Today: Coda, xFS

- Case Study: Coda File System

- Brief overview of other file systems
  - xFS
  - Log structured file systems

Distributed File System Requirements

- Transparency
  - Access, location, mobility, performance, scaling
- Concurrent file updates
- File replication
- Hardware and OS heterogeniety
- Fault tolerance
- Consistency
- Security
- Efficiency

AFS Background

- Alternative to NFS
- Originally a research project at CMU
  - Then, defunct commercial product from Transarc (IBM)
    - Public versions now available
  - Goal: Single namespace for global files (sound familiar? DNS? Web?)
- Designed for wide-area file sharing and scalability
- Example of stateful file system design
  - Server keeps track of clients accessing files
    - Uses callback-based invalidation
  - Eliminates need for timestamp checking, increases scalability
  - Complicates design

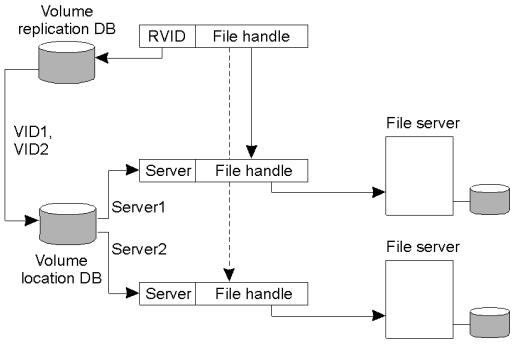
Coda Overview

- Offshoot of AFS designed for mobile clients
  - Observation: AFS does the work of migrating popular/ necessary files to your machine on access
  - Nice model for mobile clients who are often disconnected
    - Use file cache to make ***disconnection*** transparent
    - At home, on the road, away from network connection
- Coda supplements AFS file cache with user preferences
  - E.g., always keep this file in the cache
  - Supplement with system learning user behavior
- Consistency issues?

Coda Consistency

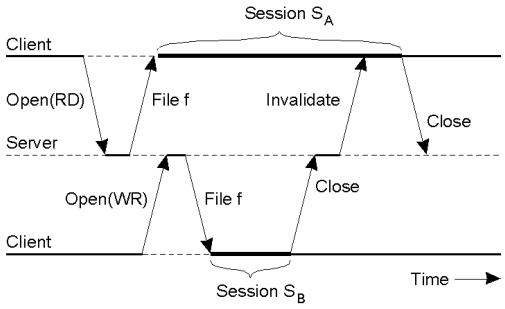
- How to keep cached copies on disjoint hosts consistent?
  - In mobile environment, "simultaneous" writes can be separated by hours/days/weeks
- Callbacks cannot work since no network connection is available
- Coda approach (in order):
  - Assume that write sharing the rare case
  - Attempt automatic patch
  - Fallback to manual (user) intervention

File Identifiers



- Each file in Coda belongs to exactly one volume
  - Volume may be replicated across several servers
  - Multiple logical (replicated) volumes map to the same physical volume
  - 96 bit file identifier = 32 bit RVID + 64 bit file handle

Sharing Files in Coda

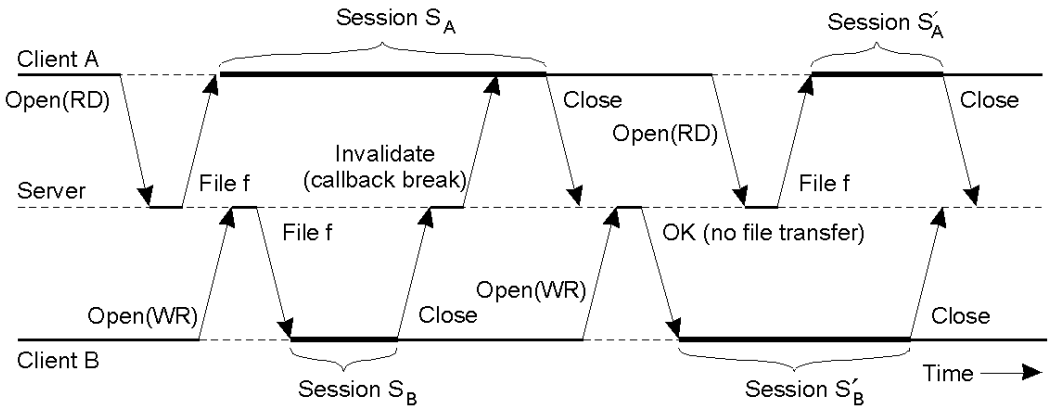


- Transactional behavior for sharing files: similar to share reservations in NFS
  - File open: transfer entire file to client machine [similar to delegation]
  - Uses session semantics: each session is like a transaction
    - Updates are sent back to the server only when the file is closed

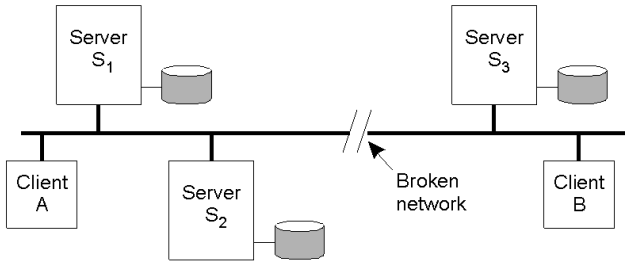
Transactional Semantics

- Network partition: part of network isolated from rest
  - Allow conflicting operations on replicas across file partitions
  - Reconcile upon reconnection
  - Transactional semantics => operations must be serializable
    - Ensure that operations were serializable after thay have executed
  - Conflict => force manual reconciliation

Client Caching

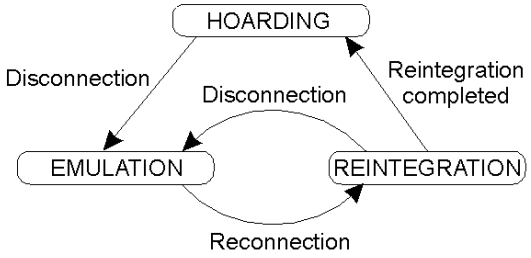


Server Replication



- Use replicated writes: read-once write-all
  - Writes are sent to all AVSG (all accessible replicas)
- How to handle network partitions?
  - Use optimistic strategy for replication
  - Detect conflicts using a Coda version vector
  - Example: [2,2,1] and [1,1,2] is a conflict => manual reconciliation

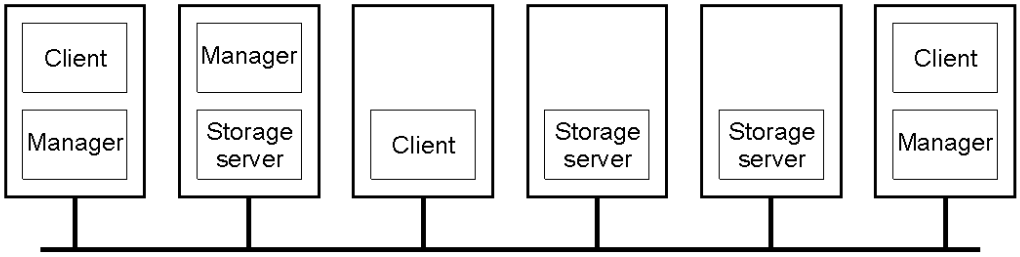
Disconnected Operation



- The state-transition diagram of a Coda client with respect to a volume.
- Use hoarding to provide file access during disconnection
  - Prefetch all files that may be accessed and cache (hoard) locally
  - If AVSG=0, go to emulation mode and reintegrate upon reconnection

Overview of xFS.

- Key Idea: fully distributed file system [serverless file system]
  - Remove the bottleneck of a centralized system
- xFS: x in "xFS" => no server
- Designed for high-speed LAN environments



xFS Summary

- Distributes data storage across disks using software RAID and log-based network striping
  - RAID == Redundant Array of Independent Disks
- Dynamically distribute control processing across all servers on a per-file granularity
  - Utilizes serverless management scheme
- Eliminates central server caching using cooperative caching
  - Harvest portions of client memory as a large, global file cache.

RAID Overview

- Basic idea: files are "striped" across multiple disks
- Redundancy yields high data availability
  - Availability: service still provided to user, even if some components failed
- Disks will still fail
- Contents reconstructed from data redundantly stored in the array
  - Capacity penalty to store redundant info
  - Bandwidth penalty to update redundant info

Slides courtesy David Patterson

Replace Small Number of Large Disks with Large Number of Small Disks! (1988 Disks)

	IBM 3390K	IBM 3.5" 0061	x70	
Capacity	20 GBytes	320 MBytes	23 GBytes	
Volume	97 cu. ft.	0.1 cu. ft.	11 cu. ft.	9X
Power	3 KW	11 W	1 KW	3X
Data Rate	15 MB/s	1.5 MB/s	120 MB/s	8X
I/O Rate	600 I/Os/s	55 I/Os/s	3900 IOs/s	6X
MTTF	250 KHrs	50 KHrs	??? Hrs	
Cost	$250K	$2K	$150K	

Disk Arrays have potential for large data and I/O rates, high MB per cu. ft., high MB per KW, but what about reliability?

Array Reliability

- Reliability of N disks = Reliability of 1 Disk ÷ N

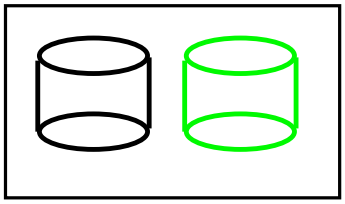
50,000 Hours ÷ 70 disks = 700 hours

Disk system MTTF: Drops from 6 years to 1 month!

- Arrays (without redundancy) too unreliable to be useful!

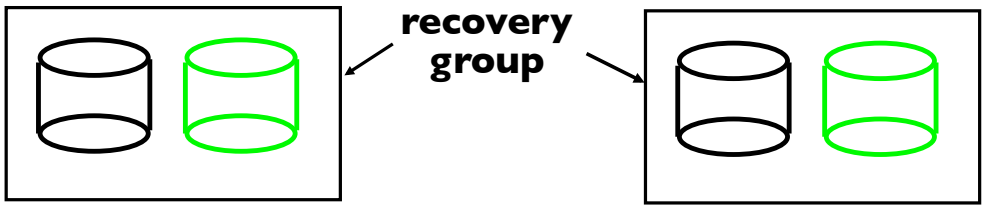
Hot spares support reconstruction in parallel with access: very high media availability can be achieved

Redundant Arrays of Inexpensive Disks
RAID 0: Striping



- Stripe data at the block level across multiple disks
- High read and write bandwidth
- Not a true RAID since no "redundancy"
- Failure of any one drive will cause the entire array to become unavailable

Redundant Arrays of Inexpensive Disks
RAID 1: Disk Mirroring/Shadowing



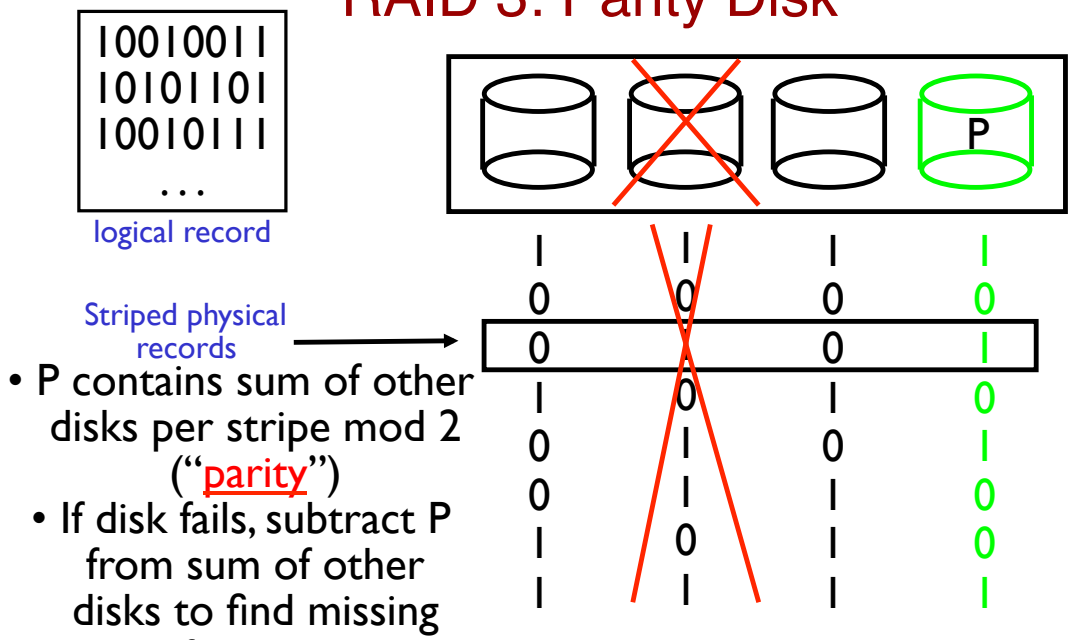
- Each disk is fully duplicated onto its "mirror"
- Very high availability can be achieved
- Bandwidth sacrifice on write:
- Logical write = two physical writes
- Reads may be optimized
- Most expensive solution: 100% capacity overhead

- (RAID 2 not interesting, so skip…involves Hamming codes)

RAID-I

- RAID-I (1989)
  - Consisted of a Sun 4/280 workstation with 128 MB of DRAM, four dual-string SCSI controllers, 28 5.25-inch SCSI disks and specialized disk striping software

Redundant Array of Inexpensive Disks
RAID 3: Parity Disk



- P contains sum of other disks per stripe mod 2 ("parity")
- If disk fails, subtract P from sum of other disks to find missing information

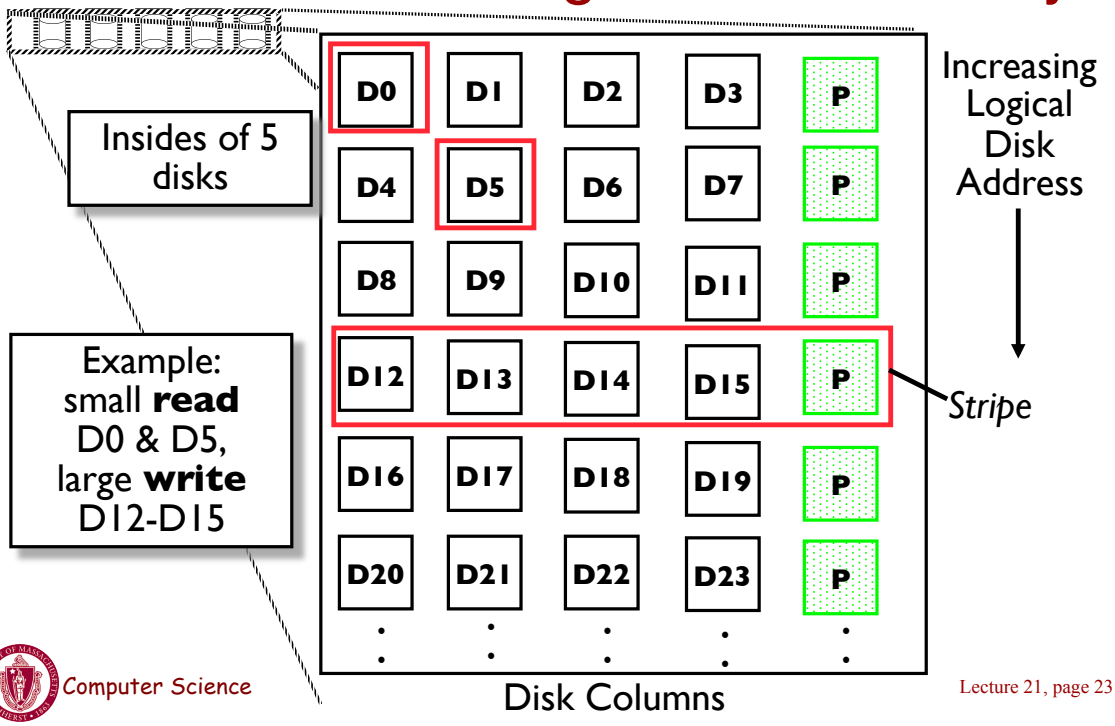
RAID 3

- Sum computed across recovery group to protect against hard disk failures, stored in P disk
- Logically, a single high capacity, high transfer rate disk: good for large transfers
- But byte level striping is bad for small files (all disks involved)
- Parity disk is still a bottleneck

Inspiration for RAID 4

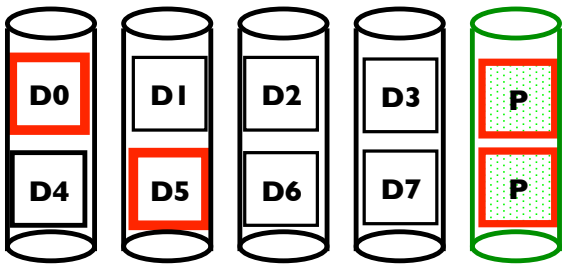
- RAID 3 stripes data at the byte level
- RAID 3 relies on parity disk to discover errors on read
- But every sector on disk has an error detection field
- Rely on error detection field to catch errors on read, not on the parity disk
- Allows independent reads to different disks simultaneously
- Increases read I/O rate since only one disk is accessed rather than all disks for a small read

Redundant Arrays of Inexpensive Disks RAID 4: High I/O Rate Parity

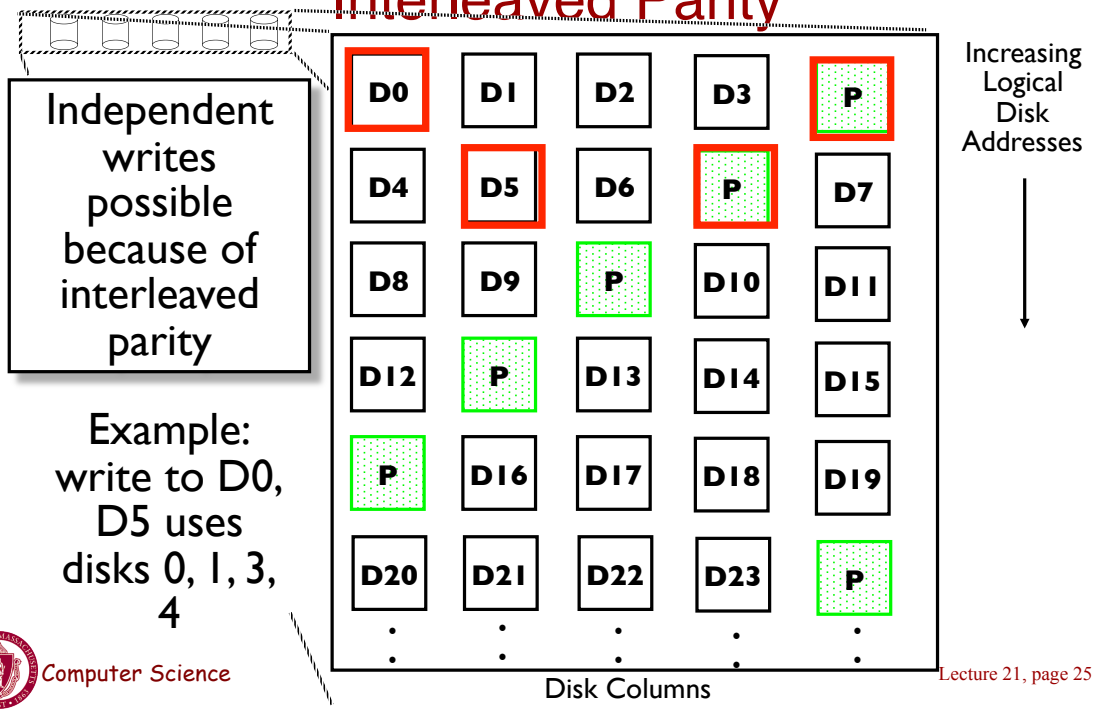


Inspiration for RAID 5

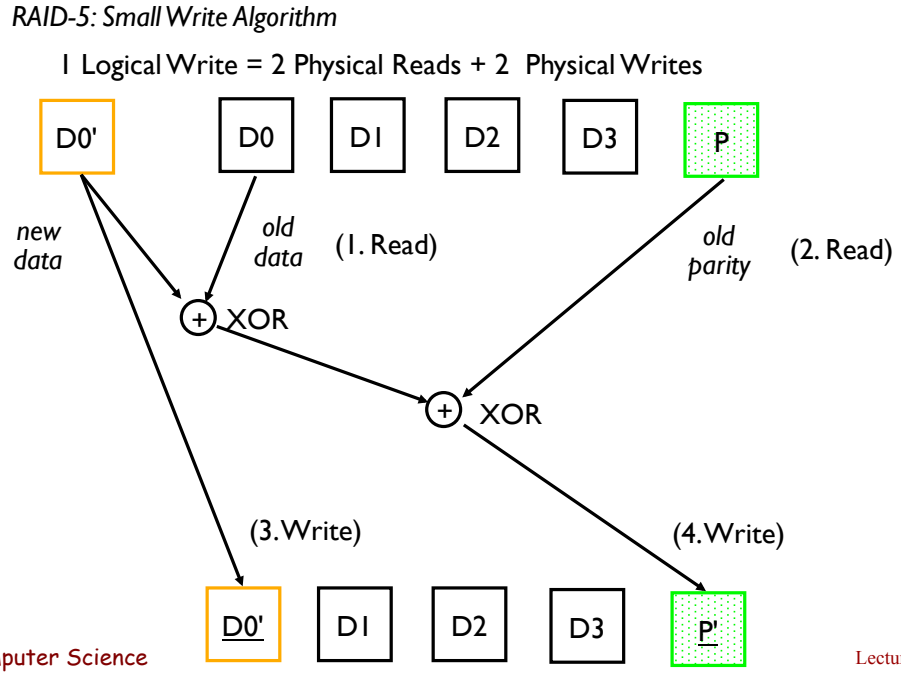
- RAID 4 works well for small reads
- Small writes (write to one disk):
  - Option 1: read other data disks, create new sum and write to Parity Disk
  - Option 2: since P has old sum, compare old data to new data, add the difference to P
- Small writes are still limited by Parity Disk: Write to D0, D5, both also write to P disk



Redundant Arrays of Inexpensive Disks RAID 5: High I/O Rate Interleaved Parity



Problems of Disk Arrays: Small Writes



xFS uses software RAID

- Two limitations
  - Overhead of parity management hurts performance for small writes
    - Ok, if overwriting all N-1 data blocks
    - Otherwise, must read old parity+data blocks to calculate new parity
    - Small writes are common in UNIX-like systems
  - Very expensive since hardware RAIDS add special hardware to compute parity

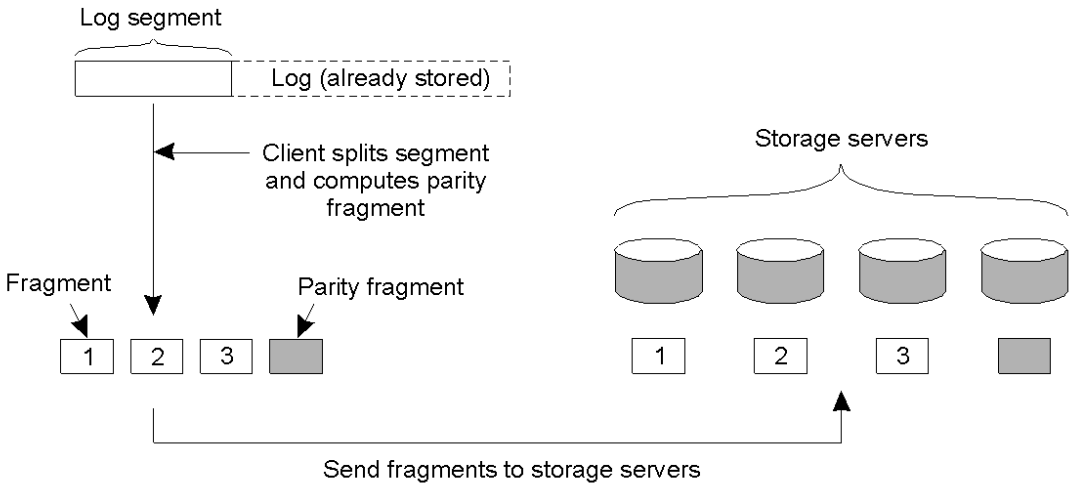
Log-structured FS

- Provide fast writes, simple recovery, flexible file location method
- Key Idea: buffer writes in memory and commit to disk in large, contiguous, fixed-size log segments
  - Complicates reads, since data can be anywhere
  - Use per-file inodes that move to the end of the log to handle reads
  - Uses in-memory imap to track mobile inodes
    - Periodically checkpoints imap to disk
    - Enables "roll forward" failure recovery
- Drawback: must clean "holes" created by new writes

Combine LFS with software RAID

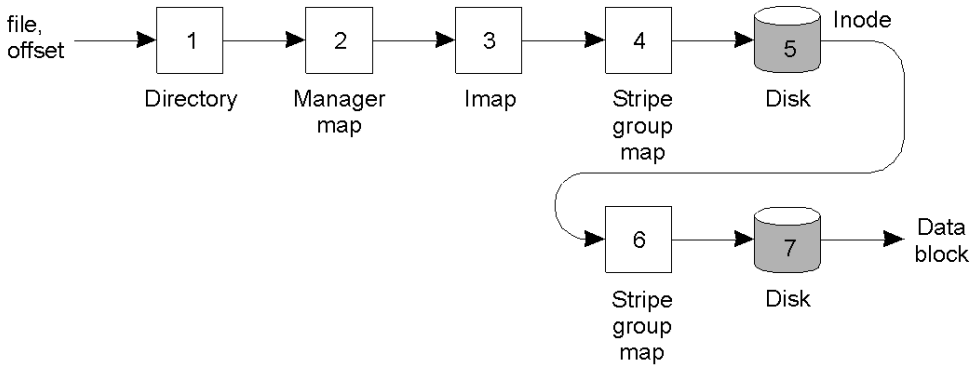
- Addresses small write problem
  - Each log segment spans a RAID stripe
  - Avoids the parity recomputation
- xFS maintains 4 maps
  - File directory Map
    - Maps human-readable names to index number
  - Manager Map
    - Determines which manager to contact for a file
  - Imap
    - Determines where to look for file in on-disk log
  - Stripe group Map
    - Mapping to set of physical machines storing segments

Processes in xFS



Reading a File Block

- Reading a block of data in xFS.



xFS Naming

- Main data structures used in xFS.

Data structure	Description
Manager map	Maps file ID to manager
Imap	Maps file ID to log address of file's inode
Inode	Maps block number (i.e., offset) to log address of block
File identifier	Reference used to index into manager map
File directory	Maps a file name to a file identifier
Log addresses	Triplet of stripe group, ID, segment ID, and segment offset
Stripe group map	Maps stripe group ID to list of storage servers

Transactional Semantics

File-associated data	Read?	Modified?
File identifier	Yes	No
Access rights	Yes	No
Last modification time	Yes	Yes
File length	Yes	Yes
File contents	Yes	Yes

- Network partition: part of network isolated from rest
  - Allow conflicting operations on replicas across file partitions
  - Reconcile upon reconnection
  - Transactional semantics => operations must be serializable
    - Ensure that operations were serializable after thay have executed
  - Conflict => force manual reconciliation