CS 422/522 Design & Implementation of Operating Systems

Lecture 17: Reliable Storage

Zhong Shao
Dept. of Computer Science
Yale University

1

Main points

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

2

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

3

File system reliability

- For performance, all must be cached!
  This is OK for reads but what about writes?
- Options for writing data:
  Write-through: write change immediately to disk
  Problem: slow! Have to wait for write to complete before you go on

  Write-back: delay writing modified data back to disk (for example, until replaced)
  Problem: can lose data on a crash!

4

Multiple updates

- If multiple updates needed to perform some operations, crash can occur between them!
  - Moving a file between directories:
    - * Delete file from old directory
    - * Add file to new directory
  - Create new file
    - * Allocate space on disk for header, data
    - * Write new header to disk
    - * Add the new file to directory

What if there is a crash in the middle? Even with write-through it can still have problems

5

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?

6

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
  - 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

7

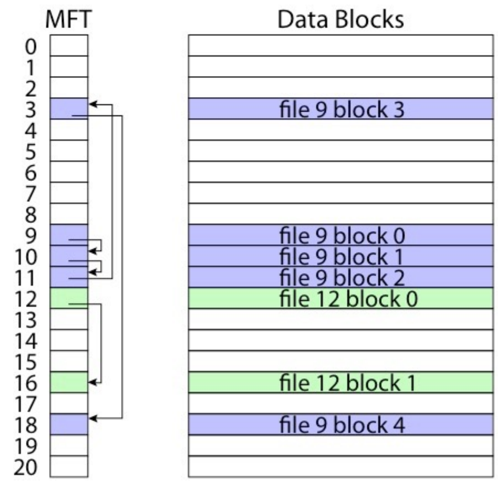
Reliability approach #1: careful ordering

- 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)

8

FAT: Append data to file

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



9

FAT: Append data to file

Normal operation:

- Add data block
- Add pointer 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, delete data block
- If access time is incorrect, update

10

FAT: Create new file

Normal operation:
- Allocate data block
- Update MFT entry to point to data block
- Update directory with file name -> file number
- Update modify time for directory

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

11

Unix approach (careful reordering)

Try to achieve consistency on both meta-data and user data !

- **Meta-data**: needed to keep file system logically consistent
  - Directories
  - Bitmap
  - File headers
  - Indirect blocks
  - ……
- **Data**: user bytes

12

Meta-data consistency (ad hoc)

- For meta-data, Unix uses "synchronous write through".
  - If multiple updates needed, Unix does them in specific order
  - If it crashes, run the special program "fsck" which scans the entire disk for internal consistency to check for "in progress" operations and then fixes up anything in progress.

  Example:

  File created, but not yet put in any directory → delete file

  Blocks allocated, but not in bitmap → update bitmap

13

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

14

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?

15

User data consistency

- For user data, Unix uses "write back" --- forced to disk every 30 seconds (or user can call "sync" to force to disk immediately).

  No guarantee blocks are written to disk in any order.

  Sometimes meta-data consistency is good enough

  How should vi save changes to a file to disk ?
    - Write new version in temp file
    - Move old version to other temp file
    - Move new version into real file
    - Unlink old version

  If crash, look at temp area; if any files out there, send email to user that there might be a problem.

16

Application level

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

17

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

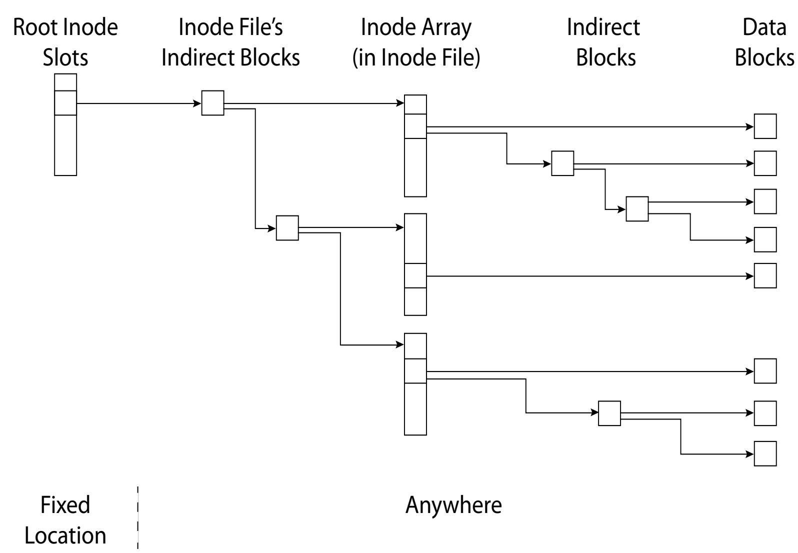
18

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)

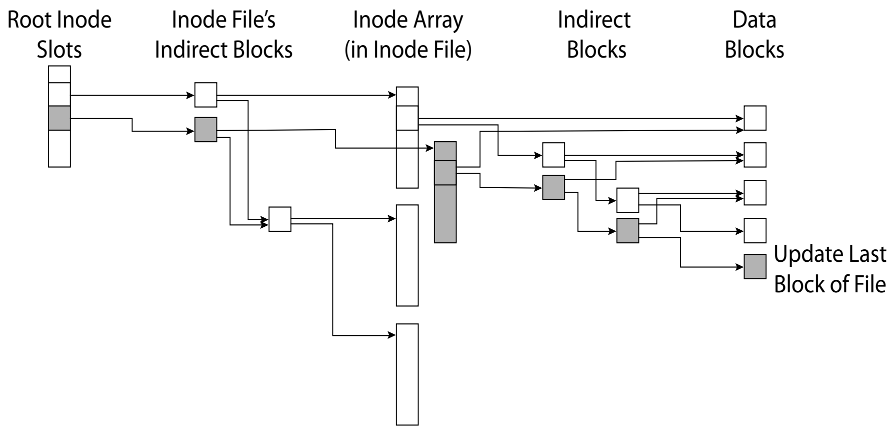
19

Copy on write/write anywhere



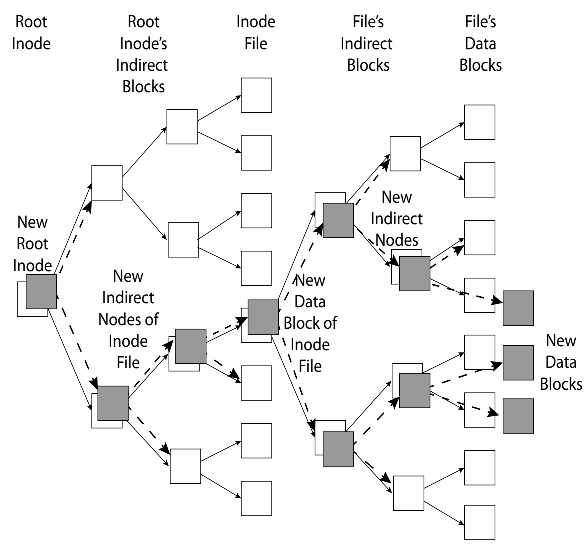
20

Copy on write/write anywhere



21

Copy on write batch update



22

Copy on write garbage collection

- For write efficiency, want contiguous sequences of free blocks
  - Spread across all block groups
  - Updates leave dead blocks scattered
- For read efficiency, want data read together to be in the same block group
  - Write anywhere leaves related data scattered

=> Background coalescing of live/dead blocks

23

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

24

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

25

Transaction concept

- **Transactions:** group actions together so that they are
  - **Atomic**: either happens or it does not (no partial operations)
  - **Serializable**: transactions appear to happen one after the other
  - **Durable**: once it happens, stays happened

  Critical sections are atomic and serializable, but not durable

  Need two more items:

  Commit --- when transaction is done (durable)

  Rollback --- if failure during a transaction (means it didn't happen at all)
- Metaphor: do a set of operations tentatively. If get to commit, ok. Otherwise, roll back the operations as if the transaction never happened.

26

Transaction implementation

- Key idea: fix problem of how you make multiple updates to disk, by turning multiple updates into a single disk write!
- Example: money transfer from account x to account y:

```
Begin transaction
   x  = x + 1
   y  =  y – 1
Commit
```

- Keep "redo" log on disk of all changes in transaction.
  - A log is like a journal, never erased, record of everything you' ve done
  - Once both changes are on log, transactions is committed.
  - Then can "write behind" changes to disk --- if crash after commit, replay log to make sure updates get to disk

27

Transaction implementation (cont' d)

X=1	Y=1	commit

write-ahead log (on disk or tape or non-volatile RAM)

Sequence of steps to execute transaction:

1. Write new value of X to log
2. Write new value of Y to log
3. Write commit
4. Write x to disk
5. Write y to disk
6. Reclaim space on log

28

Transaction implementation (cont' d)

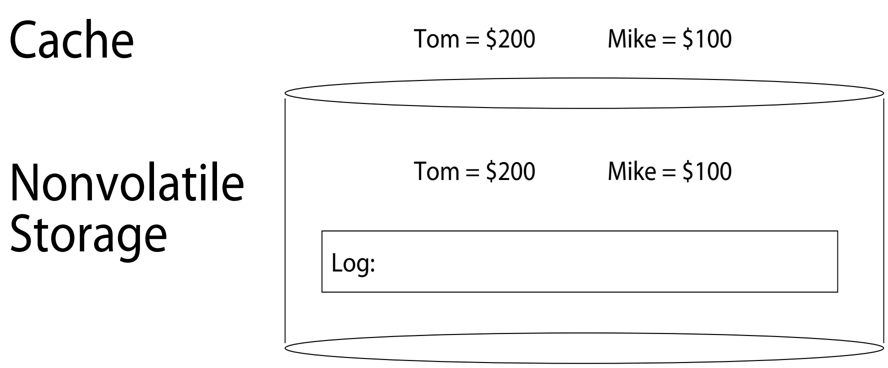
X=1	Y=1	commit

1. Write new value of X to log
2. Write new value of Y to log
3. Write commit
4. Write x to disk
5. Write y to disk
6. Reclaim space on log

- **What if we crash after 1?**
  - No commit, nothing on disk, so just ignore changes
- **What if we crash after 2?** Ditto
- **What if we crash after 3 before 4 or 5?**
  - Commit written to log, so replay those changes back to disk
- **What if we crash while we are writing "commit" ?**
  - As with concurrency, we need some primitive atomic operation or else can' t build anything. (e.g., writing a single sector on disk is atomic!)

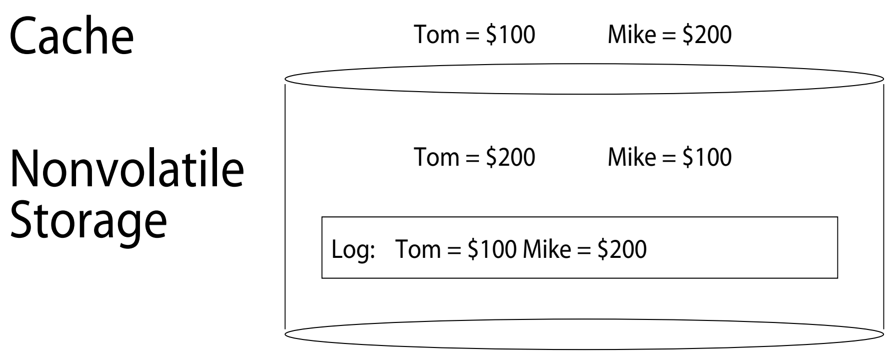
29

Another example: before transaction start



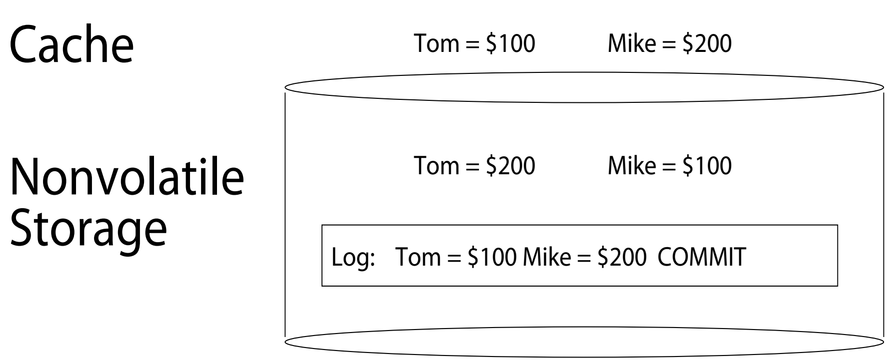
30

After updates are logged



31

After commit logged

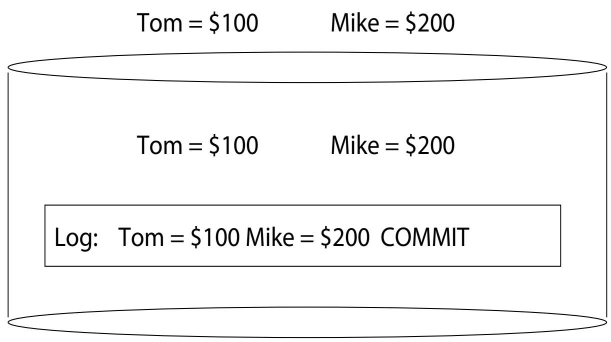


32

After copy back

Cache

Nonvolatile Storage

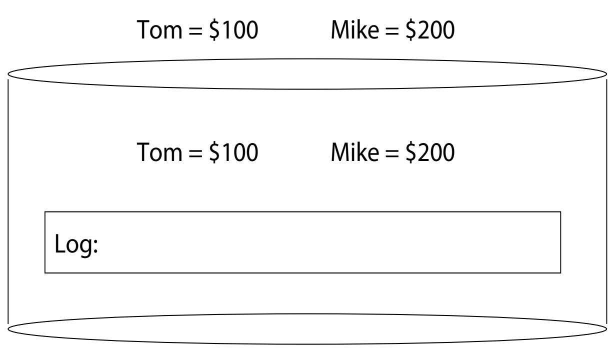


33

After garbage collection

Cache

Nonvolatile Storage



34

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

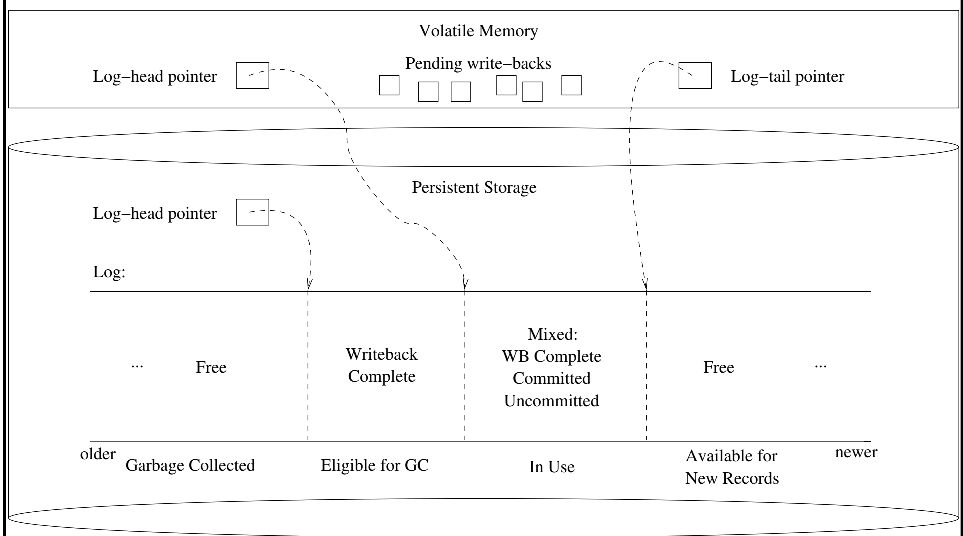
35

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

36

Redo log implementation



37

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?

38

Two-phase locking

- Don't allow "unlock" before commit.
- First phase: only allowed to acquire locks (this avoids deadlock concerns).
- Second phase: all unlocks happen at commit
- Thread B can't see any of A's changes, until A commits and releases locks. This provides serializability.

39

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

40

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

41

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

42

Question

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

43

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

44

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

45

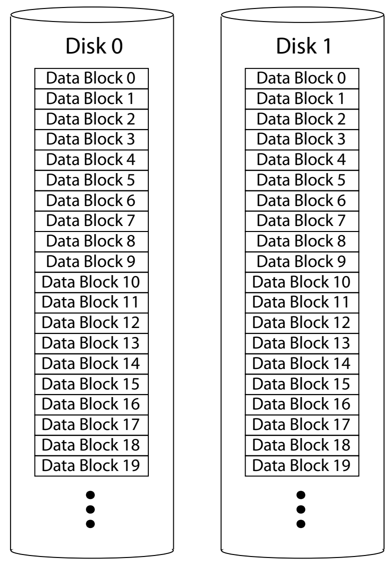
RAID

- 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

46

RAID 1: Mirroring

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



47

Parity

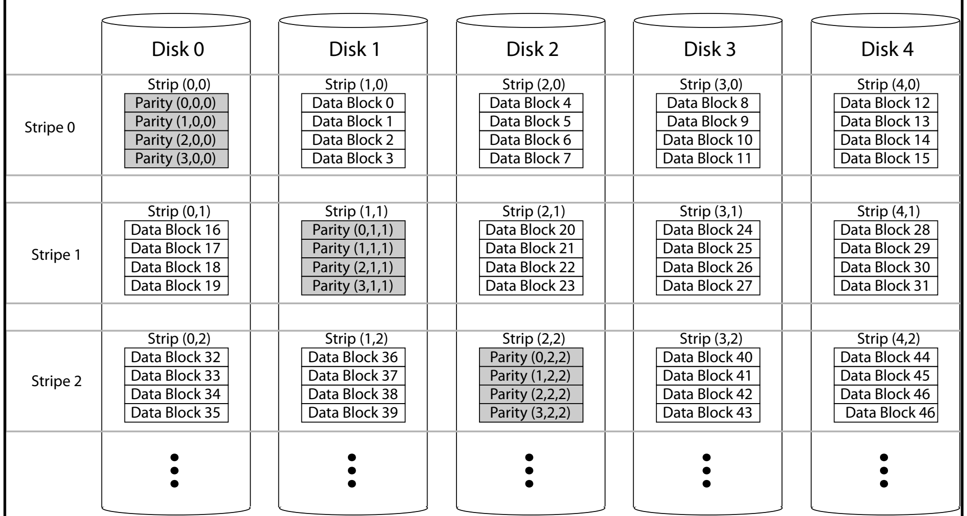
- Parity block: Block1 xor block2 xor block3 …

```
10001101        block1
01101100        block2
11000110        block3
--------------
00100111        parity block
```

- Can reconstruct any missing block from the others

48

RAID 5: Rotating parity



49

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

50

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?

51

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

52