Week 12

Advanced FS: Log-Structured Designs

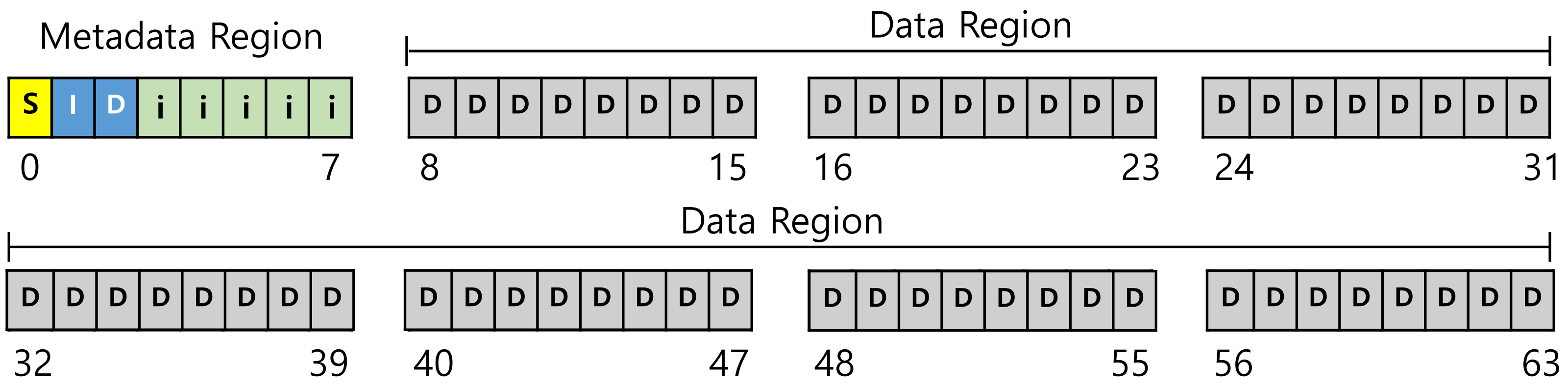
Oana Balmau
March 21, 2023

Class Admin

Office hours on Thursday moved from 2-3pm

	Graded Exercises Due C Review: Complex structs				
Week 12 **File Systems**	mar 20 **Graded Exercises Due** **C Review: Pointers & Memory Allocation II**	mar 21 **Advanced File System Implementation (1/2)**	mar 22	mar 23 **Advanced File System Implementation (2/2)** • Grades released for Scheduling Assignment	mar 24
Week 13 **File Systems**	mar 27 C Review: Advanced debugging	mar 28 **Handling Crashes & Performance (1/2)** **Optional reading: OSTEP Chapters 38, 43**	mar 29	mar 30 **Handling Crashes & Performance (2/2)** • ~~Grades released for Exercises Sheet~~ • Practice Exercises Sheet: File Systems	mar 31
Week 14 **Advanced Topics**	apr 3 No lab. Work on Assignment 3 ~~**Memory Management Assignment Due**~~	apr 4 **Advanced topics: Virtualization**	apr 5	apr 6 **Advanced topics: Operating Systems Research** (Invited Speaker: TBD) Grades released for Exercises Sheet	apr 7
Week 15 **Wrap-up**	apr 10 No Lab. Prepare for end-of-semester. **Memory Management Assignment Due**	apr 11 **End-of-semester Q&A– not recorded**	apr 12	apr 13 **End-of-semester Q&A — not recorded.** Last class!	apr 14 Grades released for Memory Management Assignment

Review W11: Disk Data Structures for FS



Review W11: File System Implementation

Key aspects of the system:

1. **Data structures**
   - On disk
   - In memory ← In memory data structures are used to make I/O more efficient
2. **Access methods**
   - Create(), open(), read(), write(), close()

Review W11: In-Memory Data Structures

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

Key Concepts

- The Log-Structured File System
- The Log-Structured Key-Value Store

Dealing With Crashes

Consider this Piece of Code

```
1. fd = Open( file )
2. Write( fd, 0 )
3. Write( fd, 1 )
4. Write( fd, 2 )
5. Write( fd, 3 )
6. Close( fd )
```

Machine Crash 1

```
1. fd = Open( file )
2. Write( fd, 0 )
3. Write( fd, 1 )
4. Write( fd, 2 )
5. Write( fd, 3 )
6. Close( fd )
```

crash

Not really a problem (old file is there)

Machine Crash 2

```
1. fd = Open( file )
2. Write( fd, 0 )
3. Write( fd, 1 )
4. Write( fd, 2 )
5. Write( fd, 3 )
6. Close( fd )
```

crash

Not really a problem (new file is there)

Machine Crash 3

```
1. fd = Open( file )
2. Write( fd, 0 )
3. Write( fd, 1 )
4. Write( fd, 2 )
5. Write( fd, 3 )
6. Close( fd )
```

crash

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

With Write-Behind

```
1. fd = Open( file )
2. Write( fd, 0 )
3. Write( fd, 1 )
4. Write( fd, 2 )
5. Write( fd, 3 )
6. 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 or
- No updates are on disk
- **Nothing in-between!**

Atomicity is Important

```
1. Read( balance_savings )
2. balance_savings -= 100
3. Write( balance_savings )
4. Read( balance_checking )
5. balance_checking += 100
6. Write( balance_checking )
```

Atomicity is Important

```
1. Read( balance_savings )
2. balance_savings -= 100
3. Write( balance_savings )     ← crash
4. Read( balance_checking )
5. balance_checking += 100
6. Write( balance_checking )
```

Your 100 CADs 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

A single sector disk write is atomic

	Disk Sector
Before `WriteSector`	old
After `WriteSector` returns successfully	new

Assumption

A single sector disk write is atomic

	Disk Sector		
Before `WriteSector`	old		
After `WriteSector` returns successfully	new		
If failure	old	**or**	new

Assumption

A single sector disk write is atomic

	Disk Sector
Before `WriteSector`	old
After `WriteSector` returns successfully	new
If failure	old **or** new
Never:	new | old

Assumption True?

With very high probability (99.999+%): **YES**

- Disk vendors work very hard at this

If failure: old **or** new

Never: new old

How to Implement Atomicity?

How to Implement Atomicity?

- Approach 1: Shadow Paging
- Approach 2: Intentions Log

Approach 1: Shadow Paging

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

Approach 1: Shadow Paging

- 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()?

- Inode entry!
- Because it is smaller than sector

How Shadow Paging Works (with Write-Through)

Open()

- Read Inode into Active File Table

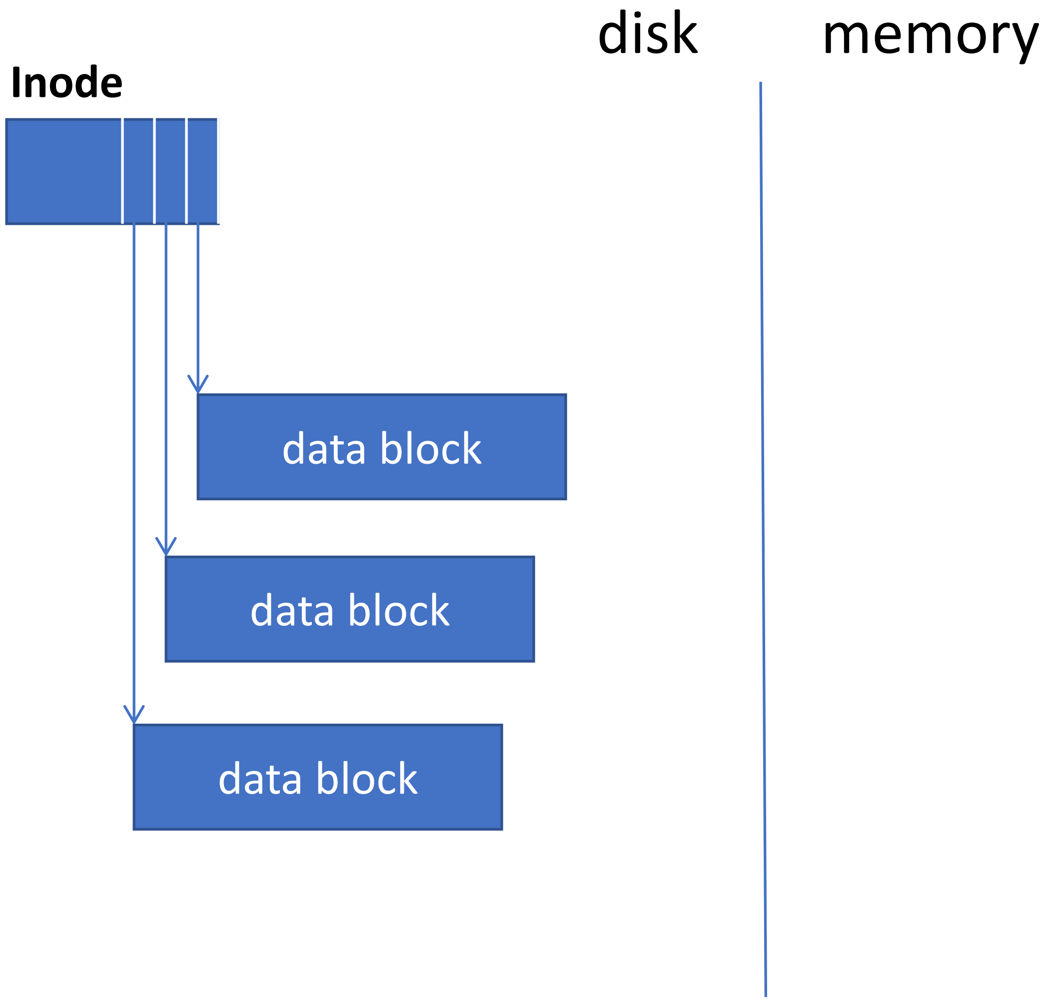
Write()s

- Allocate ***new*** blocks on disk for data
- Fill in address of new blocks in ***in-memory copy*** of Inode
- Write data blocks to cache and disk

Close()

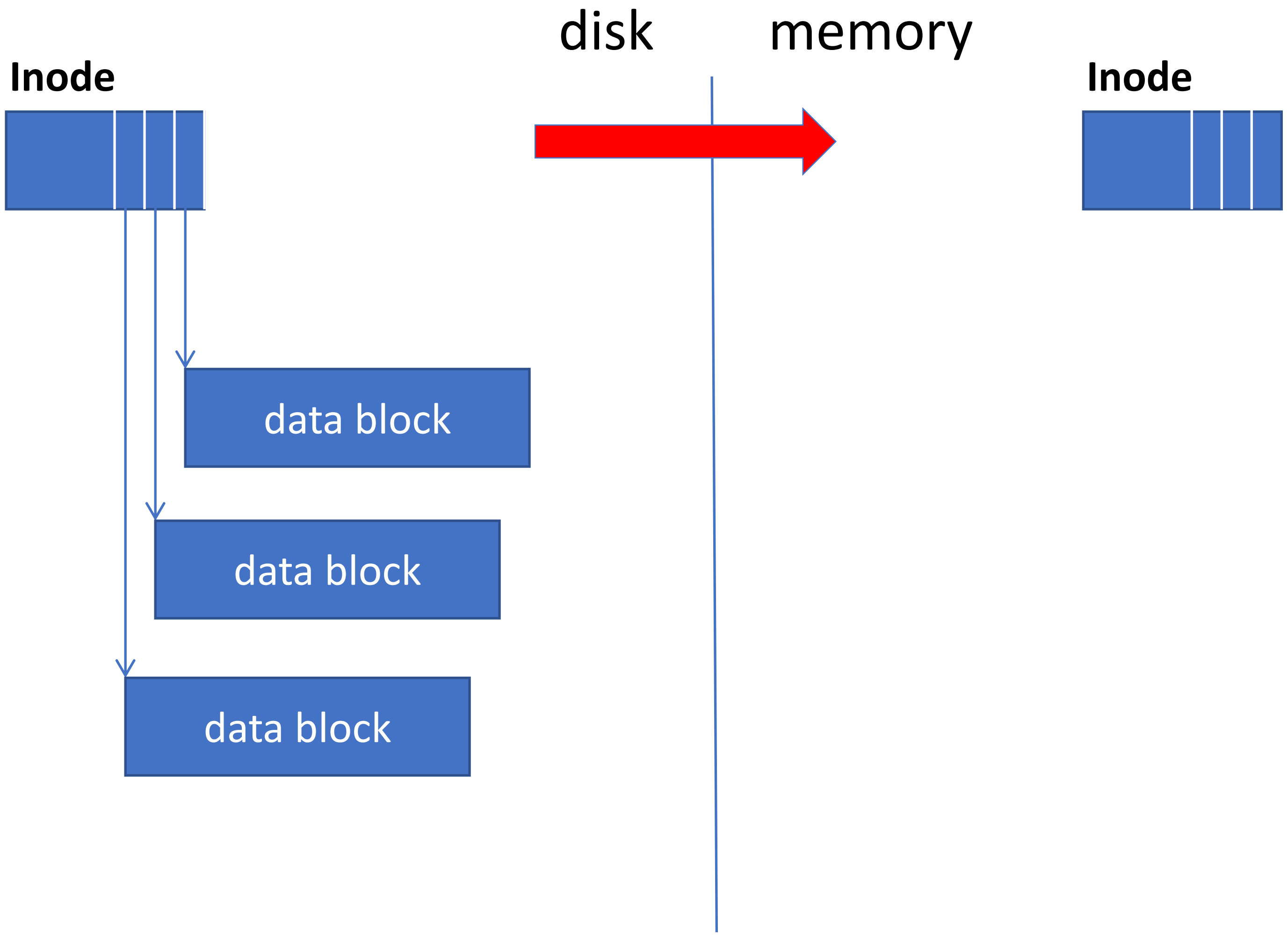
- Overwrite ***in-memory copy*** of Inode to ***disk*** Inode

Initial State



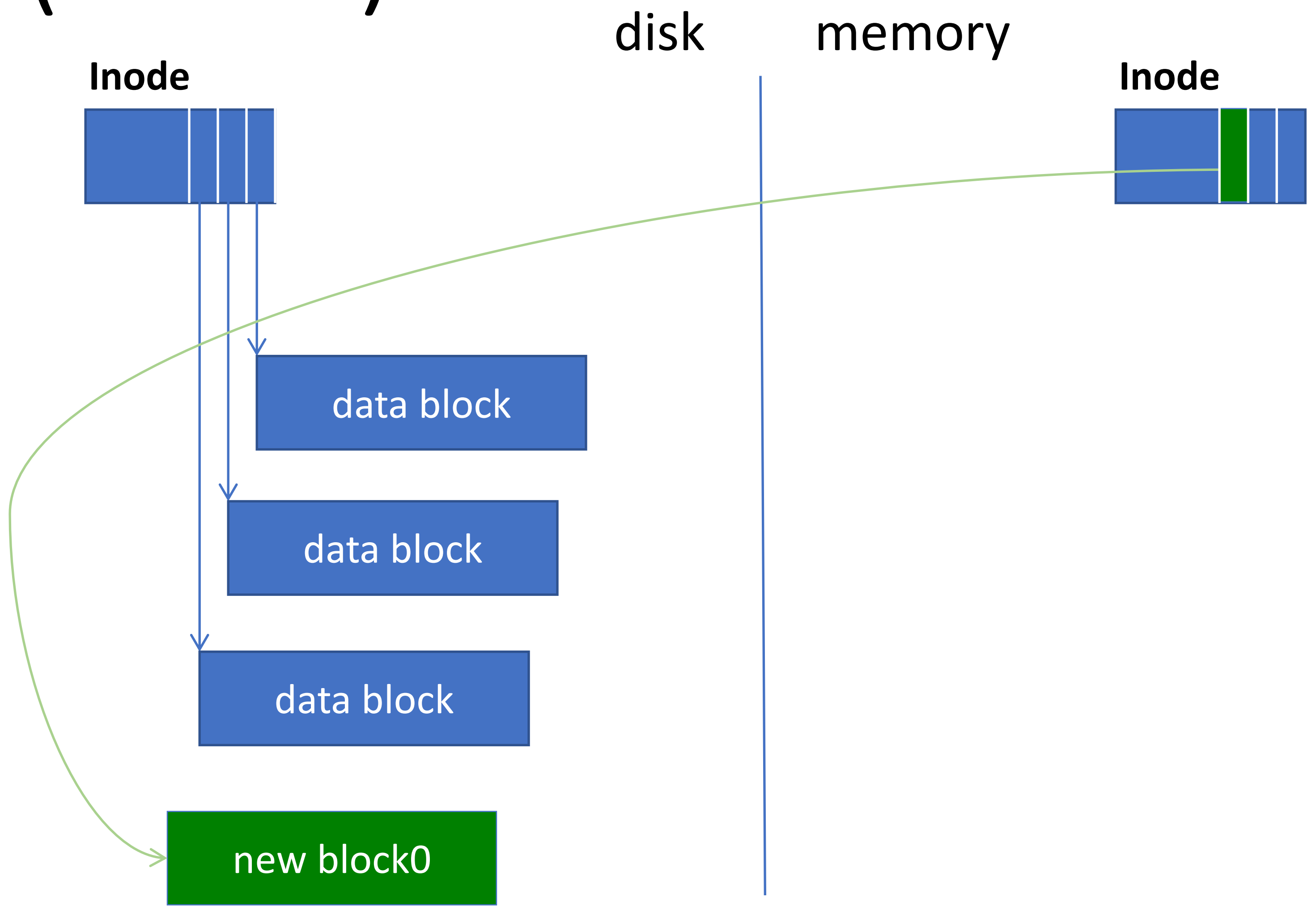
*With **write-through** cache*

Open()



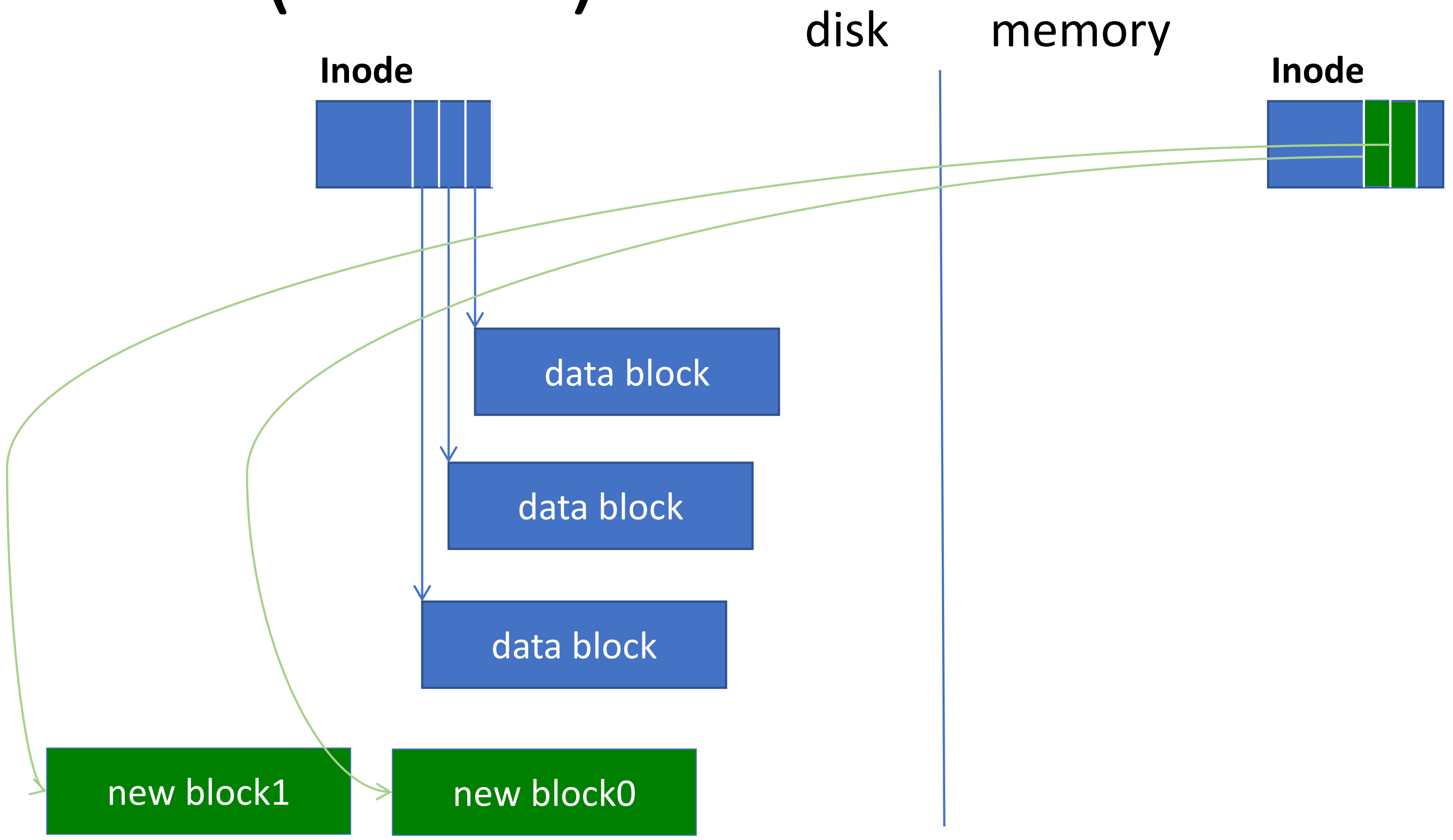
*With **write-through** cache*

Write(block 0)



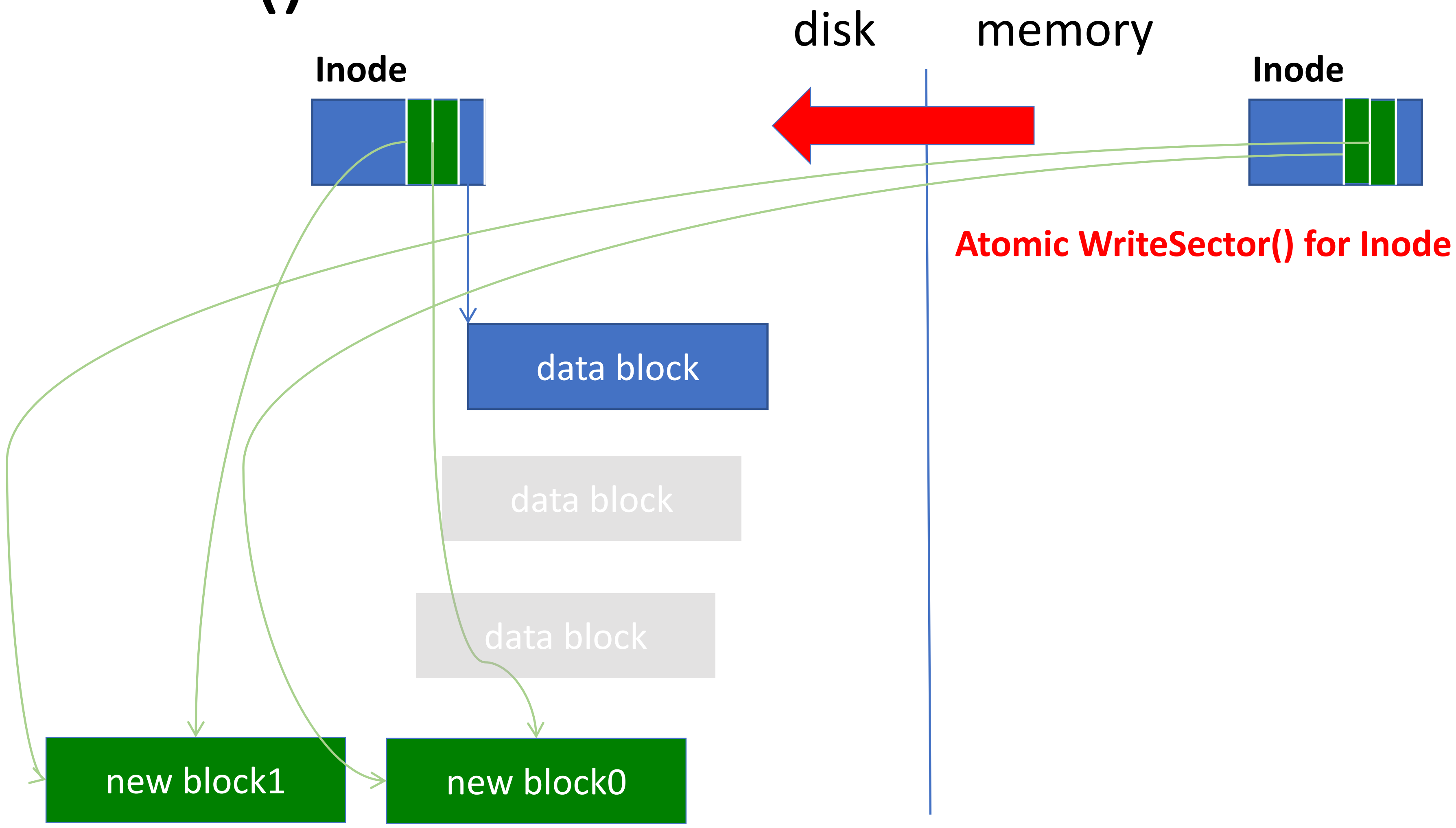
*With **write-through** cache*

Write(block 1)



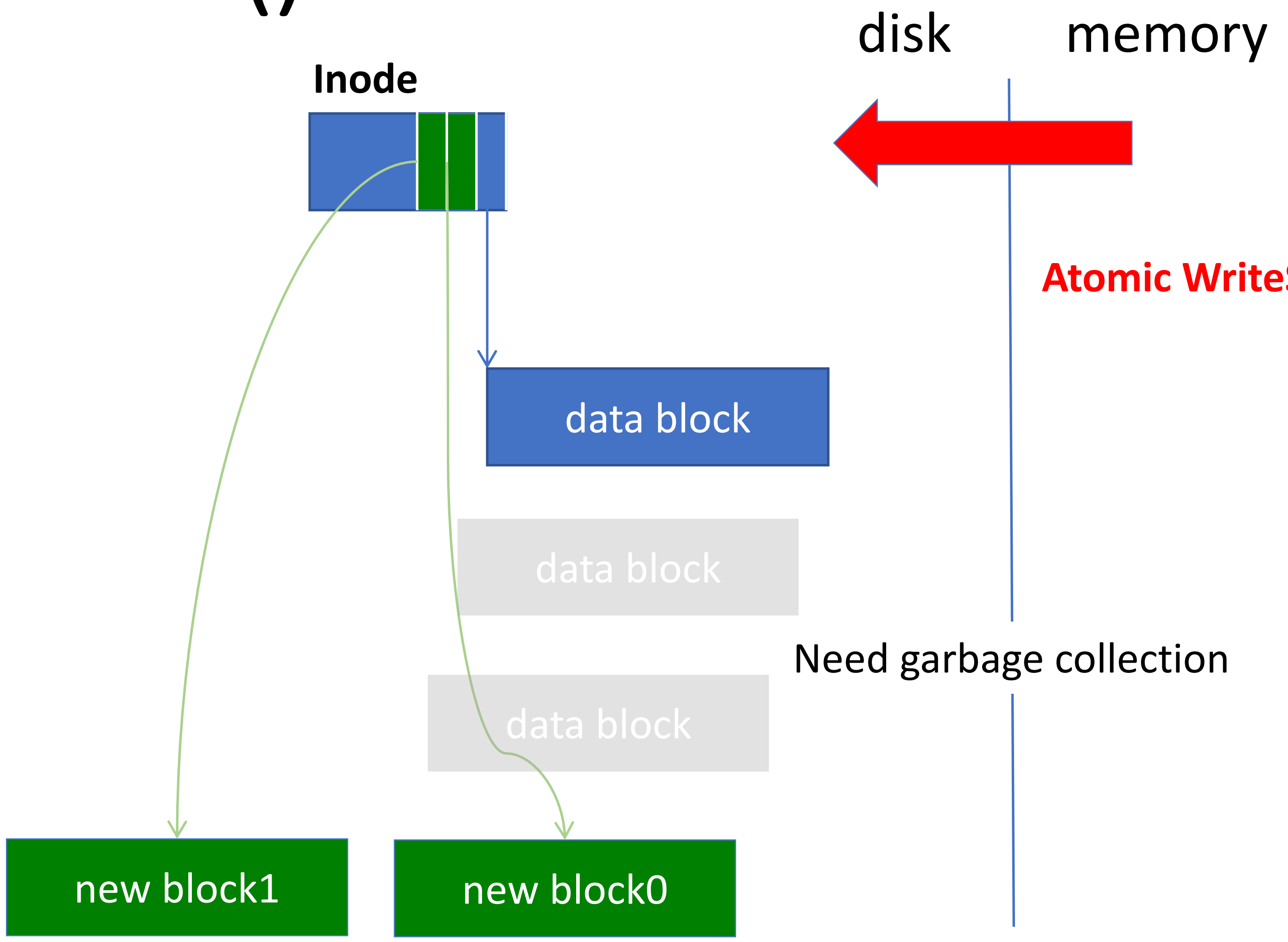
*With **write-through** cache*

Close()



*With **write-through** cache*

Close()



Atomic WriteSector() for Inode

Need garbage collection

*With **write-through** cache*

How Shadow Paging Works (with Write-Behind)

Open()

- Read Inode from disk into Active File Table

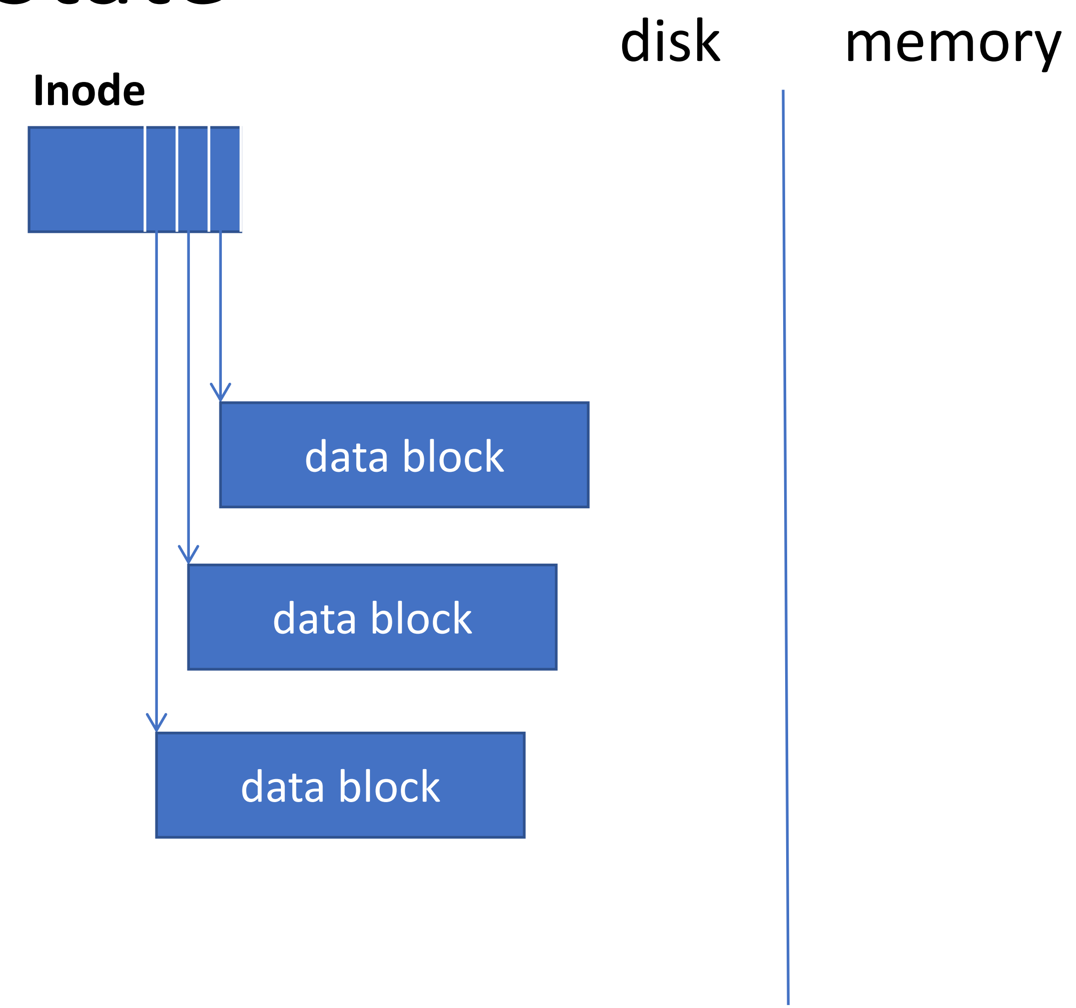
Write()s

- Allocate new blocks for new data
- Write disk addresses to in-memory copy of Inode
- Write data blocks to cache ***(don't write to disk yet)***

Close()

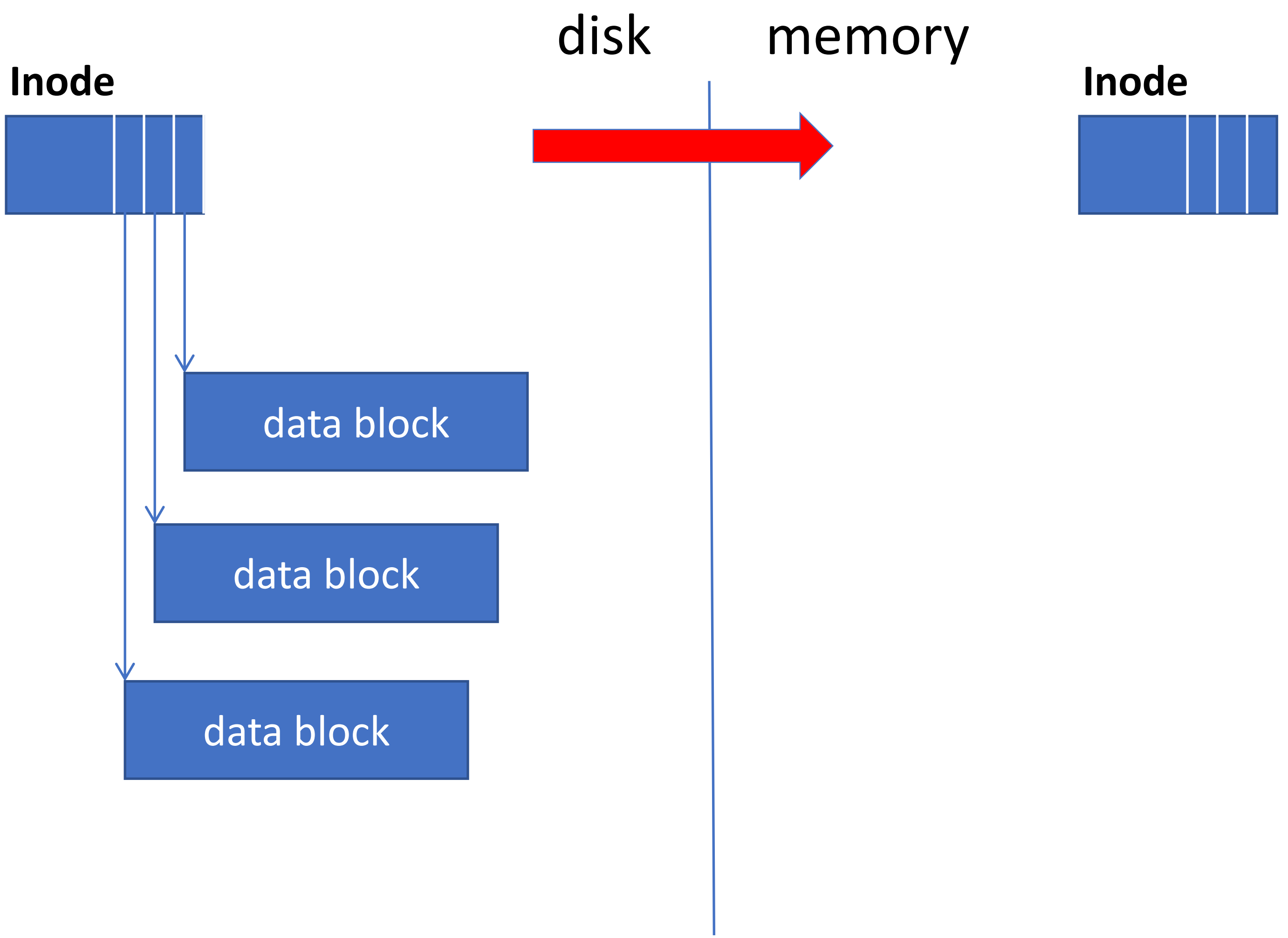
- *Write* ***all cached blocks to new disk blocks***
- Write in-memory Inode (containing all new block addresses) to disk

Initial State



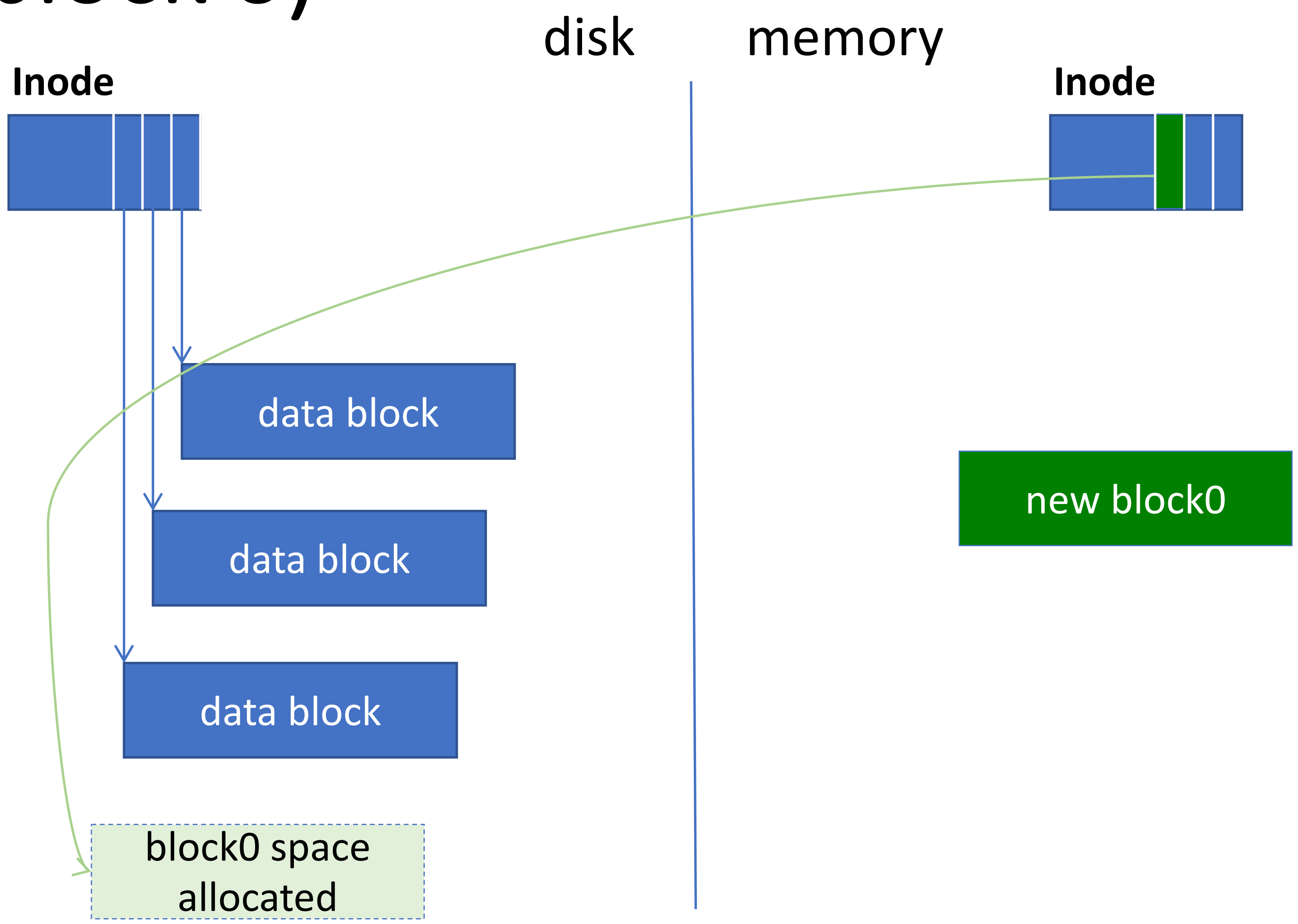
*With **write-behind** cache*

Open()



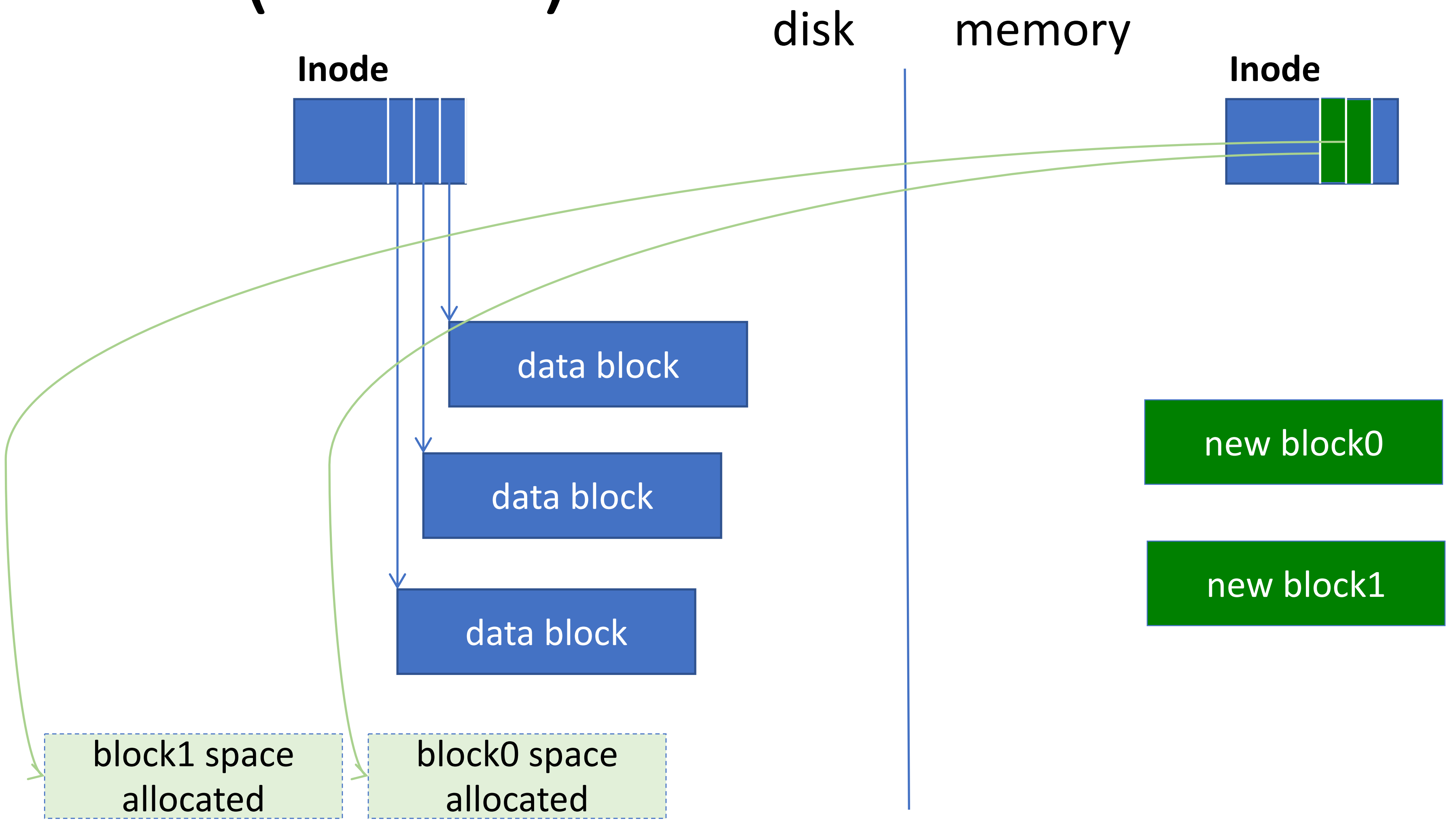
*With **write-behind** cache*

Write(block 0)



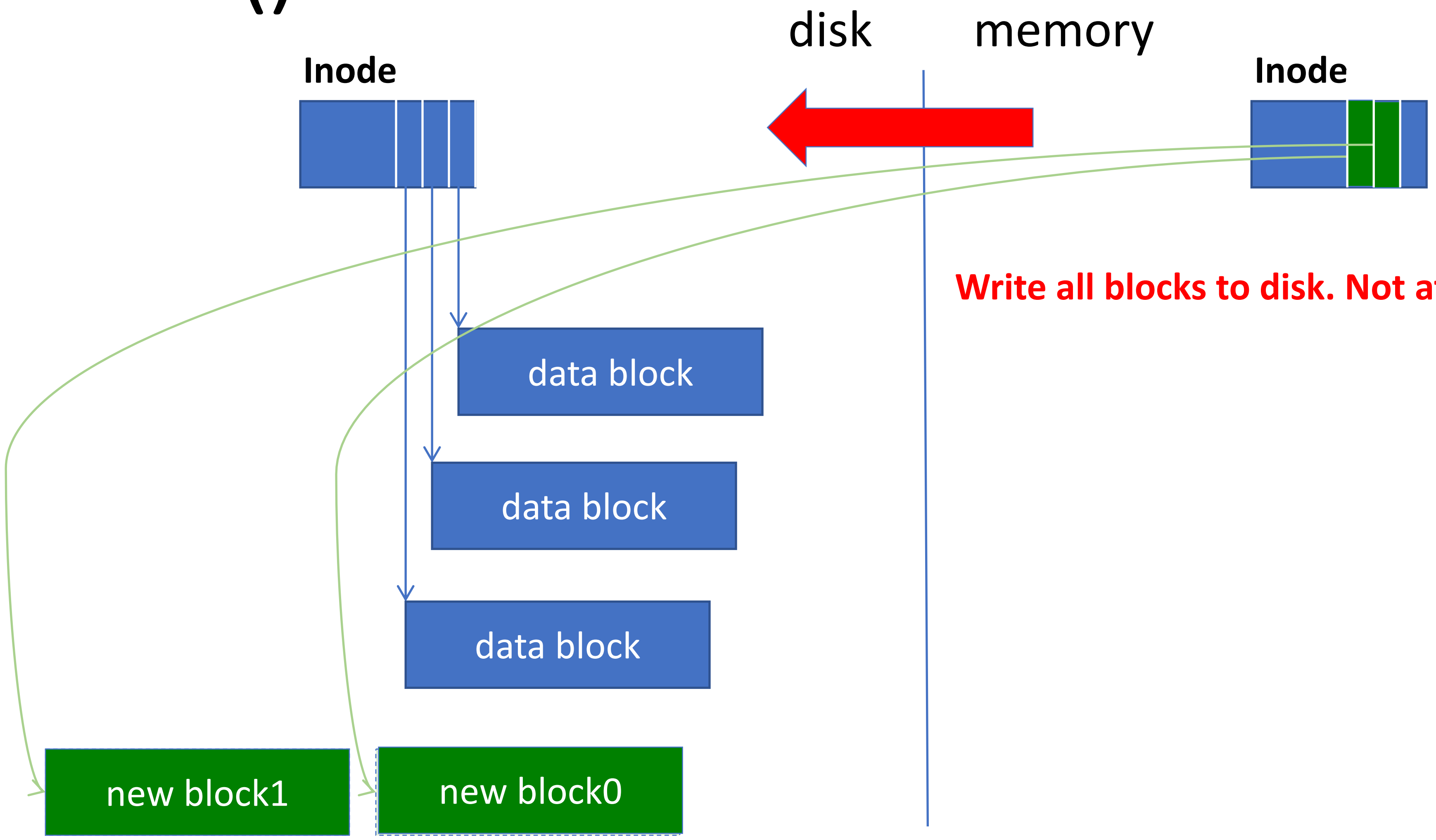
*With **write-behind** cache*

Write(block 1)



*With **write-behind** cache*

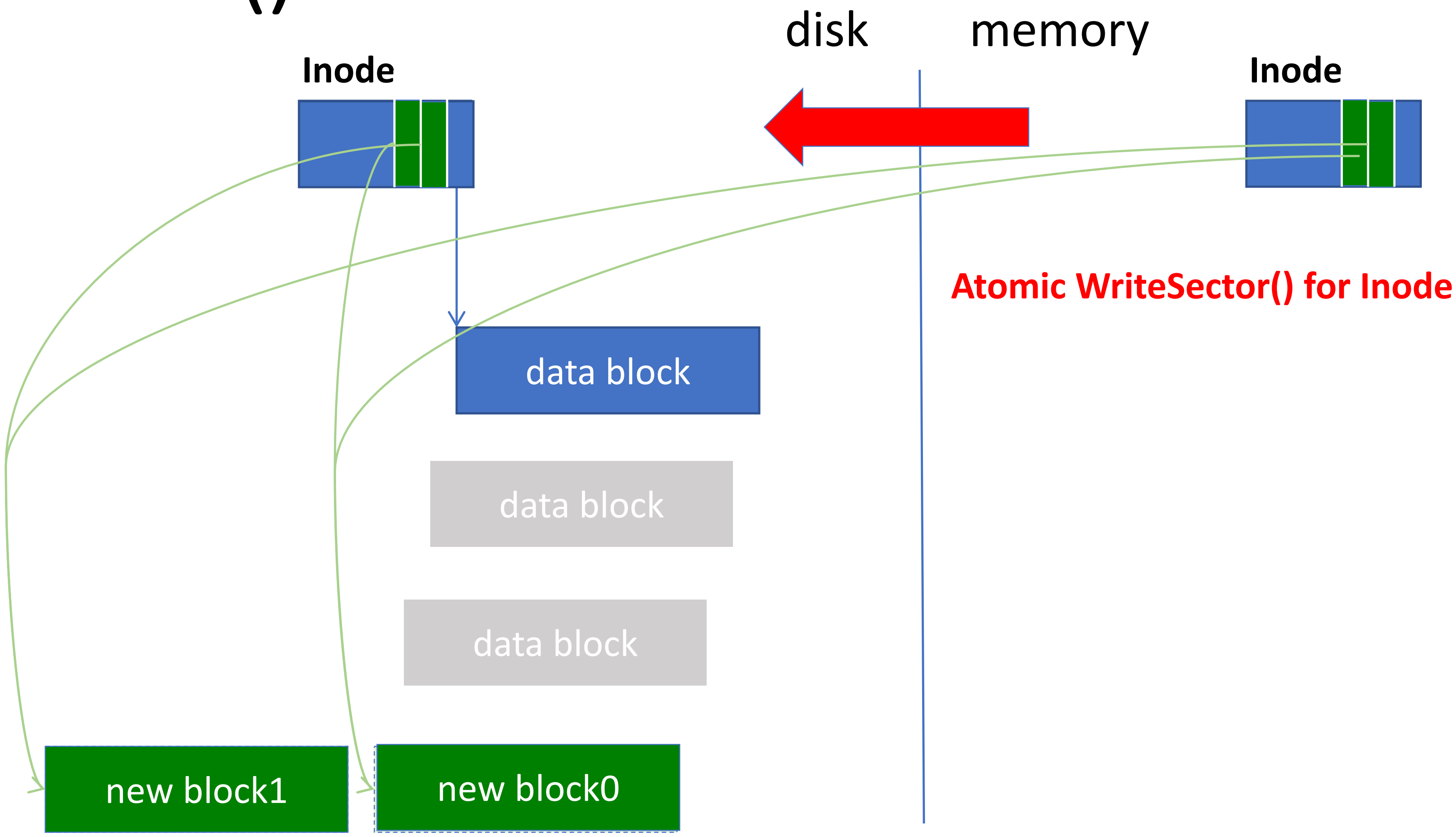
Close()



Write all blocks to disk. Not atomic.

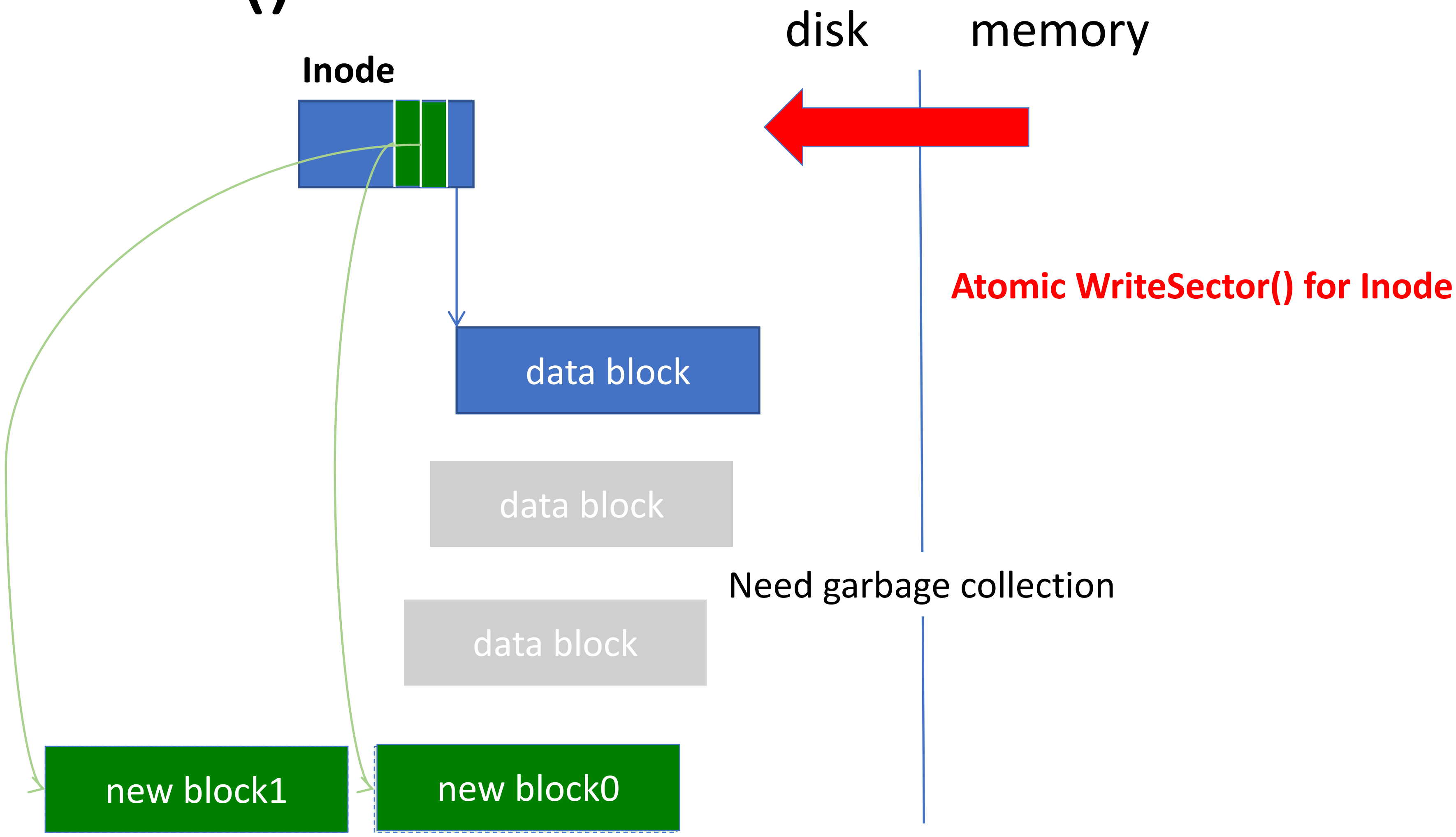
*With **write-behind** cache*

Close()



*With **write-behind** cache*

Close()



*With **write-behind** cache*

What happens to old blocks?

data block

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

Approach 2: Intentions Log

- Reserve an area of disk for (intentions) log

During Normal Operation

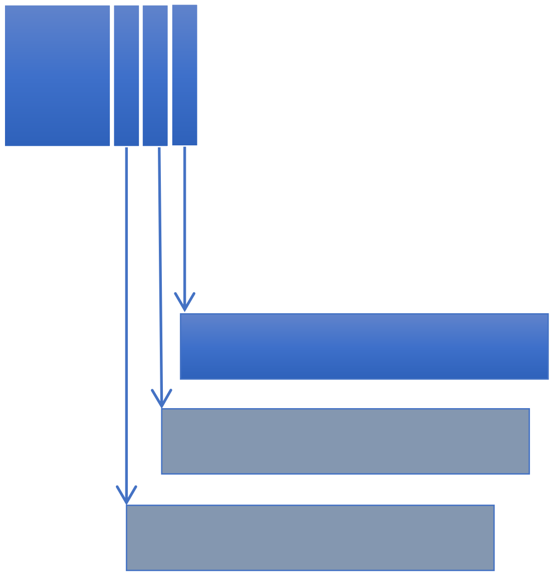
On Write():

- Write to cache
- Write to log
- **Make in-memory Inode point to update in log**

Initial State

memory

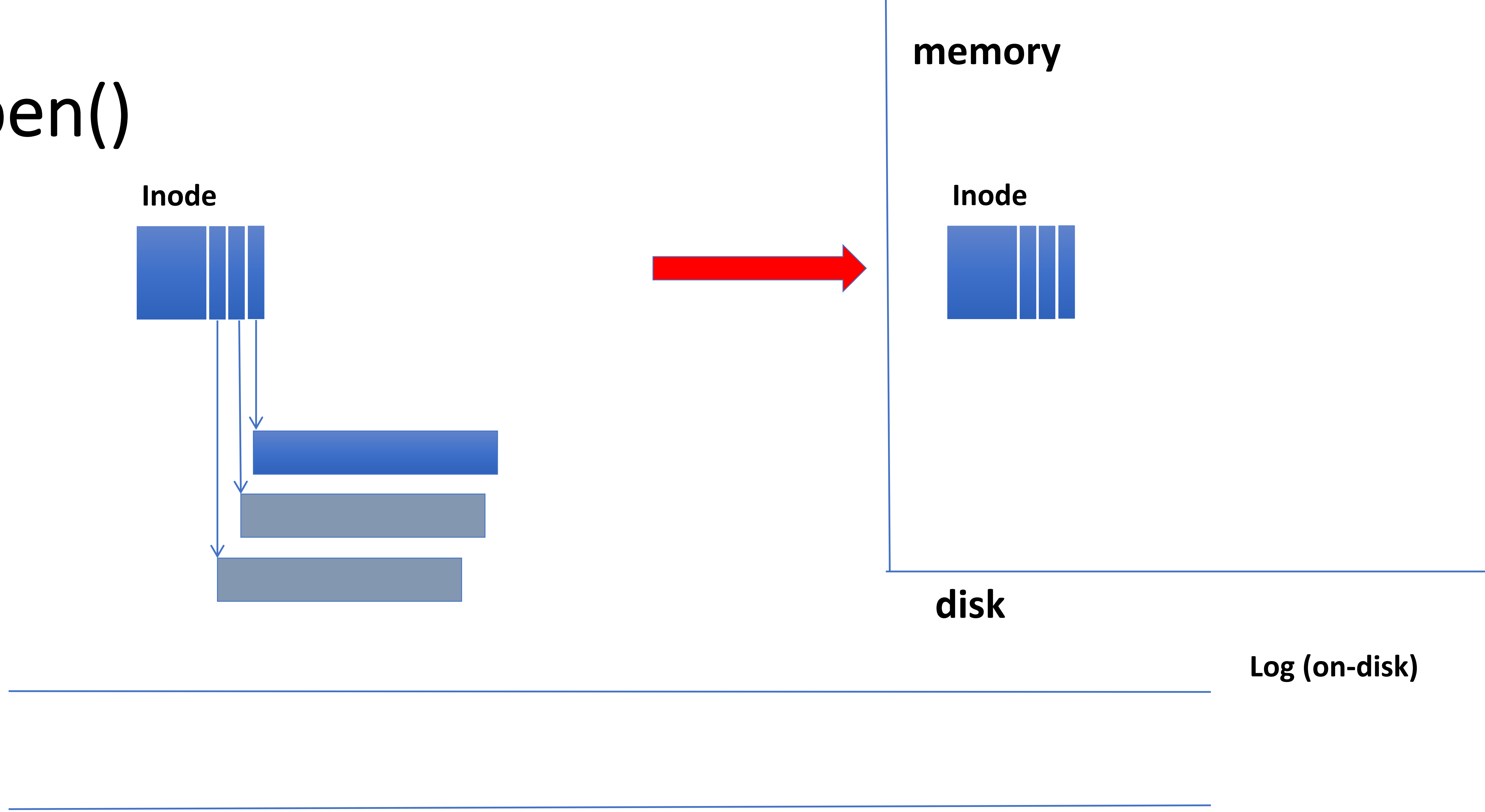
Inode



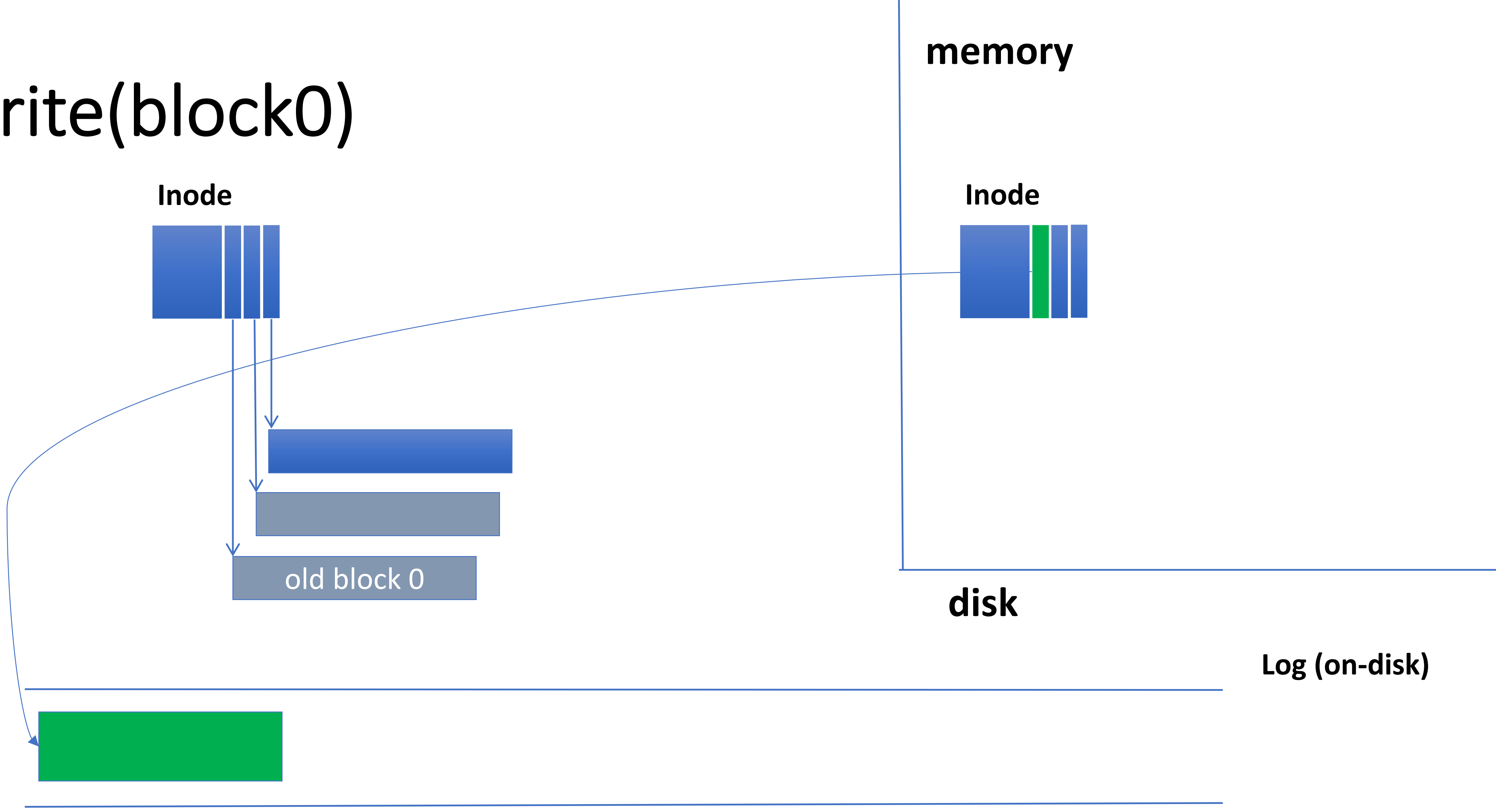
disk

Log (on-disk)

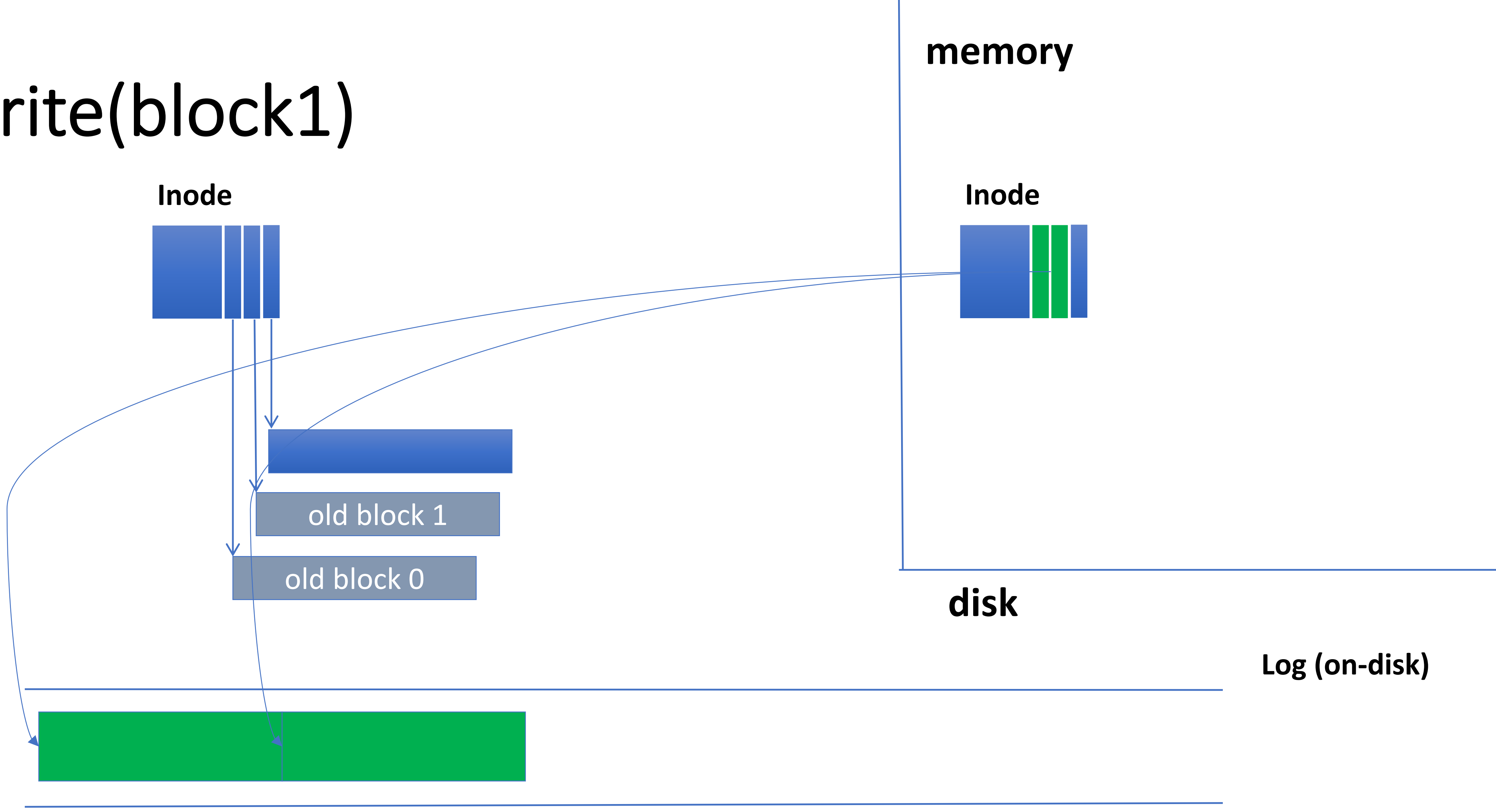
Open()



Write(block0)



Write(block1)



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

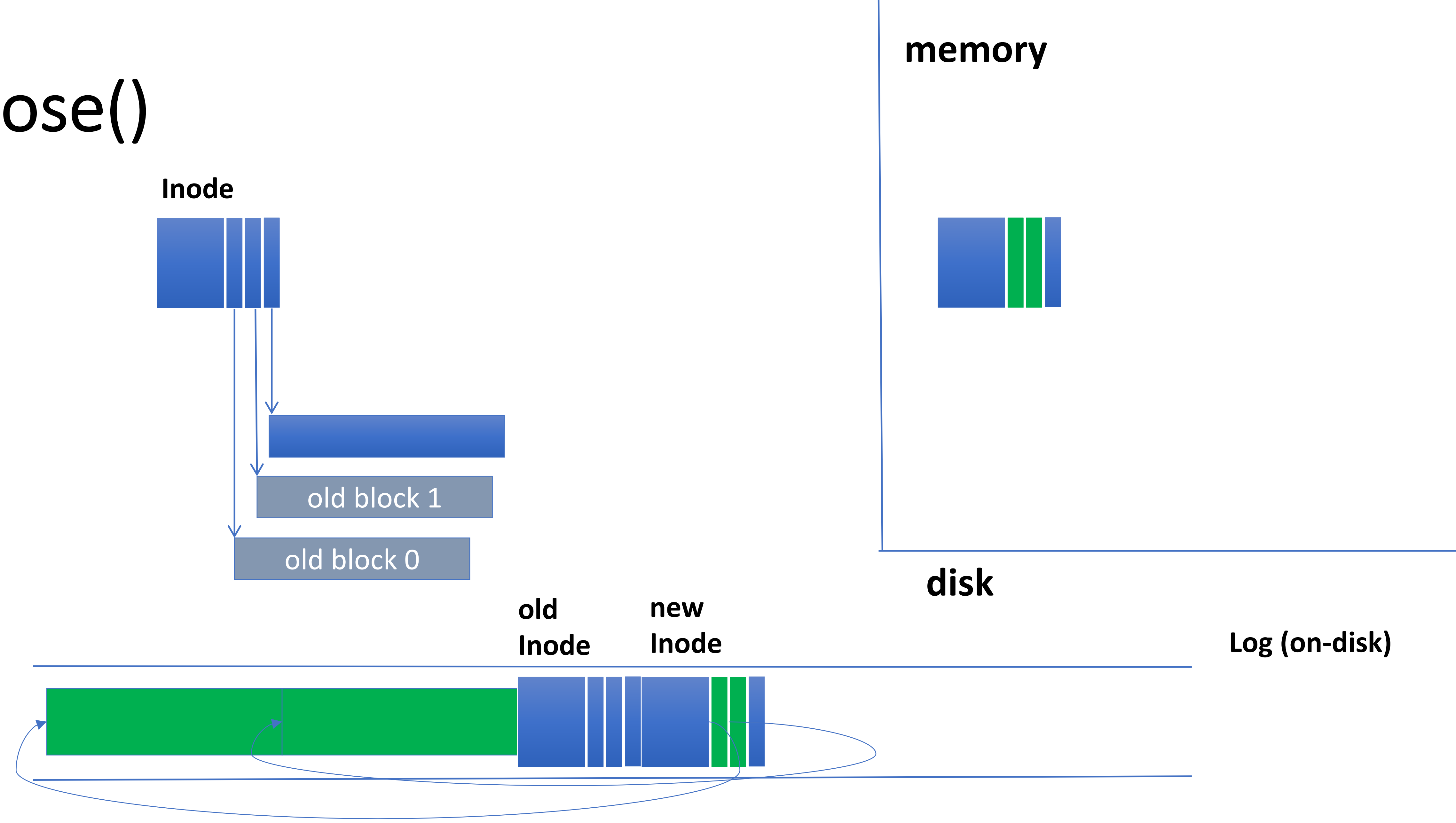
During Normal Operation

On Close():

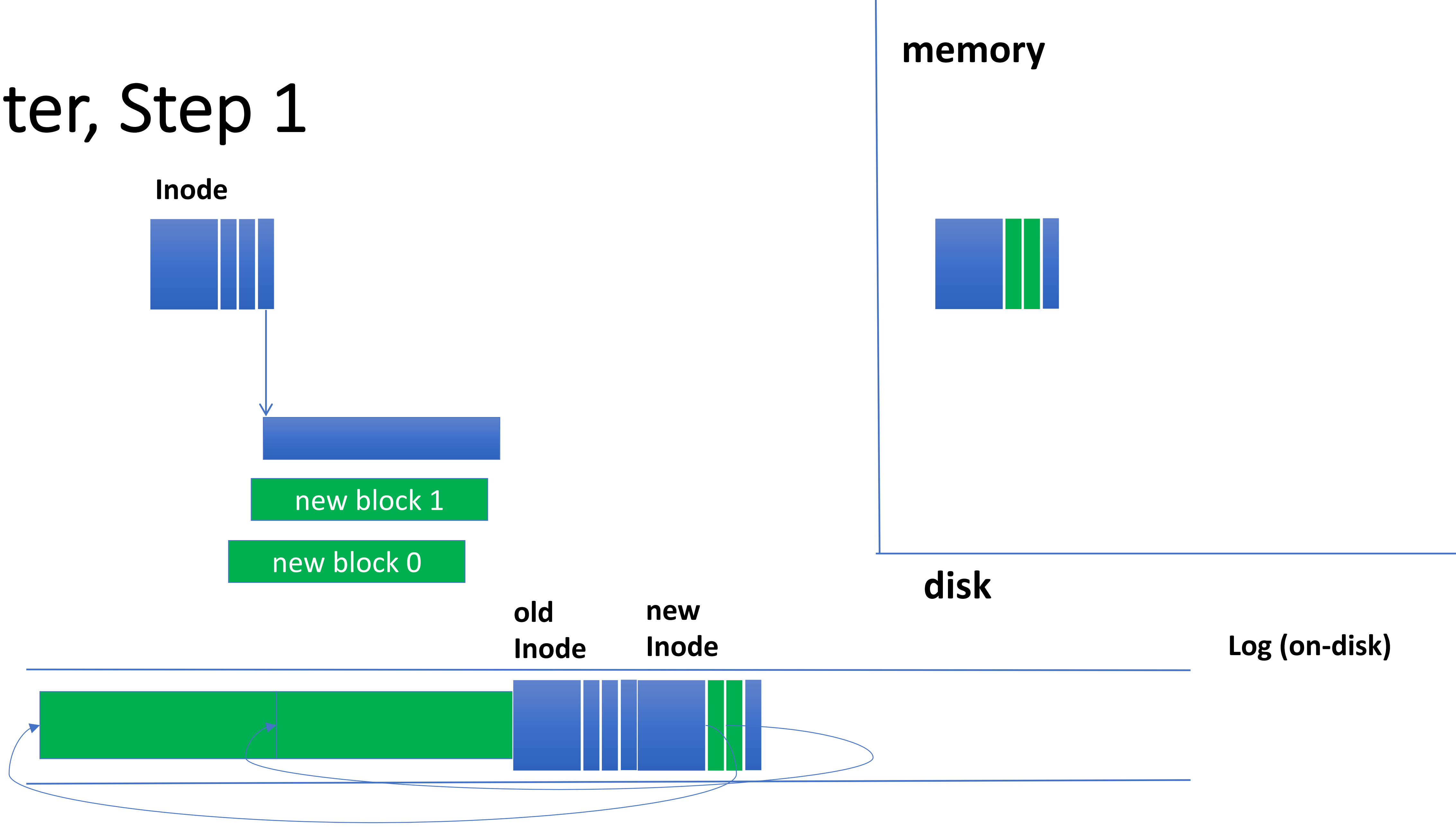
- Write old and new inode to log in one disk write
- Copy updates from log to original disk locations
- When all updates done, overwrite inode with new value
- Remove updates and old and new inode from log

Do later,
In the background

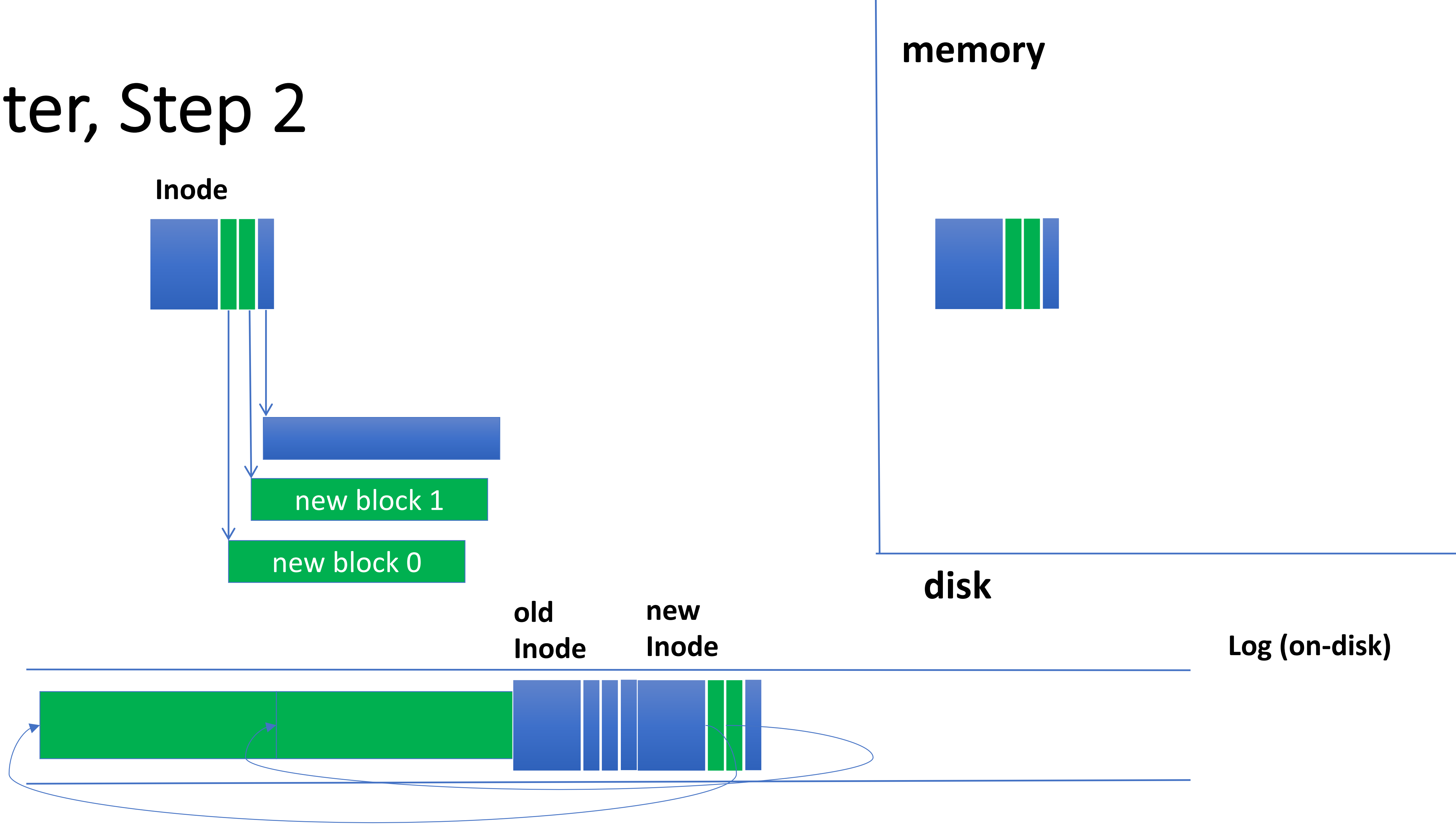
Close()


memory
Inode
old block 1
old block 0
disk
old
Inode
new
Inode
Log (on-disk)

Later, Step 1


Inode
new block 1
new block 0
old
Inode
new
Inode
memory
disk
Log (on-disk)

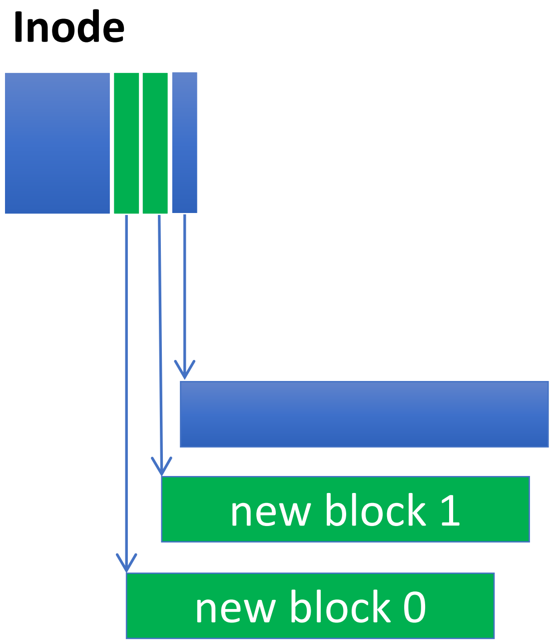
Later, Step 2



memory
Inode
new block 1
new block 0
disk
old
Inode
new
Inode
Log (on-disk)

Later, Step 3

memory



disk

Log (on-disk)

Crash Recovery

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

Intentions Log Invariant

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

Which One Works Better?

How to Compare File System Methods?

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

Which one works better?

Technique 1: Shadow Paging

Technique 2: Intentions Log

Which one works better?

Technique 1: Shadow Paging

- **two** disk writes: one for data block, one for inode.

Technique 2: Intentions Log

- **four** disk writes: two for data block, two for inode.

Surprisingly, Log works better

- ✓ Writes 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, Shadow Paging works less well

☹ Disk allocation gets messed up

☹ Fragmentation

Log-Structured File System (LFS)

- Alternative way of structuring file system
- Log = **append-only** data structure (on disk)

LFS Motivation

LFS design takes into account:

- Growing memory sizes:
  - →Most frequent reads are cached
  - →FS performance comes from write performance
  - →Optimize for writes!
- Large gap between random I/O and sequential I/O performance.

LFS Idea

Use disk purely sequentially

- Easy for writes
  - Can do all writes near each other to empty space – new copy

LFS Idea

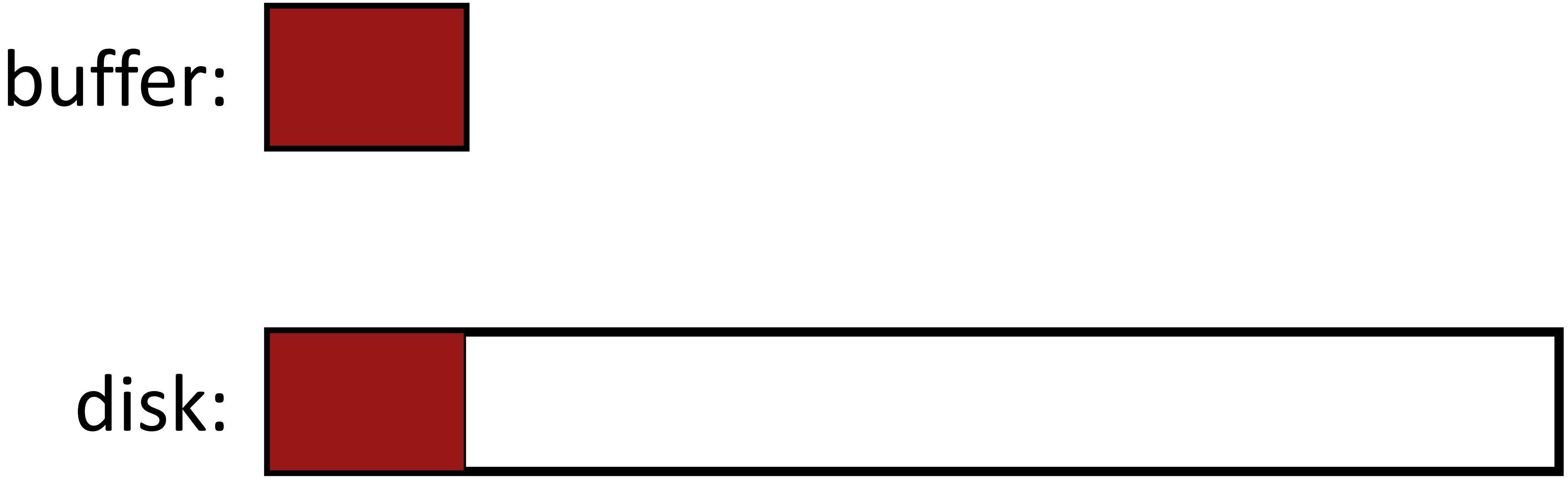
Use disk purely sequentially

- Hard for reads
  - User might read files X and Y not near each other on disk
  - Maybe not be too bad if disk reads are slow – why?
    - Memory sizes are growing (cache more reads)

LFS Strategy

- File system buffers writes in main memory until “enough” data
  - Write both Inodes and data
- Write buffered information *sequentially* to new segment on disk
  - Segment: large (MBs) contiguous regions on disk
- Never overwrite old info: old copies left behind
  - Old copies garbage collected later

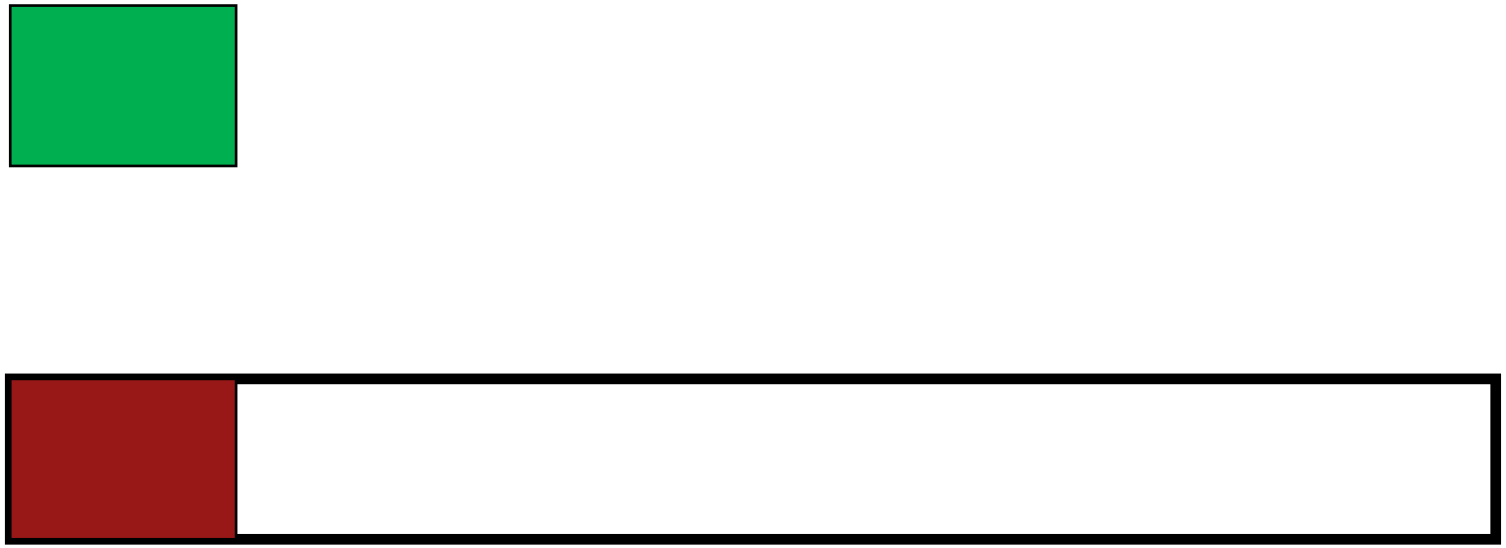
LFS Big Picture



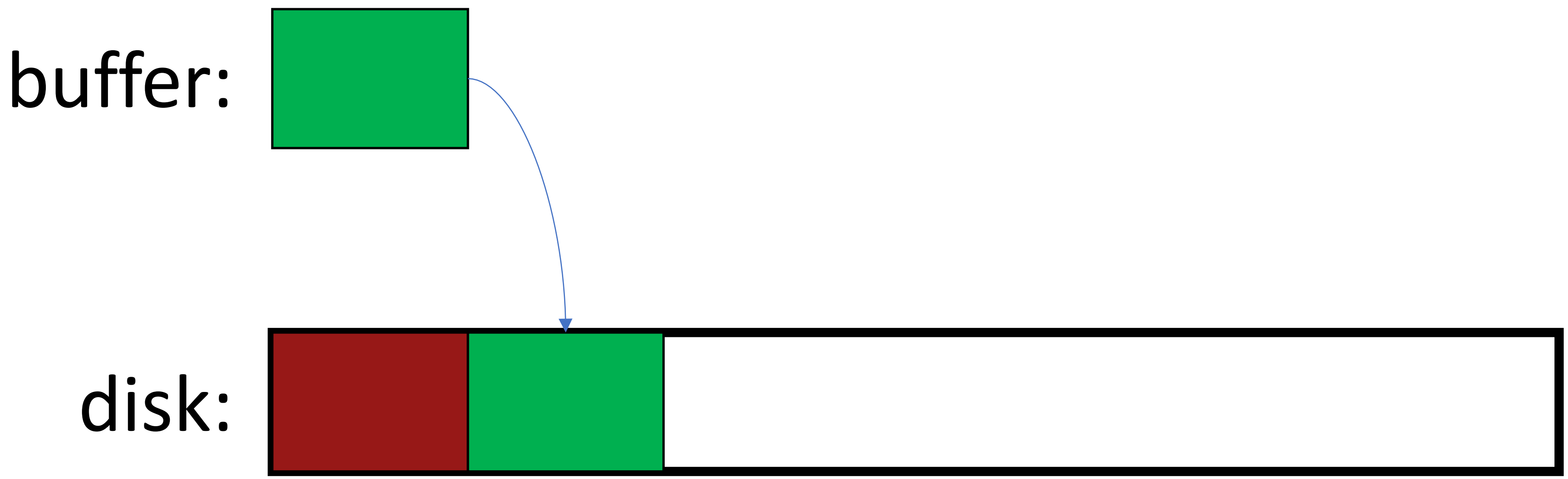
LFS Big Picture

buffer:

disk:



LFS Big Picture

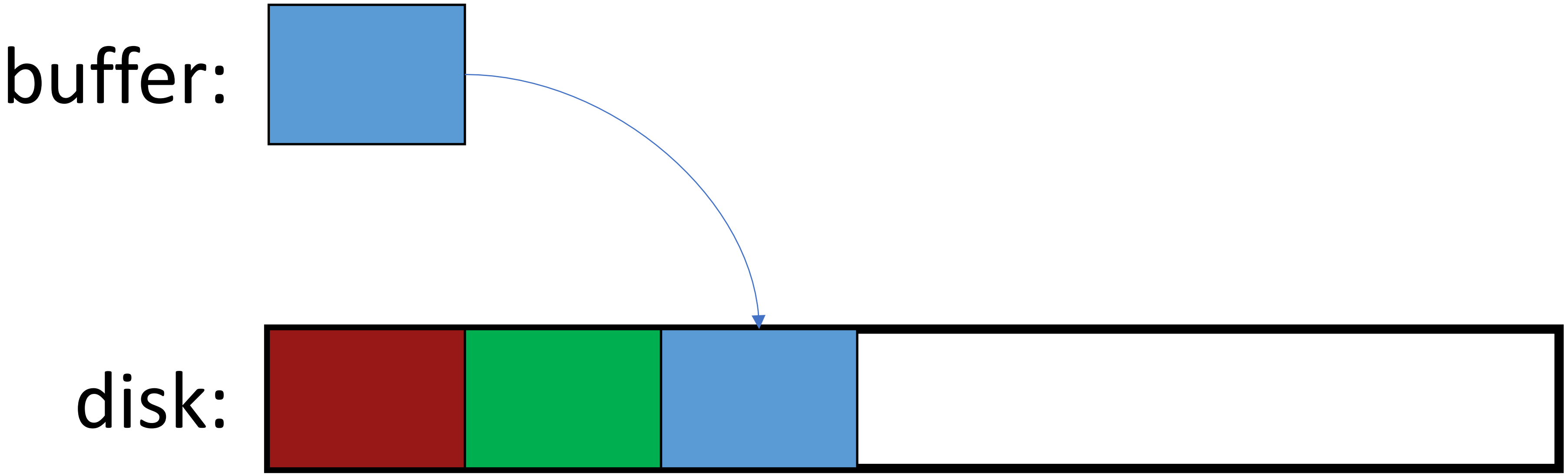


LFS Big Picture

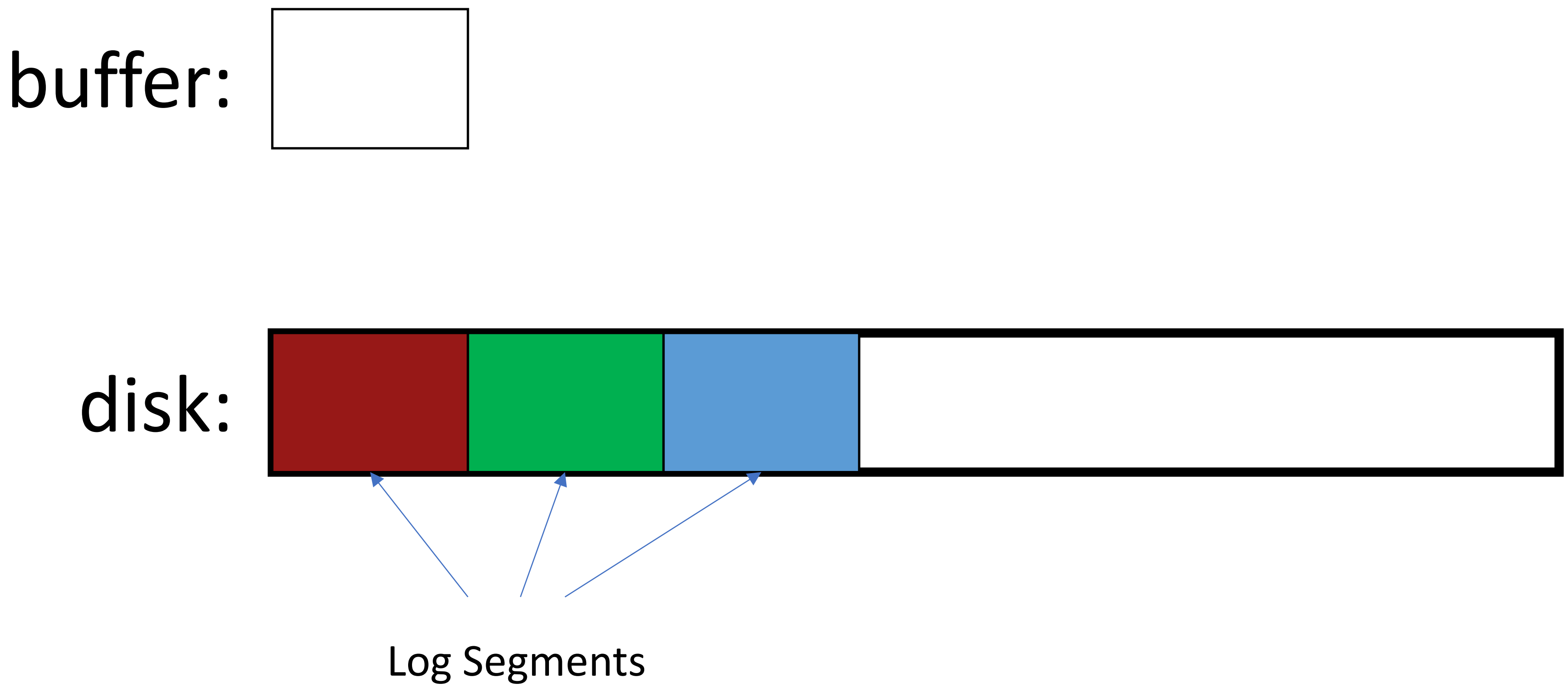
buffer:

disk:

LFS Big Picture

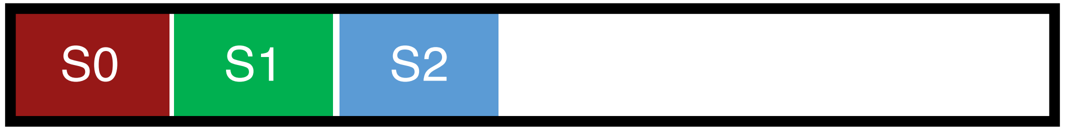


LFS Big Picture



LFS Segments

Data Structures (Attempt 1)



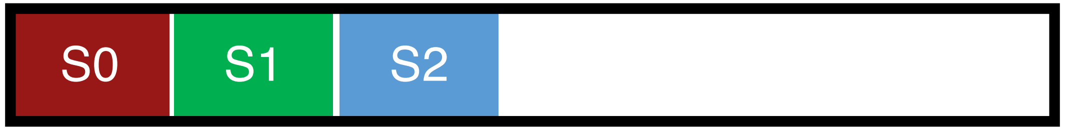
What data structures can LFS remove?

- allocation structs: data + inode bitmaps

What is much more complicated?

- Inodes are no longer at fixed offset

Data Structures (Attempt 1)

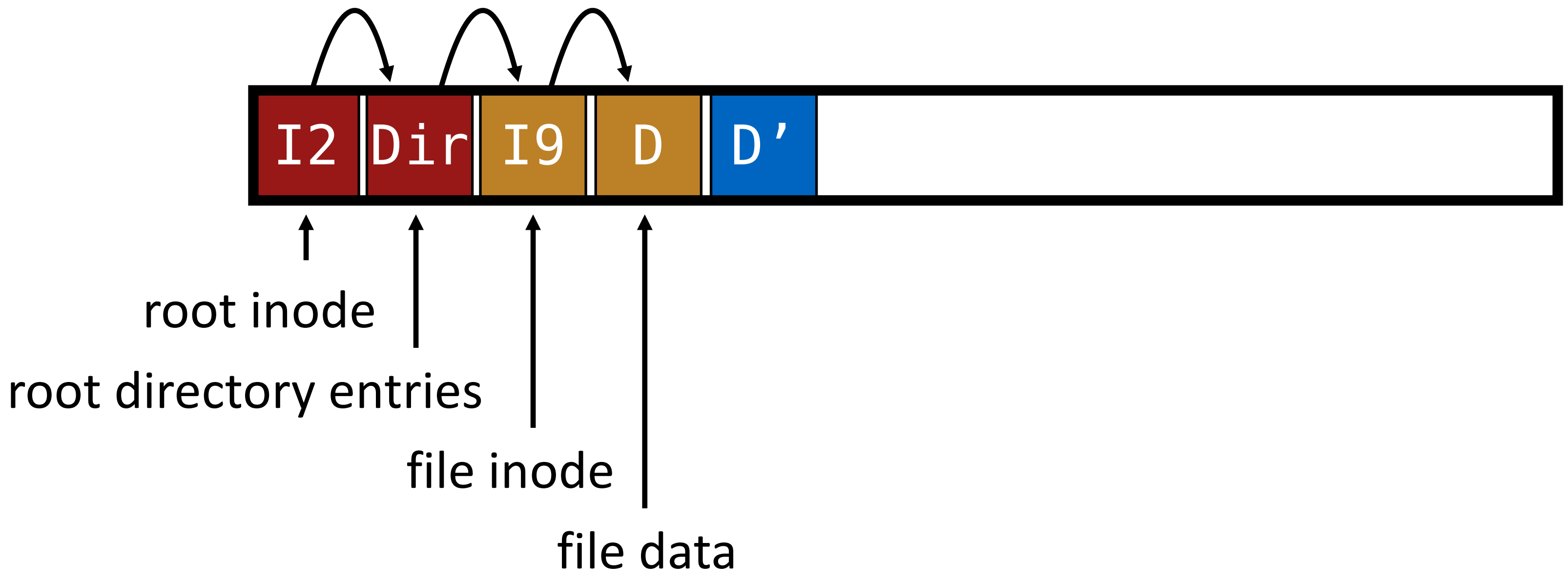


Naïve Idea:

- Use **current offset on disk** instead of table index for uid
- When update inode, inode number changes!

Attempt 1

Overwrite data in /file.txt



How to update Inode I9 to point to new D’ ?

Attempt 1

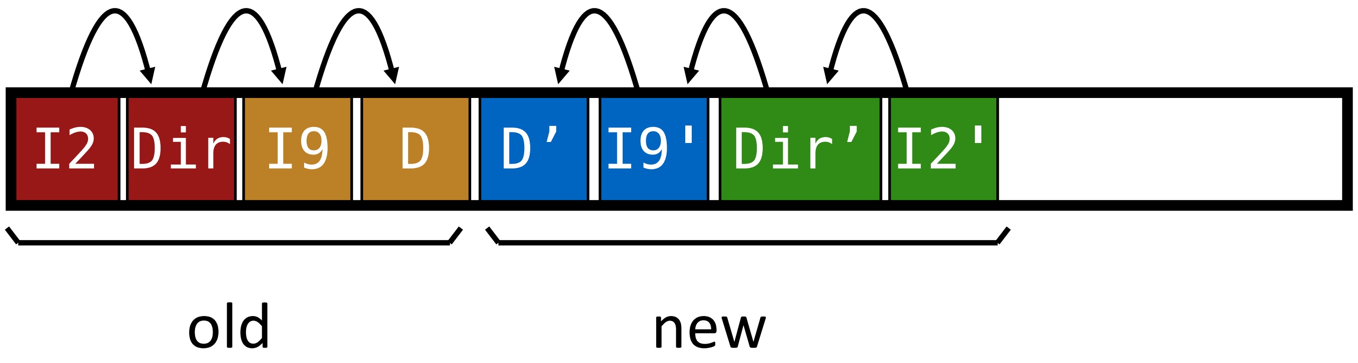
Overwrite data in /file.txt

Can LFS update Inode 9 to point to new D’?

NO! This would be a random write

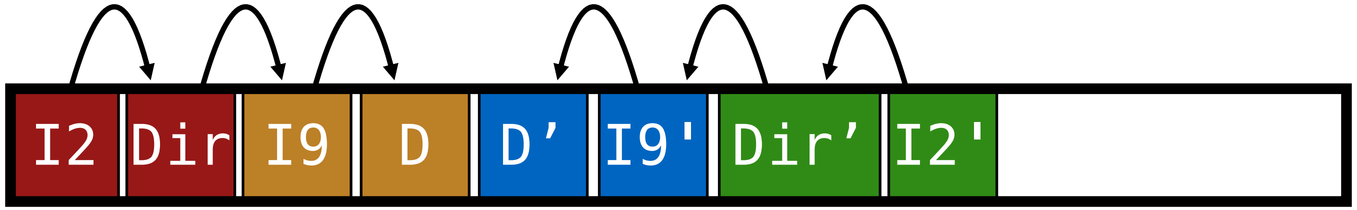
Attempt 1

Overwrite data in /file.txt



Must update all structures in sequential order to log

Attempt 1: Problem w/ Inode Numbers



Problem

For every data update

Must propagate updates all the way up directory tree to root

Why?

When inode copied, its location (inode number) changes

Solution

Keep uids (and inode numbers) constant; **don’t base name on offset**

Data Structures (Attempt 2)

Data Structures (Attempt 2)

Better Idea: add a level of indirection

- **Map:** file uid → Inode location on disk
- Data structure is called **Imap (Inode map)**

The Imap

- 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 Imap

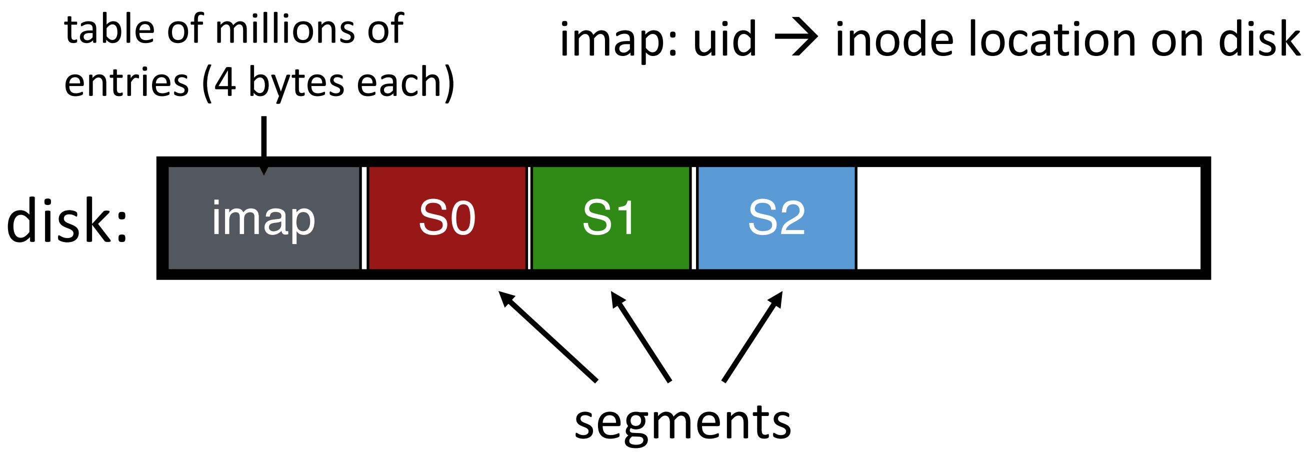
Open() :

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

Read() : as before

- Get from cache
- Get from disk address in inode

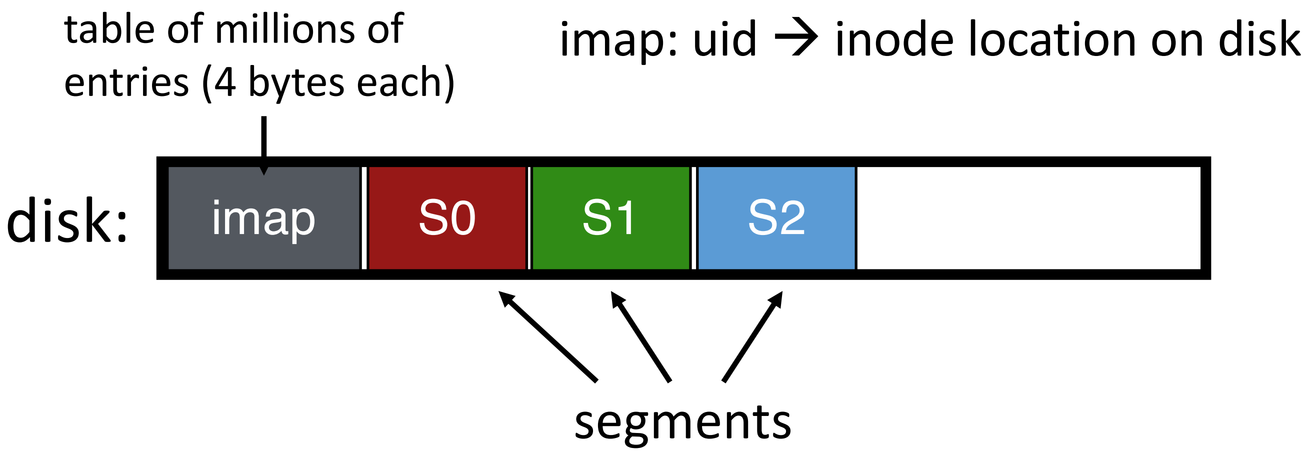
Where to keep Imap?



Where can imap be stored? Dilemma:

- imap too large to keep in memory
- don’t want random writes for imap

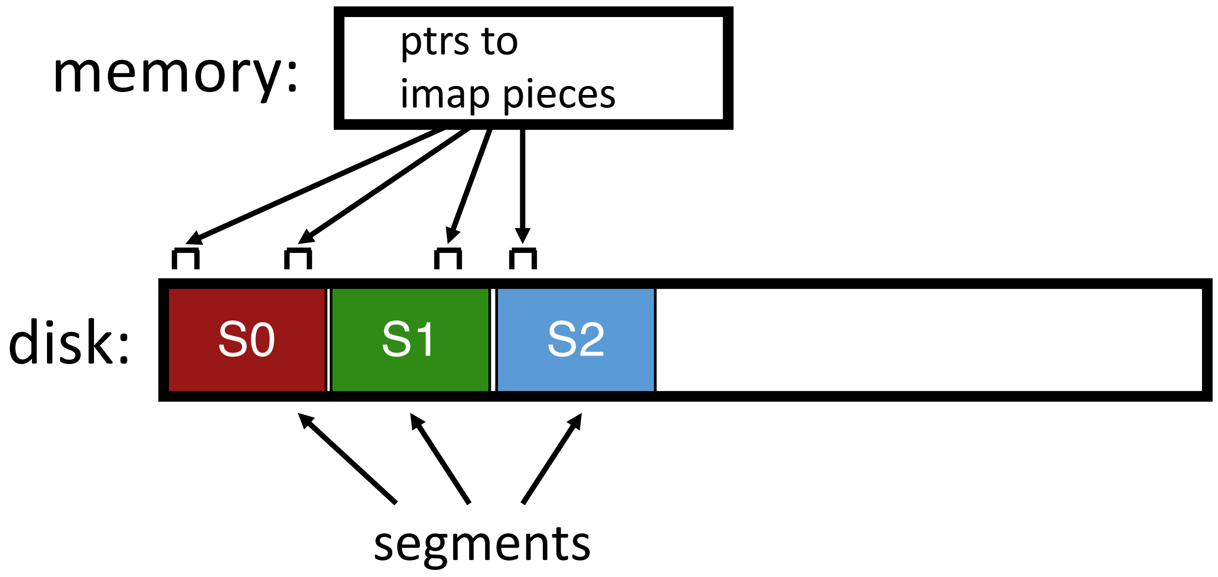
Where to keep Imap?



Solution

- Write imap in segments
- Keep pointers to pieces of imap in memory

Solution: Imap in Segments



- Keep pointers to pieces of imap in memory
- Keep recent accesses to imap cached in memory

Crash?

Crash?

What data needs to be recovered after a crash?

- Need Imap

Imap Recovery - Scan

Naïve approach

- Scan entire log to reconstruct pointers to Imap pieces
- Slow!

Imap Recovery - Checkpointing

Better approach

- Write copy of inode map to fixed location on disk
- Put marker in the log (called tail marker)
- Checkpoint periodically. Example: every 30 seconds

Time Interval between Checkpoints

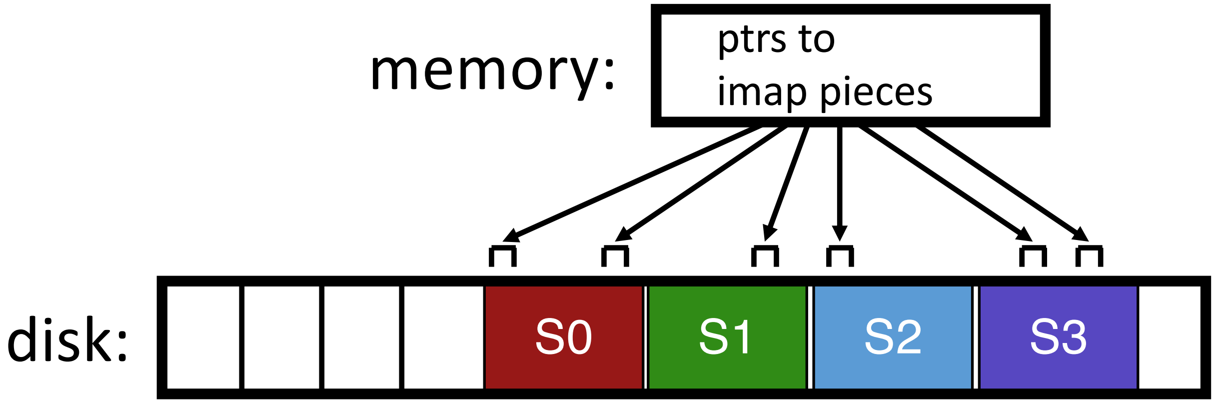
- Too short: lots of disk I/O to write checkpoints
- Too long: long recovery time (forward scan)
- Compromise
  - Crashes are rare
  - So recovery seldom happens
  - Can tolerate longer recovery time

An Aside: A General Rule

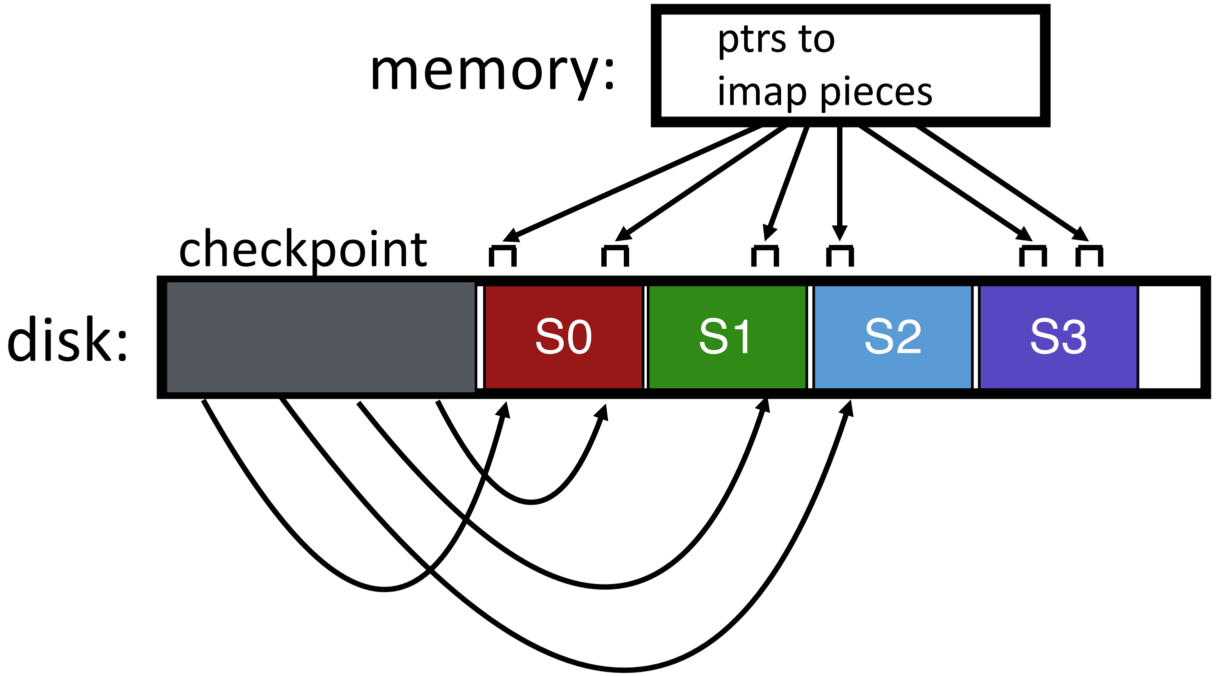
Tradeoff between

- Failure-free performance
- Recovery time

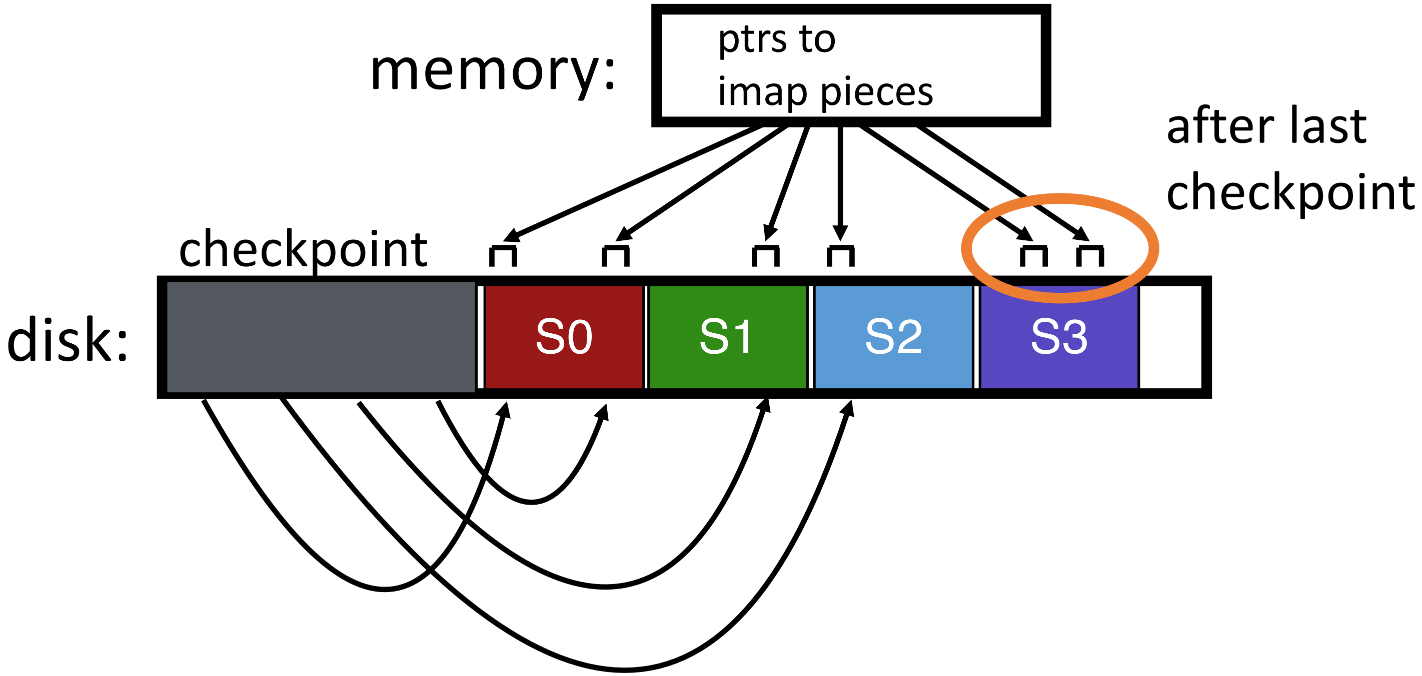
Checkpoint



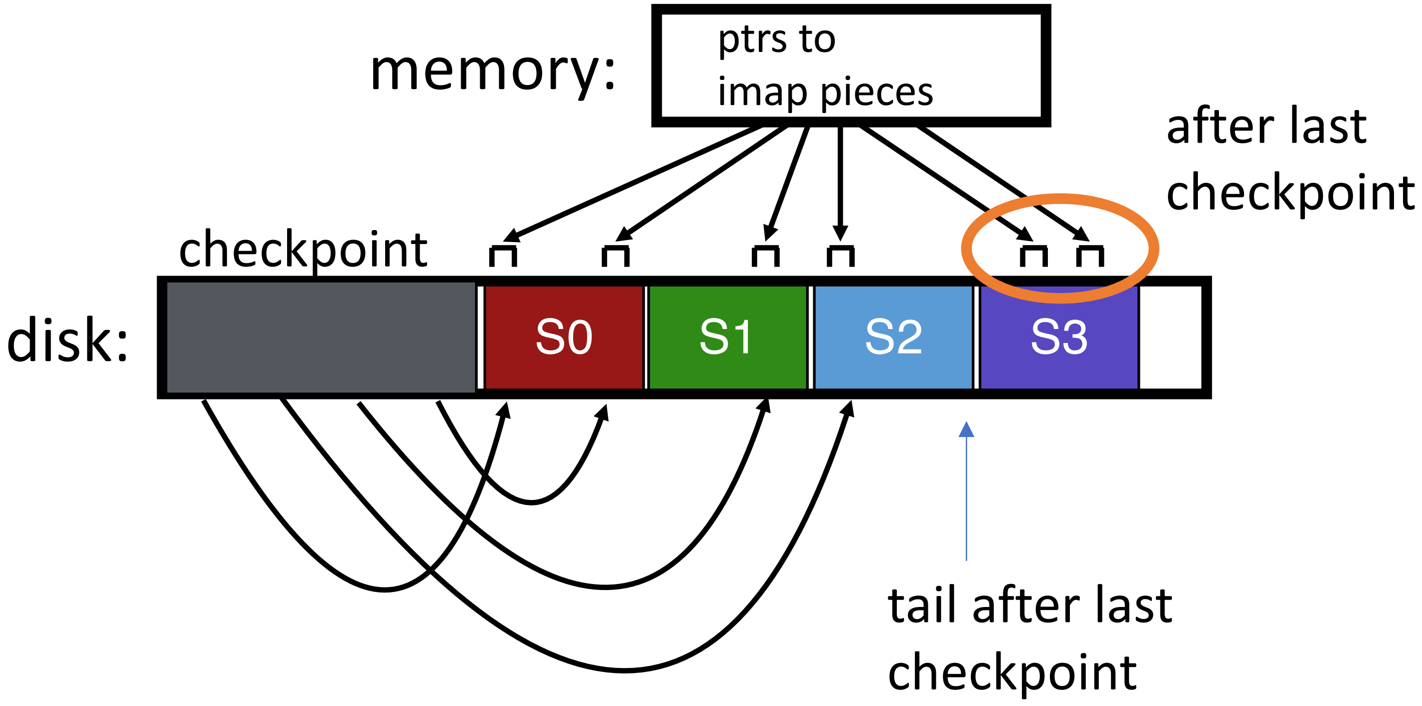
Checkpoint



Checkpoint


memory:
ptrs to
imap pieces
after last
checkpoint
checkpoint
disk:
S0
S1
S2
S3

Checkpoint



memory:
ptrs to
imap pieces
after last
checkpoint
checkpoint
disk:
S0
S1
S2
S3
tail after last
checkpoint

Crash Recovery with Checkpointing

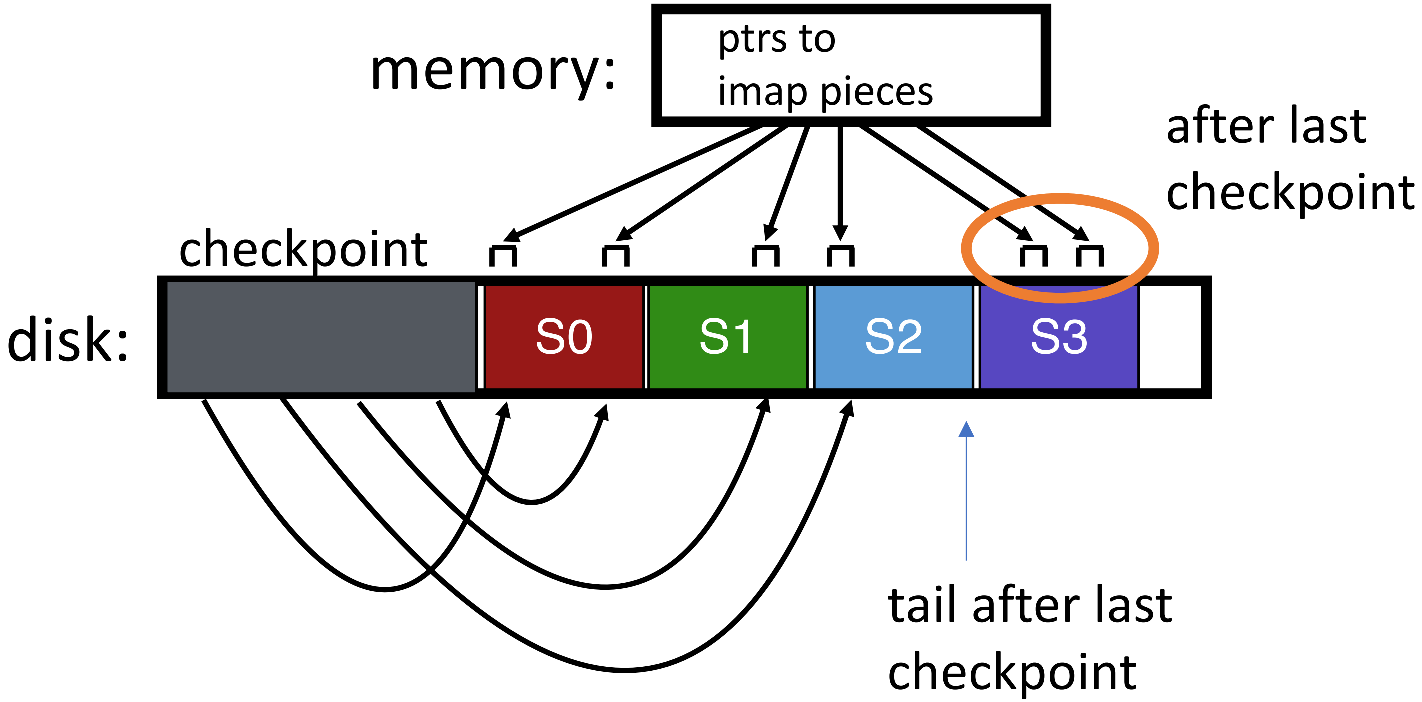
- Start from Imap in checkpoint
  - Contains addresses of all inodes written *before* last checkpoint
- How to find inodes?
  - That were in in-memory inode map before crash
  - But not written in the checkpoint

Roll Forward

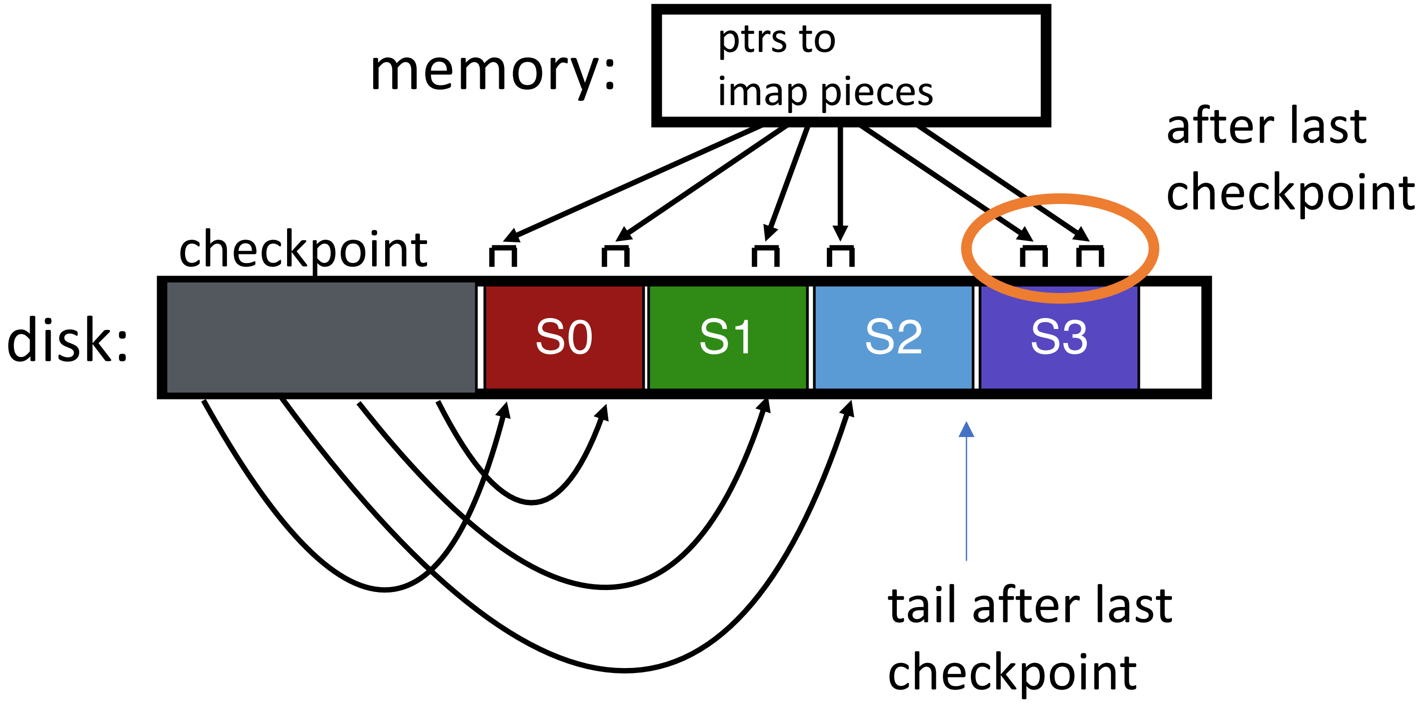
- Remember: checkpoint put marker in log
- From marker forward
  - Scan for inodes in the log
  - Add their addresses to inode map

→ Result: All inode addresses in inode map before crash are in inode map afterwards

Checkpoint


memory:
ptrs to
imap pieces
after last
checkpoint
checkpoint
disk:
S0
S1
S2
S3
tail after last
checkpoint

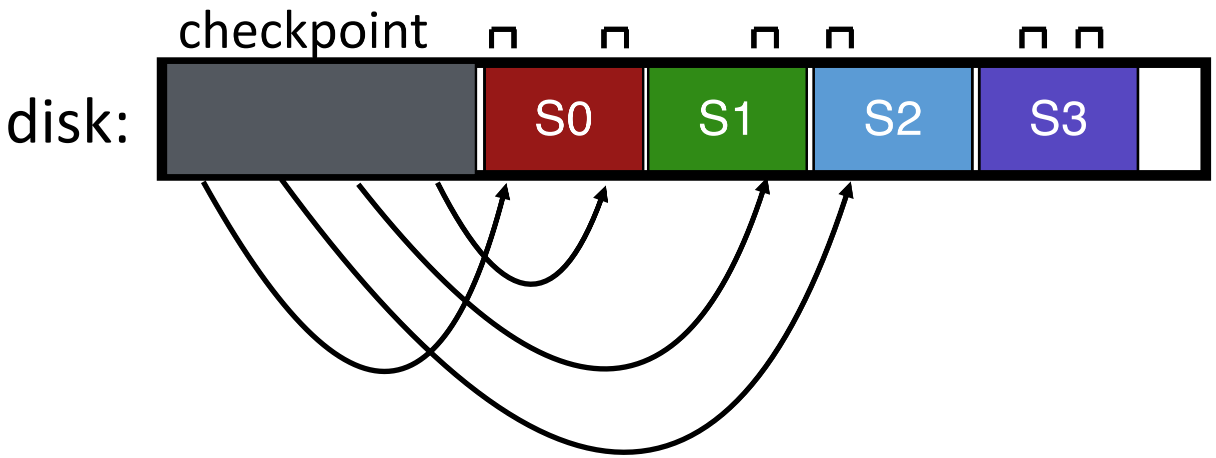
Checkpoint



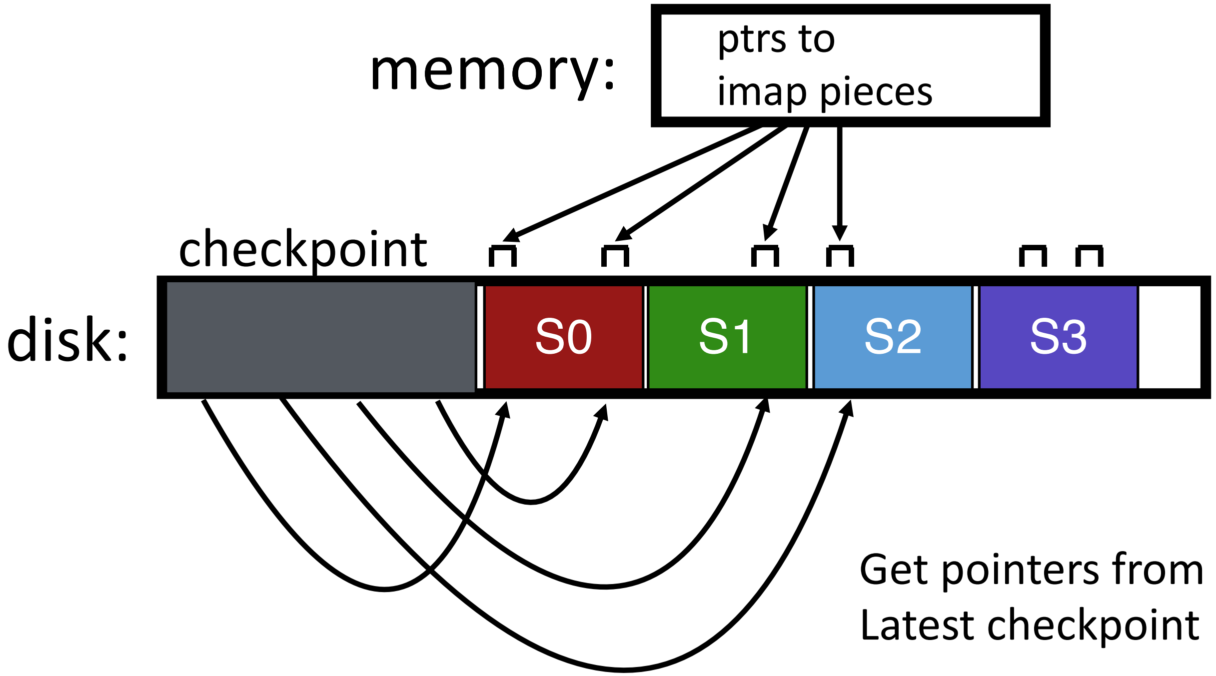
CRASH!

Checkpoint

memory:

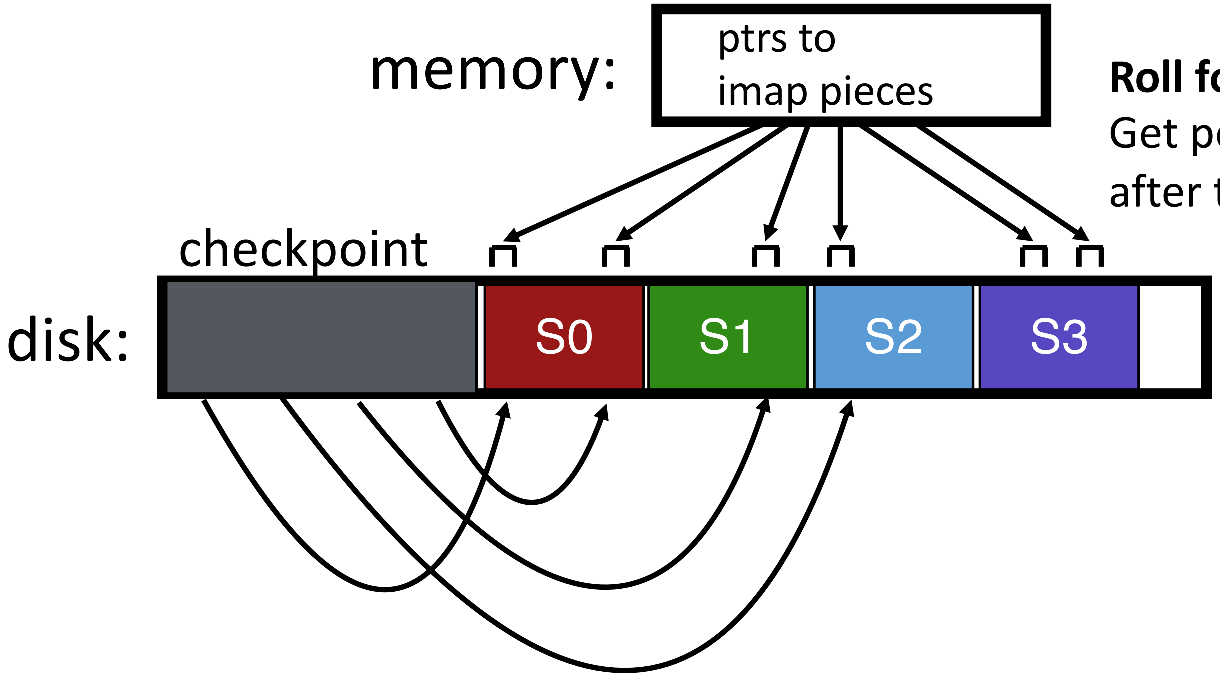


Checkpoint



RECOVERY

Checkpoint



Roll forward:
Get pointers by scanning after tail marker

RECOVERY

What if Crash During Checkpoint?

- Two checkpoint regions
- Overwrite one checkpoint at a time
- Use timestamps to identify most recent checkpoint

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
  - But not physically overwritten
    - Older version of (uid, blockno) somewhere in the log

- Segments can contain a mix of old and new data

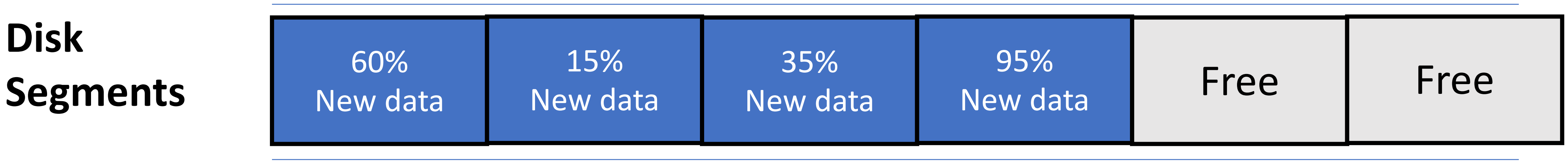
Disk Cleaning

Done one segment at a time:

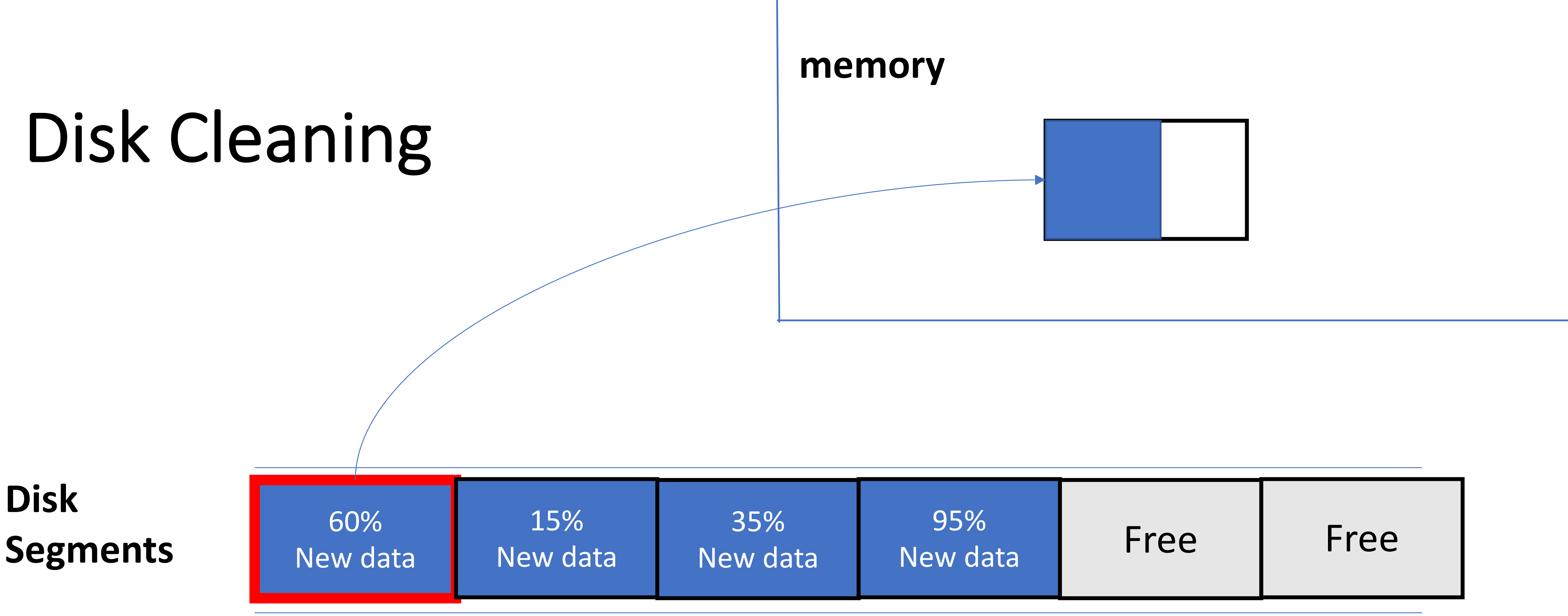
- Determine which blocks are new
- Write them into buffer
- If buffer is full, write new segment
- Cleaned segment is marked free

Disk Cleaning

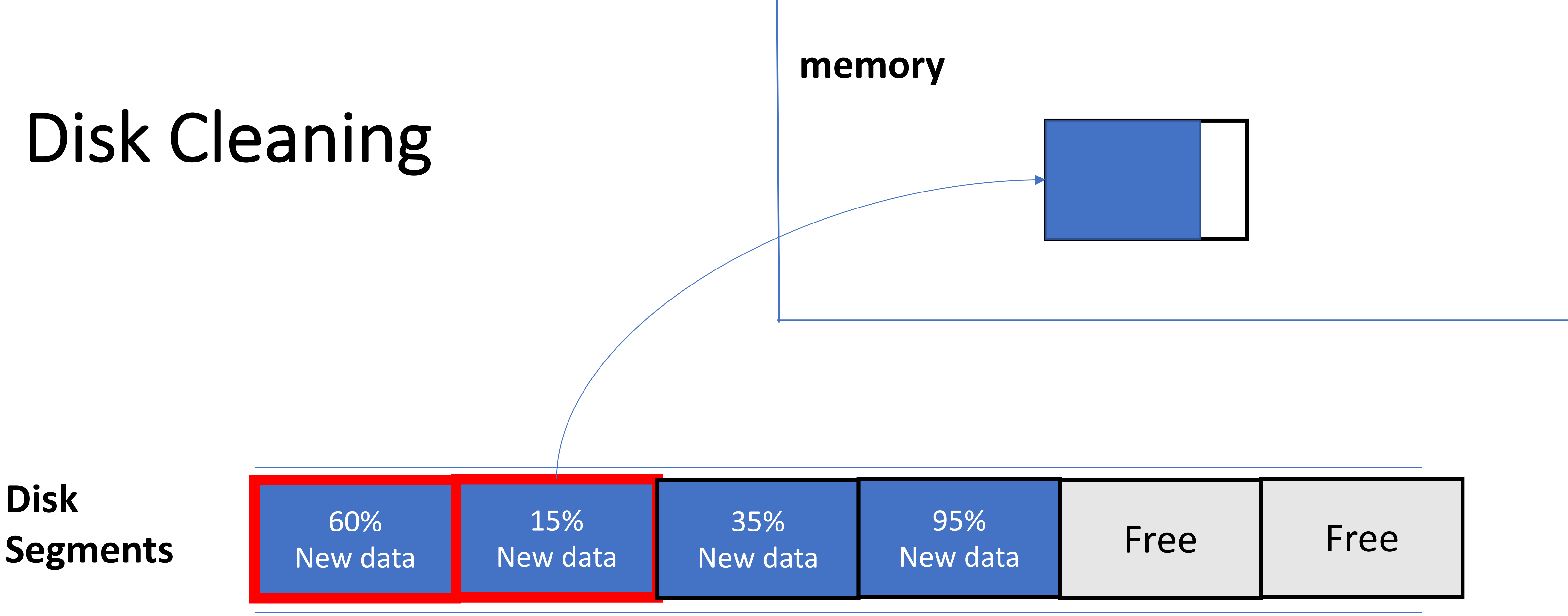
memory



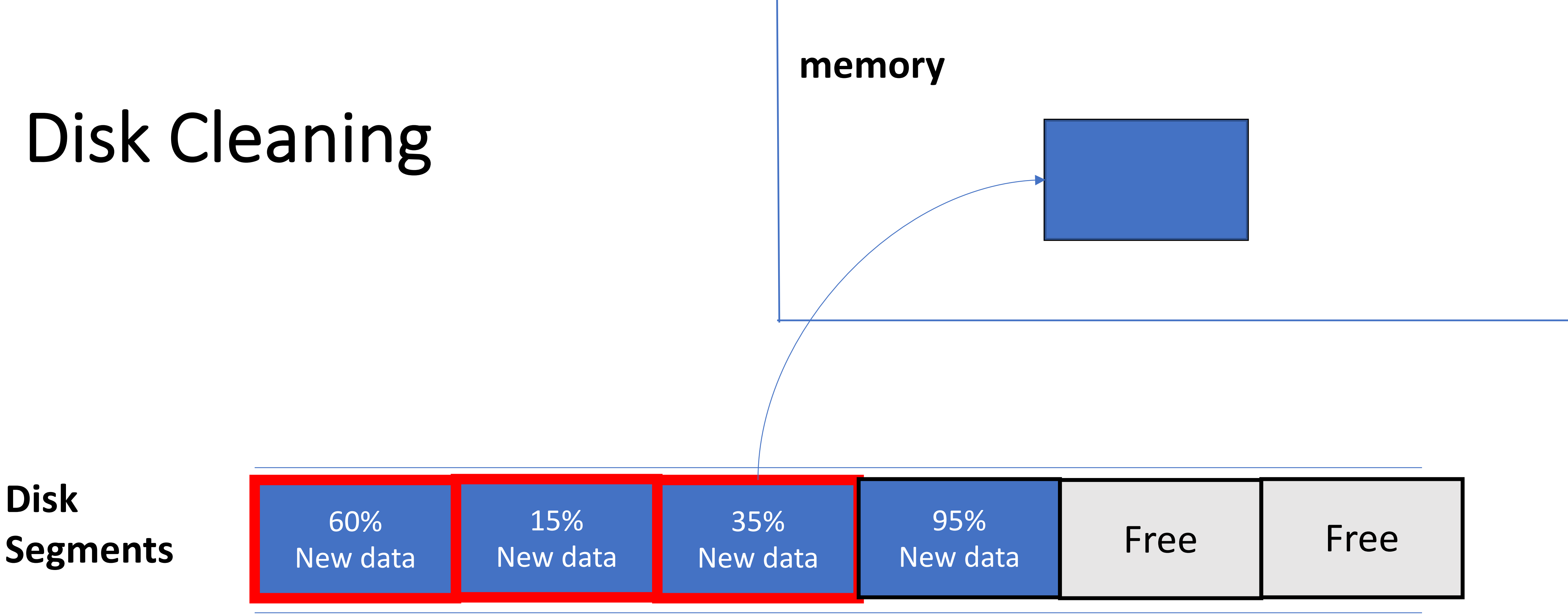
Disk Cleaning



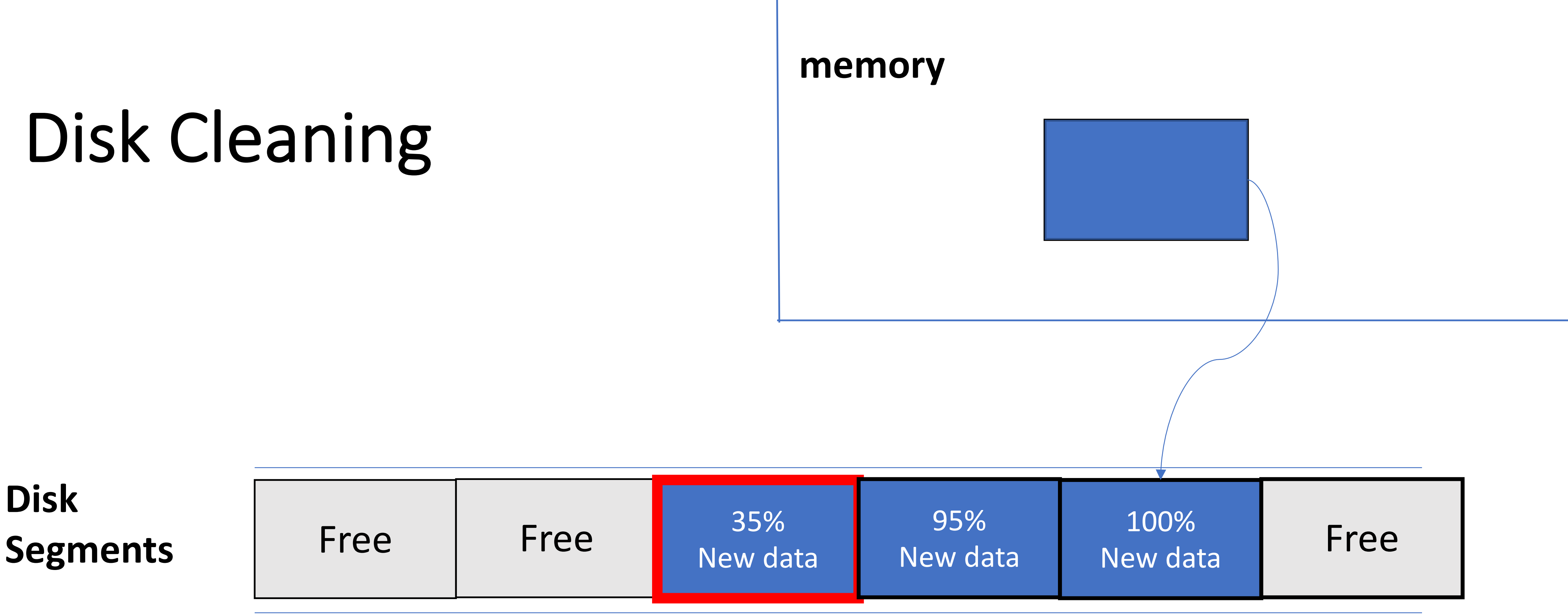
Disk Cleaning



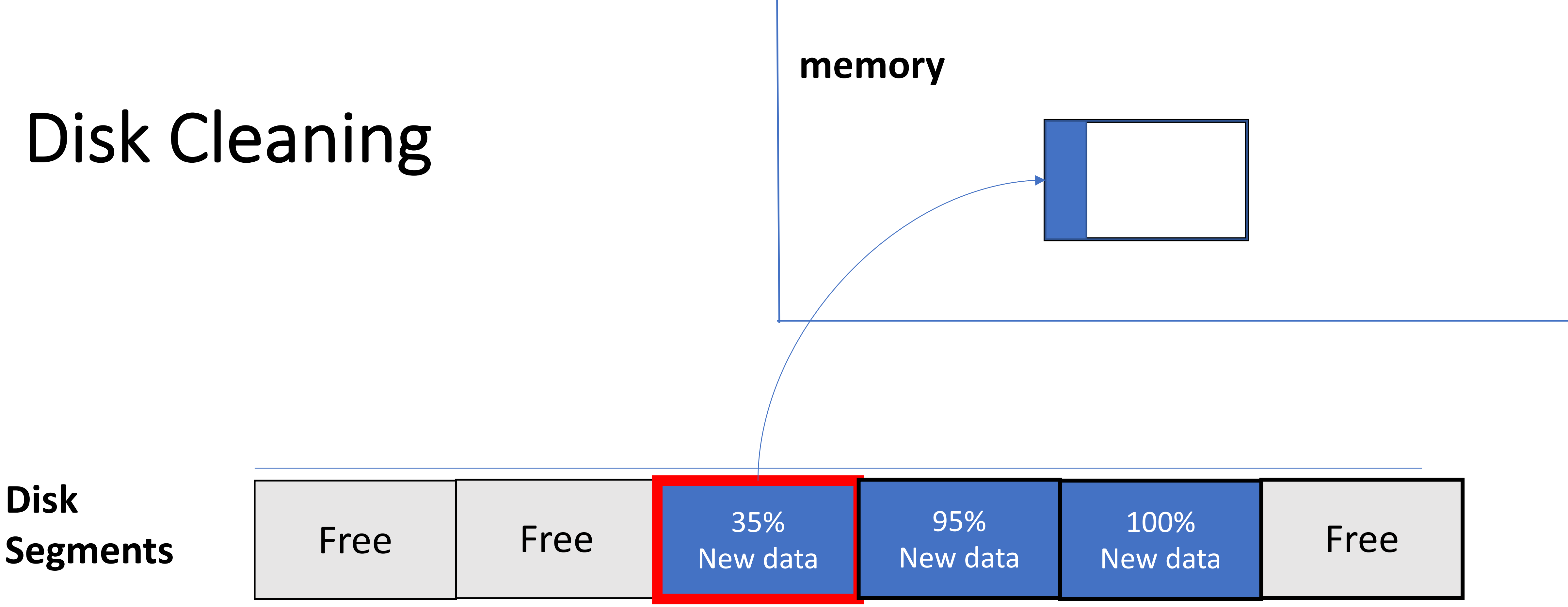
Disk Cleaning



Disk Cleaning



Disk Cleaning



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 (similar to garbage collection)
  - Unpredictable performance dips

- Similar ideas in some commercial systems
  - Log-structured merge key-value stores

← My PhD research area
will see on Thursday