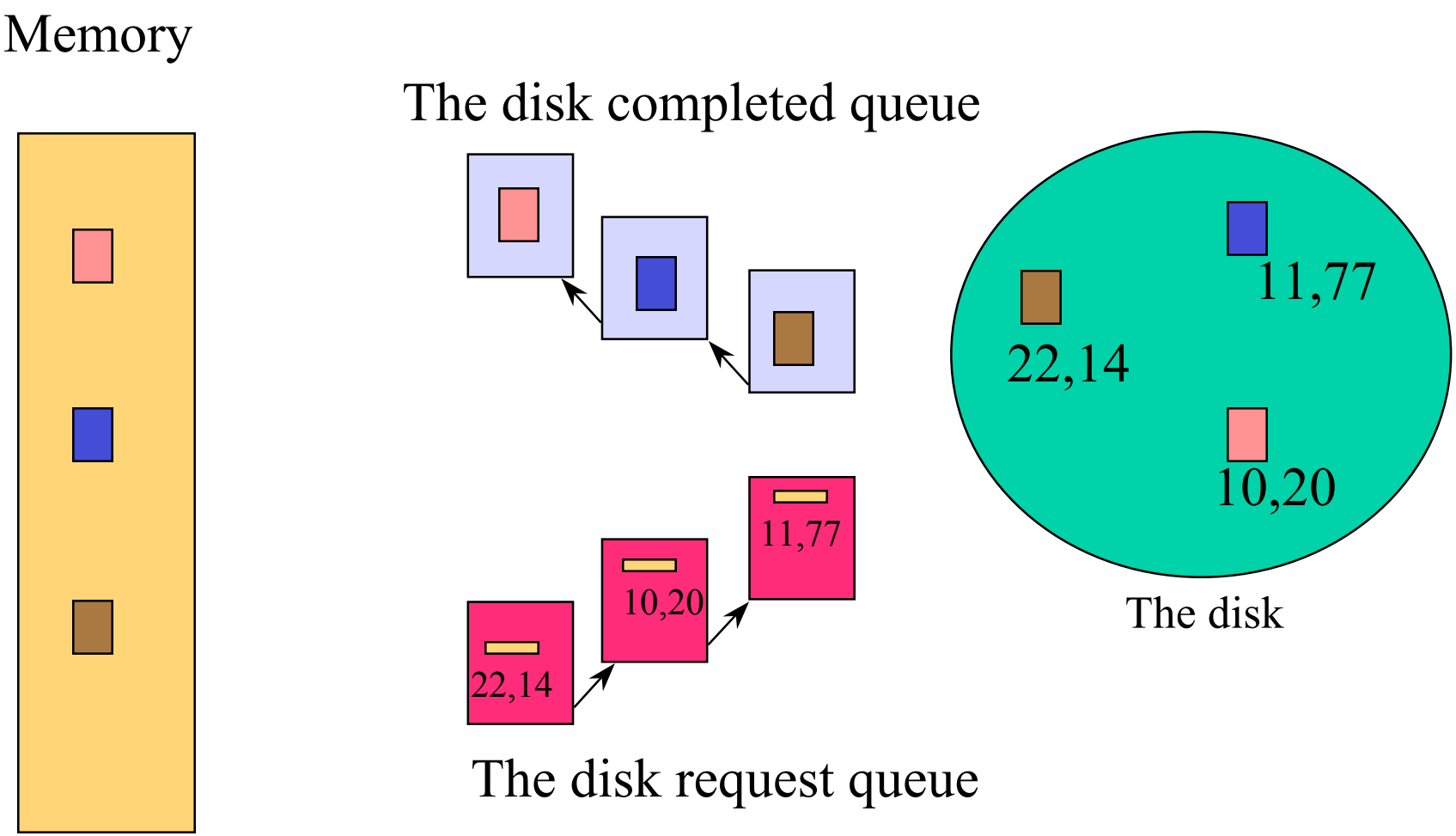
File System Implementation Issues

The Operating System's View of the Disk



Dealing with Mechanical Latencies

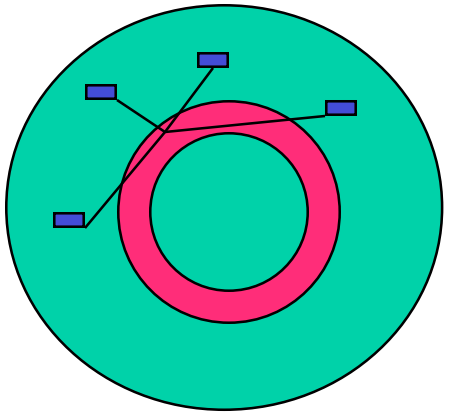
- Scheduling algorithms
  - head scheduling
- Layout
  - meta-information layout
- Caches
  - locality in file access
- RAID
  - parallelism

Disk Scheduling

- Because disks are slow and seeks are long and depend on distance, we can schedule disk accesses, e.g.:
    - FCFS (do nothing)
        - ok when load is low
        - long waiting times for long request queue
    - SSTF (shortest seek time first)
        - always minimize arm movement. maximize throughput.
        - favors middle blocks
    - SCAN (elevator) -- continue in same direction until done, then reverse direction and service in that order
    - C-SCAN -- like scan, but go back to 0 at end
- In general, unless there are request queues, it doesn't matter
    - explains why some single user systems do badly under heavy load.
- The OS (or database system) may locate files strategically for performance reasons.

Disk Structure

- There is no structure to a disk except cylinders (sets of tracks at same distance from center) and sectors, anything else is up to the OS.
- The OS imposes some structure on disks.
- Each disk contains:
    - 1. data: e.g., user files
    - 2. meta-data: OS info describing the disk structure
- For example, the free list is a data structure indicating which disk blocks are free. It is stored on disk (usually) as a bit map: each bit corresponds to one disk block.
- The OS may keep the free list bit map in memory and write it back to disk from time to time.

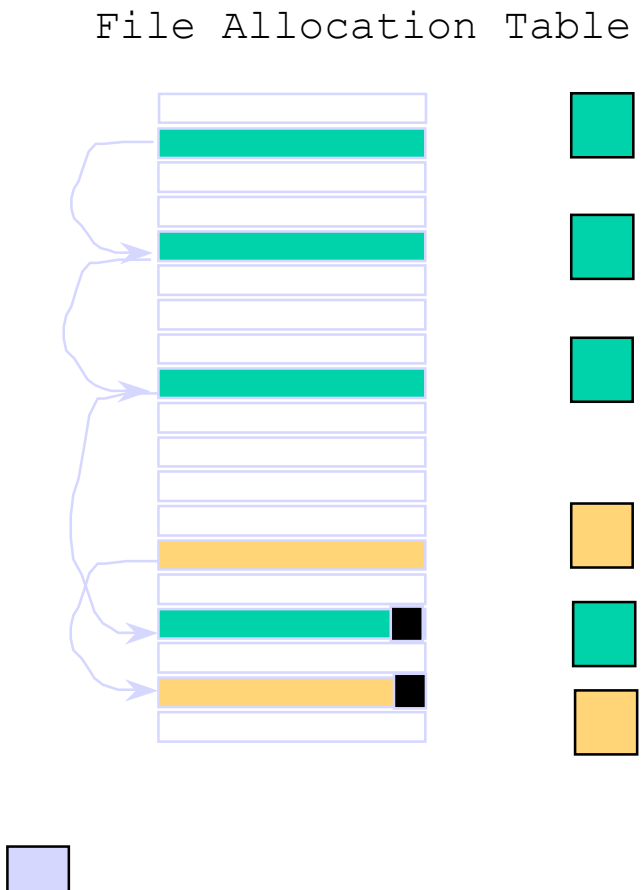


Disk Layout Strategies

- Files can be allocated on disk in different ways, e.g.:
  - 1. contiguous allocation
    - like memory
    - fast and simplifies directory access
    - inflexible, causes fragmentation, needs compaction
  - 2. linked structure
    - each block points to next block, directory points to first
    - good for sequential access (bad otherwise)
  - 3. indexed structure
    - an “index block” contains pointers to many other blocks
    - better for random access
    - may need multiple index blocks (linked together)

DOS FAT

- One file allocation table describes the layout of the entire disk.
- Each entry in the table refers to a specific cluster within a file.
  - zero says cluster not used.
  - not zero says where the next FAT entry for the file is.
- A file's directory entry points to the first FAT entry for the file.



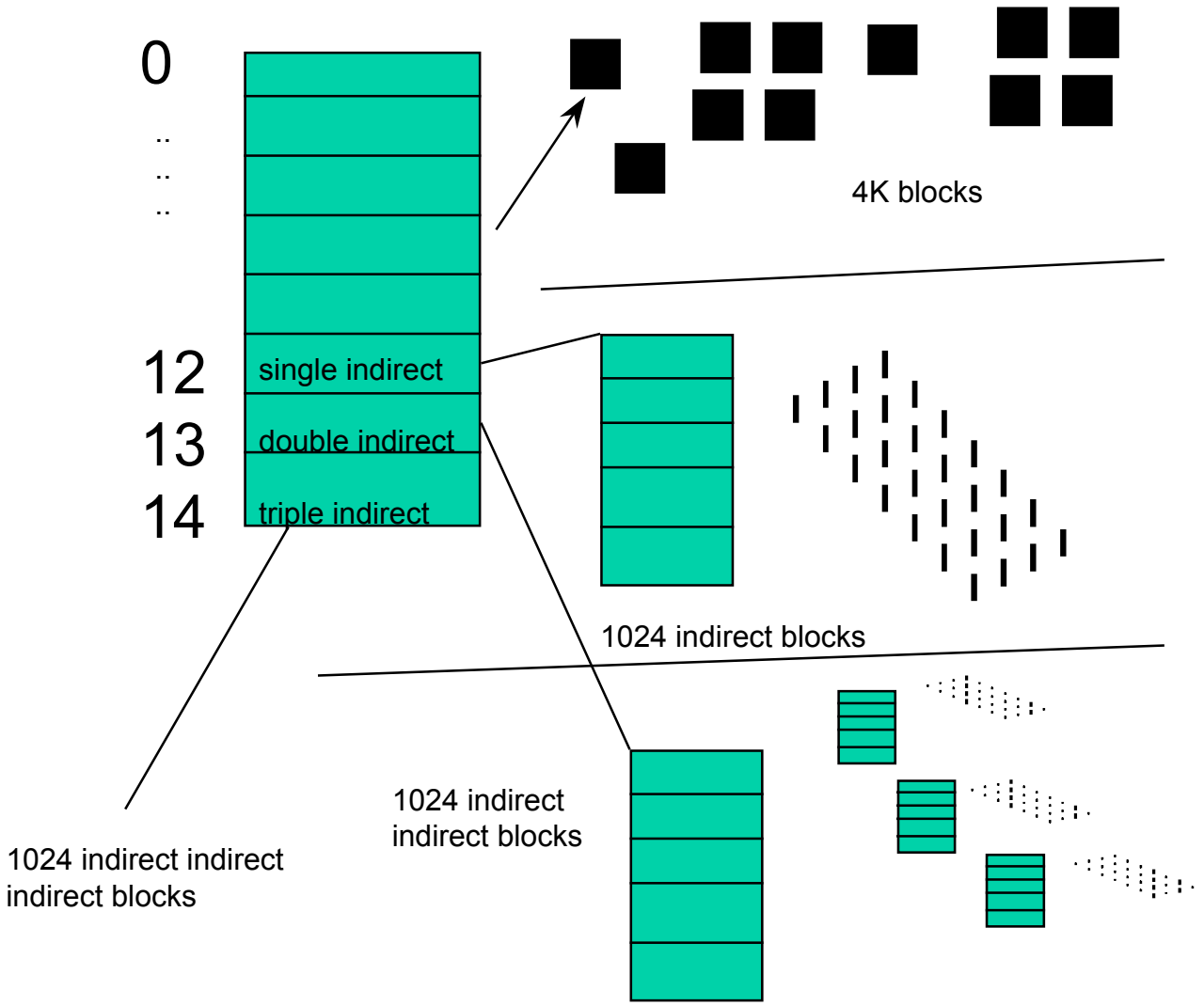
The FAT is on disk in a special sequence blocks.

Limitations of FAT

- FAT index is 16 bits.
  - 1 disk can have up 64K clusters.
- As disks get bigger, cluster size must increase.
  - eg, 16KB cluster for a 640MB disk.
- Big clusters yield internal fragmentation.
  - 10 to 20% wastage for 16KB clusters not uncommon.
- Minimum of one file per cluster.
  - limitation to 64K files.
- The FAT itself is a critical resource.
  - You lose the FAT on disk, you've lost the whole disk.

Eg, UNIX Inodes

- A UNIX inode is the metainformation for UNIX files.
- Contains control and allocation information.
- Each inode contains 15 block pointers.
  - first 11 are direct blocks
  - then single, double, and triple indirect



Inodes and Path Search

- Unix Inodes are NOT directories
  - they describe where on disk the blocks for a file are placed
    - directories are just files, so each directory also has an inode that describes where the blocks for the directory is placed
- Directory entries map file names to inodes
  - to open “/one”, use master block to find inode for “/” on disk
    - open “/”, look for entry for “one”
    - this gives the disk block number for inode of “one”
  - read the inode for “one” into memory
    - this inode says where the first data block is on disk
    - read that data block into memory to access the data in the file

Data and Inode placement

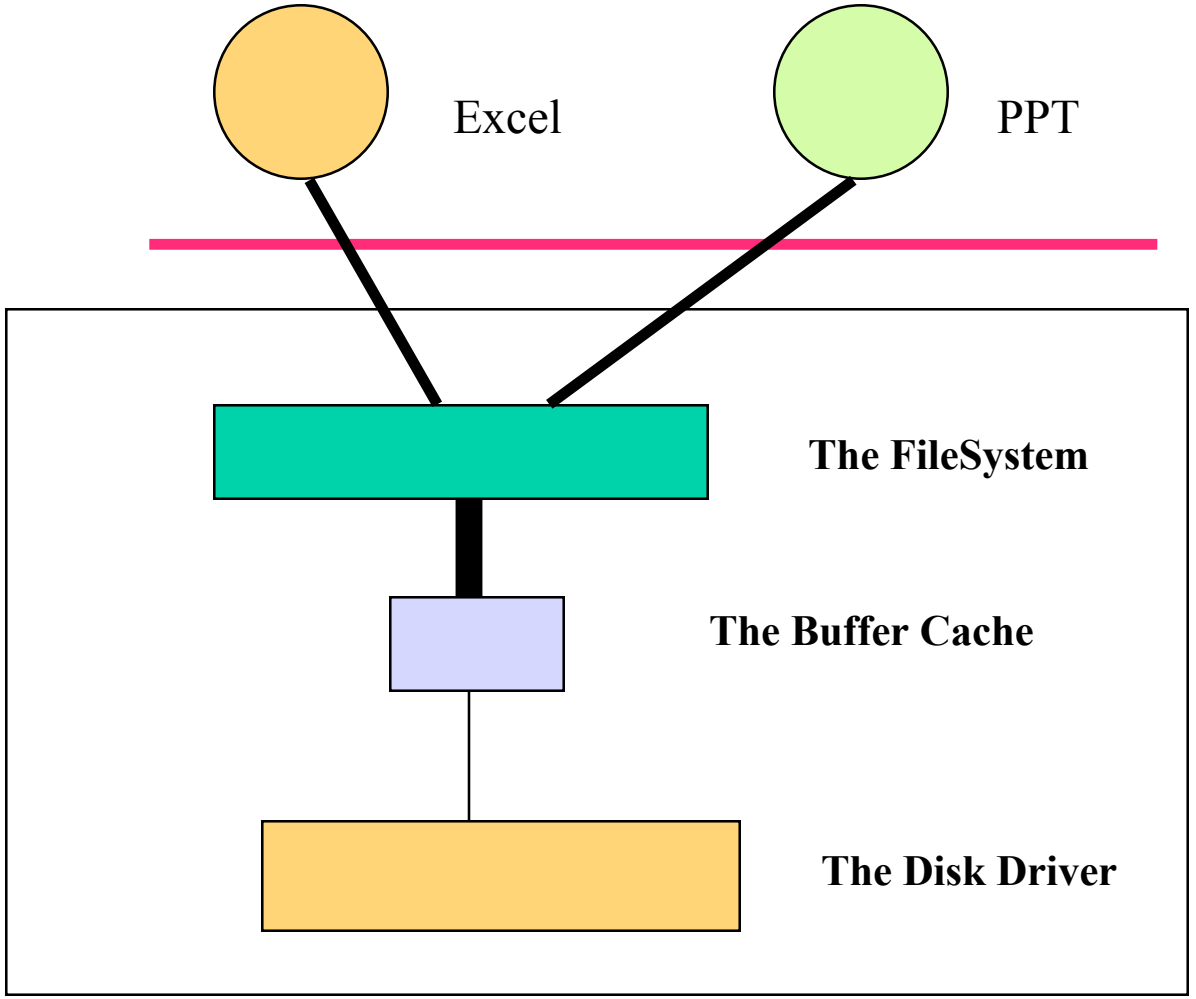
- Original (non-FFS) unix FS had two major problems:
  - 1. data blocks are allocated randomly in aging file systems
    - blocks for the same file allocated sequentially when FS is new
    - as FS “ages” and fills, need to allocate blocks freed up when other files are deleted
      - problem: deleted files are essentially randomly placed
      - so, blocks for new files become scattered across the disk!
  - 2. inodes are allocated far from blocks
    - all inodes at beginning of disk, far from data
    - traversing file name paths, manipulating files, directories requires going back and forth from inodes to data blocks
  - BOTH of these generate many long seeks!

Cylinder groups

- FFS addressed these problems using notion of a cylinder group
  - disk partitioned into groups of cylinders
  - data blocks from a file all placed in same cylinder group
  - files in same directory placed in same cylinder group
  - inode for file in same cylinder group as file's data
- Introduces a free space requirement
  - to be able to allocate according to cylinder group, the disk must have free space scattered across all cylinders
  - in FFS, 10% of the disk is reserved just for this purpose!
    - good insight: keep disk partially free at all times!
    - this is why it may be possible for df to report >100%

Disk Caching

- The idea is that data you (or someone) accessed recently is likely to be data that you (or someone) needs again.

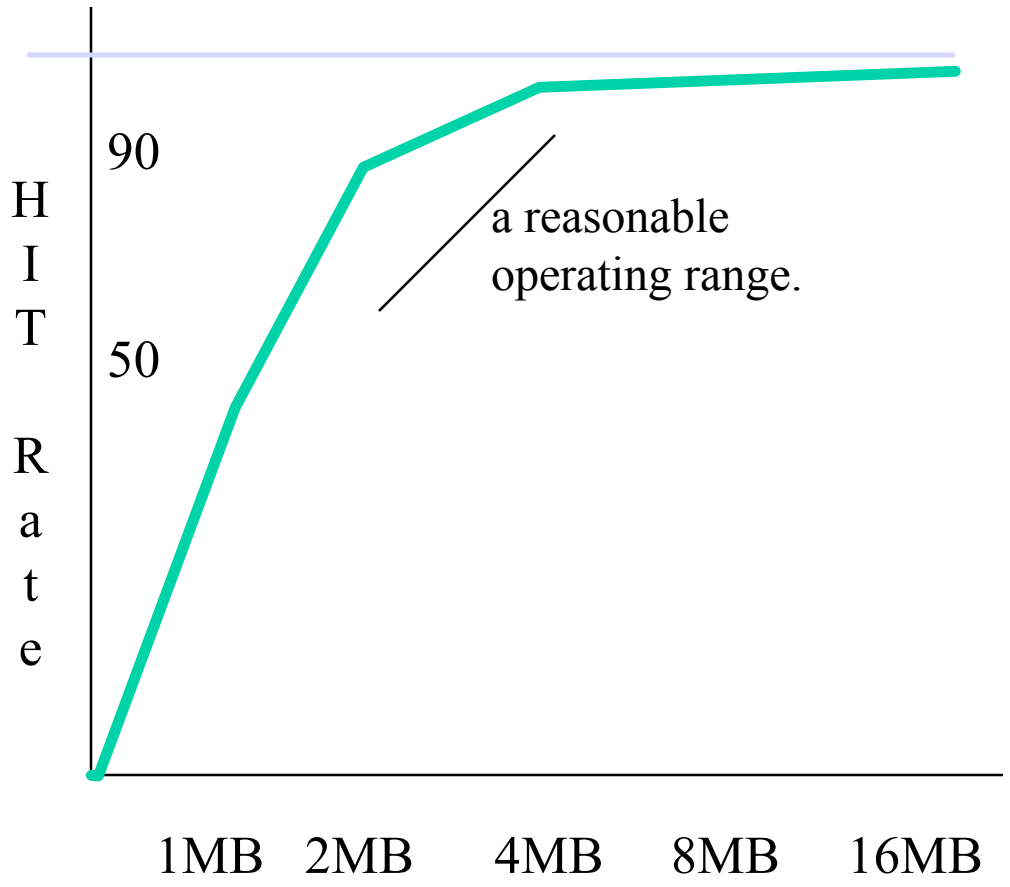


Disk Locality Works Quite Well

Most studies have shown that a disk cache on the order of 2-4 megabytes captures 90% of the read traffic for many kinds of applications

- your mileage will definitely vary
- but the impact can be enormous
  - 90% hit rate
  - hit time of .1 ms
  - miss time of 10ms
  - average access time

    (.9 * .1) + (.1 * 10) = 1.09 ms
- No cache increases disk access time by almost 1000%.



The Buffer Cache

- The buffer cache has only a finite number of blocks in it.
- On a cache miss, one block must be discarded in favor of the fresh block that is being brought in.
- Typical systems operate in an LRU order
  - the least recently used block is the one that should be discarded next.
  - favors “busier” portions of the filesystem
  - can hurt “the quiet” client.
- Some workloads LRU is really bad for
  - sequential scans
    - video, audio
  - random access
    - large database

Read Ahead

- Many file systems will implement “read ahead” whereby the file system will try to predict what the application is going to need next, go to disk, and read it in.
- The goal is to beat the application to the punch.
- For sequentially accessed files, this can be a big win.
  - as long as reads are spaced in time, the OS can schedule the next IO during the current gap.
  - enabling read ahead can give applications a 100% speedup.
    - suppose that it takes 10ms to read a block.
    - the application requests the next block in sequence every 10 ms.
    - with read ahead, wait time is 0 ms per block and execution time is
      - 10 * number of blocks read.
    - without read ahead, wait time is 20 ms per block and execution time is
      - 20 * number of blocks read.

Caching works for reads, what about writes?

- On write, it is necessary to ensure that the data makes it through the buffer cache and onto the disk.
- Consequently, writes, even with caching, can be slow.
- Systems do several things to compensate for this
  - "write-behind"
    - maintain a queue of uncommitted blocks
    - periodically flush the queue to disk
    - unreliable
  - battery backed up RAM
    - as with write-behind, but maintain the queue in battery backed up memory
    - expensive
  - log structured filed system
    - always write the next block on disk the one past where the last block was written
    - Treat the disk like a tape
      - Complicated to get right (always need a fresh 'tape')

Log-Structured File System (LFS)

- LFS was designed in response to two trends in workload and disk technology:
  - 1. Disk bandwidth scaling significantly (40% a year)
    - but, latency is not
  - 2. Large main memories in machines
    - therefore, large buffer caches
      - absorb large fraction of read requests in caches
    - can use for writes as well
      - coalesce small writes into large writes
- LFS takes advantage of both to increase FS performance

FFS problems that LFS solves

- FFS: placement improved, but can still have many small seeks
  - possibly related files are physically separated
  - inodes separated from files (small seeks)
  - directory entries separate from inodes
- FFS: metadata required synchronous writes
  - with small files, most writes are to metadata
  - synchronous writes are very slow!

LFS: The Basic Idea

- Treat the entire disk as a single log for appending
  - collect writes in the disk buffer cache, and write out the entire collection of writes in one large request
    - leverages disk bandwith with large sequential write
    - no seeks at all! (assuming head at end of log)
  - all info written to disk is appended to log
    - data blocks, attributes, inodes, directories, .etc.
- Sounds simple!
  - but it's really complicated under the covers

LFS Challenges

- There are two main challenges with LFS:
  - 1. locating data written in the log
    - FFS places files in a well-known location, LFS writes data “at the end of the log”
  - 2. managing free space on the disk
    - disk is finite, and therefore log must be finite
    - cannot always append to log!
      - need to recover deleted blocks in old part of log
      - need to fill holes created by recovered blocks

LFS: locating data

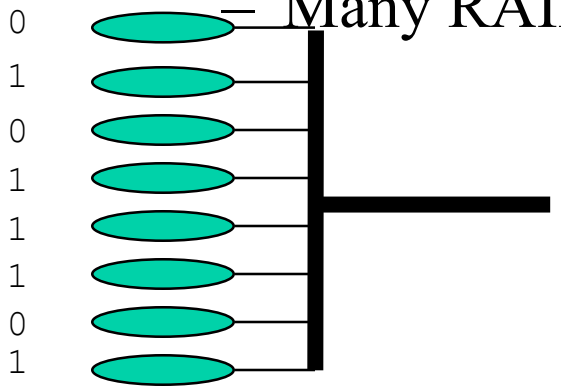
- FFS uses inodes to locate data blocks
  - inodes preallocated in each cylinder group
  - directories contain locations of inodes
- LFS appends inodes to end of log, just like data
  - makes them hard to find
- Solution:
  - use another level of indirection: inode maps
  - inode maps map file #s to inode location
  - location of inode map blocks are kept in a checkpoint region
  - checkpoint region has a fixed location
  - cache inode maps in memory for performance

LFS: free space management

- LFS: append-only quickly eats up all disk space
  - need to recover deleted blocks
- Solution:
  - fragment log into segments
  - thread segments on disk
    - segments can be anywhere
  - reclaim space by cleaning segments
    - read segment
    - copy live data to end of log
    - now have free segment you can reuse!
  - cleaning is a big problem
    - costly overhead, when do you do it?
      - "idleness is not sloth"

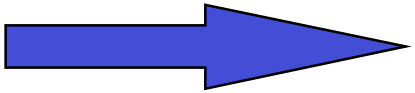
RAID

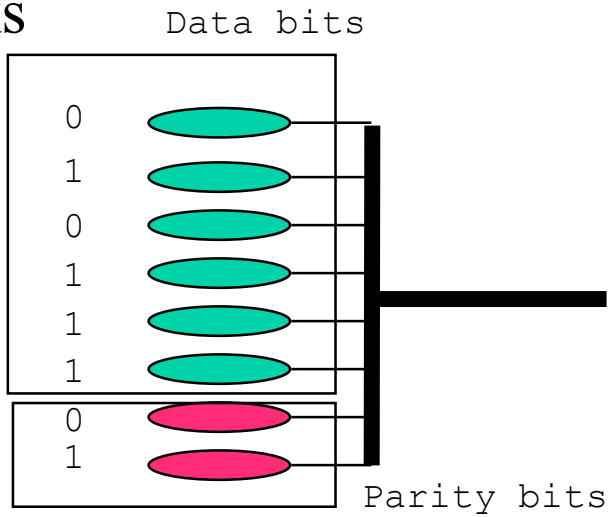
- Caching, RAM disks deal with the latency issue.
- DISKS can also be used in PARALLEL
- This is the idea behind RAIDs
  - Redundant Array of Inexpensive Disks
  - Many RAID levels (Raid0-5)



An array of inexpensive disks
(Can read 8 tracks at once)

But we have an increased reliability problem.
If any one disk fails, all 8 are effectively useless.





A **redundant** array of inexpensive disks.

RAID and Reliability

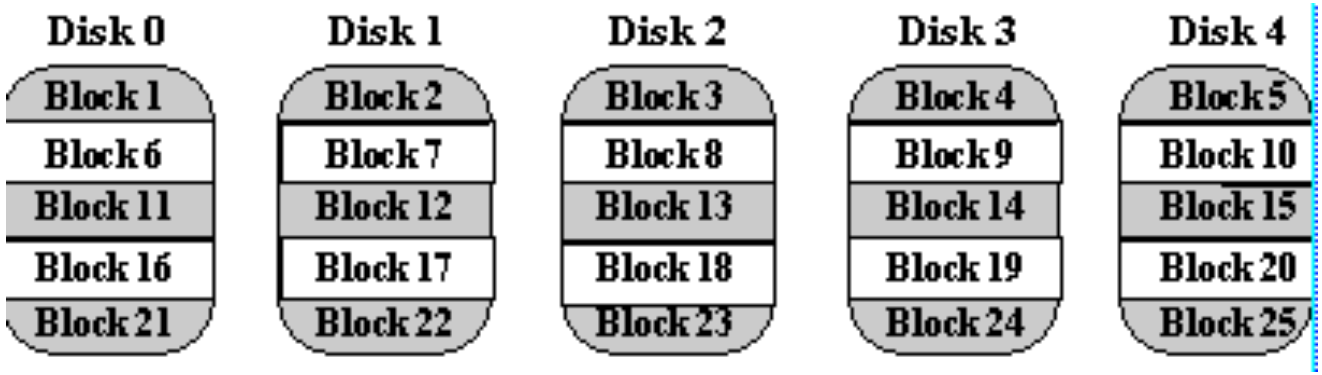
- So far, we assume that we have a single disk
- What if we have multiple disks?
  - The chance of a single-disk failure increases
- *RAID: redundant array of independent disks*
  - Standard way of organizing disks and classifying the reliability of multi-disk systems
  - General methods: data duplication, parity, and error-correcting codes (ECC)

Different RAID levels

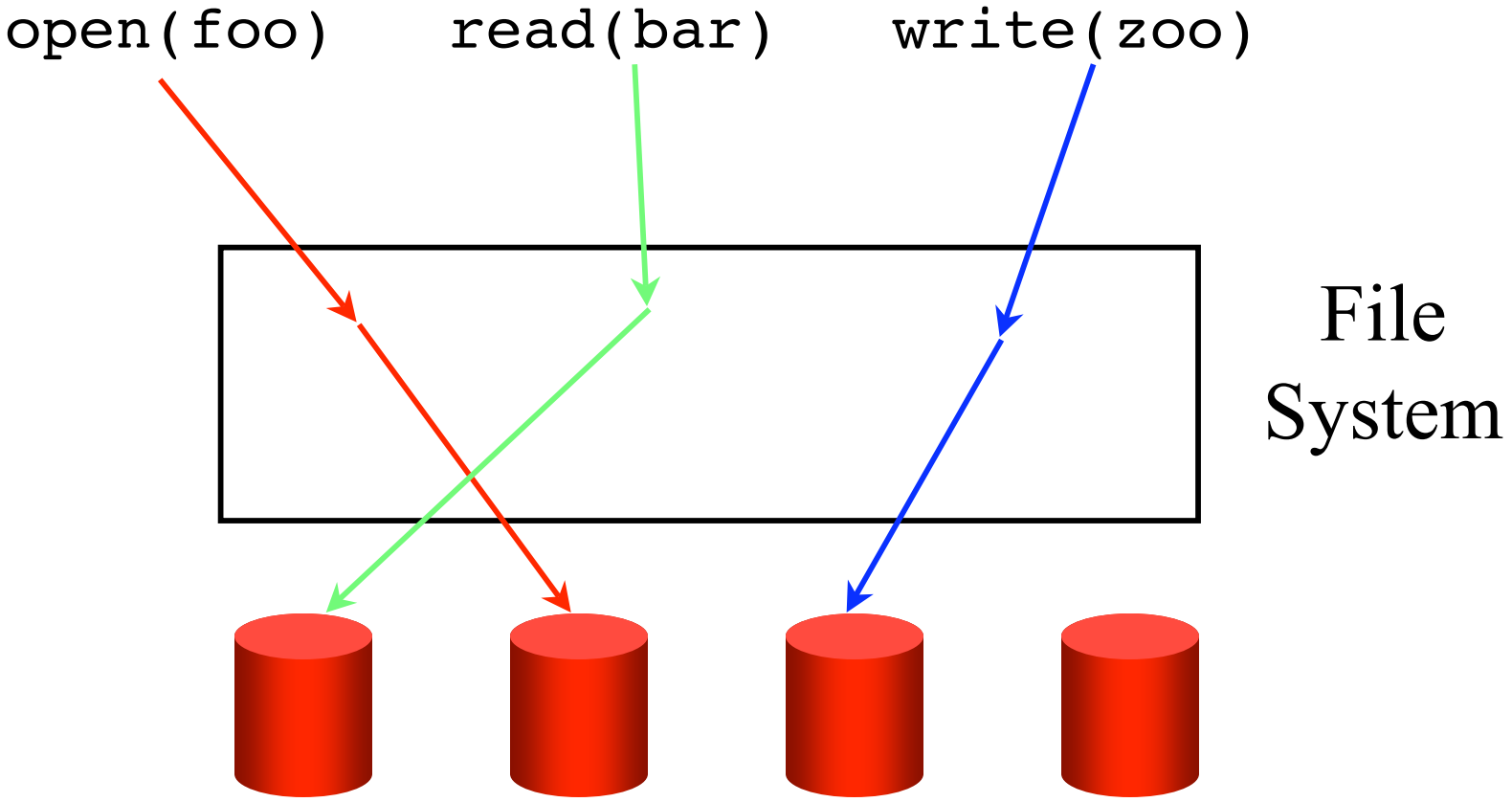
- Raid 0: Striping
  - reduce disk queue length, good for random access. no reliability
- Raid 1: Mirroring
  - write data to both disks, simple expensive
- Raid 2: ECC
  - stripe at bit level, multiple parity bits to determine failed bit. expensive (eg, 10 data disks require 4 ECC disks), read/write 1 bit, means read/write all in a parity stripe!
- Raid 3,4,5: PARITY only.
  - disks say they break. Only need to reconstruct.
  - Difference is where we place the parity bits

RAID 0

- No redundancy
- Failure causes data loss

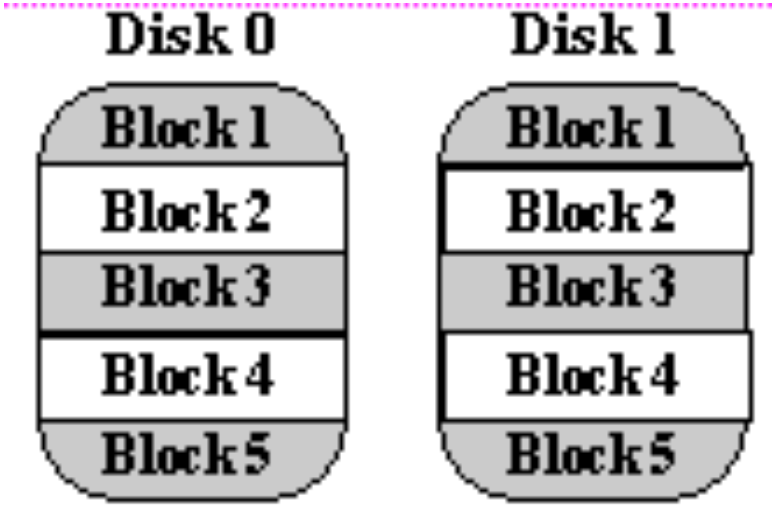


Non-Redundant Disk Array Diagram (RAID Level 0)



Mirrored Disks (RAID Level 1)

- Each disk has a second disk that mirrors its contents



Mirrored Disks (RAID Level 1)

- Writes go to both disks

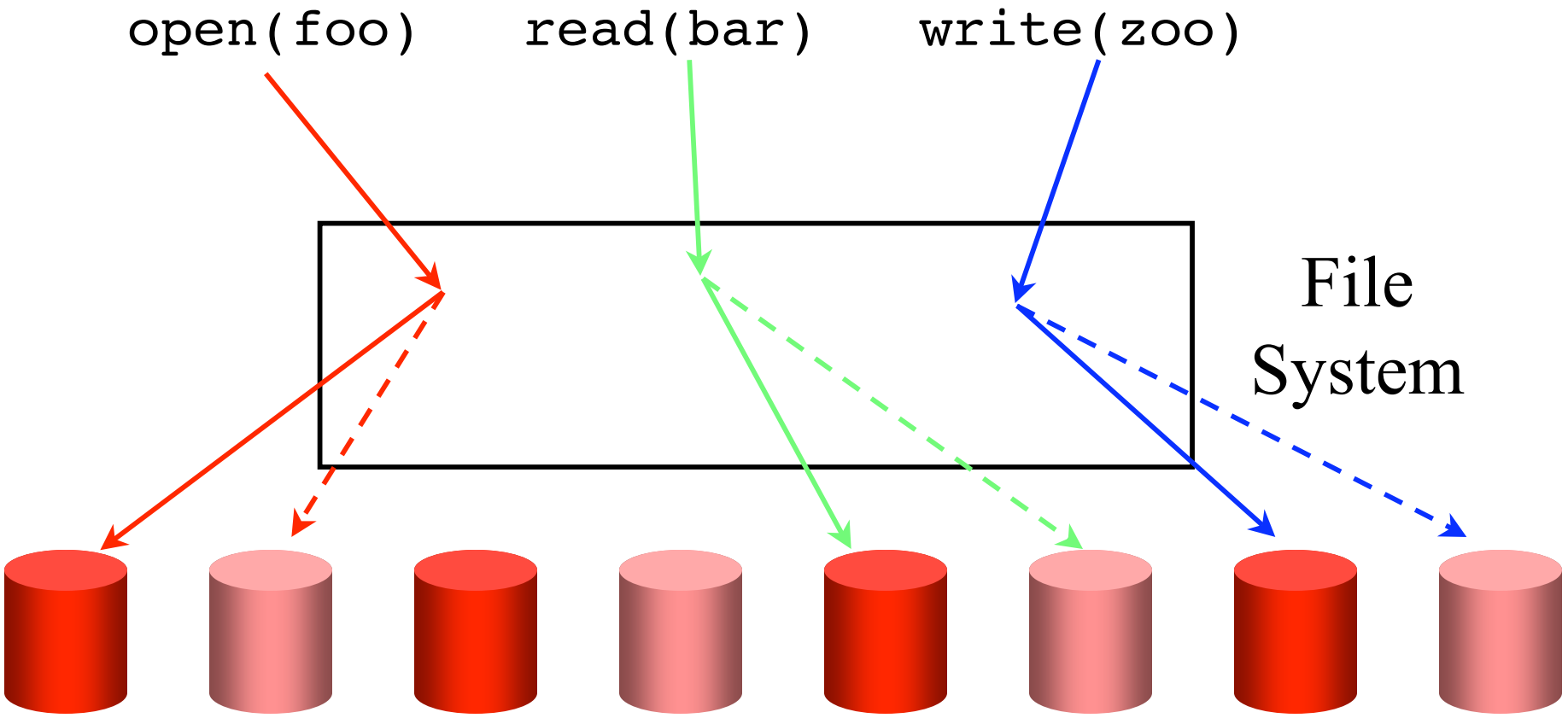
+ Reliability is doubled

+ Read access faster

- Write access slower

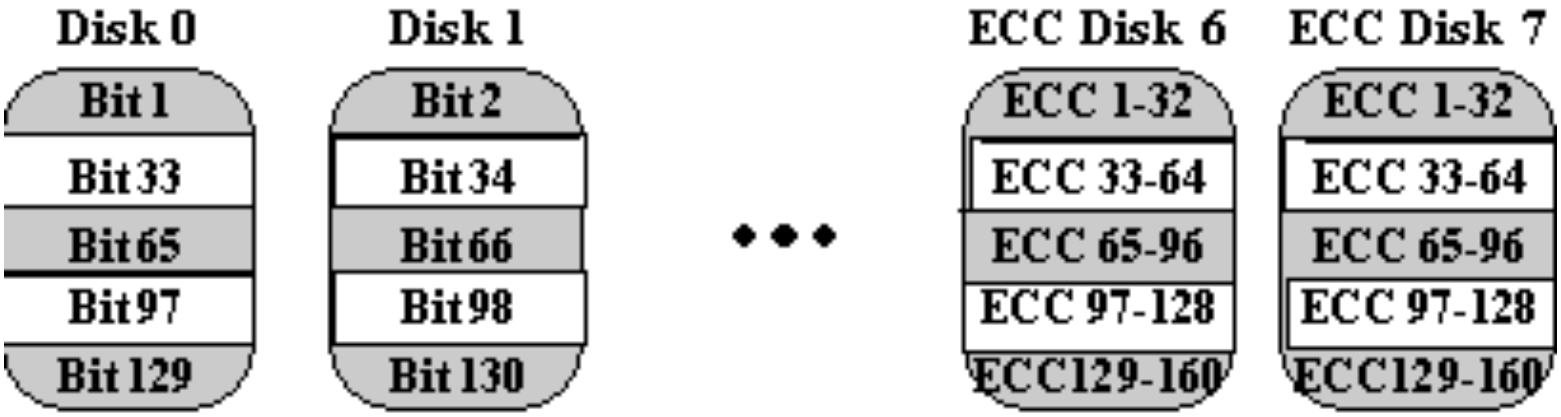
- Expensive and inefficient

Mirrored Disk Diagram (RAID Level 1)



Memory-Style ECC (RAID Level 2)

- Some disks in array are used to hold ECC
  - Using Hamming codes as the ECC
    - **detect & correct** one bit error in a 4 bits code word requires 3 redundant bits.



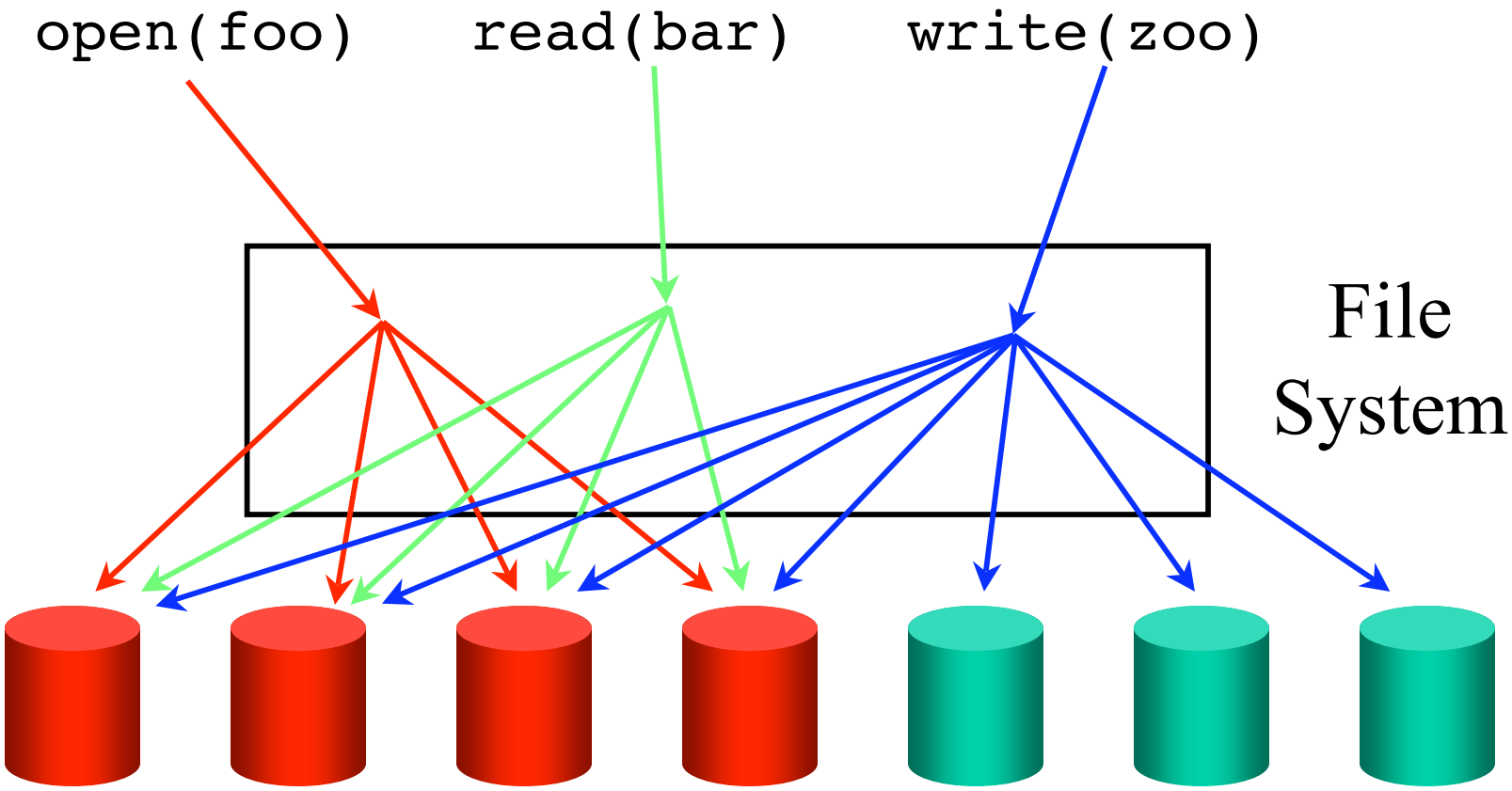
Memory-Style ECC (RAID Level 2)

+ More efficient than mirroring

+ Can correct, not just detect, errors

- Still fairly inefficient
  - e.g., 4 data disks require 3 ECC disks

Memory-Style ECC Diagram (RAID Level 2)

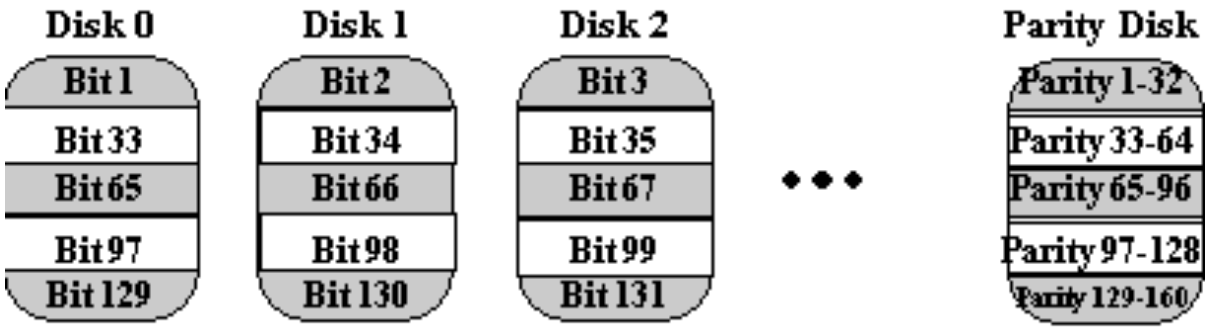


Bit-Interleaved Parity (RAID Level 3)

- One disk in the array stores parity for the other disks
  - Enough to correct the error when the disk **controller** tells which disk fails.

+ More efficient that Levels 1 and 2

- Parity disk doesn't add bandwidth

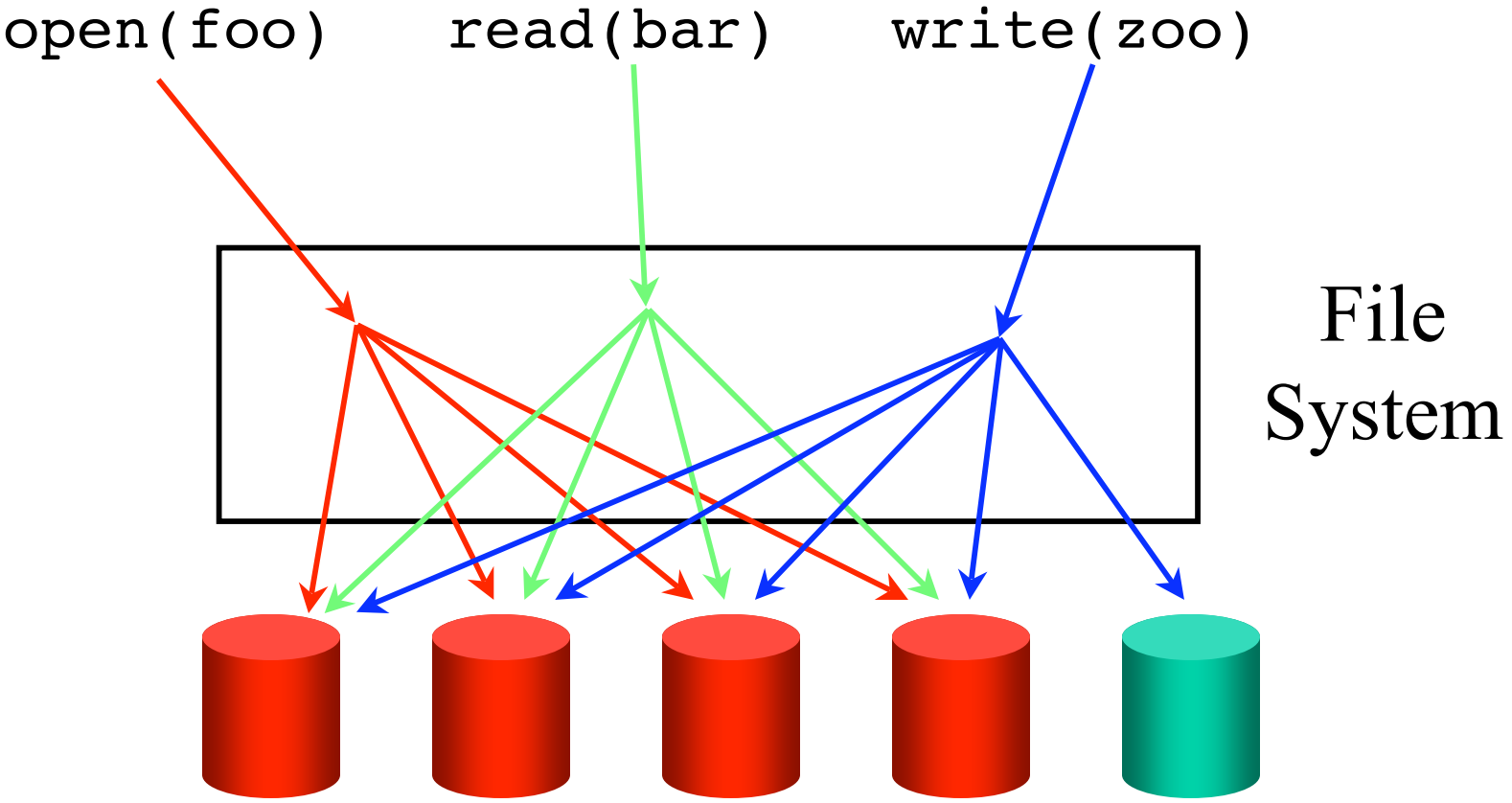


Parity Method

- Disk 1: 1001
- Disk 2: 0101
- Disk 3: 1000
- Parity: 0100 (even parity: the number of 1's is an even number)

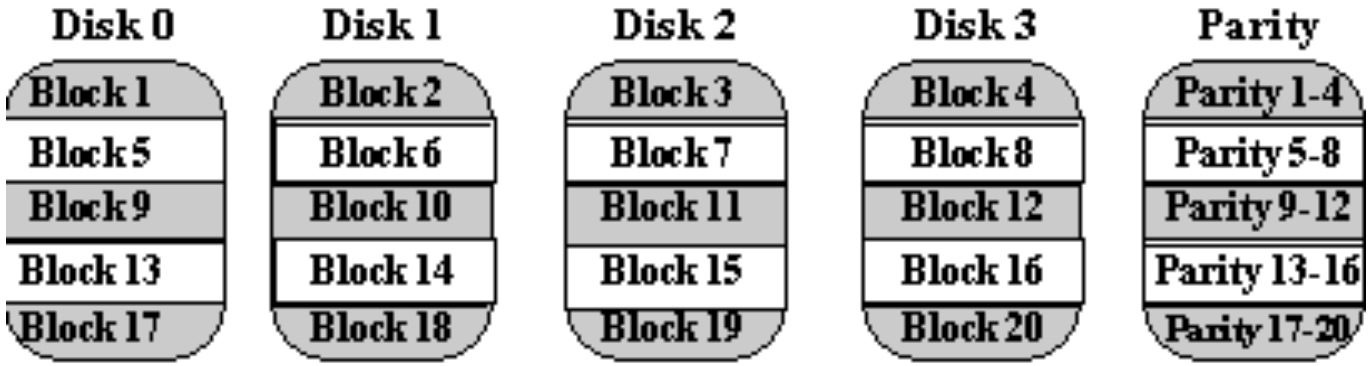
- How to recover disk 2?

Bit-Interleaved RAID Diagram (Level 3)



Block-Interleaved Parity (RAID Level 4)

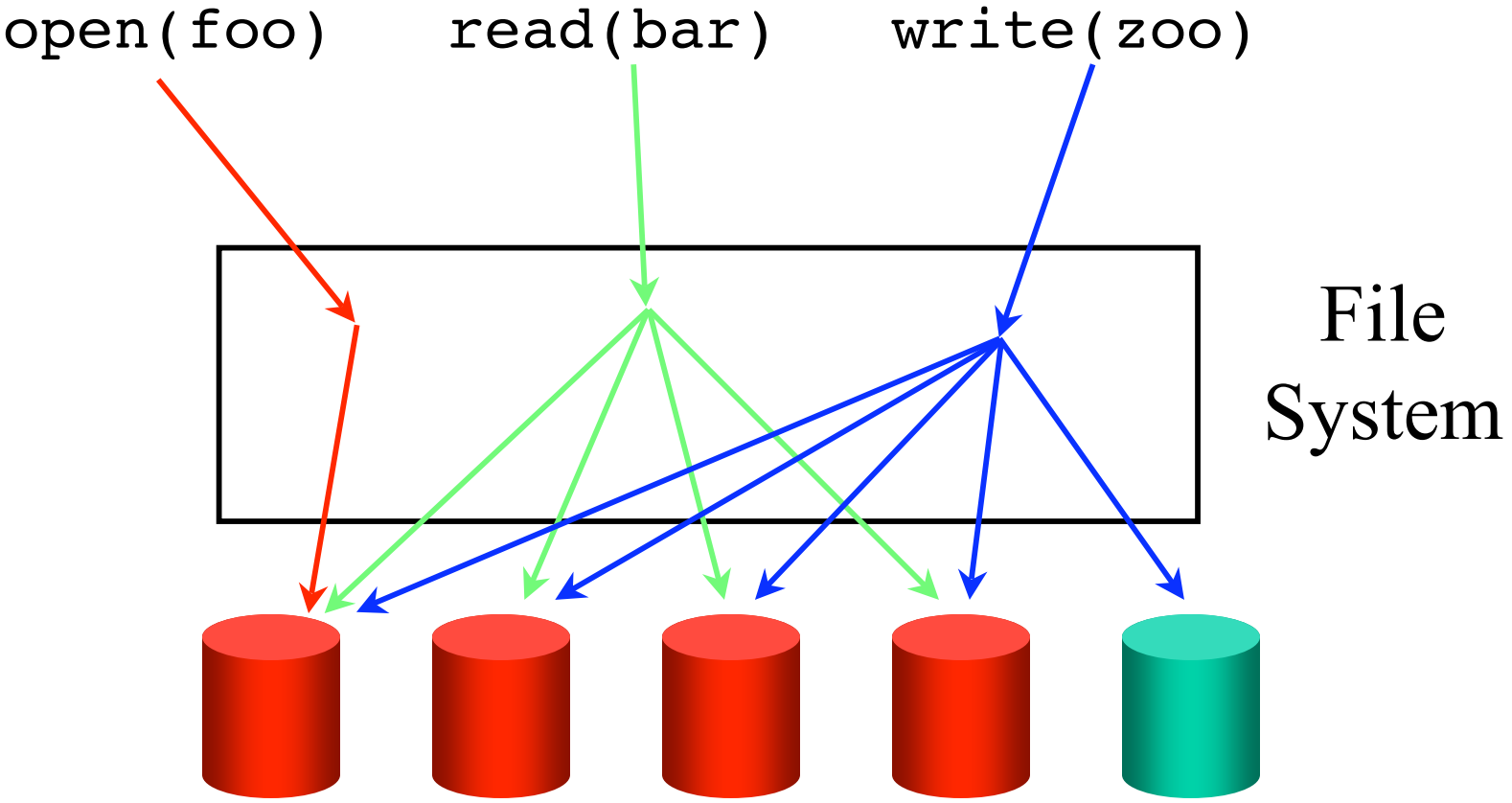
- Like bit-interleaved, but data is interleaved in blocks



Block-Interleaved Parity (RAID Level 4)

+ More efficient data access than level 3
  - esp for small reads
- Parity disk can be a bottleneck
  - Every write needs to write the parity disk.
- Small writes require 4 I/Os
  - Read the old block
  - Read the old parity
  - Write the new block
  - Write the new parity

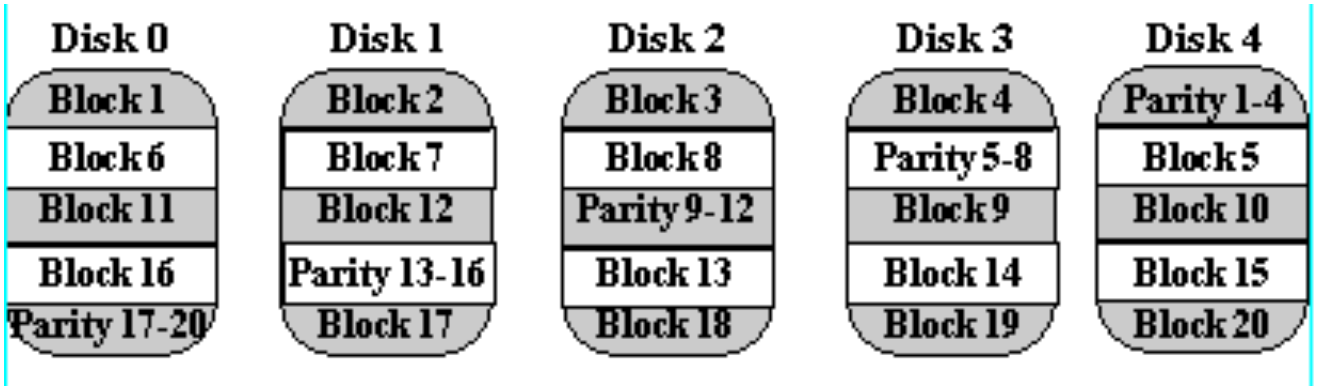
Block-Interleaved Parity Diagram (RAID Level 4)



Block-Interleaved Distributed-Parity (RAID Level 5)

- Sort of the most general level of RAID
- Spreads the parity out over all disks

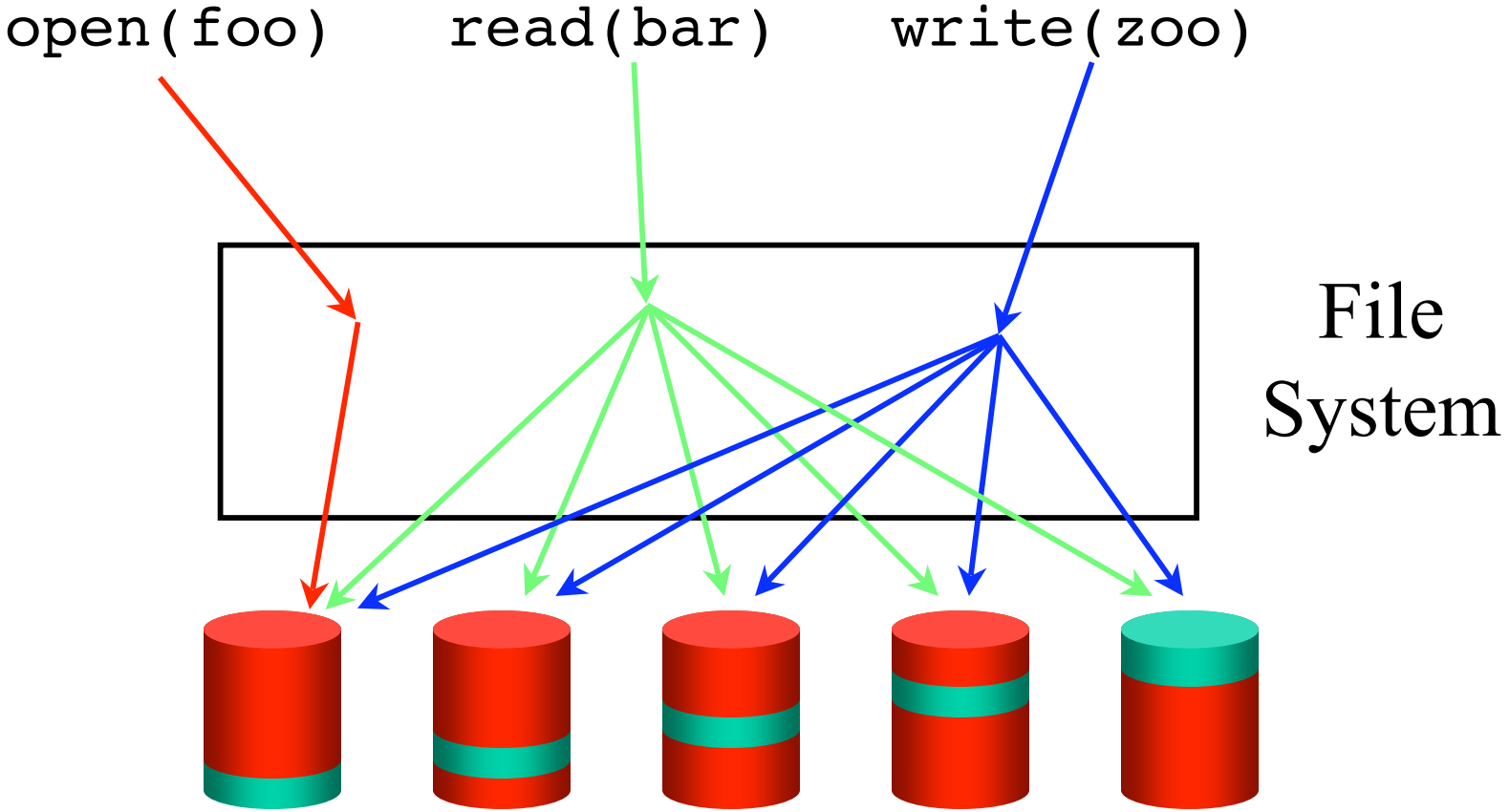
+ No parity disk bottleneck



Block-Interleaved Distributed-Parity (RAID Level 5)

+ All disks contribute read bandwidth
– Requires 4 I/Os for small writes
  – How to fix??

Block-Interleaved Distributed-Parity Diagram (RAID Level 5)



Summary

- Latency and bandwidth are a big problem in file systems
- Caching and read-ahead can help a tremendous amount
  - a little bit of mechanism can go a long way.
  - partly the reason why newer versions of an operating system can run so much faster.
  - partly also why your competitor's may be running so much slower.
- Many of the "interesting" issues in file systems are at the implementation level
  - within the operating system
- APIs are relatively similar, even across operating systems.