
Lecture 21: Reliable, High Performance Storage

CSC 469H1F

Fall 2006

Angela Demke Brown

Review

- We've looked at fault tolerance via server replication
 - Continue operating with up to f failures
 - Recovery requires restoring state on failed server
 - With replicated state machines, can rebuild state from non-faulty replicas unless another fault occurs
 - May require each server to maintain local state on stable storage to allow recovery to consistent global state following catastrophic failure

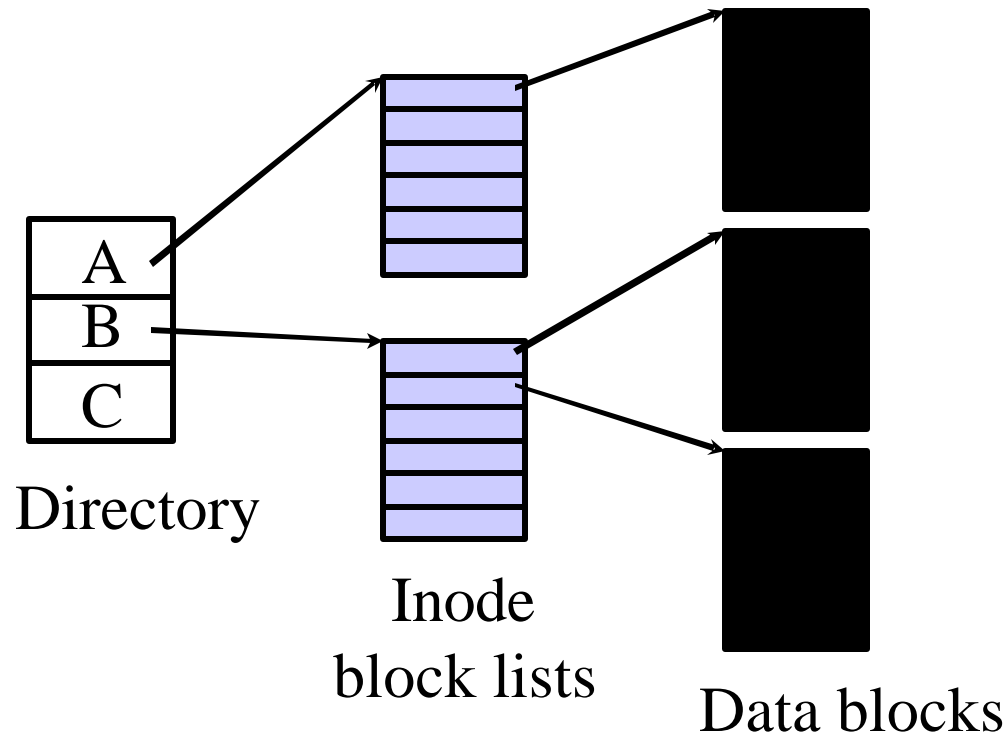
Overview

- Now we're going to look at some example file and storage systems designed for high performance and reliability
 - BSD Unix Fast File System (FFS) (review)
 - Log-structured File System (LFS)
 - Soft Updates
 - Redundant Array of Inexpensive Disks (RAID)

Basic Disk & File System properties

- Disks provide large-scale non-volatile storage
- Access time determined by:
 - Seek time
 - Rotational latency
 - Transfer time
- Good performance depends on
 - Reducing seeks
 - Creating large transfers
- File systems provide logical organization and management of data on disk

Unix Inodes: Indirection & Independence



File size grows dynamically, allocations are independent
Problem: hard to achieve closeness and amortization

Original Unix File System

- Recall FS sees storage as linear array of blocks
 - Each block has a logical block number (LBN)

Superblock

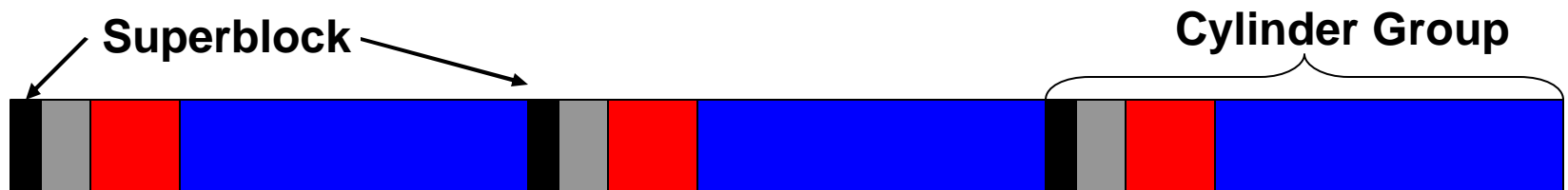
Default usage of LBN space



- Simple, straightforward implementation
 - Easy to implement and understand
 - Poor bandwidth utilization (lots of seeking)
 - Poor reliability (lose superblock, lose everything)

Cylinder Groups

- BSD FFS addressed placement problems using the notion of a **cylinder group** (aka allocation groups in lots of modern FS's)
 - Data blocks in same file allocated in same cylinder group
 - Files in same directory allocated in same cylinder group
 - Inodes for files allocated in same cylinder group as file data blocks
 - Superblock replicated to improve reliability



Cylinder group organization

Log-structured File System

- The Log-structured File System (LFS) was designed in response to two trends in workload and technology:
 1. Disk bandwidth is scaling significantly (40% a year)
 - Latency is not
 2. Large main memories in machines
 - Large buffer caches
 - Absorb large fraction of read requests
 - Can use for writes as well
 - Coalesce small writes into large writes
- LFS takes advantage of both of these to increase FS performance
 - Rosenblum and Ousterhout (Berkeley, '91)

LFS vs. FFS

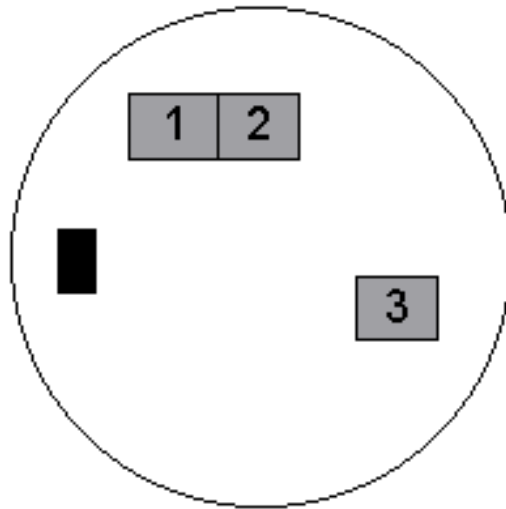
- LFS addresses some problems with FFS
 - Placement is improved, but still have many small seeks
 - Possibly related files are physically separated
 - Inodes separated from files (small seeks)
 - Directory entries separate from inodes
 - Metadata requires synchronous writes
 - With small files, most writes are to metadata (synchronous)
 - Synchronous writes very slow

LFS Approach

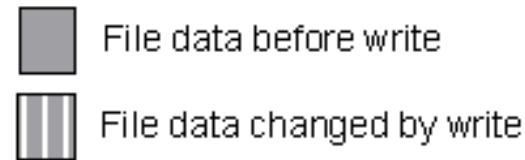
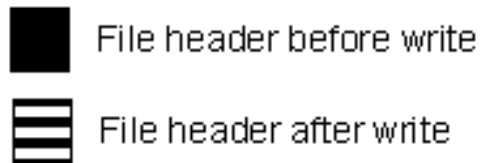
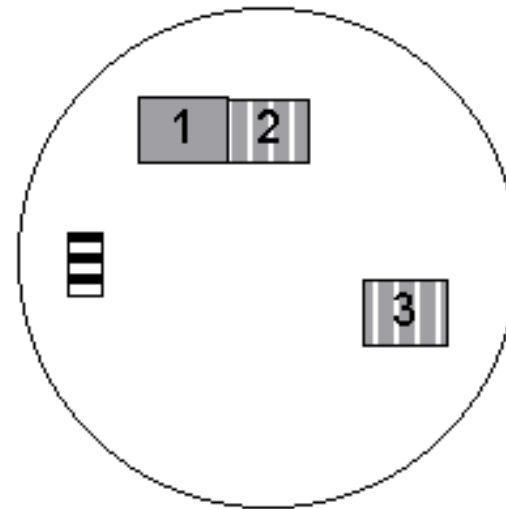
- Treat the disk as a single log for appending
 - Collect writes in disk cache, write out entire collection in one large disk request
 - Leverages disk bandwidth
 - No seeks (assuming read/write head is at end of log)
 - All info written to disk is appended to log
 - Data blocks, attributes, inodes, directories, etc.
- Simple, eh?
 - Alas, only in abstract

Example: Update file in FFS

Before...



After updating blocks 2 and 3...



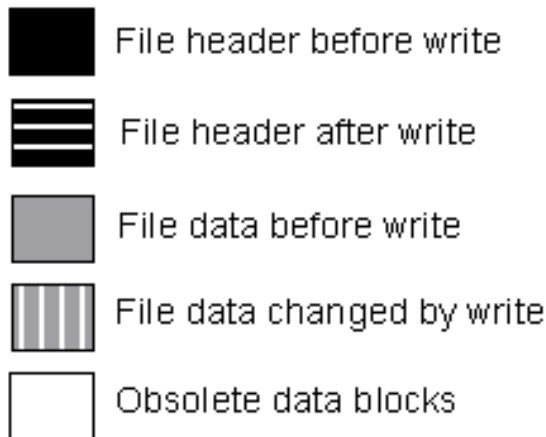
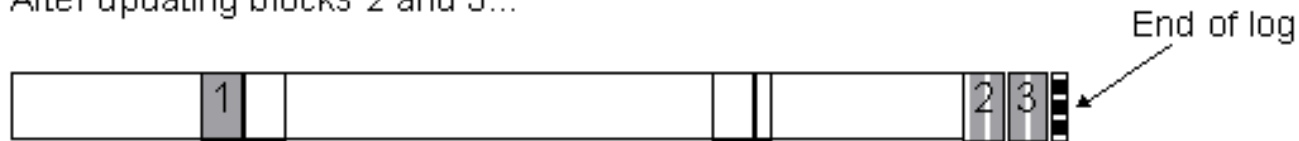
- Blocks (data & metadata are updated in place

Example: Update file in LFS

Before ...



After updating blocks 2 and 3...



- Modified blocks (data & metadata) are written to new location at end of log

LFS Challenges

- LFS has two challenges it must address for it to be practical
 1. Locating data written to the log
 - FFS places files in a location, LFS writes data "at the end"
 2. Managing free space on the disk
 - Disk is finite, so log is finite, cannot always append
 - Need to recover deleted blocks in old parts of log

LFS: Locating Data

- FFS uses inodes to locate data blocks
 - Inodes pre-allocated in each cylinder group
 - Directories contain locations of inodes
- LFS appends inodes to end of the log just like data
 - Makes them hard to find
- Approach
 - Use another level of indirection: **Inode maps**
 - **Inode maps** map file #s to inode location
 - Inode map blocks are written to log like any other block
 - Location of inode map blocks kept in **checkpoint region**
 - Checkpoint region has a fixed location
 - Cache inode maps in memory for performance

LFS: Free Space Management

- LFS append-only quickly runs out of disk space
 - Need to recover deleted blocks
- Approach:
 - Fragment log into segments
 - Thread active 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

LFS: Crash Recovery

- Traditional Unix FS: changes can be made anywhere
 - Must scan entire disk to restore metadata consistency after crash
- LFS: most recent changes are at end of log
 - Much faster/easier to achieve a consistent state
- LFS borrows 2 database techniques for recovery after a crash:
 - Checkpoints define a consistent file system state
 - Roll-forward recovers information written since the last checkpoint

LFS Checkpoints

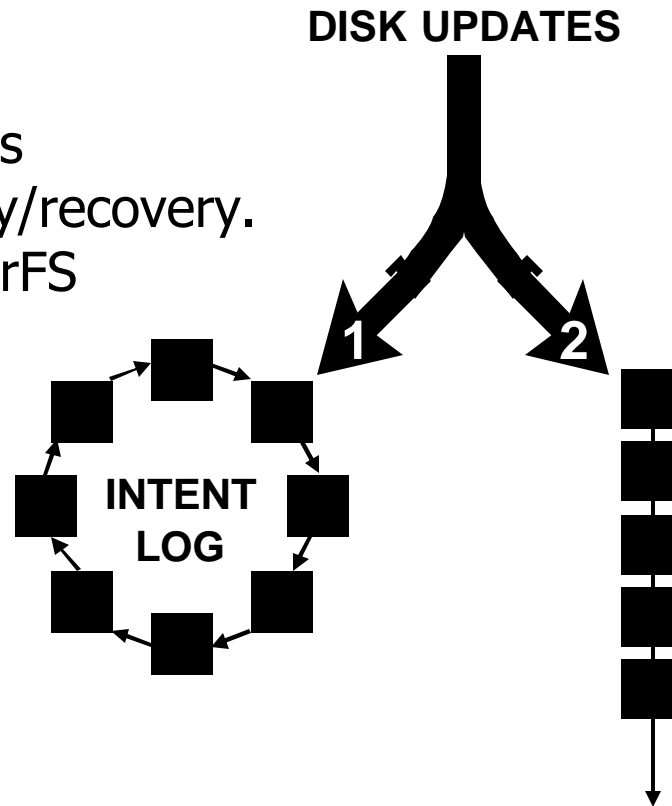
- Special, fixed location on disk containing:
 - Addresses of all inode map blocks
 - Segment usage table (for cleaning)
 - Address of last segment written
 - Timestamp
- Crash can occur while writing the checkpoint
 - Keep two checkpoint regions
 - Write timestamp out last
 - On recovery, use region with most recent timestamp

LFS Roll-Forward

- Can recover data written since last checkpoint
- Examine segments written since segment identified by checkpoint region
 - Segments contain special summary blocks and directory operation logs that record changes to file system metadata
 - Operation log entry is guaranteed to appear before modified directory or inode blocks
 - If directory or inode is inconsistent, operation log entry can be used to correct it

Write-ahead logging

Benefit of logging is strictly for reliability/recovery.
Used by JFS, ReiserFS



This approach journals file system updates to an intent log (1) before modifying the file itself (2).

Soft Updates

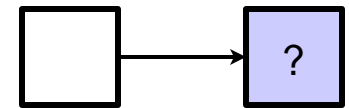
- Protect consistency of disks by controlling order of writes
 - use write-back caching for all (non-fsync) updates
 - make sure updates propagate to disk in the correct order
 - works great, but only goes as far as update ordering can go

Basic Update Ordering Rules

- Purpose: integrity of metadata pointers
 - in face of unpredictable system failures
 - similar to rules of programming with pointers

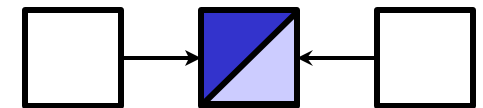
- Resource Allocation

- initialize resource before setting pointer



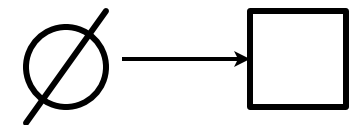
- Resource De-allocation

- nullify previous pointer before reuse



- Resource "Movement"

- set new pointer before nullifying old one



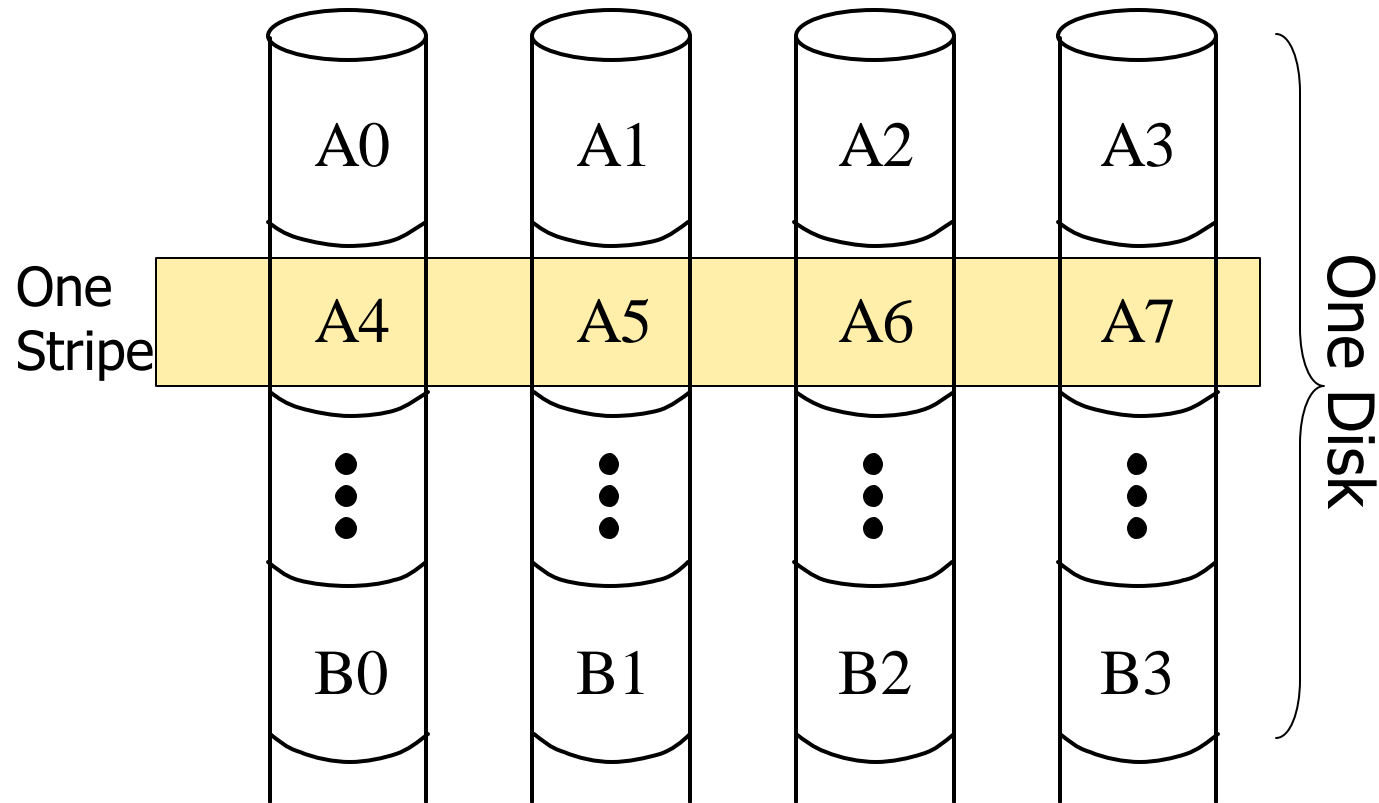
- Notice that something always left dangling

- ... assuming a badly-timed crash

RAID

- Redundant Array of Inexpensive Disks (RAID)
 - A storage system, not a file system
 - Patterson, Katz, and Gibson (Berkeley, '88)
- Idea: Use many disks in parallel to increase storage bandwidth, improve reliability
 - Files are striped across disks
 - Each stripe portion is read/written in parallel
 - Bandwidth increases with more disks

RAID Level 0: Disk Striping



Disk striping details

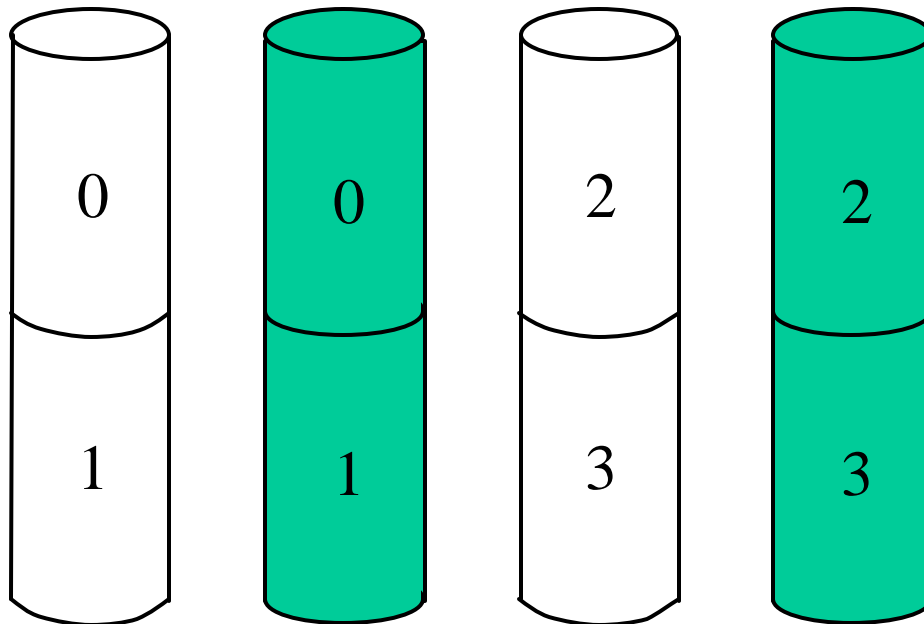
- How disk striping works
 - Break up total space into fixed-size **stripe units**
 - Distribute the stripe units among disks in round-robin fashion
 - Compute location of block #B as follows
 - $\text{disk\#} = B \% N$ ($\% = \text{modulo}$, $N = \# \text{ of disks}$)
 - $\text{LBN\#} = B / N$ (computes the LBN on given disk)
- Key design decision: picking the stripe unit size
 - too big: no parallel transfers and imperfect load balancing
 - too small: small transfers span stripe unit boundaries
 - also, should be a multiple of block size to assist alignment
- No redundancy

RAID 0 Challenges

- Small files (small writes less than a full stripe)
 - Need to read entire stripe, update with small write, then write entire stripe out to disks
- Reliability
 - More disks increases the chance of media failure (MTBF)
- Turn reliability problem into a feature
 - Use one disk to store parity data
 - XOR of all data blocks in stripe
 - Can recover any data block from all others + parity block
 - Hence "redundant" in name
 - Introduces overhead, but, hey, disks are "inexpensive"

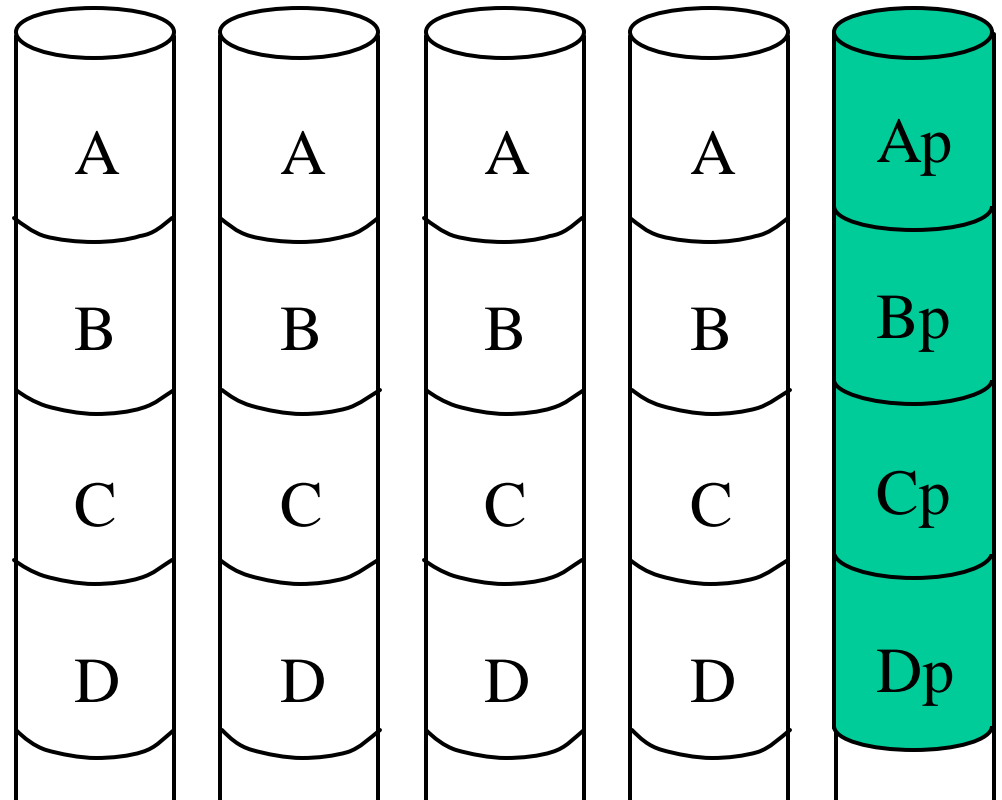
RAID Level 1: Mirroring

- Redundancy via replication, two (or more) copies
 - mirroring, shadowing, duplexing, etc.
- Write both, read either



Raid Level 2&3: Parity disks

- Both:
 - very small stripe unit (single byte or word)
 - All writes update parity disk(s)
 - Can correct single-bit errors
- Level 2:
 - #parity disks = $\log_2(\#data\ disks)$
 - overkill
- Level 3:
 - One extra disk



RAID Levels 4-6

- Levels 4-6 introduce independence:
 - Each disk may be writing different data
- Level 4 is similar to Level 3
- Level 5 removes parity disk bottleneck →
 - Distributes parity bits across all data disks
- Level 6 tolerates 2 errors

