Week 10 - Recap

Pamela Delgado

May 8, 2019

Crashes: Atomicity

- Atomic means:
  - Either all data written is on disk (new version)
  - Or none is on disk (old version)
- Single disk sector write is atomic
- Multiple disk sector writes are not atomic

How to Implement Atomicity?

- Keep old and new copies (no overwrites)
- Switch atomically by sector write

Two Techniques to Implement Atomicity

- Shadow paging:
  - Use inode sector overwrite
- Intentions log:
  - Write data and inodes to log
  - Copy data in-place later

Rationale for LFS

- Large memories → large buffer caches
- Most reads served from cache
- Most disk traffic is write traffic
- How to optimize disk I/O?
  - By optimizing disk writes
- How to optimize disk writes?
  - By writing sequentially to disk

Key Idea in LFS

- “All” writes go to log, including
  - data
  - inode
- “All” = All except for checkpoints

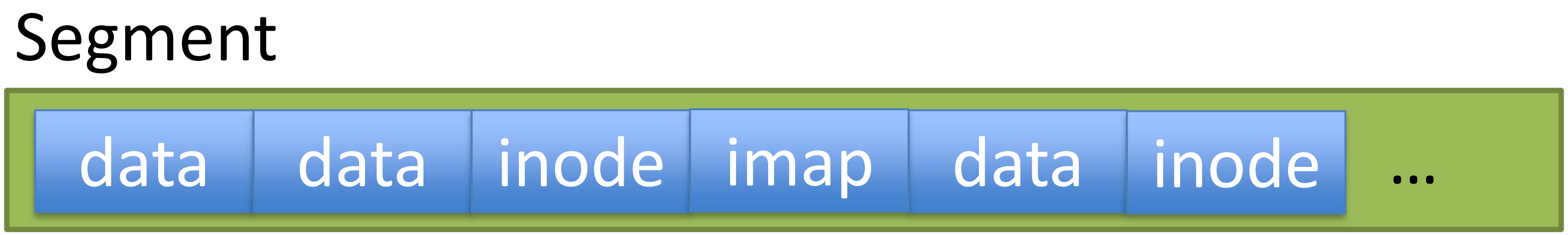
LFS Data Structures on Disk: Checkpoint and Log

Checkpoint region: at fixed location on disk

Log: uses the remainder of the disk

Segment: large (MBs) contiguous regions on disk

LFS Data Structures on Disk
In-Use Segments



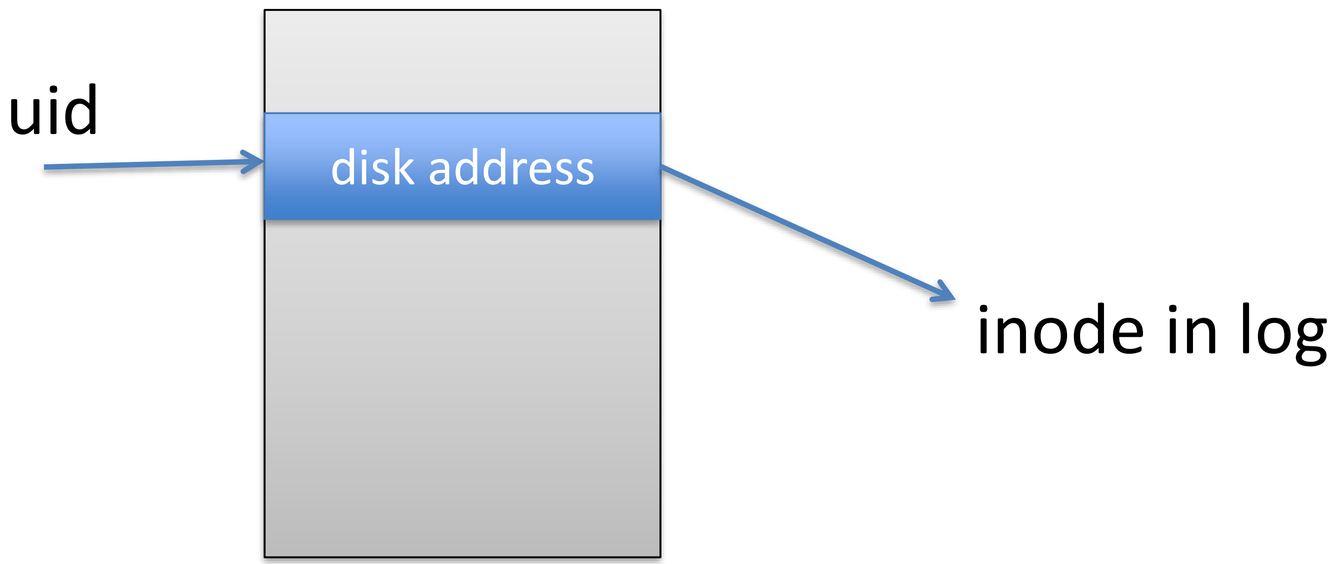
Data: modified user data sector (includes uid and block no)
inode: modified inode sector

LFS Data Structures in Memory: Cache, Segment Buffer

- Cache: regular write-behind buffer cache
- Segment buffer: segment being written

LFS Data Structures in Memory: inode Map

- Array
- Indexed by *uid*
- Point to last-written inode for *uid*



LFS Data Structures in Memory

- Also the usual
  - Active file table
  - Open file tables

Write() in LFS - 1

- Writes go into (write-behind) cache
  – Both inode and data sectors
- Writes go into (in-memory) segment buffer
  - Both inode and data sectors
- When segment buffer full
  – Write to free segment in disk log
- (Almost) no seeks on writes!

Write() in LFS - 2

- If inode is written to log
- imap[uid] = disk address of inode

Open()

- Get inode address from inode map
- Read inode from disk into Active File Table

Read()

- Get from cache
- If not in cache
  - Get from disk
  - Using disk address in inode
- As before

What if the Disk is Full?

- No sector is ever overwritten
  - Always written to end of log
- No sector is ever put on free list
- So disk will get full (quickly)
- Need to “clean” the disk

Disk Cleaning

- Reclaim “old” data
- “Old” here means
  - Logically overwritten
    - Later write to (uid, blockno)
  - But not physically overwritten
    - Older version of (uid, blockno) somewhere in the log

Disk Cleaning

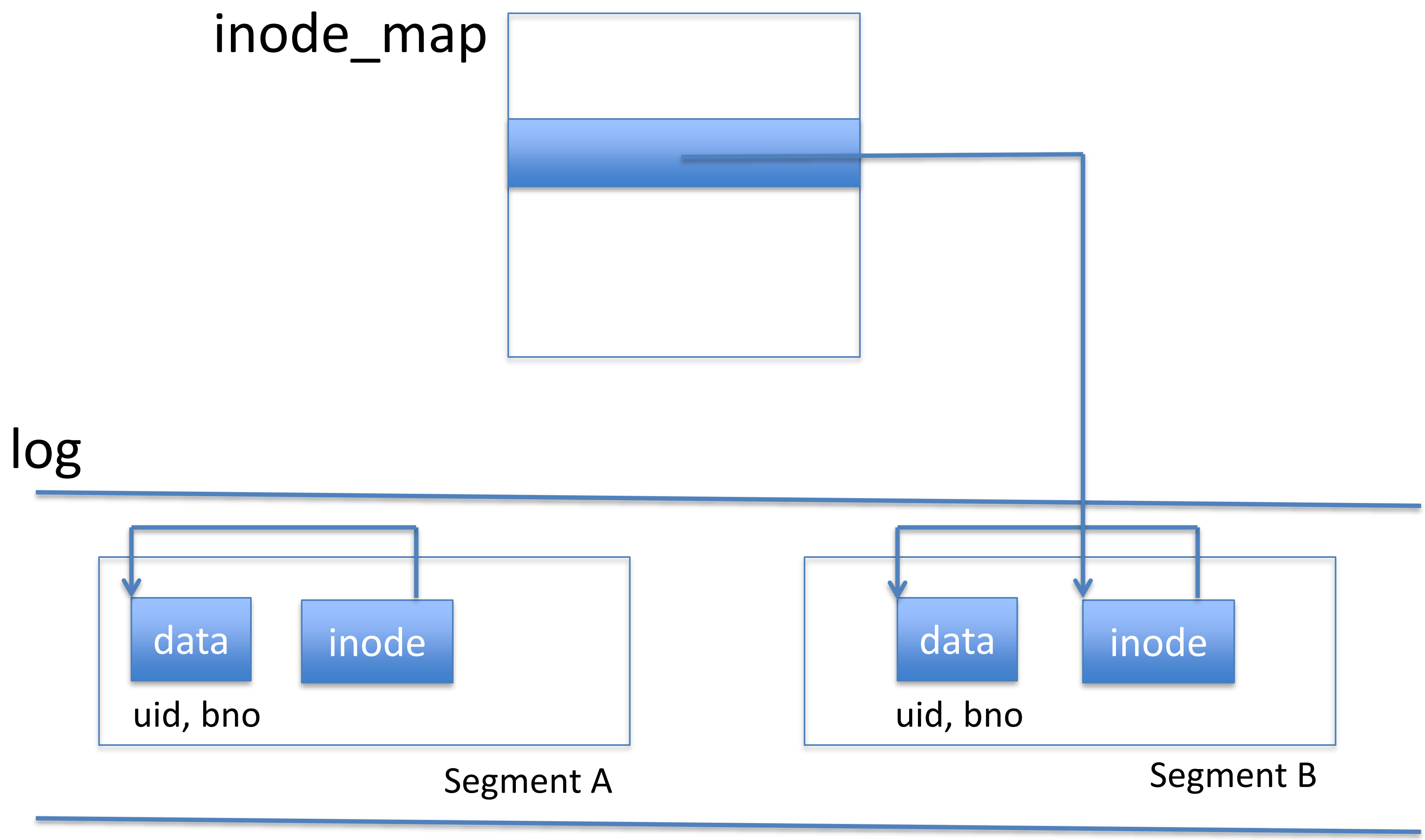
- Done one segment at a time

- Determine which blocks are new
- Write them into (in-memory) buffer
- If buffer is full, write new segment to log
- Cleaned segment is marked free

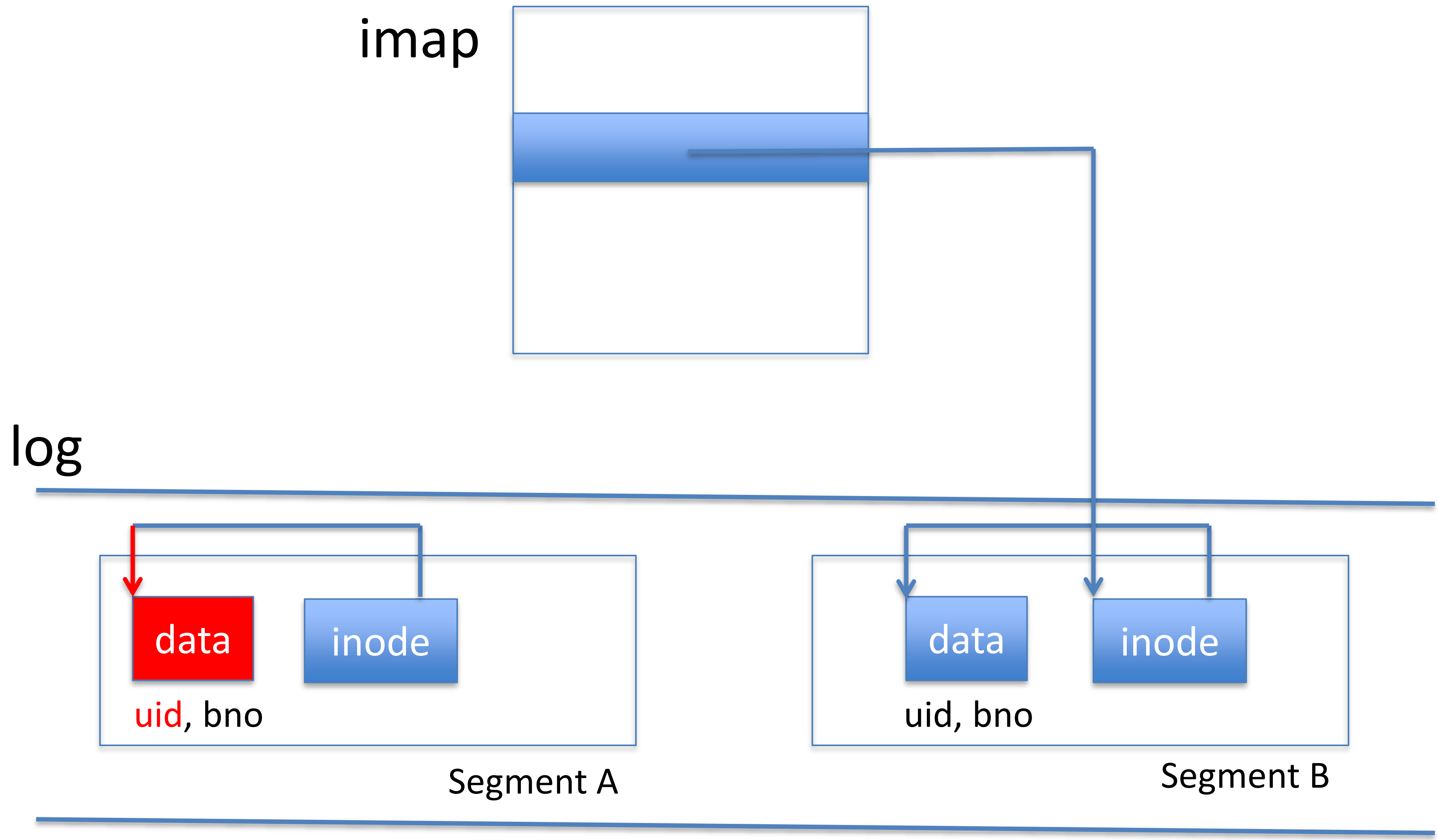
Determining a Block is Old

- For a data block
- Take its disk address
- Take its uid and block number
- Look in inode map and then in inode
- If inode has different disk address →old

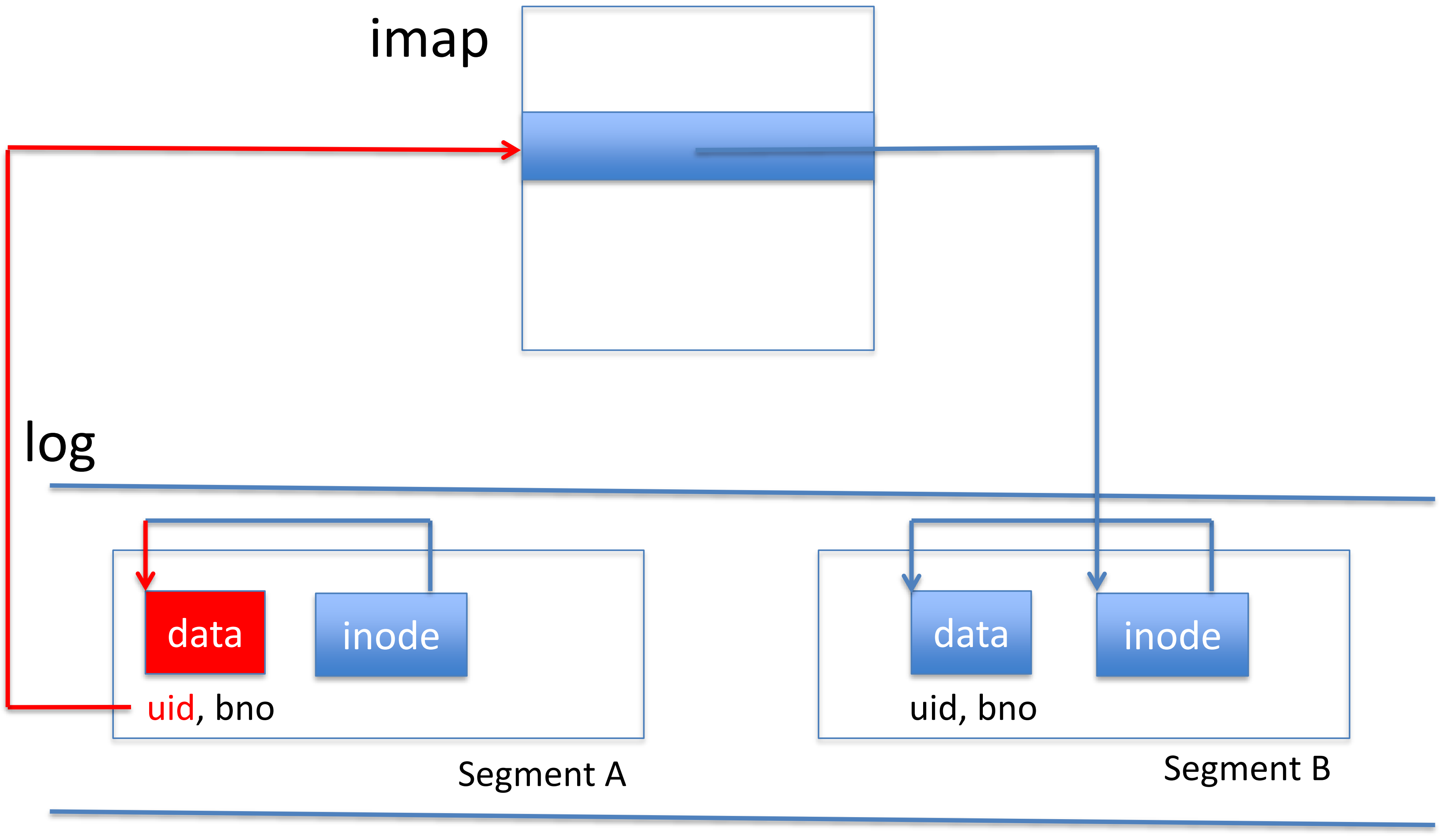
Determining if a Block is Old



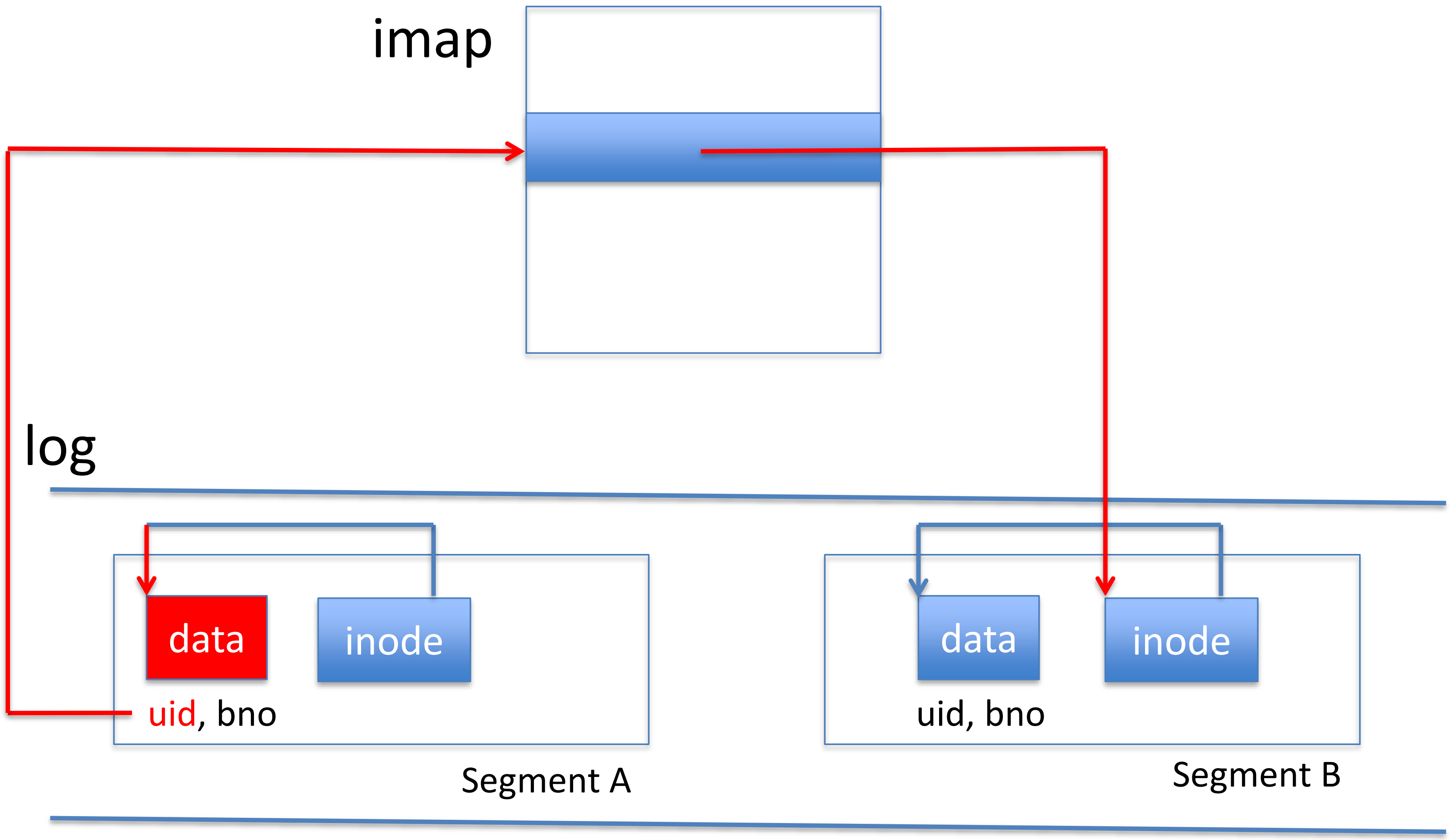
Cleaning Segment A



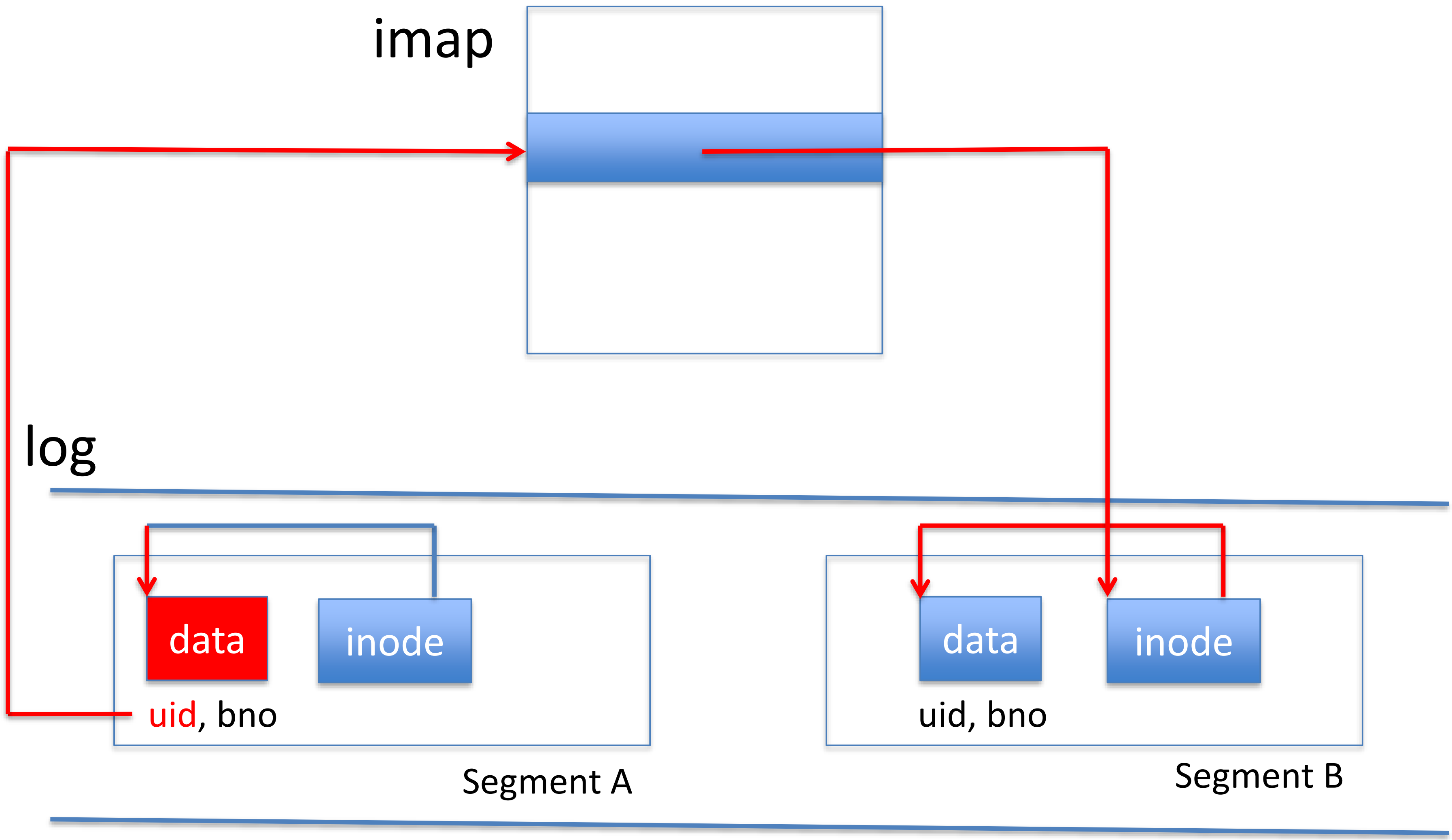
Cleaning Segment A



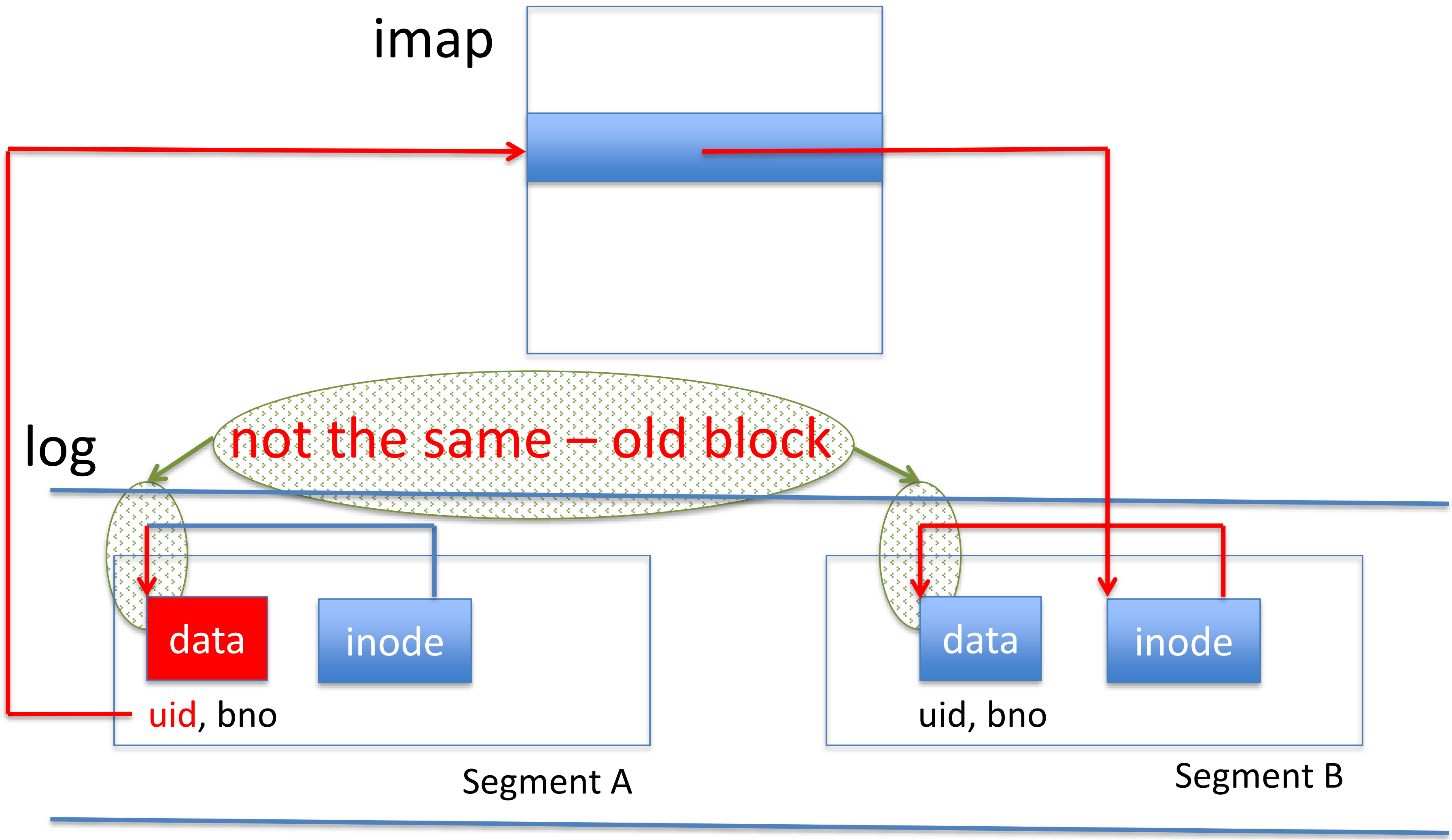
Cleaning Segment A



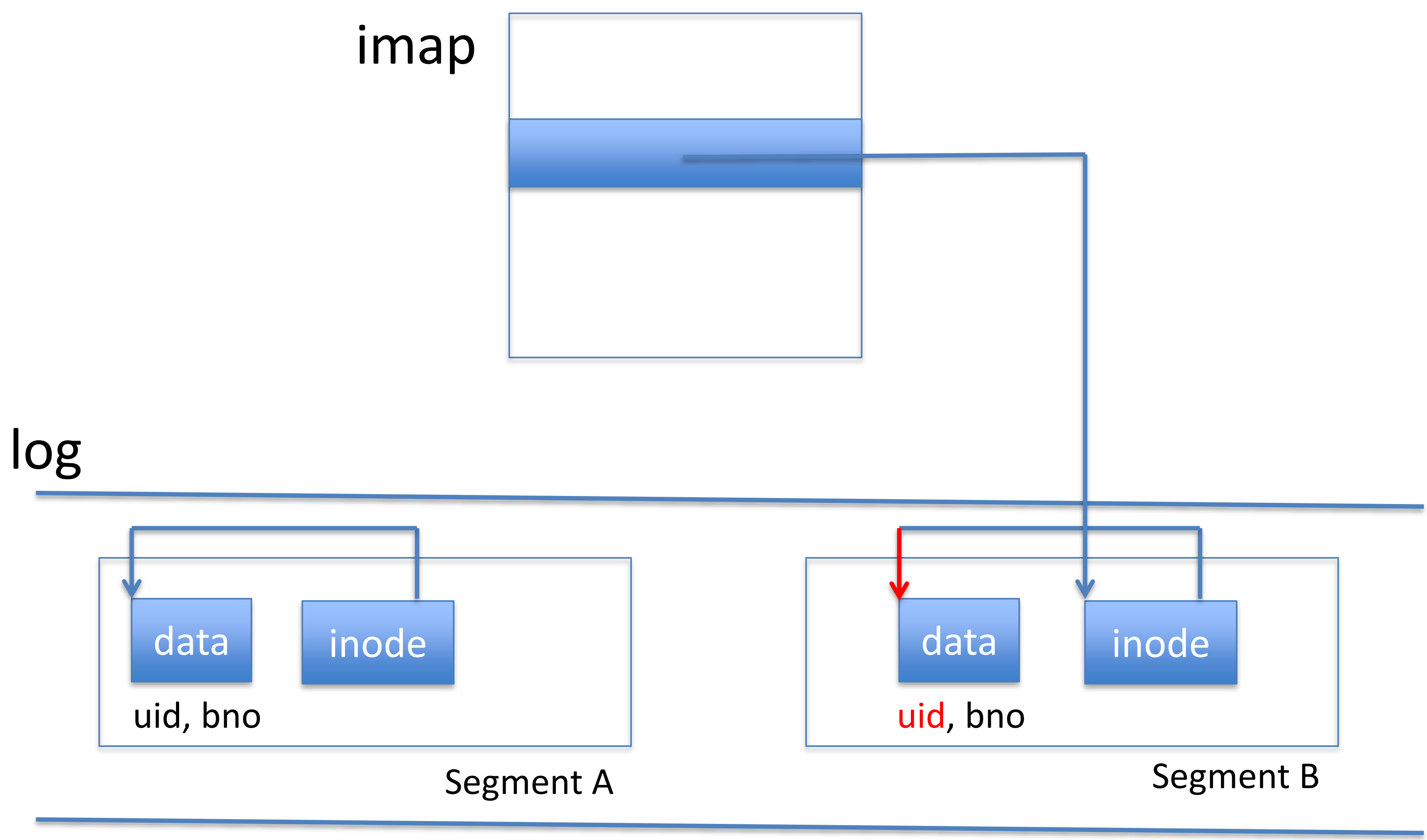
Cleaning Segment A



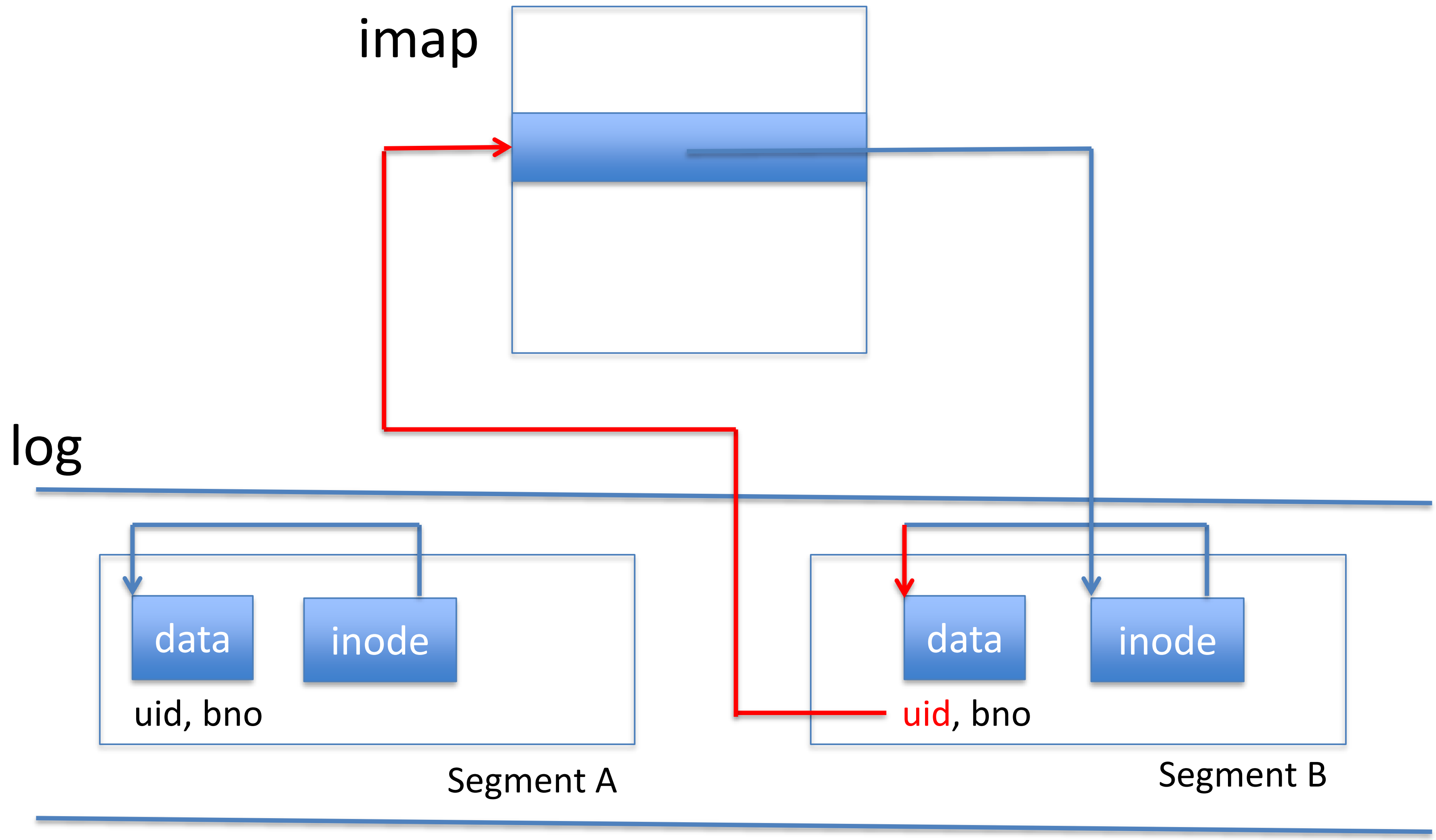
Cleaning Segment A



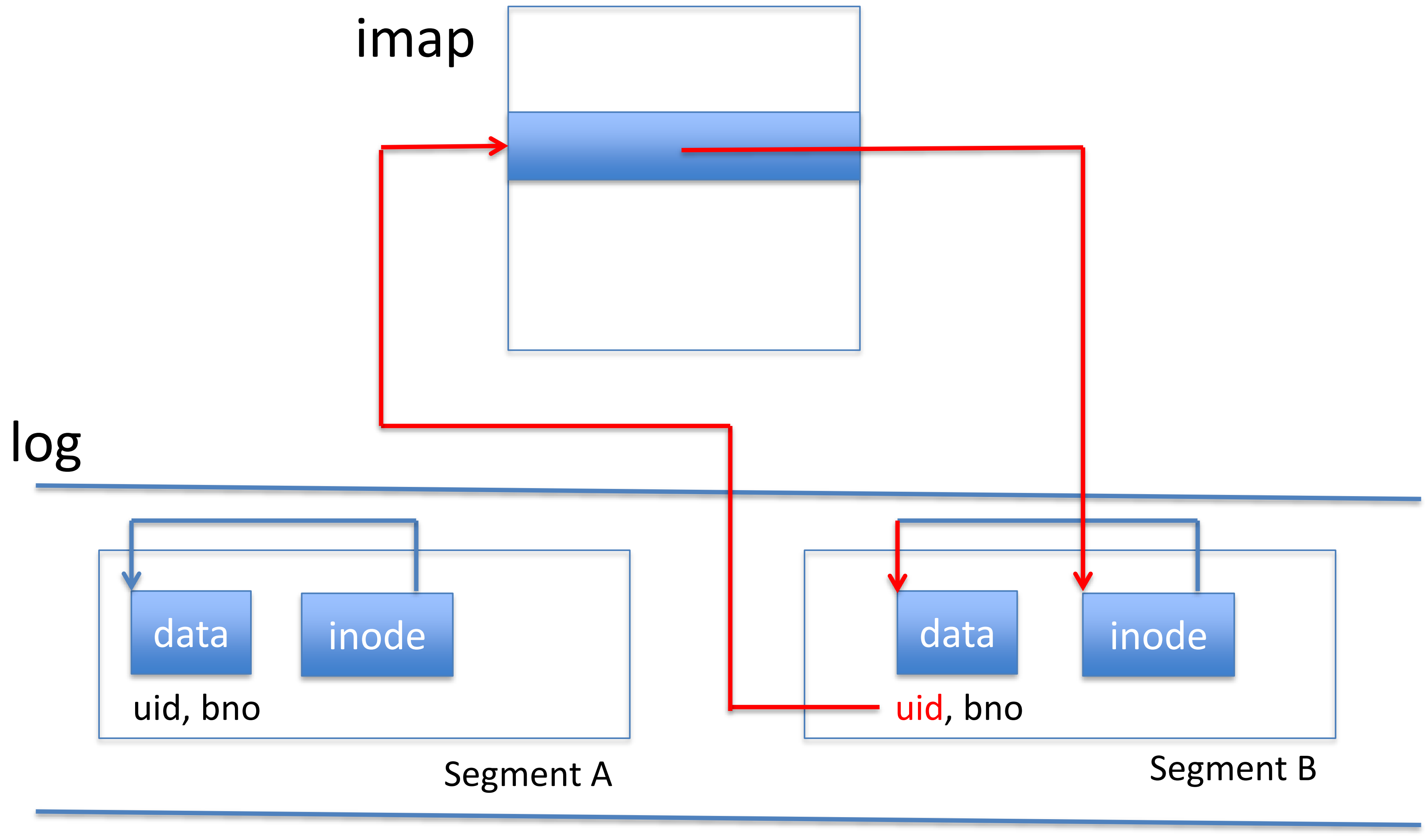
Cleaning Segment B



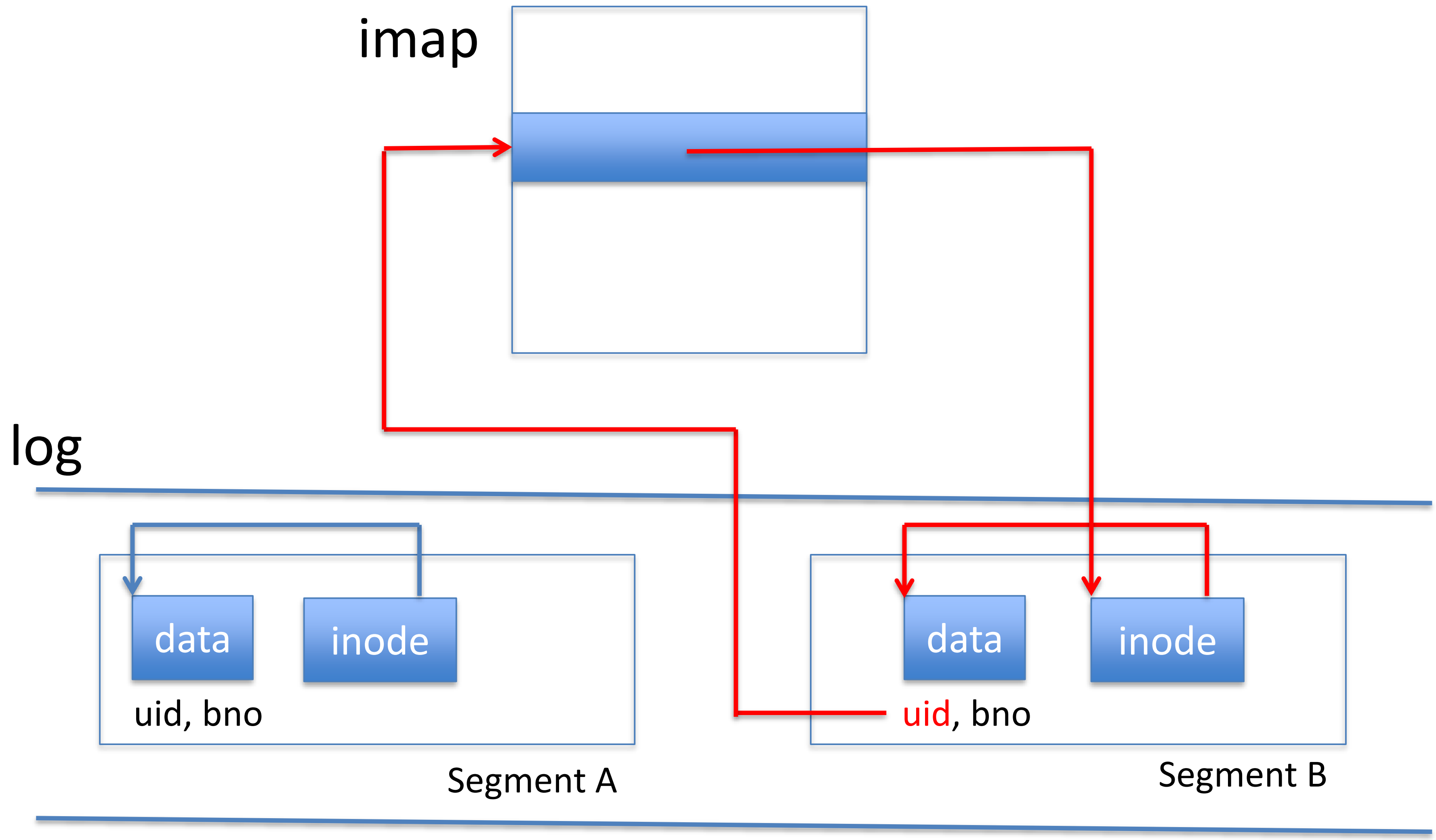
Cleaning Segment B



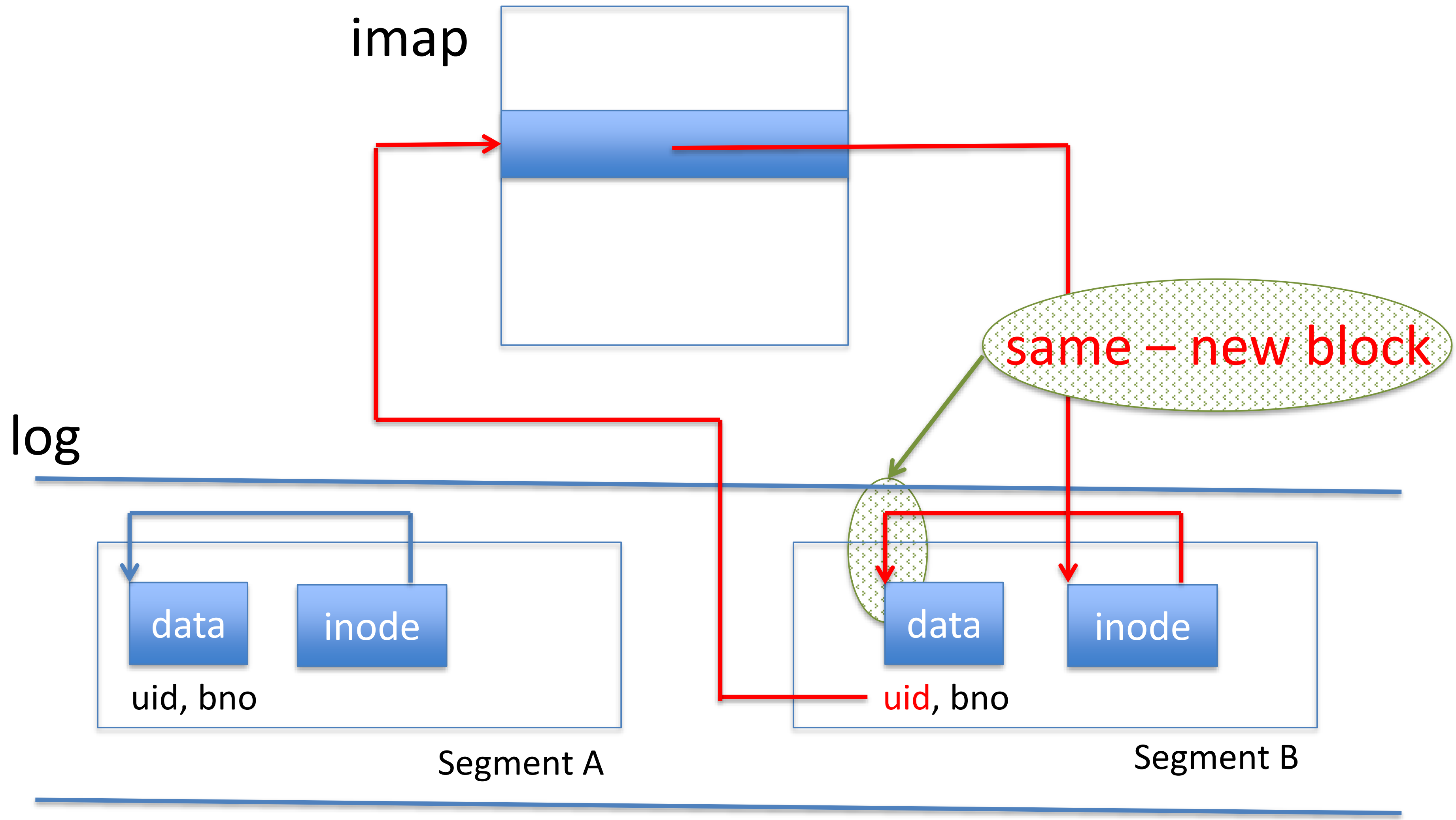
Cleaning Segment B



Cleaning Segment B



Cleaning Segment B



Summary: LFS

- Reads mostly from cache
- Writes to disk heavily optimized: few seeks
- Reads from disk: bit more expensive but few
- Cost of cleaning

Summary: LFS

- Is more complicated than what was presented
- Has not become mainstream
  - Cost of cleaning is considerable
  - Note similarity with garbage collection
  - Unpredictable performance dips

What Has Become Mainstream

- Journaling file system
- Uses log (called a “journal”) for reliability
- In Linux, the ext file systems (currently ext4)
- Not covered in this course

Week 11
Alternative Storage Media: RAID and SSD

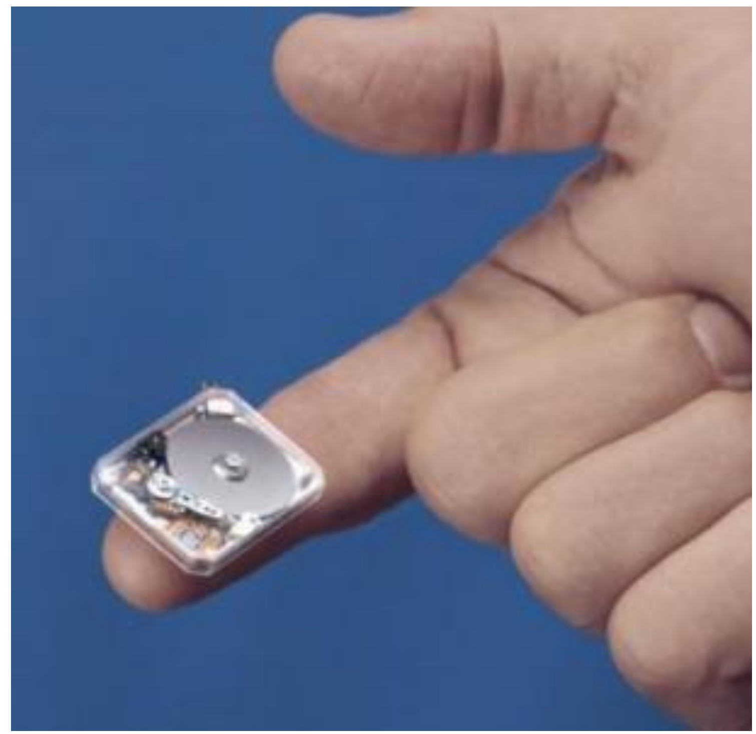
Pamela Delgado

May 15, 2019

Disk Evolution

1970s

Now



Disk Evolution

	1970s	Now
Physical size	Very large	Tiny
Cost	Very expensive	Cheap
Capacity	Small (MBs)	Large (TB)
Bandwidth	10 Mbps	100s Mbps
Latency (seek)	50-100msec	5-10msec

Disk Evolution

	1970s	Now
Size	Very large	Tiny
Cost	Very expensive	Cheap
Capacity	Small (MBs)	Large (TB)
Bandwidth	10 Mbps	100s Mbps
Latency (seek)	50-100msec	5-10msec

Disk Evolution

	1970s	Now
Size	Very large	Tiny
Cost	Very expensive	Cheap
Capacity	Small (MBs)	Large (TB)
Bandwidth	10 Mbps	100s Mbps
Latency (seek)	50-100msec	5-10msec

A factor of 10 when CPU have improved by a factor of thousands

Disk Issues

- Bandwidth:
  - Servers
  - "Big data" computations
- Response time:
  - Desktops and laptops
  - Transaction systems

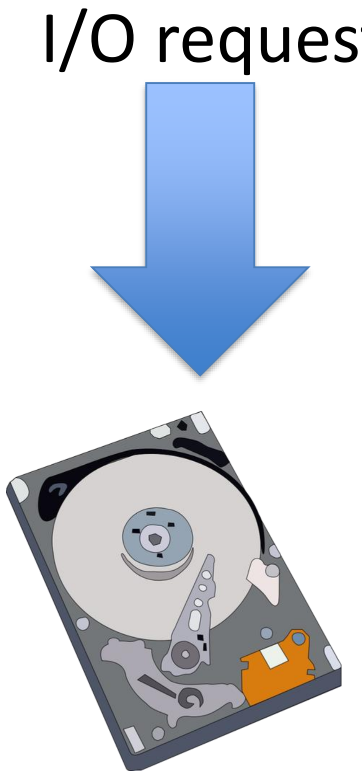
Storage Developments

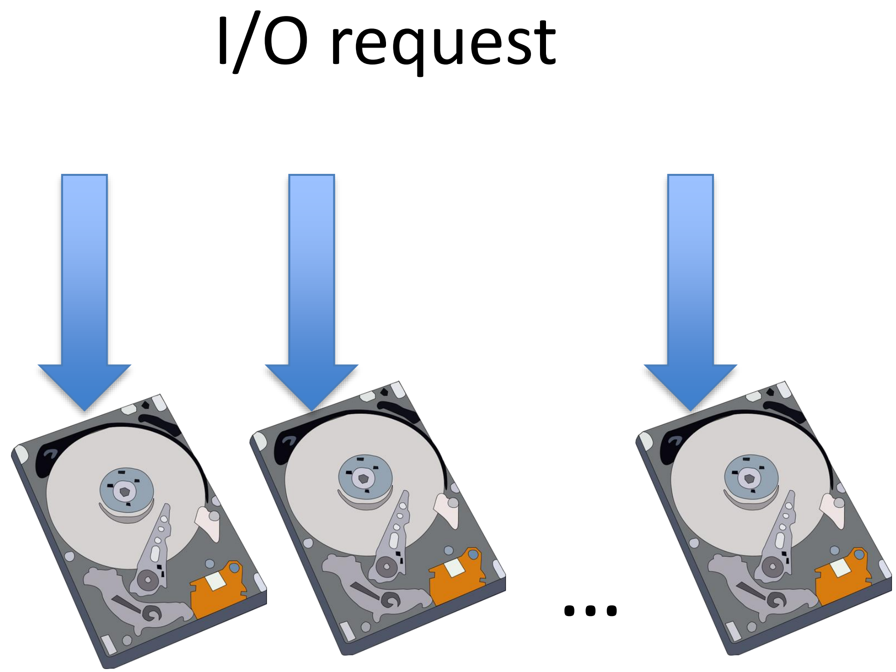
- RAID – bandwidth
- SSD – response time (and bandwidth)

- But both more costly than disk

RAID rationale

- Disks are cheaper and smaller





RAID

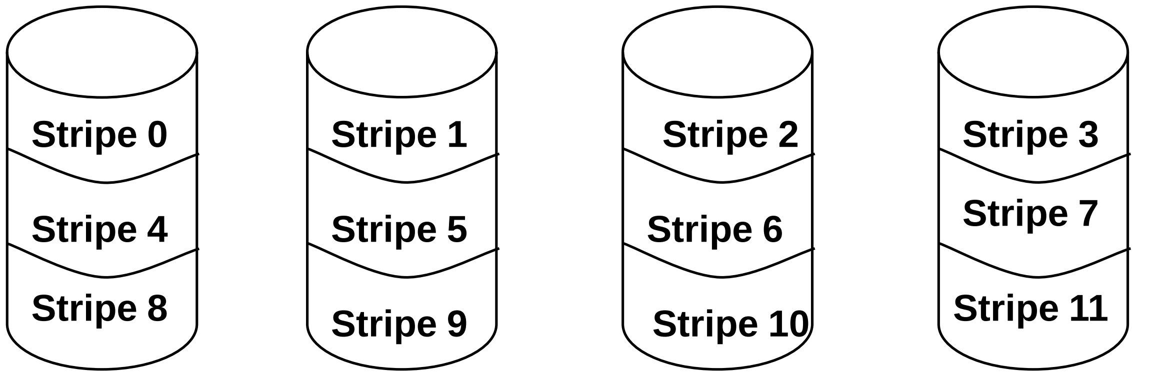
- Redundant Array of Independent Disks

- Essential idea
  - Optimize I/O bandwidth through parallel I/O
  - Parallel I/O = I/O to multiple disks at once

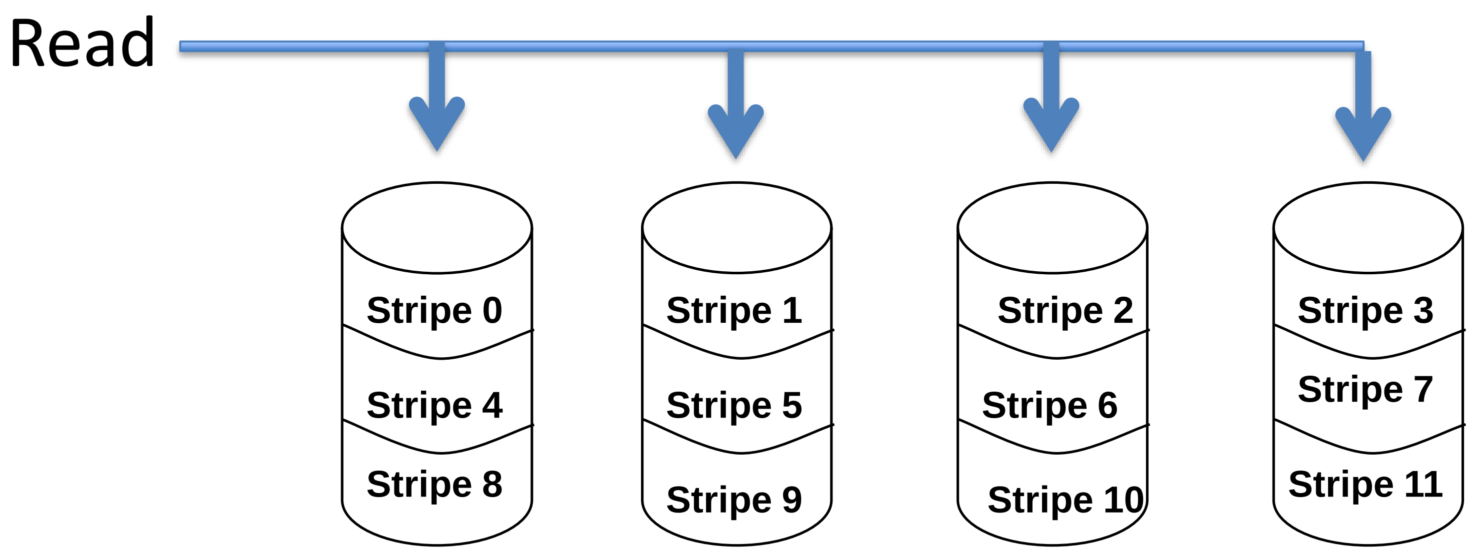
Striping

- Rather than put file on one disk
- Stripe it across a number of disks
  - File = Stripe0 Stripe1 Stripe2 …
  - Stripe0 on disk0
  - Stripe1 on disk1
  - …
- Read and write in parallel

Striping


Stripe 0
Stripe 4
Stripe 8
Stripe 1
Stripe 5
Stripe 9
Stripe 2
Stripe 6
Stripe 10
Stripe 3
Stripe 7
Stripe 11

Striping Read/Write



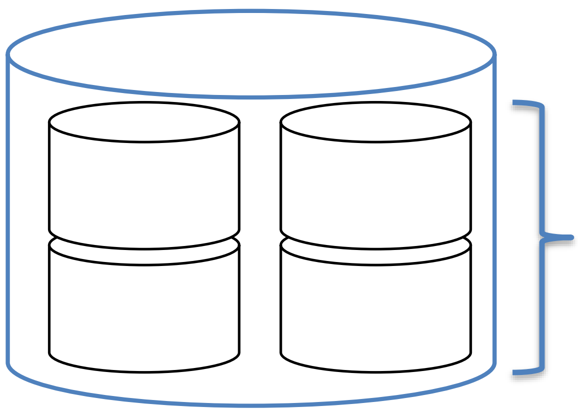
In the Best of Worlds

- Since disk is the bottleneck
- Bandwidth of RAID with n disks =

  n * bandwidth of individual disks
- At some point other factors
  - Bandwidth of I/O bus, controller, etc.
- But still, bandwidth of RAID >> bandwidth disk

RAID Format

- Disks now cheap and small
- Many can go into a RAID box
- To OS: RAID box looks like disk



Standalone unit

- Disks
- Memory
- processor(s)

- Also possible: RAID in software
  - Disks directly attached to buses

Problem with (Naïve) RAID?

Problem with (Naïve) RAID

- One disk fails → all data unavailable

Problem with (Naïve) RAID

- One disk fails → all data unavailable
- MTBF: Mean Time Between Failures
- MTBF (RAID) = MTBF (disk) / n
- MTBF (disk) ~ 50,000 hours ~ 5 years
- MTBF (RAID with 10 disk) ~ 0.5 year
- Not acceptable

Solution: Redundancy

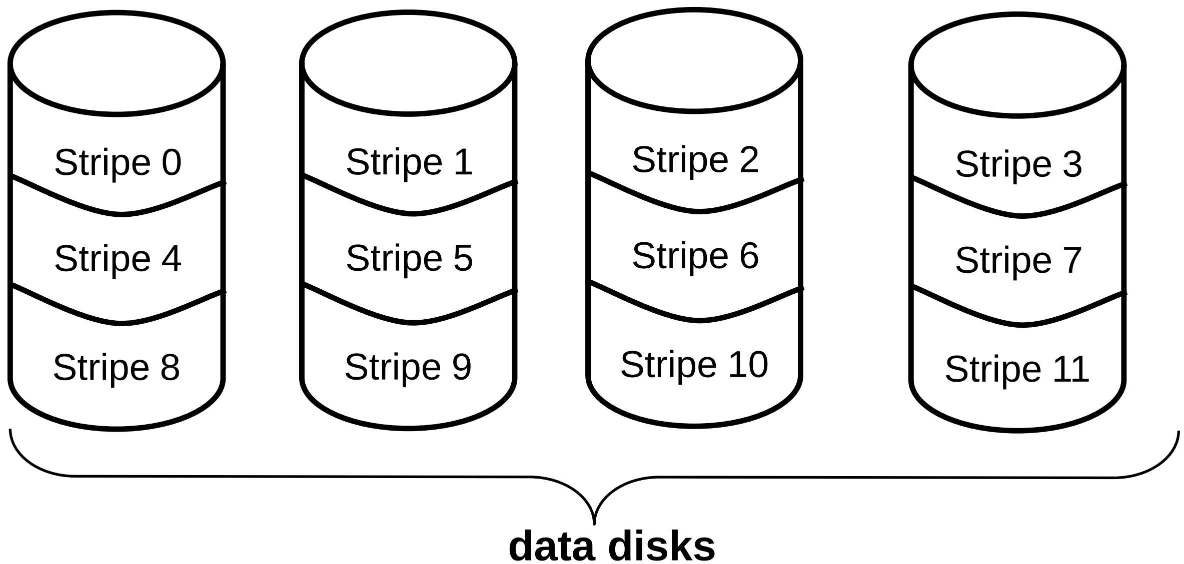
- Store redundant data on different disks
- One disk fails → data still available

RAID Levels

- Are redundancy levels
- What we have seen so far
  - RAID-0: No redundancy
- In reality:
  - RAID-1: Mirroring
  - RAID-2/3: No longer used, not covered in this class
  - RAID-4: Parity disk
  - RAID-5: Distributed parity

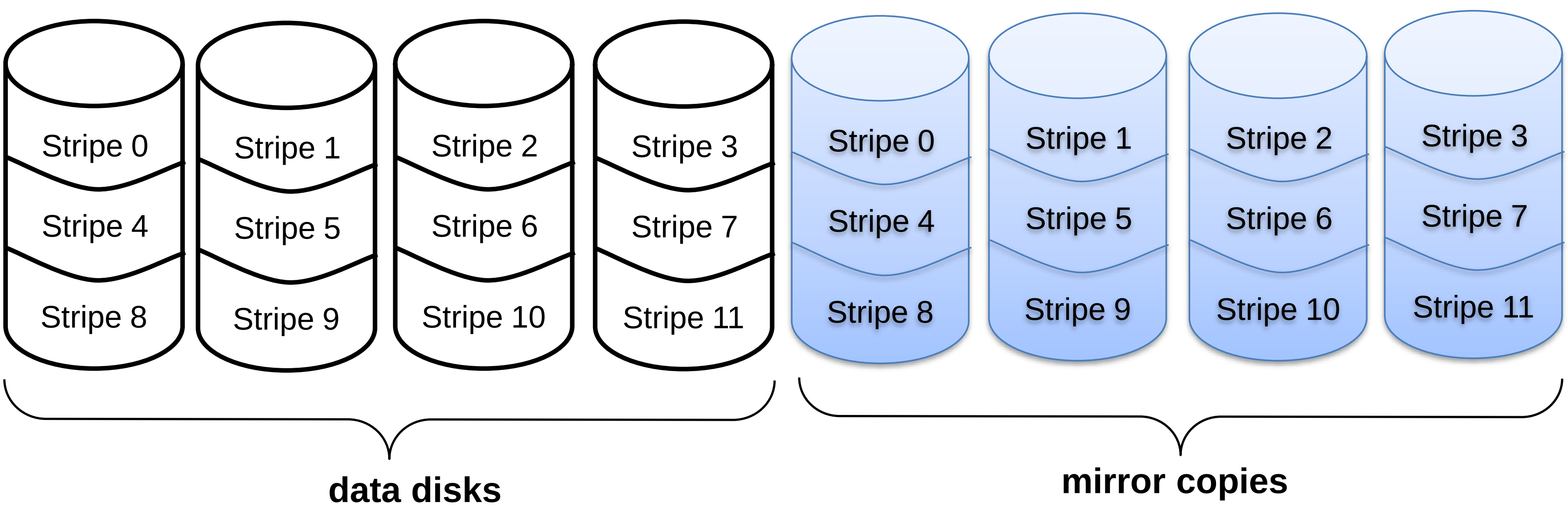
RAID-0

- Non-redundant disk array
- Best possible read and write bandwidth
- Failure results in data loss



RAID-1

- Mirrored disks
- Write: to data and to mirror disk
- Read: from either data or mirror
- After crash: from surviving disk



RAID-1

- For the same number of disks as RAID-0
- Storage capacity is half
- Survives disk failure of data or mirror

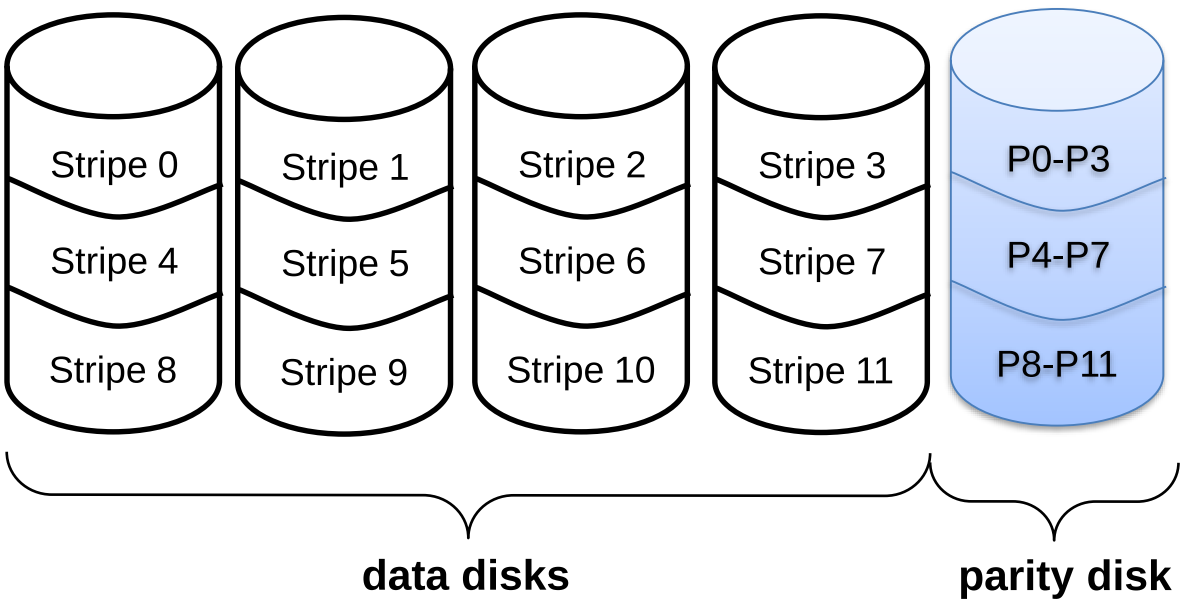
RAID-1

- For the same number of disks as RAID-0
- Storage capacity is half
- Survives disk failure of data or mirror

How to do better?

RAID-4

- N data disks + 1 parity disk



Parity

- A simple form of error detection and repair
- Not specific to RAID
- Also used in communications

Parity

- 4 bits: x_0, x_1, x_2, x_3
- Parity $p = x_0$ XOR x_1 XOR x_2 XOR x_3
- If you lose one bit, say x_2
- Reconstruct as $x_2 = x_0$ XOR x_1 XOR x_3 XOR p

XOR reminder
Even # of 1s→ 0
Uneven # of 1s → 1

Parity Example

- 4 bits: $x_0x_1x_2x_3$ = 0101
- Parity p = x_0 XOR x_1 XOR x_2 XOR x_3 = 0
- If you lose one bit, say x_2
- Reconstruct as x_2 = x_0 XOR x_1 XOR x_3 XOR p

0 XOR 1 XOR 1 XOR 0

0

XOR reminder

Even # of 1s→ 0

Uneven # of 1s → 1

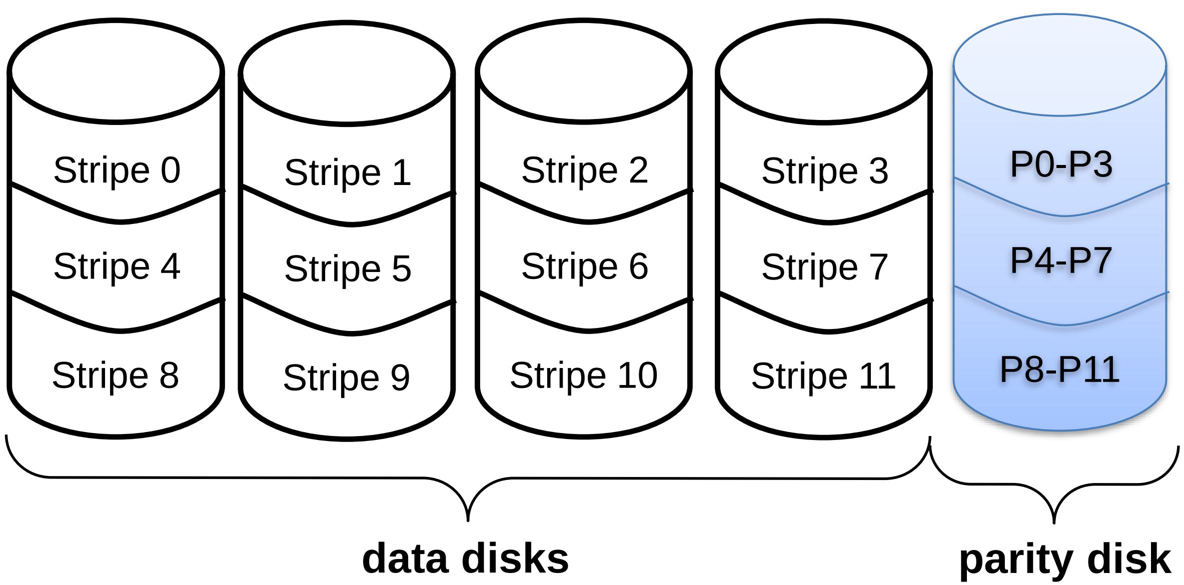
RAID Parity Block

- Same idea at the disk block level
- Block on parity disk =

  XOR of bits of data blocks at same position

RAID-4

- Read: read data disks
- Write: write data disks and parity disk
- Crash: recover from data and parity disk



Issue with RAID-4?

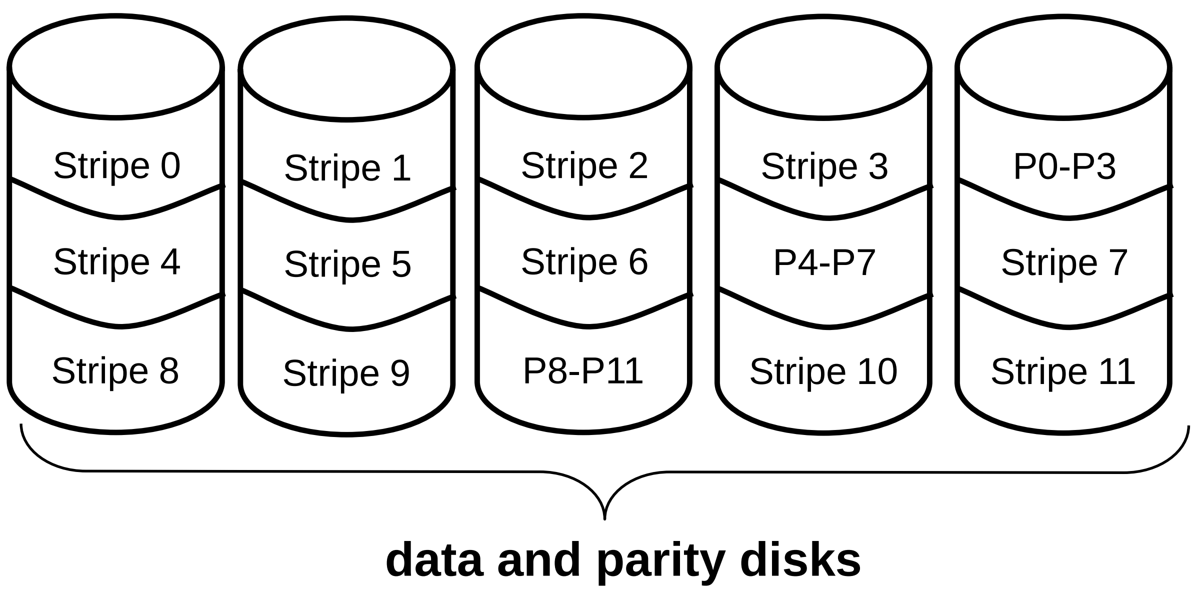
Issue with RAID-4

- Every write has to access parity disk
- Becomes bottleneck for write-heavy workload

How to do better?

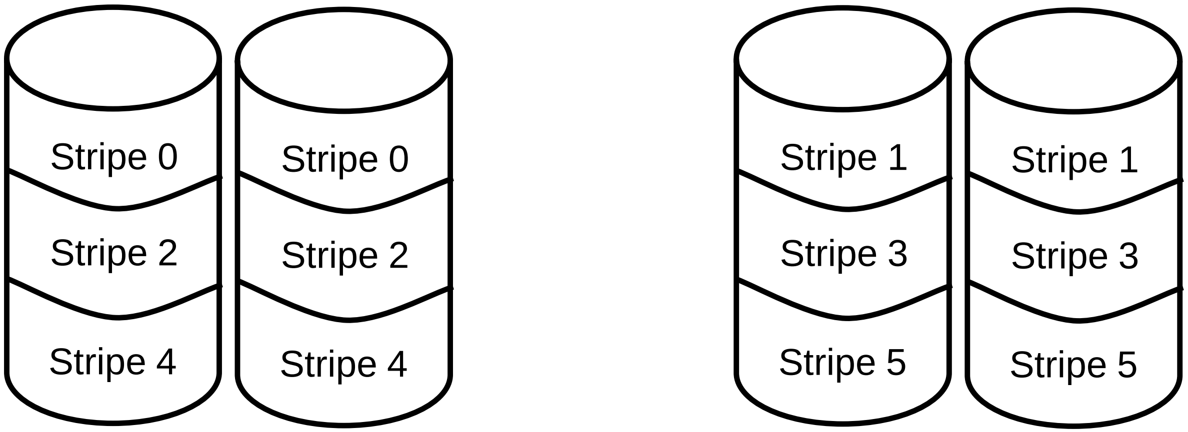
RAID-5

- Block interleaved distributed parity
- As RAID-4, but parity distributed over all disks
- Balances parity write load over disks



Additional Levels

- RAID-6: double parity
  - Like RAID-5 but double parity
- RAID-1+0 = RAID-0 of RAID-1 configurations

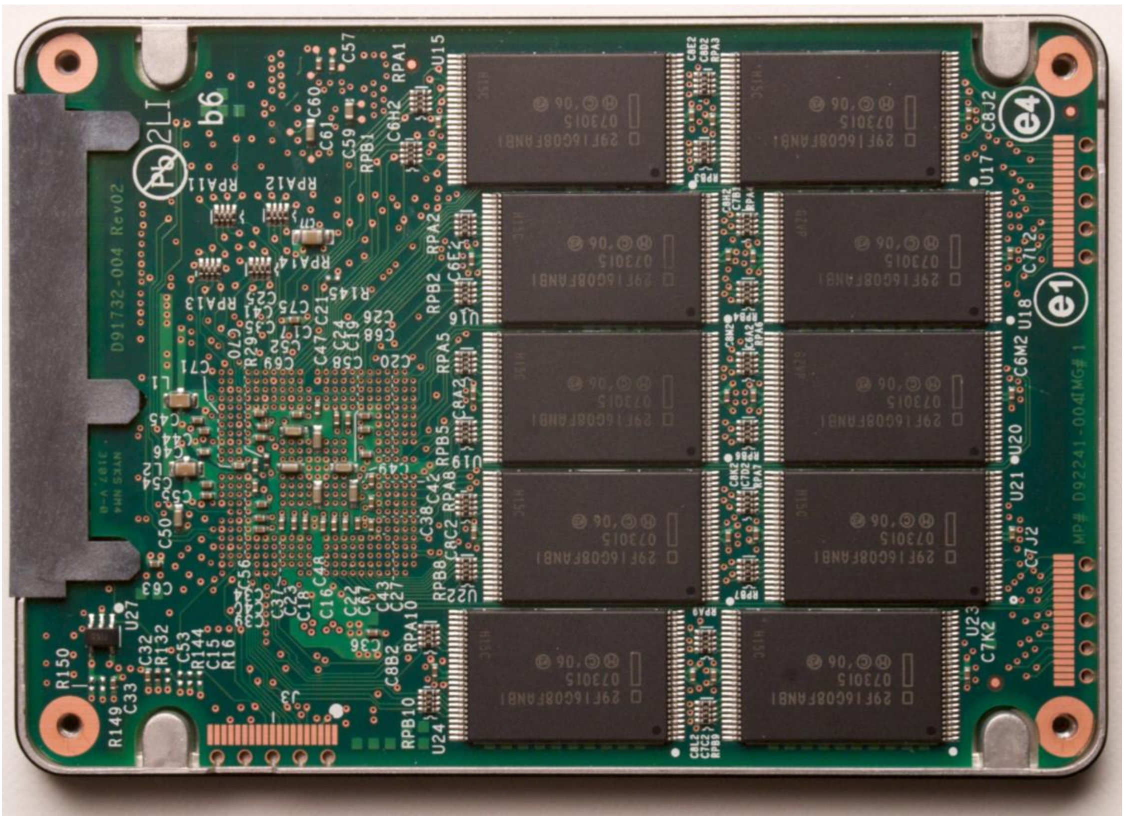


  - Good performance + reliability
  - Expensive

Summary: RAID

- Disk bandwidth not improving very fast
- Disk size and cost improving fast
- RAID provides higher
  1. Performance → parallel I/O, better bandwidth
  2. Reliability → data spread, redundancy
  3. Capacity
  4. Transparency → easy to deploy

SSD: Solid State Disk



* DRAM + battery
- Flash memory technologies
- …

SSD: Solid State Disk

- Is not a disk!
- Purely electronic (NAND Flash)
- Has no moving parts

- Made to look like disk
  - To the hardware (same form factor)
  - To the software (same sector interface)

Basics about NAND Flash

- Basic unit: page – 4k
- Block = set of pages – e.g., 64 pages

NAND Flash Operations

- Read(page) 10s microseconds
- Write/Program(page) 100s microseconds
  - *Cannot rewrite a page*
- Erase(block) few milliseconds
  - Necessary before page in block can be rewritten
  - Set all block bits to 1
  - Limited number of erase cycles (100,000s)

NAND Flash Operations

- Since block must be completely erased before any single page in a block is written
- Typically pages written sequentially in a block

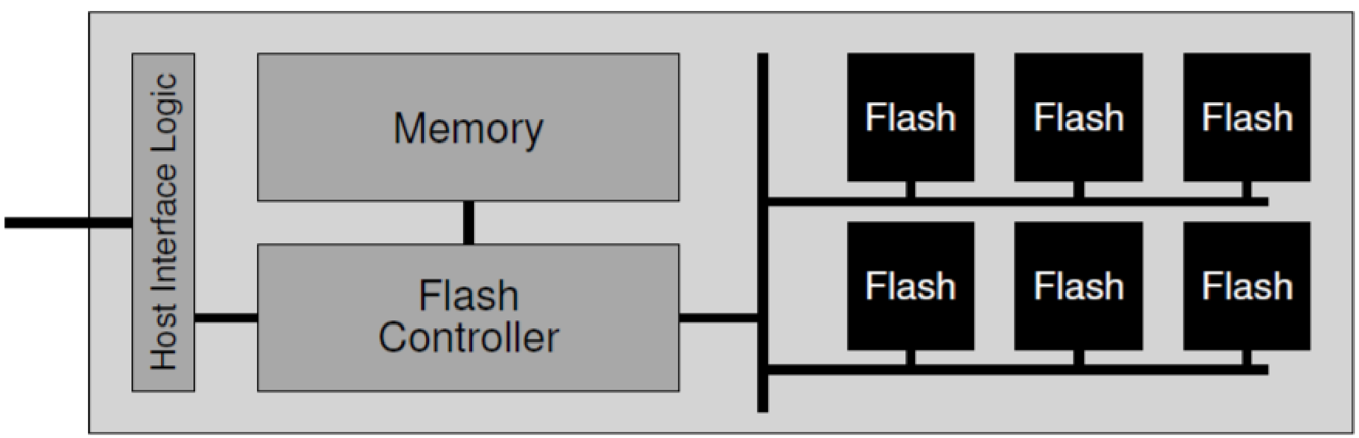
SSD Interface

- Very much like a disk
- ReadSector(logical_sector_no, buffer)
- WriteSector(logical_sector_no, buffer)

- Logical sector map maintained on device

SSD Characteristics

- Bandwidth higher than disk
- Latency: much lower than disk
  - 10 usec for read, 100usec for write
- Several outstanding commands



Flash based SSD: logical diagram

SSD Uses

- Mobile consumer electronics
- Laptop disk replacement
- High end acceleration for short reads/writes
- Often in addition to disk

Building a File System for SSD

- Need to write sequentially
- Cannot overwrite
- Need to erase block before writing again

- What does this remind you of?

File System for SSD ~= LFS

- Clean block before erasing
- Move live data to new block
- Erase block

The TRIM Command

- TRIM(range of logical_sector_no’s)
- Indicate which data blocks no longer in use
- No need to do cleaning, just erase

How to Pick Which Block to Clean?

Wear Leveling

- How to pick block to clean?
- Try to even out number of erase cycles

Evolution of SSDs

- Older SSDs:
  - Need to build LFS-like file system in software
- Newer SSDs:
  - Done inside firmware of device

Latest Evolution

- NVM = Non-Volatile Memory
- Allows byte-level access to NV storage
  - Just like memory
  - Somewhat slower
  - But not as slow as conventional NV

A Note: NVMe

- Confusingly called NVMe
- High-speed SSD on PCIe I/O bus
- Not memory!
- But supports many parallel I/Os
- Lower latency, higher bandwidth

Summary: SSD

- Solid State Disk
- Good for
  - Response time
  - Bandwidth
  - Robustness
- Not so good for
  - Price
  - Capacity