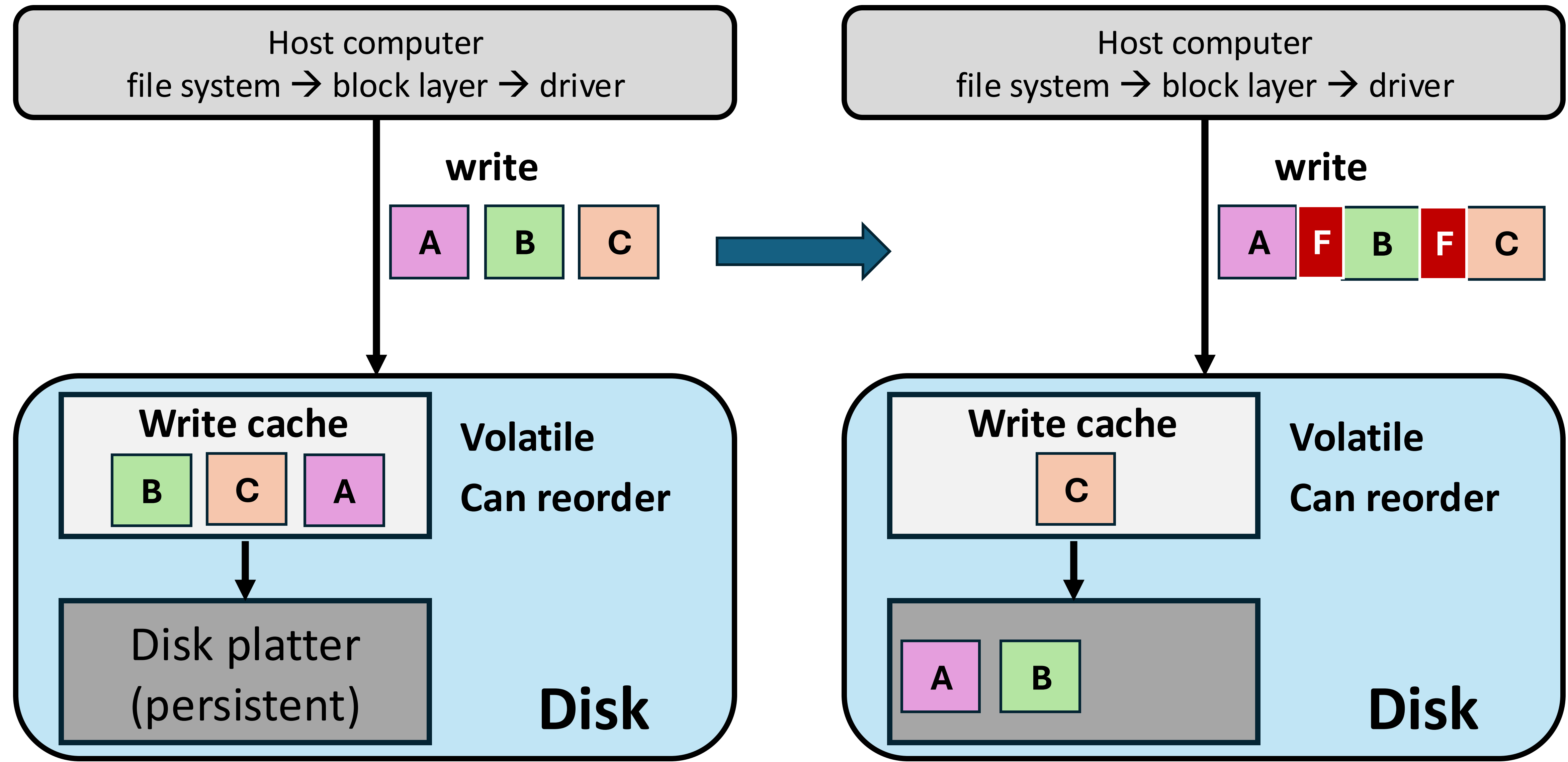
Issue with reordering

Time	Host writes:	Cache:	Disk:
t0	TxBegin + Data	[TxB, D]	[]
t1	TxCommit	[TxB,D,TxC]	[]
t2	Disk decides to write TxB first	[D,TxC]	[TxB]
t3	Disk writes TxC	[D]	[TxB, TxC]
t4	⚡ CRASH! ⚡	LOST!	[TxB, TxC]

- Result: Journal has TxBegin and TxCommit, but **no data**!
- Recovery will replay garbage → **corruption**
- This bug actually existed in early ext3 implementations → led to data loss after power failure

EPFL

Ordering writes using write barrier



EPFL

Journaling in action (ext4)

1. Start a transaction
2. Get journal access to blocks to modify
3. Modify the buffer in memory
4. Mark buffer as dirty
5. Commit transaction

Checkpoint thread

```
-> for each committed transaction:
->  for each buffer in transaction:
->    write buffer to final disk location
->  mark transaction as checkpointed
-> reclaim journal space
```

jdb2 behind the scenes

1. Gather all dirty buffers in transaction
2. Write TxBegin + all buffers to journal
3. Issue barrier (blkdev_issue_flush)
4. Write TxCommit
5. Wait for commit to reach disk
6. Mark transaction as committed
7. Wake up checkpoint thread

EPFL

Several write barrier implementations

1. FUA (force unit access)
   - Set FUA bit in SCSI/SATA command
   - Disk guarantees: write to media, not just cache
2. FLUSH CACHE command
   - Explicit command to disk: persist cache
3. Disable write cache entirely
   - Slow but simple

- Linux defines as **`blkdev_issue_flush()`** (block/blk-flush.c)

EPFL

Journaling modes: Full data mode

1. data + metadata
   - `Journal: [TxB][Data][Metadata][Flush][TxE]`
   - Then checkpoint to final locations

- Pros: Maximum safety - can recover file contents
- Cons: Every block written TWICE (journal + final)
- Performance: ~70% of ordered mode
- Use case: Mission-critical systems

EPFL

Journaling modes: Ordered mode

2. Metadata only with the guarantee of data being available
    - `Data: [Write to final location][Flush]`
    - `Journal: [TxB][Data][Metadata][Flush][TxE]`
    - Then checkpoint to final locations

- Guarantees that no garbage in files after crash (due to data flush)
- Pros: Good performance, safe metadata
- Cons: Recent appends may be lost (crash occurred before journal)
- Performance: 100% (default mode)
- Use case: Default for most systems

EPFL

Journaling modes: Writeback mode

3. Metadata only but no guarantee about data
   - `Journal: [TxB][Data][Metadata][Flush][TxE]`
   - `Data: [Write anytime, no ordering]`
   - Then checkpoint to final locations

- Issue: file may contain garbage
- Pros: best performance
- Cons: Files can have old data from other files
- Performance: > 100%
- Use case: Temporary/scratch file systems

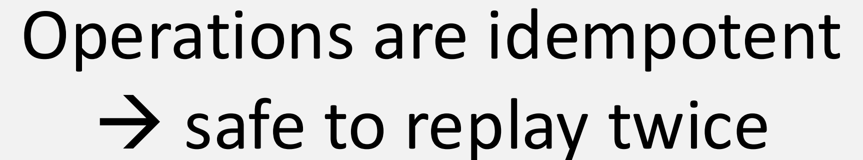
EPFL

Recovery protocol in ext4

- Set of operations on mount after crash:
  1. Load journal information
  2. Check if the disk was clean during mount time
  3. Phase 1: Scan
     1. Find all committed transactions
     2. Build list of blocks to replay
     3. Detect last valid transaction
  4. Phase 2: Replay
     - For each committed transaction:

       For each block in transaction:

       Write block to the final location



Q. What if the crash happens again while replaying?

Log-structured file system (LFS)

- "The Design and Implementation of a Log-Structured File System"

  By Mendel Rosenblum and John K. Ousterhout (ACM TOCS 1992)

  Key idea: Treat disk as a big sequential log!

- Observations (early 1990s):
  - RAM sizes growing exponentially
  - File system caches getting huge
  - Most **reads** now satisfied from the page cache
  - Disk I/O is mostly **write dominated**

LFS

- Traditional FS write pattern
  - One write → multiple seeks
    1. Seek to inode location
    2. Seek to data block
    3. Seek to data bitmap
    4. Seek to parent directory

    = 4 seeks x 5-10ms = 20-40 ms latency
- LFS insight:
  - Optimize for writes if disk is write-limited
  - Buffer data in memory
  - Write sequentially in large segments to the disk once
  - No seek required

LFS: key idea

- Use the whole disk as a **log**
  - Log allows for writing only at the head
  - If disk is managed as a log → No effective movement of head for seek
- Log allows for adding file sequentially at the end of the log

Traditional FS: Fixed metadata locations

```
| Boot   | Inodes   | Bitmaps   | Data   | Data |
            ↑
    Inode i is always at location:
    inode_table_start + (i * sizeof(inode))
```

LFS: Disk is a log

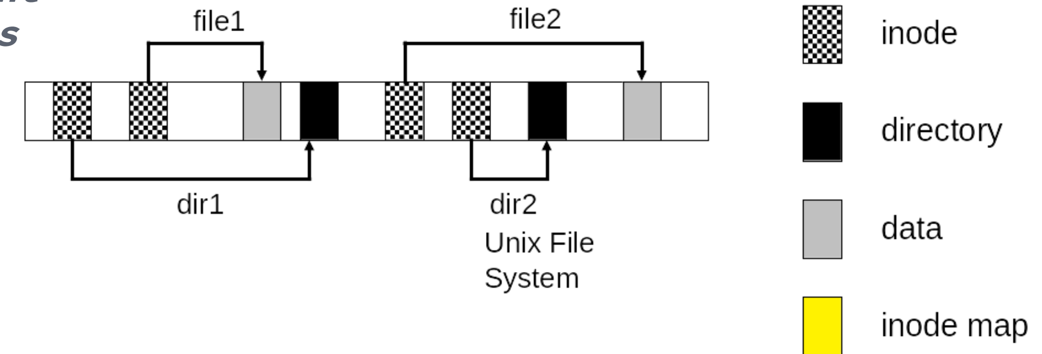
```
| [Seg0: I5,D,D,M]   [Seg1: I12,D,I5',D] [Seg2...] |
        ↑                       ↑
    Inode 5 v1            Inode 5 v2 (updated!)
Segments written sequentially, never overwrite
```

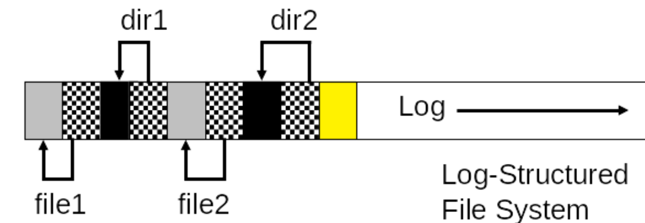
LFS: data structures and organization

```
struct lfs_segment {
  segment_summary_t summary;      // What's in this segment
  char data[SEGMENT_SIZE];        // 512KB - 1MB of writes
};

// Inode map: Where is each inode on the disk?
// (Because inodes move when updated!)
struct inode_map {
  disk_addr_t inode_locations[MAX_INODES];
};

// Checkpoint region (fixed location):
struct checkpoint {
  disk_addr_t imap_location;      // Where is inode map?
  timestamp_t timestamp;
  uint64_t segment_head;          // Where to write next
};
```





Blocks written to create two 1-block files: dir1/file1 and dir2/file2, in UFS and LFS

EPFL

LFS: Read operation

- Similar to UNIX reads

1. Read the checkpoint (cached in memory)
2. Read inode map (cached in memory) → contains all inode information
3. Read inode
4. Read data blocks

LFS: Write operation

- Get the currently active segment
- Add data blocks and inode information to the segment
- When segment full → write to disk (one giant IO)
- Update inode map `imap[ino_num] = new_inode_disk_location`
- Periodically write the checkpoint

Q. What is the issue with write operations?

LFS garbage collection

- Issue: Segments fill up with garbage data
  - Example: Update file block 100 times

  → 100 versions of the block in the log

  → Only latest version is "live"

  → 99 versions are "dead" (garbage)

```
Segment state over time:
Initial:                [D1 D2 D3 D4 D5 D6 D7 D8] 100% live
After update:           [D1'D2'D3 D4 D5 D6 D7 D8] 75% live (D1,D2 dead)
After delete:           [D1'D2'D3 X D5 D6 D7 D8] 63% live (D4 dead)
```

EPFL

LFS garbage collection via cleaner process

- LFS requries contgiuous free space on the disk
- **Cleaner:** Compact segments to free up space
    1. Choose victim segments (policy decision)
    2. For each live blocks acrosss segments, copy them to new segment
    3. Free old segments
- Victim selection policy dictates performance
    - Segment utilization: Higher utilization (more live blocks) – larger copy overhead
    - Age: Recently written segment → more likely to change
- Maintain per-segment summary data structure to track live blocks

Crash consistency in LFS

- No special mechanism required
- Log is always consistent
- Last checkpoint is required for valid states

LFS design in Linux

- F2FS (flash-friendly file system)
  - Optimized for LFS for NAND flash SSDs
  - Used in mobile devices

- LFS design is widely adopted
  - Flash translation layer (FTL), firmware for NAND flash, is LFS managing a single file
    - Maintains a mapping between blocks used on SSD and the logical blocks exposed to file systems

Copy-on-write (COW) file system

- “File System Design for an NFS File Server Appliance”
  - Dave Hitz et al, USENIX Winter 1994
  - **Write-Anywhere File Layout (WAFL):** the core design of NetApp
- Inspired by LFS:
  - Never overwrite a block (like LFS)
  - Do not clean a segment (**unlike** LFS) → keep older versions for **free**
    - Snapshots become easy
- **Key idea:** Represent file system as a single tree and never overwrite blocks
- Modern file system: WAFL, ZFS, btrfs, bcachefs

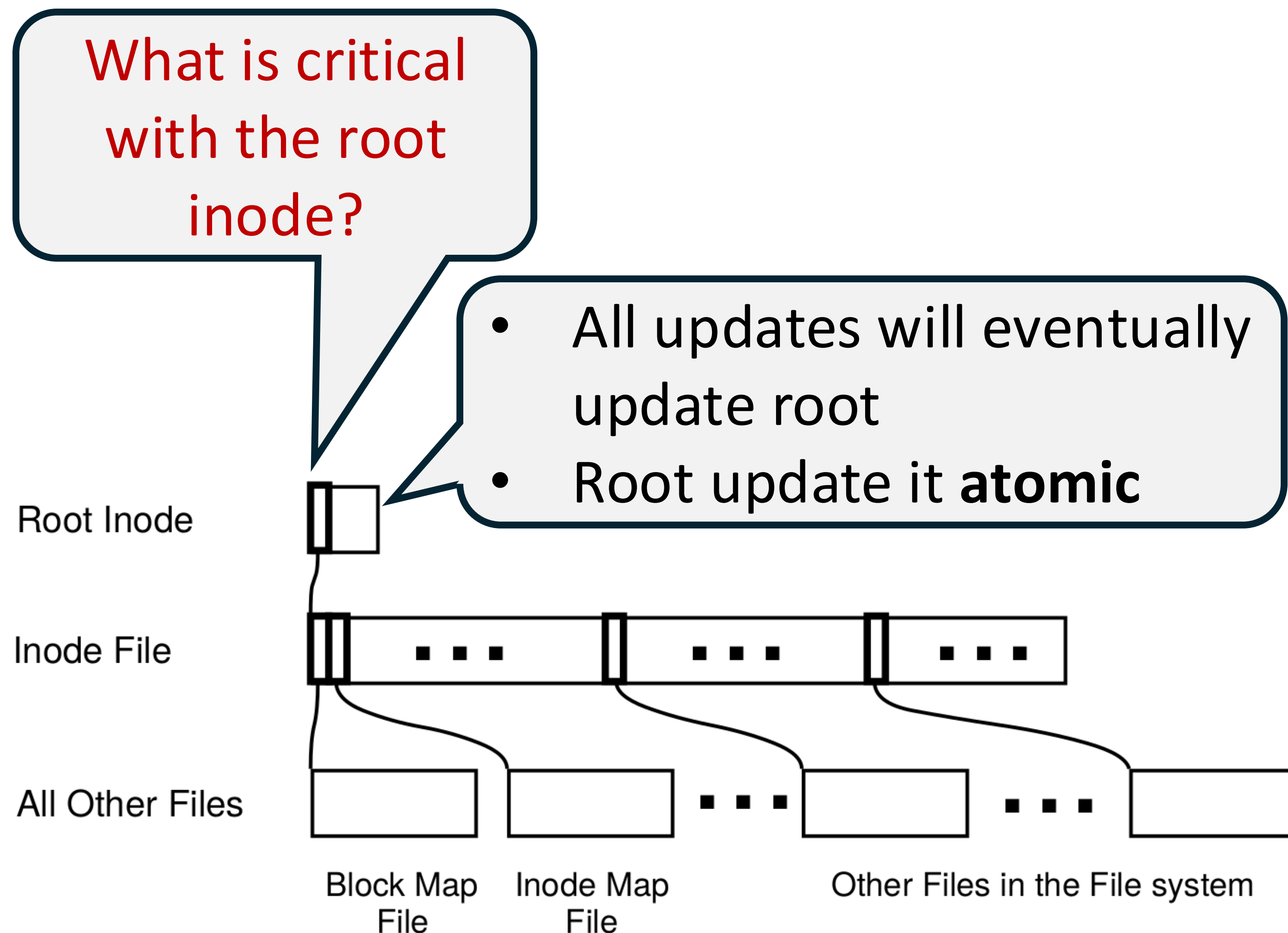
EPFL

CoW FS principle

- **Update copy, not original**
- Traditional FS:
  - *write($block_{100}$)* → *Overwrite block 100*
- CoW FS:
  - write($block_{100}$) → Allocate $block_{200}$
    → Write new data to $block_{200}$
    → Update pointer: 100 → 200
    → Old block 100 remains (for snapshots)

WAFL design

- Everything is a file, including metadata
- Layout: a tree of blocks
  - A root inode: root of everything (static location)
  - An inode file contains all inodes
  - A block map file contains free blocks
  - An inode map file contains free inodes



EPFL

Why keep metadata in files?

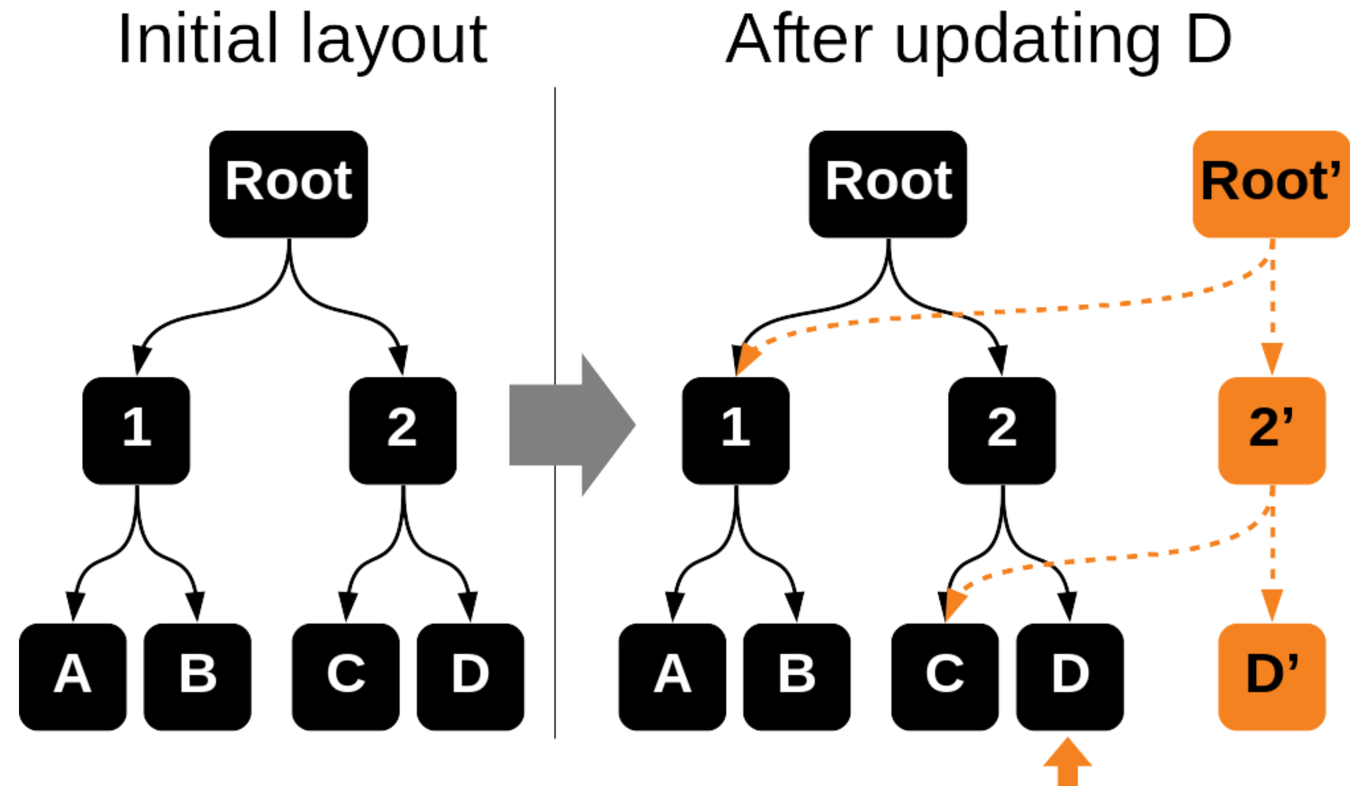
- Allow metadata blocks to be written anywhere on disk
  - Origin of “Write Anywhere File Layout”
- Easy to increase the size of the file system dynamically
  - Adding a disk requires only adding i-nodes
- Enable copy-on-write to create snapshots
  - Copy-on-write new data and metadata on new disk locations
  - Fixed metadata locations are cumbersome

EPFL

WAFL read/write operation

- Reads same as UNIX reads
  - Root inode → inode file → inode
  - inode: file offset → disk block mapping
- Writes updates the whole tree

1. Allocate new data blocks
2. Allocate new inode block
3. Update inode file
   - Recursively update parent blocks
4. Atomically update the root



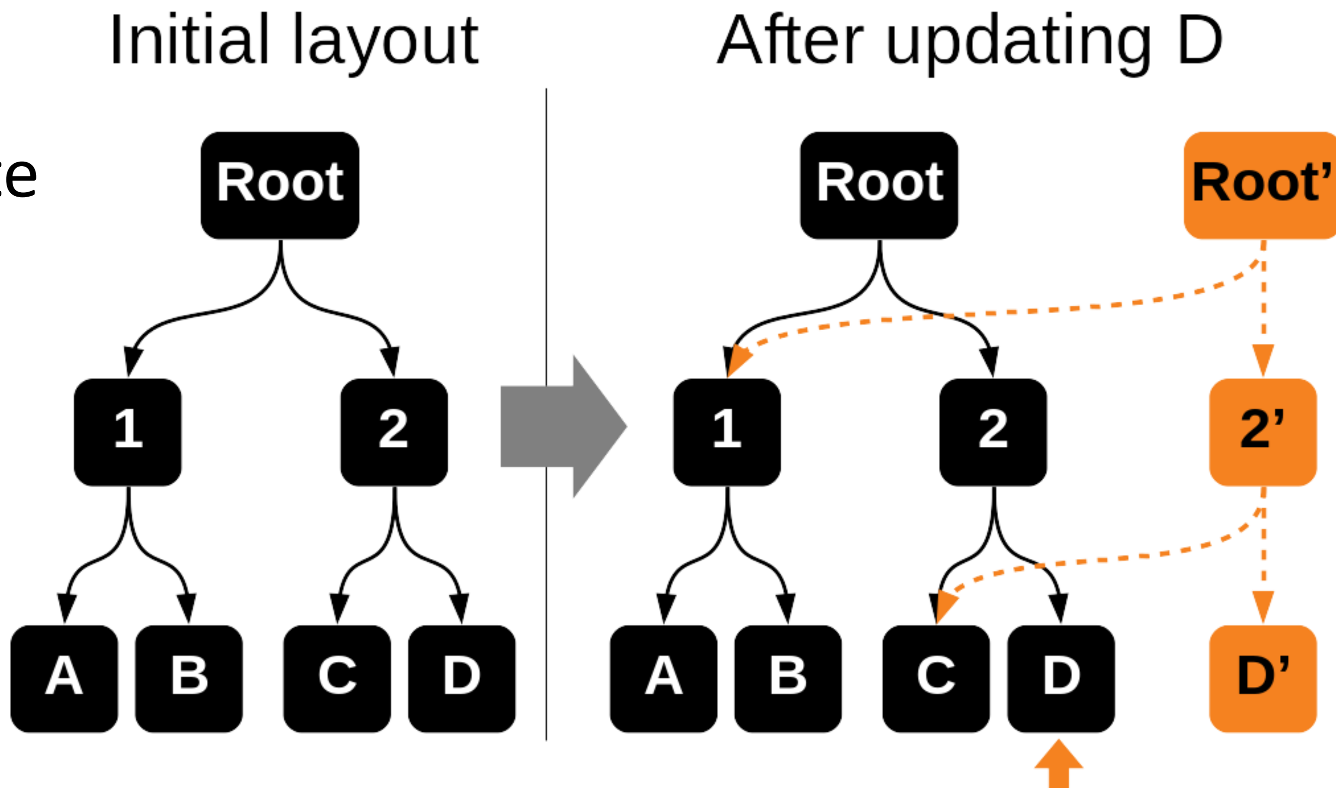
Q. Why is it easy to provide crash consistency?

EPFL

Crash consistency in CoW

Root inode → consistent snapshot

- Old root → old consistent state
- New root → new consistent state
- Crash anywhere: Use last valid root
- No write-ahead log required



CoW in Linux

- btrfs: B-Tree FS
  - CoW at block layer
  - Snapshots and subvolumes
  - Built-in RAID
  - Online resize
  - Checksum on each data and metadata
  - Compression

Comparison between ext4 and btrfs

- Advantages over ext4:
  - Snapshots (free, instant)
  - Data integrity (checksums catch corruption)
  - Flexible space management
  - Online defragmentation
- Disadvantages:
  - Buggy (not as mature as ext4)
  - More complex (harder to debug)
  - Write amplification (CoW overhead)

Write amplification factor (WAF)

WAF: Physical writes / logical writes

Update 1 block in a file:

WAF = 16/4 = 4x

- ext4 (ordered mode):
  - Write data block = 4KB
  - Write inode (to journal) = 4KB
  - Write commit block = 4KB
  - Write inode (final) = 4KB

WAF = 20/4 = 5x

- ex4 (journal mode)
  - Write data (to journal) = 4KB
  - Write inode (to journal) = 4KB
  - Write commit block = 4KB
  - Write data (final) = 4KB
  - Write inode (final) = 4KB

Update 1 block in a file:

WAF = 24/4 = 6x

- btrfs (CoW):
  - Write data block = 4KB
  - Write inode block = 16KB (updated path)
  - Write root block = 4KB

WAF = 4/4 = 1x

- f2fs (LFS)
  - Write to current segment = 4KB
- But: Garbage collection adds overhead

EPFL

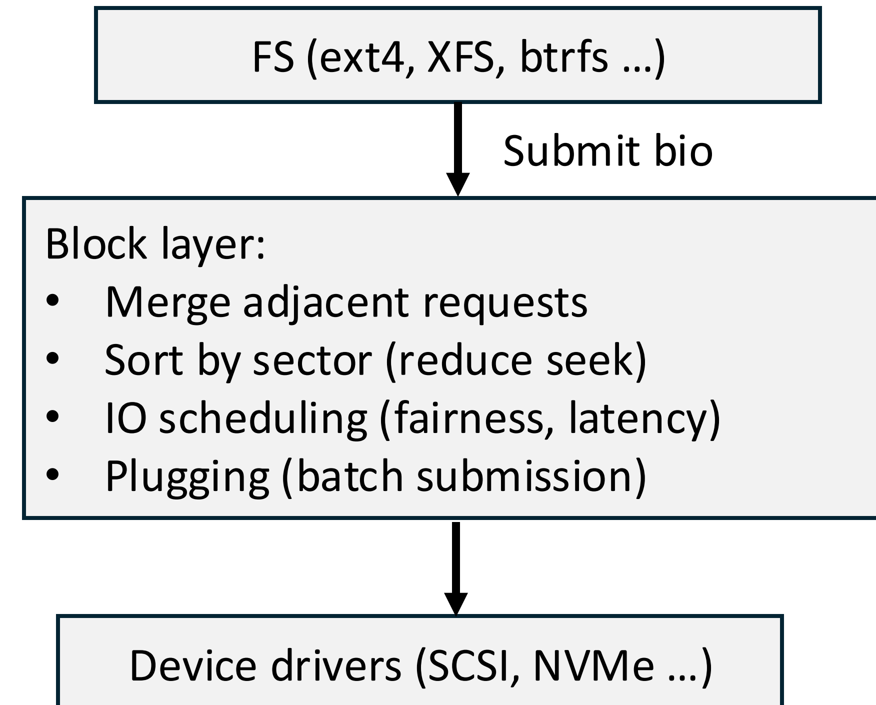
Today's agenda

- Importance of crash consistency
  - Journaling
  - LFS
  - COW
- **Block layer**
  - bio structure
  - Request scheduling
  - IO scheduler

Importance of block layer

- Without block layer:
  - Each FS talks directly to device drivers
  - Duplicate code for merging, sorting, scheduling
  - No optimization for disk seek patterns
- With block layer:

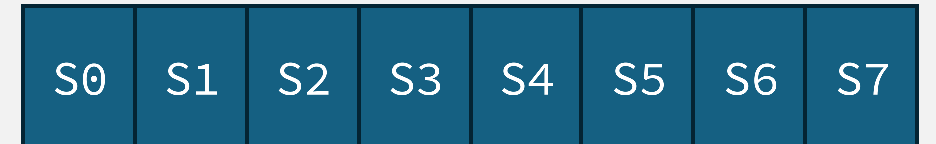
  Block layer virtualizes disk access like process scheduler virtualizes the CPU



EPFL

Anatomy of a block device

- Sectors (hardware view)
  - Minimum addressable unit of a device
  - Physical property → hard sector, device block
    - Typically 512 bytes (2KB for CD-ROM)
    - Modern drives: 4KB "advanced format"
- Block (software view)
  - Unit of file system IO
  - Multiple of sector size (device constraint)
  - Power of 2, <= page size (kernel constraint)
  - Typically 4KB or 8KB



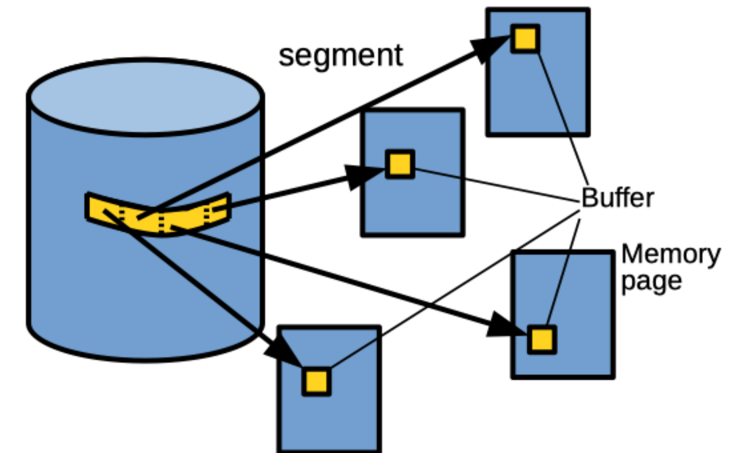
Kernel communicates in blocks, but the device driver speaks in sectors

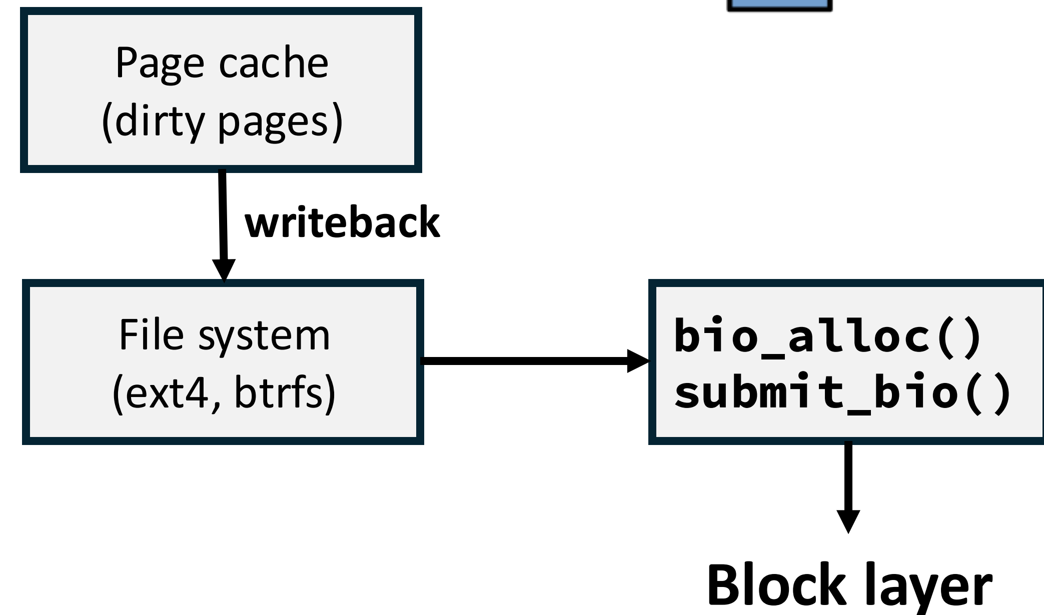
Q. Why do we care about this relationship?

EPFL

The `bio` structure

- Basic container for an active block IO operation
- A single IO can span multiple **non-contiguous memory pages**
- Scatter-gather support
- Light-weight, good for large IO





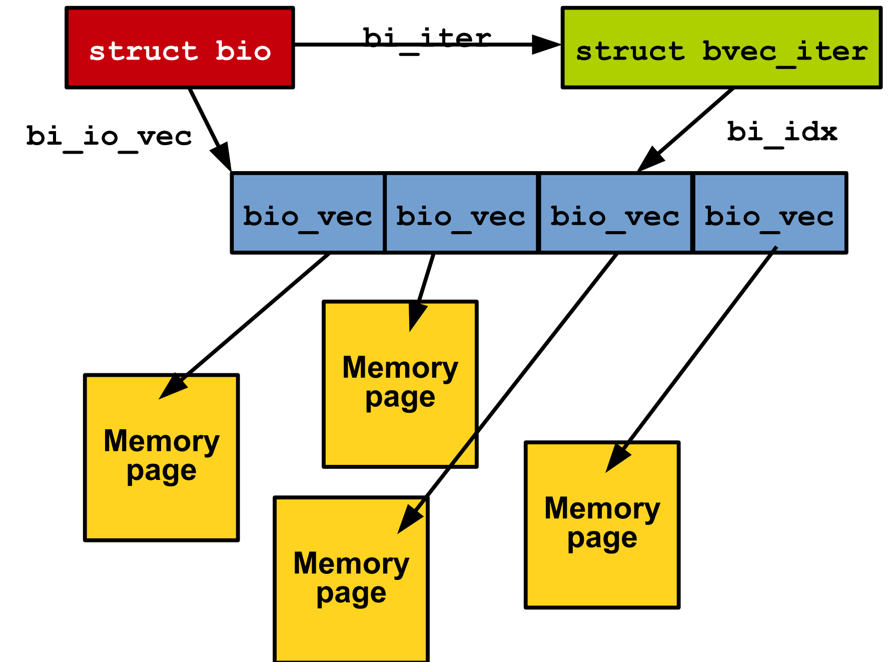
EPFL

bio struct

```
struct bio {
  struct bio *bi_next;                  /* request list */
  struct block_device *bi_bdev;         /* target device */
  unsigned short bi_flags;              /* status flags */
  struct bvec_iter bi_iter;             /* position */
  bio_end_io_t *bi_end_io;              /* completion */
  void *bi_private;                     /* owner data */
  unsigned short bi_vcnt;               /* # of bio_vecs */
 struct bio_vec *bi_io_vec;             /* segment array */
};

struct bio_vec {
  struct page *bv_page;                 /* physical page */
  unsigned int bv_len;                  /* segment length */
  unsigned int bv_offset;               /* offset in page */
};

struct bvec_iter {
  sector_t bi_sector;                   /* target on disk */
  unsigned int bi_size;                 /* remaining bytes */
  unsigned int bi_idx;                  /* current bio_vec */
};
```

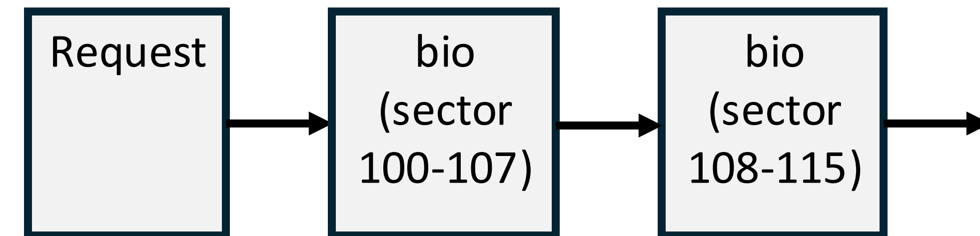


Satter-gather: single disk operation, multiple memory locations! DMA-friendly

EPFL

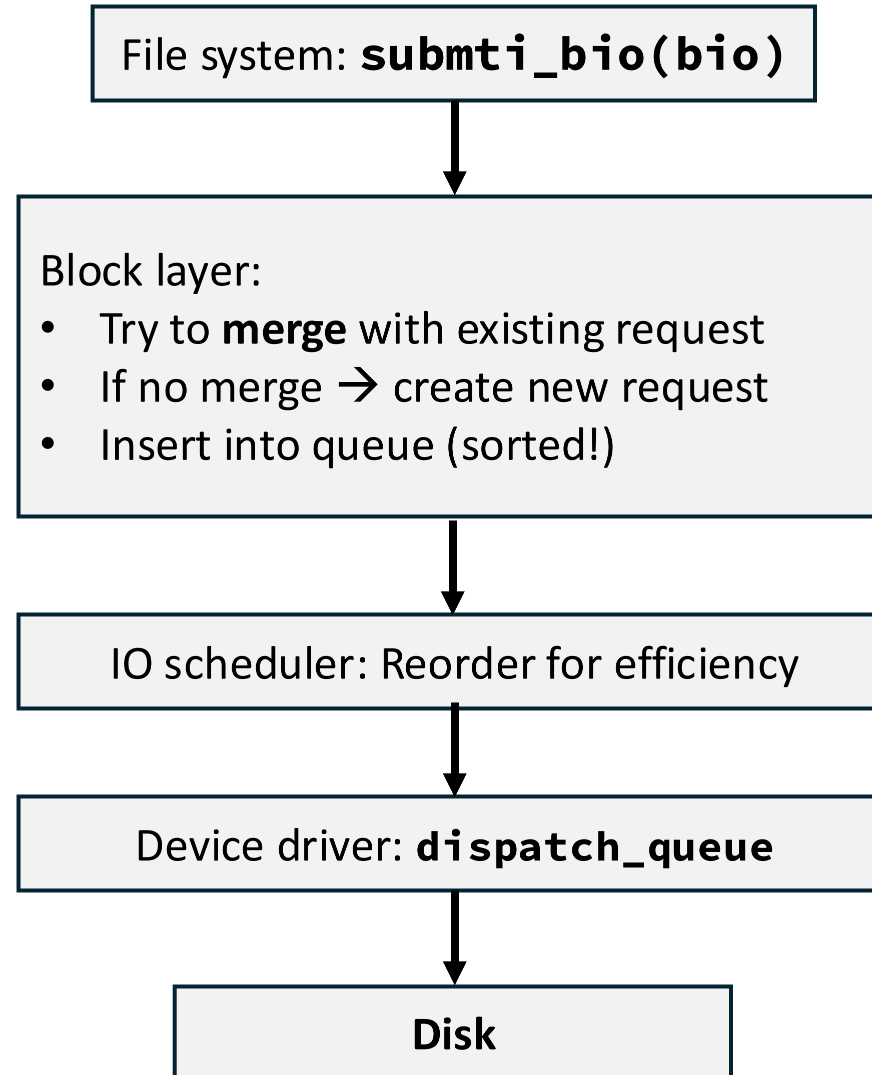
Request queue

- Block devices maintain request queue for pending IO operations
- File system adds a request to the queue
- Block device driver removes the requests from the queue and submits to the device
- A single request:
  - Represented by struct request
  - Can operate on multiple consecutive disk blocks
    - Composed of one or more bio objects



EPFL

Request flow



EPFL

IO scheduler: The issue

- Directly sending the requests to disk is sub-optimal
  - HDD seek time problem:
    - Random access (~10ms per seek) is 1000x slower than sequential (~0.01ms per block)
  - Example: Request order: 18, 5, 10, 1
    - Without scheduling: (|18 – 5|) + (|5 - 10|) + (|10 – 1|) = 27 sectors traveled
    - With sorting: 1, 5, 10, 18 ➔ 17 sectors traveled
    - Almost 40% reduction in head movement
  - Sorting algorithm is also known as the elevator algorithm
    - Define where an upcoming request should be added into the queue:
      - Front merge, back merge
      - Sorted insertion
    - Goal: minimize disk seek, best global throughput

Linux IO shedulers

- Noop/none: FIFO + merge only
  - Best for: SSD, NVMe, VM guests
- deadline: Sorted + deadline for starvation
  - Reads: 500 ms, writes: 5s
  - Best for: HDD, latency sensitive
- kyber: Simple 2-queue model
  - No seek time, best for NVMe SSDs

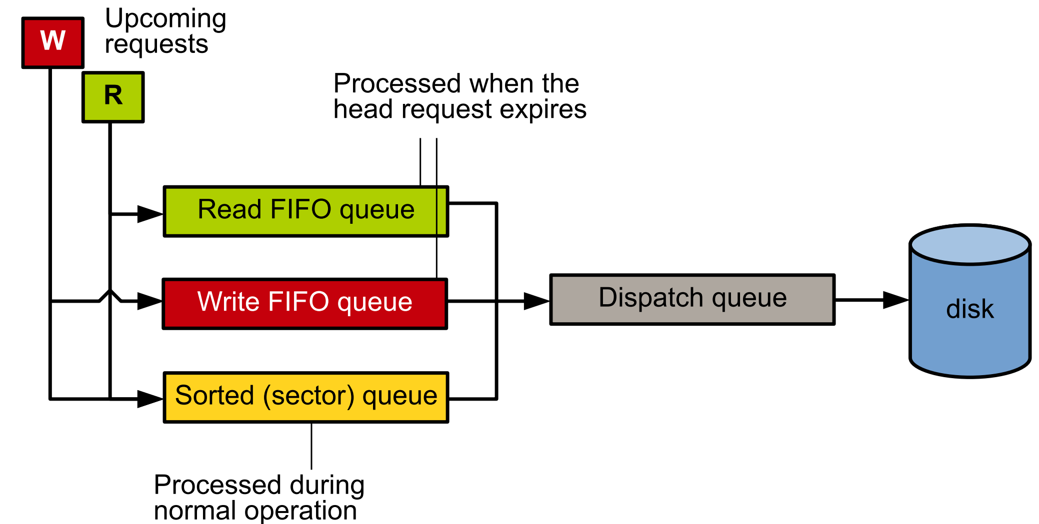
Tradeoff:

- HDD: Sort by sector → minimize seek
- SSD: No seek penalty → minimize overhead

EPFL

Deadline scheduler deep dive

- Problem with pure sorting: Writes can starve reads
  - Writes: buffered (async) → can wait
  - Reads: blocking (sync) → user waiting
- Solution: 3 queues + deadline
- Insight:
  - Normal case: Use sector-sorted order (good throughput)
  - Emergency case: Deadline expires → serve immediately (bounded latency)
  - Fairness: After some batches, allow writes to finish



Simplified deadline scheduler algorithm

- On request arrival:
  - Set a deadline for a specific request type (read: 0.5 s, write: 5s)
  - Add to FIFO queue (by arrival time)
  - Add to sorted tree (by sector number)
- On dispatch:
  - Priority 1: serve the oldest expired reads
  - Priroity 2: serve the oldest expired writes
  - Priority 3: Normal operation
    - If reads are available and writes are not starved, then serve reads
      - Example: serve 16 reads before serving a write
    - Else, serve the next write by sector order

EPFL

Further reading

- Optimistic crash consistency
- Consistency Without Ordering
- Lightweight Application-Level Crash Consistency on Transactional Flash Storage
- SFS: Random Write Considered Harmful in Solid State Drives
- LWN: A block layer introduction part 1: the bio layer
- LWN: A block layer introduction part 2: the request layer
- LWN: The multiqueue block layer
- LWN: The future of DAX