Winter 2025

EECS 482: Introduction to Operating Systems

Lecture 23: RAID & LFS

Prof. Ryan Huang

Project 4

Use assertions to catch errors early

- # of free disk blocks matches file system contents?
- Are you unlocking a lock that you hold?
- Verify initial file system is not malformed

Use showfs to verify that contents of file system match your expectations

Test cases: cover all states of data structures (e.g., direntries array)

A detour back to crash consistency

Implementing transactions

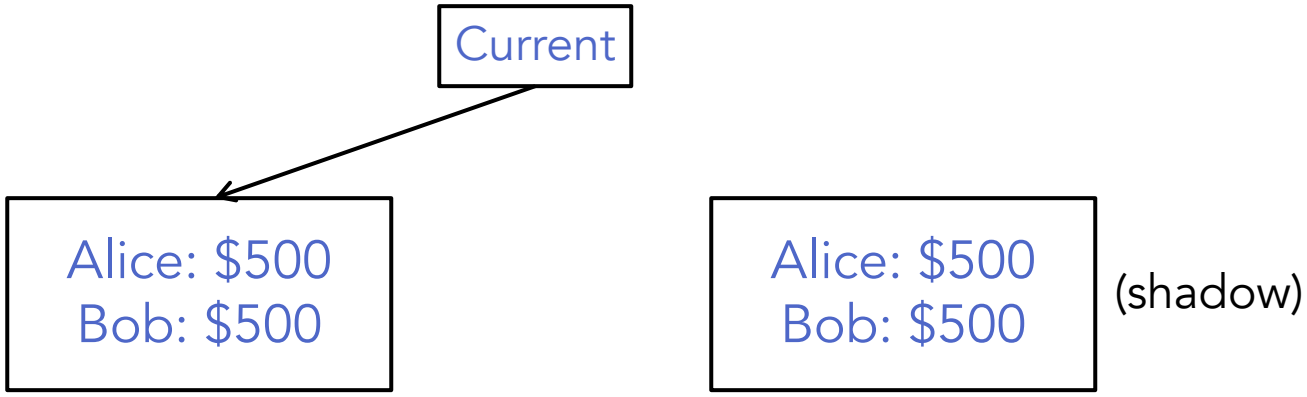
How to make a set of disk writes atomic?

- Logging (covered in lecture 20)
- Shadowing

Transactions via shadowing

Keep two copies of the data: current version + backup (shadow)

Store pointer (also on disk) to current version

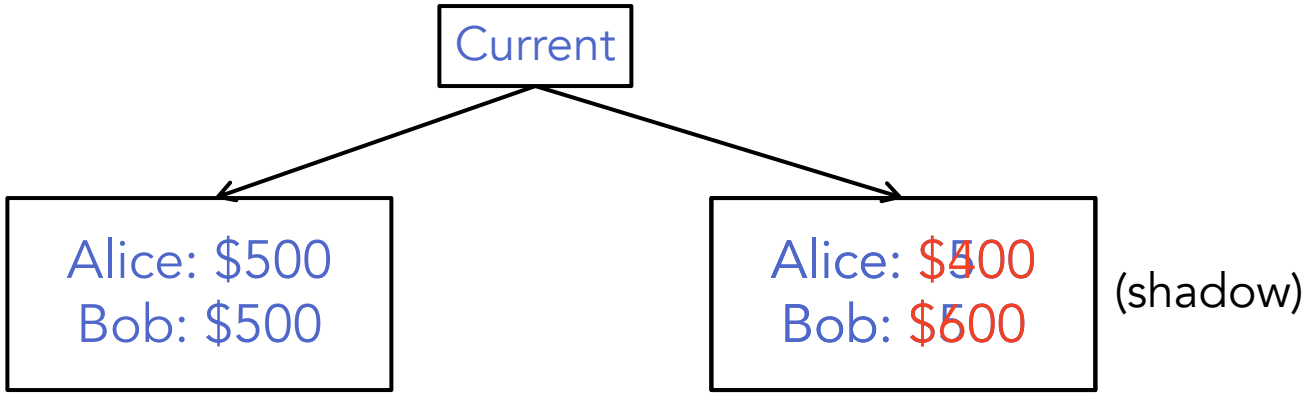


At beginning of transaction, both copies are identical

Shadowing

During transaction, write all new data to the backup (shadow)

At end of transaction, switch the pointer to point to the backup



A single-sector disk write commits multiple disk writes

- Indirection to the rescue (again)!

Shadowing summary

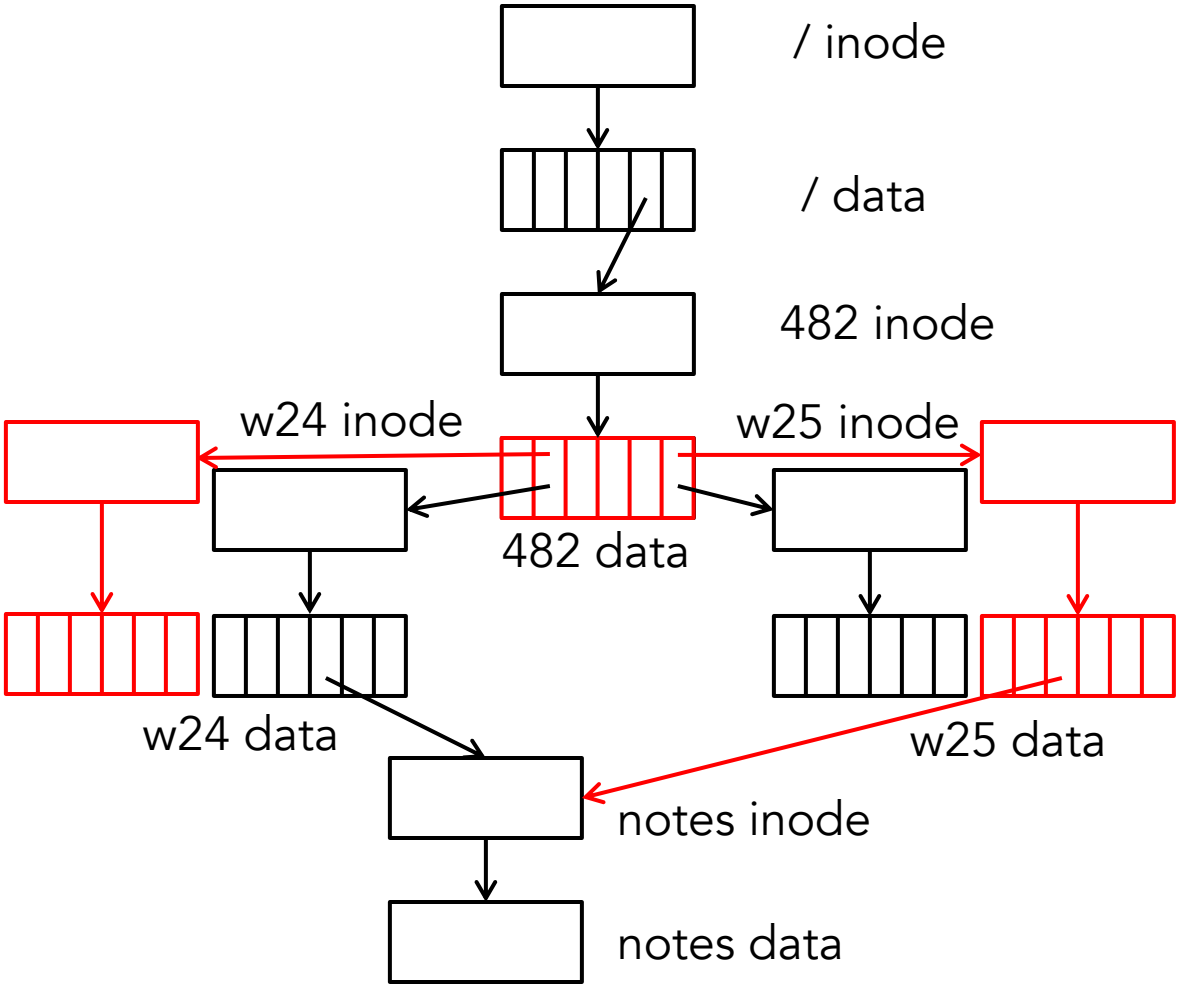
Makes an arbitrary set of updates atomic

Problems?

Reduce cost by shadowing on demand

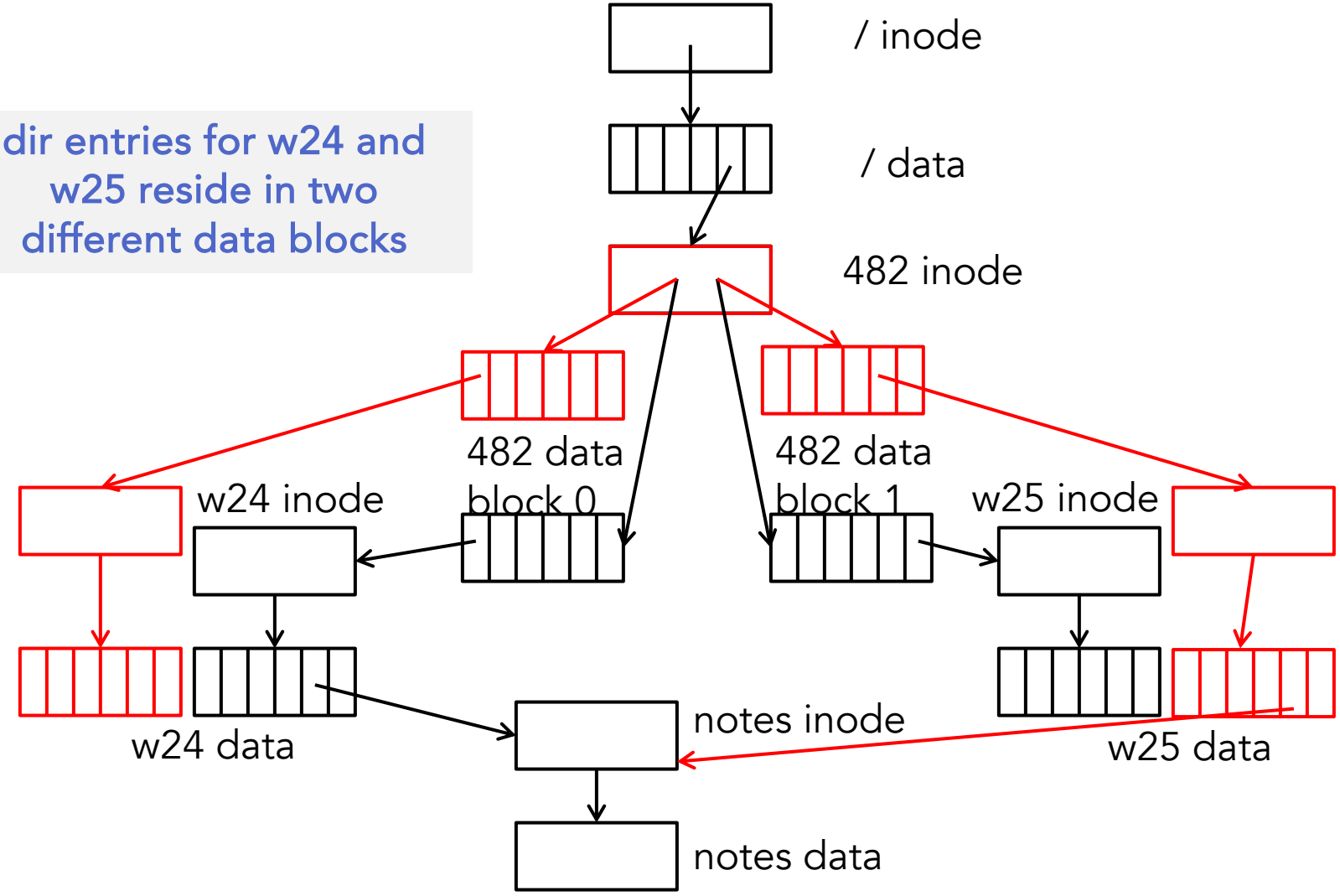
- Defer and avoid work!
- Leverages the ability to store more than a single pointer in a disk sector

Example: move `notes` from `/482/w24/` to `/482/w25/`



Example: move `notes` from `/482/w24/` to `/482/w25/`

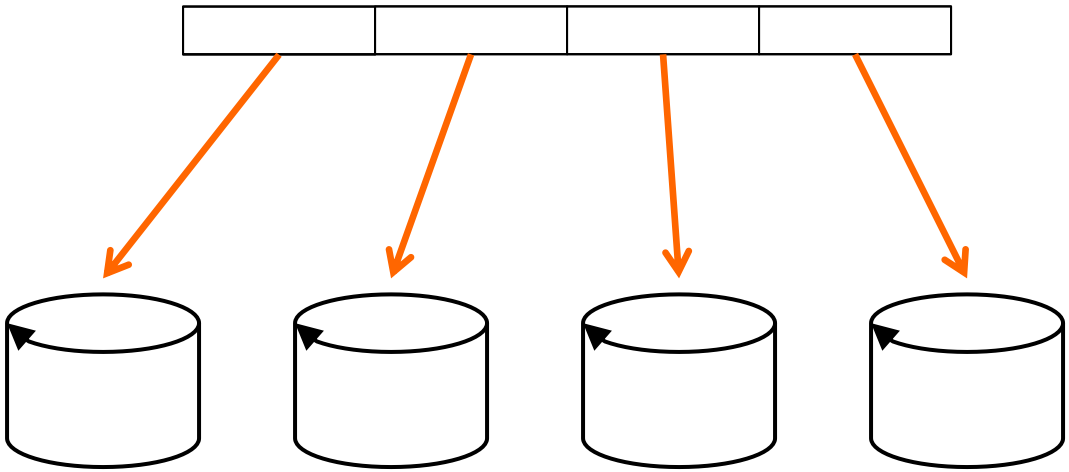
dir entries for w24 and w25 reside in two different data blocks



RAID

Parallelize I/O across multiple disks to increase storage bandwidth and improve reliability

- Files are striped across disks
- Each file portion is read/written in parallel
- Bandwidth increases with more disks (storage concurrency instead of CPU concurrency)



RAID reliability

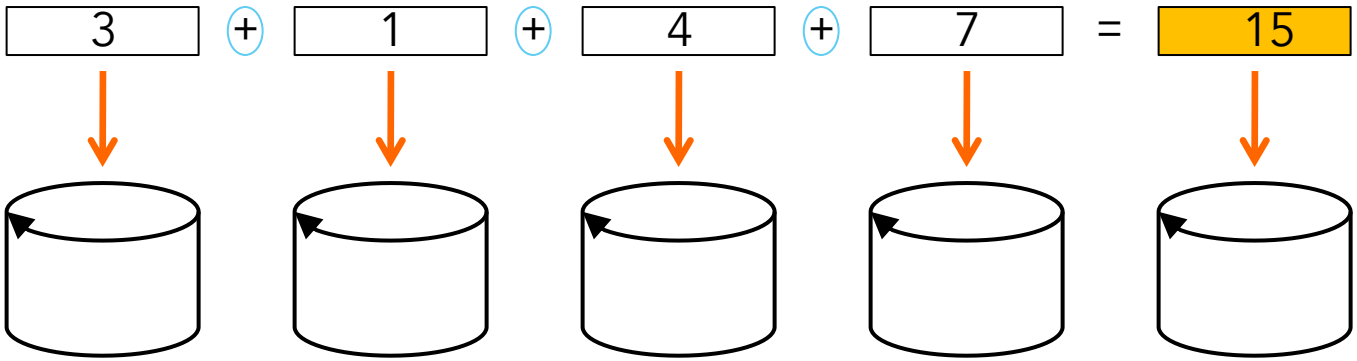
Using more disks increases probability of failure

- E.g., if a disk has 10% probability of failing in one year, and disks fail independently, what's the probability of at least 1 of the 4 disks disks failing in one year?

RAID with parity

Improve reliability via redundancy

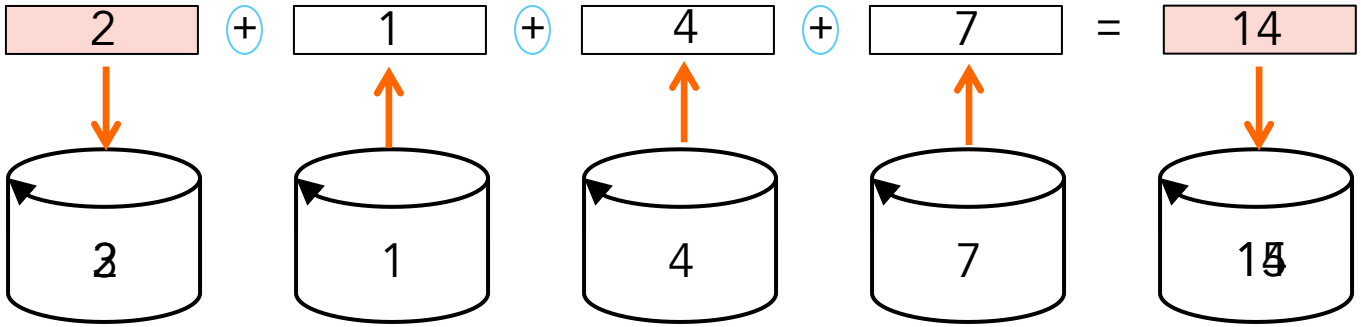
- For each set of data blocks, add one parity block, which stores parity (XOR) of other blocks.
- Add extra disk to store parity blocks.
- Can recover any data block from all others + parity block



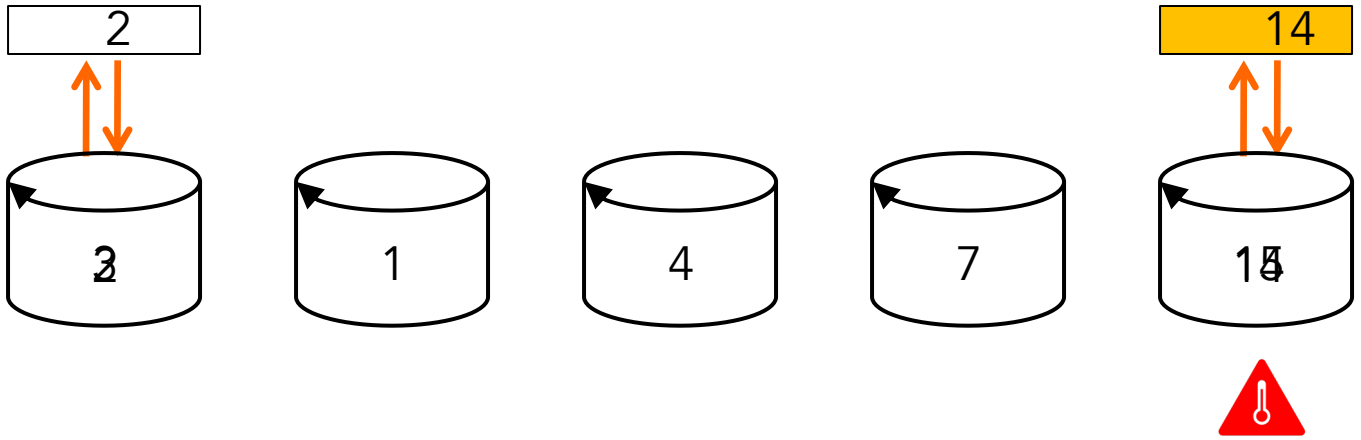
- Problems?

RAID with parity

How to write a partial stripe?



Can we reduce the disk I/O?



RAID levels

RAID 0: Striping

- Good performance but poor reliability

RAID 1: Mirroring

- Maintain full copy of all data
- Good read performance, but 100% overhead for storage and writes

RAID 5: Floating parity

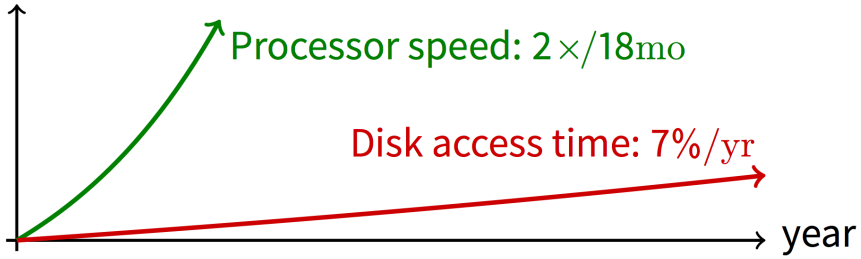
- Parity blocks for different stripes written to different disks
- No single parity disk → no bottleneck at that disk

Log-structured File System (LFS)

A classic example of system designs driven by technology trends

Motivation

- Faster CPUs: I/O becomes more and more of a bottleneck



- More memory: file cache is effective for reads
- Implication: writes compose most of disk traffic

Motivation

Problems with previous FS

- Perform many small writes
  - Good performance on large, sequential writes, but many writes are still small, random
- Synchronous operation to avoid data loss
- Depends upon knowledge of disk geometry
  - Fast File System

LFS idea

Insight: treat disk like a tape-drive

- Best performance from disk for sequential access

File system buffers writes in main memory until "enough" data

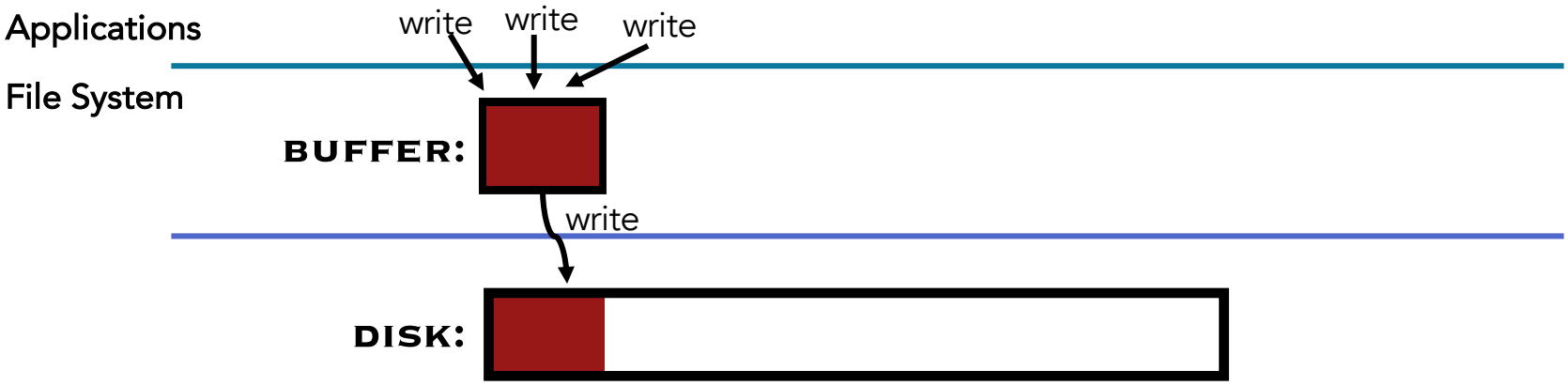
- How much is enough?
- Enough to get good sequential bandwidth from disk (MB)
- Unit called a "segment"

Write data to a sequential log

Write buffered data to a new segment on disk in a sequential log

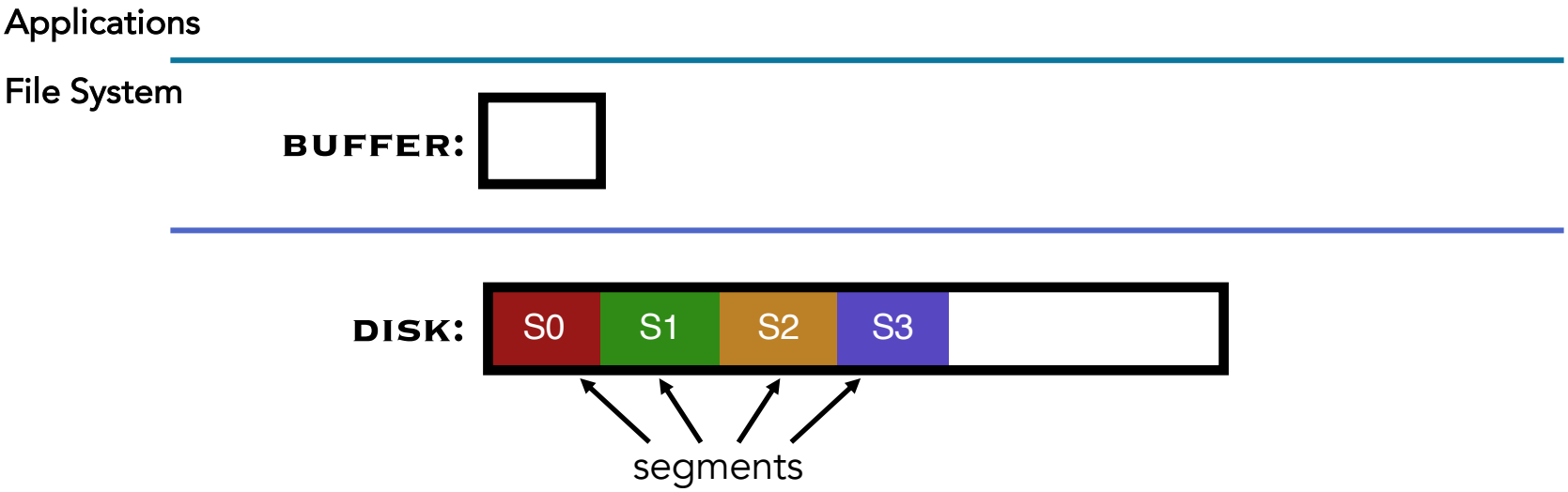
- Transfer all updates into a series of sequential writes
- Do not overwrite old data on disk
  - i.e., old copies left behind
- Write both data and metadata in one operation

Writes in LFS

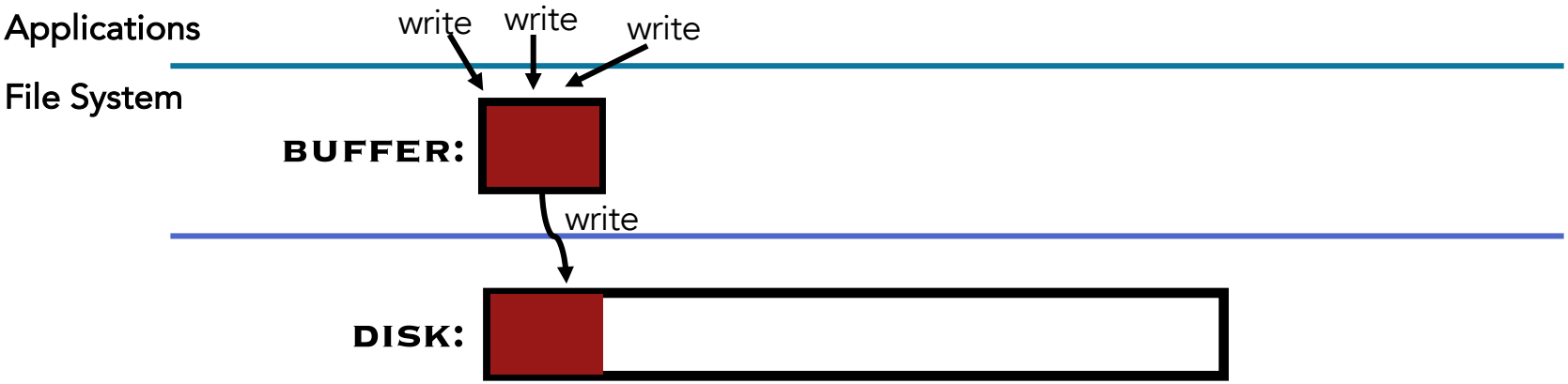


Absorb many small writes into one buffer write!

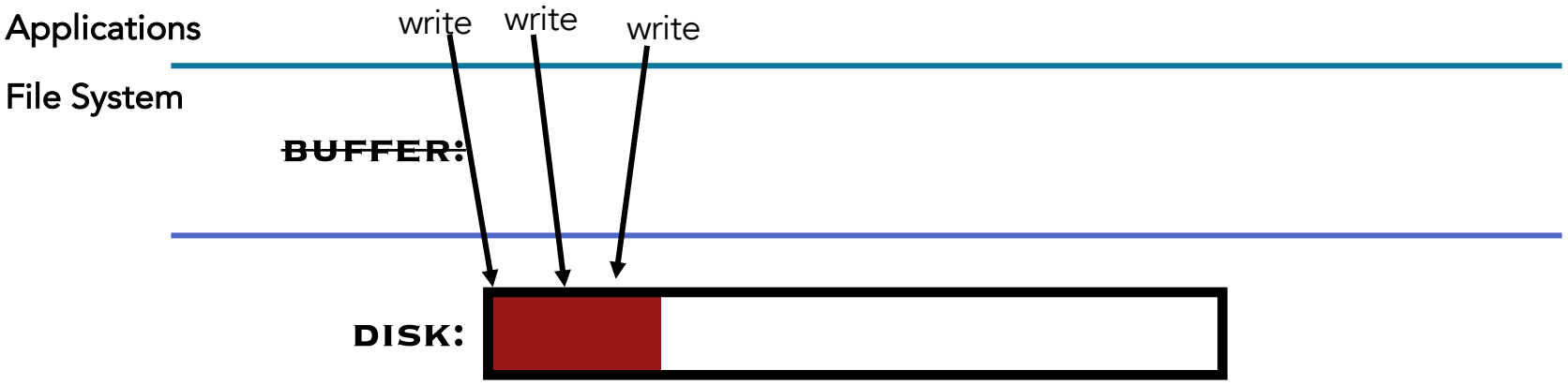
Writes in LFS



Why buffer the writes?



Why buffer the writes?



Why not directly write to the log on disk sequentially?

- Sequential write alone is not enough
- Disk is constantly rotating!
- Must issue a large number of contiguous writes

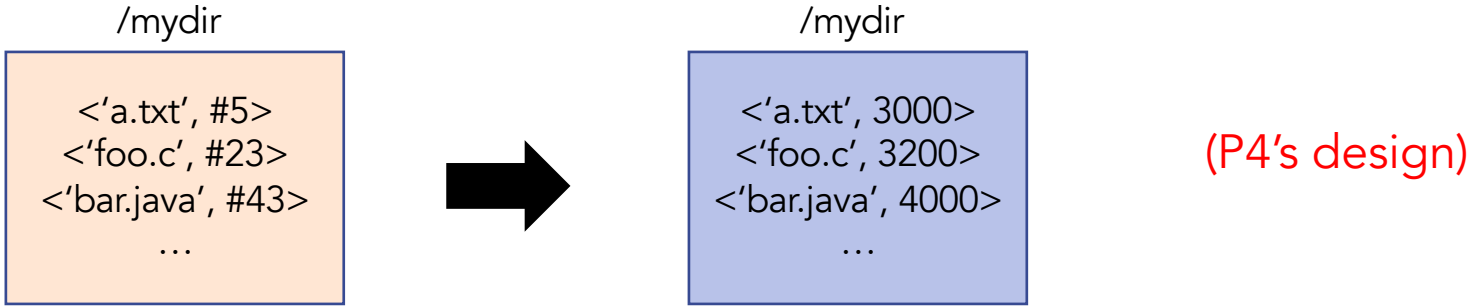
Data structures for LFS – attempt 1

What data structures from Unix FS can LFS remove?

- Allocation structs: data + inode bitmaps (why?)

What type of structure is more complicated?

- Inodes are no longer at fixed location on disk!
- Use offset on disk instead of array index for name in dir entries



Previously, each dir entry is <name, inode #>

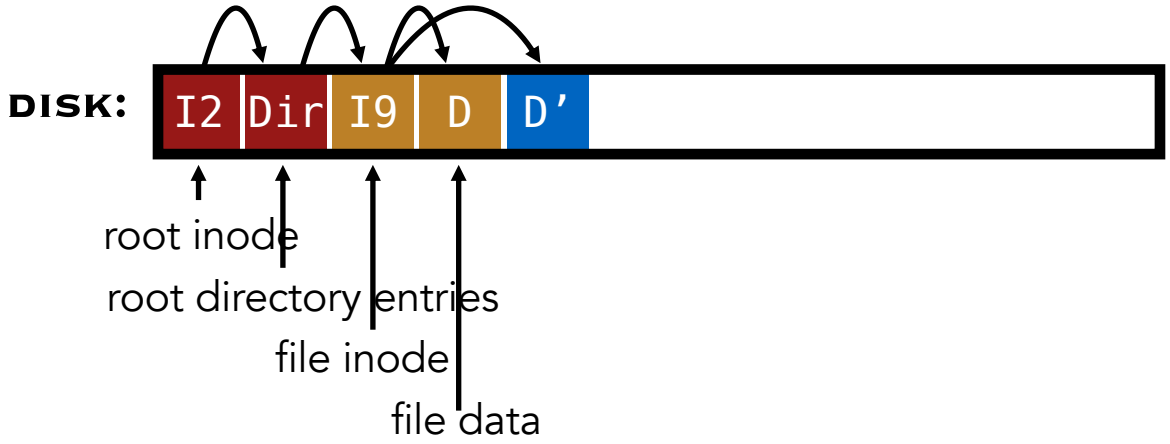
Now, each dir entry is <name, disk offset>

(P4's design)

- Would this attempt work?

Update data in LFS – attempt 1

Update data in /file.txt



Write updated data block D' for file.txt

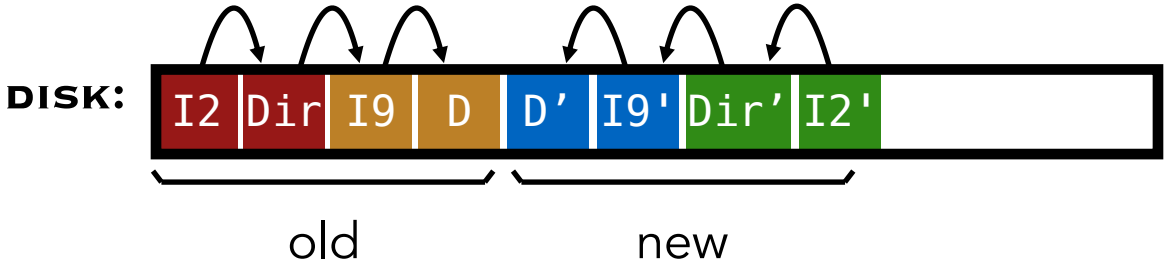
How to update inode 9 to point to new D' ?

Can LFS update inode 9 to point to new D'?

- No! This would be a random write..

Update data in LFS – attempt 1

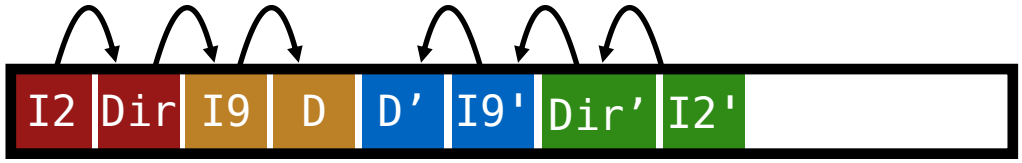
Update data in /file.txt



- Write updated data block
- Write updated inode for file
- Write updated dir entry for file
- Write updated inode for directory
- Repeat until root inode is updated...

Must update *all* structures in order to the log

Problem with attempt 1



Problem:

- For every data update, must propagate updates all the way up directory tree to root

Why?

- When we copy & modify the inode, its location changes
  - Must update dir entries (<name, disk offset>)

Solution:

- Keep inode numbers constant; don’t base name on offset

Data structures for LFS (attempt 2)

What data structures from Unix FS can LFS remove?

- Allocation structs: data + inode bitmaps

What type of structure is more complicated?

- Inodes are no longer at fixed location on disk!
- ~~Use offset on disk instead of array index for name in dir entries~~
- Keep inode number in dir constant
- Use an additional structure inode map, imap, to map inode number => most recent inode location on disk

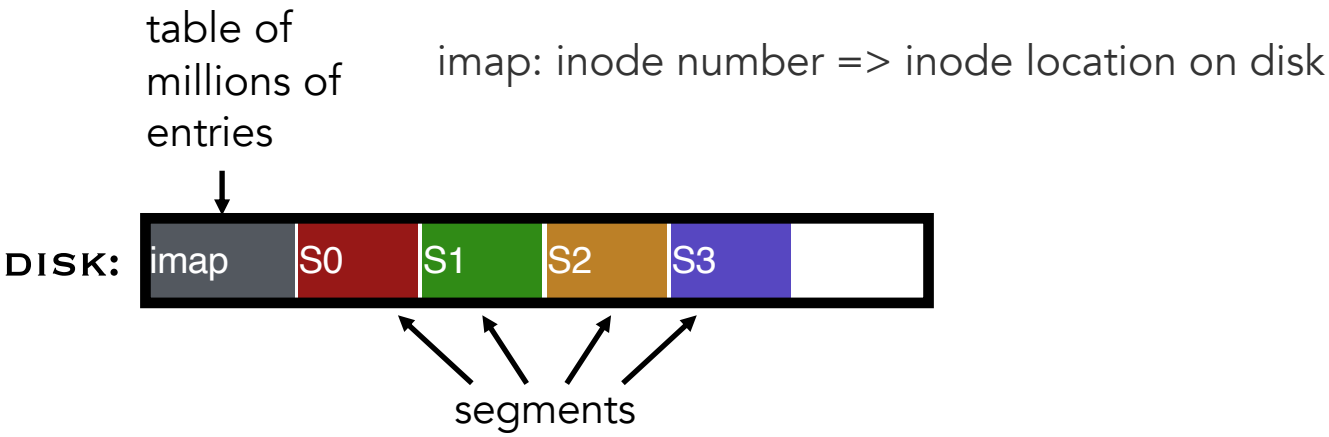
Original Unix FS finds an inode with math

- Start disk sector no. of inode array + inode #

How now?

- imap

Where to keep imap?



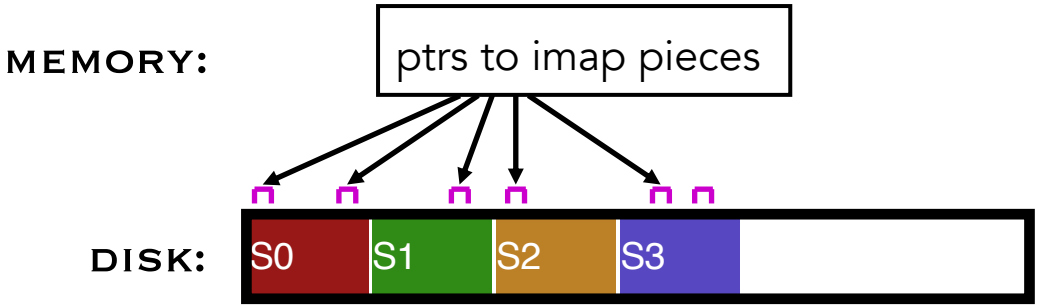
Where can imap be stored? Dilemma:

1. imap too large to keep in memory
2. don't want to perform random writes for imap

Solution: Write imap in segments

- Keep pointers to pieces of imap in memory

Solution: imap in segments



Solution:

- Write imap in segments
- Keep pointers to pieces of imap in memory
- Keep recent accesses to imap cached in memory
- Periodically write all imaps to a known checkpoint region on disk for crash recovery (and initial lookup)

Log cleaning

LFS append-only log can quickly run out of disk space

- Overwriting and deletion create garbage
- Need an efficient cleaning (garbage collection) process

LFS reclaims at segment granularity

- Tricky, since segments can be partly valid
- Copy & compact cleaning:
- Each segment keeps a segment summary block
  - Used to determine live blocks

Cleaning is expensive under high utilization