File systems: case studies

Hung-Wei Tseng

Outline

- BSD's Fast File System
- Log-structured File System

A Fast File System for UNIX

Marshall K. McKusick, William N. Joy, Samuel J. Leffler and Robert S. Fabry

Computer Systems Research Group

Why do we care about fast file system

- We want better performance!!!
- We want new features!

Let’s make file systems great again!

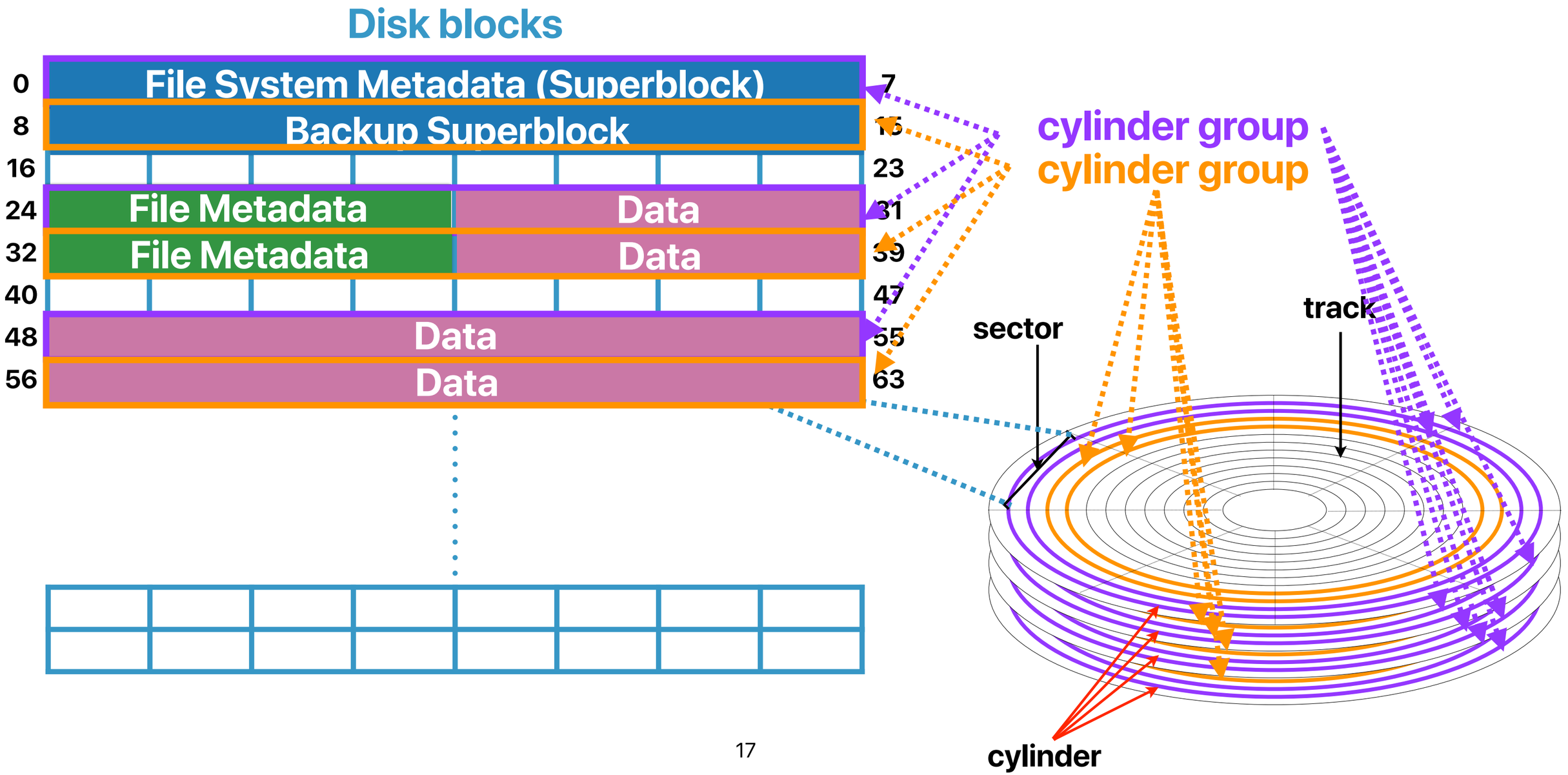
Problems in the “old” file system

- Lots of seeks when accessing a file
  - inodes are separated from data locations
  - data blocks belong to the same file can be spread out
- Low bandwidth utilization
  - only the very last is retrieving data
  - 1 out 11 in our previous example — less than 10% if files are small
- Limited file size
- Crash recovery
- Device oblivious

What does fast file system propose?

- Cylinder groups
- Larger block sizes
- Fragments
- Allocators
- New features
  - long file names
  - file locking
  - symbolic links
  - renaming
  - quotas

How FFS use disk blocks



Cylinder groups

- Consists of one or more consecutive cylinders on a disk
- Each cylinder group contains the following
  - redundant copy of the superblock
    - what's the benefit?
    - why not a cylinder group for all superblocks?
  - inode space
  - bitmap of free blocks within the cylinder group
  - summary of block usage
  - data
- Improves average disk access time
  - Allocating blocks within the same cylinder group for the same file
  - Placing inode along with data within the same cylinder group

Larger block sizes

- The block size of the old file system is aligned with the block (sector) size of the disk
  - Each file can only contain a fixed number of blocks
  - Cannot fully utilize the I/O interface bandwidth
- The new file system supports larger block sizes
  - Supports larger files
  - Each I/O request can carry more data to improve bandwidth
- However, larger block size leads to internal fragments

How larger block sizes improves bandwidth

- SATA II (300MB/s in theory), 7200 R.P.M., seek time around 8 ms. Assume the controller overhead is 0.2ms. What's the bandwidth of accessing 512B sectors and 4MB consecutive sectors?

Latency = seek time + rotational delay + transfer time + controller overhead

$$= 8\ ms + 4.17\ ms + 13.33\ ms + 0.2\ ms = 25.69\ ms$$

Bandwidth = volume_of_data over period_of_time

$$= \frac{4MB}{25.69ms} = 155.7\ MB/sec$$

Trading latencies with bandwidth

$$= 8\ ms + 4.17\ ms + 0.00167\ us + 0.2\ ms = 12.36\ ms$$

$$= \frac{0.5KB}{12.36ms} = 40.45KB/sec$$

Fragments

- Addressable units within a block
- Allocates fragments from a block with free fragments if the writing file content doesn’t fill up a block

Allocators

- Global allocators
  - Try to allocate inodes belong to same file together
  - Spread out directories across the disk to increase the successful rate of the previous
- Local allocators — allocate data blocks upon the request of the global allocator
  - Rotationally optimal block in the same cylinder
  - Allocate a block from the cylinder group if global allocator needs one
  - Search for blocks from other cylinder group if the current cylinder group is exhausted

Writes

- Larger overheads than the old file system as the new file system allocates blocks after write requests occur — Why not optimize for writes?
  - 10% of overall time in allocating blocks
  - writes are a lot faster already
- Writing metadata is synchronous rather than asynchronous — What's the benefit of synchronous writes?
  - Consistency

allocation [13]. This technique was not included because block allocation currently accounts for less than 10 percent of the time spent in a write system call and, once again, the current throughput rates are already limited by the speed of the available processors.

What does fast file system propose?

- Cylinder groups — improve spread-out data locations
- Larger block sizes — improve bandwidth and file sizes
- Fragments — improve low space utilization due to large blocks
- Allocators — address device oblivious
- New features
  - long file names
  - file locking
  - symbolic links
  - renaming
  - quotas

The design and implementation of a log-structured file system

Mendel Rosenbaum and John K. Ousterhout
Univ. of California, Berkeley

Why LFS?

- Small, random writes will dominate the traffic between main memory and disks — Unix FFS is designed under the assumption that large files are more important **UFS is published in 1984**
  - Who is wrong?
  - As system memory grows, frequently read data can be cached efficiently
  - Every modern OS aggressively caches — use "free" in Linux to check
  - Target environments are different
- Gaps between sequential access and random access
- Conventional file systems are not RAID aware

2. Design for file systems of the 1990's

This has two effects on file system behavior. First, larger file caches alter the workload presented to the disk by absorbing a greater fraction of the read requests[1, 6]. Most write requests must eventually be reflected on disk for safety, so disk traffic (and disk performance) will become more and more dominated by writes.

Workloads dominated by sequential accesses to large files, such as those found in supercomputing environments, also pose interesting problems, but not for file system software. A number of techniques exist for ensuring that such files are laid out sequentially on disk, so I/O performance tends to be limited by the bandwidth of the I/O and memory subsystems rather than the file allocation policies.

Why LFS?

- How many of the following problems is/are Log-structured file systems trying address?
  - ① The performance of small random writes
  - ② The efficiency of large file accesses
  - ③ The space overhead of metadata in the file system
  - ④ Reduce the main memory space used by the file system

  A. 0
  B. 1
  C. 2
  D. 3
  E. 4

Problems with BSD FFS

- Data are spread out the whole disk
  - Can achieve sequential access within each file, but the distance between files can be far
  - An inode needs a standalone I/O in addition to file content
  - Creating files take at least five I/Os with seeks — can only use 5% bandwidth for data
    - 2 for file attributes
      - You have to check if the file exists or not
      - You have to update after creating the file
    - 1 for file data
    - 1 for directory data
    - 1 for directory attributes
- Writes to metadata are synchronous
  - Good for crash recovery, bad for performance

gies and workloads of the 1990's. First, they spread information around the disk in a way that causes too many small accesses. For example, the Berkeley Unix fast file system (Unix FFS)[9] is quite effective at laying out each file sequentially on disk, but it physically separates different files. Furthermore, the attributes ("inode") for a file are separate from the file's contents, as is the directory entry containing the file's name. It takes at least five separate disk I/Os, each preceded by a seek, to create a new file in Unix FFS: two different accesses to the file's attributes

The second problem with current file systems is that they tend to write synchronously: the application must wait for the write to complete, rather than continuing while the write is handled in the background. For example even

What does LFS propose?

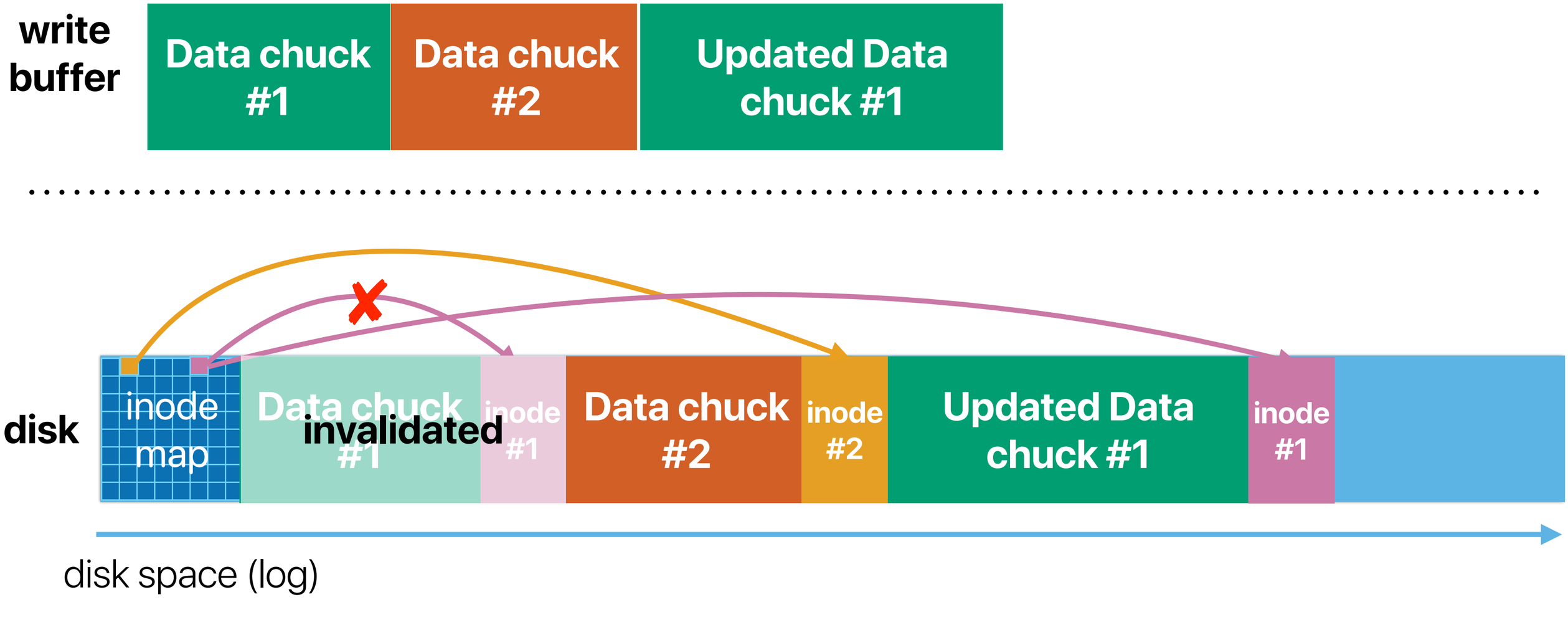
- Buffering changes in the system main memory and commit those changes sequentially to the disk with fewest amount of write operations

Three components of technology are particularly significant for file system design: processors, disks, and main memory. Processors are significant because their speed is increasing at a nearly exponential rate, and the improvements seem likely to continue through much of the 1990's. This puts pressure on all the other elements of the computer system to speed up as well, so that the system doesn't become unbalanced.

Disk technology is also improving rapidly, but the improvements have been primarily in the areas of cost and capacity rather than performance. There are two components of disk performance: transfer bandwidth and access time. Although both of these factors are improving, the rate of improvement is much slower than for CPU speed. Disk transfer bandwidth can be improved substantially with the use of disk arrays and parallel-head disks[5] but no major improvements seem likely for access time (it is determined by mechanical motions that are hard to improve). If an application causes a sequence of small disk transfers separated by seeks, then the application is not likely to experience much speedup over the next ten years, even with faster processors.

The third component of technology is main memory, which is increasing in size at an exponential rate. Modern file systems cache recently-used file data in main memory, and larger main memories make larger file caches possible.

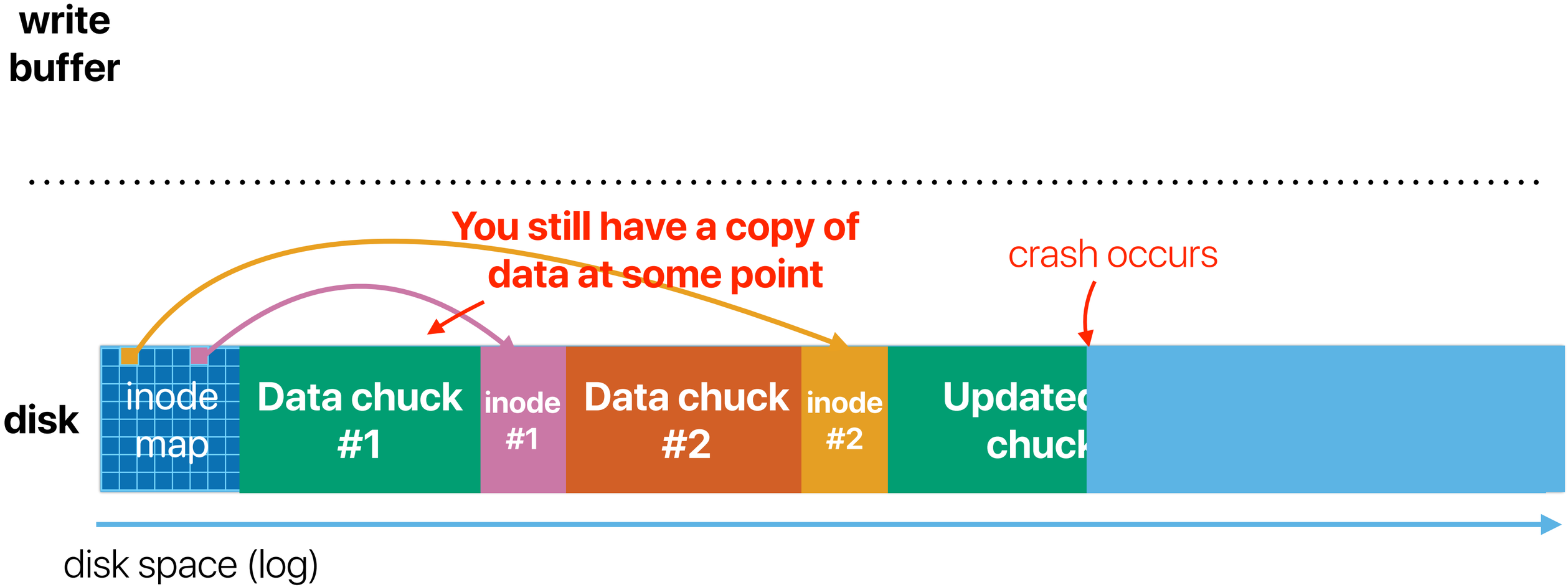
LFS in motion



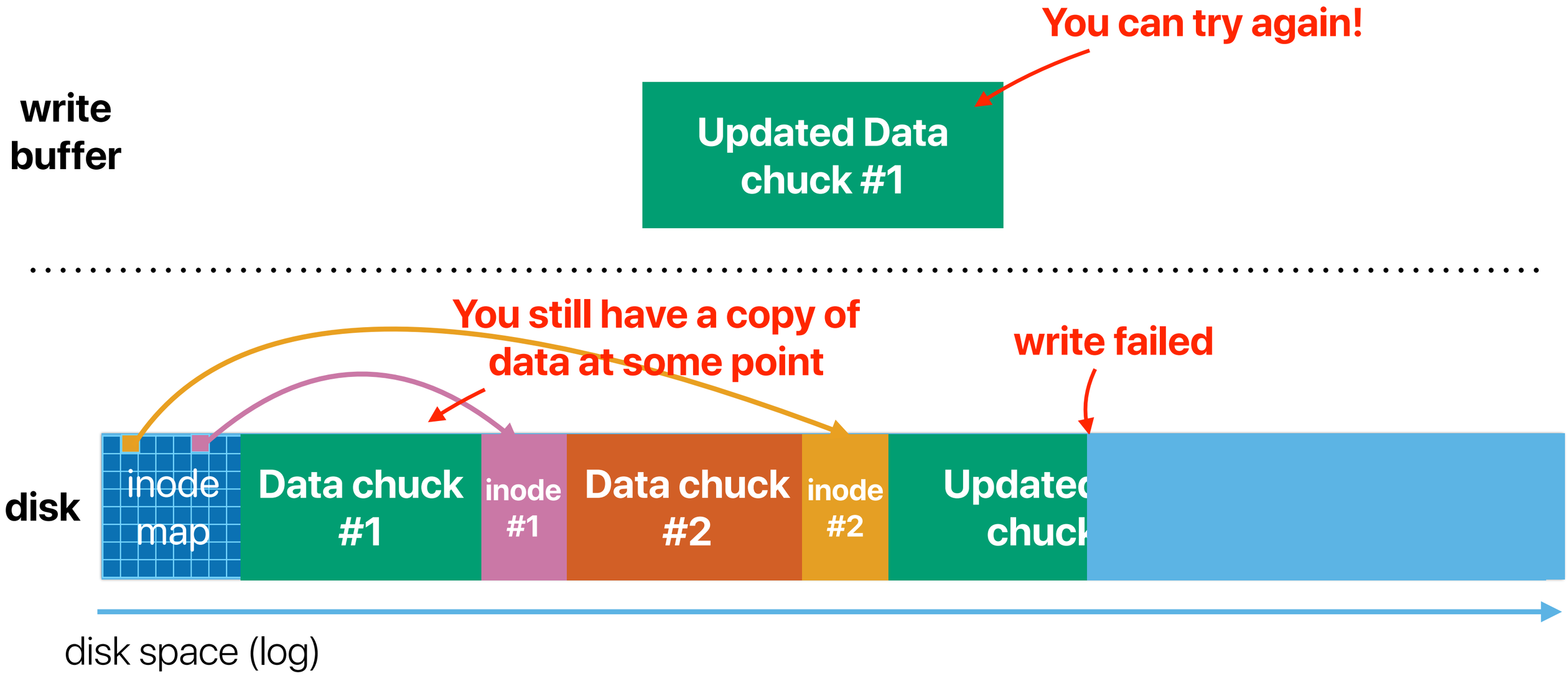
Crash recovery

- Checkpointing
  - Create a redundant copy of important file system metadata periodically
- Roll-forward
  - Scan through/replay the log after checkpointing

LFS v.s. crash


write buffer
You still have a copy of data at some point
crash occurs
disk
inode map
Data chuck #1
inode #1
Data chuck #2
inode #2
Updated chuck
disk space (log)

LFS v.s. write failed



Segment cleaning/Garbage collection

- Reclaim invalidated segments in the log once the latest updates are checkpointed
- Rearrange the data allocation to make continuous segments
- Must reserve enough space on the disk
  - Otherwise, every writes will trigger garbage collection
  - Sink the write performance

Modern file system design — Extent File Systems

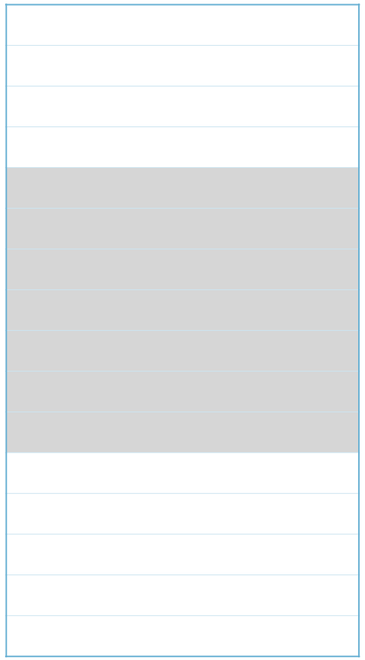
Extent file systems — ext2, ext3, ext4

- Basically optimizations over FFS + Extent + Journaling (write-ahead logs)

How do we allocate space?

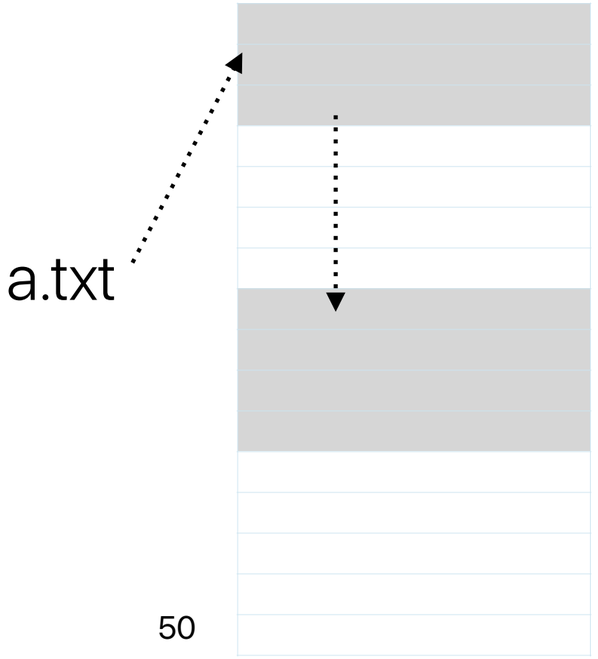
- Contiguous: the file resides in continuous addresses

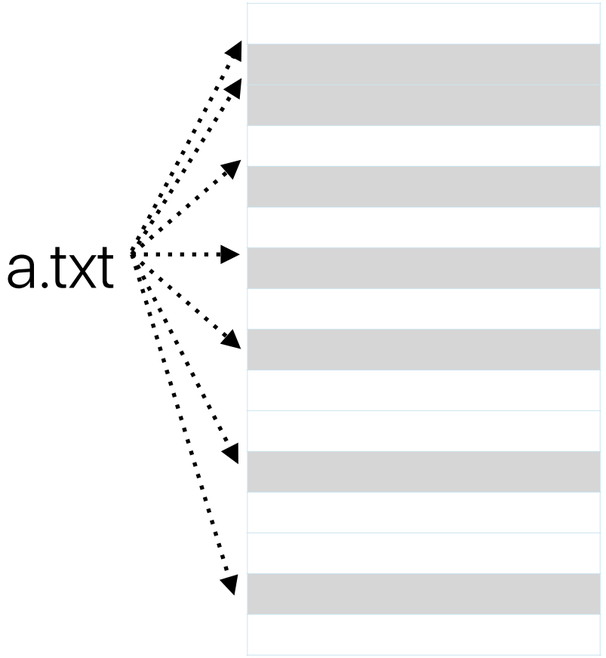
a.txt



- Non-contiguous: the file can be anywhere

- Extents: the file resides in several group of smaller continuous address

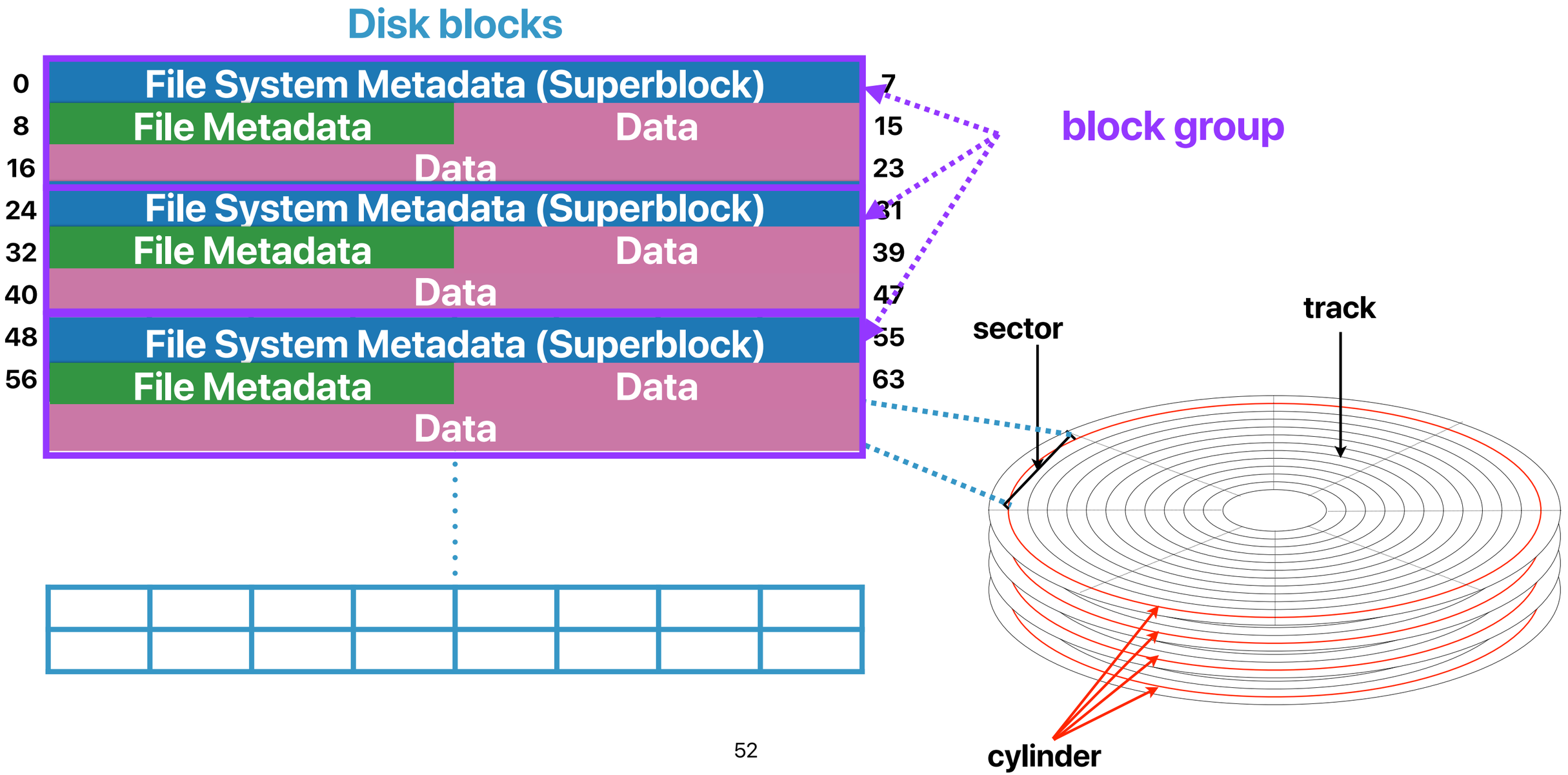




Using extents in inodes

- Contiguous blocks only need a pair <start, size> to represent
- Improve random seek performance
- Save inode sizes
- Encourage the file system to use contiguous space allocation

How ExtFS use disk blocks



Write-ahead log

- Basically, an idea borrowed from LFS to facilitate writes and crash recovery
- Write to log first, apply the change after the log transaction commits
  - Update the real data block after the log writes are done
  - Invalidate the log entry if the data is presented in the target location
  - Replay the log when crash occurs