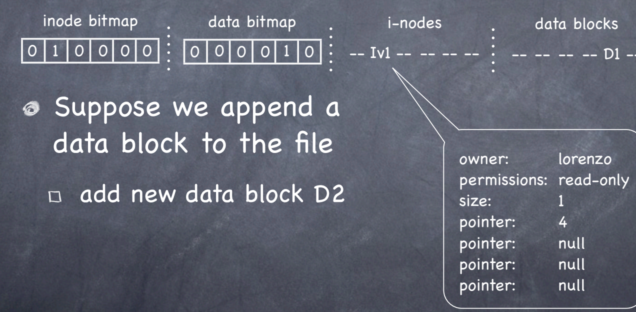
Caching and consistency

- File systems maintain many data structures
  - bitmap of free blocks
  - bitmap of inodes
  - directories
  - inodes
  - data blocks
- Data structures cached for performance
  - works great for read operations...
  - ...but what about writes?
    - modified cached data will be lost on a crash
- Solutions:
  - write-back caches: delay writes for higher performance at the cost of potential inconsistencies
  - write through caches: write synchronously but poor performance
    - do we get consistency at least?

Example: a tiny ext2

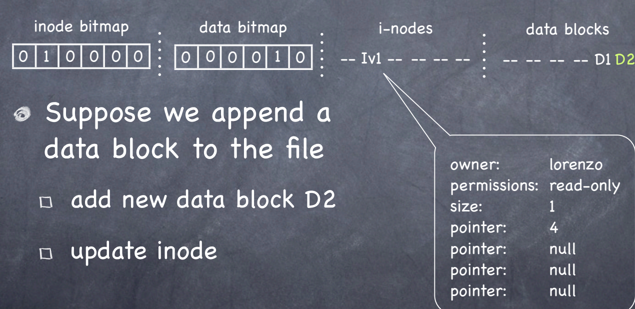
- 6 blocks, 6 inodes



- Suppose we append a data block to the file
  - add new data block D2

Example: a tiny ext2

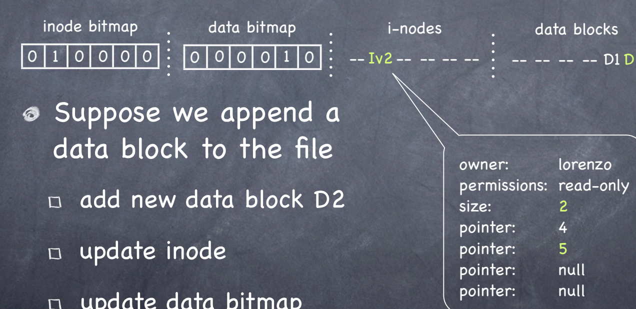
- 6 blocks, 6 inodes



- Suppose we append a data block to the file
  - add new data block D2
  - update inode

Example: a tiny ext2

- 6 blocks, 6 inodes

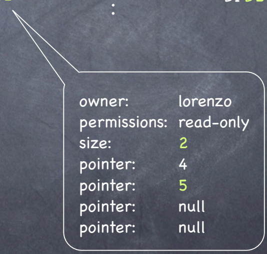


- Suppose we append a data block to the file
  - add new data block D2
  - update inode
  - update data bitmap

Example: a tiny ext2

- 6 blocks, 6 inodes

- Suppose we append a data block to the file
  - add new data block D2
  - update inode
  - update data bitmap



What if a crash or power outage occurs between writes?

What if only a single write succeeds?

- Just the data block (D2) is written to disk
  - data is written, but no way to get to it – in fact, D2 still appears as a free block
  - as if write never occurred
- Just the updated inode (Iv2) is written to disk
  - if we follow the pointer, we read garbage
  - file system inconsistency: data bitmap says block is free, while inode says it is used. Must be fixed
- Just the updated bitmap is written to disk
  - file system inconsistency: data bitmap says data block is used, but no inode points to it.
  - No idea which file the data block was to belong to!

What if two writes succeed?

- Inode and data bitmap updates succeed
  - file system is consistent
  - but reading new block returns garbage
- Inode and data block updates succeed
  - file system inconsistency. Must be fixed
- Data bitmap and data block succeed
  - file system inconsistency
  - no idea which file data block was to belong to!

The Consistent Update problem

- Several file systems operations update multiple data structures
  - Move a file between directories
    - delete file from old directory
    - add file to new directory
  - Create new file
    - update inode bitmap and data bitmap
    - write new inode
    - add new file to directory file
- Even with write through we have a problem!

Ad hoc solutions: metadata consistency

- Synchronous write through for metadata
- Updates performed in a specific order
  - File create
    - write data block
    - update inode
    - update inode bitmap
    - update data bitmap
    - update directory
    - if directory grew: 1) update data bitmap; 2) update directory inode
- On file crash
  - fsck
    - scans entire disk for inconsistencies
    - prior to update of inode bitmap: writes disappear
    - data block referenced in inode, but not in data bitmap: update bitmap
    - file created but not in any directory: delete file
- Issues
  - need to get ad-hoc reasoning exactly right
  - synchronous writes lead to poor performance
  - recovery is sloooow: must scan entire disk

Ad hoc solutions: user data consistency

- Asynchronous write back
  - forced after a fixed interval (e.g. 30 sec)
  - can lose up to 30 sec of work
- Rely on metadata consistency
  - updating a file in vi
    - delete old file
    - write new file

Ad hoc solutions: user data consistency

- Asynchronous write back
  - forced after a fixed interval (e.g. 30 sec)
  - can lose up to 30 sec of work
- Rely on metadata consistency
  - updating a file in vi
    - write new version to temp
    - move old version to other temp
    - move new version to real file
    - unlink old version
      - if crash, look in temp area and send "there may be a problem" email to user

Ad hoc solutions: implementation tricks

- Block I/O Barriers
  - allow a block device user to enforce ordering among I/O issued to that block device
  - client need not block waiting for write to complete
  - instead, OS builds a dependency graph
    - no write goes to disk unless all writes it depends on have

A principled apporach: Transactions

- Group together actions so that they are
  - Atomic: either all happen or none
  - Consistent: maintain invariants
  - Isolated: serializable (schedule in which transactions occur is equivalent to transactions executing sequentially
  - Durable: once completed, effects are persistent
- Critical sections are ACI, but not Durable
- Transaction can have two outcomes:
  - Commit: transaction becomes durable
  - Abort: transaction never happened
    - may require appropriate rollback

Journaling (write ahead logging)

- Turns multiple disk updates into a single disk write
  - "write ahead" a short note to a "log", specifying changes about to be made to the FS data structures
  - if a crash occurs while updating the FS data structure, consult log to determine what to do
    - no need to scan entire disk!

Data Jounaling: an example

- We start with

inode bitmap	data bitmap	i-nodes	data blocks
0 1 0 0 0 0	0 0 0 0 1 0	-- Iv1 -- -- -- --	-- -- -- -- D1 --

- We want to add a new block to the file
- Three easy steps
  - Write to the log 5 blocks: TxBegin | Iv2 | B2 | D2 | TxEnd
    - write each record to a block, so it is atomic
  - Write the blocks for Iv2, B2, D2 to the FS proper
  - Mark the transaction free in the journal
- What happens if we crash before the log is updated?
  - no commit, nothing to disk - ignore changes!
- What happens if we crash after the log is updated?
  - replay changes in log back to disk

Journaling and Write Order

- Issuing the 5 writes to the log TxBegin | Iv2 | B2 | D2 | TxEnd sequentially is slow
- Issue at once, and transform in a single sequential write
- Problem: disk can schedule writes out of order
  - first write TxBegin, Iv2, B2, TxEnd
  - Disk loses power → then write D2
- Log contains: TxBegin | Iv2 | B2 | ?? | TxEnd
  - syntactically, transaction log looks fine, even with nonsense in place of D2!
- Set a Barrier before TxEnd
  - TxEnd must block until data on disk (or "Rethink the sync"!)

What about performance?

- All data is written twice... surely it is horrible?
- 100 1KB random writes vs. log + write-back
  - Direct write: 100 x T_{rw} ≈ 100 x 10ms ≈ 1s
  - Pessimistic log
    - 100 x T_{sw} + 100 x T_{rw} ≈ 100/(50x10^3) + 1s = 2ms + 1s
  - Realistic (write-back performed in the background)
    - more opportunities for disk scheduling
    - 100 random writes may take less time than in direct write case

COW file systems (copy-on-write)

- Data and metadata not updated in place, but written to new location
  - transforms random writes to sequential writes
- Several motivations
  - small writes are expensive
  - small writes are expensive on RAID
    - expensive to update a single block (4 disk I/O) but efficient for entire stripes
  - caches filter reads
  - widespread adoption of flash storage
    - wear leveling, which spreads writes across all cells, important to maximize flash life
    - COW techniques used to virtualize block addresses and redirect writes to cleared erasure blocks
  - large capacities enable versioning

The early 90s

- Growing memory sizes
  - file systems can afford large block caches
  - most reads can be satisfied from block cache
  - performance dominated by write performance
- Growing gap in random vs sequential I/O performance
  - transfer bandwidth increases 50%-100% per year
  - seek and rotational delay decrease by 5%-10% per year
  - using disks sequentially is a big win
- Existing file system perform poorly on many workloads
  - 6 writes to create a new file of 1 block
    - new inode | inode bitmap | directory data block that includes file | directory inode (if necessary) | new data block storing content of new file | data bitmap
  - lots of short seeks

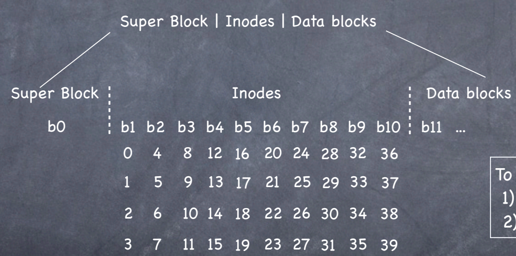
Log structured file systems

- Use disk as a log
  - buffer all updates (including metadata!) into a segment
  - when segment is full, write to disk in a long sequential transfer to unused part of disk
- Virtually no seeks
  - much improved disk throughput
- But how does it work?
  - suppose we want to add a new block to a 0-sized file
  - LFS paces both data block and inode in its in-memory segment

Fine.
But how do we find the inode?

Finding inodes

- in UFS, just index into inode array



512 bytes/block
128 bytes/inode

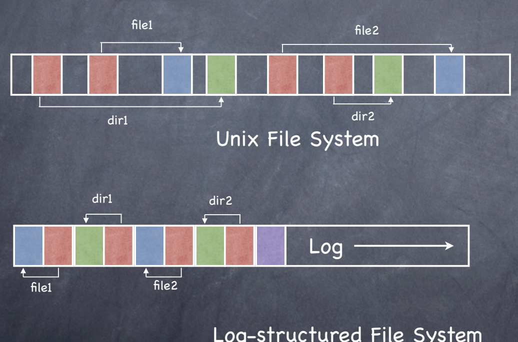
To find address inode 11:
1) 11/3 = 3
2) 11 mod 4 = 3

- Same in FFS (but Inodes are at divided (at known locations) between block groups

Finding inodes in LFS

- inode map: a table indicating where each inode is on disk
  - inode map blocks written as part of the segment
  - ... so need not seek to write to imap
- but how do we find the inode map?
  - table in a fixed checkpoint region
    - updated periodically (every 30 seconds)
- The disk then looks like

LFS vs UFS



Blocks written to create two 1-block files: dir1/file1 and dir2/file2 in UFS and LFS

Reading from disk in LFS

- Suppose nothing in memory...
  - read checkpoint region
  - from it, read and cache entire inode map
  - from now one, everything as usual
    - read inode
    - use inode's pointers to get to data blocks
- When the imap is cached, LFS reads involve virtually the same work as reads in traditional file systems

modulo an imap lookup

Garbage collection

- As old blocks of files are replaced by new, segment in log become fragmented
- Cleaning used to produce contiguous space on which to write
  - compact M fragmented segments into N new segments, newly written to the log
  - free old M segments
- Cleaning mechanism:
  - How can LFS tell which segment blocks are live and which dead?
- Cleaning policy
  - How often should the cleaner run?
  - How should the cleaner pick segments?

Segment summary block

- For each data block, stores
  - the file it belongs (inode#)
  - the offset (block#) within file
- During cleaning
  - allows to determine whether data block D is live
    - use inode# to find in imap where inode is currently on disk
    - read inode (if not already in memory)
    - check whether pointer for block block# refers to D's address
  - allows to update file's inode with correct pointer if D is live and compacted to new segment

Which segments to clean, and when?

- When?
  - periodically
  - when you have nothing better to do
  - when disk is full
- Which segments?
  - utilization: how much it is gained by cleaning
    - segment usage table tracks how much live data in segment
  - age: how likely is the segment to change soon
    - better to wait on cleaning a hot block

Crash recovery

- The journal is the file system!
- On recovery
  - read checkpoint region
    - may be out of date (written periodically)
  - roll forward
    - start from where checkpoint says log ends
    - read through next segments to find valid updates not recorded in checkpoint
      - when a new inode is found, update imap
      - when a data block is found that belongs to no inode, ignore
  - consistency between directory entries and inodes is tricky
    - one of inode or directory block could have made it to disk without the other
      - create in log a special record for each directory change (journaling!)
      - use Barrier to ensure that record is written in log before inode or directory block

Error detection and correction

- A layered approach
  - At the hardware level, checksums and device-level checks
    - error correcting codes
  - At the system level, redundancy, as in RAID
  - End-to-end checks
    - Safestore, Depot
      - no need for trust

Storage device failures and mitigation - I

- sector/page failure
  - data lost, rest of device operates correctly
    - can be permanent (e.g. due to scratches) or transient (e.g due to "high fly writes")
    - non recoverable read error: one bad sector/page per 10^{14} to 10^{18} bits read
  - mitigations
    - data encoded with additional redundancy (error correcting codes)
    - remapping (device includes spare sectors/pages)
  - pitfalls
    - non-recoverable error rates are negligible
      - not on a 2TB disk!
    - non-recoverable error rates are constant
      - the vary depending on load, age, or workload
    - failures are independent
      - errors often correlated in time or space
    - error rates are uniform
      - different causes can contribute differently to nonrecoverable read errors

Example: unrecoverable read errors

- Your 500GB laptop disk just crashed BUT you have just made a full backup on a 500GB USB
  - non recoverable read error rate: 1 sector/10^{14} bits read
- What is the probability of reading successfully the entire USB drive during restore?

Expected number of failures while reading the data:

$$500 \text{ GB} \times \frac{8 \times 10^9 \text{ bits}}{\text{GB}} \times \frac{1 \text{ error}}{10^{14} \text{ bits}} = 0.04$$

Probability of at least one failure is a little lower (there is a small chance of multiple failures)

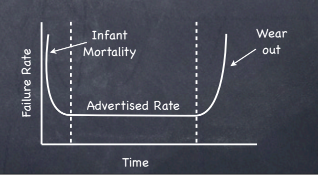
Assume each bit has a 10^{-14} chance of being wrong and that failures are independent

Probability to read all bits successfully:

$$(1 - 10^{-14})^{(500 \times 8 \times 10^9)} = 0.9608$$

Storage device failures and mitigations - II

- Device failures
  - device stops to be able to serve reads and writes to all sectors/pages (e.g. due to capacitor failure, damaged disk head, wear-out)
  - annual failure rate
    - fraction of disks expected to fail/year
      - 2011: 0.5% to 0.9%
  - mean time to failure (MTTF)
    - inverse of annual failure rate
      - 2011: 10^6 hours to 1.7×10^6 hours
  - pitfalls
    - MTTF measures a device's useful life (MTTF applies to device's intended service life)
    - advertised failure rates are trustworthy
    - failures are independent
    - failure rates are constant
    - devices behave identically
    - devices fail with no warning



Example: disk failures in a large system

- File server with 100 disks
- MTTF for each disk: 1.5×10^6 hours
- What is the expected time before one disk fails?

Assuming independent failures and constant failure rates:

MTTF for some disk = MTTF for single disk / 100 = 1.5×10^4 hours

Probability that some disk will fail in a year:

$$(365 \times 24) \text{ hours} \times \frac{1}{1.5 \times 10^4} \frac{\text{errors}}{\text{hours}} = 58.5\%$$

Pitfalls:

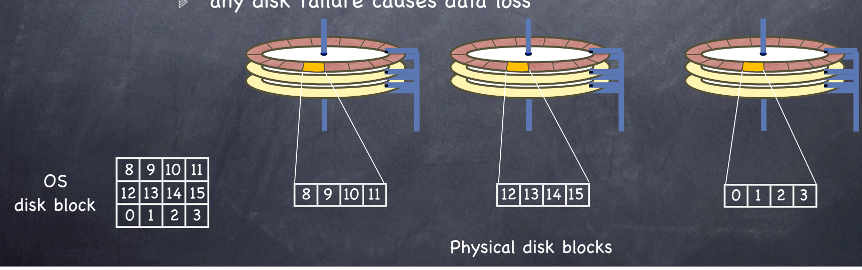
actual failure rate may be higher than advertised
failure rate may not be constant

RAID

- Redundant Array of Inexpensive Disks
  - disks are cheap, so put many (10s to 100s) of them in one box to increase storage, performance, and availability
  - data plus some redundant information is striped across disks
  - performance and reliability depend on how precisely it is striped

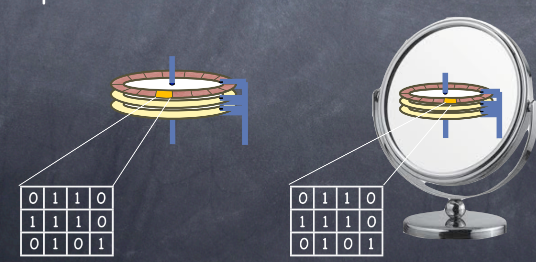
RAID-0: increasing throughput

- Disk striping (RAID-0)
  - blocks broken in sub-blocks stored on separate disks
    - similar to memory interleaving
  - higher disk bandwidth through larger effective block size
  - poor reliability
    - any disk failure causes data loss



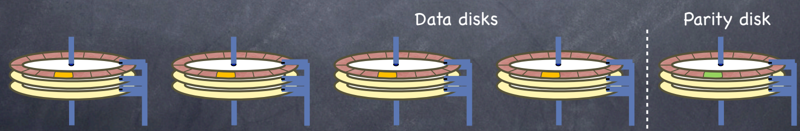
RAID-1 mirrored disks

- Data written in two places
  - on failure, use surviving disk
- On read, choose fastest to read
- Expensive



RAID-3

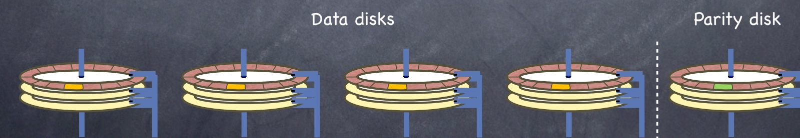
- Bit striped, with parity
  - given G disks,
    - parity = $data_0 \oplus data_1 \oplus \ldots \oplus data_{G-1}$
    - $data_0$ = parity $\oplus data_1 \oplus \ldots \oplus data_{G-1}$
- Reads access all data disks
- Writes accesses all data disks plus parity disk



Disk controller can identify faulty disk
- single parity disk can detect and correct errors

RAID-4

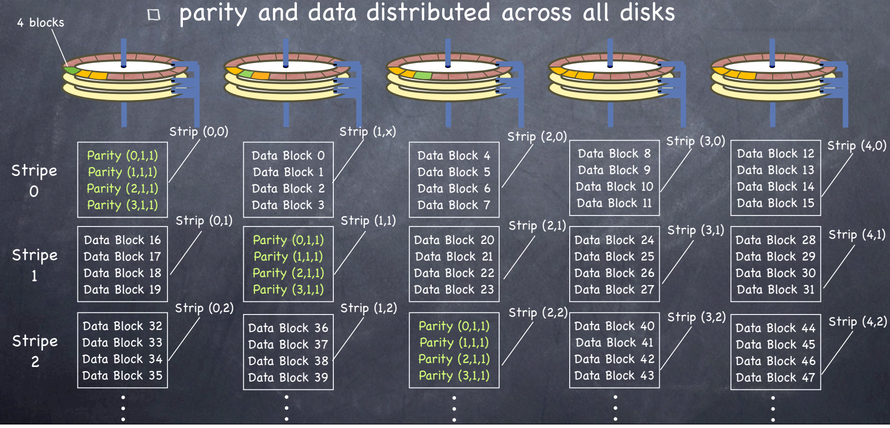
- Block striped, with parity
- Combines RAID-0 and RAID-3
  - reading a block accesses a single disk
  - writing always accesses parity disk
    - Heavy load on parity disk



Disk controller can identify faulty disk
- single parity disk can detect and correct errors

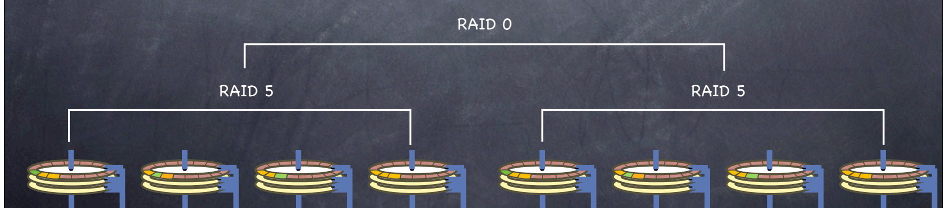
RAID-5

- Block Interleaved Distributed Parity
  - no single disk dedicated to parity
  - parity and data distributed across all disks



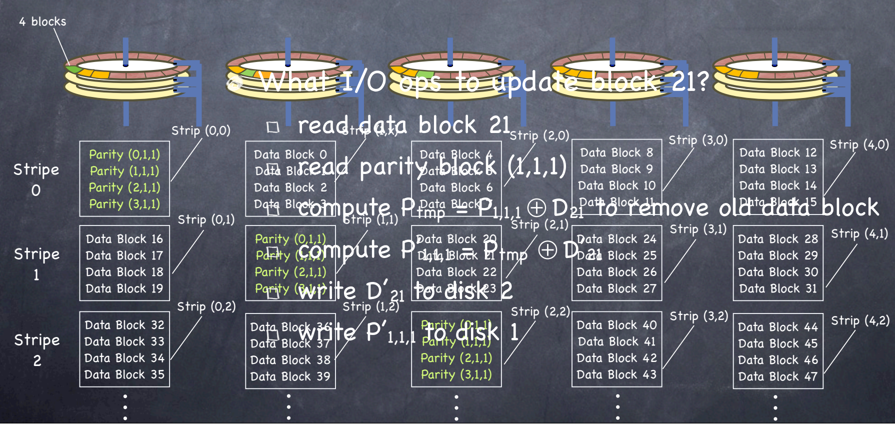
RAID 10 and RAID 50

- RAID 10
  - stripes (RAID 0) across reliable logical disks, implemented as mirrored disk paier (RAID 1)
- RAID 50
  - stripes (RAID 0) across groups of diskswth block intelraved distributed parity



Example: Updating a RAID with rotating parity

- What I/O ops to update block 21?
  - read data block 21
  - read parity block (1,1,1)
  - compute $P_{tmp} = P_{1,1,1} \oplus D_{21}$ to remove old data block
  - compute $P'_{1,1,1} = P_{tmp} \oplus D'_{21}$
  - write D'_{21} to disk 2
  - write $P'_{1,1,1}$ to disk 1



RAID Reliability: Double Disk Failure

Two full-disk failures

- N disks, 1 Pblock/G disks
- disk fail independently
- MTTDL, MTTF, MTTR
- MTTR << MTTF

Expected time to first failure: MTTF/N

Probability of 2nd fault before repair:

$$\frac{MTTR}{MTTF/(G-1)}$$

Number of "coin flips" to get 2nd fault:

$$\frac{MTTF/(G-1)}{MTTR}$$

$$MTTDL = \frac{MTTF^2}{N \times (G-1) \times MTTR}$$

MTTDL: mean time to data loss

- inverse of Failure Rate

Example

- 100 disks, G = 10 (9+1 for parity)
- disk fail independently
- MTTF = 10^6 hours
- MTTR = 10 hours

$$MTTDL = \frac{MTTF^2}{N \times (G-1) \times MTTR} = \frac{10^{12}}{10^2 \times 9 \times 10}$$

$$\approx 10^8 \text{ hours}$$

RAID Reliability: sector failures

One disk full failure + sector failure

$$MTTDL = \frac{MTTF}{N} \times \frac{1}{P_{fail\_recovery\_read}}$$

Failure of two sectors sharing a redundant sector:

Negligible risk

Example

- Latent sector errors: 1 every 1^{15} bits read
- Disk fail independently
- 100 1TB disks; G = 10
- MTTF = 10^6 hours

$$P_{success\_recovery\_read} = (1 - 10^{-15})^{(9 \times 8 \times 10^{12})} \approx$$

$$\approx (1 - 72 \times 10^{-3}) = 0.928$$

$$MTTDL = \frac{MTTF}{N} \times \frac{1}{P_{fail\_recovery\_read}} =$$

$$\frac{10^6}{10^2} \times \frac{1}{1 - 0.928} \approx 1.39 \times 10^5 \text{ hours}$$

Overall data loss rate

- Assuming independent failures and constant failure rate:

$$FailureRate_{indep+const} = FailureRate_{2Disks} + FailureRate_{disk+sector} =$$

$$= \frac{1}{MTTDL_{2Disks}} + \frac{1}{MTTDL_{disk+sector}} =$$

$$= \frac{N}{MTTF} \times \left(\frac{MTTR \times (G-1)}{MTTF} + P_{fail\_recovery\_read} \right)$$

Improving RAID Reliability

- Increase redundancy
  - RAID 6: Reed-Solomon to tolerate 2 failures per stripe

$$\text{FailureRate}_{dual+independent+const} = \frac{N}{MTTF} \times \frac{MTTR \times (G-1)}{MTTF} \times \left(\frac{MTTR \times (G-2)}{MTTF} + P_{fail\_recovery\_read} \right)$$

- Reduce non recoverable read error rates
  - Scrubbing
    - periodically read entire content of disk and reconstruct lost data
  - Use more reliable disks
    - entrerprise disks 100 smaller error rate than PCs
- Reduce MTTR
  - hot spares (but bottleneck is often writing reconstructed data
  - declustering
    - in HDFS, each block written to 3 randomly chosen disks