File System Reliability (part 2)

Main Points

- Approaches to reliability
  - Careful sequencing of file system operations
  - Copy-on-write (WAFL, ZFS)
  - Journalling (NTFS, linux ext4)
  - Log structure (flash storage)
- Approaches to availability
  - RAID

Last Time: File System Reliability

- Transaction concept
  - Group of operations
  - Atomicity, durability, isolation, consistency
- Achieving atomicity and durability
  - Careful ordering of operations
  - Copy on write

Reliability Approach #1: Careful Ordering

- Sequence operations in a specific order
  - Careful design to allow sequence to be interrupted safely
- Post-crash recovery
  - Read data structures to see if there were any operations in progress
  - Clean up/finish as needed

- Approach taken in FAT, FFS (fsck), and many app-level recovery schemes (e.g., Word)

Reliability Approach #2: Copy on Write File Layout

- To update file system, write a new version of the file system containing the update
  - Never update in place
  - Reuse existing unchanged disk blocks
- Seems expensive! But
  - Updates can be batched
  - Almost all disk writes can occur in parallel
- Approach taken in network file server appliances (WAFL, ZFS)

Copy On Write

- Pros
  - Correct behavior regardless of failures
  - Fast recovery (root block array)
  - High throughput (best if updates are batched)
- Cons
  - Potential for high latency
  - Small changes require many writes
  - Garbage collection essential for performance

Logging File Systems

- Instead of modifying data structures on disk directly, write changes to a journal/log
  - Intention list: set of changes we intend to make
  - Log/Journal is **append-only**
- Once changes are on log, safe to apply changes to data structures on disk
  - Recovery can read log to see what changes were intended
- Once changes are copied, safe to remove log

Redo Logging

- Prepare
  - Write all changes (in transaction) to log
- Commit
  - Single disk write to make transaction durable
- Redo
  - Copy changes to disk
- Garbage collection
  - Reclaim space in log

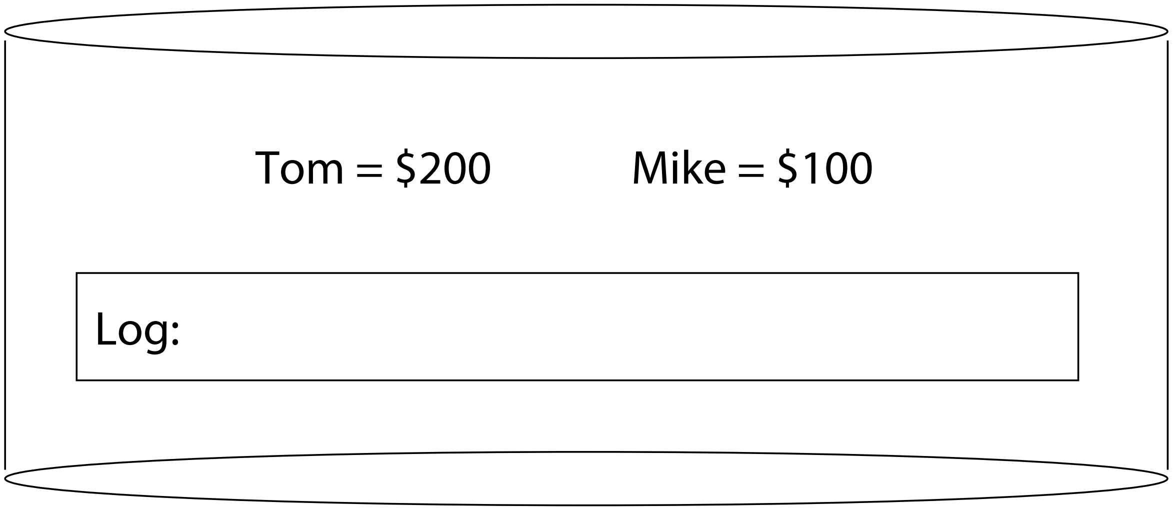
- Recovery
  - Read log
  - Redo any operations for committed transactions
  - Garbage collect log

Before Transaction Start

Cache

Tom = $200 Mike = $100

Nonvolatile Storage



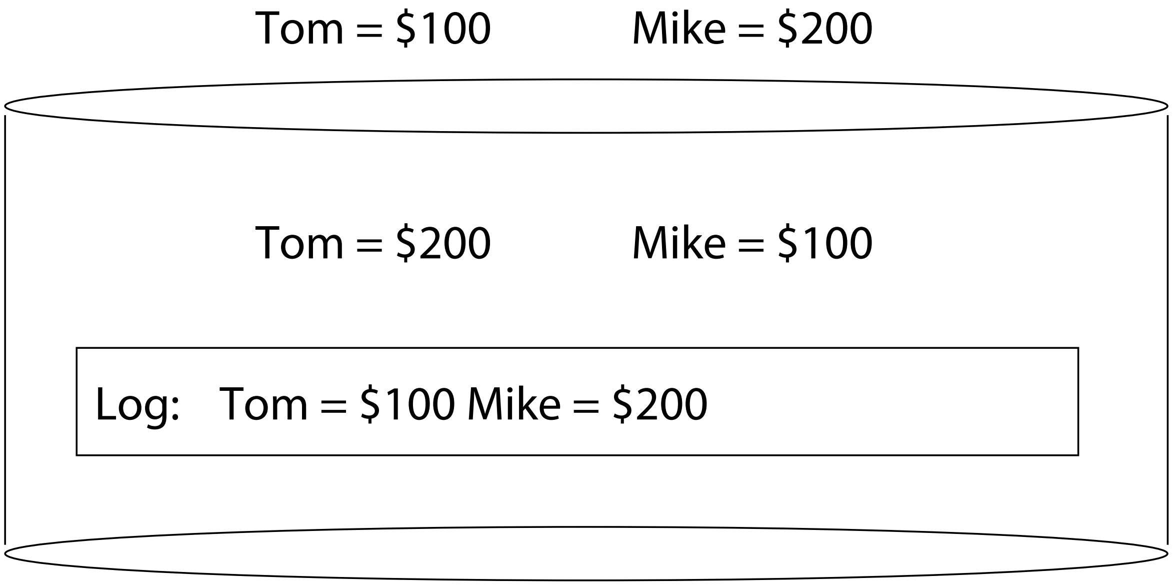
Tom = $200 Mike = $100

Log:

After Updates Are Logged

Cache

Nonvolatile Storage



Tom = $100 Mike = $200

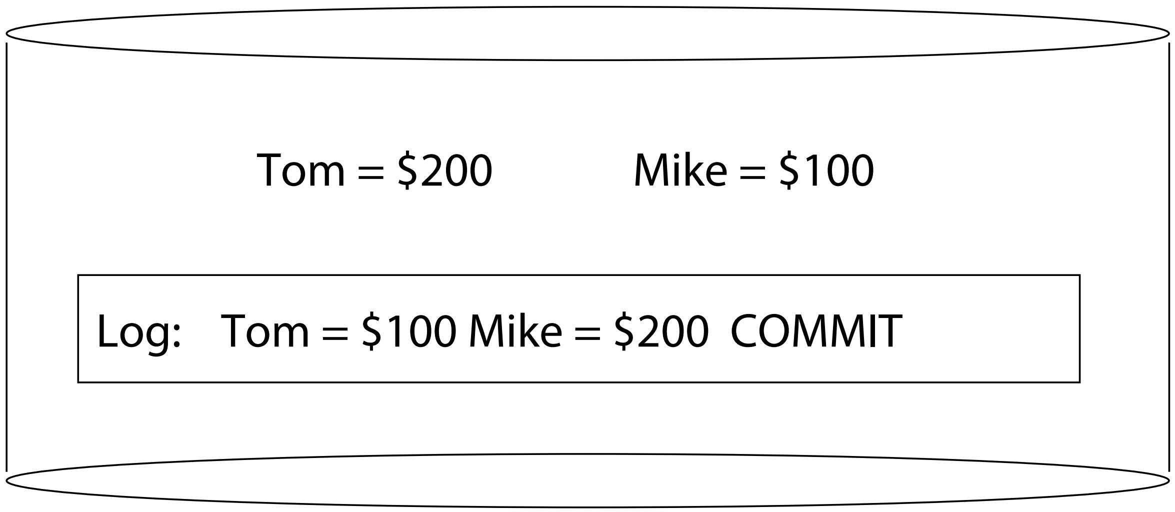
Tom = $200 Mike = $100

Log: Tom = $100 Mike = $200

After Commit Logged

Cache

Tom = $100 Mike = $200



Nonvolatile Storage

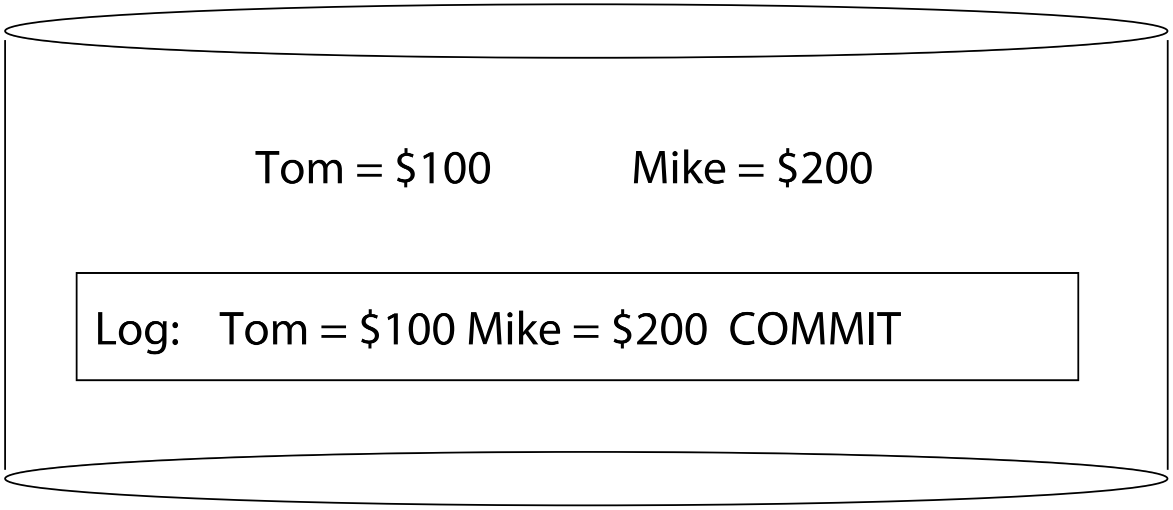
Tom = $200 Mike = $100

Log: Tom = $100 Mike = $200 COMMIT

After Copy Back

Cache

Tom = $100 Mike = $200



Nonvolatile Storage

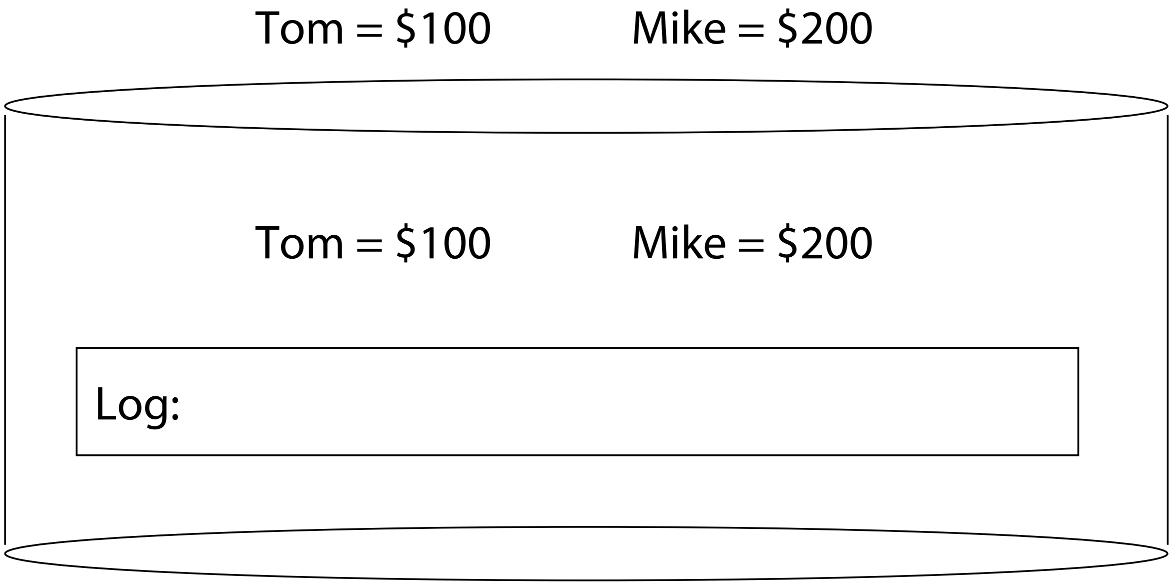
Tom = $100 Mike = $200

Log: Tom = $100 Mike = $200 COMMIT

After Garbage Collection

Cache

Nonvolatile Storage



Tom = $100 Mike = $200

Tom = $100 Mike = $200

Log:

Redo Logging

- Prepare
  - Write all changes (in transaction) to log
- Commit
  - Single disk write to make transaction durable
- Redo
  - Copy changes to disk
- Garbage collection
  - Reclaim space in log

- Recovery
  - Read log
  - Redo any operations for committed transactions
  - Garbage collect log

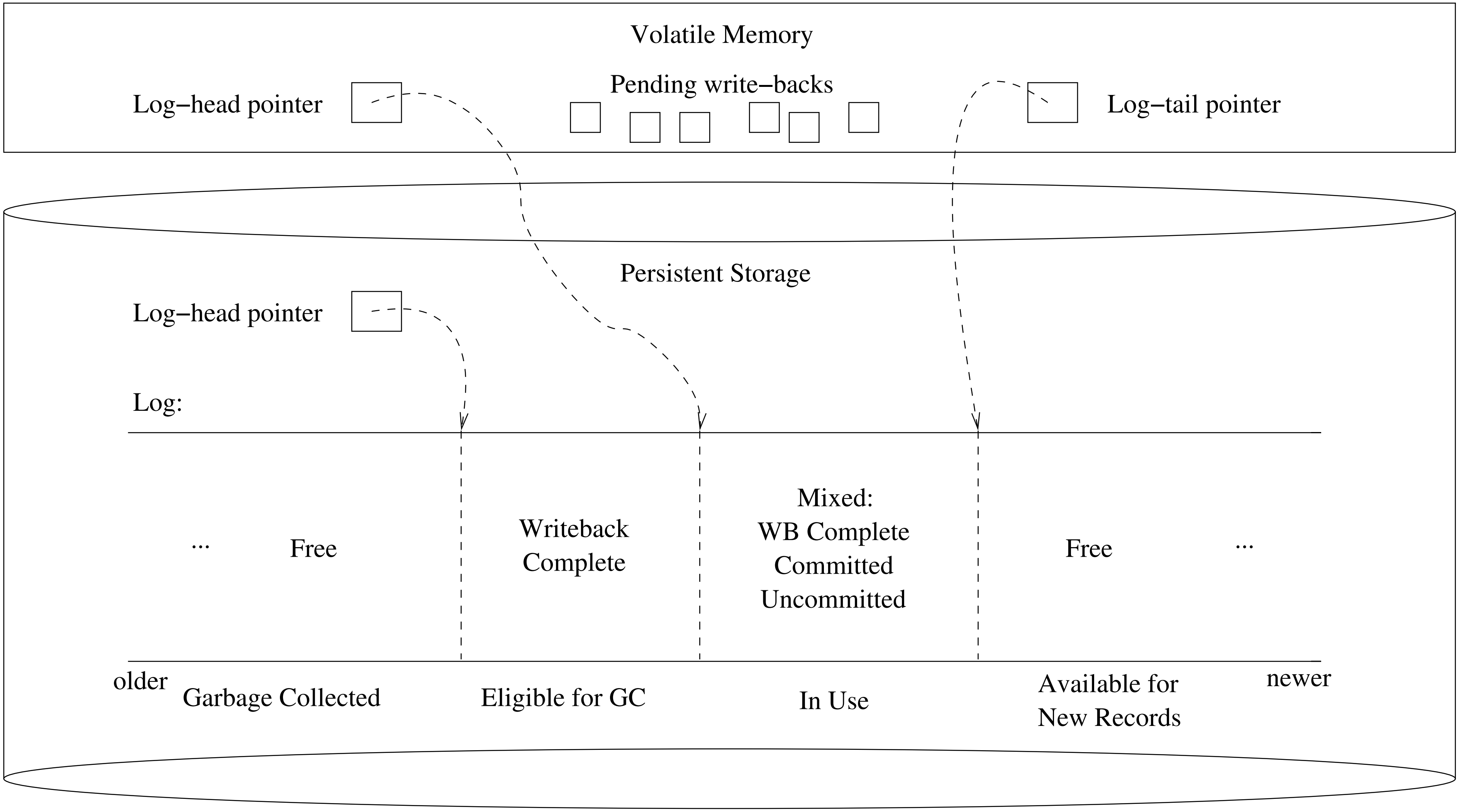
Questions

- What happens if machine crashes?
  - Before transaction start
  - After transaction start, before operations are logged
  - After operations are logged, before commit
  - After commit, before write back
  - After write back before garbage collection
- What happens if machine crashes during recovery?

Performance

- Log written sequentially
  - Often kept in flash storage
- Asynchronous write back
  - Any order as long as all changes are logged before commit, and all write backs occur after commit
- Can process multiple transactions
  - Transaction ID in each log entry
  - Transaction completed iff its commit record is in log

Redo Log Implementation



Transaction Isolation

Process A

move file from x to y

```
mv x/file y/
```

Process B

grep across x and y

```
grep x/* y/* > log
```

What if grep starts after changes are logged, but before commit?

Two Phase Locking

- Two phase locking: release locks only AFTER transaction commit
  - Prevents a process from seeing results of another transaction that might not commit

Transaction Isolation

Process A

Lock x, y

move file from x to y

```
mv x/file y/
```

Commit and release x,y

Process B

Lock x, y, log

grep across x and y

```
grep x/* y/* > log
```

Commit and release x, y, log

Grep occurs either before or after move

Serializability

- With two phase locking and redo logging, transactions appear to occur in **a** sequential order (serializability)
  - Either: grep then move or move then grep
- Other implementations can also provide serializability
  - Optimistic concurrency control: abort any transaction that would conflict with serializability

Caveat

- Most file systems implement a transactional model internally
  - Copy on write
  - Redo logging
- Most file systems provide a transactional model for individual system calls
  - File rename, move, …
- Most file systems do NOT provide a transactional model for user data
  - Historical artifact (imo)

Question

- Do we need the copy back?
  - What if update in place is very expensive?
  - Ex: flash storage, RAID

Log Structure

- Log is the data storage; no copy back
  - Storage split into contiguous fixed size segments
    - Flash: size of erasure block
    - Disk: efficient transfer size (e.g., 1MB)
  - Log new blocks into empty segment
    - Garbage collect dead blocks to create empty segments
  - Each segment contains extra level of indirection
    - Which blocks are stored in that segment
- Recovery
  - Find last successfully written segment

Reliability vs. Availability

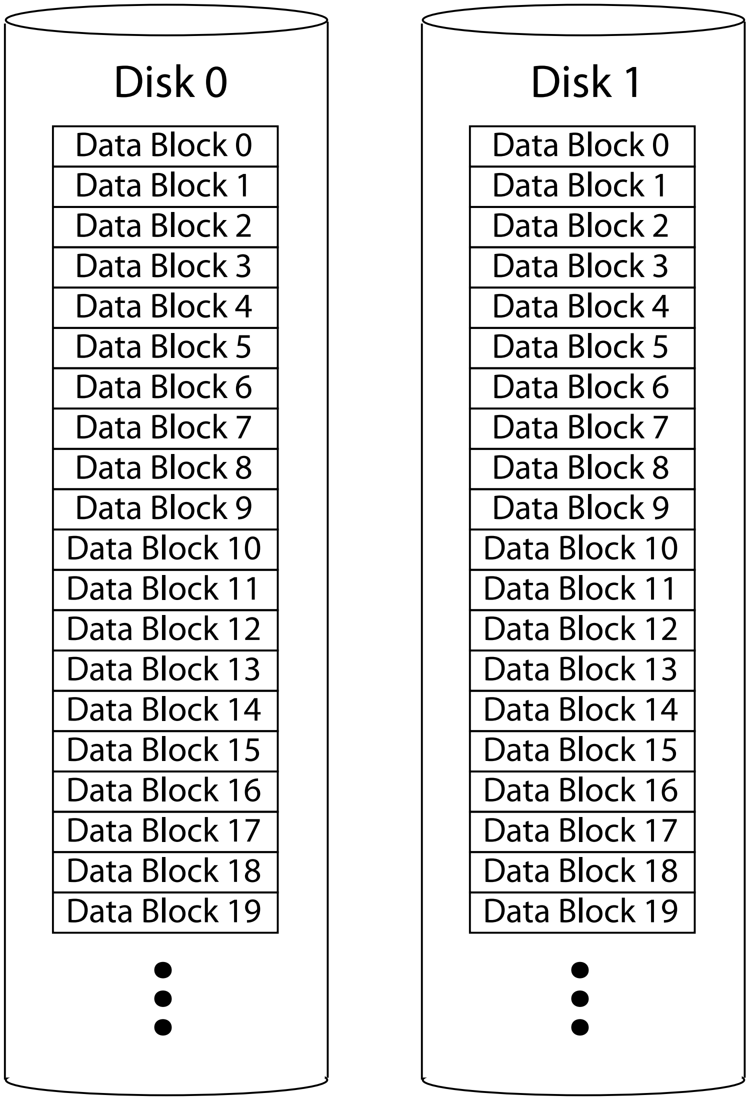
- Storage reliability: data fetched is what you stored
  - Transactions, redo logging, etc.
- Storage availability: data is there when you want it
  - What if there is a disk failure?
- What if you have more data than fits on a single disk?
  - If failures are independent and data is spread across k disks, data available ~ Prob(disk working)^k

RAID

- Replicate data for availability
  - RAID 0: no replication
  - RAID 1: mirror data across two or more disks
    - Google File System replicated all data on three disks, spread across multiple racks
  - RAID 5: split data across disks, with redundancy to recover from a single disk failure
  - RAID 6: RAID 5, with extra redundancy to recover from two disk failures

RAID 1: Mirroring

- Replicate writes to both disks
- Reads can go to either disk



Disk 0	Disk 1
Data Block 0	Data Block 0
Data Block 1	Data Block 1
Data Block 2	Data Block 2
Data Block 3	Data Block 3
Data Block 4	Data Block 4
Data Block 5	Data Block 5
Data Block 6	Data Block 6
Data Block 7	Data Block 7
Data Block 8	Data Block 8
Data Block 9	Data Block 9
Data Block 10	Data Block 10
Data Block 11	Data Block 11
Data Block 12	Data Block 12
Data Block 13	Data Block 13
Data Block 14	Data Block 14
Data Block 15	Data Block 15
Data Block 16	Data Block 16
Data Block 17	Data Block 17
Data Block 18	Data Block 18
Data Block 19	Data Block 19
⋮	⋮

Parity

- Parity block:
  - Block1 xor block2 xor block3 …

  100011

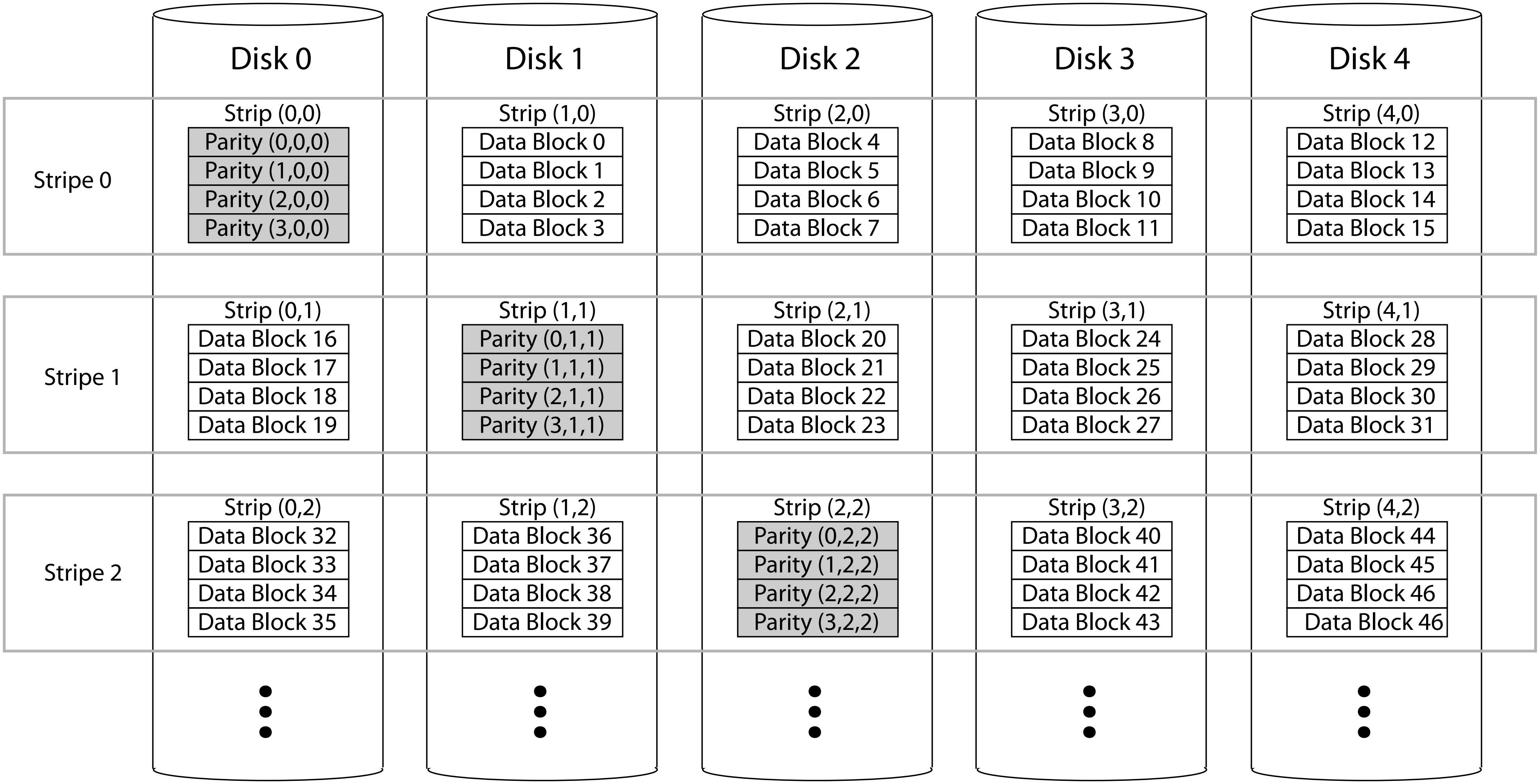
  011011

  110001

  ----------

  101001

RAID 5



	Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
Stripe 0	Strip (0,0) Parity (0,0,0) Parity (1,0,0) Parity (2,0,0) Parity (3,0,0)	Strip (1,0) Data Block 0 Data Block 1 Data Block 2 Data Block 3	Strip (2,0) Data Block 4 Data Block 5 Data Block 6 Data Block 7	Strip (3,0) Data Block 8 Data Block 9 Data Block 10 Data Block 11	Strip (4,0) Data Block 12 Data Block 13 Data Block 14 Data Block 15
Stripe 1	Strip (0,1) Data Block 16 Data Block 17 Data Block 18 Data Block 19	Strip (1,1) Parity (0,1,1) Parity (1,1,1) Parity (2,1,1) Parity (3,1,1)	Strip (2,1) Data Block 20 Data Block 21 Data Block 22 Data Block 23	Strip (3,1) Data Block 24 Data Block 25 Data Block 26 Data Block 27	Strip (4,1) Data Block 28 Data Block 29 Data Block 30 Data Block 31
Stripe 2	Strip (0,2) Data Block 32 Data Block 33 Data Block 34 Data Block 35	Strip (1,2) Data Block 36 Data Block 37 Data Block 38 Data Block 39	Strip (2,2) Parity (0,2,2) Parity (1,2,2) Parity (2,2,2) Parity (3,2,2)	Strip (3,2) Data Block 40 Data Block 41 Data Block 42 Data Block 43	Strip (4,2) Data Block 44 Data Block 45 Data Block 46 Data Block 46
	⋮	⋮	⋮	⋮	⋮

RAID Update

- Mirroring
  - Write every mirror
- RAID-5: one block
  - Read old data block
  - Read old parity block
  - Write new data block
  - Write new parity block
    - Old data xor old parity xor new data
- RAID-5: entire stripe
  - Write data blocks and parity

Non-Recoverable Read Errors

- Disk devices can lose data
  - One sector per 10^15 bits read
  - Causes:
    - Physical wear
    - Repeated writes to nearby tracks
- What impact does this have on RAID recovery?

Read Errors and RAID recovery

- Example
  - 10 1TB disks
  - 1 fails
  - Read remaining disks to reconstruct missing data
- Probability of recovery =

  (1 – 10^15)^(9 disks * 8 bits * 10^12 bytes/disk)

  = 93%
- Solutions:
  - RAID-6 (more redundancy)
  - Scrubbing – read disk sectors in background to find latent errors