File System Reliability

Main Points

- Problem posed by machine/disk failures
- Transaction concept
- Reliability
  - Careful sequencing of file system operations
  - Copy-on-write (WAFL, ZFS)
  - Journalling (NTFS, linux ext4)
  - Log structure (flash storage)
- Availability
  - RAID, Reed Solomon

File System Reliability

- What can happen if disk loses power or machine software crashes?
  - Some operations in progress may complete
  - Some operations in progress may be lost
  - Overwrite of a block may only partially complete
- File system wants durability (as a minimum!)
  - Data previously stored can be retrieved (maybe after some recovery step), regardless of failure

Storage Reliability Problem

- Single logical file operation can involve updates to multiple physical disk blocks
  - inode, indirect block, data block, bitmap, …
  - With remapping, single update to physical disk block can require multiple (even lower level) updates
- At a physical level, operations complete one at a time
  - Want concurrent operations for performance
- How do we guarantee consistency regardless of when crash occurs?

Transaction Concept

- Transaction is a group of operations
  - Atomic: operations appear to happen as a group, or not at all (at logical level)
    - At physical level, only single disk/flash write is atomic
    - To empty disk/flash block with consistency check
  - Durable: operations that complete stay completed
    - Future failures do not corrupt previously stored data
  - Isolation: other transactions do not see results of earlier transactions until they are committed
  - Consistency: sequential memory model

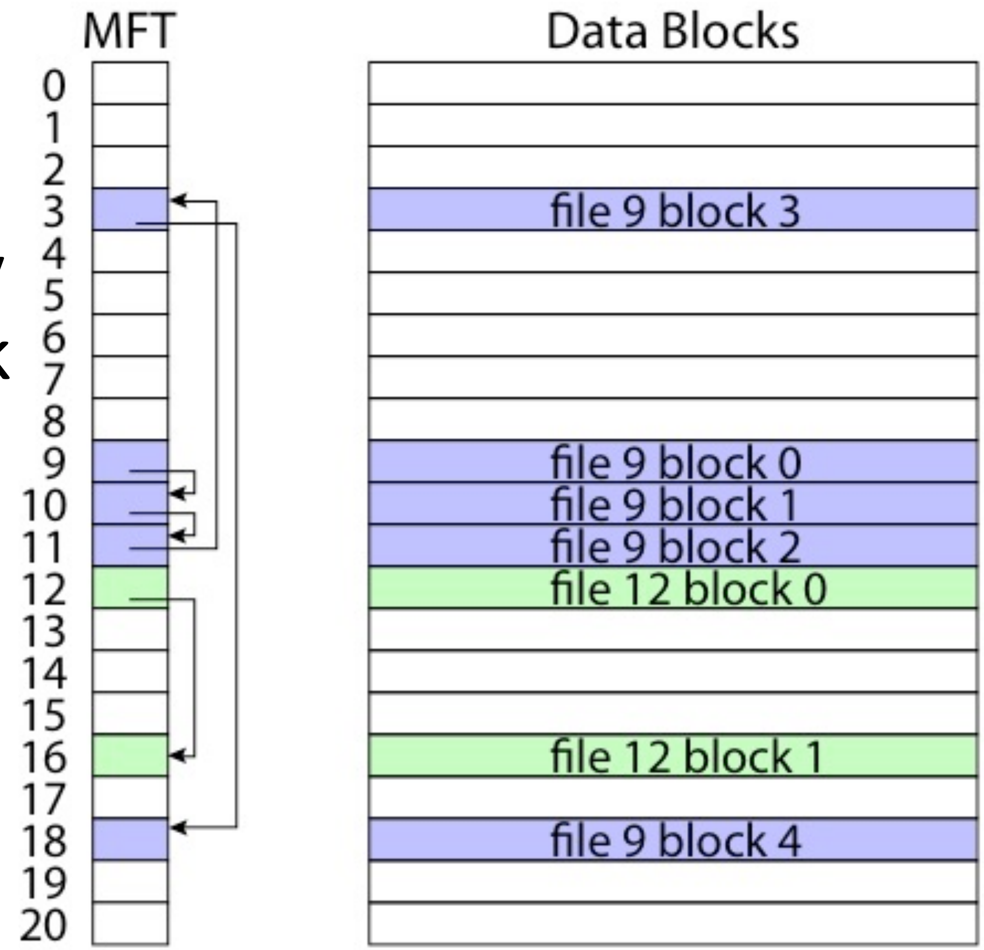
Reliability Approach #1: Careful Ordering

- 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)

FAT: Append Data to File

- Allocate data block
- Write data
- Write new MFT entry to point to data block
- Update file tail to point to new MFT entry
- Update access time at head of file



FAT: Append Data to File

Normal operation:

- Allocate data block
- Write data
- Write new MFT entry to point to data block
- Update file tail to point to new MFT entry
- Update access time at head of file

Recovery:

- Scan MFT
- If entry is unlinked, mark as unused
- If access time is incorrect, update

FAT: Create New File

Normal operation:

- Allocate data block
- Write MFT entry to point to data block
- Update directory with file name -> file number
  - What if directory spans multiple disk blocks?
- Update modify time for directory

Recovery:

- Scan MFT
- If any unlinked files (not in any directory), delete
- Scan directories for missing update times

FFS: Create a File

Normal operation:

- Allocate data block
- Write data block
- Allocate inode
- Write inode block
- Update bitmap of free blocks
- Update directory with file name -> file number
- Update modify time for directory

Recovery:

- Scan inode table
- If any unlinked files (not in any directory), delete
- Compare free block bitmap against inode trees
- Scan directories for missing update/access times

Time proportional to size of disk

FFS: Move a File

Normal operation:

- Remove filename from old directory
- Add filename to new directory

Recovery:

- Scan all directories to determine set of live files
- Consider files with valid inodes and not in any directory
  - New file being created?
  - File move?
  - File deletion?

FFS: Move and Grep

Process A

move file from x to y

mv x/file y/

Process B

grep across x and y

grep x/* y/*

Will grep always see contents of file?

Application Save File

Normal operation:

- Write name of each open file to app folder
- Write changes to backup file
- Rename backup file to be file (atomic operation provided by file system)
- Delete list in app folder on clean shutdown

Recovery:

- On startup, see if any files were left open
- If so, look for backup file
- If so, ask user to compare versions

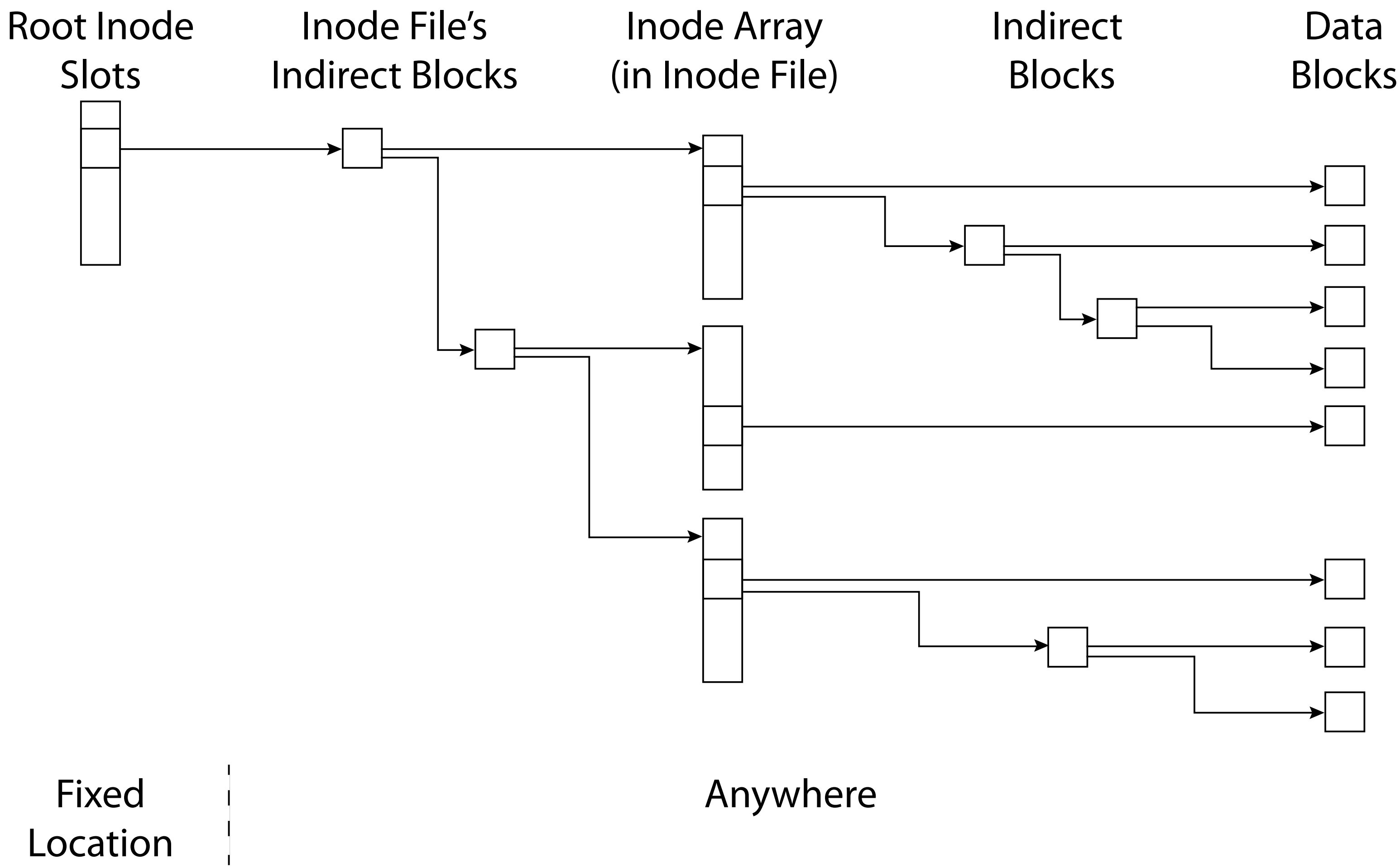
Careful Ordering

- Pros
  - Works with minimal support in the disk drive
  - Works for most multi-step operations
- Cons
  - Can require time-consuming recovery after a failure
  - Difficult to reduce every operation to a safely interruptible sequence of writes
  - Difficult to achieve consistency when multiple operations occur concurrently
  - Need to control when writes are flushed to storage

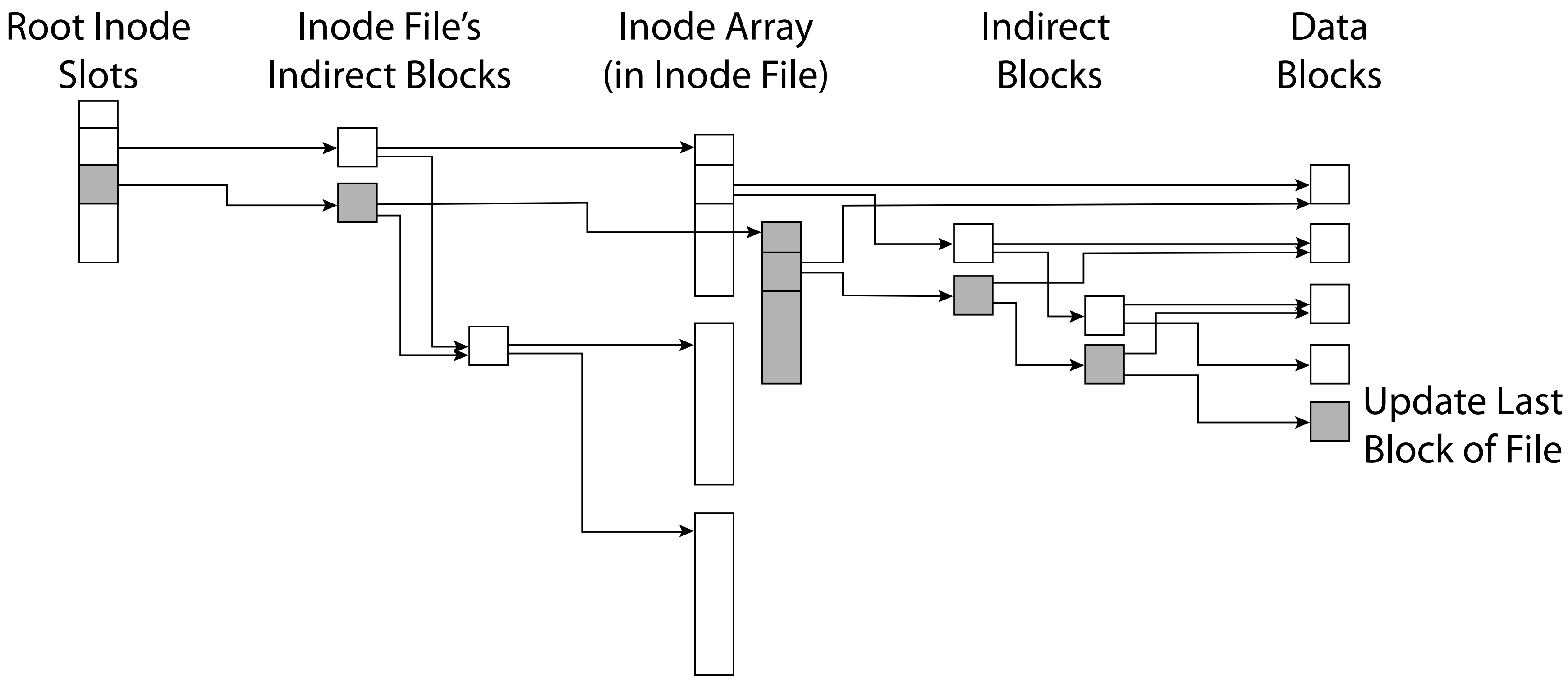
Reliability Approach #2: Copy on Write/Write Anywhere

- 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)

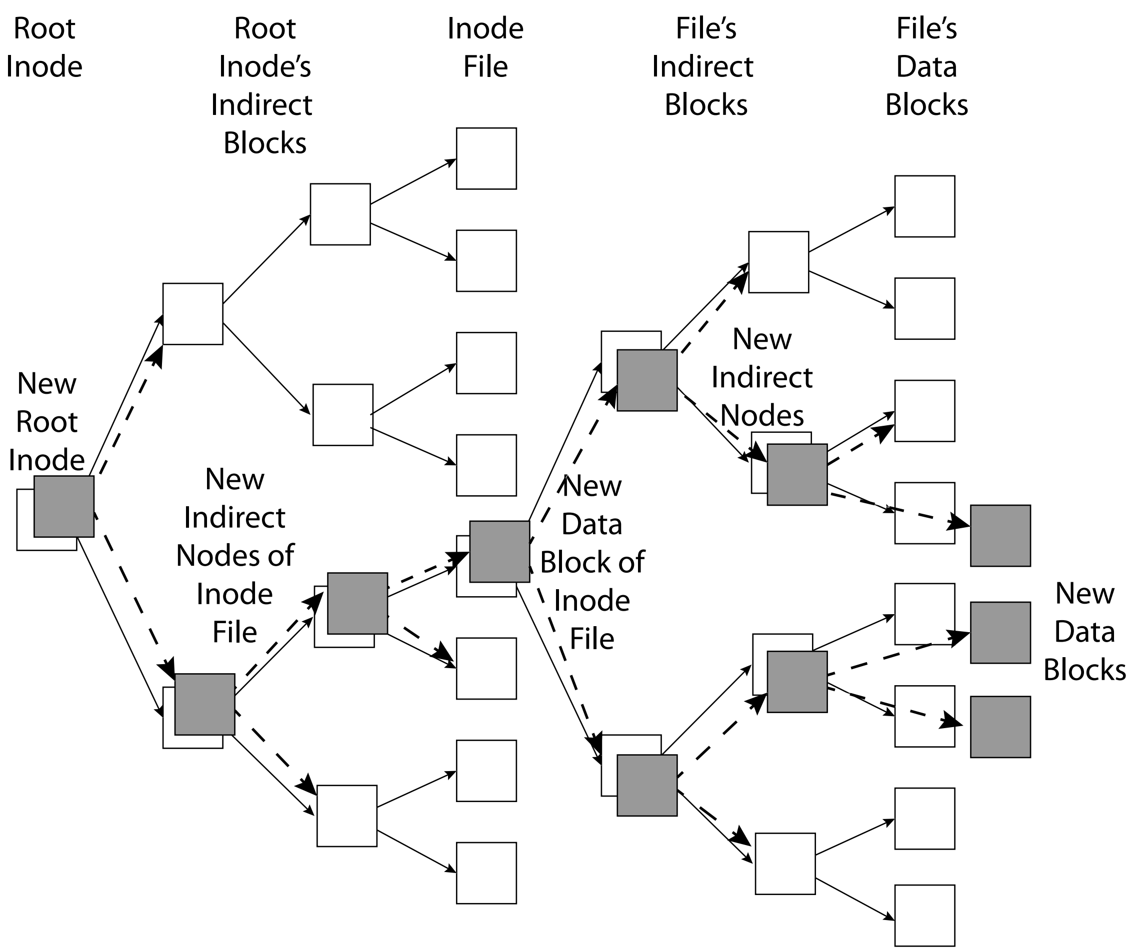
Copy on Write/Write Anywhere



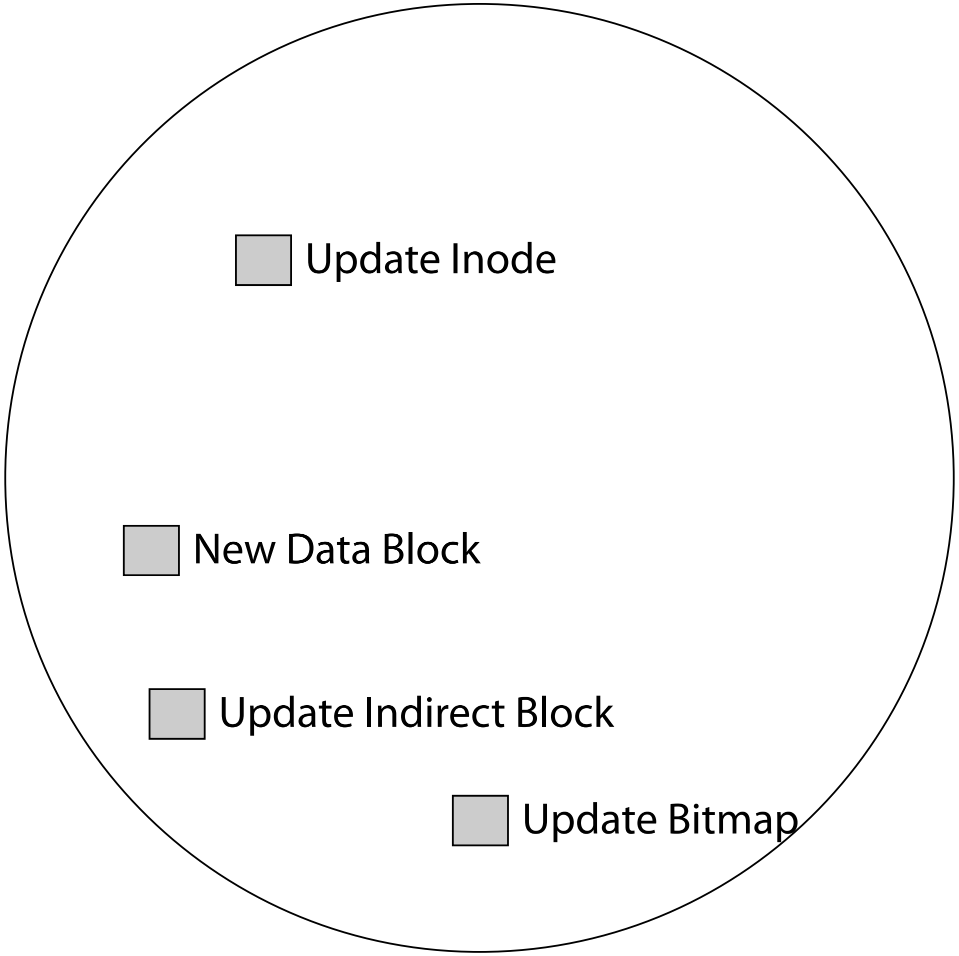
Copy on Write/Write Anywhere



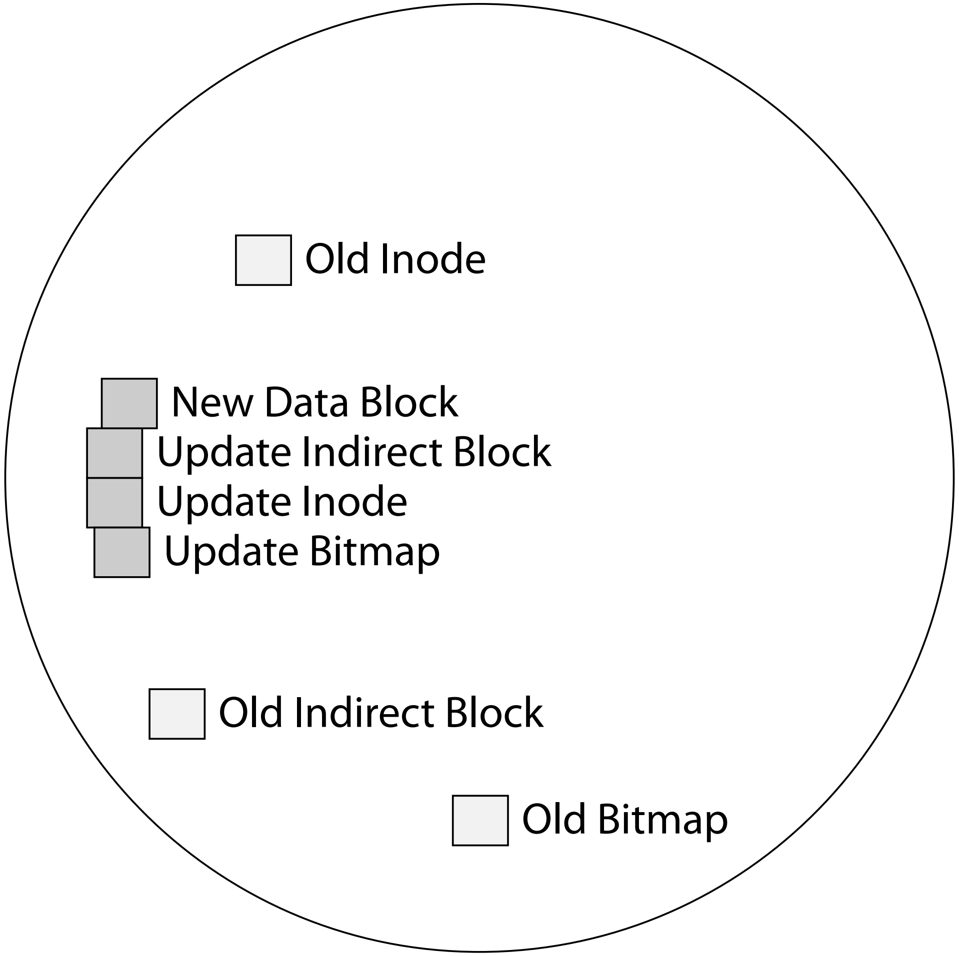
Copy on Write Batch Update



FFS Update in Place



WAFL Write Location



Copy on Write Garbage Collection

- For write efficiency, want contiguous sequences of free blocks
  - In every block group, near the disk head at all times
  - But updates scatter dead blocks
- For read efficiency, want related data to be in the same block group
  - But write anywhere can scatter related data

=> Background coalescing of live/dead blocks

Write Anywhere/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

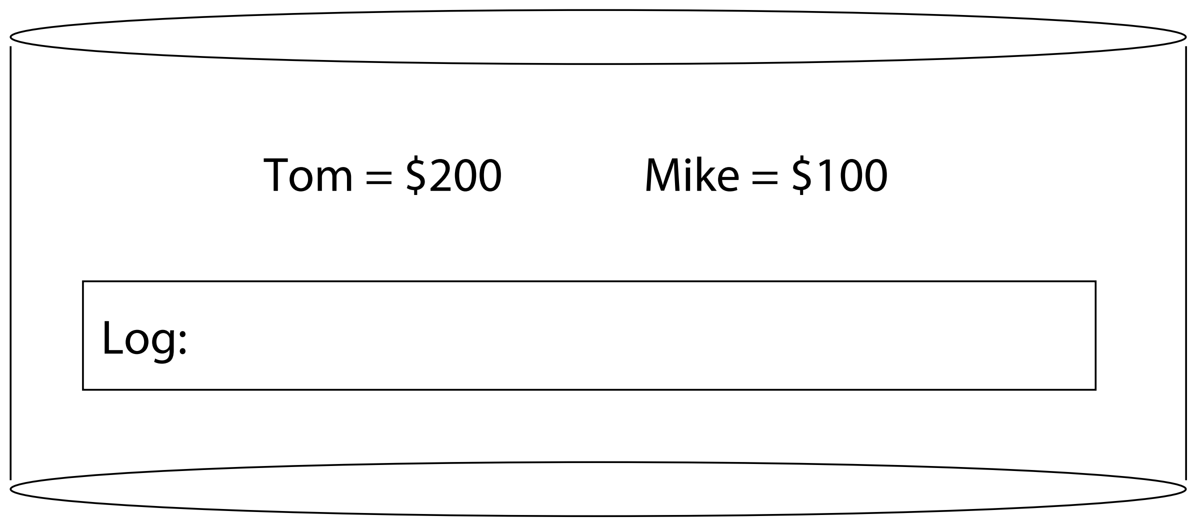
- Prepare
  - Write all changes (in transaction) to log
- Commit
  - Single disk write to make transaction durable
- Redo
  - Copy changes to disk
- Garbage collection
  - Reclaim space in log
- Recovery
  - Read log
  - Redo any operations for committed transactions
  - Garbage collect log

Before Transaction Start

Cache

Tom = $200 Mike = $100

Nonvolatile Storage

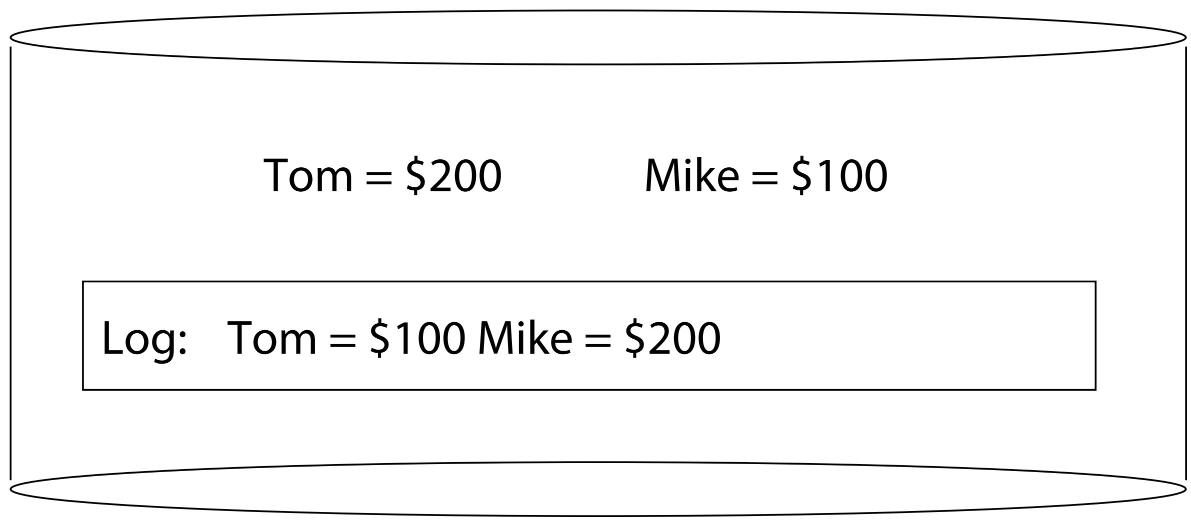


After Updates Are Logged

Cache

Tom = $100 Mike = $200

Nonvolatile Storage

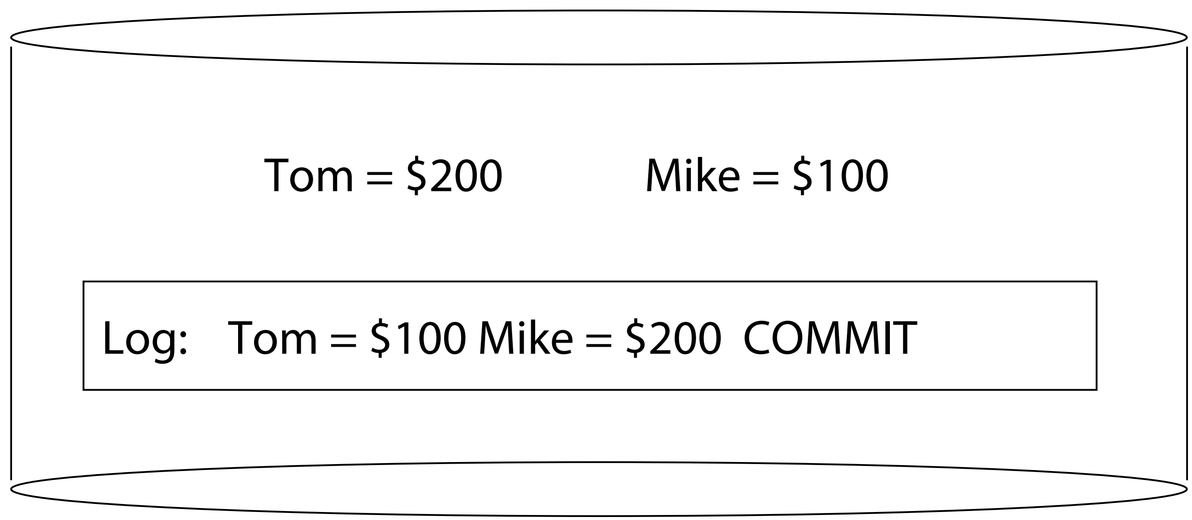


After Commit Logged

Cache

Tom = $100 Mike = $200

Nonvolatile Storage

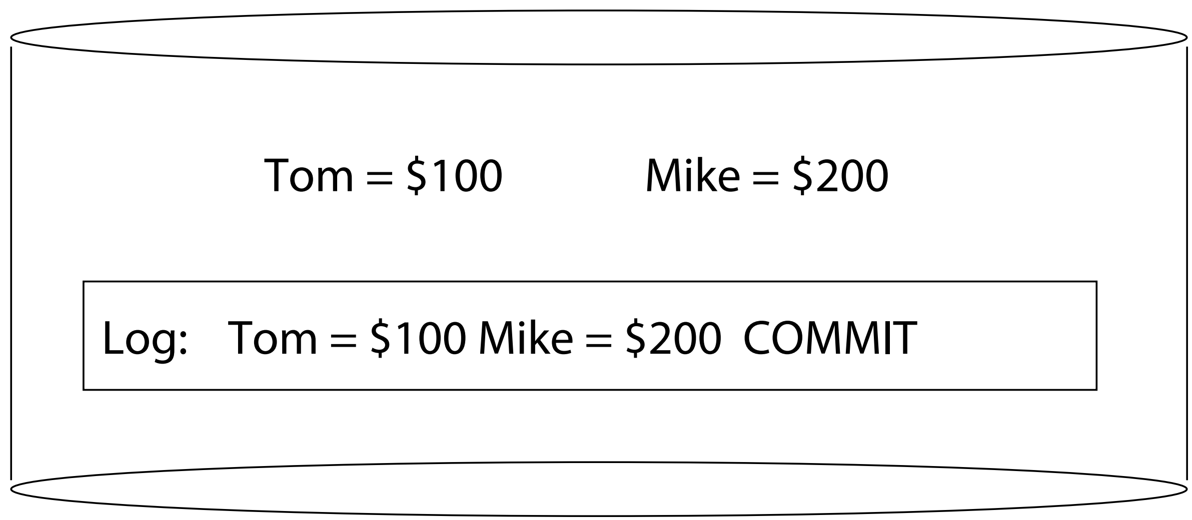


After Copy Back

Cache

Tom = $100 Mike = $200

Nonvolatile Storage

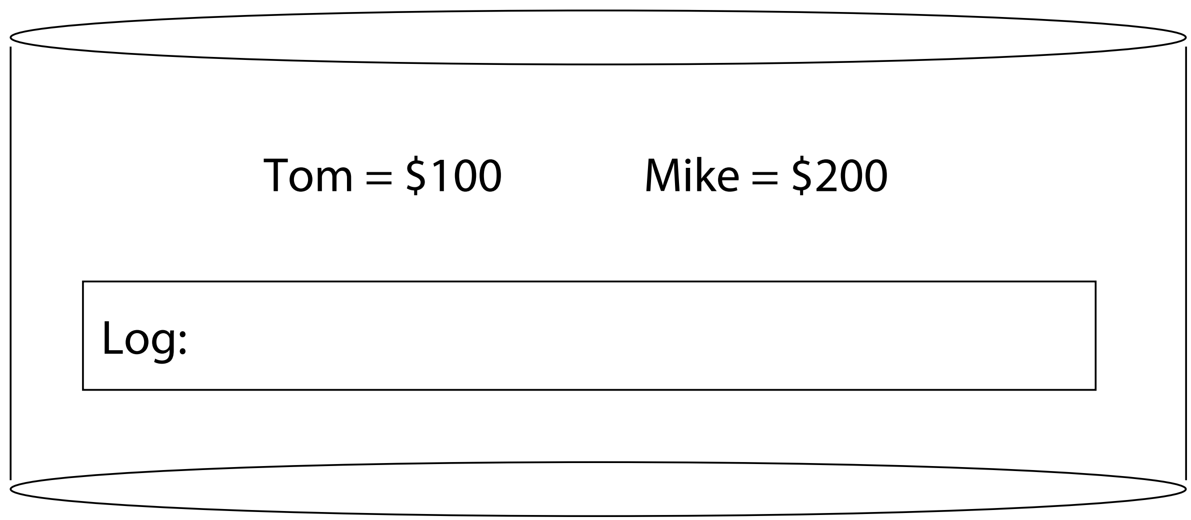


After Garbage Collection

Cache

Tom = $100 Mike = $200

Nonvolatile Storage



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

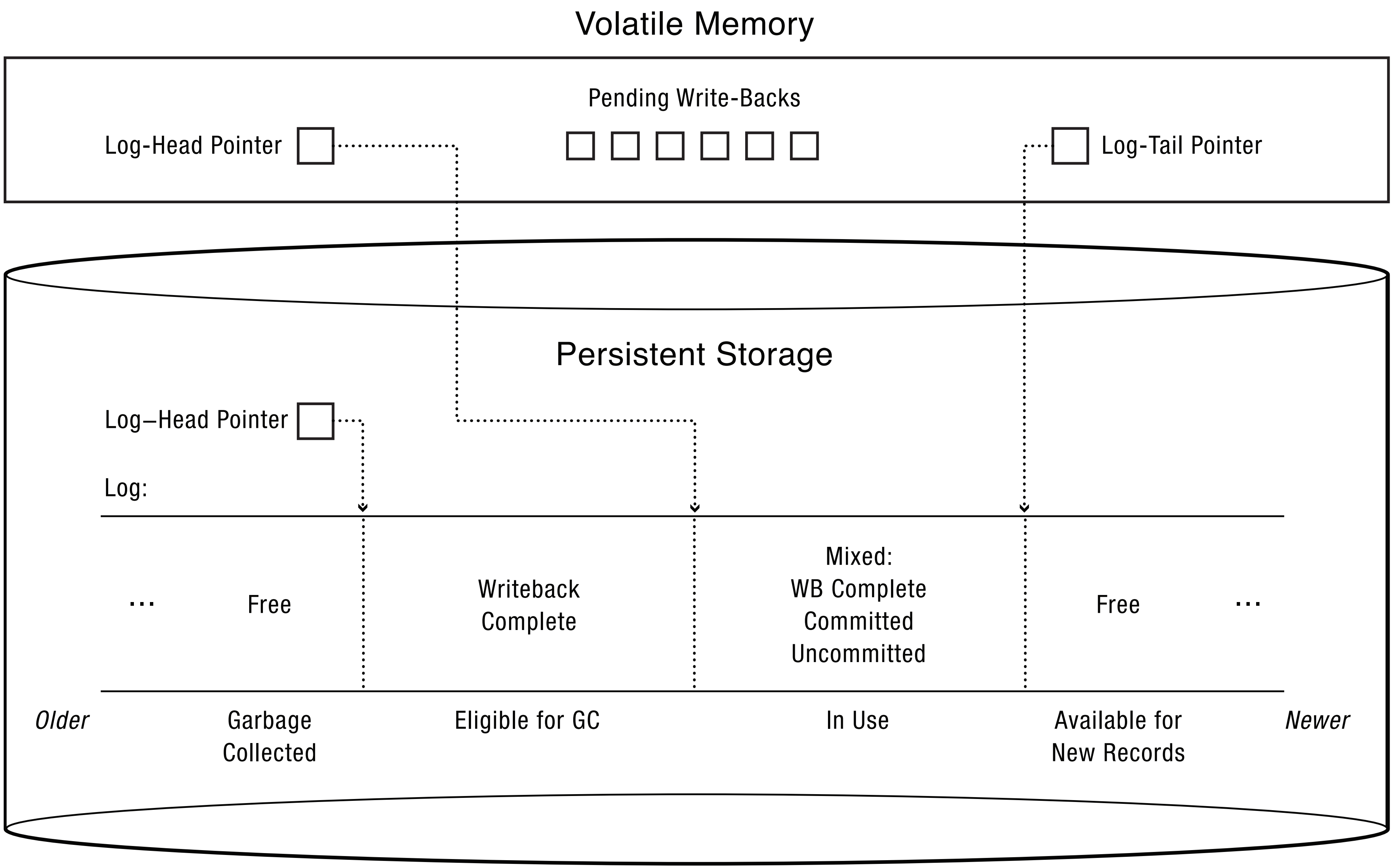
Questions

- 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

Flash Translation Layer

- Location independence
  - Virtual storage blocks can be stored in any physical location that contains zeros
  - Virtual blocks moved around to create completely empty erasure blocks (16-64 logical blocks)
  - Typically 10-20% more physical blocks than virtual
- Each erasure block has a map
  - Logical blocks stored in that erasure block
  - Stored persistently in flash, cached in DRAM
- Log structure
  - Write new blocks as a segmented log

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., 10MB)
  - Log new blocks into empty segment
  - Garbage collect dead blocks to create empty segments
  - Each segment contains map
    - Which blocks are stored in that segment
- Recovery
  - Read all segment headers
  - Find last successfully written segment

Storage Availability

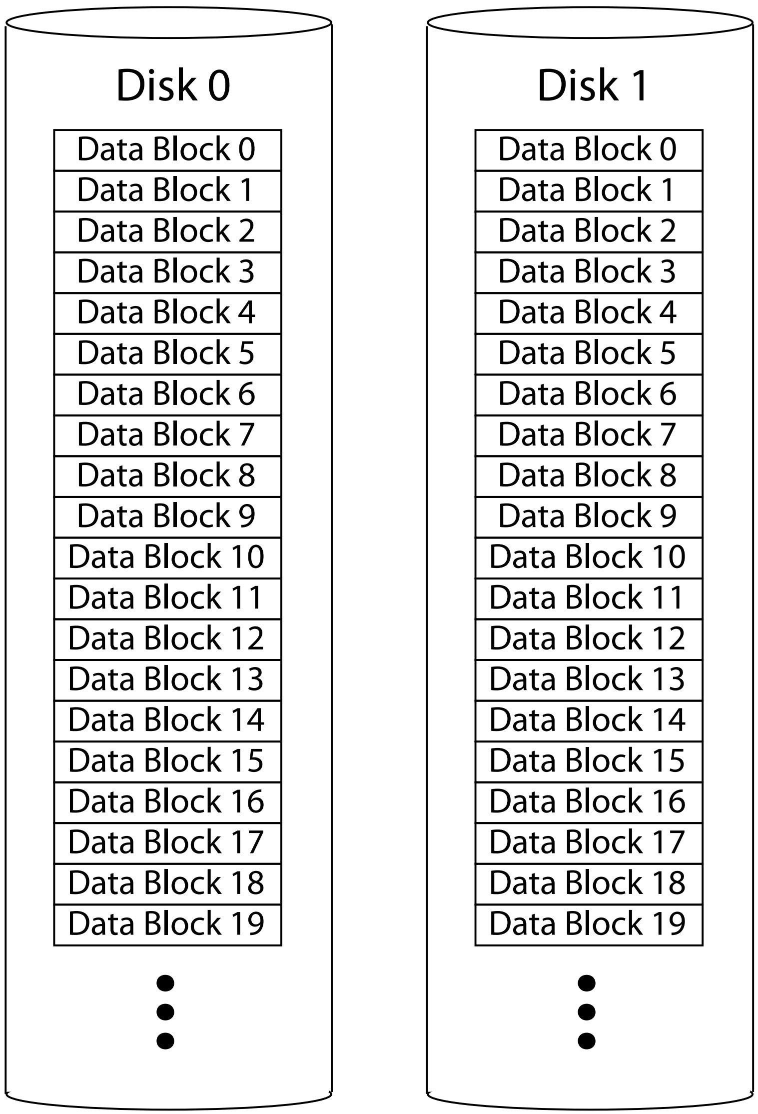
- Storage reliability: data fetched is what you stored
  - Transactions, redo logging, etc.
- Storage availability: data is there when you want it
  - More disks => higher probability of some disk failing
  - Data available ~ Prob(disk working)^k
    - If failures are independent and data is spread across k disks
  - For large k, probability system works -> 0

RAID

- Replicate data for availability
  - RAID 0: no replication
  - RAID 1: mirror data across two or more disks
    - Google File System replicated its 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



Parity

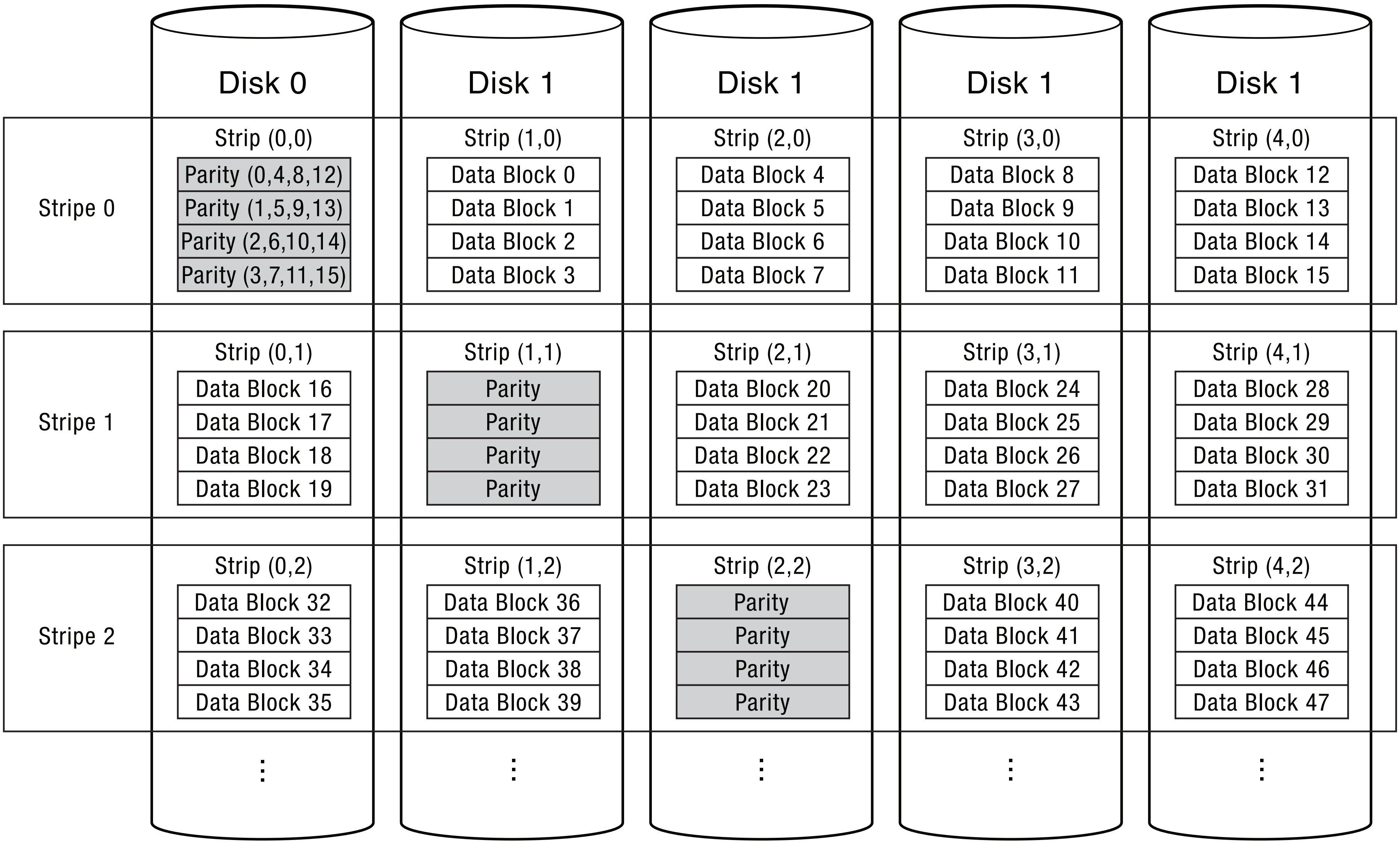
- Parity block: Block1 xor block2 xor block3 …

10001101	block1
01101100	block2
11000110	block3

00100111	parity block

- Can reconstruct any missing block from the others

RAID 5: Rotating Parity



RAID Update

- Mirroring
  - Write every mirror
- RAID-5: to write 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: to write 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 1 TB disks, and 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: two redundant disk blocks
    - parity, linear feedback shift
  - Scrubbing: read disk sectors in background to find and fix latent errors

Question

- Can you write a self-replicating C program?
  - program that when run, outputs itself
  - without reading any input files!
- printf(“printf(\”printf(\\\”printf(…