Week 9 - Recap

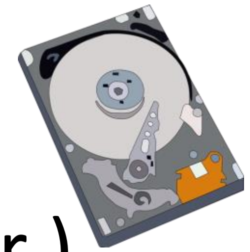
Pamela Delgado

May 1, 2019

File System Implementation

- The main task of the file system is to translate

- From user interface methods
- Read(uid, buffer, bytes)

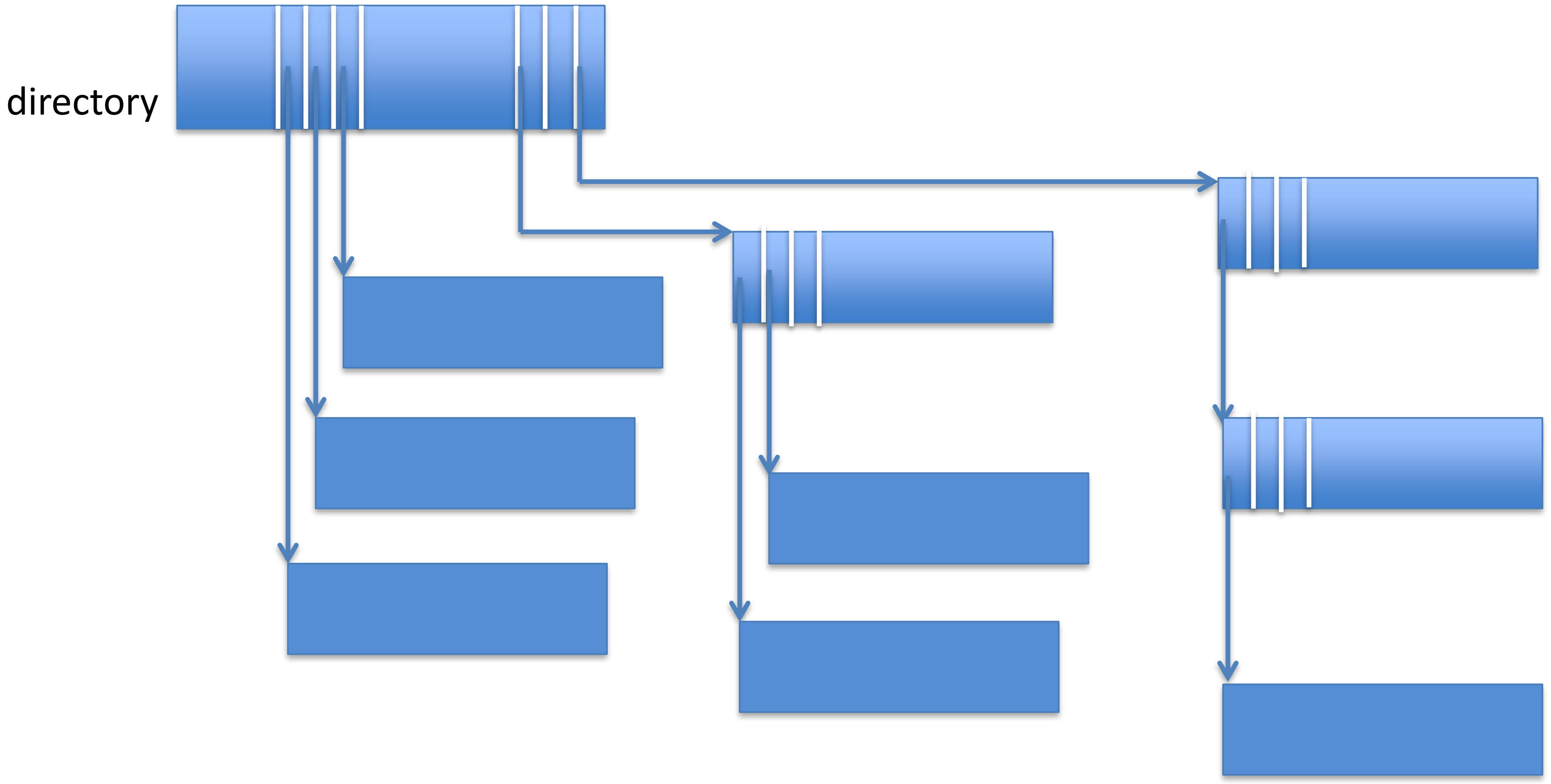


- To disk interface methods
- ReadSector(logical_sector_number, buffer)

Disk Data Structures

- Boot block
- Device directory
- User data
- Free space

Indexed Allocation with Indirect and Double-Indirect Blocks



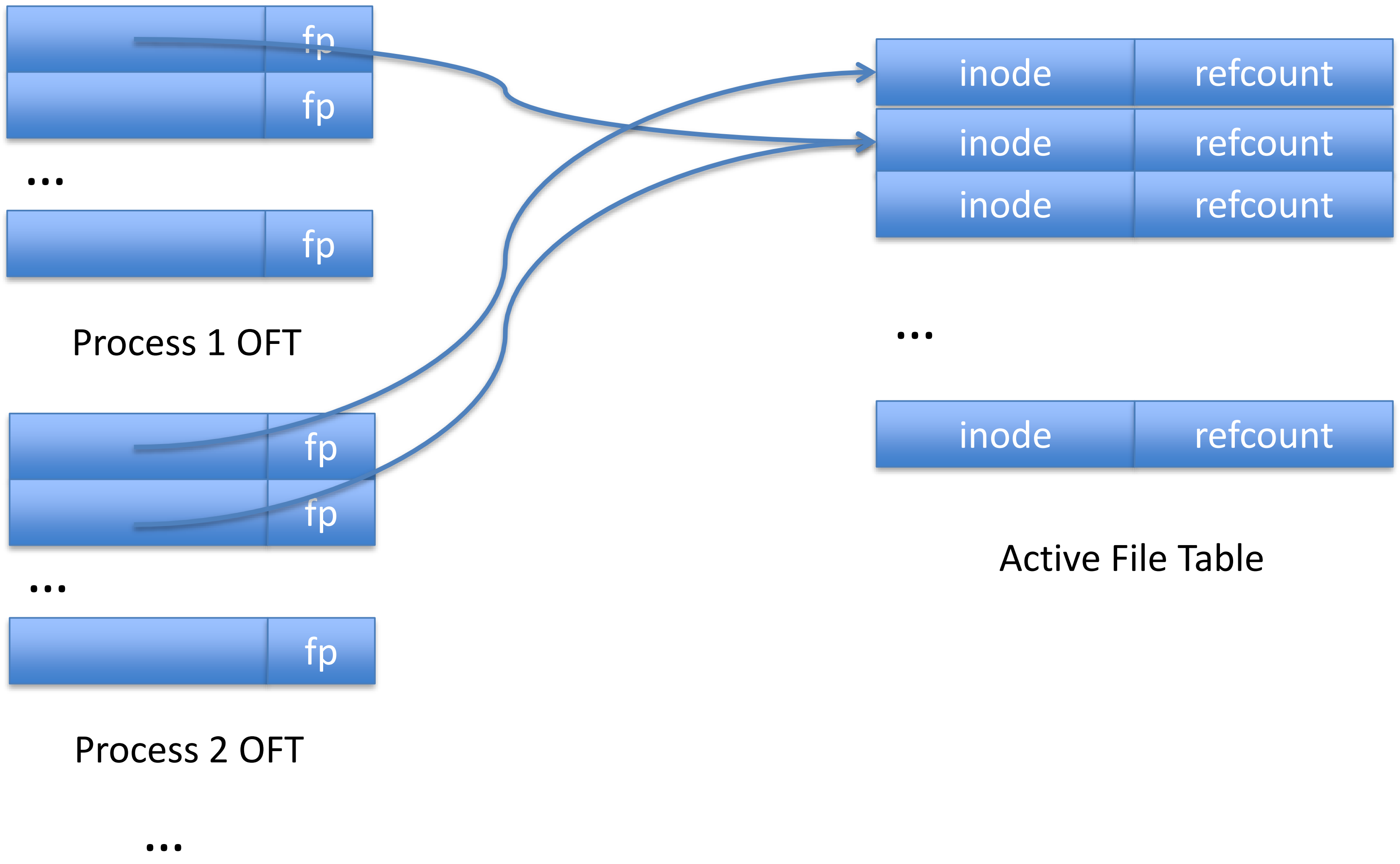
Exemplifies Good System Design

- Optimizes the common case (small files)
- Accommodates the uncommon case (large files)

In-Memory Data Structures

- Cache
- Cache directory
- Queue of pending disk requests
- Queue of pending user requests
- Active file table
- Open file tables

Picture of Open File Tables



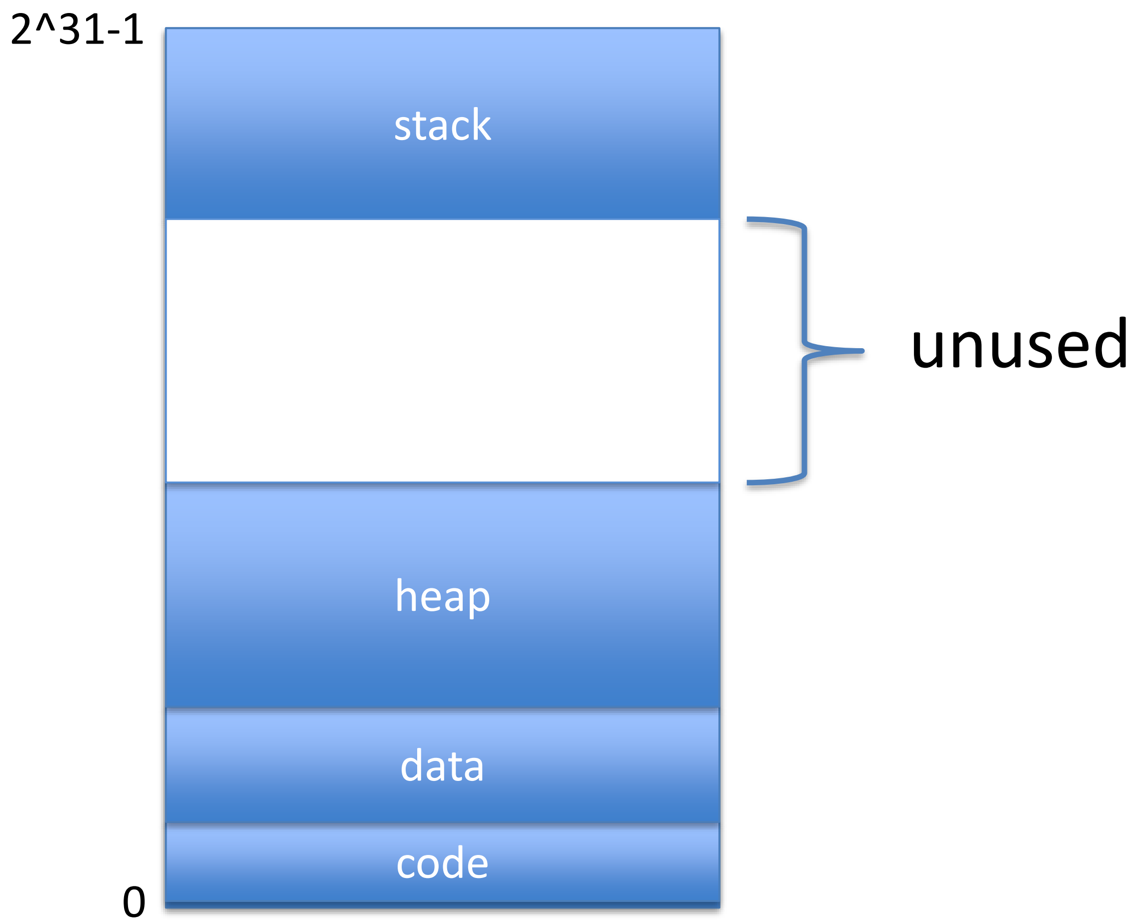
Exemplifies good system design

- Allows open file data to be shared
- But with separate file pointer per open

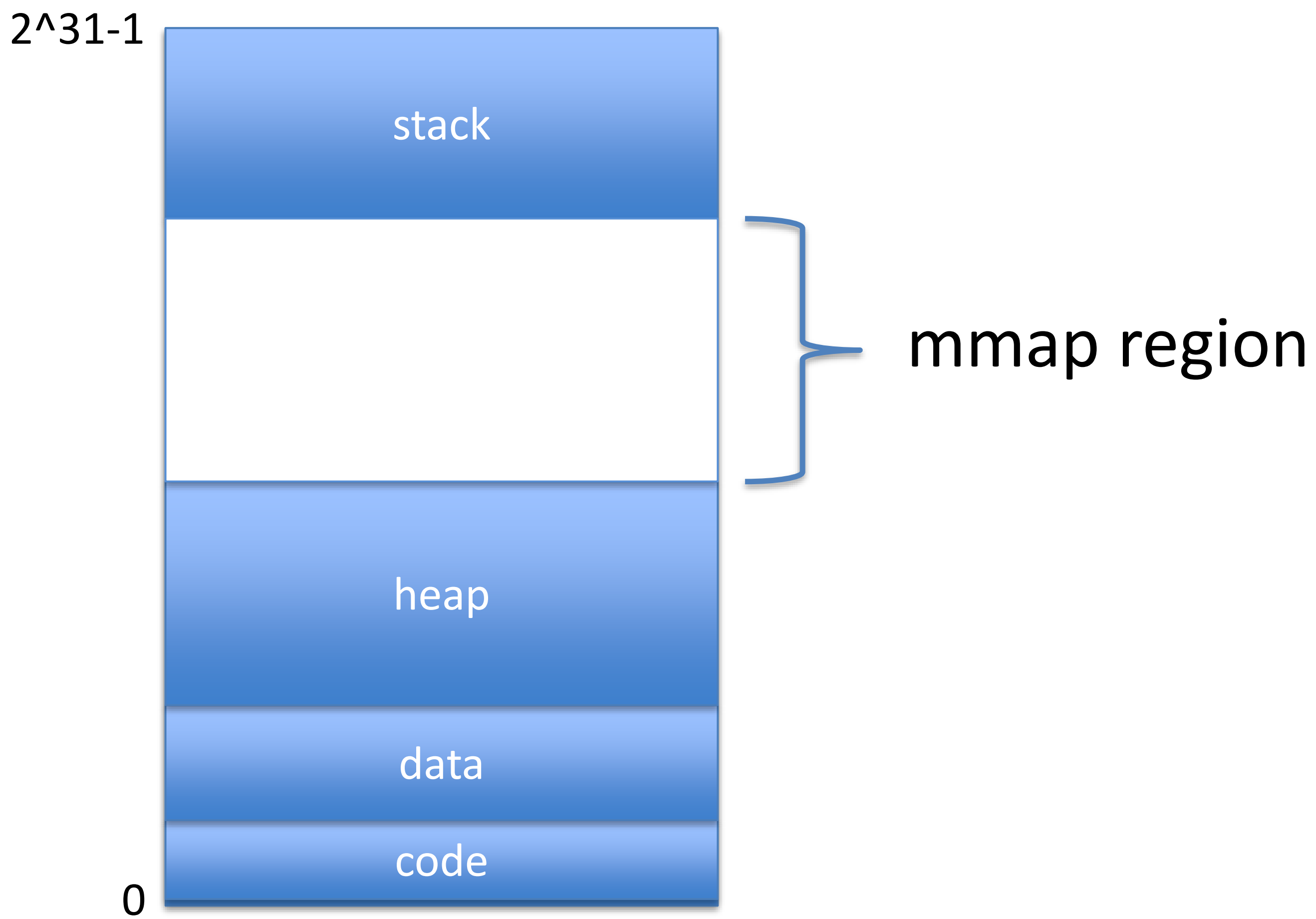
Alternative File Access Method: Memory Mapping

- mmap()
  - Map the contents of a file in memory
- munmap()
  - Remove the mapping

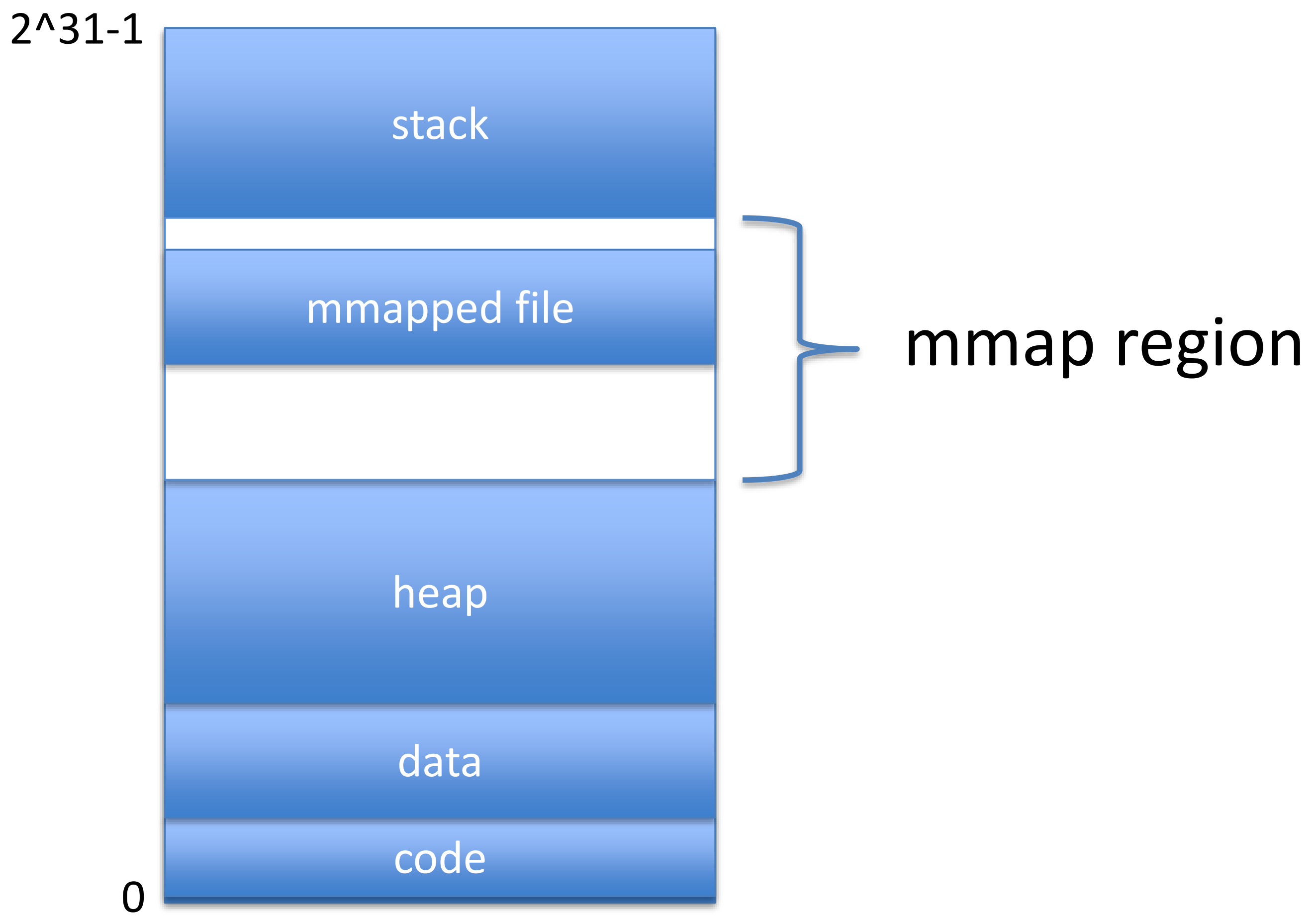
Remember this Picture? Typical Virtual Address Space



Remember this Picture?
Typical Virtual Address Space



Remember this Picture?
Typical Virtual Address Space



Access to mmap()-ed Files

- Access to memory region mmap()-ed
- Causes page fault
- Causes page/block of file to be brought in

mmap() implementation

- On mmap()
  - Allocate page table entries
  - Set valid bit to “invalid”
- On access,
  - Page fault
  - File = backing store for mapped region of memory
  - Just like in demand paging
  - Except paged from mapped file
- After page fault handling
  - Set valid bit to true

How to get data to disk for mmap?

- Through normal page replacement
- Or through an explicit call *msync()*

What is mmap() good for?

- Random access to large file

mmap vs read()/lseek()

- read
  - ✗ Entire file read into memory
- lseek
  - ✗ Not easy to write-reuse (lseek+read every time)
- mmap
  - ✓ Only load needed portions
  - ✓ Easy to write-reuse

Huge advantage for large files sparsely accessed

Issues with mmap()

- Alignment on page boundary → unused space
- Not easy to extend a file
- For small files
  - Read() more efficient than mmap() + page fault

Week 10
Dealing with Crashes

Pamela Delgado

May 8, 2018

based on:
- W. Zwaenepoel slides
- Arpaci-Dusseau book

Consider this Piece of Code

- fd = Open(file)
- Write(fd, 0)
- Write(fd, 1)
- Write(fd, 2)
- Write(fd, 3)
- Close(fd)

Machine Crash 1

- fd = Open(file)
- Write(fd, 0)
- Write(fd, 1)
- Write(fd, 2)
- Write(fd, 3)
- Close(fd)

$\leftarrow$ crash

- Not really a problem (old file is there)

Machine Crash 2

- fd = Open(file)
- Write(fd, 0)
- Write(fd, 1)
- Write(fd, 2)
- Write(fd, 3)
- Close(fd)

$\longleftarrow$ crash

- Not really a problem (new file is there)

Machine Crash 3

- fd = Open(file)
- Write(fd, 0)
- Write(fd, 1)
- Write(fd, 2) ← crash
- Write(fd, 3)
- Close(fd)

- It is a problem: half of old, half of new file

With Write-Behind

- fd = Open(file)
- Write(fd, 0)
- Write(fd, 1)
- Write(fd, 2)
- Write(fd, 3)
- Close(fd)

← crash

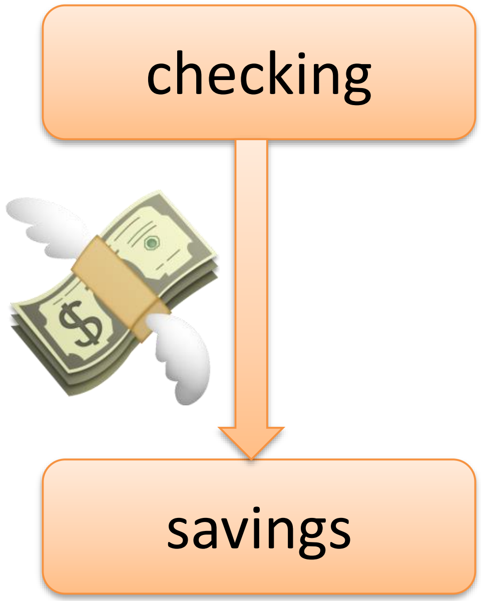
- It could be a problem (new file perhaps not there)

The Notion of Atomicity

- Atomicity means “all or nothing”
- Atomicity in a file system means
  - All updates are on disk
  - No updates are on disk
  - Nothing in-between!

It can be Important

- Read(balance_checking)
- Balance_savings -= 100
- Write(balance_checking)
- Read(balance_savings)
- Balance_checking += 100
- Write(balance_savings)



It can be Important

- Read(balance_savings)
- Balance_savings -= 100
- Write(balance_savings)
- Read(balance_checking) ← crash
- Balance_checking += 100
- Write(balance_checking)

- Your 100CHFs are gone!

How to Implement Atomicity

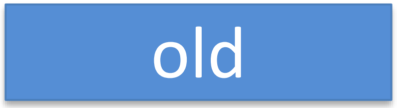
- In other words:
- How to make sure that all or no updates to an open file get to disk?

Assumption

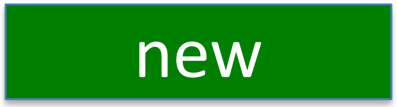
- A single sector disk write is atomic

Assumption

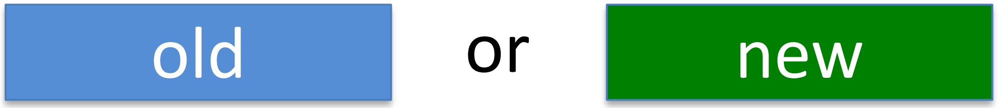
- Before WriteSector



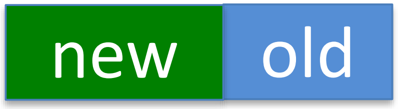
- After WriteSector returns successfully



- If failure



never:



Assumption True?

- With very high probability (99.999+%): yes
- Disk vendors work very hard at this
- We will assume it is true

How to Implement Atomicity?

- Make sure you have old copy on disk
- Make sure you have new copy on disk
- Switch atomically between the two

How to Switch Atomically?

- By doing a WriteSector()
- What to write in WriteSector()?

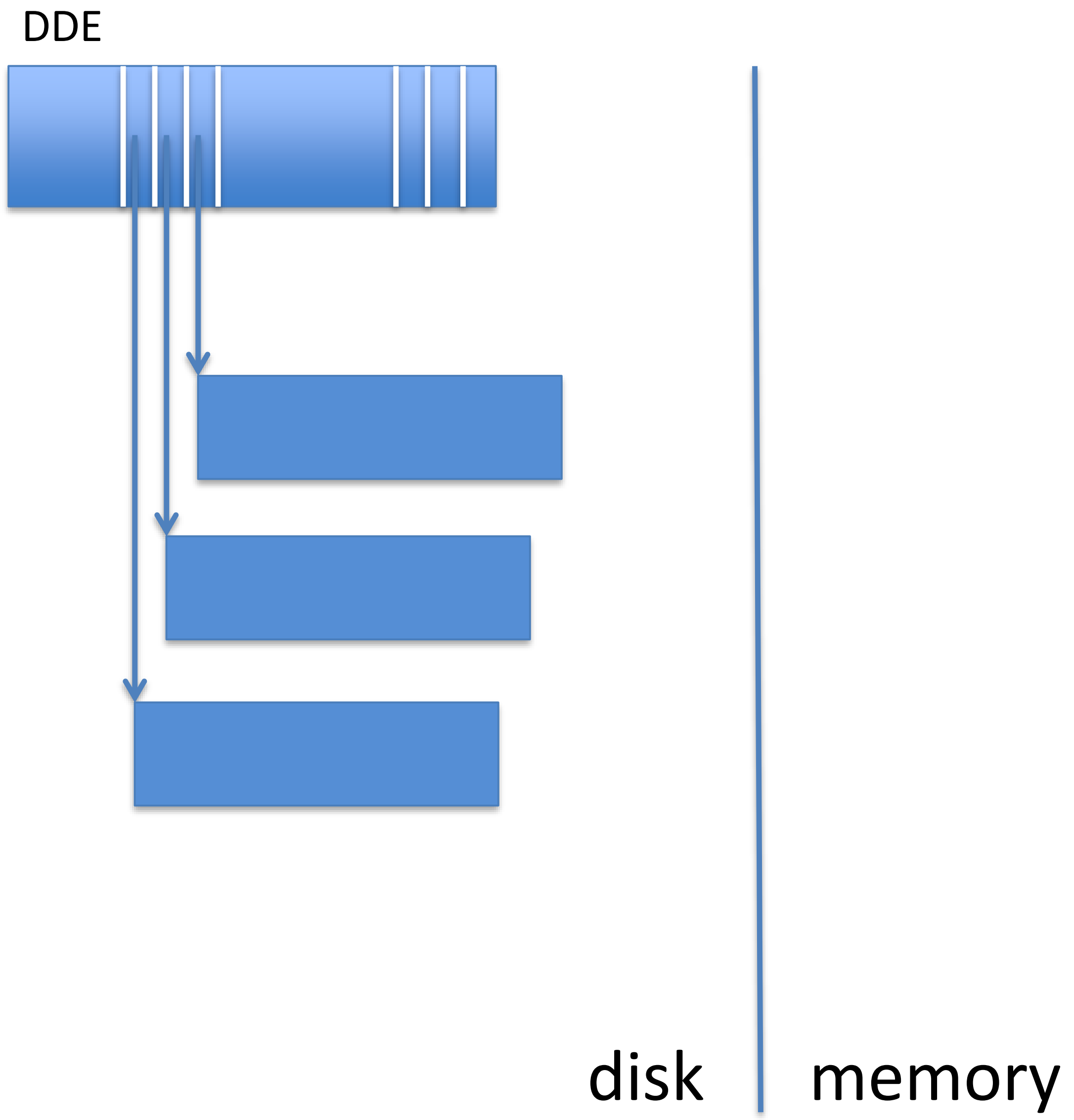
How to Switch Atomically?

- By doing a WriteSector()
- What to write in WriteSector()?
  - Device directory entry!
  - Because it is smaller than sector

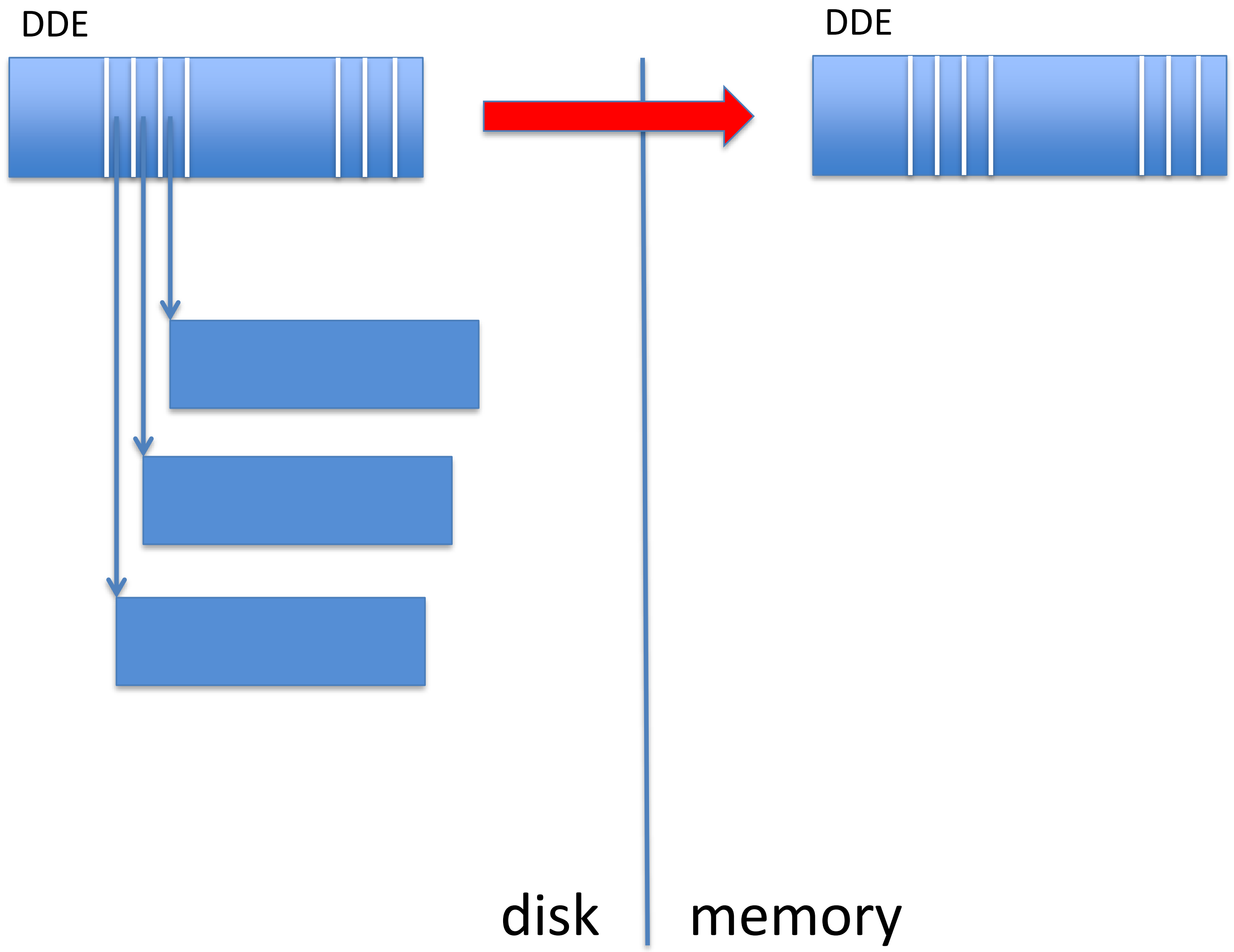
How It Works (with Write-Through)

- Open()
  - Read DDE into AFT
- Write()s
  - Allocate **new** blocks on disk for data
  - Fill in address of new blocks to *memory* DDE
  - Write to cache and disk
- Close()
  - Write *memory* DDE to *disk* DDE

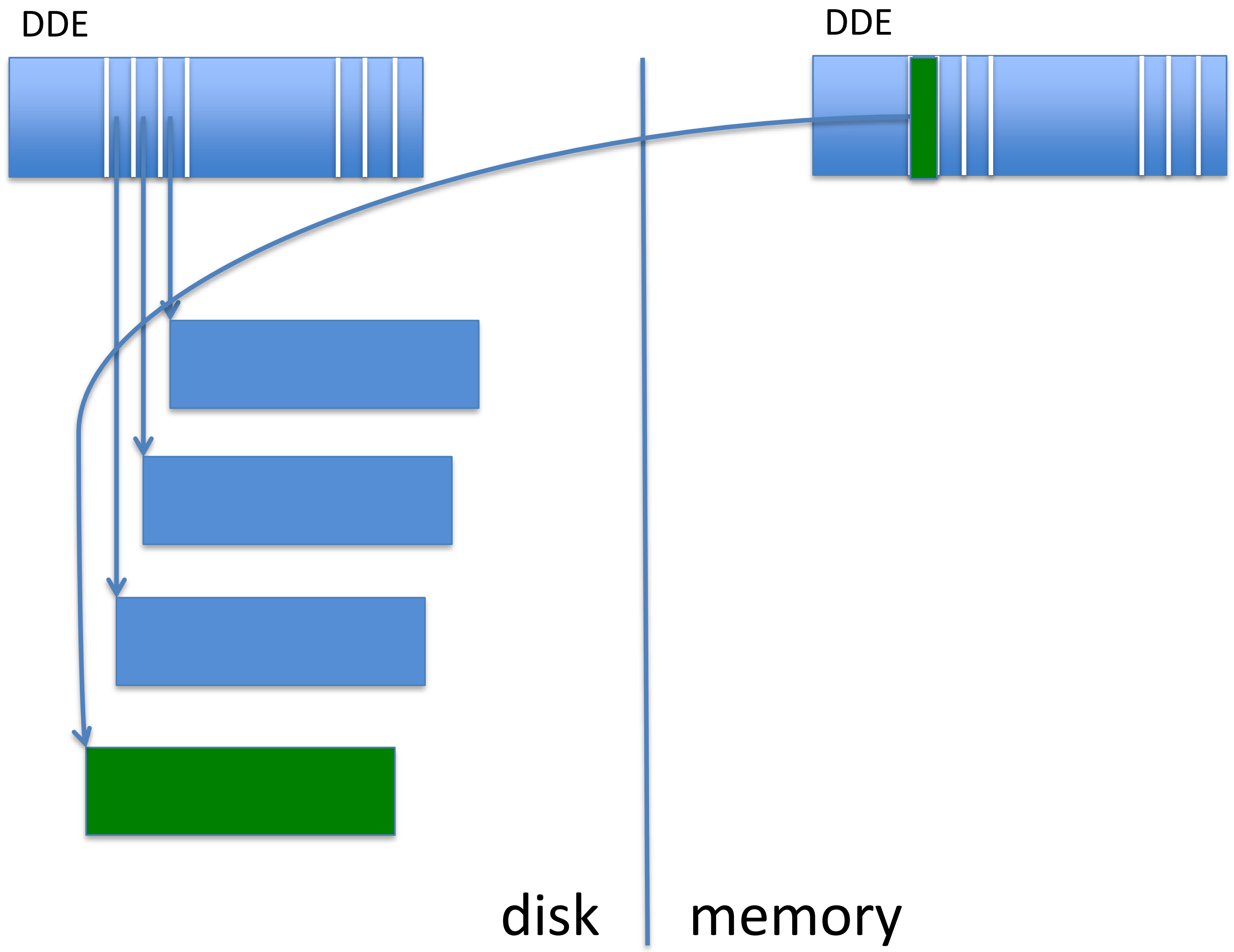
Initial State



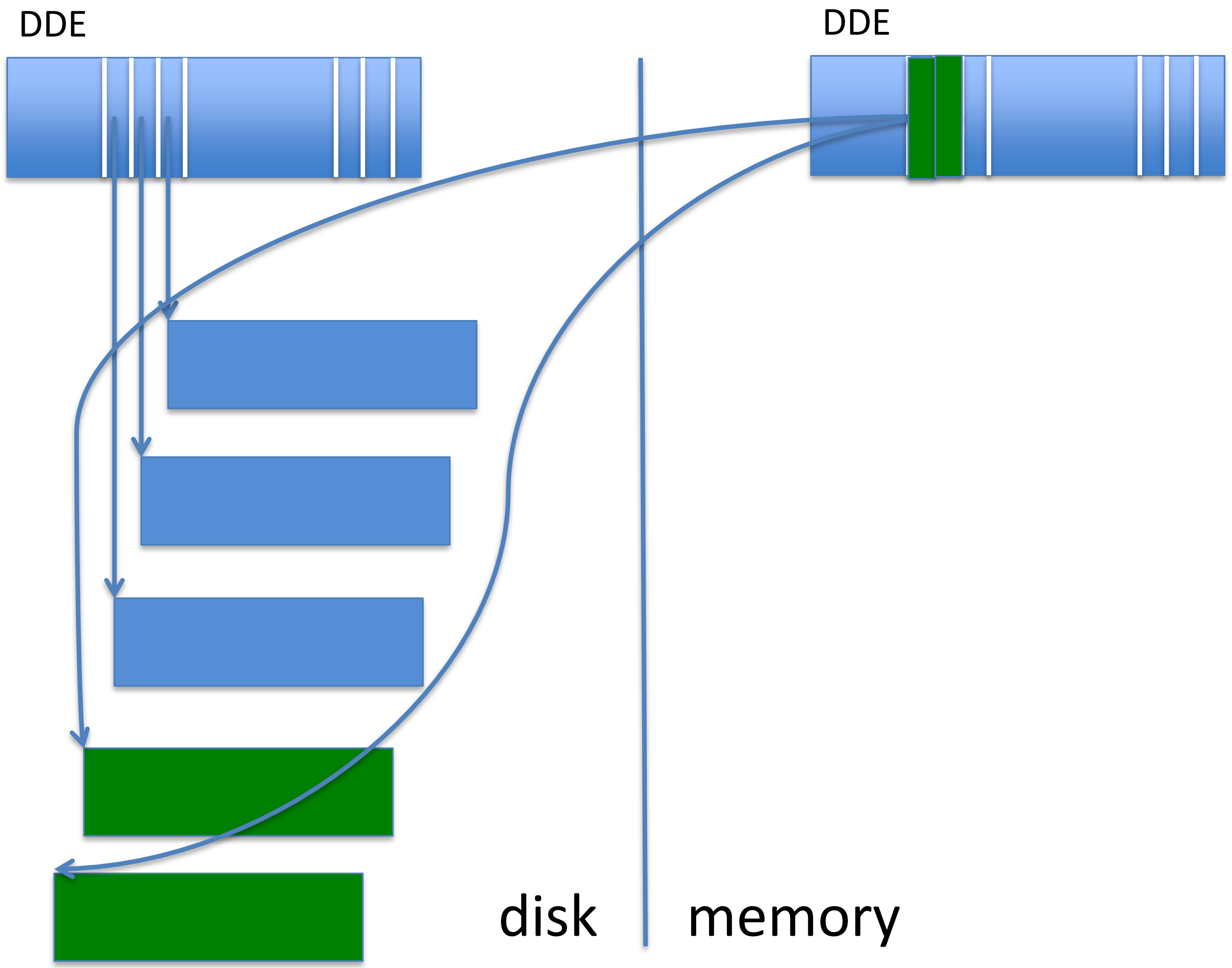
Open()



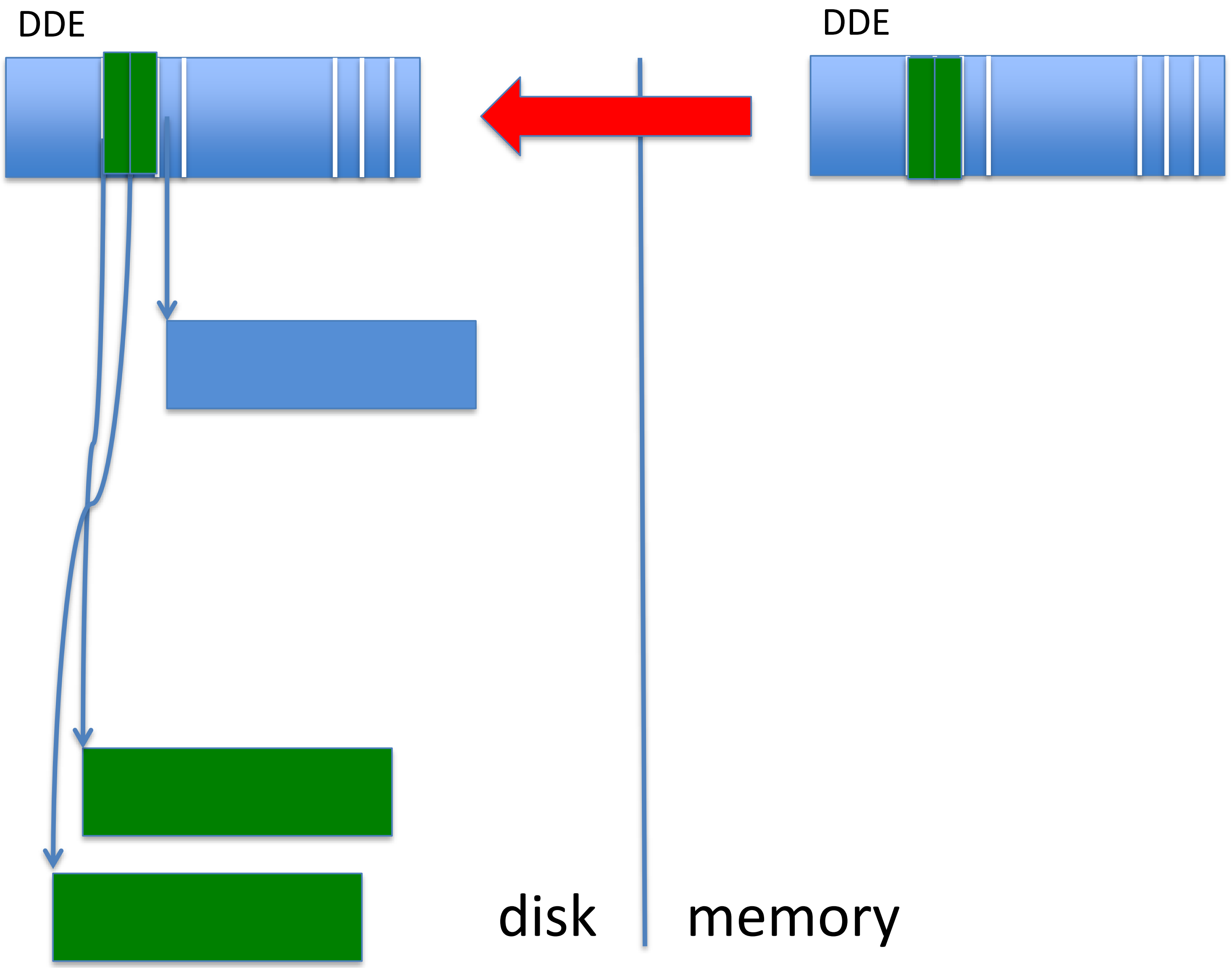
Write(block0)



Write(block1)



Close()



How it Works (with Write-Behind)

- Open()
  - Read DDE from disk into AFT
- Write()s
  - Allocate **new** blocks for new data
  - Fill in address of new blocks to *memory* DDE
  - Write to cache
- Close()
  - Write all **cached blocks** to new disk blocks
  - Write *memory* DDE to *disk* DDE

What happens to old blocks?

What happens to old blocks?

- De-allocate them
- If crash before de-allocate, file system check

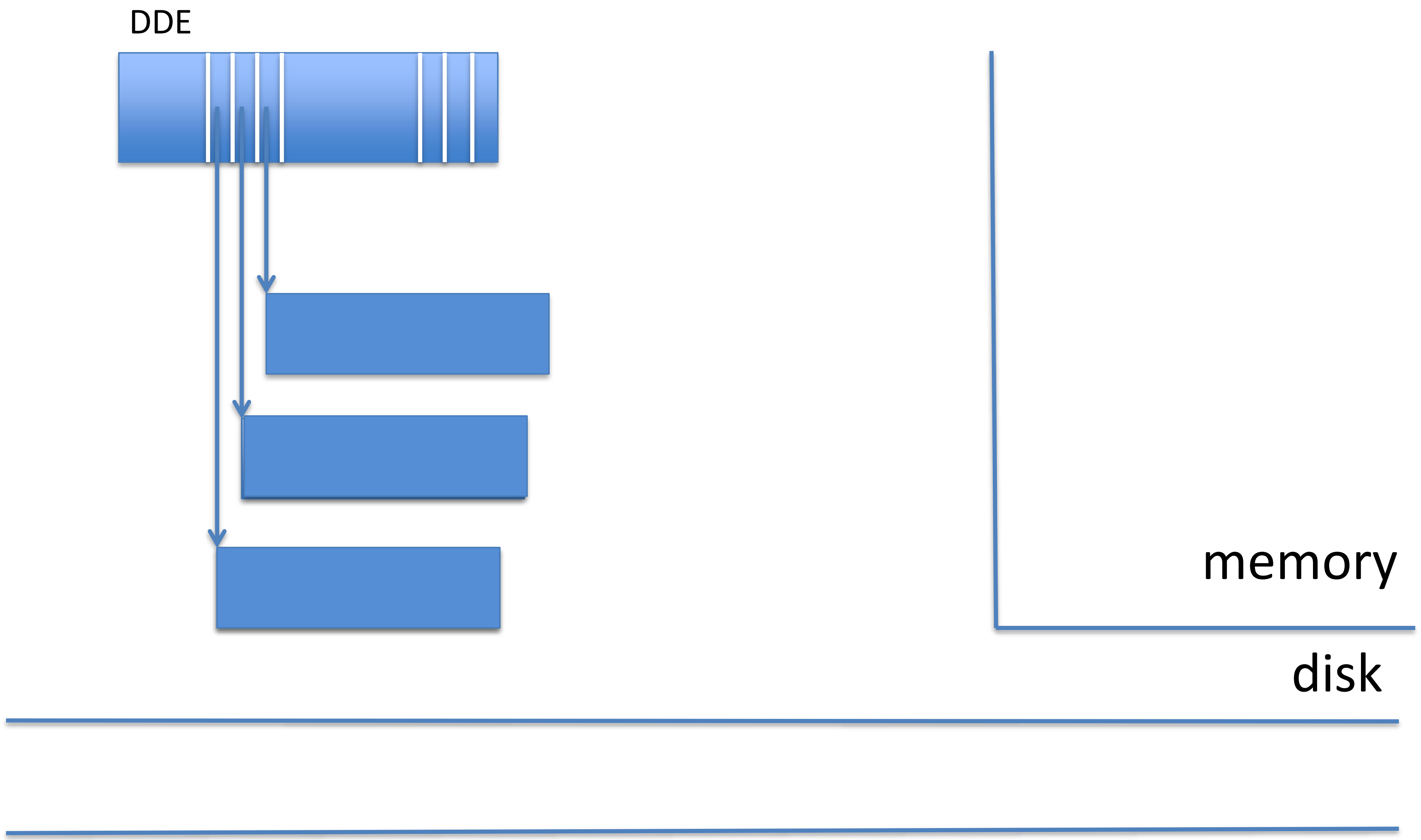
An Alternative Method: Intentions Log

- Reserve an area of disk for (intentions) log

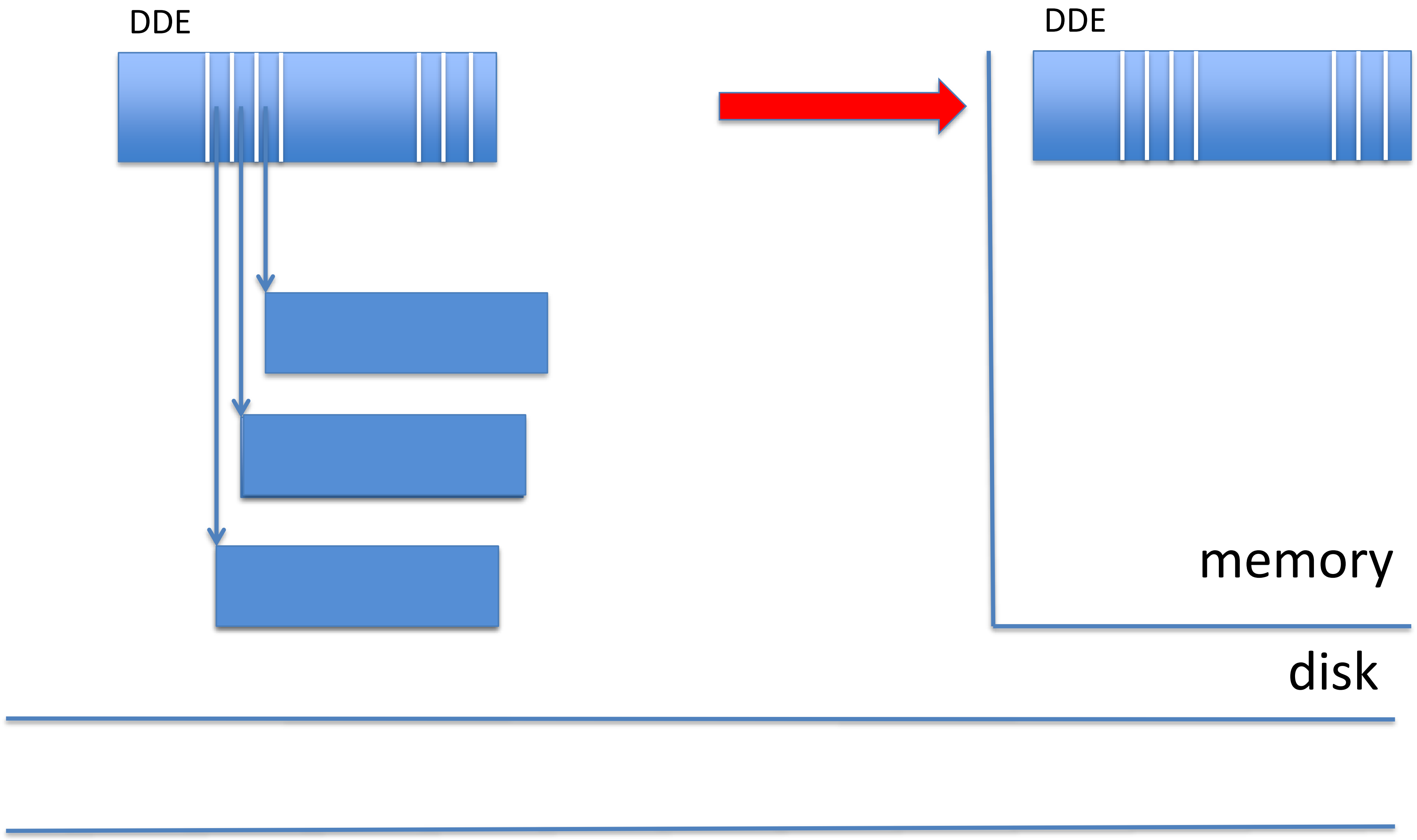
During Normal Operation

- On write:
  - Write to cache
  - Write to log
  - Make in-memory inode point to update in log

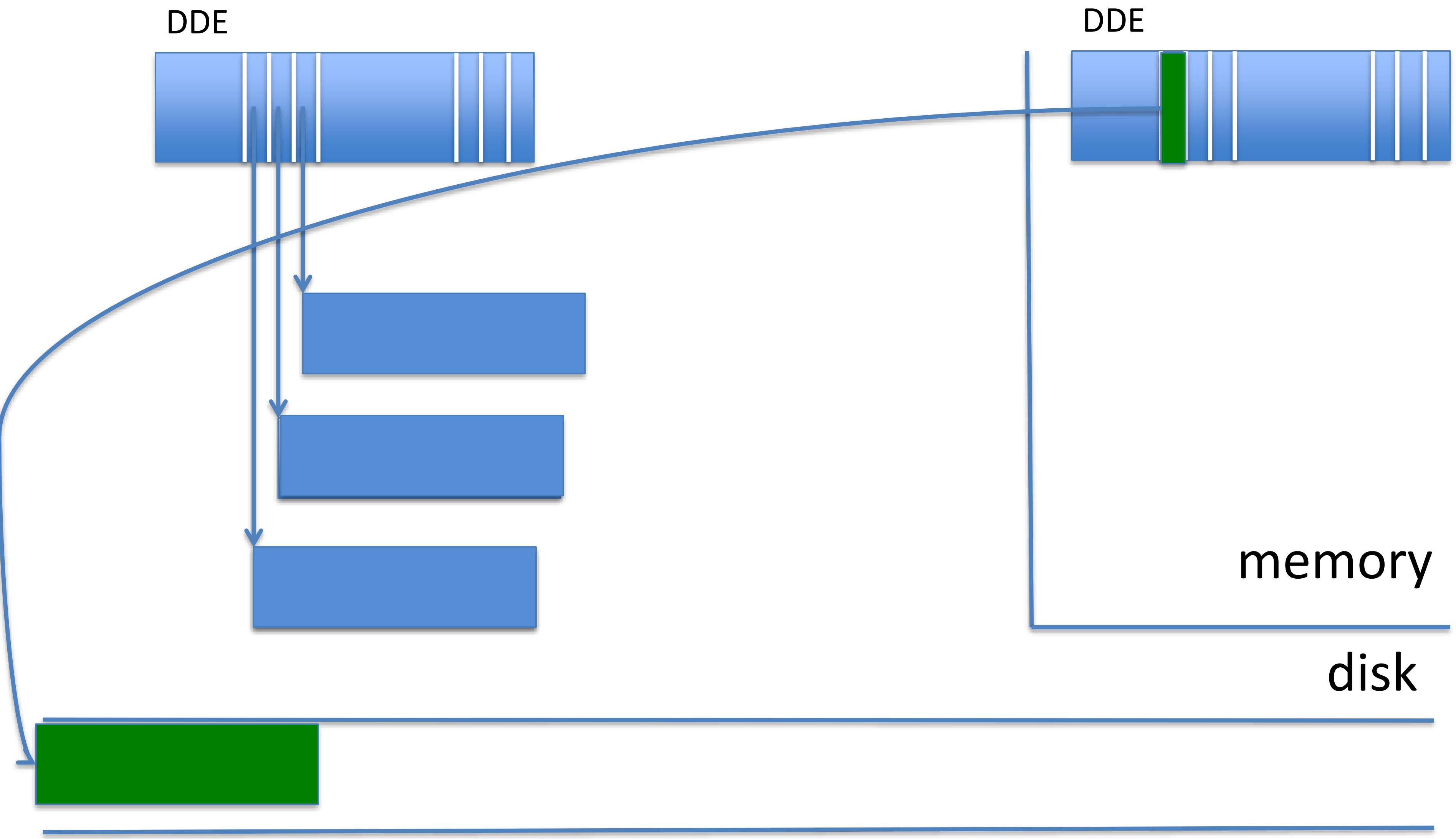
Initial State



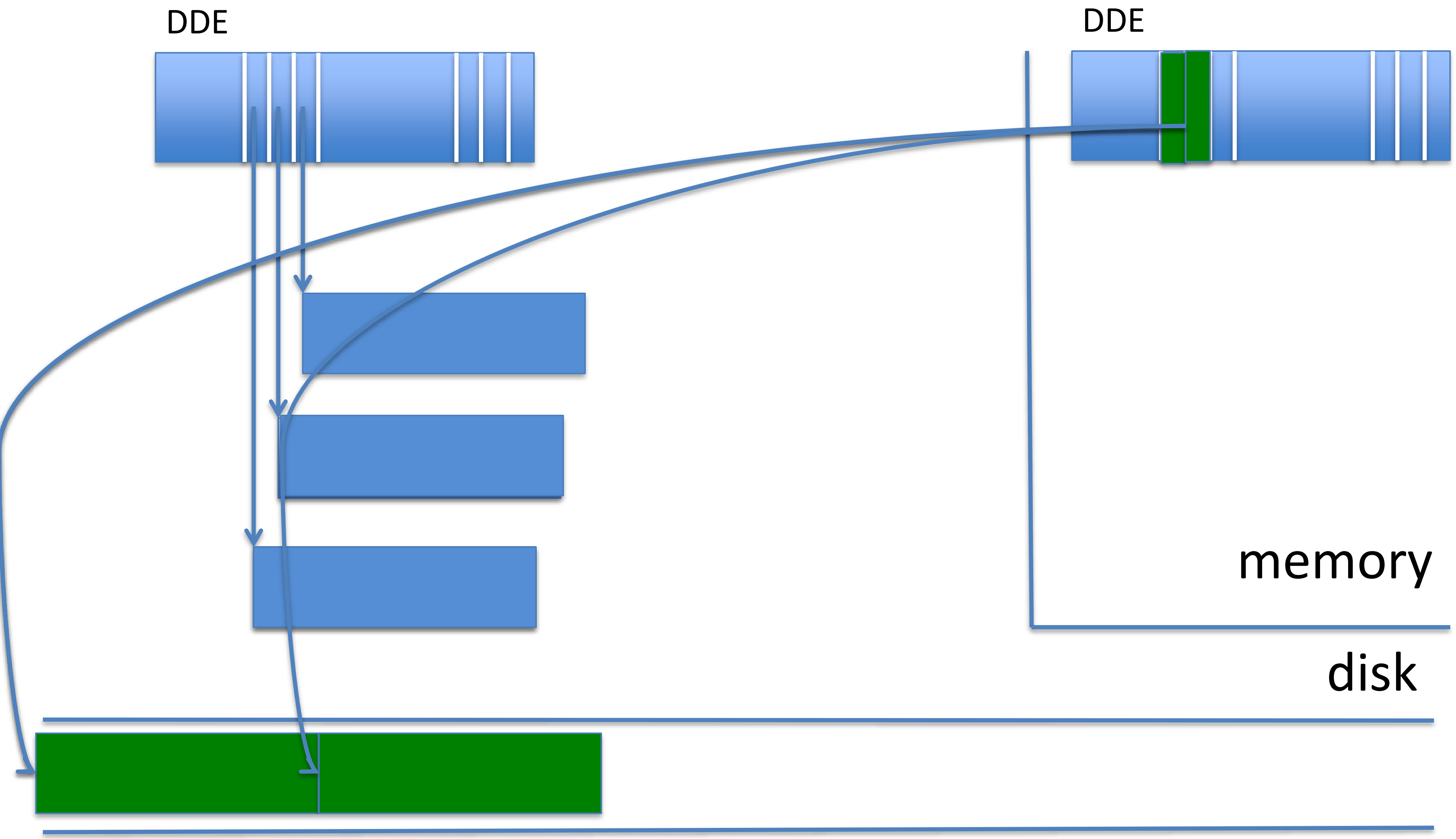
Open()



Write(block 0)



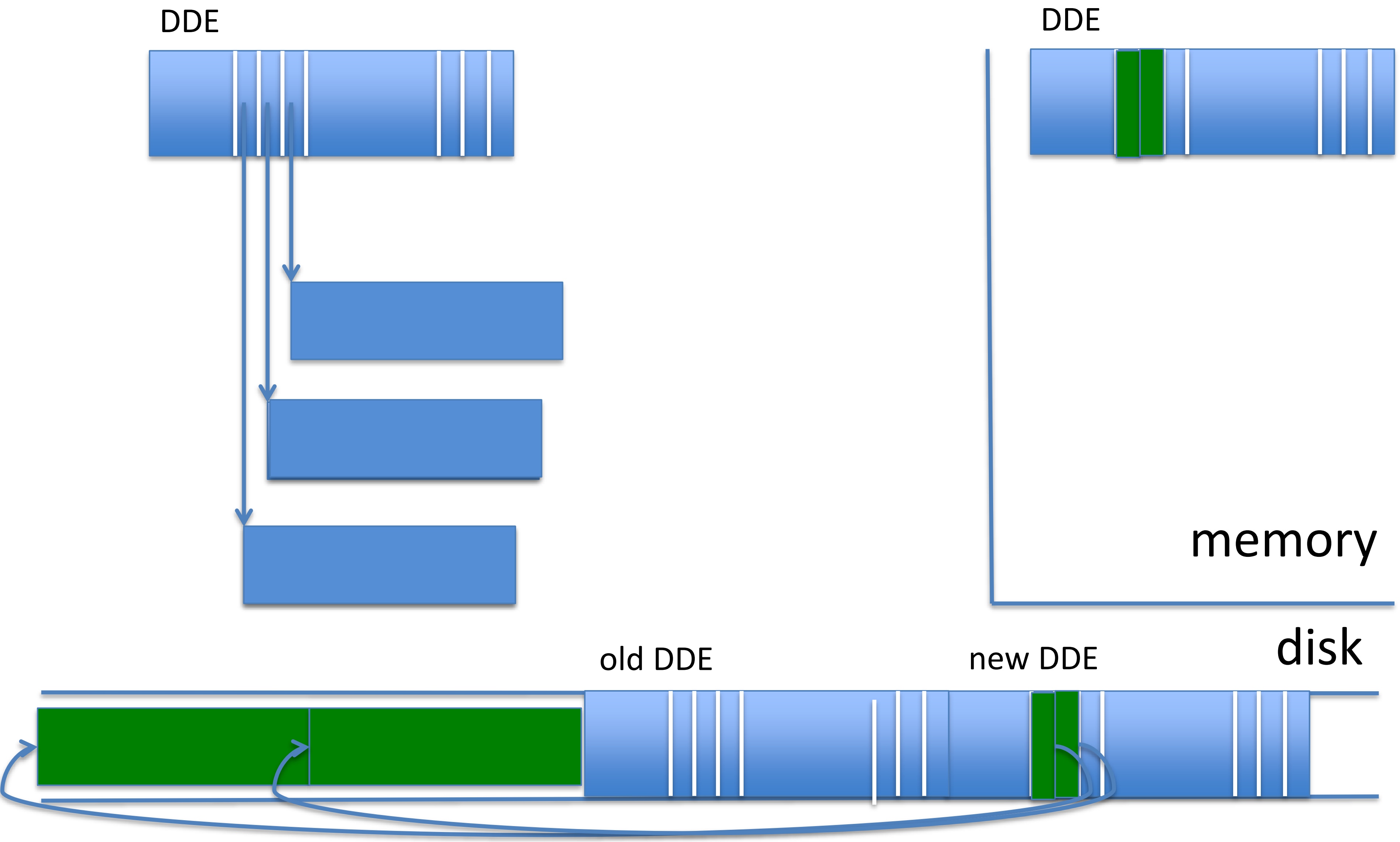
Write(block 1)



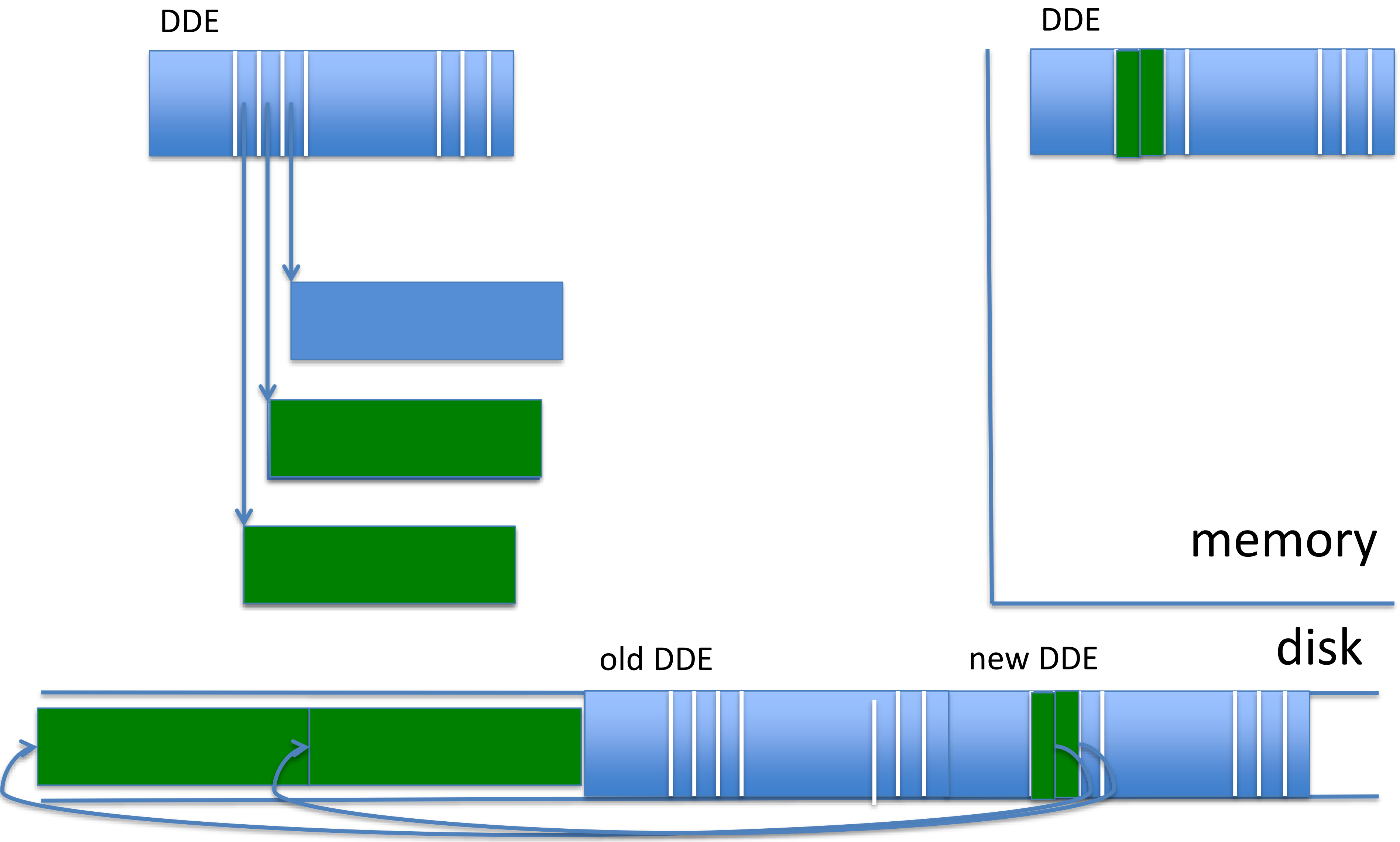
During Normal Operation

- On close:
  - Write old and new inode to log in one disk write
  - Copy updates from log to original disk locations
  - When all updates done, overwrite inode with new value
  - Remove updates and old and new inode from log

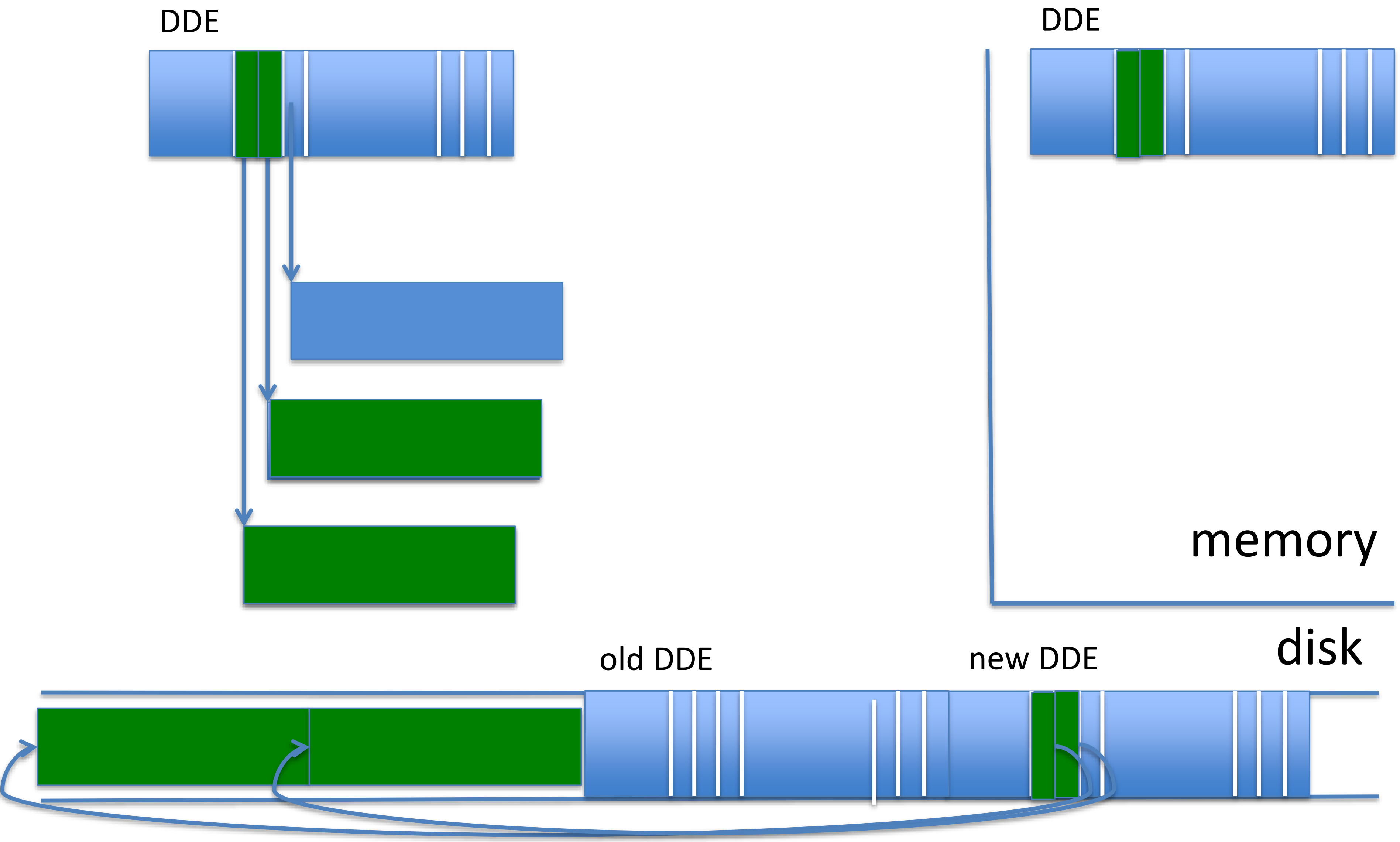
Close()



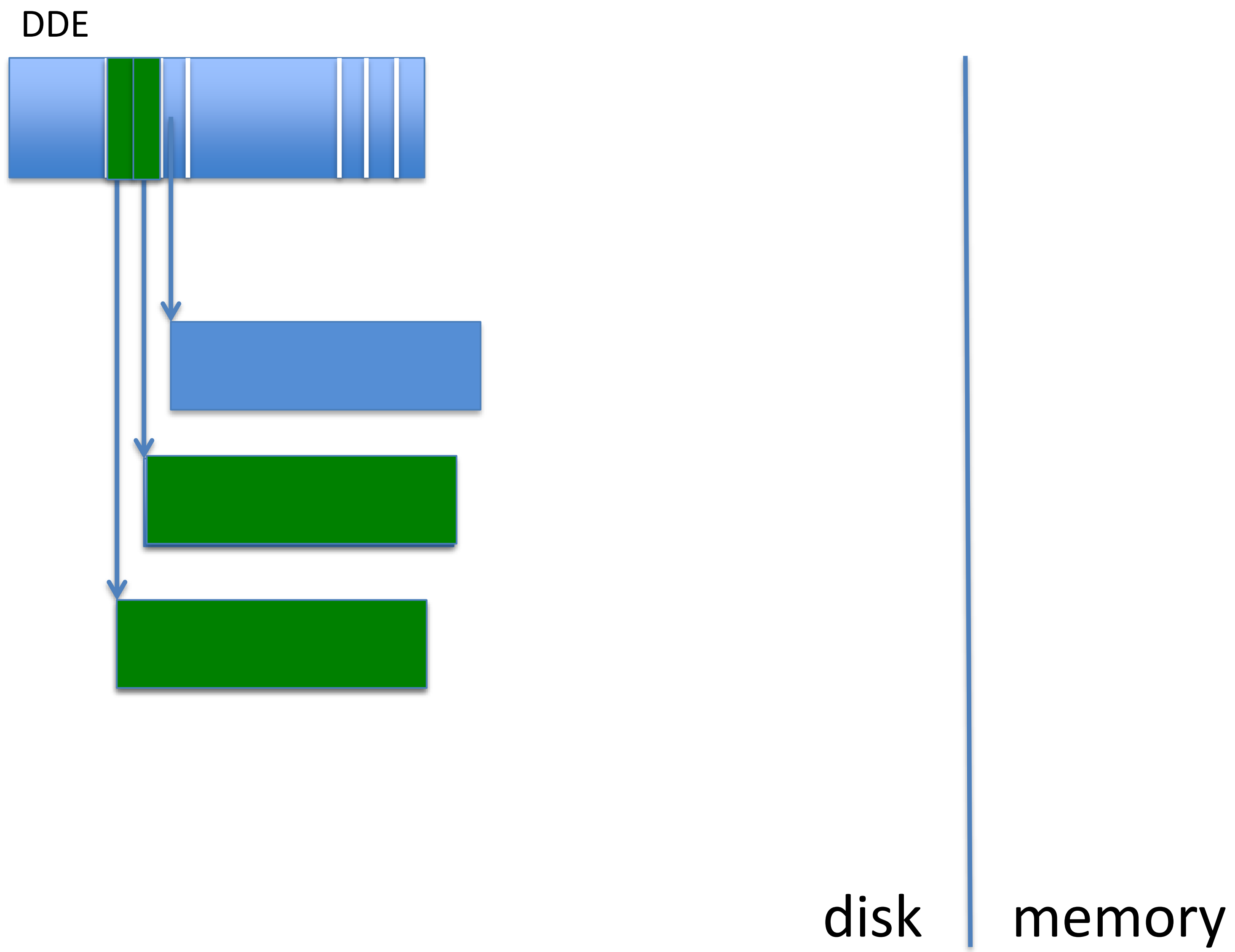
Later: Step 1



Later: Step 2



Later: Step 3



Crash Recovery

- Search forward through log
- For each new inode found
  - Find and copy updates to their original location
  - If/when all updates are done, write new inode
  - Remove updates and old and new inode from log

Invariant

- If new inode in the log and crash: new copy
- If new inode not in the log and crash: old copy

- Even if you crash during crash recovery
  - You may copy an update multiple times

Which One Works Better?

How to Compare File System Methods?

How to Compare File System Methods?

- Count the number of disk I/Os
- Count the number of random disk I/Os

Which Works Better? - DDE

- Write() →
- Close() →

Which Works Better? - DDE

- Write() → one disk write (new block)
- Close() → one disk write (update DDE)

Which Works Better? - Log

- Write() →
- Close() →

Which Works Better? - Log

- Write() → one disk write (block in log)
- Close() → two disk writes (inodes in log, block in data)

Surprisingly, Log works Better

- Write()’s to log are sequential (no seeks)
- Data blocks stay in place
- Good disk allocation stays!
- Write from cache or log to data – when disk is idle or cache replacement

Surprisingly, DDE Works Less Well

- Disk allocation gets messed up
- Fragmentation

Week 10
Log-Structured File System (LFS)

Pamela Delgado

May 8, 2019

based on:
- W. Zwaenepoel slides
- Arpaci-Dusseau book

Log-Structured File System (LFS)

- Alternative way of structuring file system
- Takes idea of log writes to extreme

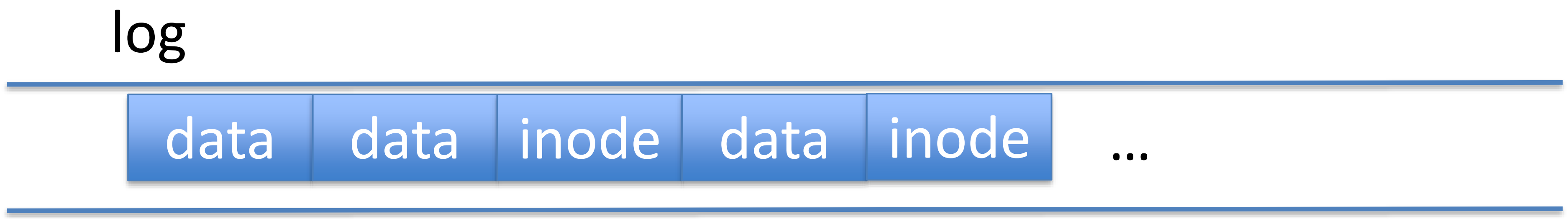
Rationale for LFS

- 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

- Log = append-only data structure (on disk)
- *All* writes are to a log, including
  - Data
  - inode modifications

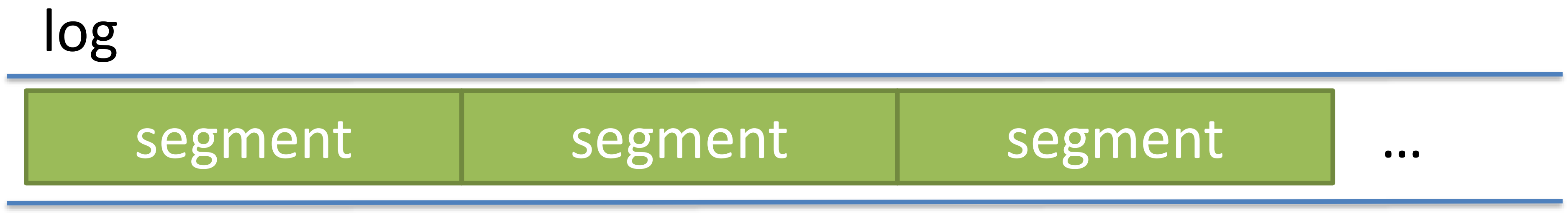
Key Idea in LFS



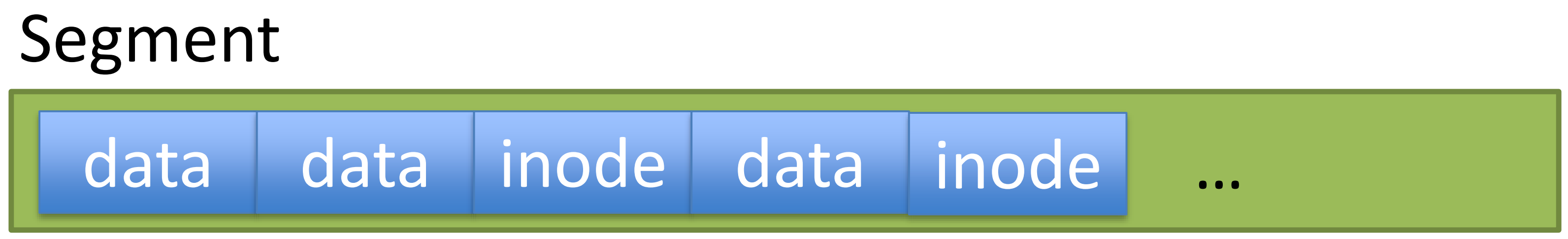
Write() in LFS

- Writes first go into cache (write-behind)
  - Both inodes and data
- Writes also go into (in-memory) buffer
- When buffer full, append to log
- Called *segment* of log
- No seeks on writes!

LFS Log



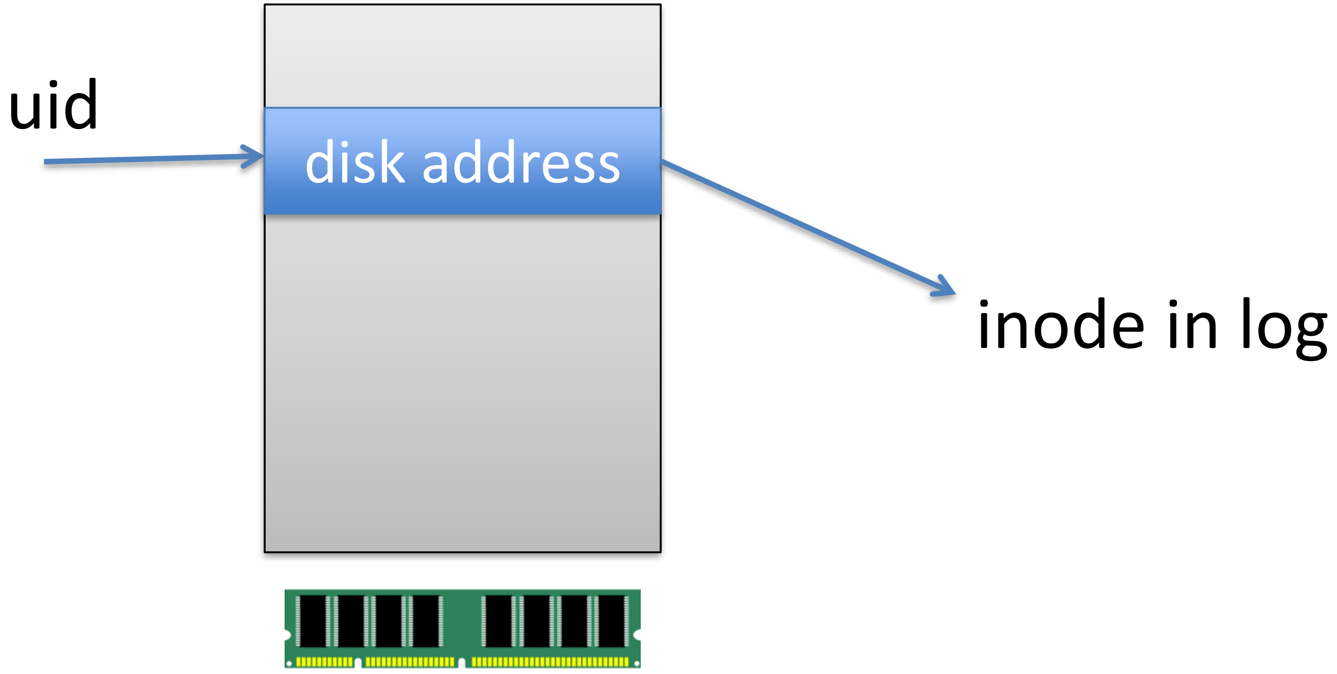
LFS Segments



But how to Read?

The inode Map

- (In-memory) table of inode disk addresses
  - Maps *uid* to disk address of last inode for that *uid*
- Updated every time inode is written to disk



Using the inode Map

- Open() :
  - Get inode address from inode map
  - Read inode from disk into Active File Table
- Read() : as before
  - Get from cache
  - Or get from disk address in inode

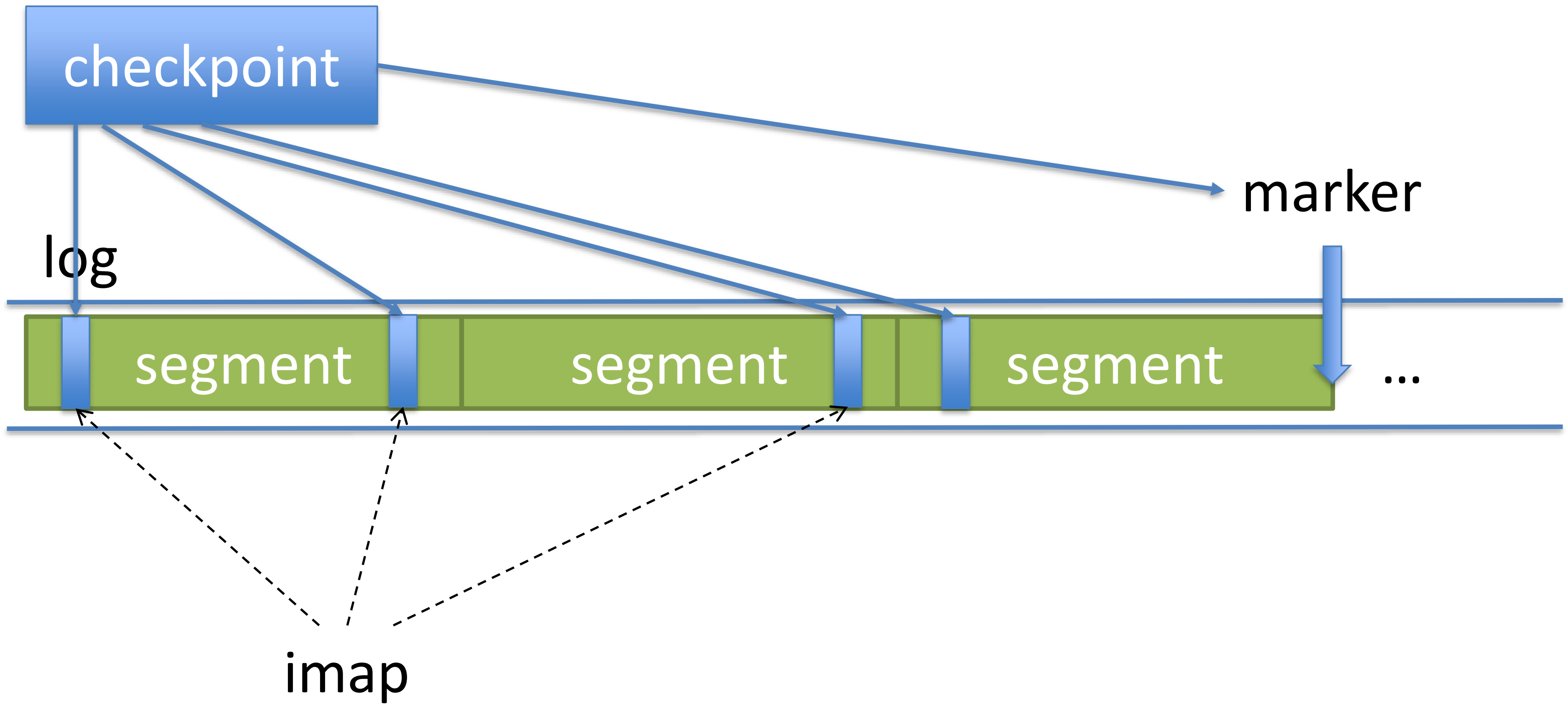
Using inode Map

- Reading seems more complicated
- Because indirection through inode map
- But performance is determined by disk reads
- So little difference

Get the inode Map to Disk

- inode map (imap) needs to be persisted!
- Where to put it?
  - Mixed with data and inodes: to avoid seeks
- How to find iMap?
- Checkpoint region:
  - Fixed location in disk
  - Contains addresses of imap
  - Contains current log head position
  - Updated periodically

LFS Log + Checkpoint



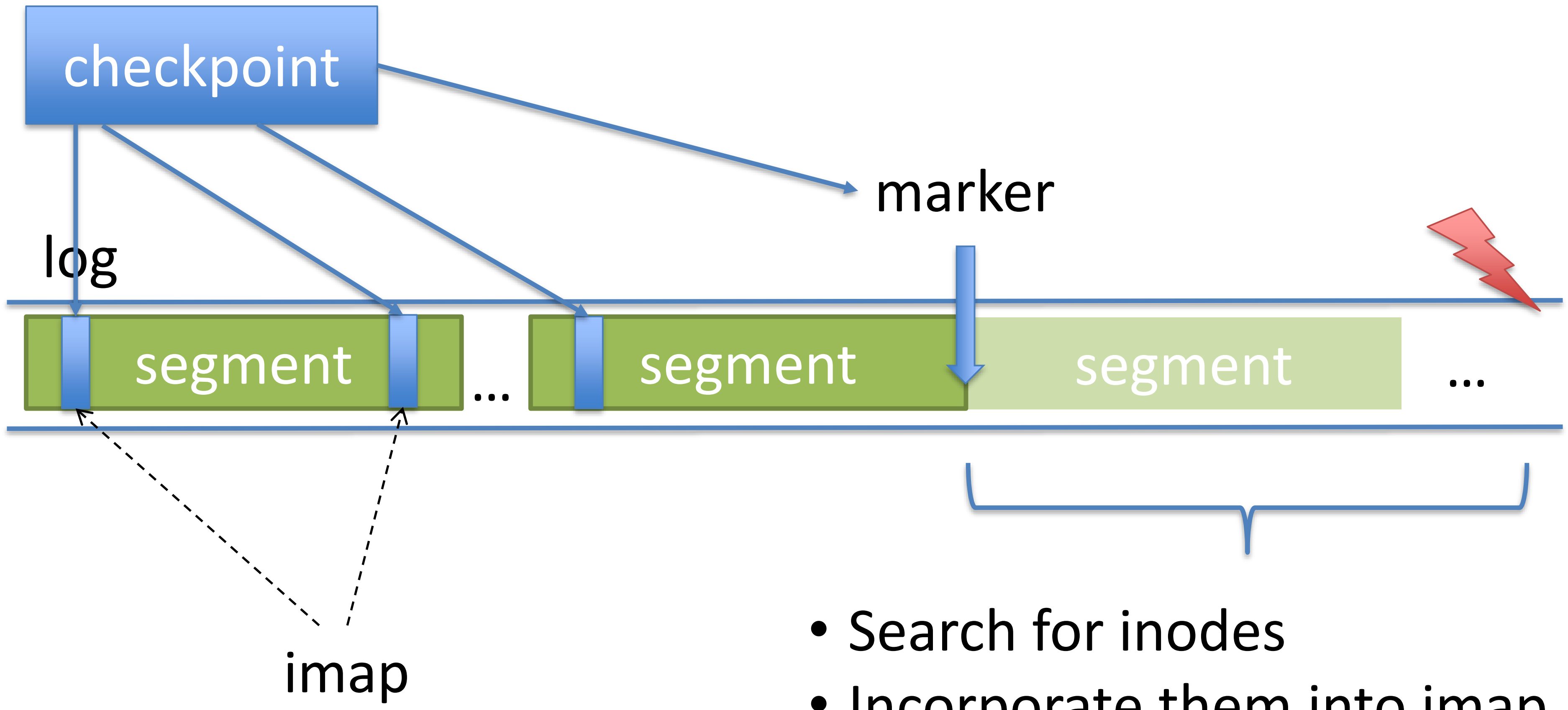
Crash?

- Start from inode map in checkpoint
  - Contains addresses of all inodes written *before* last checkpoint
- How to find inodes?
  - That were in in-memory inode map before crash
  - But not written in the checkpoint

Roll Forward

- Remember: checkpoint put marker in log
- From marker forward
  - Scan for inodes in the log
  - Add their addresses to inode map
- Result: All inode addresses not in inode map before crash are in inode map afterwards

LFS Roll forward



- Search for inodes
- Incorporate them into imap

Time Interval between Checkpoints

- Too short: lots of disk I/O to write checkpoints
- Too long: long recovery time (forward scan)
- Compromise
  - Crashes are rare
  - So recovery seldom happens
  - Can tolerate longer recovery time

An Aside: A General Rule

- Tradeoff between
  - Failure-free performance
  - Recovery time

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

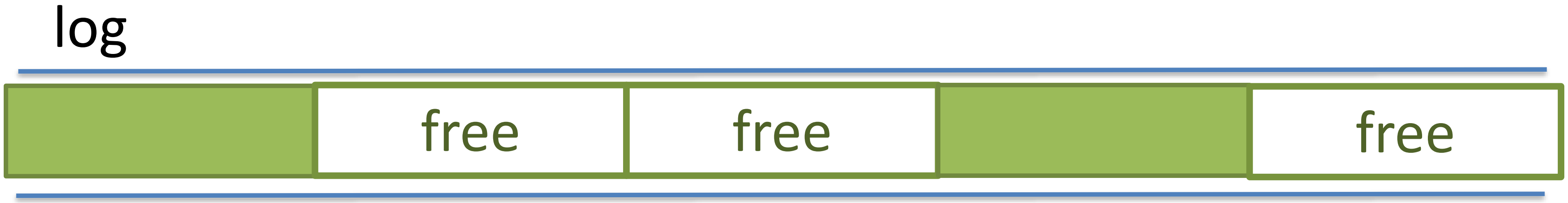
- Done one segment at a time

- Determine which blocks are new
- Write them into buffer
- If buffer is full, write new segment
- Cleaned segment is marked free

How Cleaning is Done - Log

- Log is more complicated than simple linear log
- Log = sequence of segments
  - Some in use
  - Some free

LFS Log



Write()

- Rather than append to log
- Write to free segment in log

- Segments are large (Mbs)
- Still get benefits from sequential access

How to?

- Determine that a block is old or new?

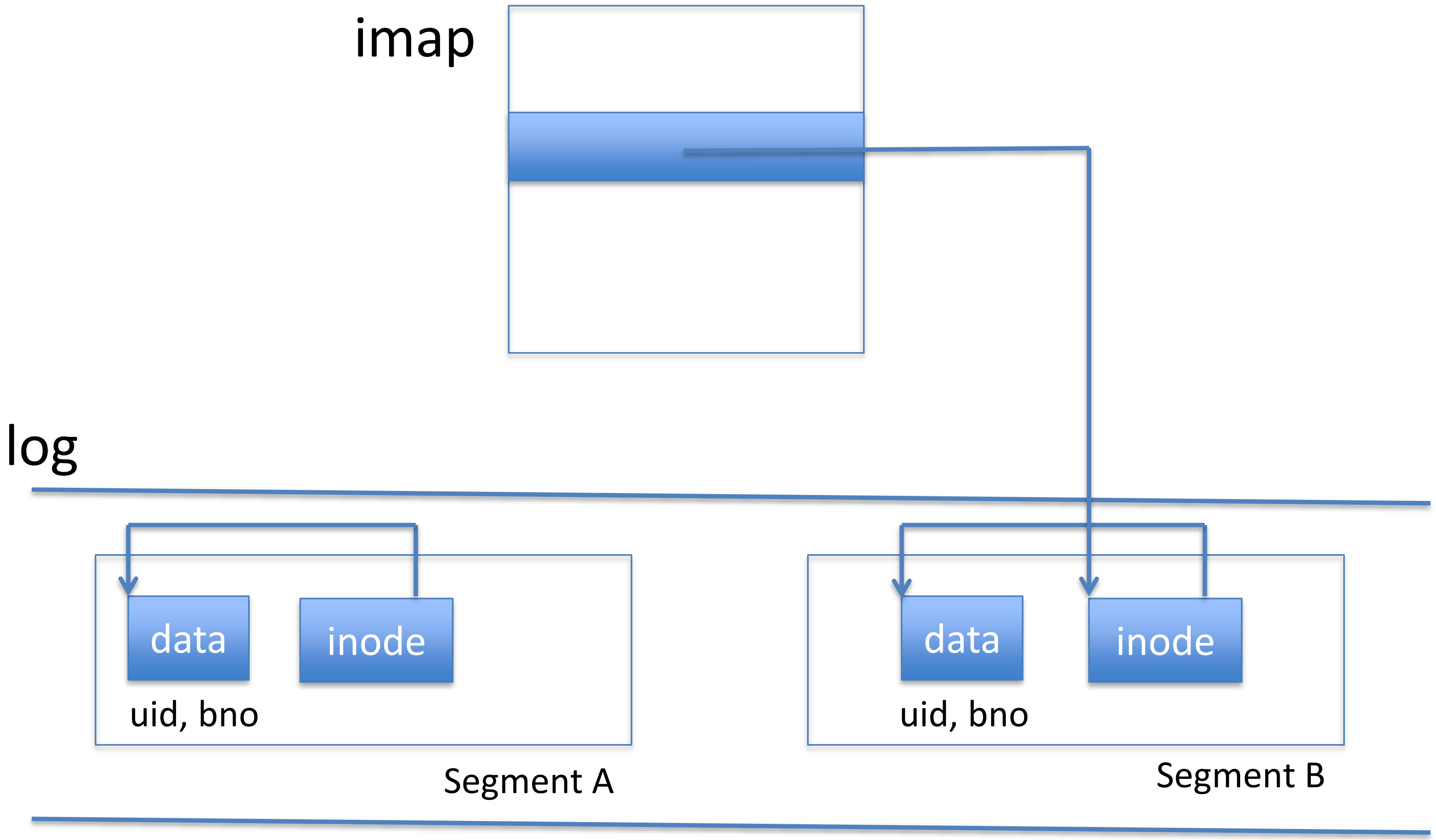
Change Write() a Bit

- Instead of just writing to buffer (and to log)
  - Data
- Write the following
  - Data + uid + blockno

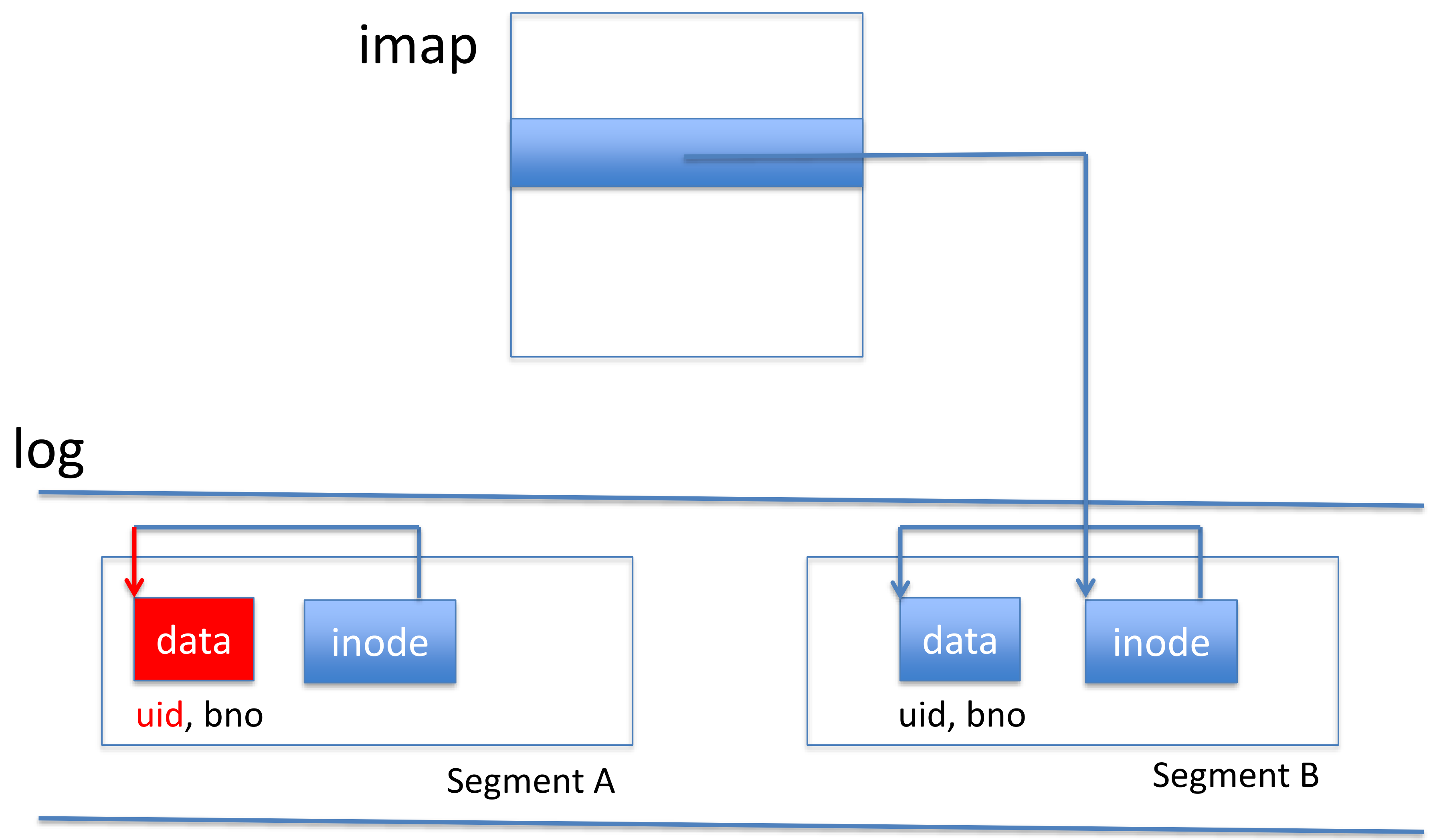
Determining a Block is Old

- For a data block
- Take its disk address
- Take its uid and block no
- Look in inode map and then in inode
- If inode has different disk address → old

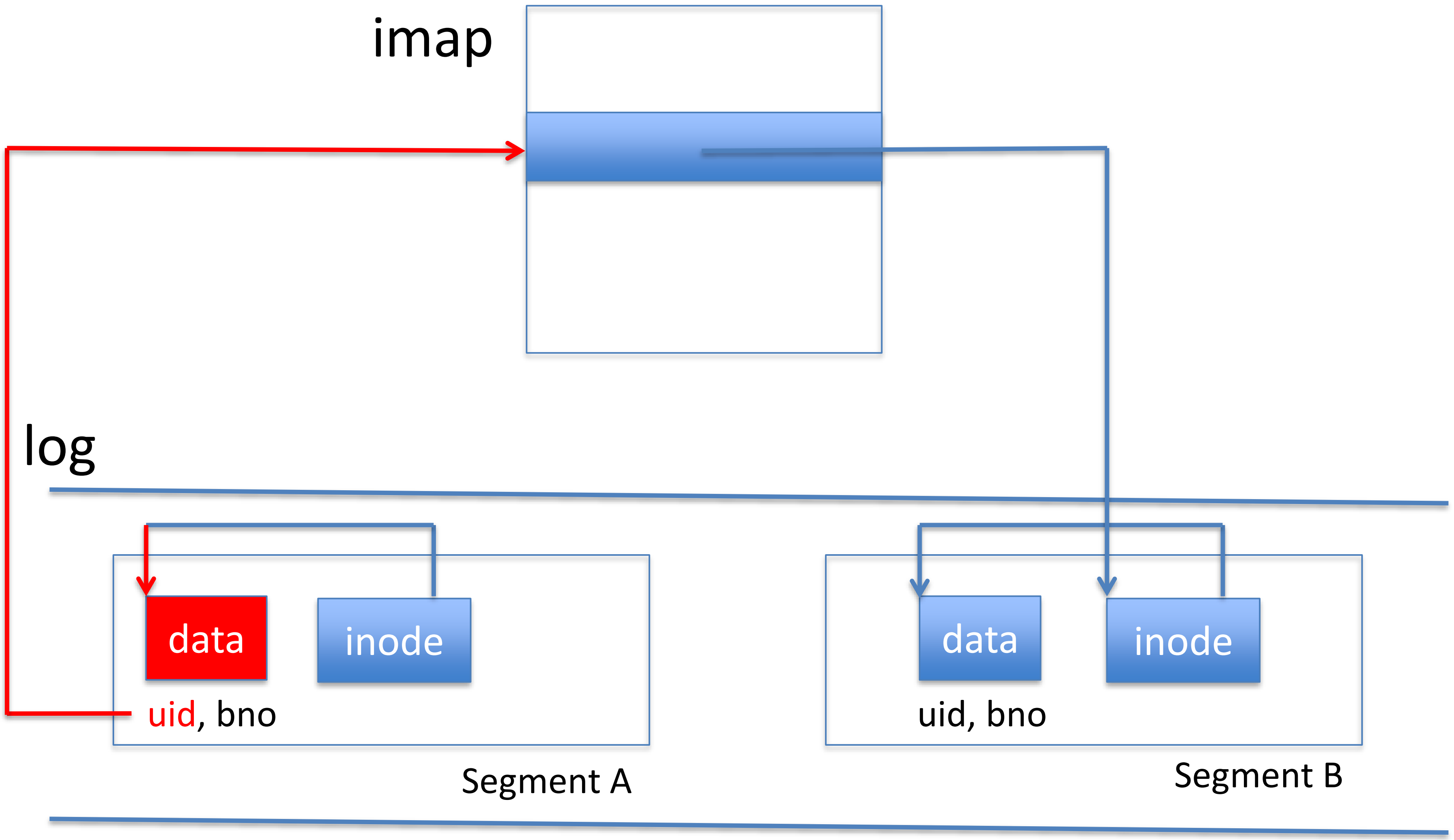
Determining if a Block is Old



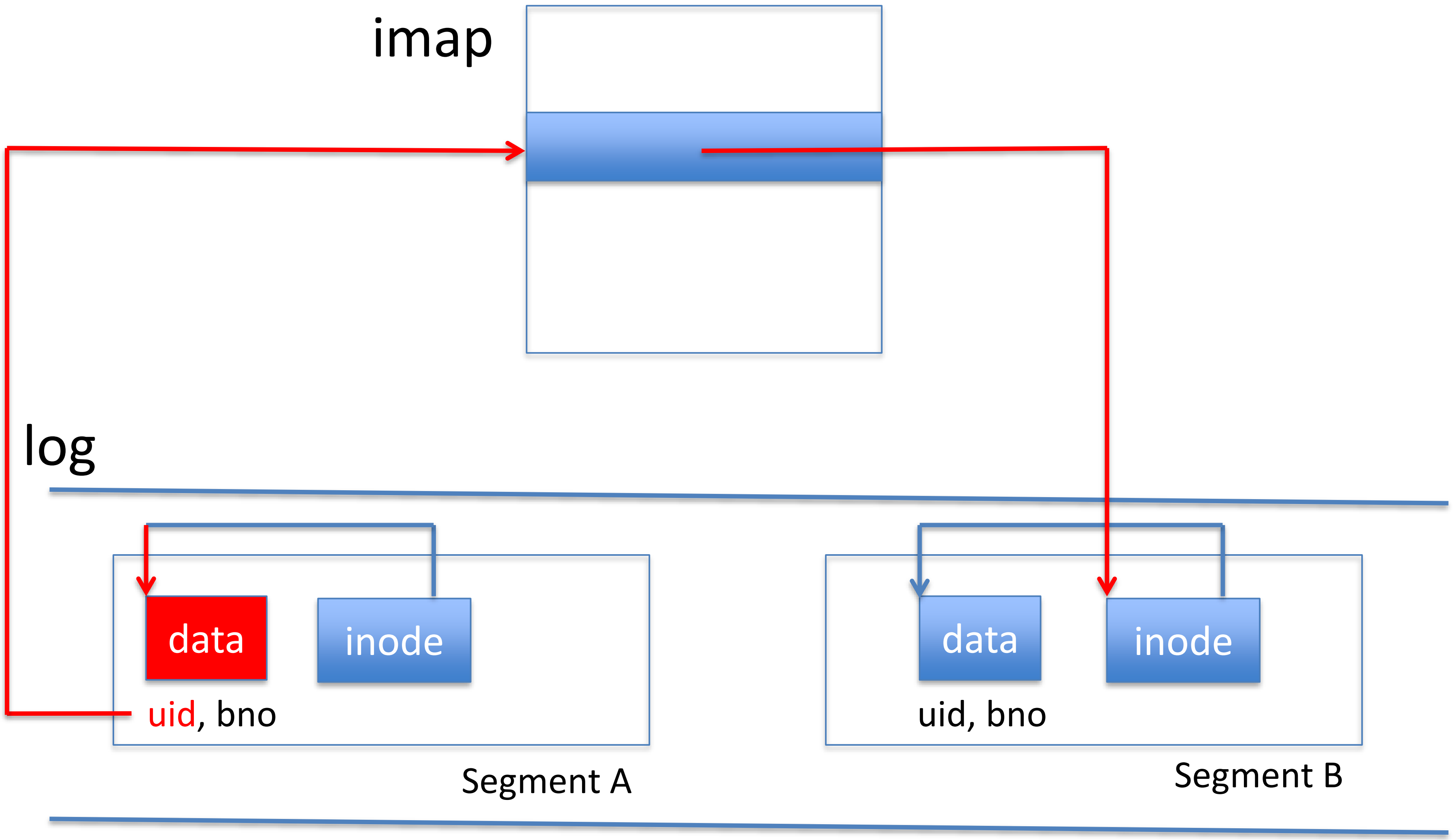
Cleaning Segment A



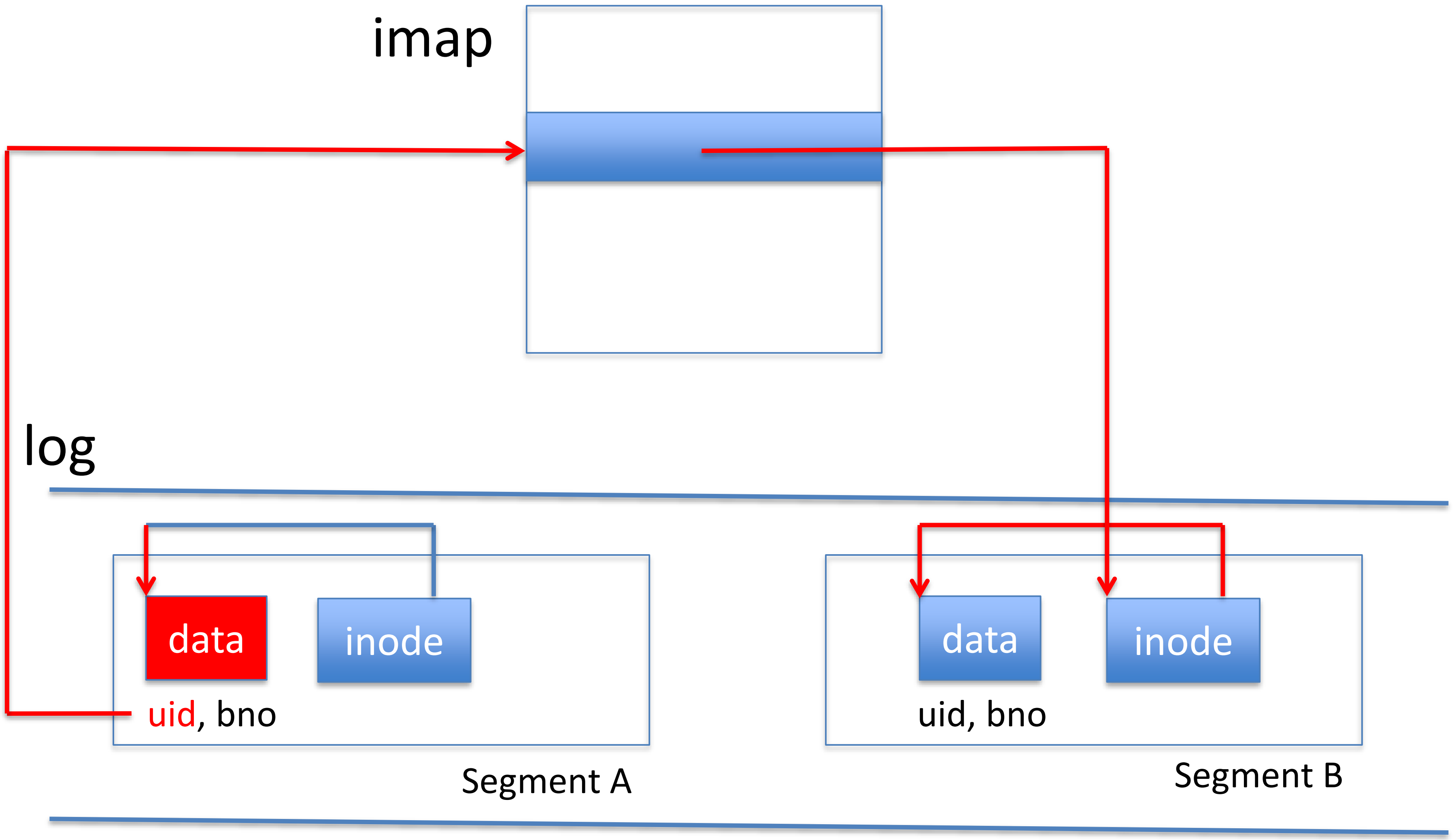
Cleaning Segment A



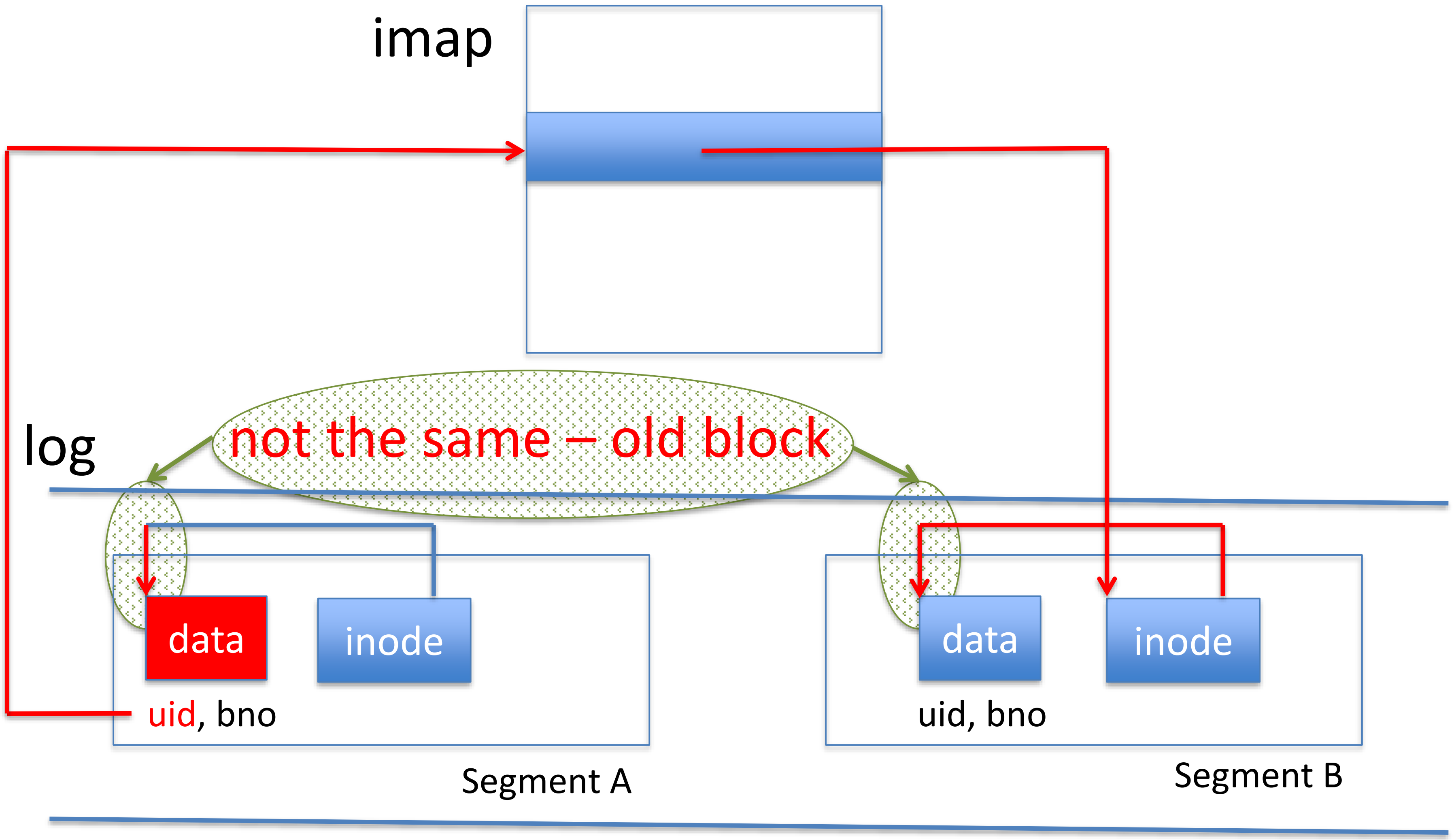
Cleaning Segment A



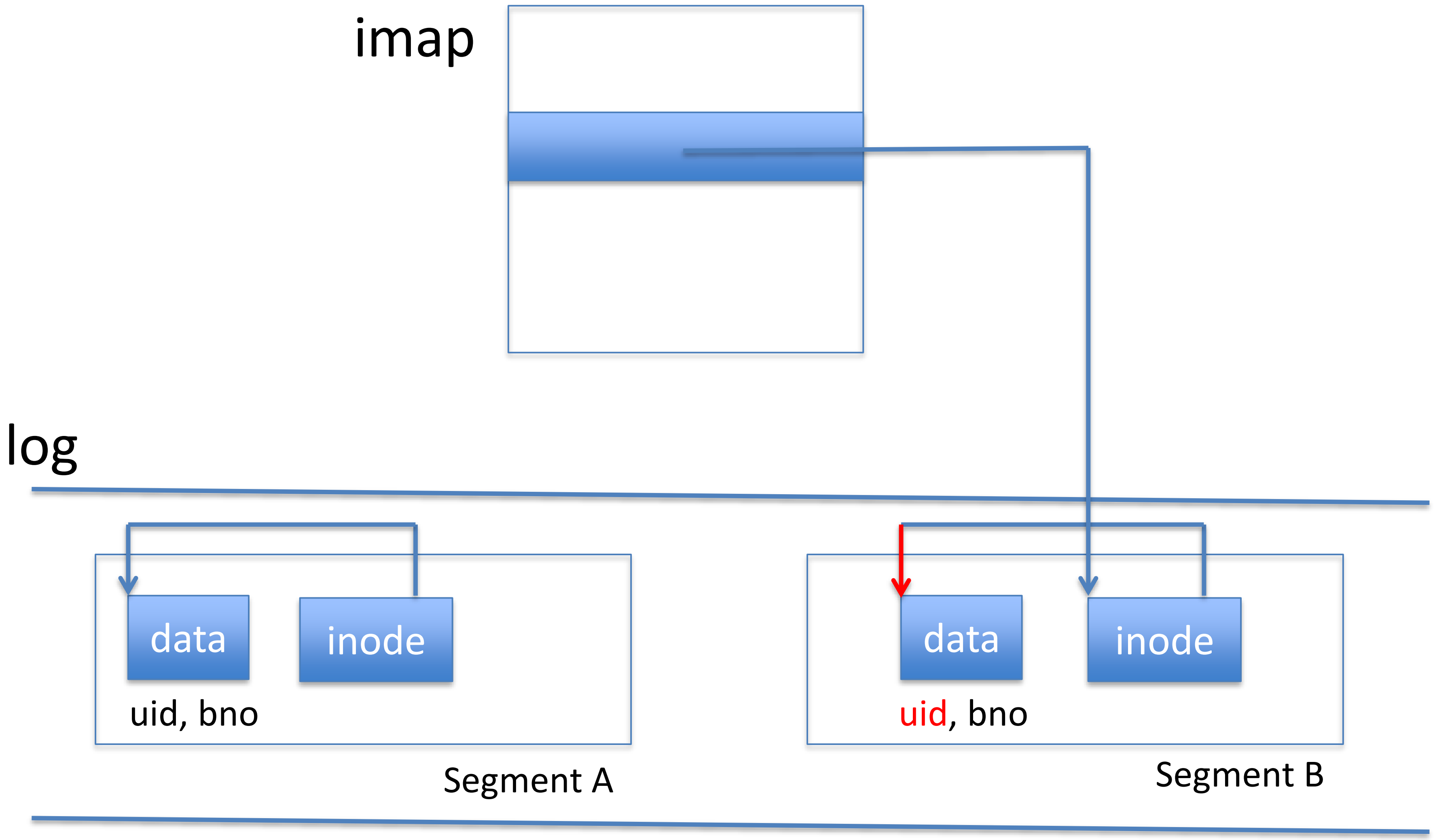
Cleaning Segment A



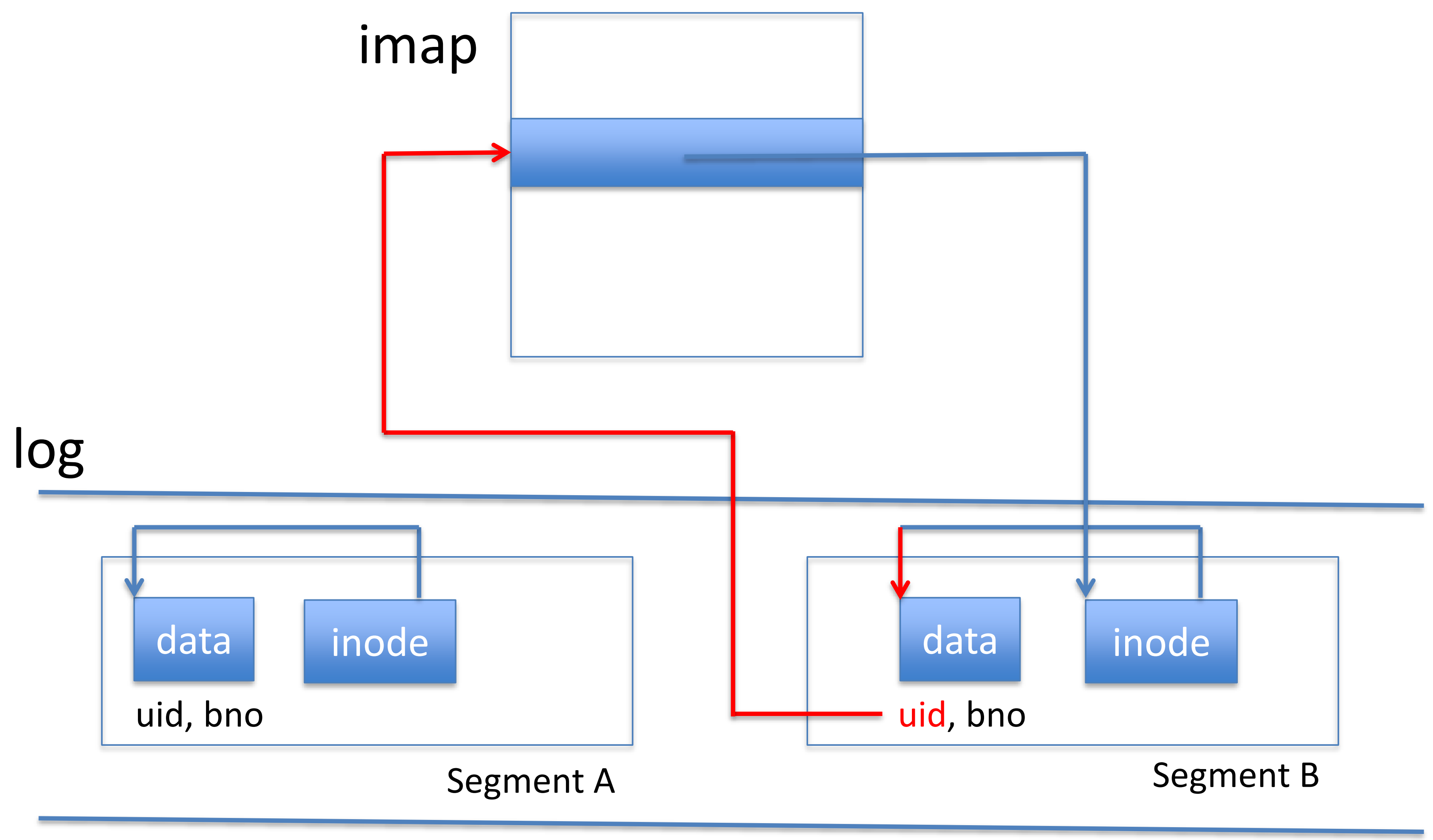
Cleaning Segment A



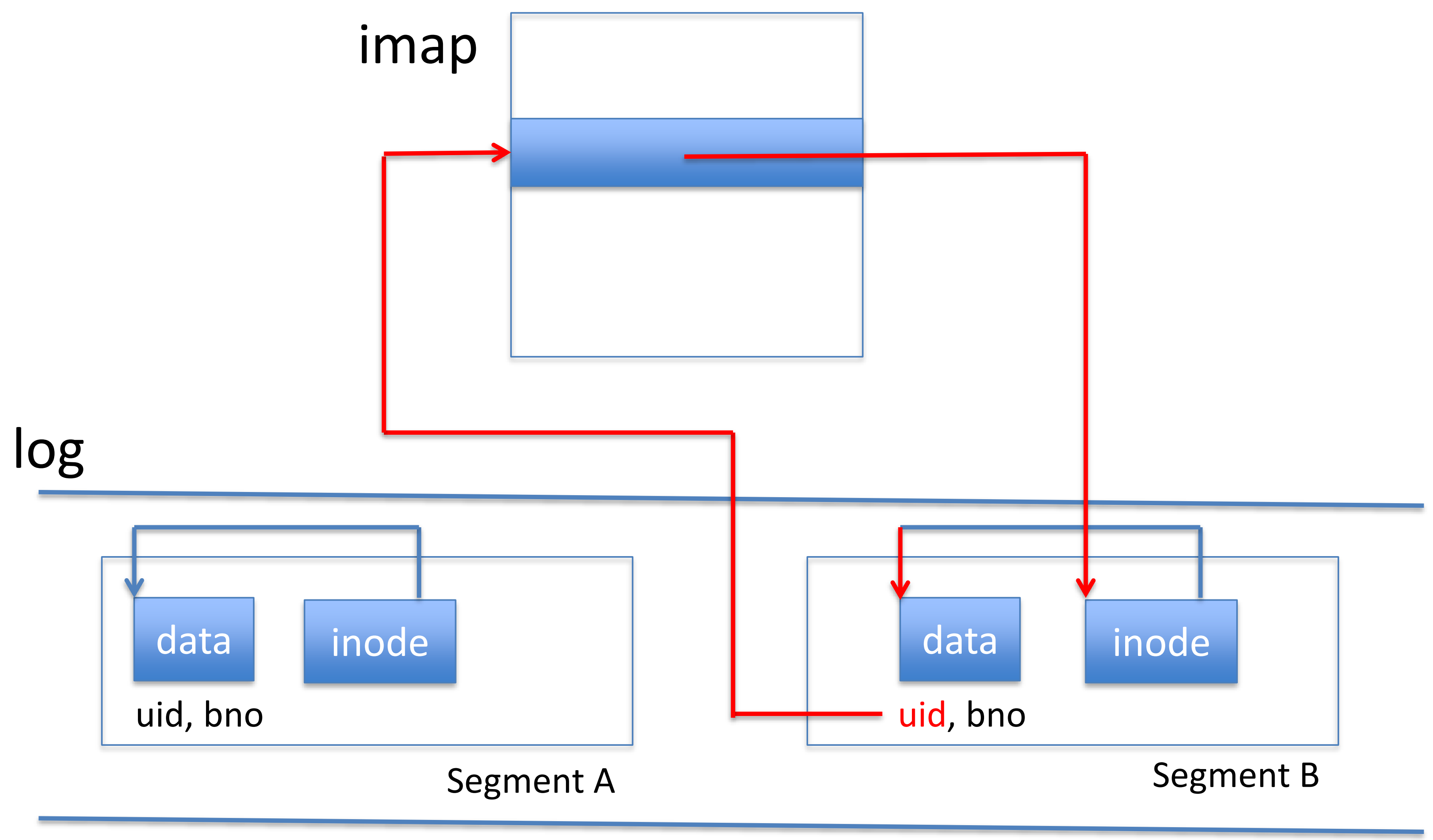
Cleaning Segment B



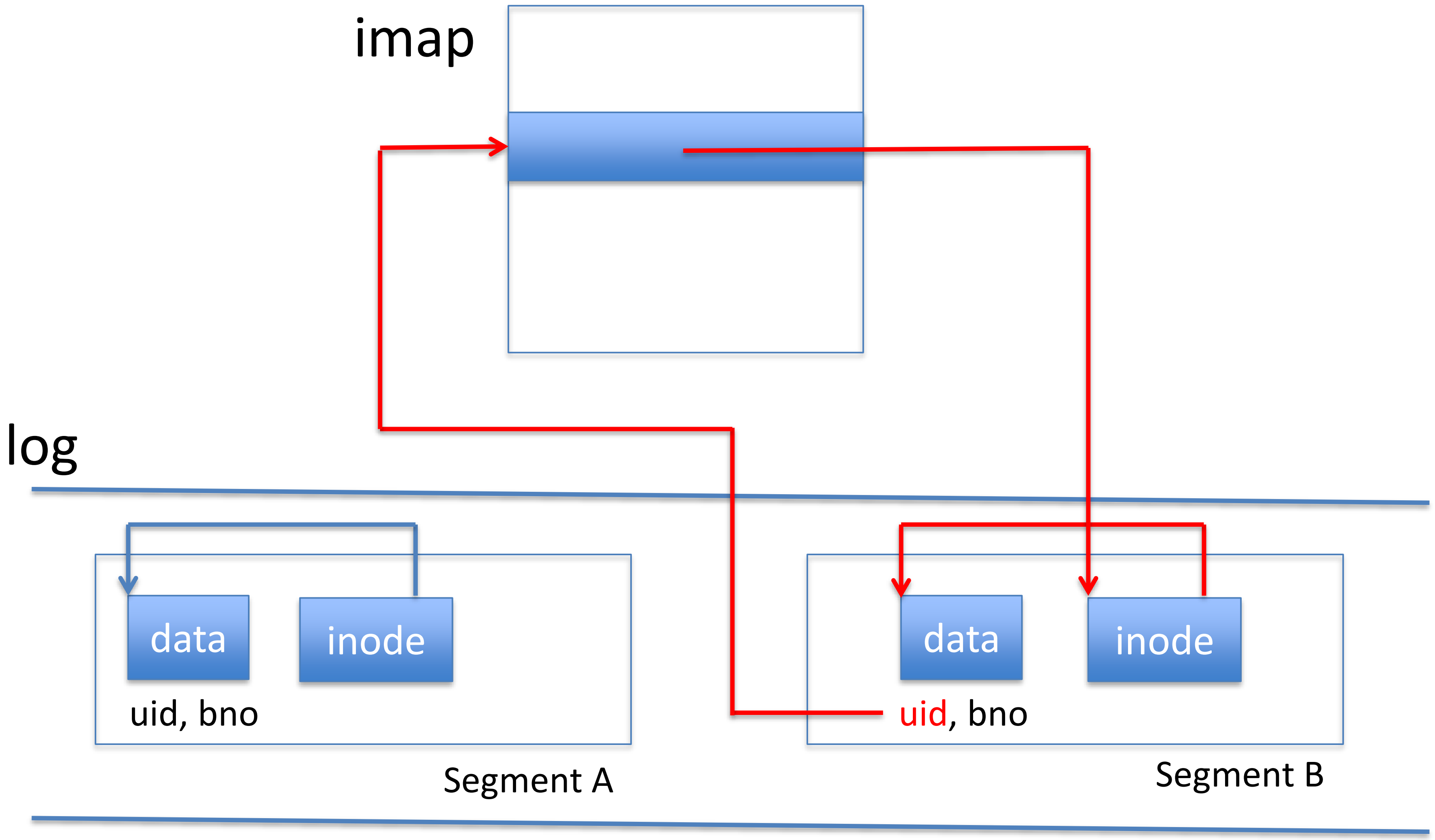
Cleaning Segment B



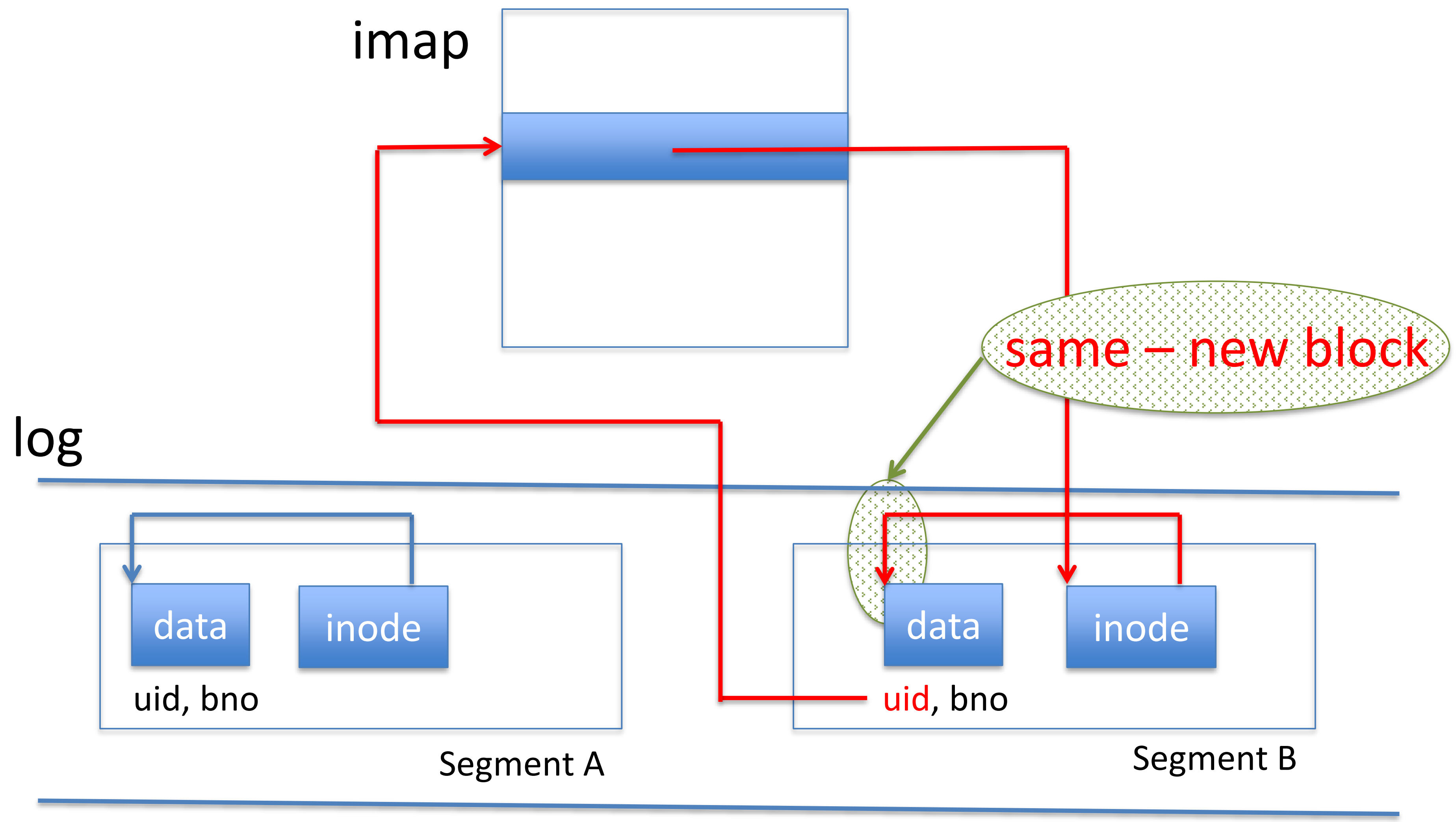
Cleaning Segment B



Cleaning Segment B



Cleaning Segment B



Determining a Block is Old

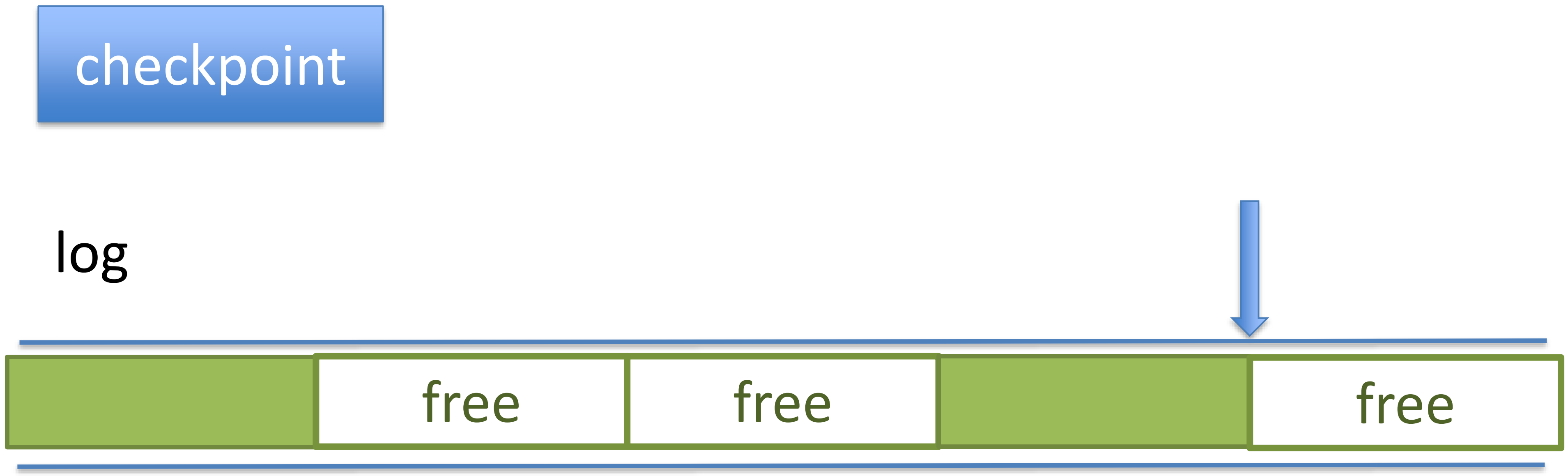
- For a data block
- Take its disk address
- Take its uid and block no
- Look in inode map and then in inode
- If inode has different disk address → old

Putting it all Together

Key Idea in LFS

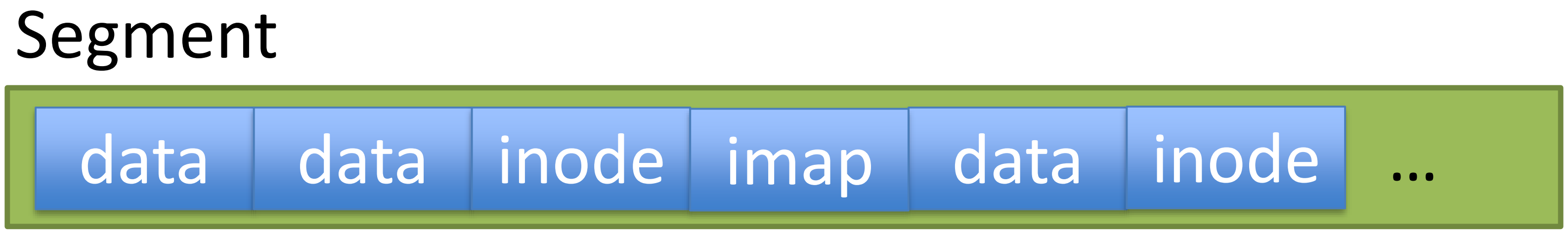
- “All” writes go to log, including
  - Data
  - inode
- “All” = All except for checkpoints

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



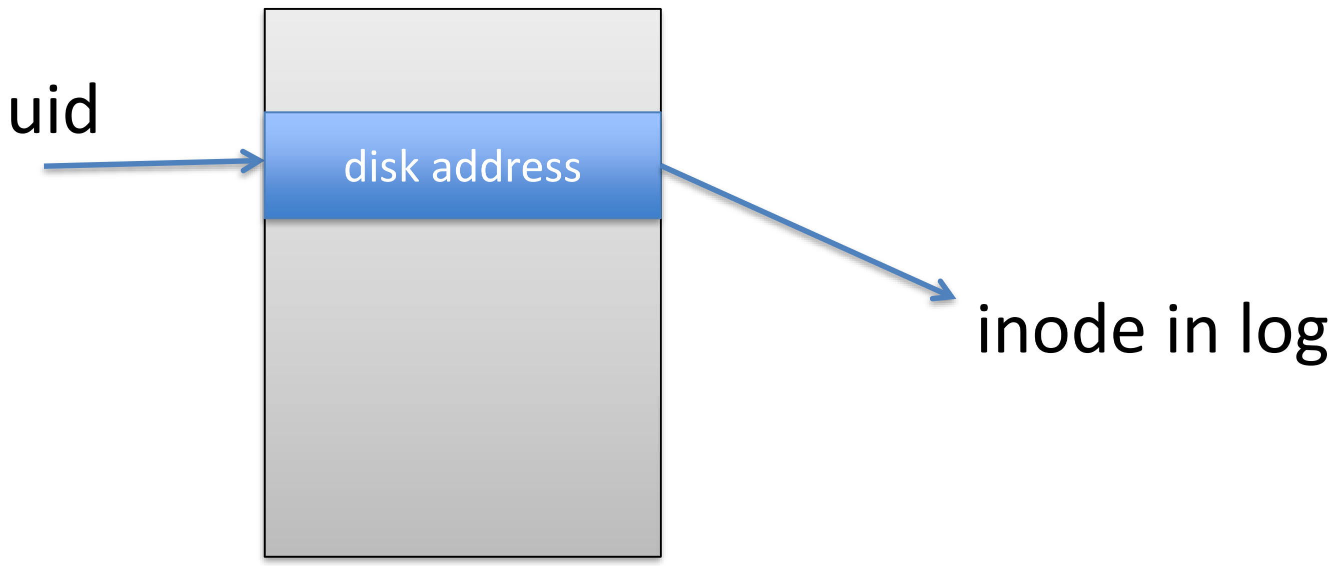
data: modified user data sector (includes uid and block no)
inode: modified inode sector

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
- inode_map[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

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
- Similar ideas in some commercial systems