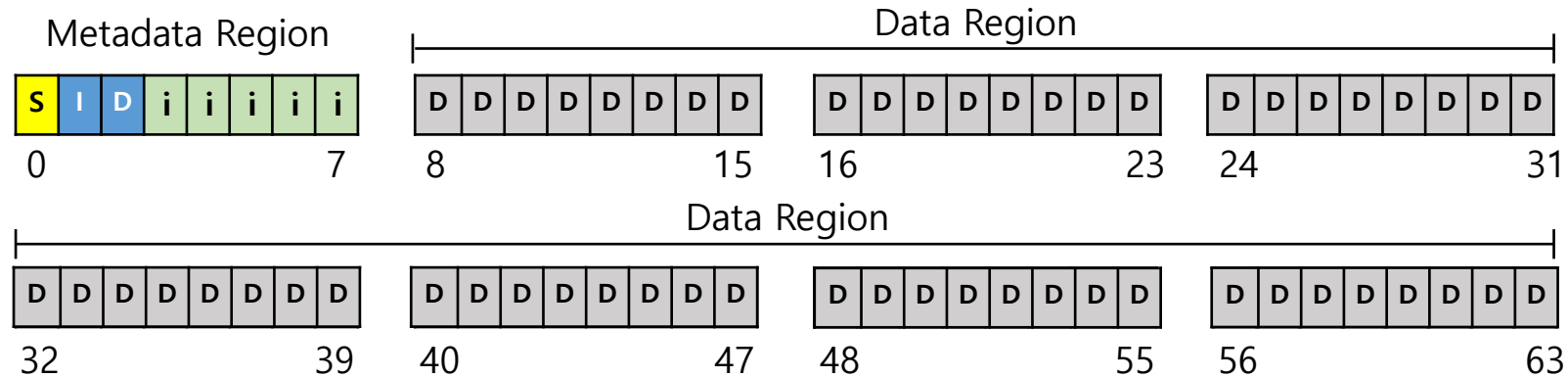


Week 12

Advanced FS (1): Shadow Paging

Max Kopinsky
27 March, 2025

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 ) ← crash
2. Write( fd, 0 )
3. Write( fd, 1 )
4. Write( fd, 2 )
5. Write( fd, 3 )
6. Close( fd )
```

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 (blocks might not be written)

The Notion of Atomicity

Atomicity means one uninterruptible operation

For a file system, atomicity means

- **“All or Nothing”**
- 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 )  
4. Read( balance_checking )  
5. balance_checking += 100  
6. Write( balance_checking )
```

← crash

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

Before WriteSector

Disk Sector

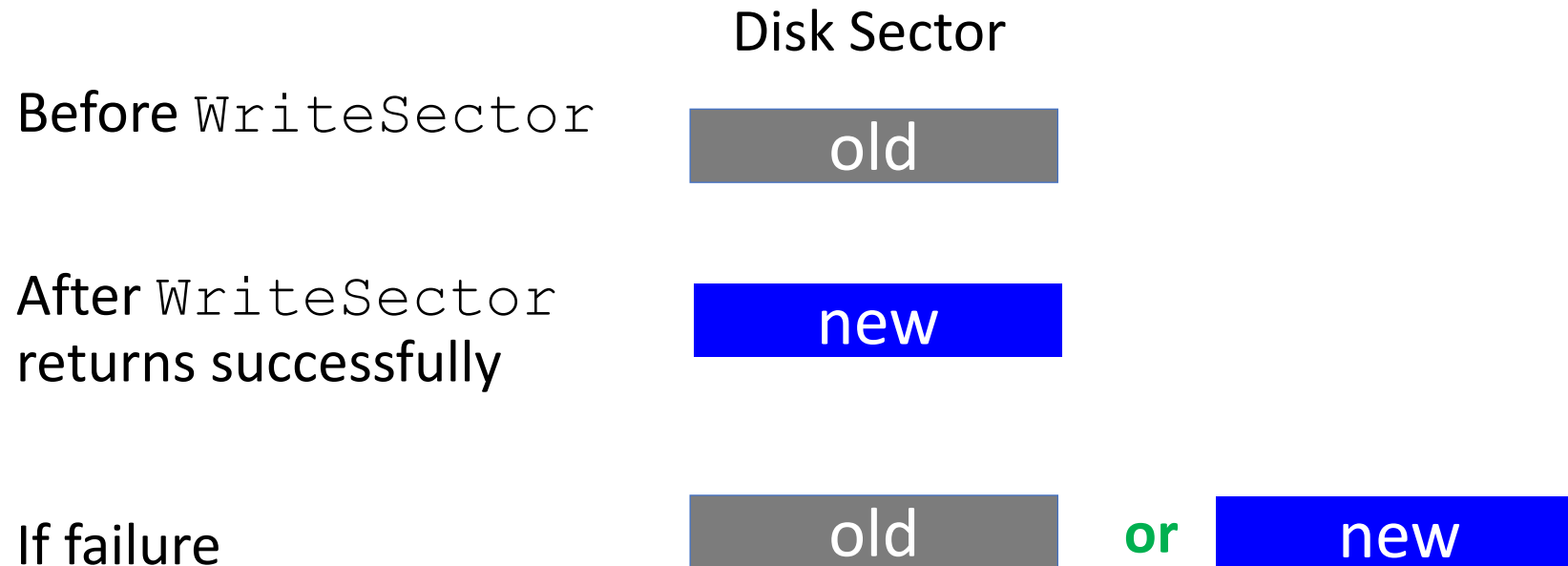
old

After WriteSector
returns successfully

new

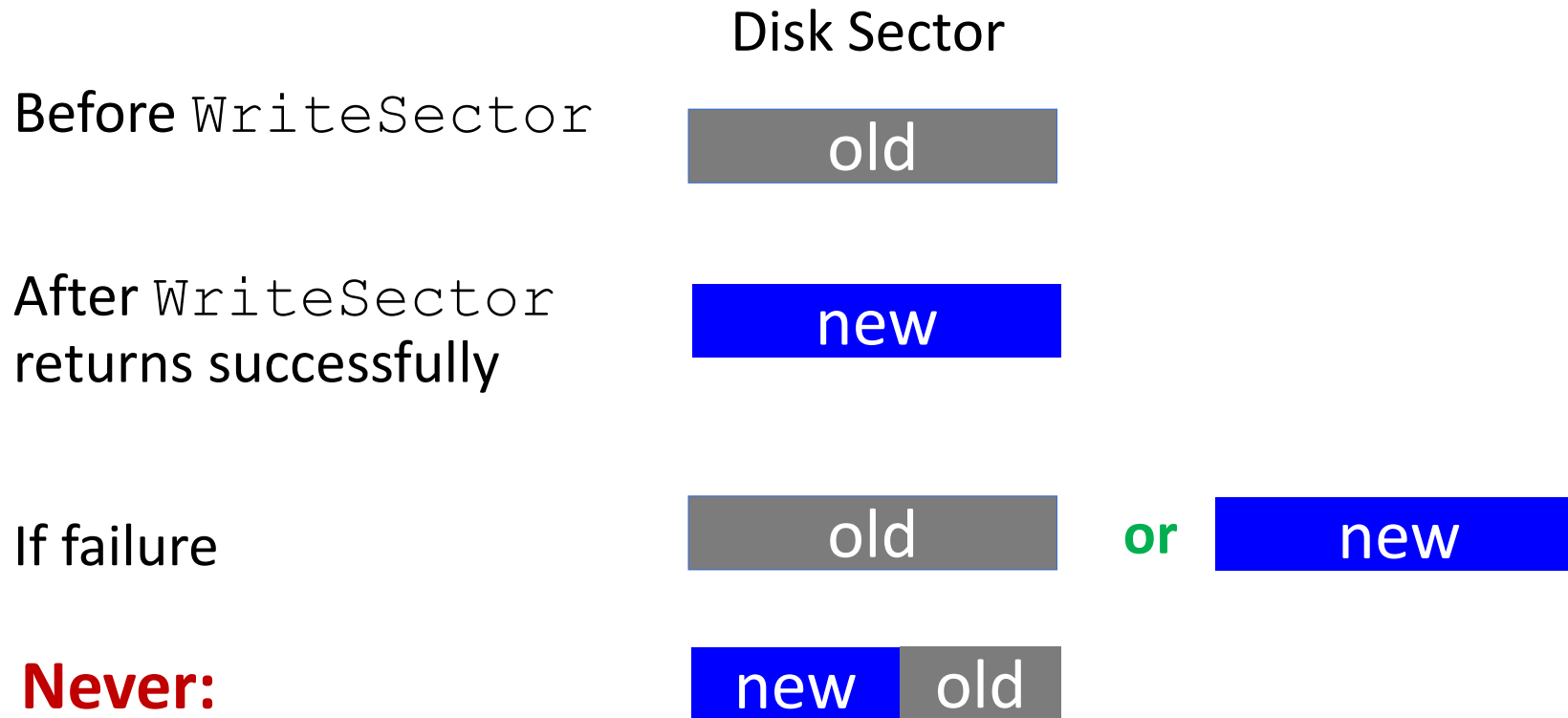
Assumption

A single sector disk write is atomic



Assumption

A single sector disk write is atomic



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

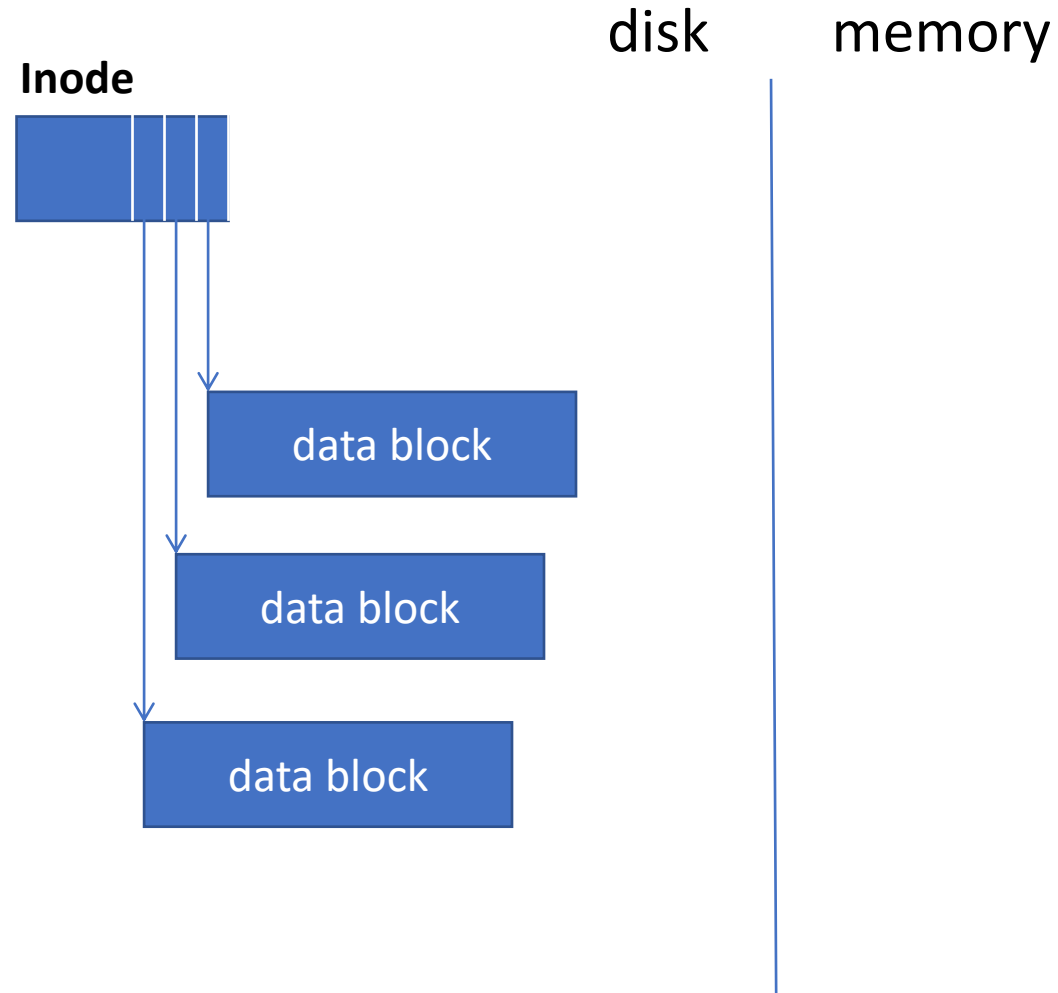
Write()s

- 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()

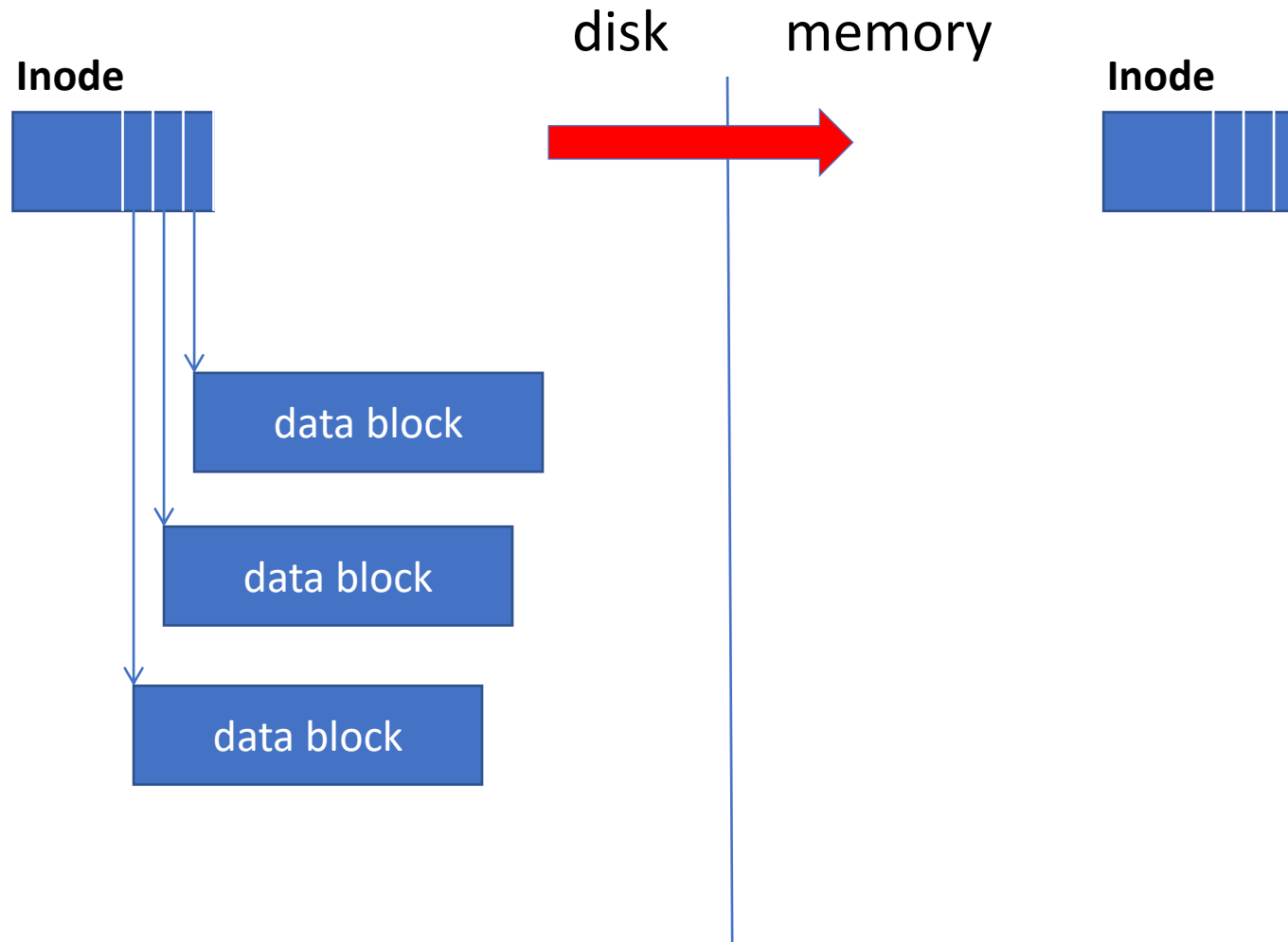
- Overwrite ***in-memory copy*** of Inode to ***disk*** Inode

Initial State



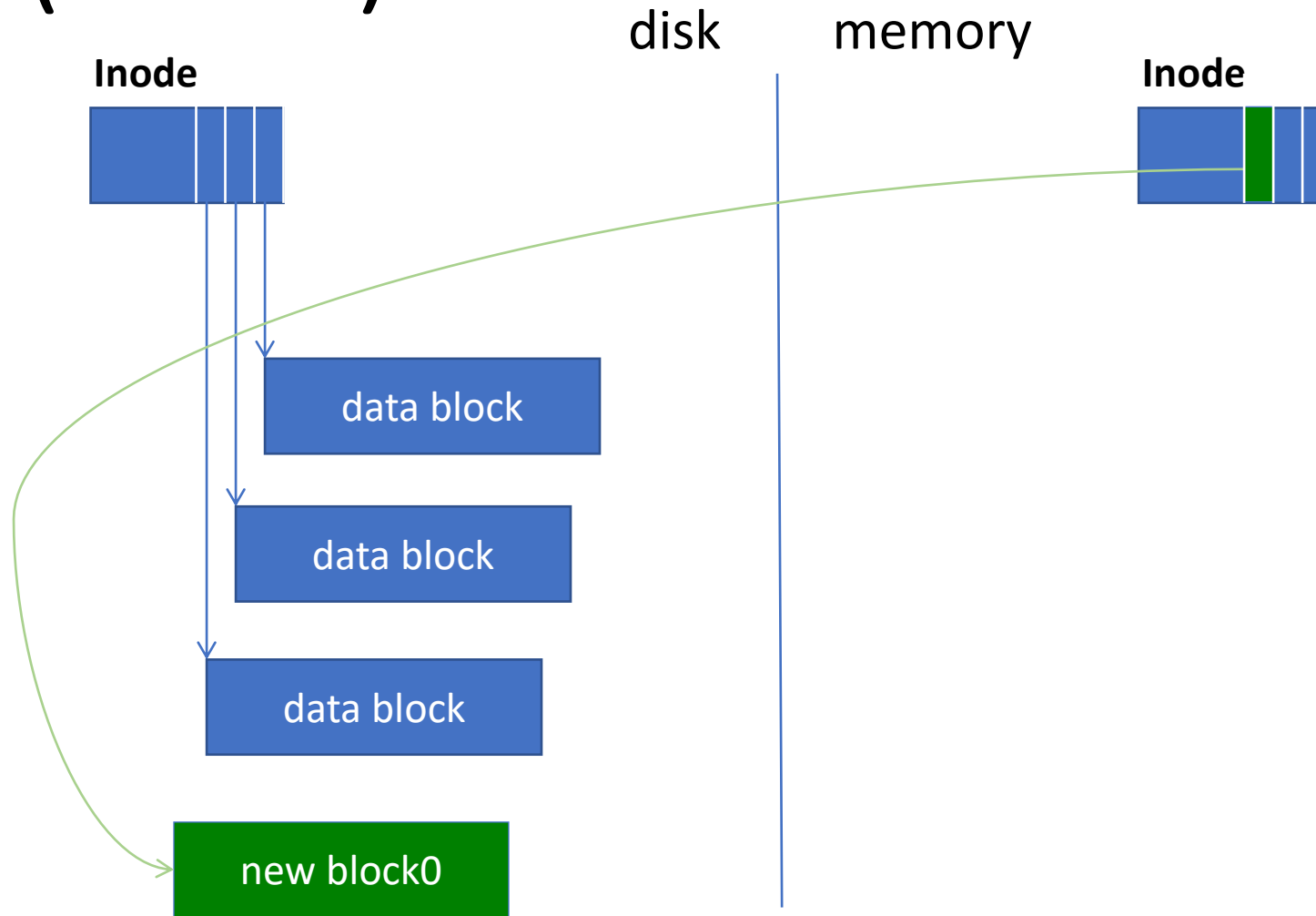
With ***write-through*** cache

Open()



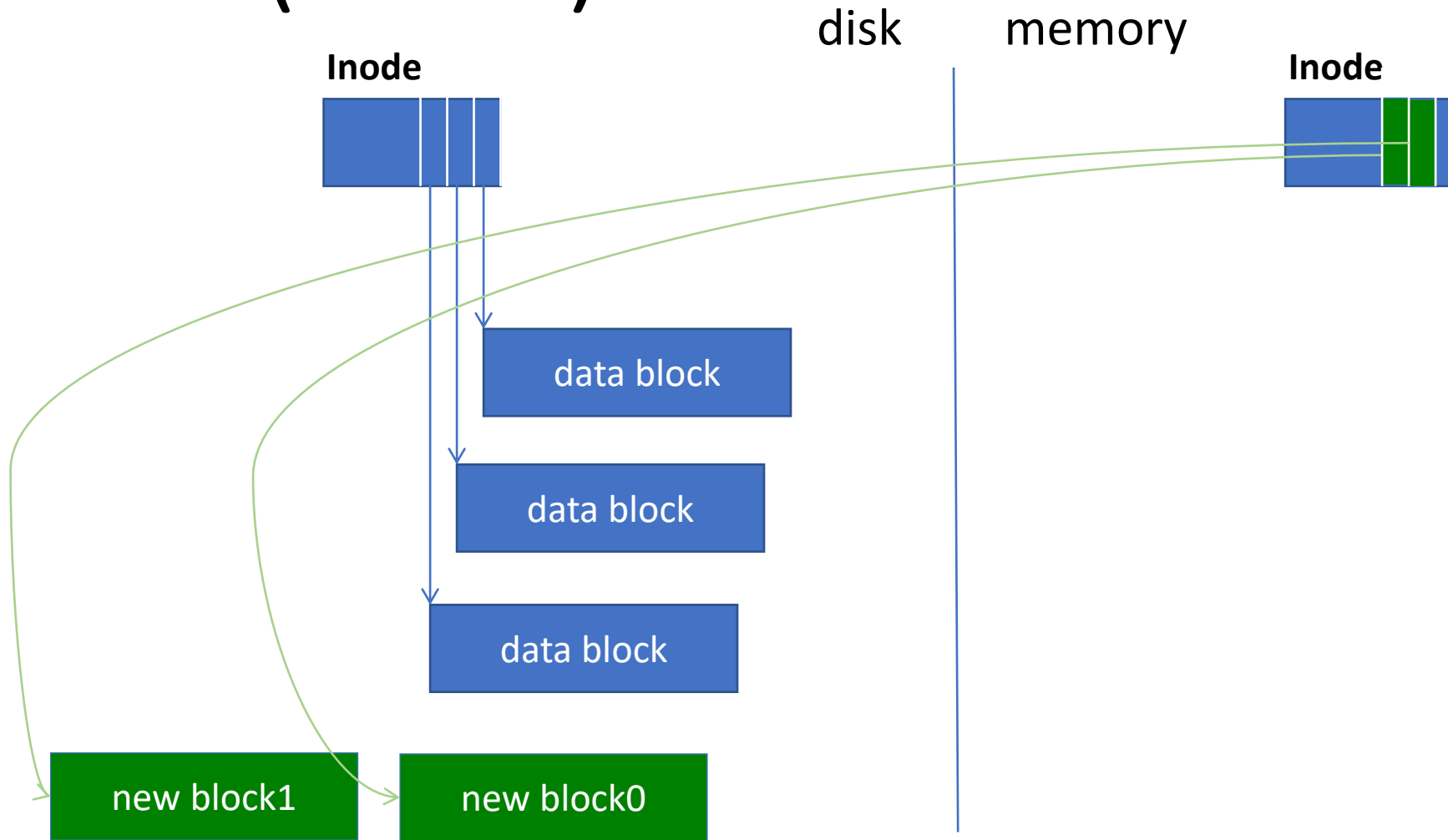
*With **write-through** cache*

Write(block 0)



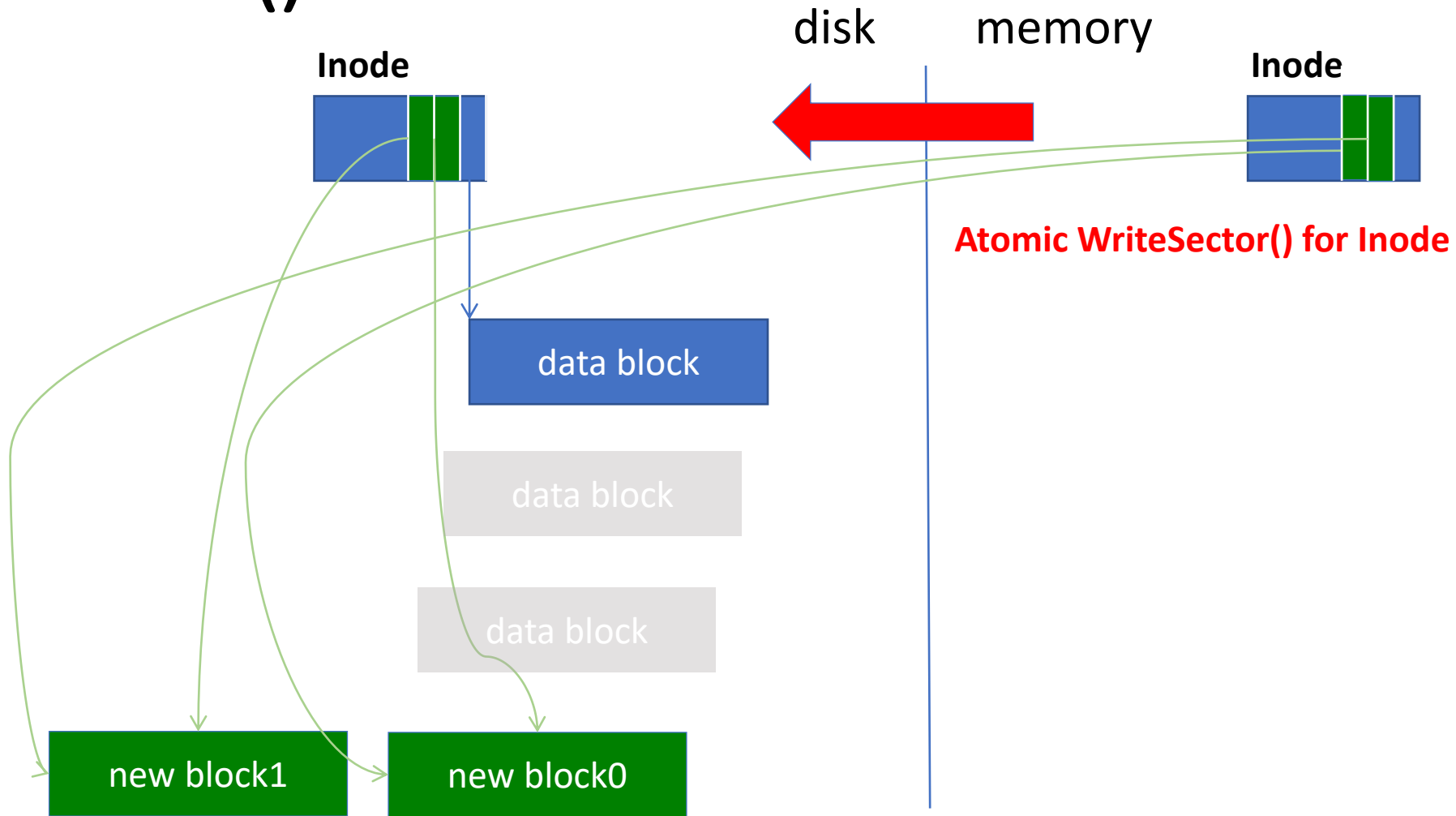
With ***write-through*** cache

Write(block 1)



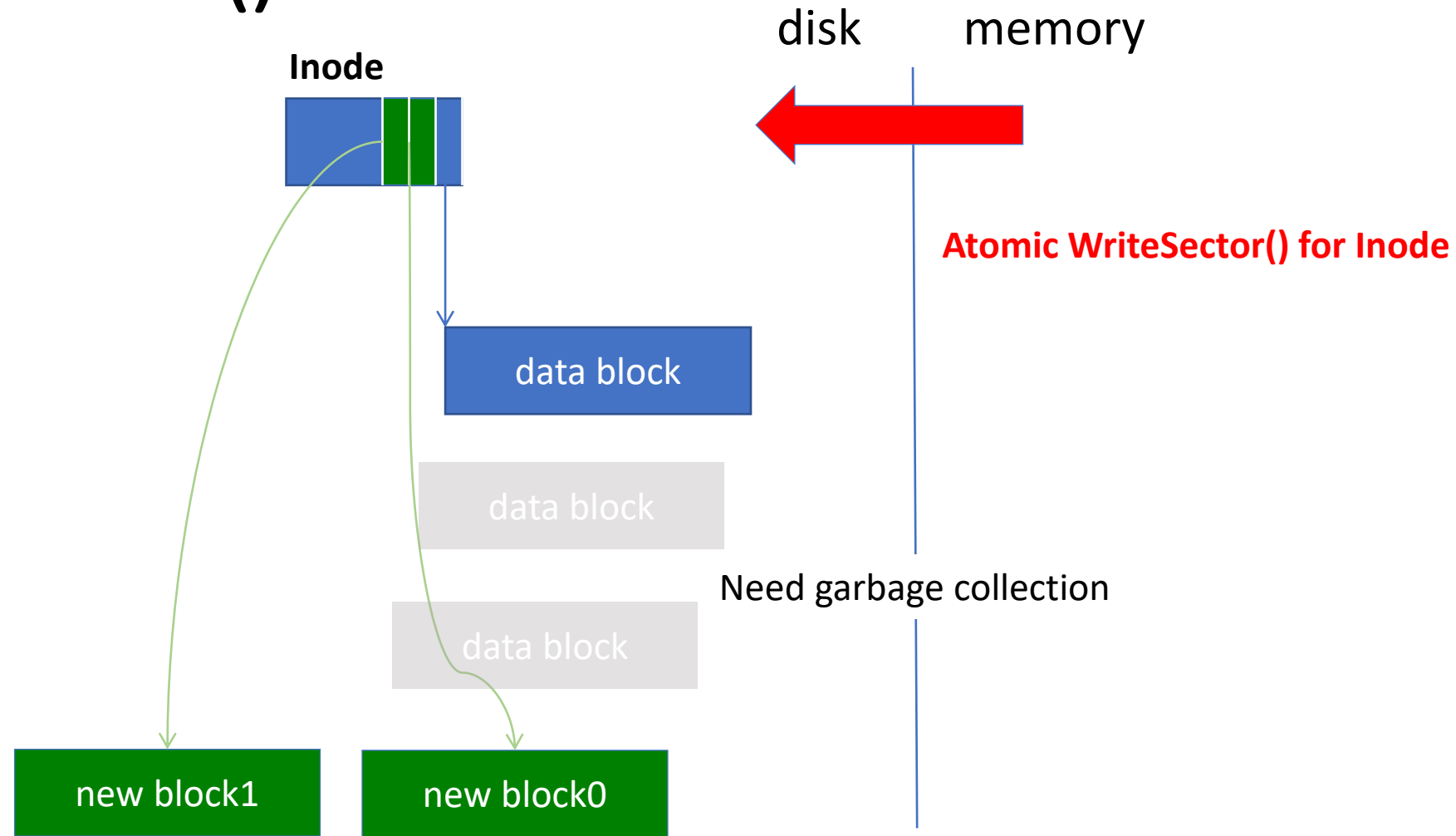
With ***write-through*** cache

Close()



*With **write-through** cache*

Close()



*With **write-through** cache*

How Shadow Paging Works (with Write-Behind)

Open()

- Read Inode from disk into Active File Table

Write()s

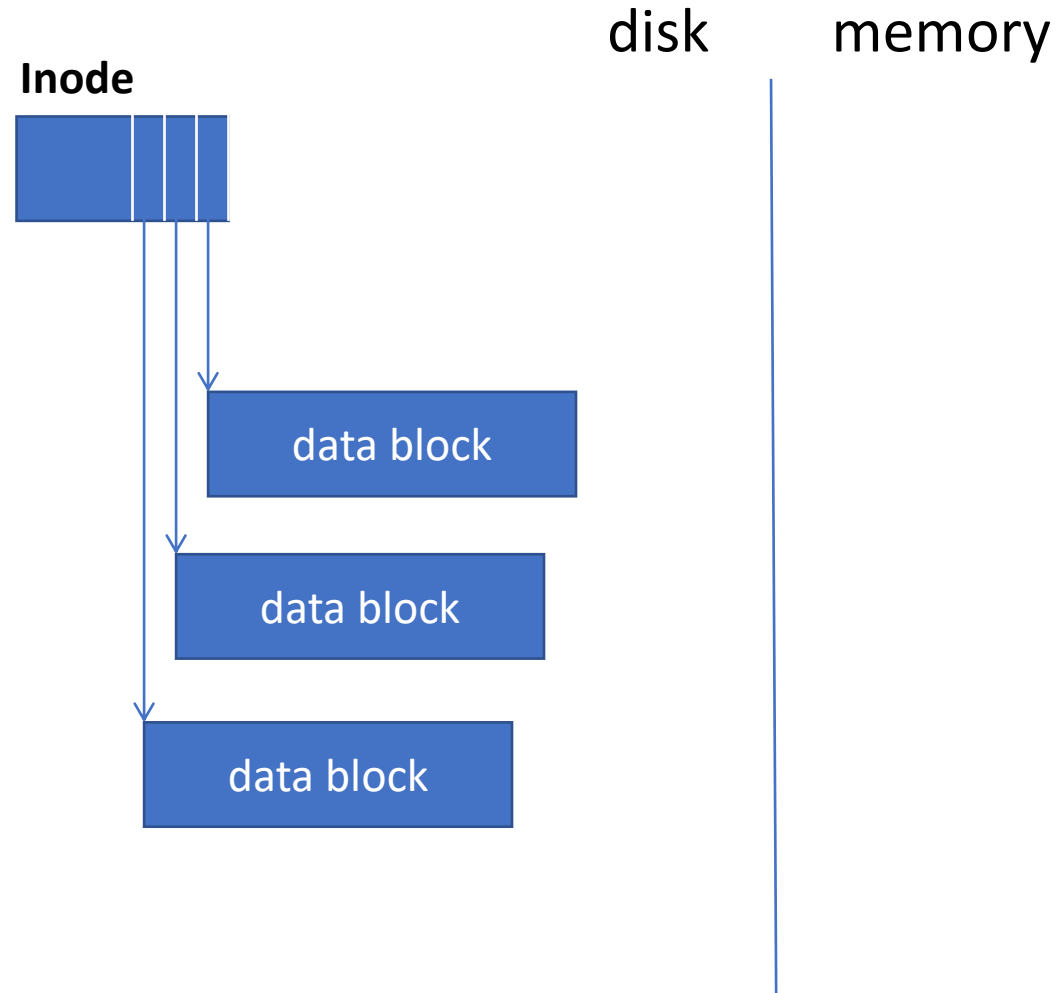
- Allocate new blocks for new data
- Write disk addresses to in-memory copy of Inode
- Write data blocks to cache (***don't have to write to disk yet***)

Close()

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

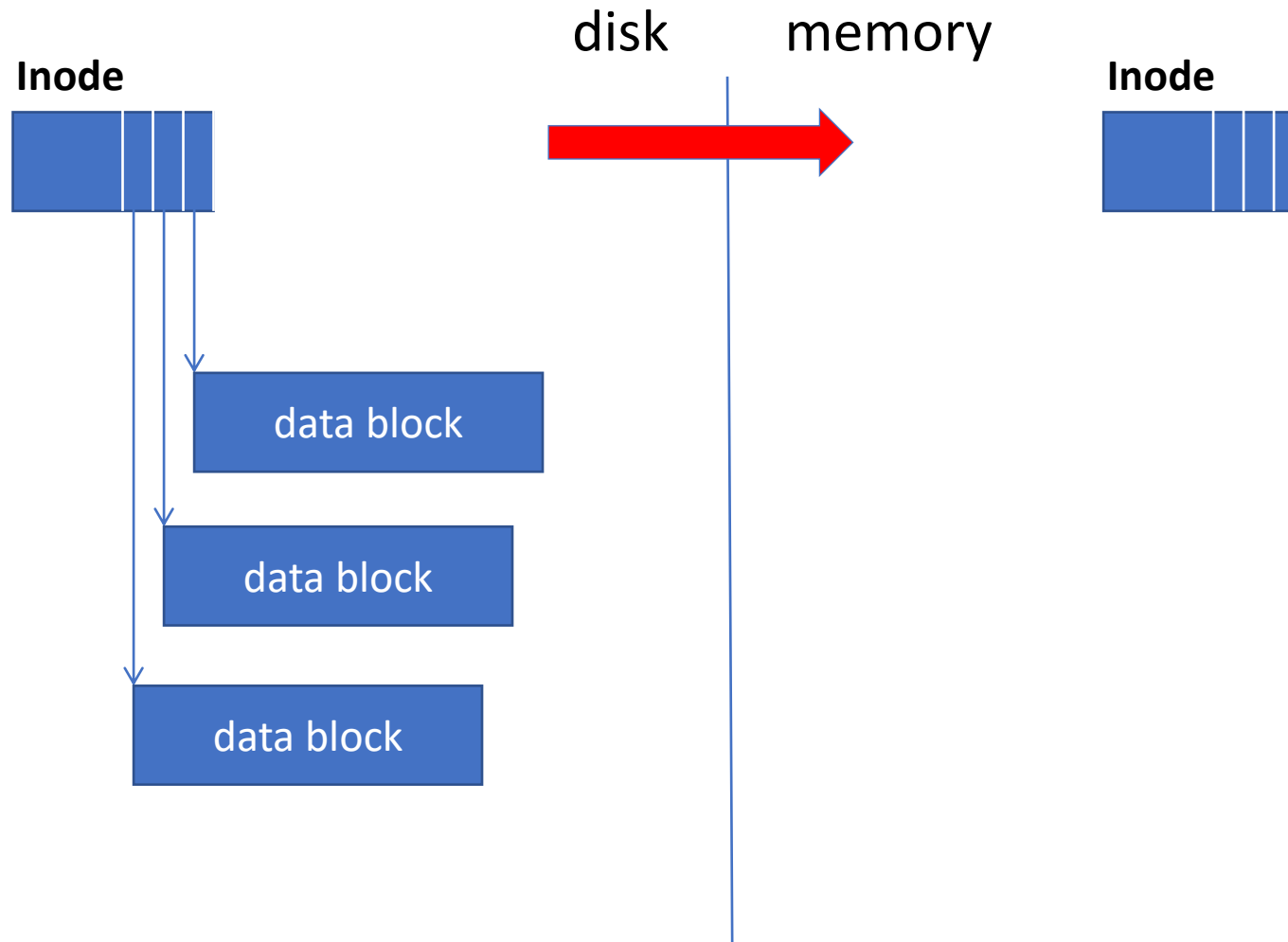
Inode write is atomic =>
FS on-disk state always consistent

Initial State



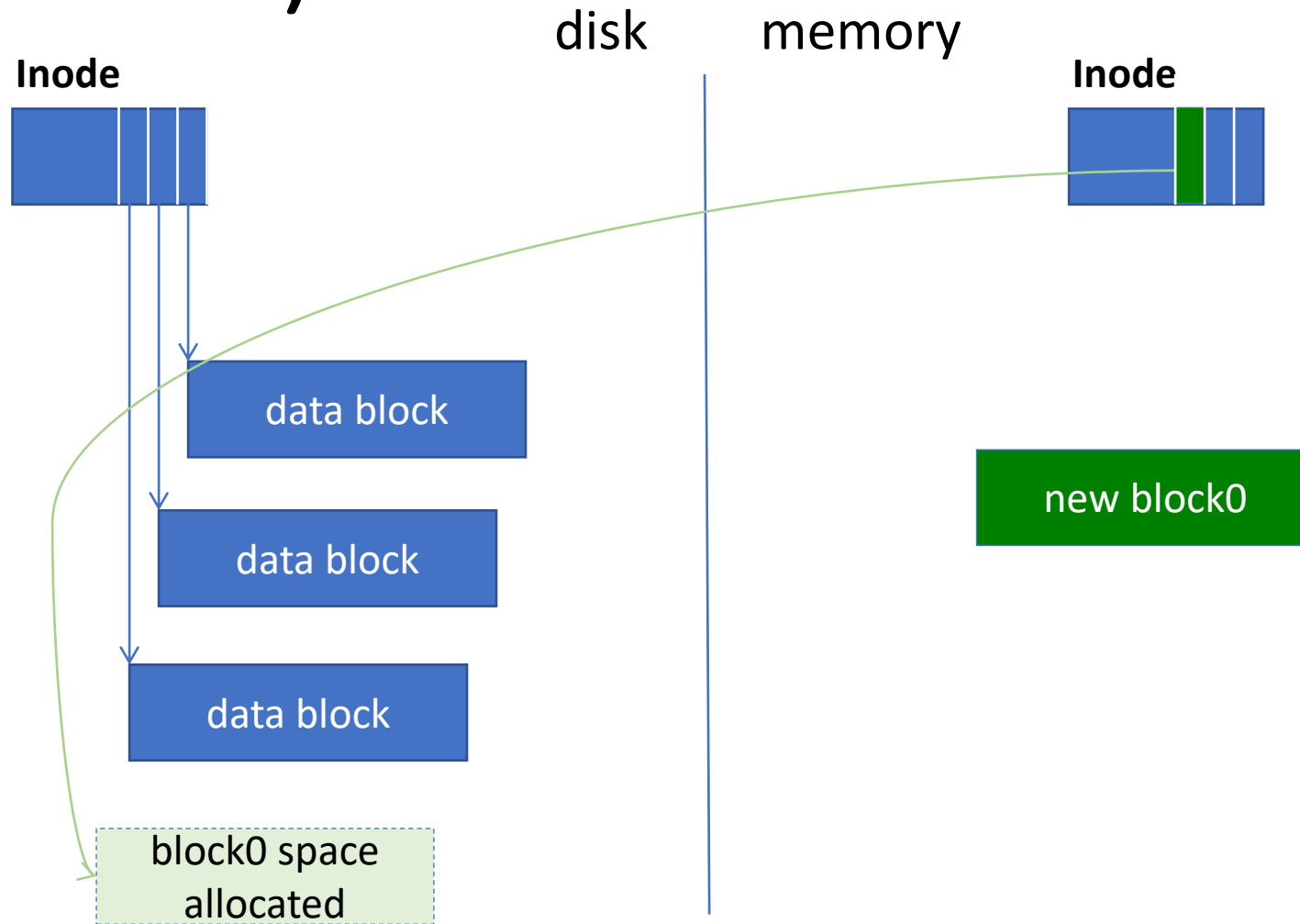
*With **write-behind** cache*

Open()



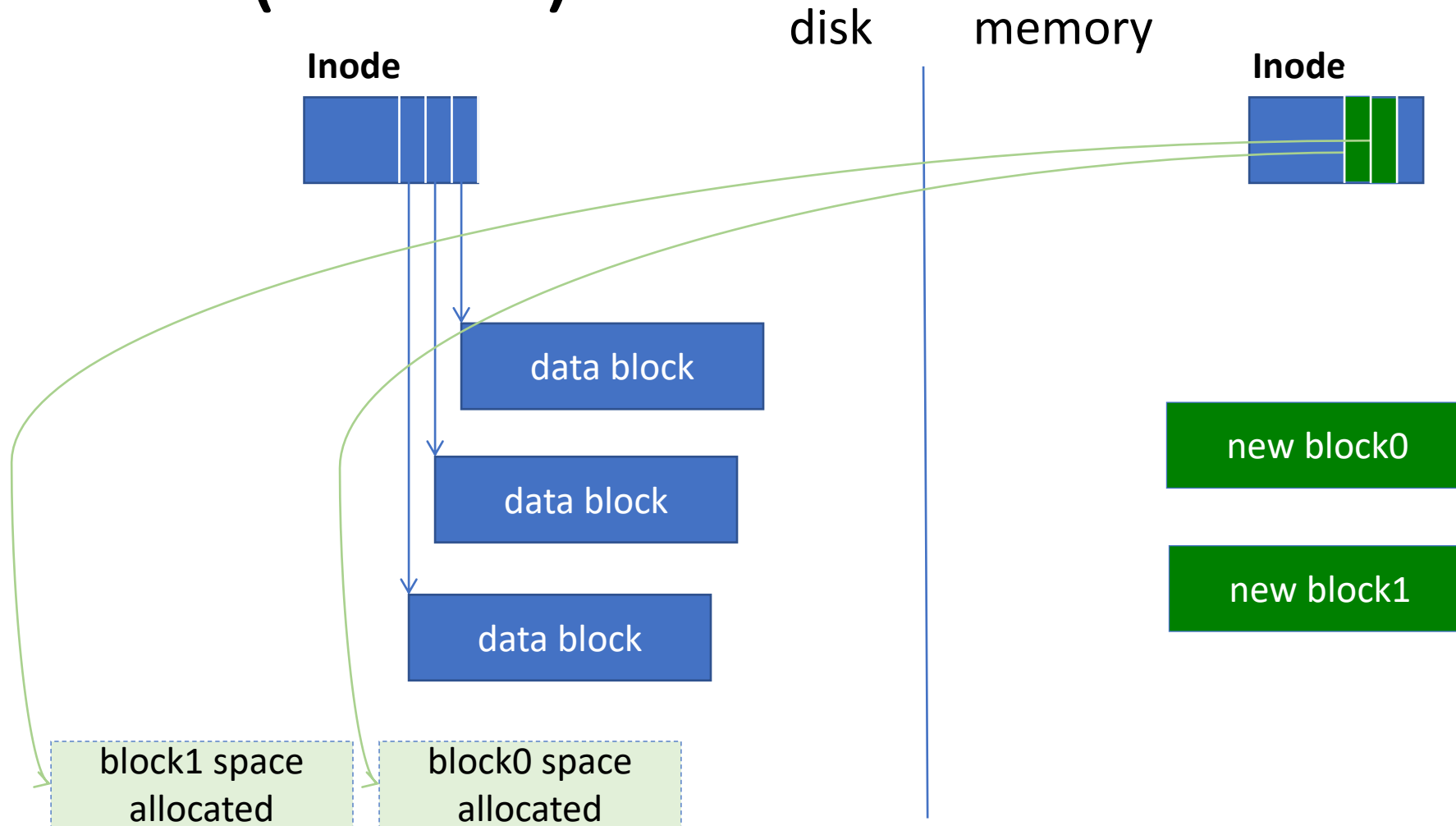
*With **write-behind** cache*

Write(block 0)



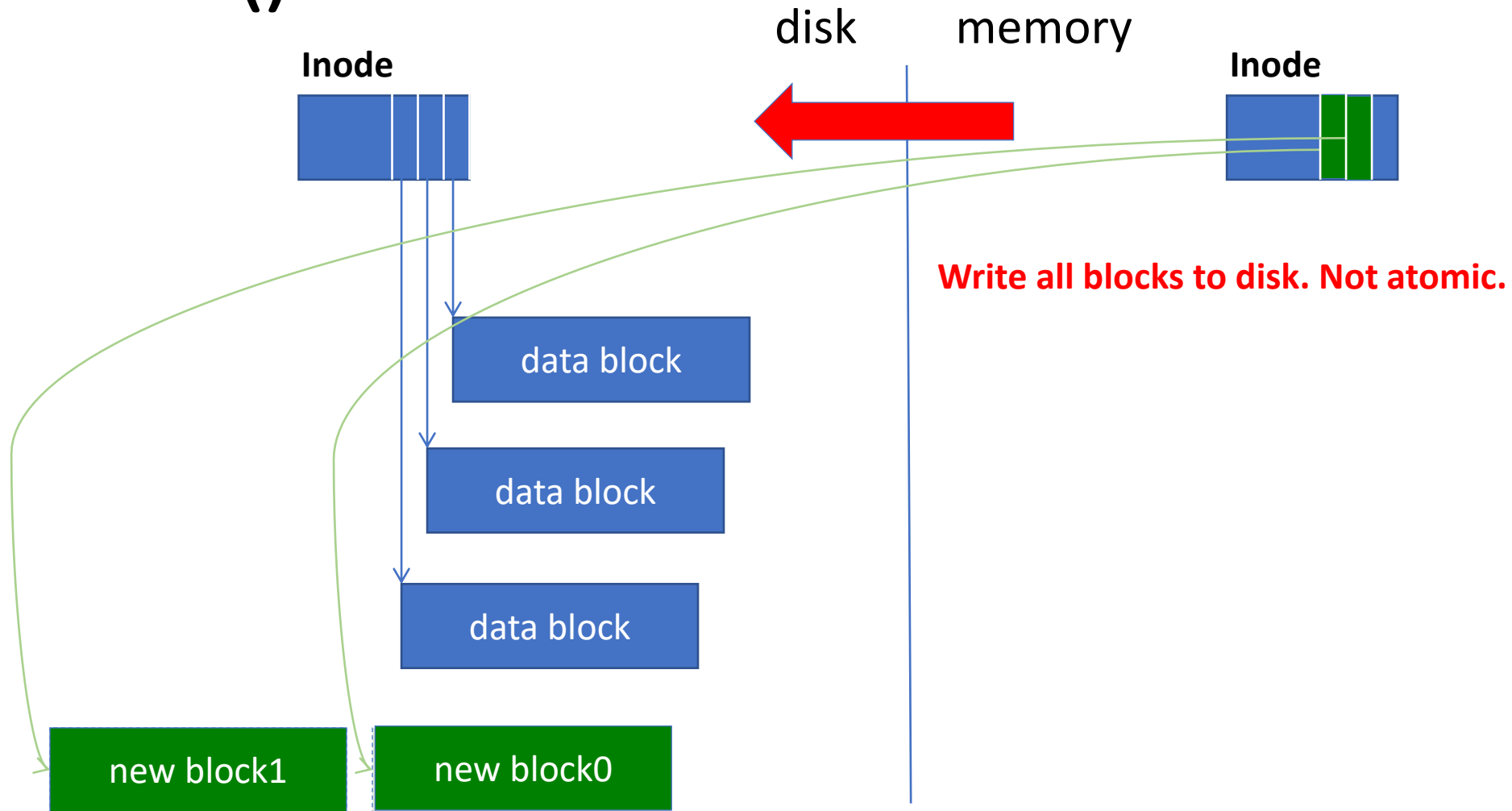
With ***write-behind*** cache

Write(block 1)



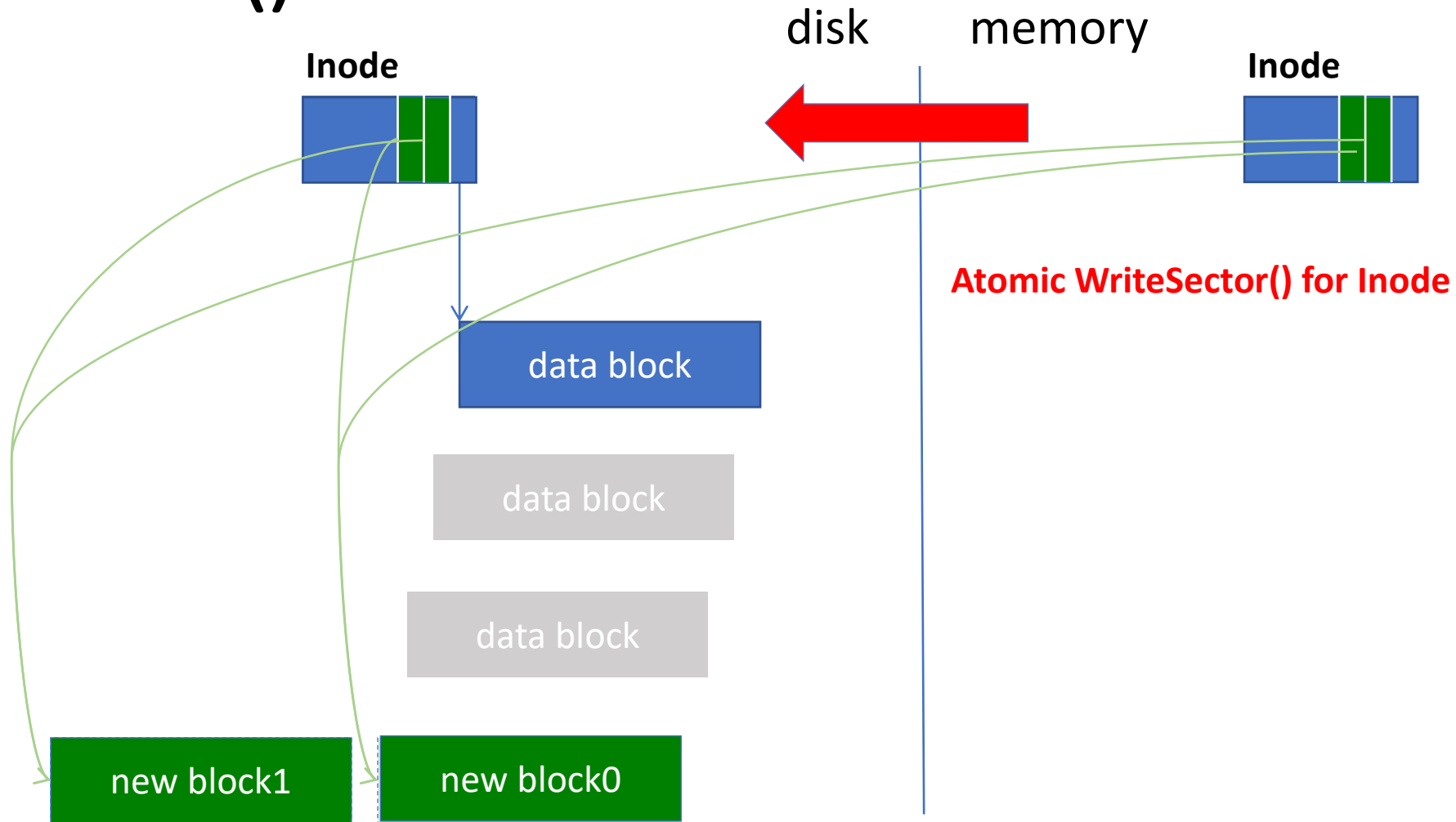
With ***write-behind*** cache

Close()



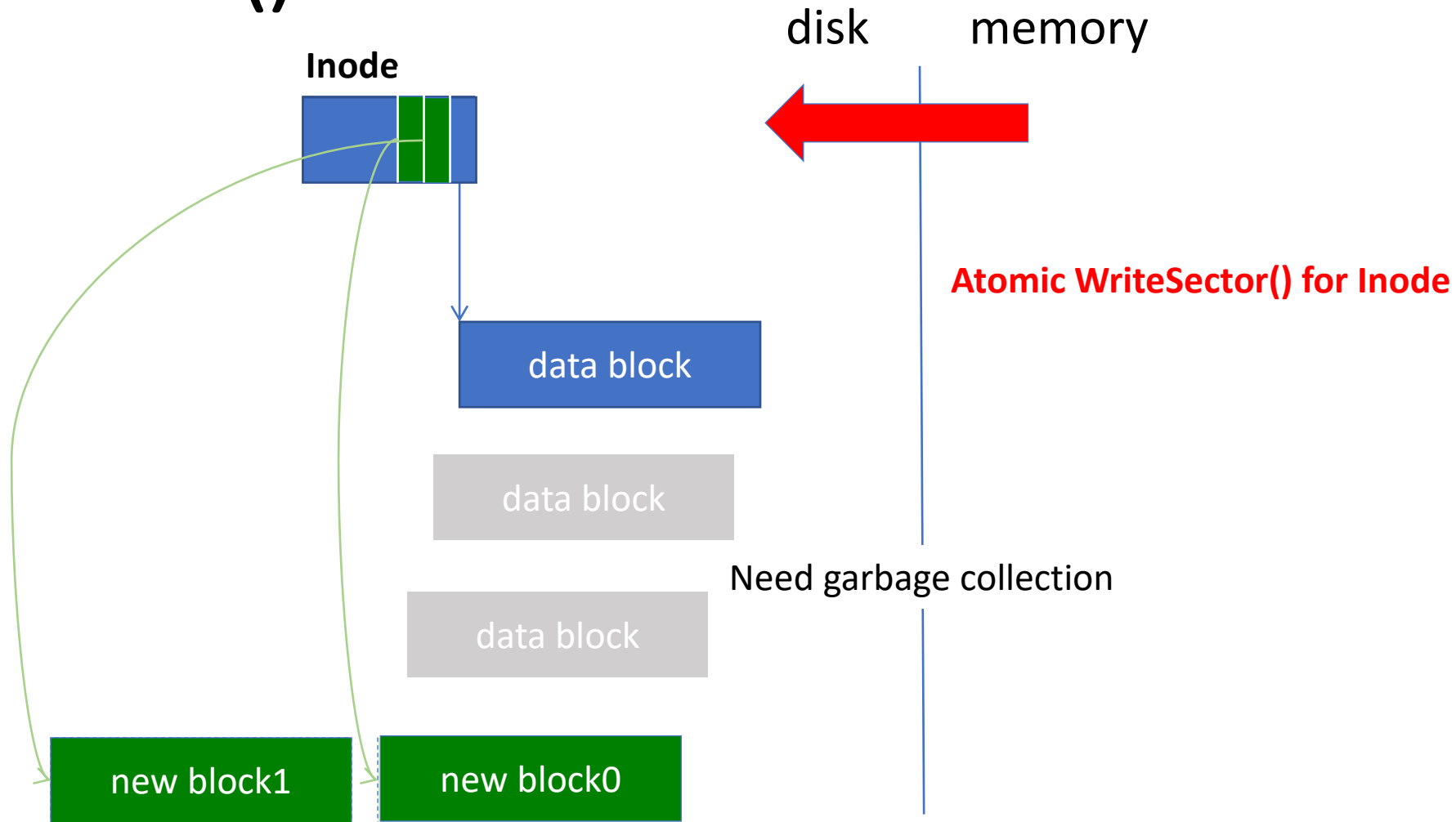
*With **write-behind** cache*

Close()



With ***write-behind*** cache

Close()



*With **write-behind** cache*

What happens to old blocks?

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



data block

Approach 2: Intentions Log

- Reserve an area of disk for (intentions) log

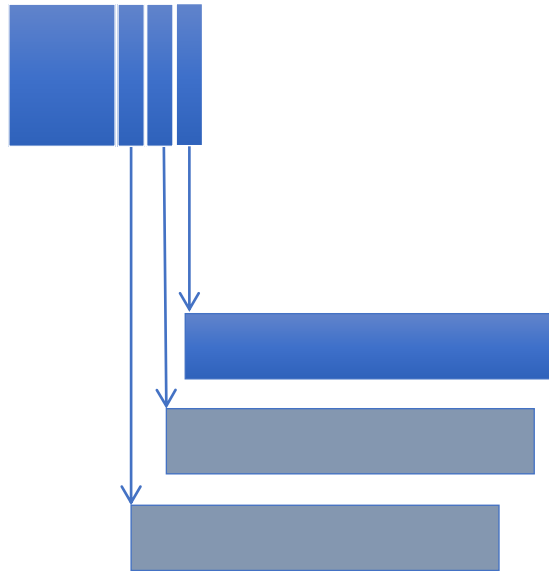
During Normal Operation

On Write():

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

Initial State

Inode



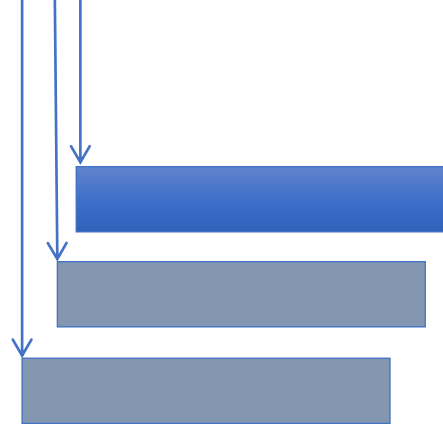
memory

disk

Log (on-disk)

Open()

Inode



memory

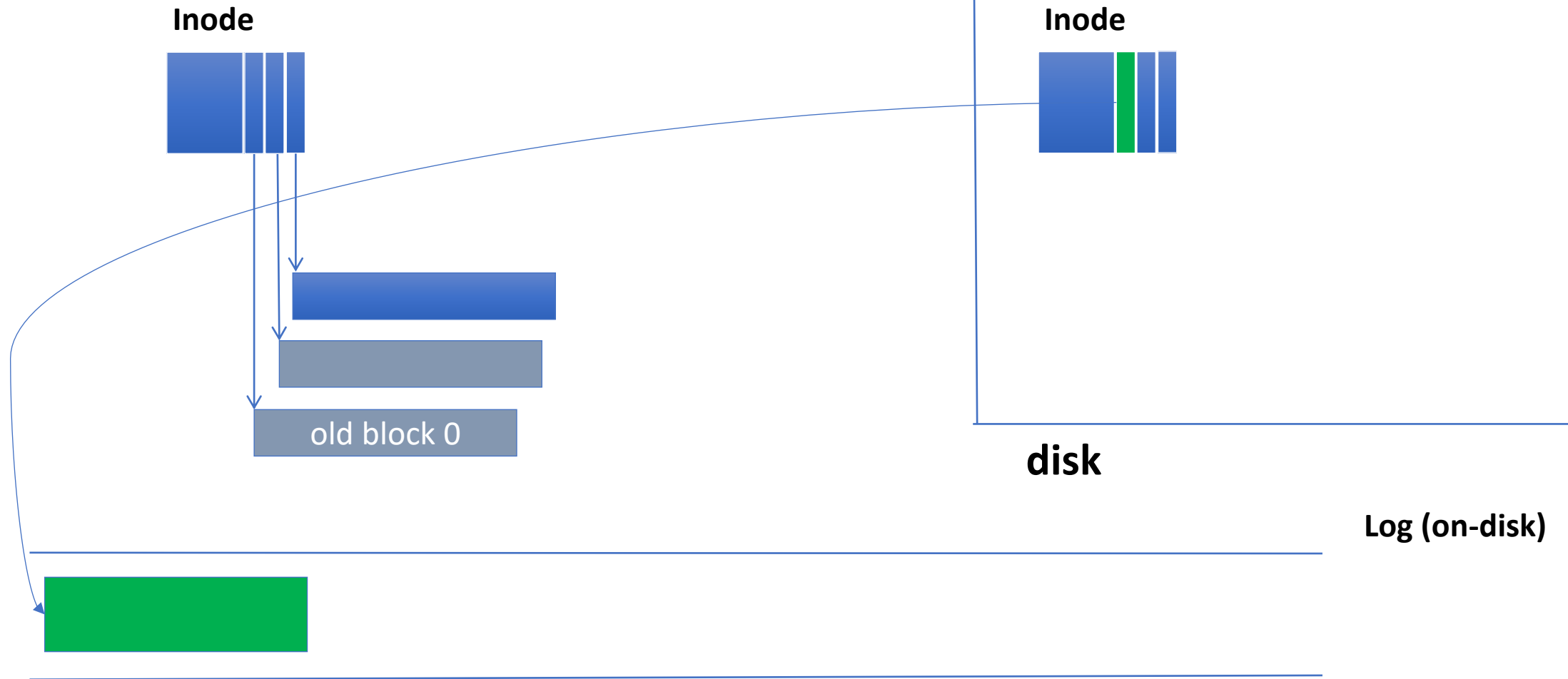
Inode



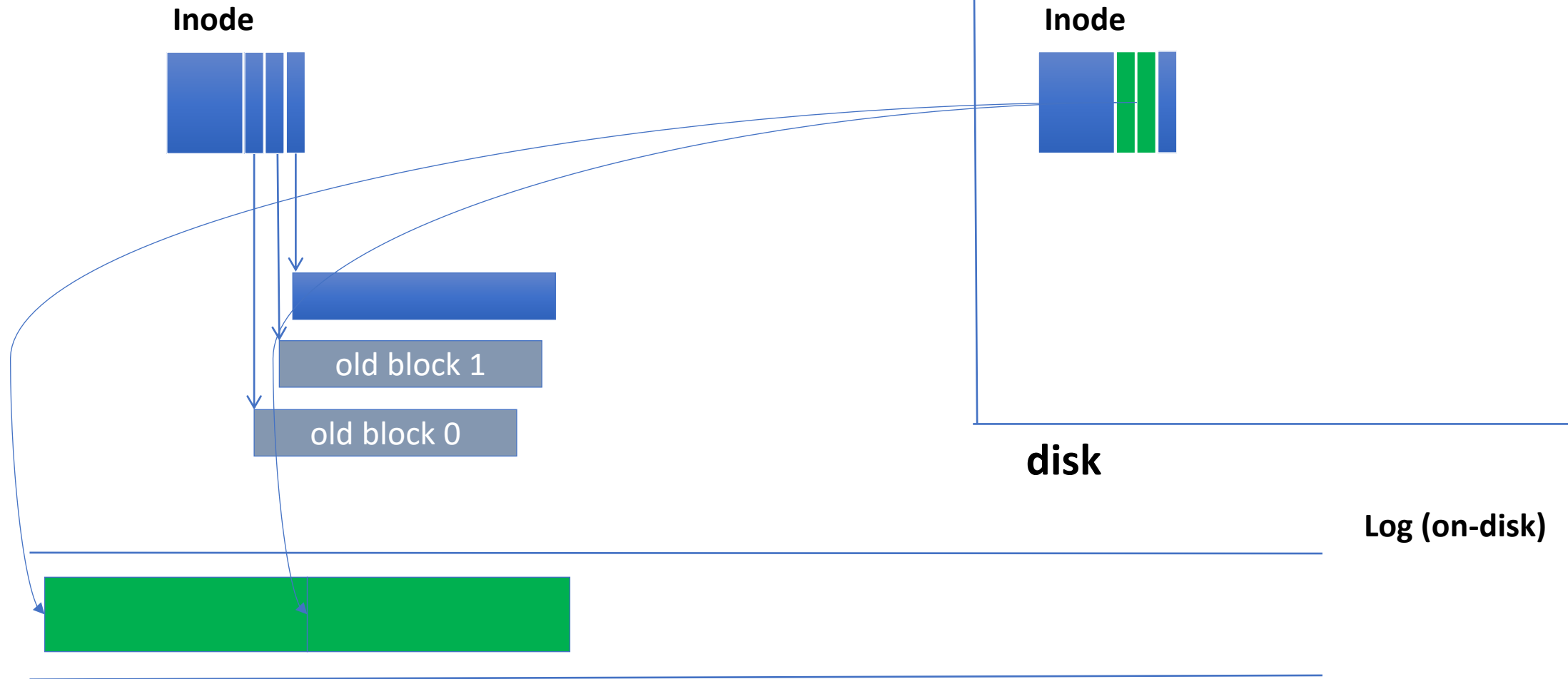
disk

Log (on-disk)

Write(block0)



Write(block1)



During Normal Operation

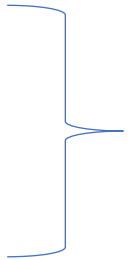
On Close():

- 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

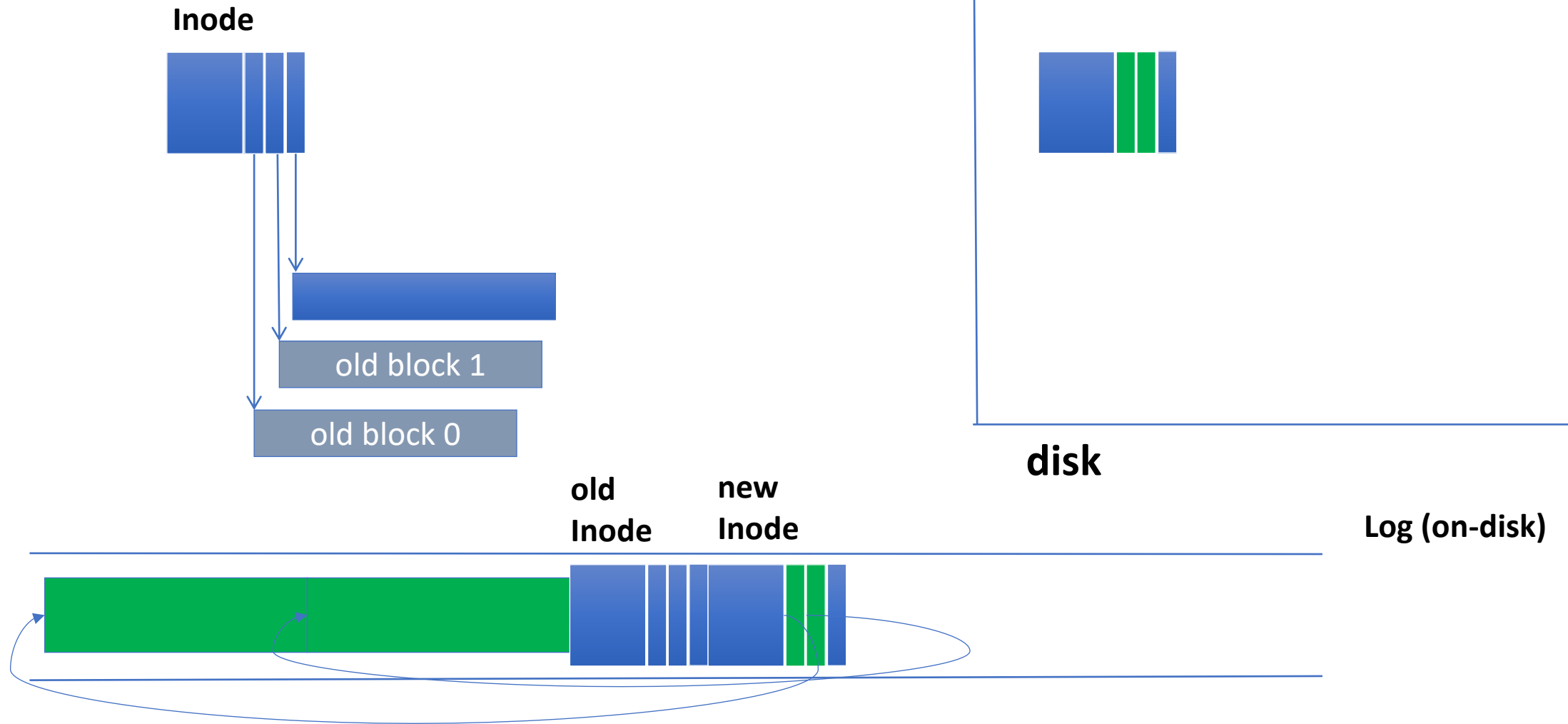
On Close():

- 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

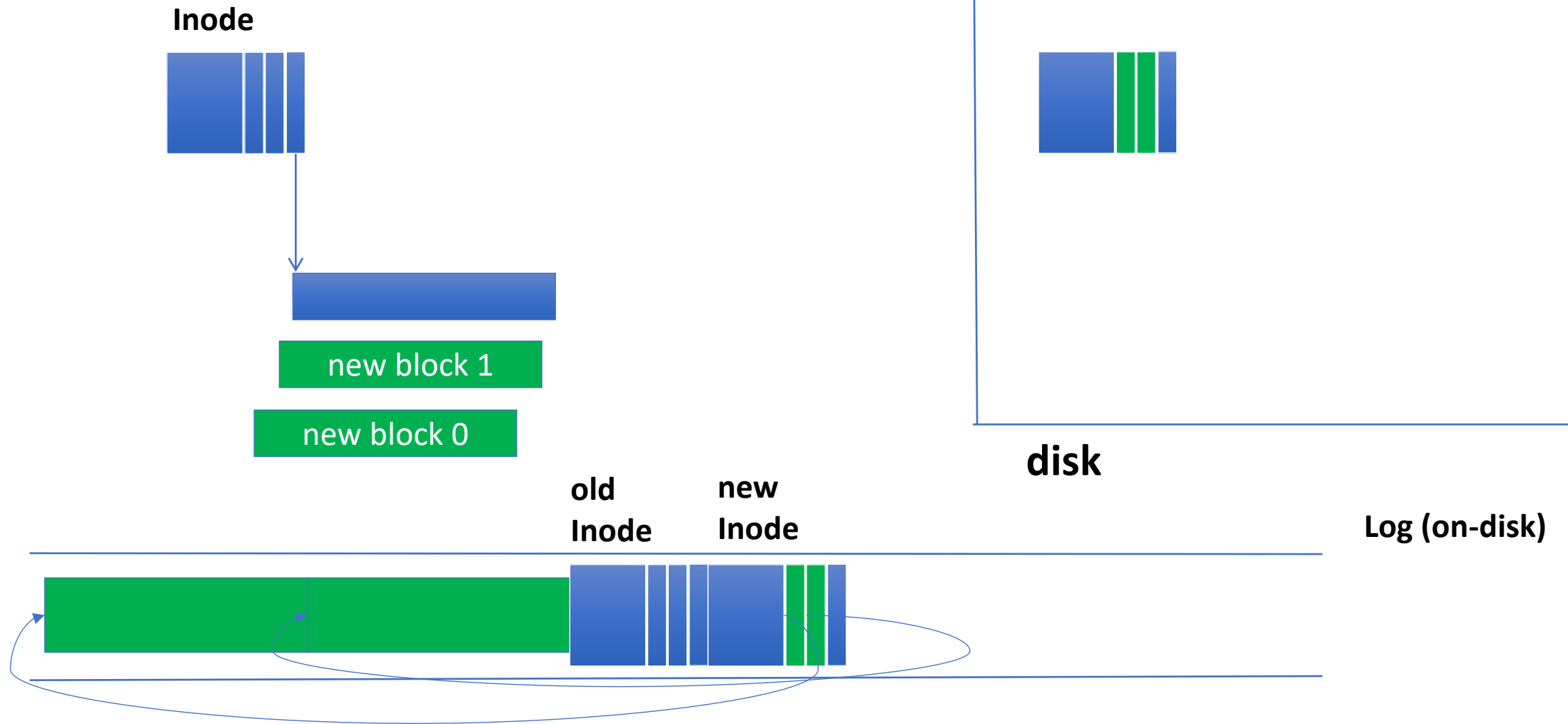


Do later,
In the background

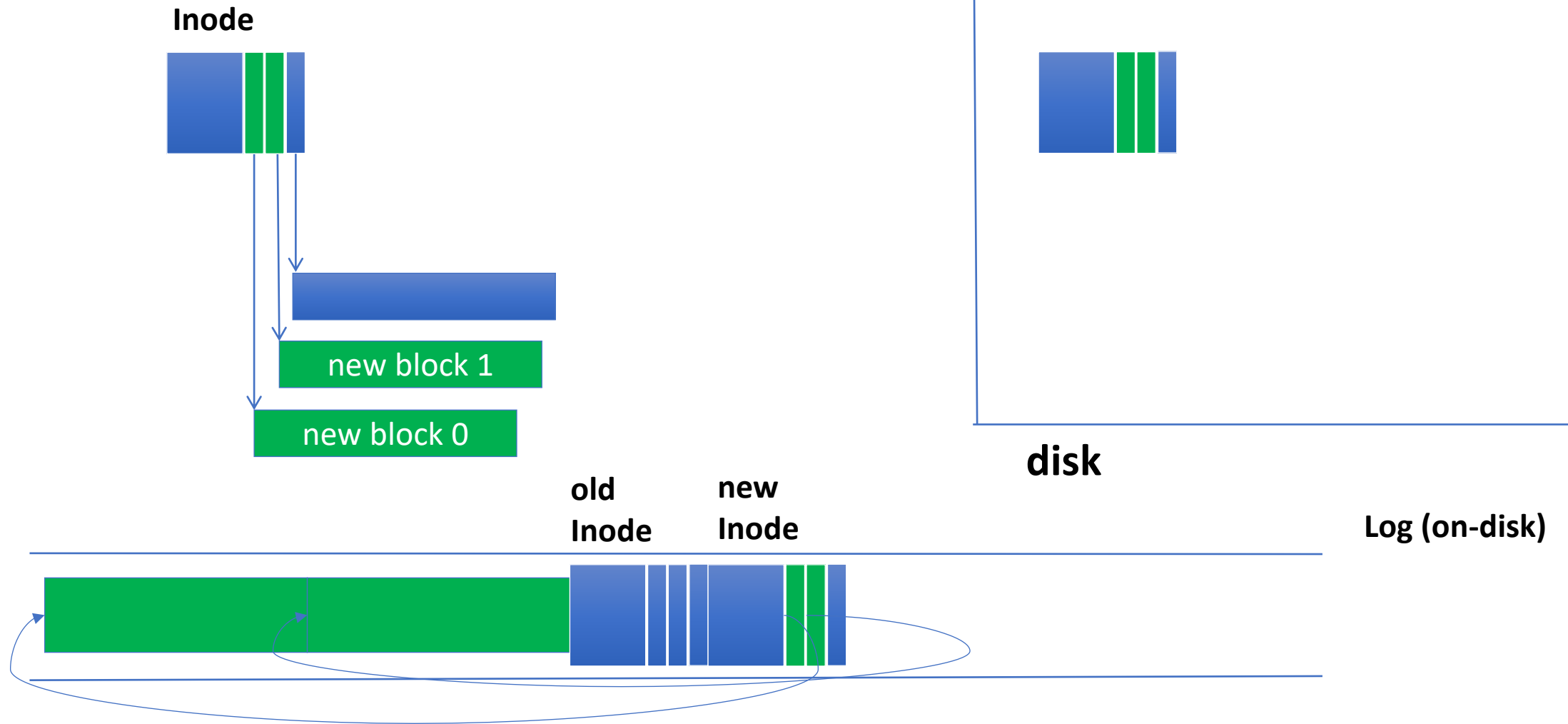
Close()



Later, Step 1



Later, Step 2



Later, Step 3

Inode



new block 1

new block 0

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

Note that this means “the file system state refers to the new/old copy.”

We might have to do some recovery work as in previous slides.

Maybe both copies are on disk somewhere, but one is now garbage.

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 – when cache replacement (can't avoid)
- ✓ Write from log to data – when disk is idle (never a bottleneck)

Surprisingly, Shadow Paging works less well

- ☹ Disk allocation gets messed up – need to “be smart” twice
- ☹ Random access pattern – shadow pages can and will be anywhere

Further Reading

Operating Systems: Three Easy Pieces by R. & A. Arpaci-Dusseau

Chapters 38, 43.

<https://pages.cs.wisc.edu/~remzi/OSTEP/>

Credits:

Some slides adapted from the OS courses of Profs. Remzi and Andrea Arpaci-Dusseau (University of Wisconsin-Madison), Prof. Willy Zwaenepoel (University of Sydney), and Prof. Youjip Won (Hanyang University).