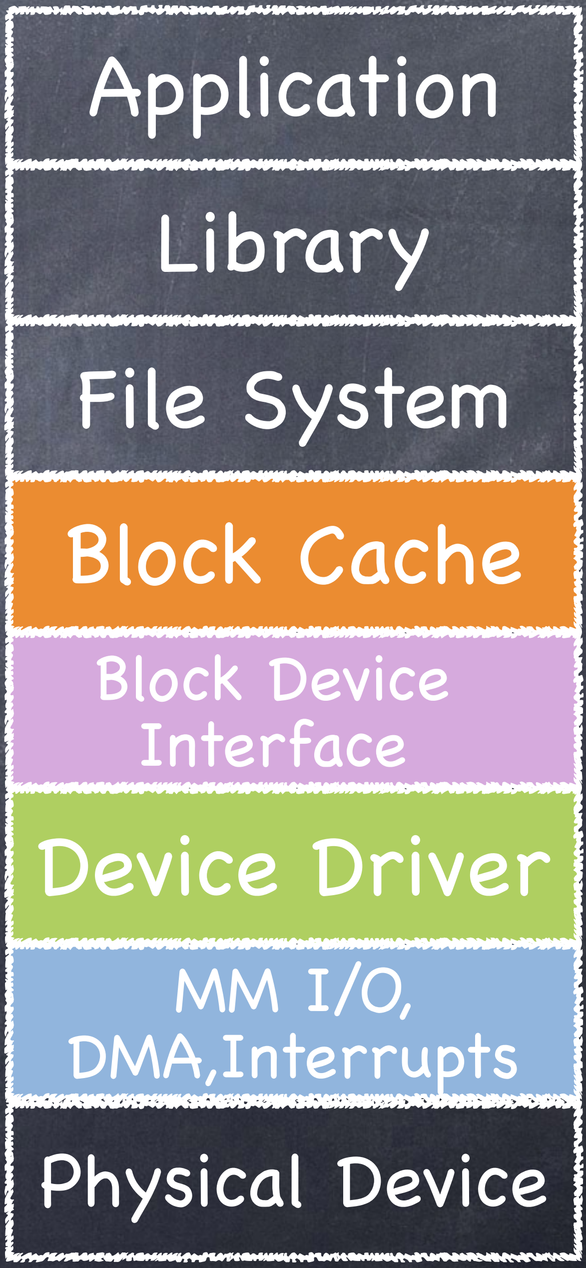
Storage stack:
Log-structured File System
RAID

Recall: The Storage Stack

- I/O systems are accessed through a series of layered abstractions
  - Caches blocks recently read from disk
  - Buffers recently written blocks
  - Single interface to many devices, allows data to be read/written in fixed sized blocks
  - Translates OS abstractions and hw specific details of I/O devices
  - Control registers, bulk data transfer, OS notifications



Recall: Storing Files

- Files can be allocated in different ways
  - Contiguous allocation
    - all bytes together, in order
  - Linked Structure
    - Each points to the next block
  - Indexed Structure
    - Index block, pointing to many other blocks

Recall: The Consistent Update Problem

- Several file systems operations update multiple data structures
  - Create new file
    - update inode bitmap and data bitmap
    - write new inode
    - add new file to directory file
- Would like to atomically move FS from one consistent state to another

Recall: Solution 1: File System Checker

- Ethos: If it happens, I'll do something about it
  - Let inconsistencies happen and fix them post facto
    - during reboot
- Classic example: fsck
  - Unix, 1986
- Fixing inconsistencies post facto can be VERY slow

Recall: Solution 2: Ordered Updates & Transactions

- Group together actions so that they are
  - Atomic: either all happen or none
  - Consistent: maintain invariants
  - Isolated: serializable (schedule in which transactions occur is equivalent to transactions executing sequentially)
  - Durable: once completed, effects are persistent
- Transaction can have two outcomes:
  - Commit: transaction becomes durable
  - Abort: transaction never happened
    - may require appropriate rollback

Questions?

Solution 3: Journaling (write ahead logging)

- Turns multiple disk updates into a single disk write
  - "write ahead" a short note to a "log", specifying changes about to be made to the FS data structures
  - if a crash occurs while updating FS data structures, consult log to determine what to do
    - no need to scan entire disk!

Data Jounaling: an example

- We start with

inode bitmap	data bitmap	i-nodes	data blocks
0 | 1 | 0 | 0 | 0 | 0	0 | 0 | 0 | 0 | 1 | 0	-- Iv1 -- -- -- --	-- -- -- -- D1 --

- We want to add a new block to the file
- Three easy steps
  - Write to the log 5 blocks: TxBegin | Iv2 | Bv2 | D2 | TxEnd (includes TxID and blocks' final addresses)
    - write each record to a block, so it is atomic
  - Write the blocks for Iv2, Bv2, D2 to the FS proper [a.k.a checkpoint]
  - Mark the transaction free in the journal
- What if we crash before the log is updated?
  - if no commit, nothing made it into FS - ignore changes!
- What if we crash after the log is updated?
  - replay changes in log back to disk!

Journaling and Write Order

- Issuing the 5 writes to the log TxBegin | Iv2 | B2 | D2 | TxEnd sequentially is slow
  - Issue at once, and transform in a single sequential write!?
- Problem: disk can schedule writes out of order
  - first write TxBegin, Iv2, B2, TxEnd
  - then write D2

Disk loses power →

- Log contains: TxBegin | Iv2 | B2 | ?? | TxEnd
  - syntactically, transaction log looks fine, even with nonsense in place of D2!
- TxEnd must block until prior blocks are on disk
  - Transaction committed when TxEnd on disk

Log Structured File Systems

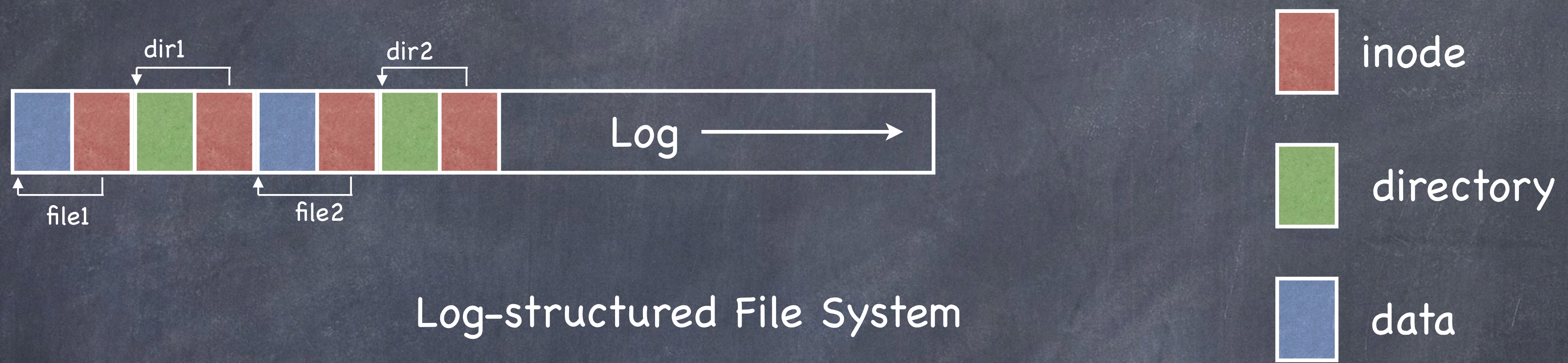
- Instead of adding a log to the existing FS disk layout, use all disk as a log
  - buffer all updates (including metadata!) into an in-memory data structure
  - Periodically, write to persistent storage in a long sequential transfer
- Never overwrite existing data
  - always write data to "next" free locations
  - Sequential writes: much improved throughput

Log Structured File Systems

- But how does it work?
  - suppose we want to add a new block to a 0-sized file
  - not enough to write to log just the data block...
  - ...we have to update the inode too!
- LFS places both data block and inode in-memory

- Leverages write buffering to write a chunk of updates all at once

LFS



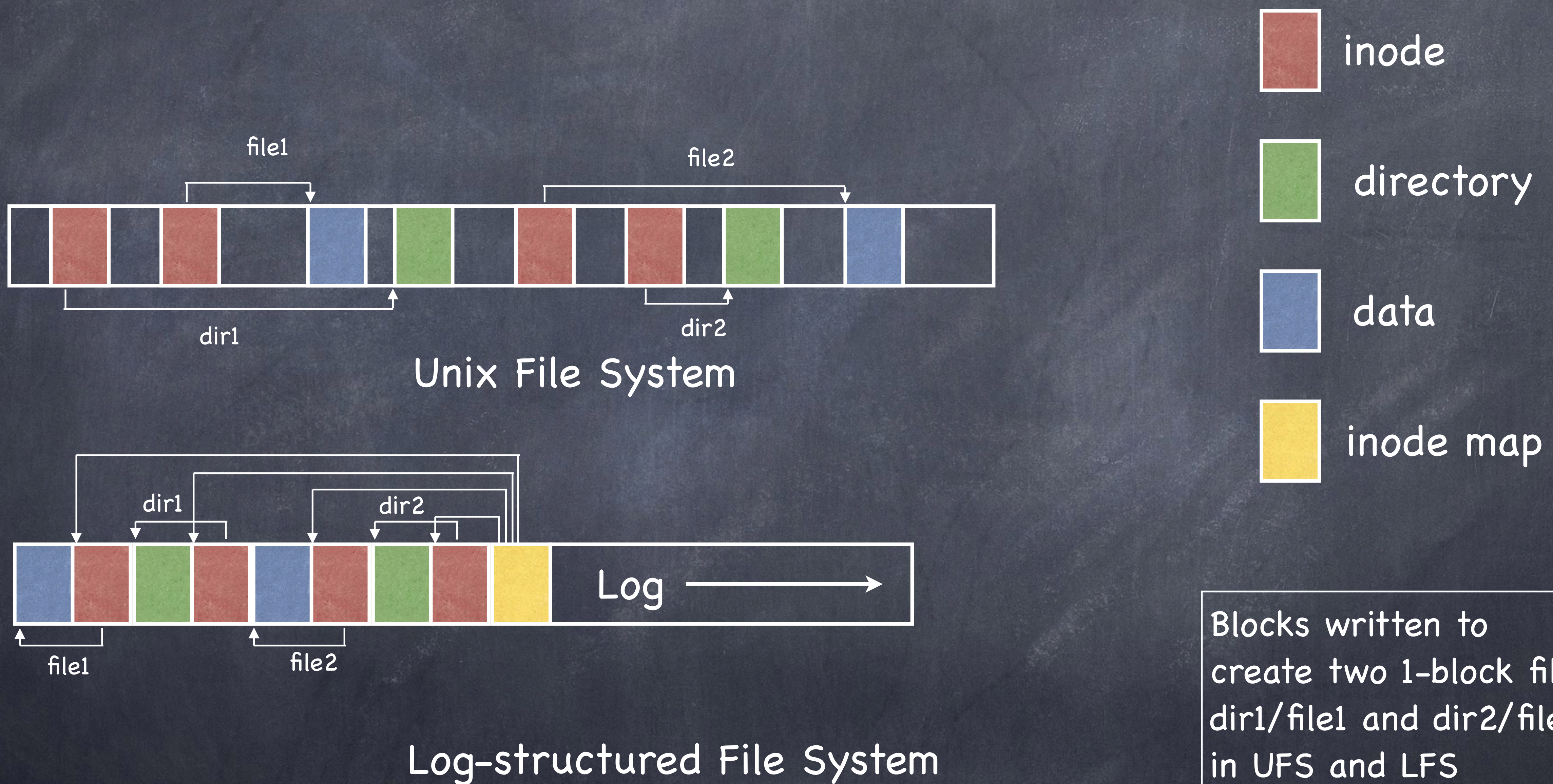
Log-structured File System

How do we quickly find inodes?

Finding inodes in LFS

- Inode map: a table indicating where each inode is on disk: Imap(i#) -> disk address of i#
- Keep it in memory, and periodically push it to disk
- In case of failures, reconstruct
  - e.g., by scanning data on disk

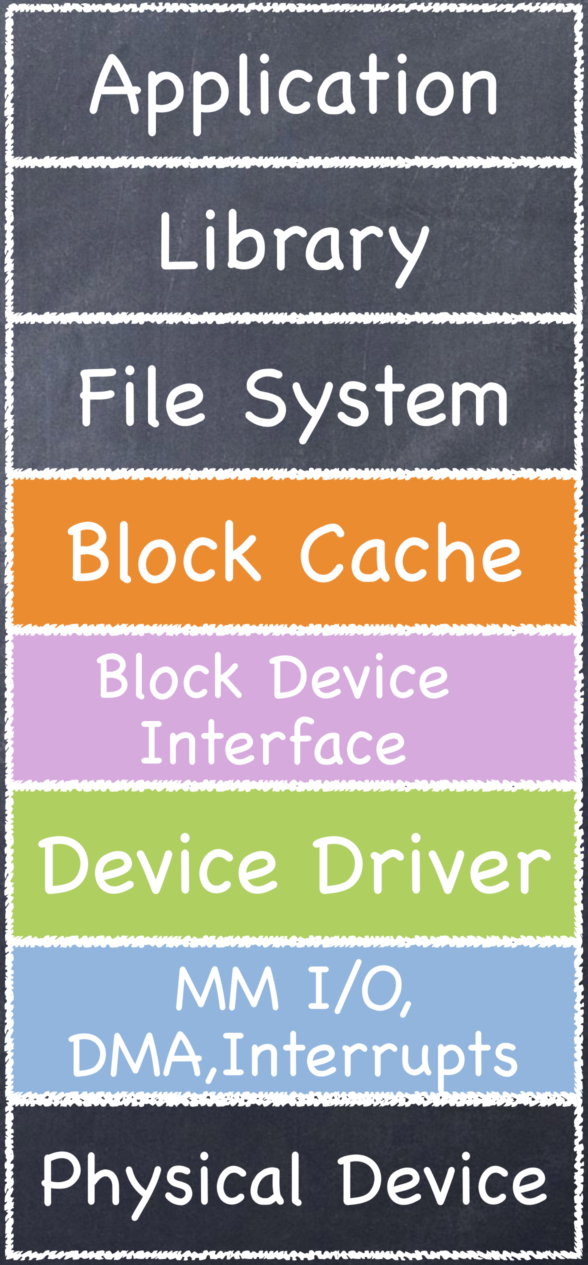
LFS vs UFS



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

Recall: The Storage Stack

- I/O systems are accessed through a series of layered abstractions
  - Caches blocks recently read from disk
  - Buffers recently written blocks
  - Single interface to many devices, allows data to be read/written in fixed sized blocks
  - Translates OS abstractions and hw specific details of I/O devices
  - Control registers, bulk data transfer, OS notifications



RAID

Redundant Array of Inexpensive* Disks

* In industry, “inexpensive” has been replaced by “independent” :-)

High-level idea

- Implement the abstraction of a faster, bigger and more reliable disk using a collection of slower, smaller, and more likely to fail disks
  - different configurations offer different tradeoffs
- Key feature: transparency
  - The Power of Abstraction™
  - to the OS looks like a single, large, highly performant and highly reliable single disk
    - a linear array of blocks
    - mapping needed to get to actual disk
    - cost: one logical I/O may translate into multiple physical I/Os
- In the box:
  - microcontroller, DRAM (to buffer blocks) [sometimes non-volatile memory, parity logic]

Failure Model

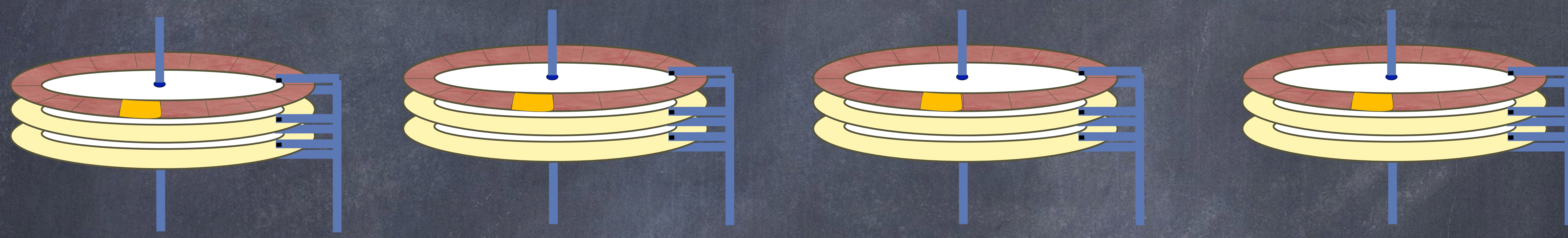
- RAID adopts the strong, somewhat unrealistic Fail-Stop failure model (electronic failure, wear out, head damage)
  - component works correctly until it crashes, permanently
    - Storage device is either working: all blocks can be read and written
    - or has failed: it is permanently lost
  - failure of the component is immediately detected
    - RAID controller can immediately observe a disk has failed and accesses return error codes
- In reality, storage devices can also suffer from isolated failures
  - Permanent: physical malfunction (wear out, scratches, contaminants)
  - Transient: data is corrupted, but new data can be successfully read from/ written

How to Evaluate a RAID

- Capacity
  - what fraction of the sum of the storage of its constituent disks does the RAID make available?
- Reliability
  - How many disk faults can a specific RAID configuration tolerate?
- Performance
  - Workload dependent

RAID-0: Striping

Spread blocks across disks using round robin



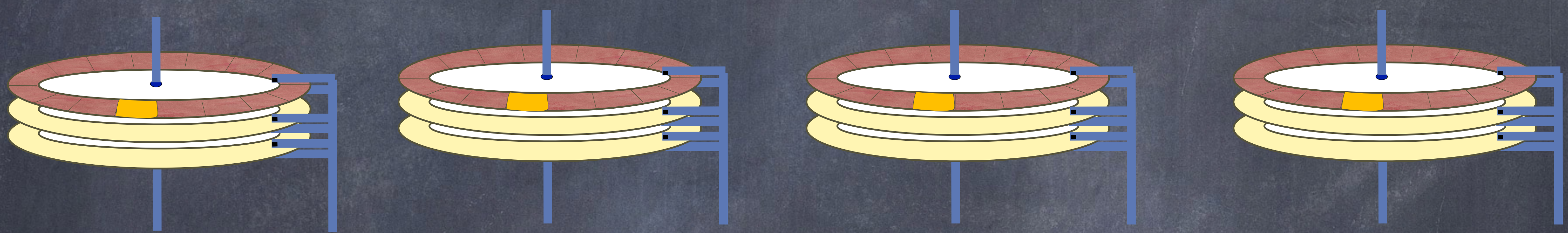
Stripe	0	1	2	3
Stripe	4	5	6	7
Stripe	8	9	10	11
Stripe	12	13	14	15

+ Excellent parallelism
  - can read/write from multiple disks

– Worst-case latency
  - wait for largest latency across all ops

RAID-0: Striping (Big Chunk Edition)

Spread blocks across disks using round robin



Stripe	0	2	4	6
	1	3	5	7
Stripe	8	10	12	14
	9	11	13	15

+ improve sequential throughput

– decrease parallelism

RAID-0: Evaluation

- Capacity
  - Excellent: N disks, each holding B blocks support the abstraction of a single disk with NxB blocks
- Reliability
  - Poor: Striping reduces reliability
    - Any disk failure causes data loss
- Performance
  - Workload dependent, of course
  - We'll consider two workloads
    - Sequential: single disk transfers S MB/s
    - Random: single disk transfer R MB/s
    - S >> R

RAID-0: Performance

- Single-block read/write throughput
  - about the same as accessing a single disk
- Latency
  - Read: T ms (latency of one I/O op to disk)
  - Write: T ms
- Steady-state read/write throughput
  - Sequential: N x S MB/s
  - Random: N x R MB/s

RAID-1: Mirroring

Each block is replicated twice

0	0	1	1
2	2	3	3
4	4	5	5
6	6	7	7

Read from any

Write to both

RAID-1: Evaluation

- Capacity
  - Poor: N disks of B blocks yield (N x B)/2 blocks
- Reliability
  - Good: Can tolerate the loss (not corruption!) of any one disk
- Performance
  - Fine for reads: can choose any disk
  - Poor for writes: every logical write requires writing to both disks
    - suffers worst-case delay of the two writes

RAID-1: Performance

- Steady-state throughput
  - Sequential Writes: N/2 x S MB/s
    - Each logical Write involves two physical Writes
  - Sequential Reads: N x S MB/s

0	0	1	1
2	2	3	3
4	4	5	5
6	6	7	7

Suppose we want to read

0, 1, 2, 3, 4, 5, 6, 7

RAID-1: Performance

- Steady-state throughput
  - Sequential Writes: N/2 x S MB/s
    - Each logical Write involves two physical Writes
  - Sequential Reads: N x S MB/s

0	0	1	1
2	2	3	3
4	4	5	5
6	6	7	7

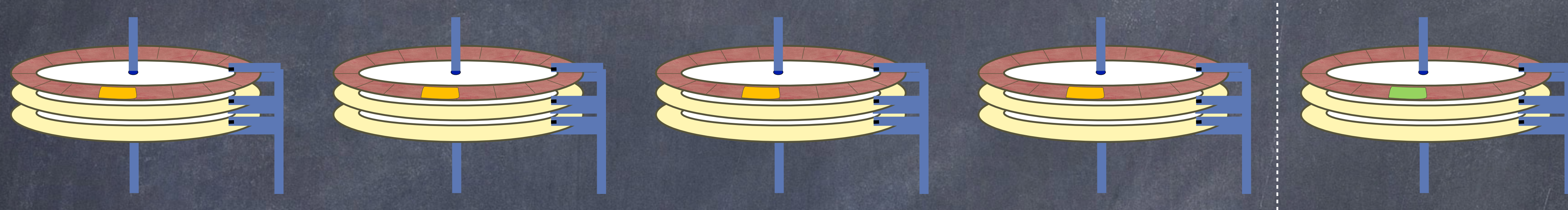
Suppose we want to read

0, 1, 2, 3, 4, 5, 6, 7

  - Random Writes: N/2 x R MB/s
    - Each logical Write involves two physical Writes
  - Random Reads: N x R MB/s
    - Reads can be distributed across all disks
- Latency for Reads and Writes: T ms

RAID-4: Block Striped, with Parity

Data disks | Parity disk



Stripe	0	1	2	3	P0
Stripe	4	5	6	7	P1
Stripe	8	9	10	11	P2
Stripe	12	13	14	15	P3

1	1	0
0	1	0
0	0	1

1	0	0
1	1	0
0	1	1

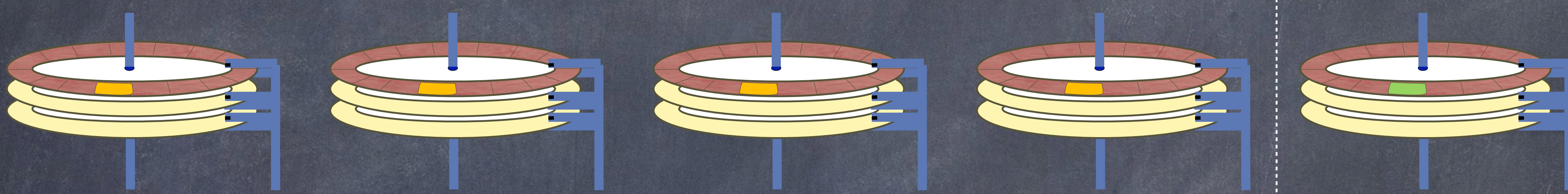
1	0	0
0	1	0
1	0	1

1	1	0
1	1	1
0	0	1

0	0	0
0	0	1
1	1	0

RAID-4: Block Striped, with Parity

Data disks | Parity disk



	Disk 0	Disk 1	Disk 2	Disk 3	Parity
Stripe	0	1	2	3	P0
Stripe	4	5	6	7	P1
Stripe	8	9	10	11	P2
Stripe	12	13	14	15	P3

1	1	0
0	1	0
0	0	1

1	0	0
1	1	0
0	1	1

1	0	0
0	1	0
1	0	1

1	1	0
1	1	1
0	0	1

0	0	0
0	0	1
1	1	0

Disk controller can identify faulty disk

- single parity disk can detect and correct errors

RAID-4: Evaluation

- Capacity
  - N disks of B blocks yield (N-1) x B blocks
- Reliability
  - Tolerates the failure of any one disk
- Performance
  - Fine for sequential read/write accesses and random reads
  - Random writes are a problem!

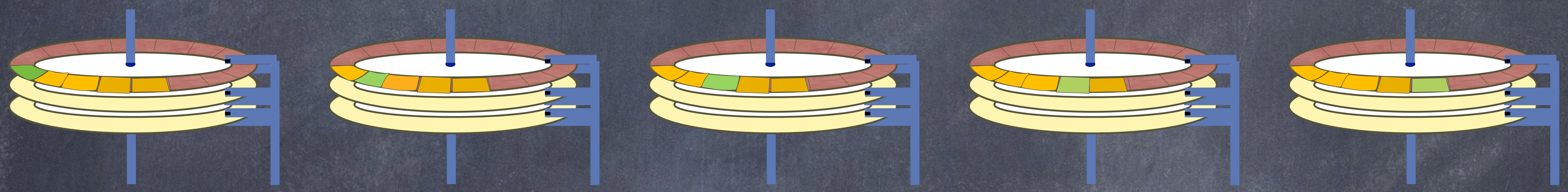
RAID-4: Performance

- Sequential Reads: (N-1) x S MB/s
- Sequential Writes: (N-1) x S MB/s
  - compute & write parity block once for the full stripe
- Random Read: (N-1) x R MB/s
- Random Writes: R/2 MB/s (N is gone! Yikes!)
  - need to read block from disk and parity block
  - Compute $P_{new} = (B_{old} \text{ XOR } B_{new}) \text{ XOR } P_{old}$
  - Write back B_{new} and P_{new}
  - Every write must go through parity disk, eliminating any chance of parallelism
  - Every logical I/O requires two physical I/Os at parity disk: can at most achieve 1/2 of its random transfer rate (i.e. R/2)

- Latency: Reads: T ms; Writes: 2T ms

RAID-5: Rotating Parity (avoids the bottleneck)

Parity and Data distributed across all disks



0	1	2	3	P0
5	6	7	P1	4
10	11	P2	8	9
15	P3	12	13	14
P4	16	17	18	19

RAID-5: Evaluation

- Capacity & Reliability
  - As in Raid-4
- Performance
  - Sequential read/write accesses as in RAID-4
    - (N-1) x S MB/s
  - Random Reads are slightly better
    - N x R MB/s (instead of (N-1) x R MB/s)
  - Random Writes much better than RAID-4: R/2 x N/2
    - as in RAID-4 writes involve two operations at every disk: each disk can achieve at most R/2
    - but, without a bottleneck parity disk, we can issue up to N/2 writes in parallel (each involving 2 disks)