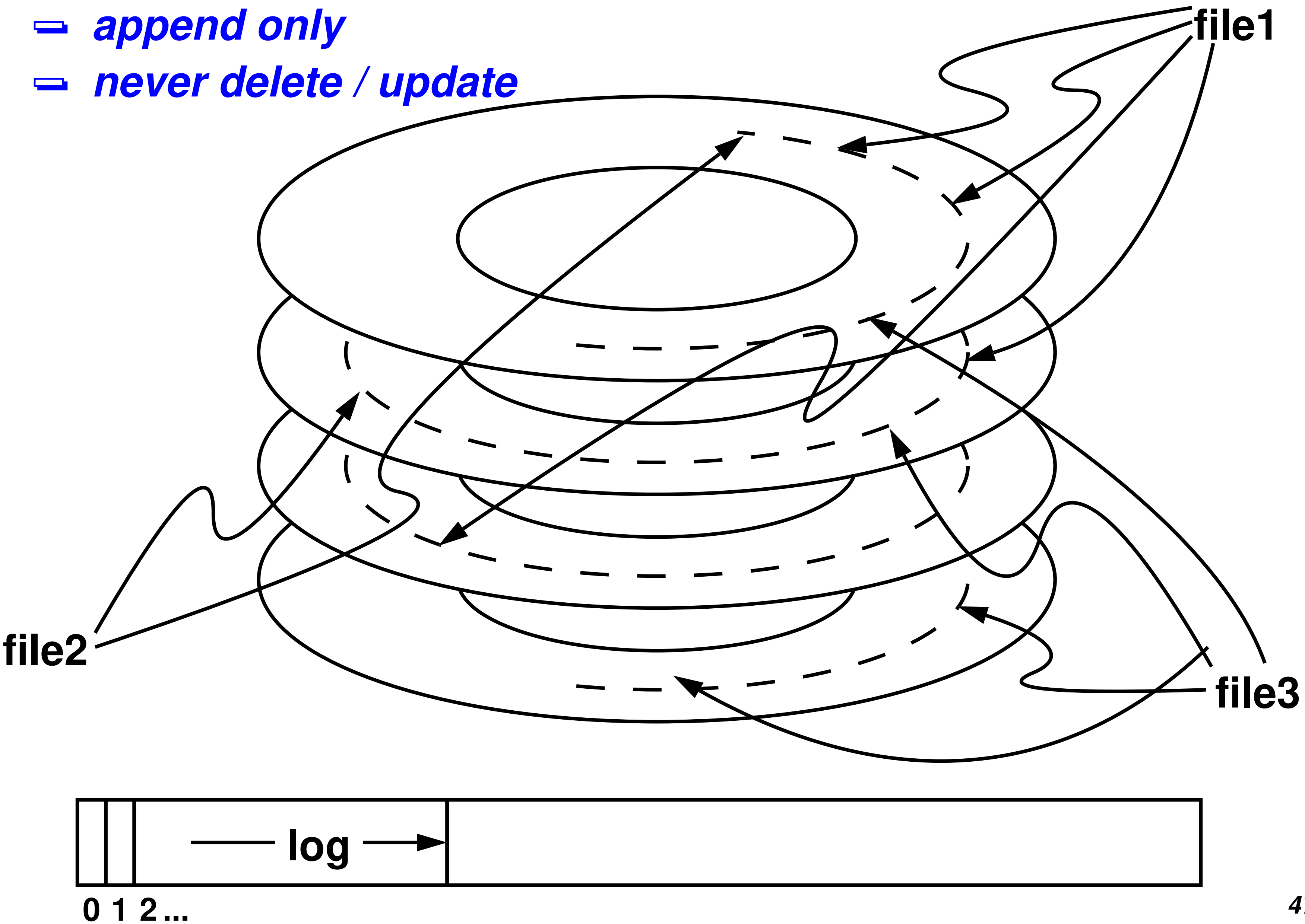
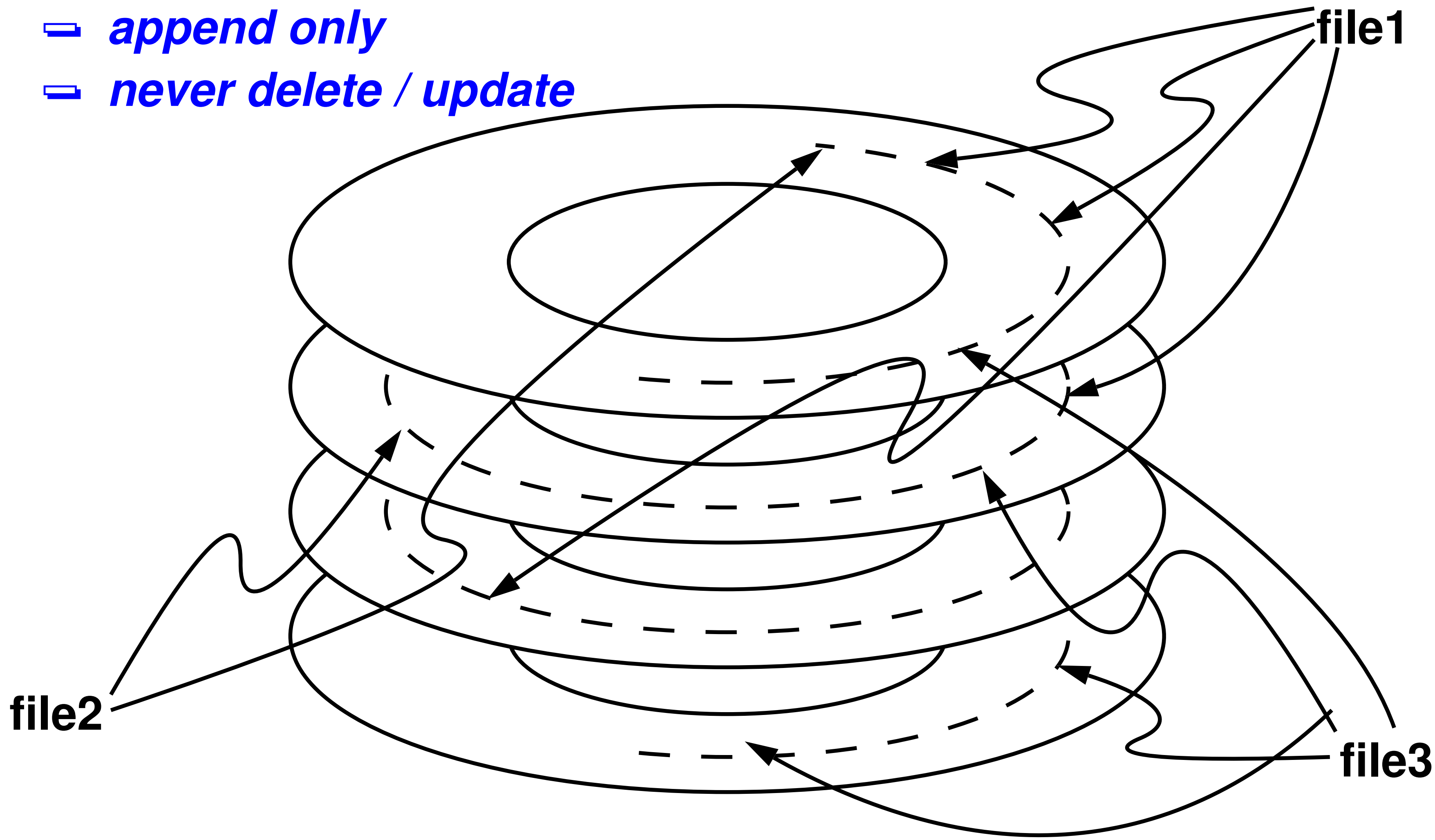
Log-Structured File Systems

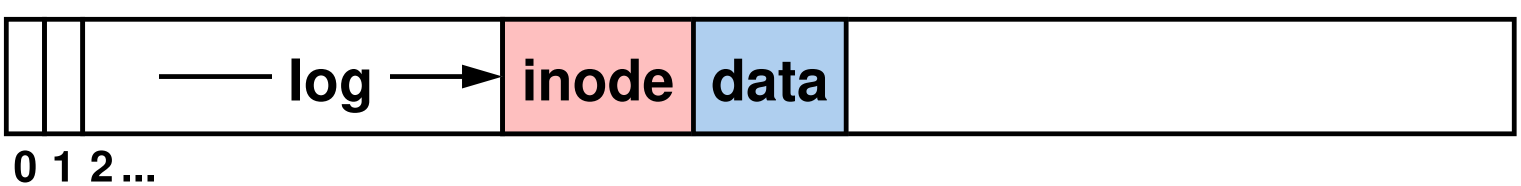
- **Main principles**
  - *append only*
  - *never delete / update*



Log-Structured File Systems

- **Main principles**
  - *append only*
  - *never delete / update*



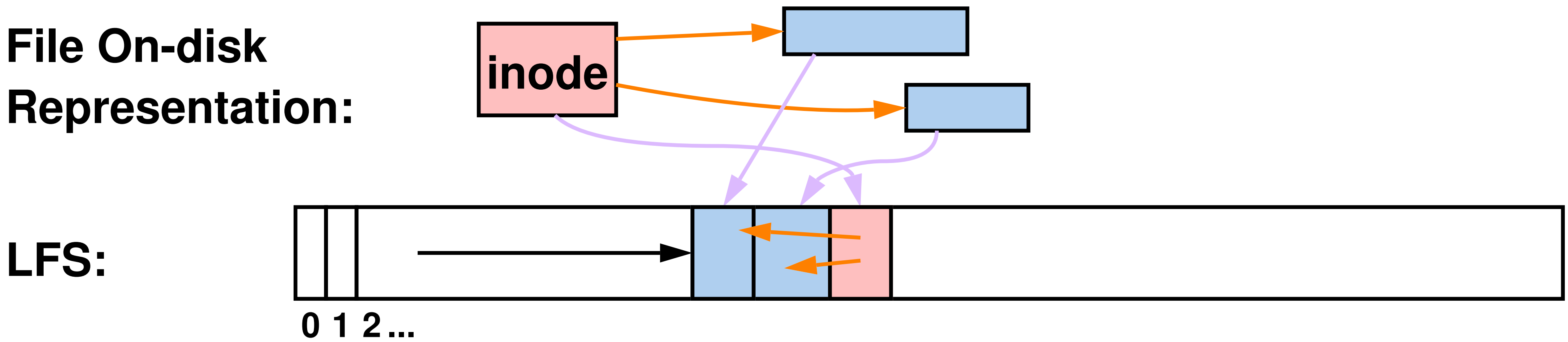


Log-Structured File Systems

- How does "append only" and "never delete / update" help with *write* performance?
  - minimize seek latency
    - one seek followed by many many writes
  - minimize rotational latency
    - write a cylinder at a time

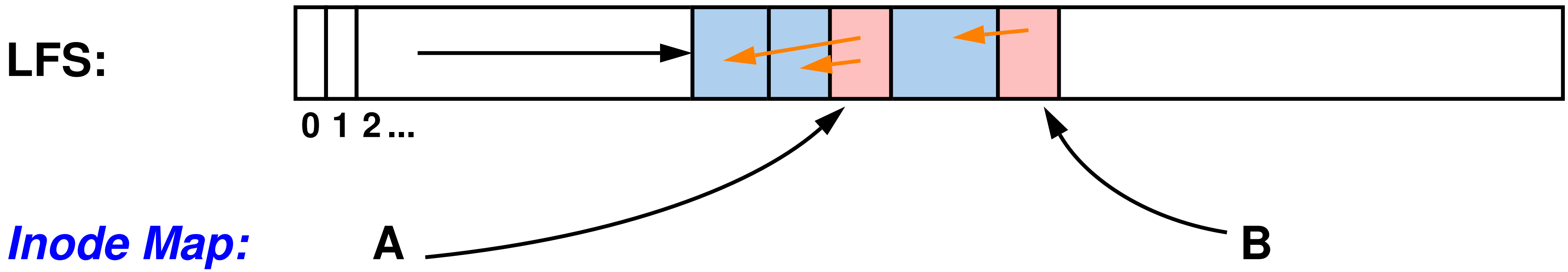
- *Sprite FS* (a log-structured file system)
  - through batching, a single, long write can write out everything

LFS Data Placement Example



LFS Data Placement Example

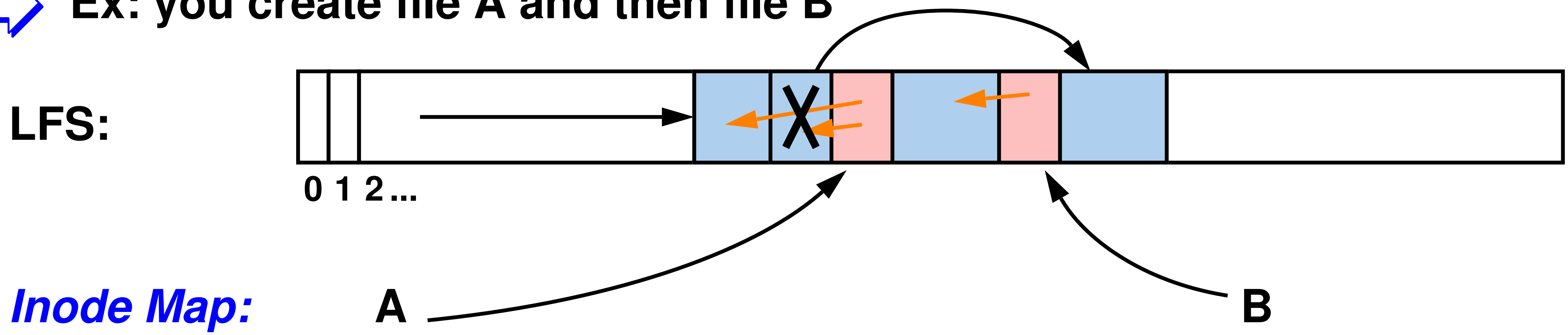
- What happens if you want to modify the file?
  - how does "append-only" really work?
- Ex: you create file A and then file B



  - you modify file A, e.g., append to the last block of file A

LFS Data Placement Example

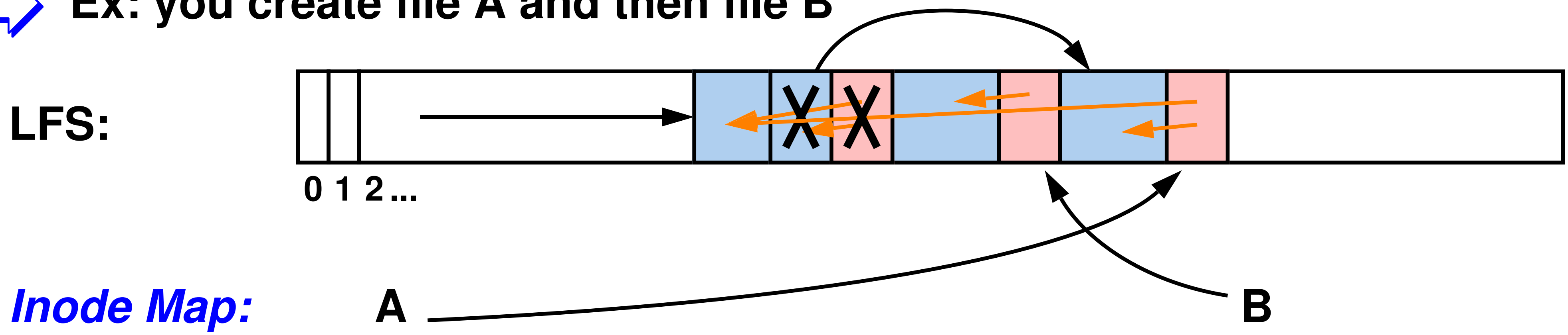
- What happens if you want to modify the file?
  - how does "append-only" really work?
- Ex: you create file A and then file B



  - you modify file A, e.g., append to the last block of file A

LFS Data Placement Example

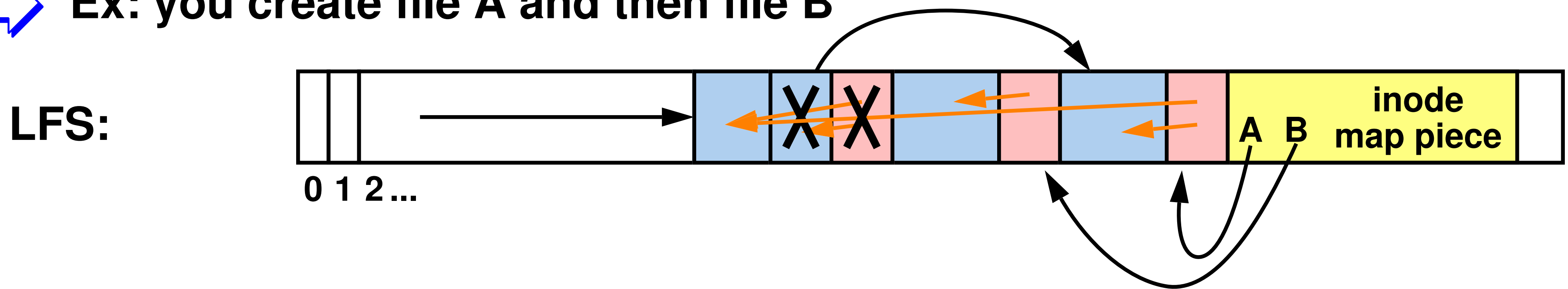
- What happens if you want to modify the file?
  - how does "append-only" really work?
- Ex: you create file A and then file B



  - you modify file A, e.g., append to the last block of file A
  - the updated file is still file A
    - but the inode has changed

LFS Data Placement Example

- What happens if you want to modify the file?
  - how does "append-only" really work?
- Ex: you create file A and then file B

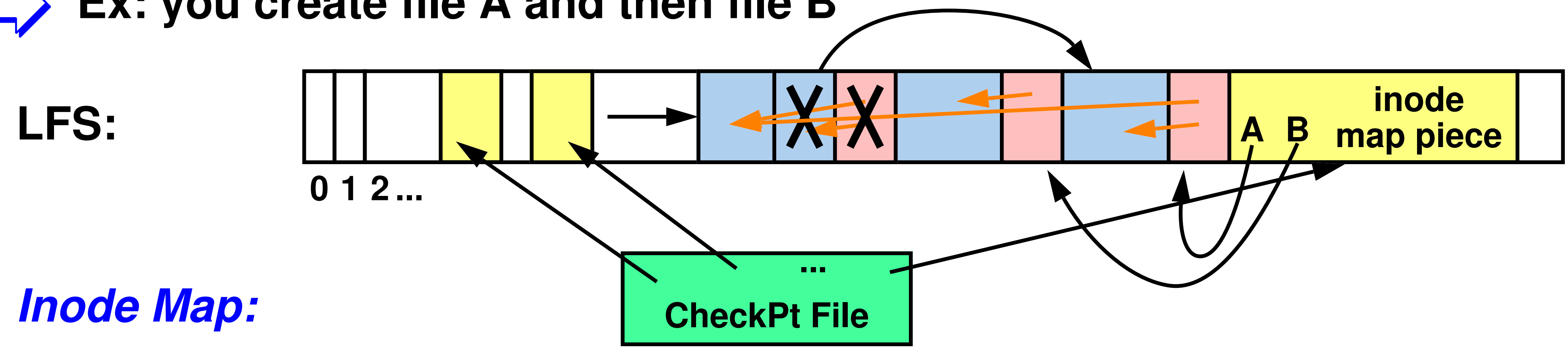


Inode Map:

- you modify file A, e.g., append to the last block of file A
- the updated file is still file A
  - but the inode has changed
- a *piece* of the *inode map* is appended to the log
  - this piece is the one that contains the disk address of inode A

LFS Data Placement Example

- What happens if you want to modify the file?
  - how does "append-only" really work?
- Ex: you create file A and then file B

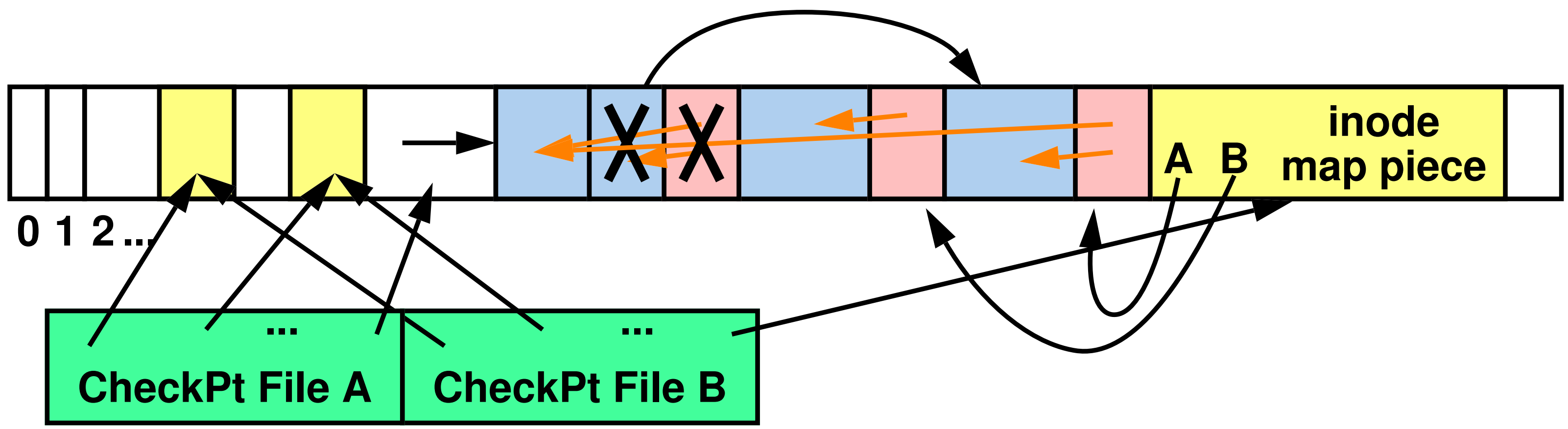


Inode Map:

- you modify file A, e.g., append to the last block of file A
- the updated file is still file A
  - but the inode has changed
- a *piece* of the *inode map* is appended to the log
  - this piece is the one that contains the disk address of inode A
  - fixed regions (previous version and current version) on the disk keeps track of *all* the *inode map pieces*
    - known as *checkpoint file*

More On Inode Map

- *Inode Map* cached in primary memory
  - indexed by inode number
  - points to inode on disk
  - written out to disk in pieces as updated
  - *checkpoint file* contains locations of pieces
    - written to disk occasionally
    - two copies: current and previous
    - outside of the "log" part of the LFS



- Commonly/Recently used inodes and other disk blocks cached in primary memory

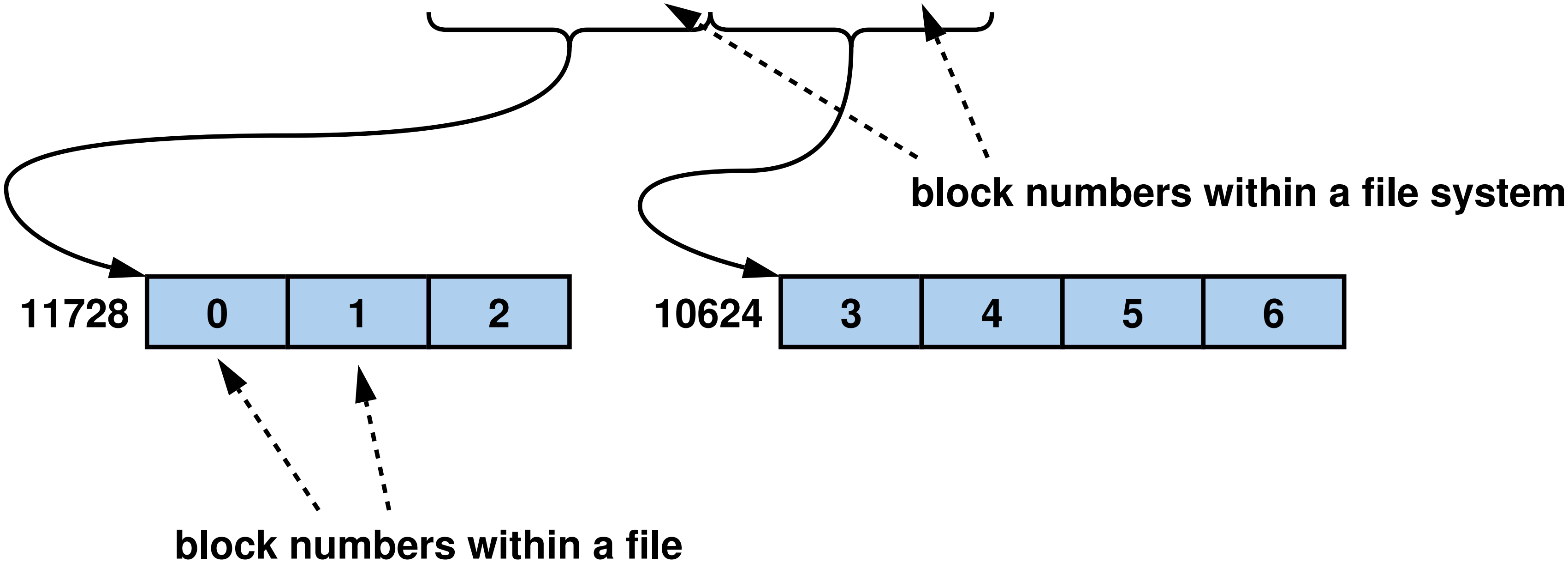
LFS Summary

- Advantages
  - good performance for *writes*
  - can *recover from crashes* easily through the use of *checkpoint files*
- Disadvantages
  - can waste a lot of disk space
    - cannot reclaim disk space and will run out of disk space

Extents in FAT16 & FAT 32

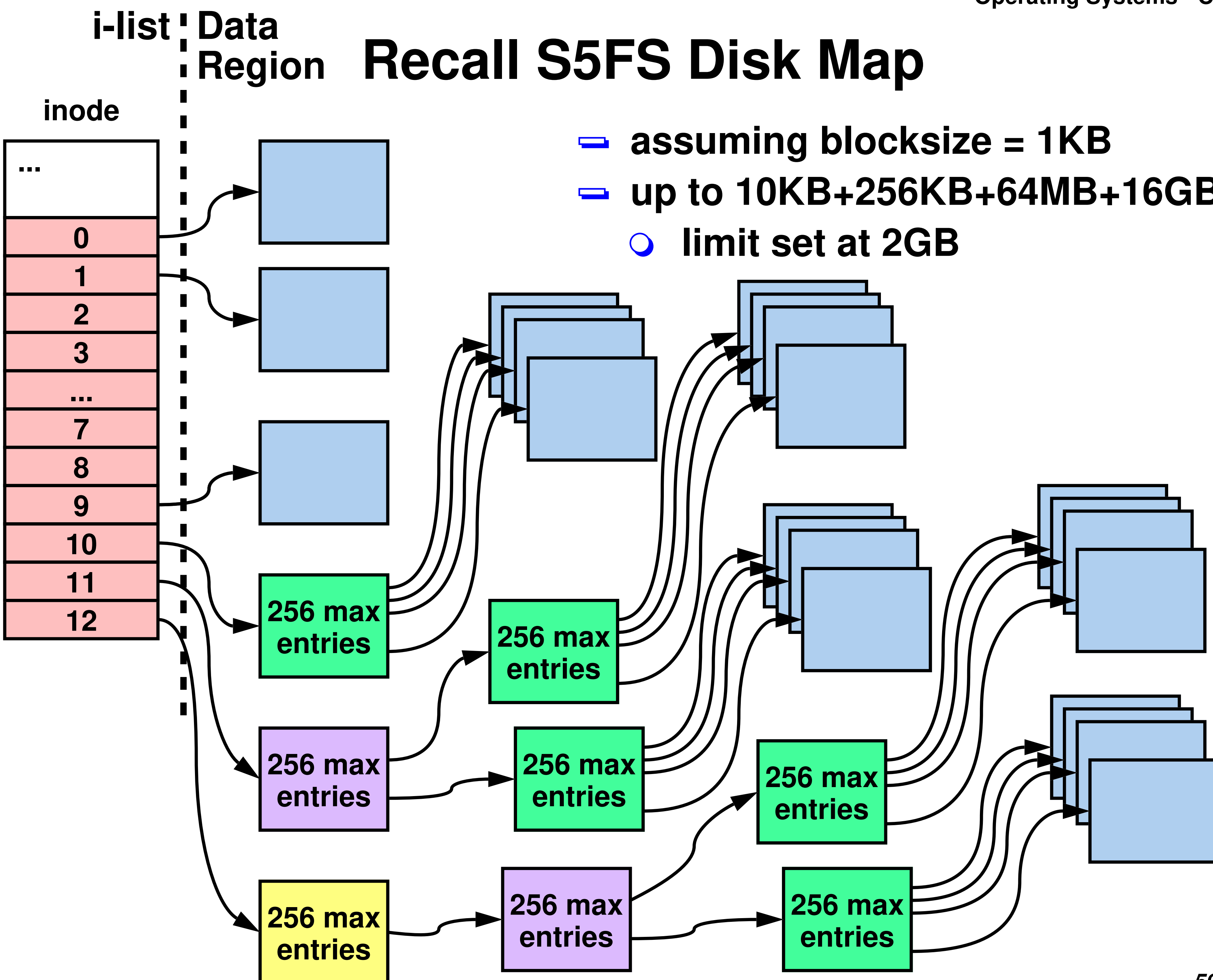
- Windows' equivalent of ***disk map*** in S5FS is ***extent***
  - an ***extent*** is a ***list*** of ***runs*** (consecutive disk blocks)

runlist							
length	offset	length	offset	length	offset	length	offset
3	11728	4	10624	...	...	...	...



Recall S5FS Disk Map

- assuming blocksize = 1KB
- up to 10KB+256KB+64MB+16GB
  - limit set at 2GB



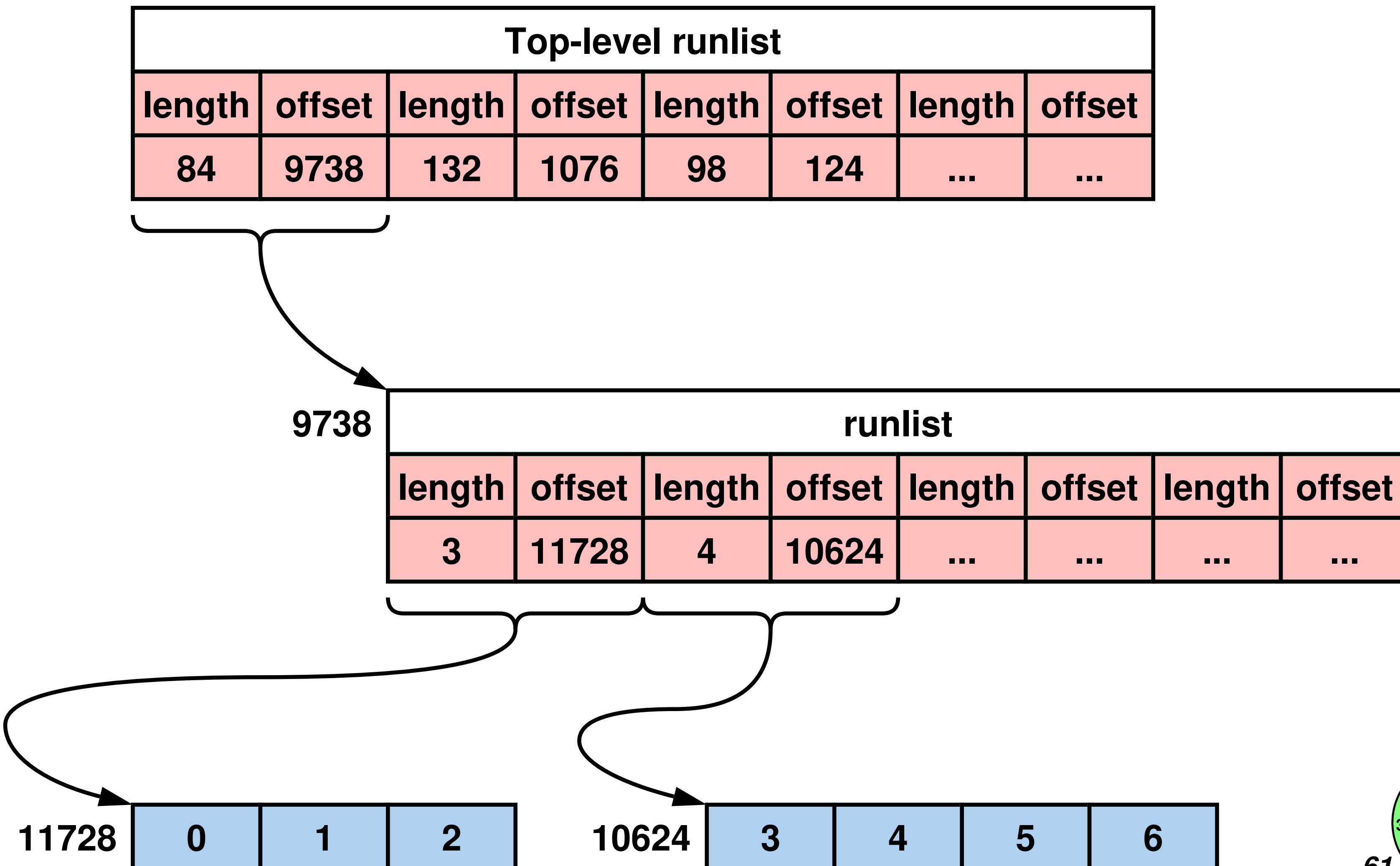
Problems with Extents in FAT16 & FAT 32

- Could result in highly *fragmented* disk space
  - lots of small areas of free space
    - *external fragmentation*
  - solution: use a *defragmenter* to *coalesce* free space

- Random access
  - *linear search* through a long list of extents
    - *O(n)* to find a disk block, recall that a disk map in S5FS is *O(1)*
  - solution: *multiple levels*
    - usually two levels

Extents in NTFS

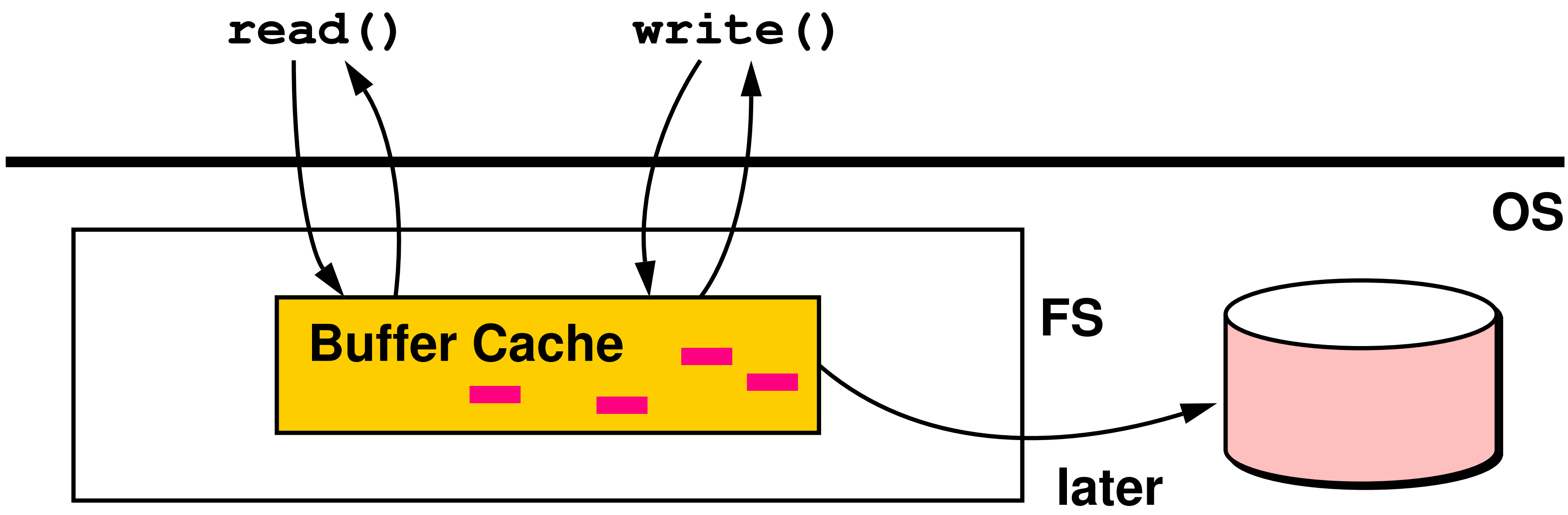
- **Two-level runlists**
  - make sure that every runlist fits inside *one disk block*
  - better performance, but still needs de-frag



6.2 Crash Resiliency

- What Goes Wrong
- Dealing with Crashes

Buffer Cache With Write-back

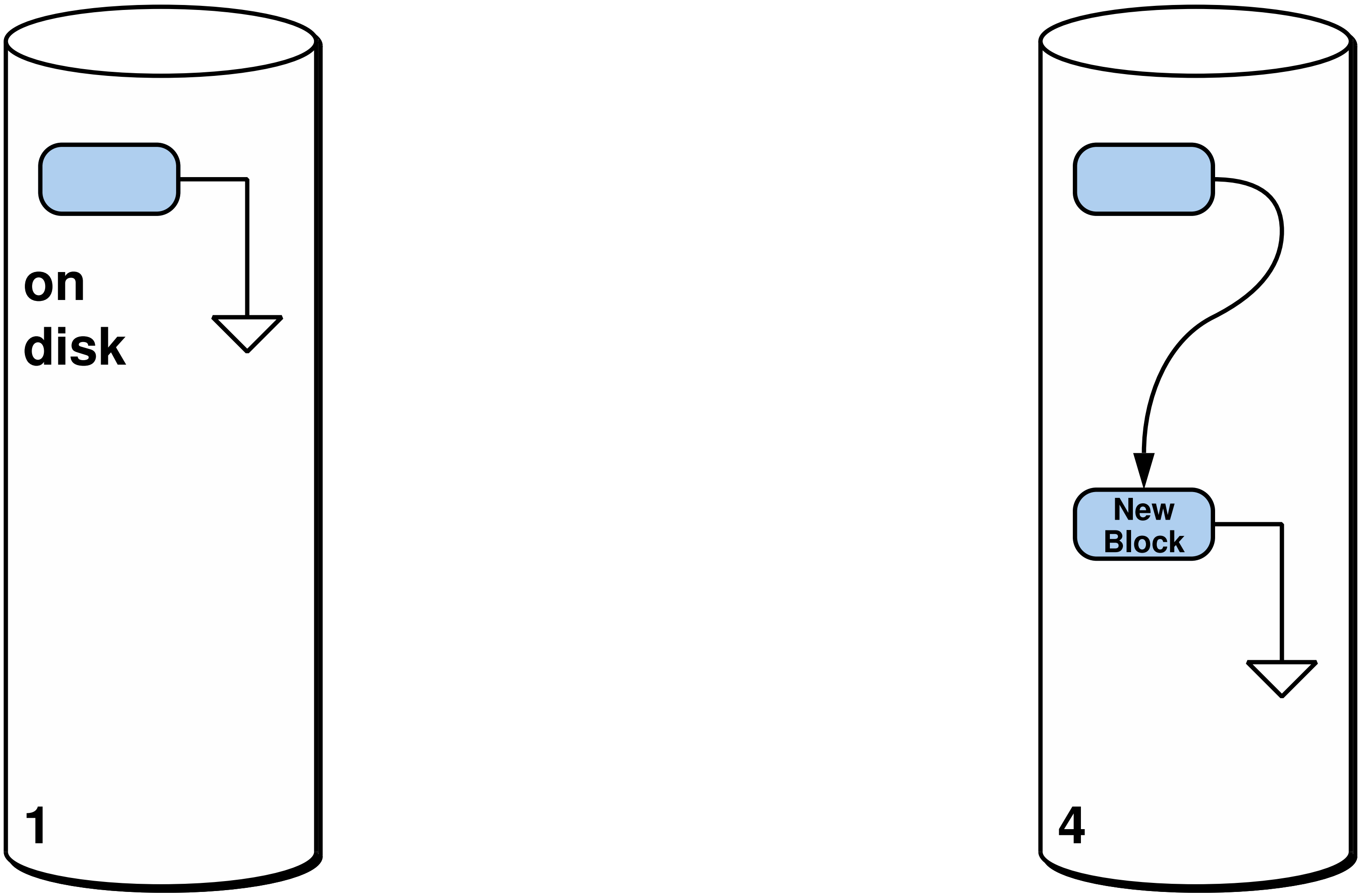


- *Dirty/modified blocks* in buffer cache
  - disk blocks are read in and cached in the buffer cache
    - originally "clean/unmodified"
  - a write operation would modify a disk block in the buffer cache
    - the block is labeled *"dirty/modified"*
  - *disk update:* the file system periodically gathers all the dirty blocks, update the disk, and clear the "dirty bits"
    - update is done *one disk block at a time*

In the Event of a Crash ...

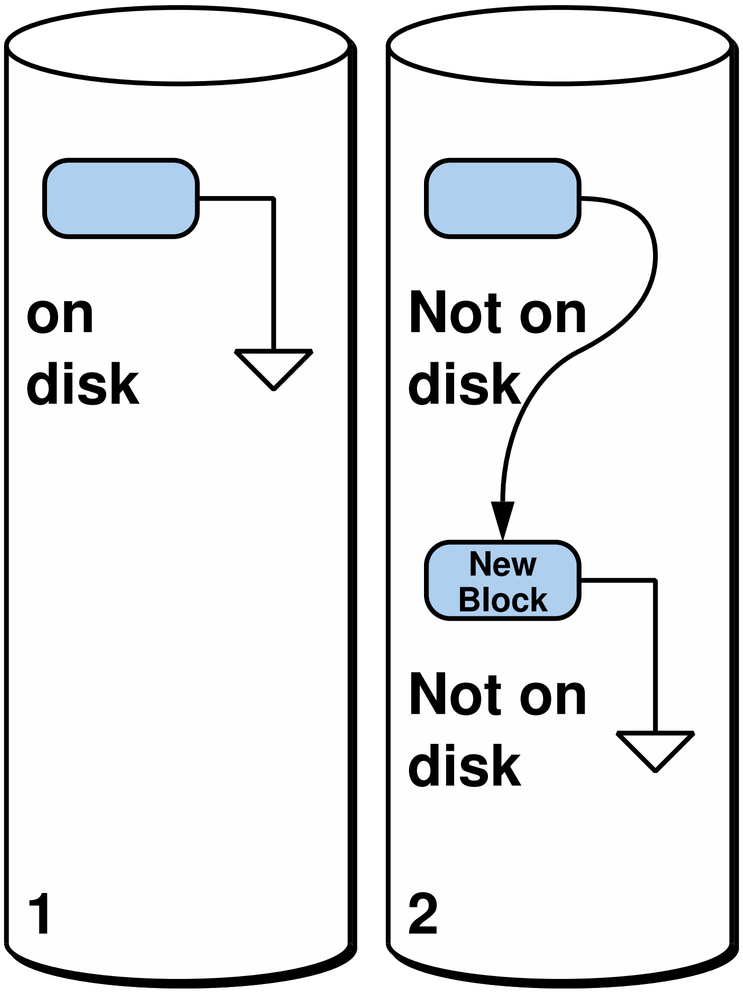
- Most recent updates did not make it to disk
  - is this a big problem?
  - equivalent to crash happening slightly earlier
    - but you may have received (and believed) a message:
      - "file successfully updated"
      - "homework successfully handed in"
      - "stock successfully purchased"
  - there’s worse ...

File-System Consistency (1)



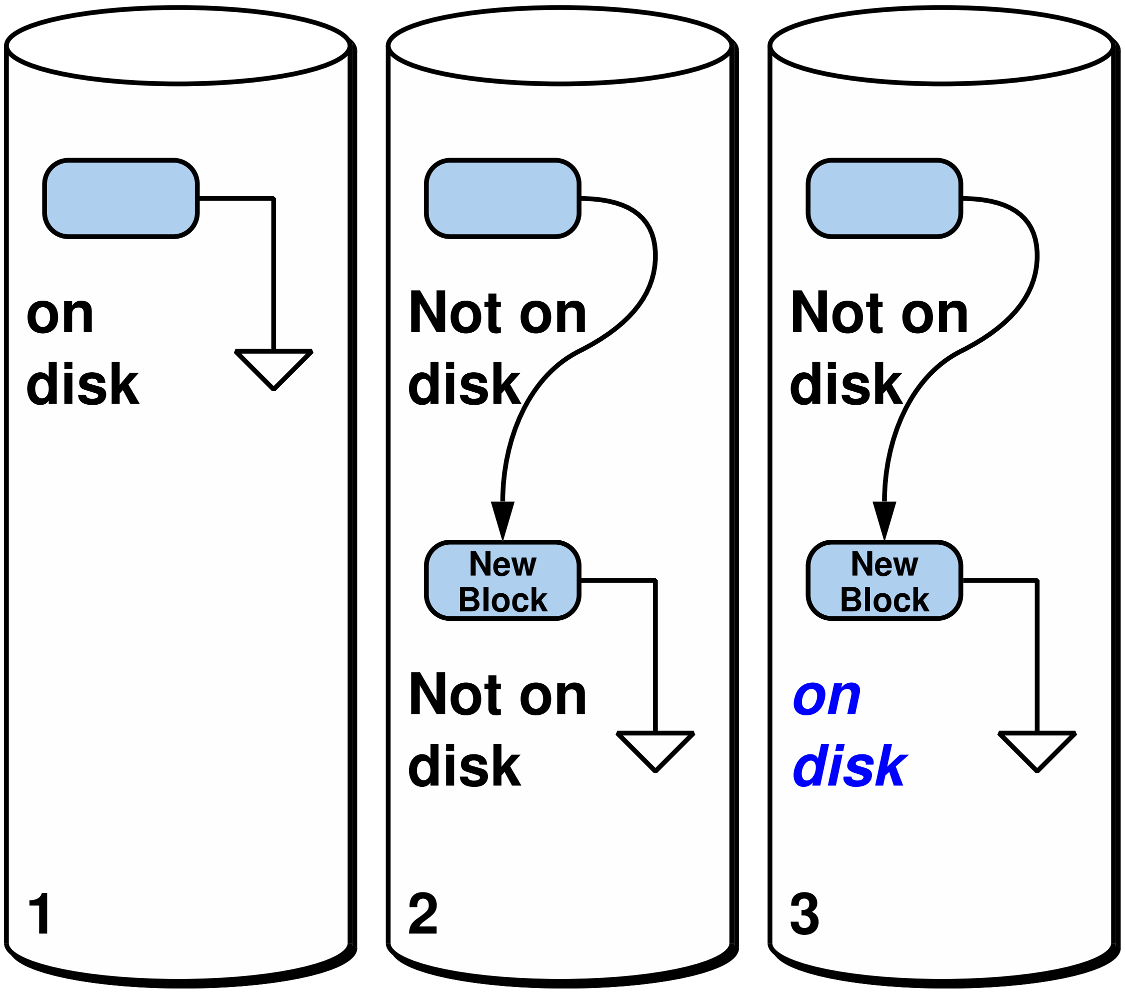
- How to go from 1 to 4 *atomically*?

File-System Consistency (1)



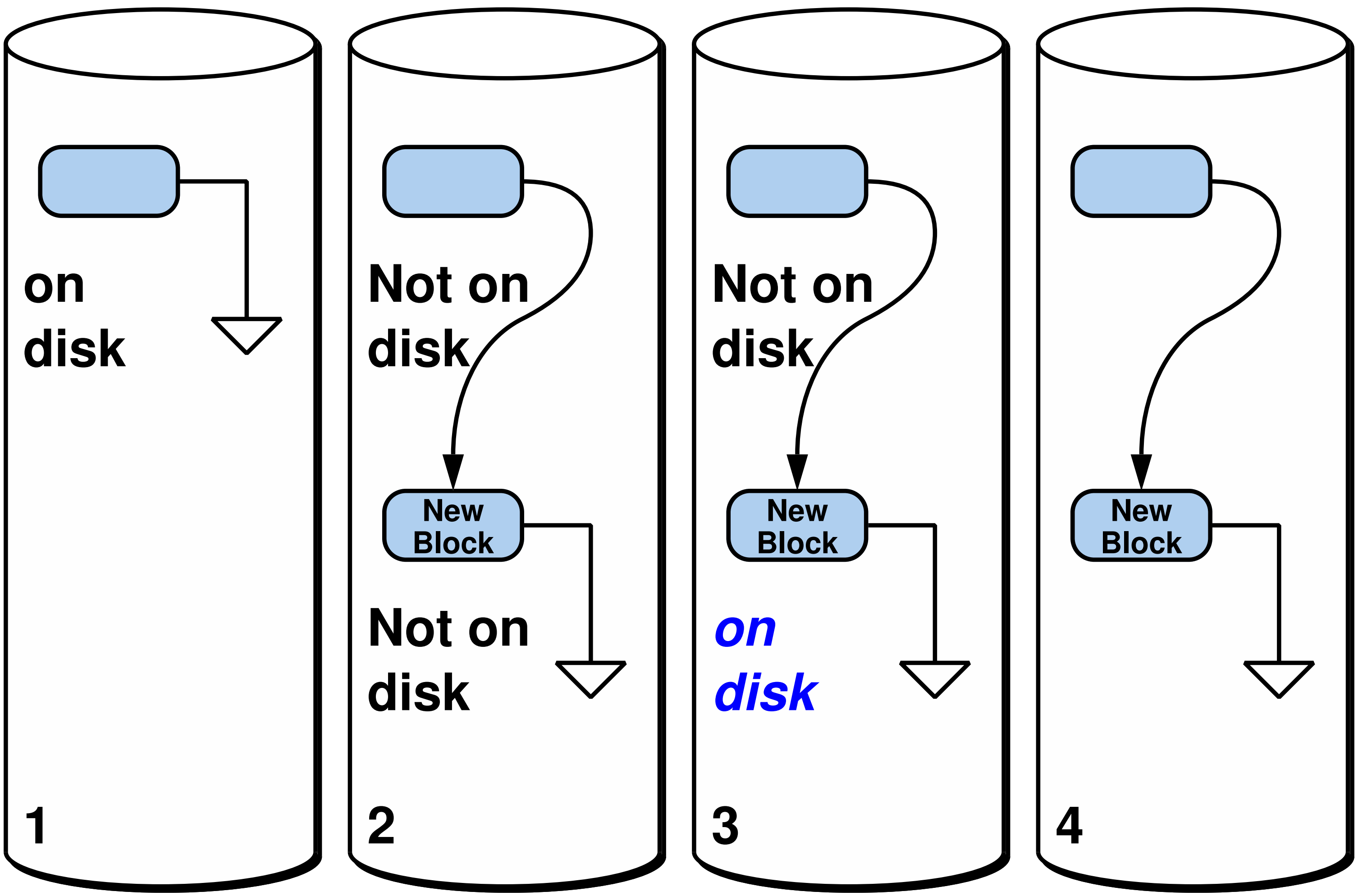
- How to go from 1 to 4 ***atomically***?
  - release dirty blocks to disk update thread

File-System Consistency (1)



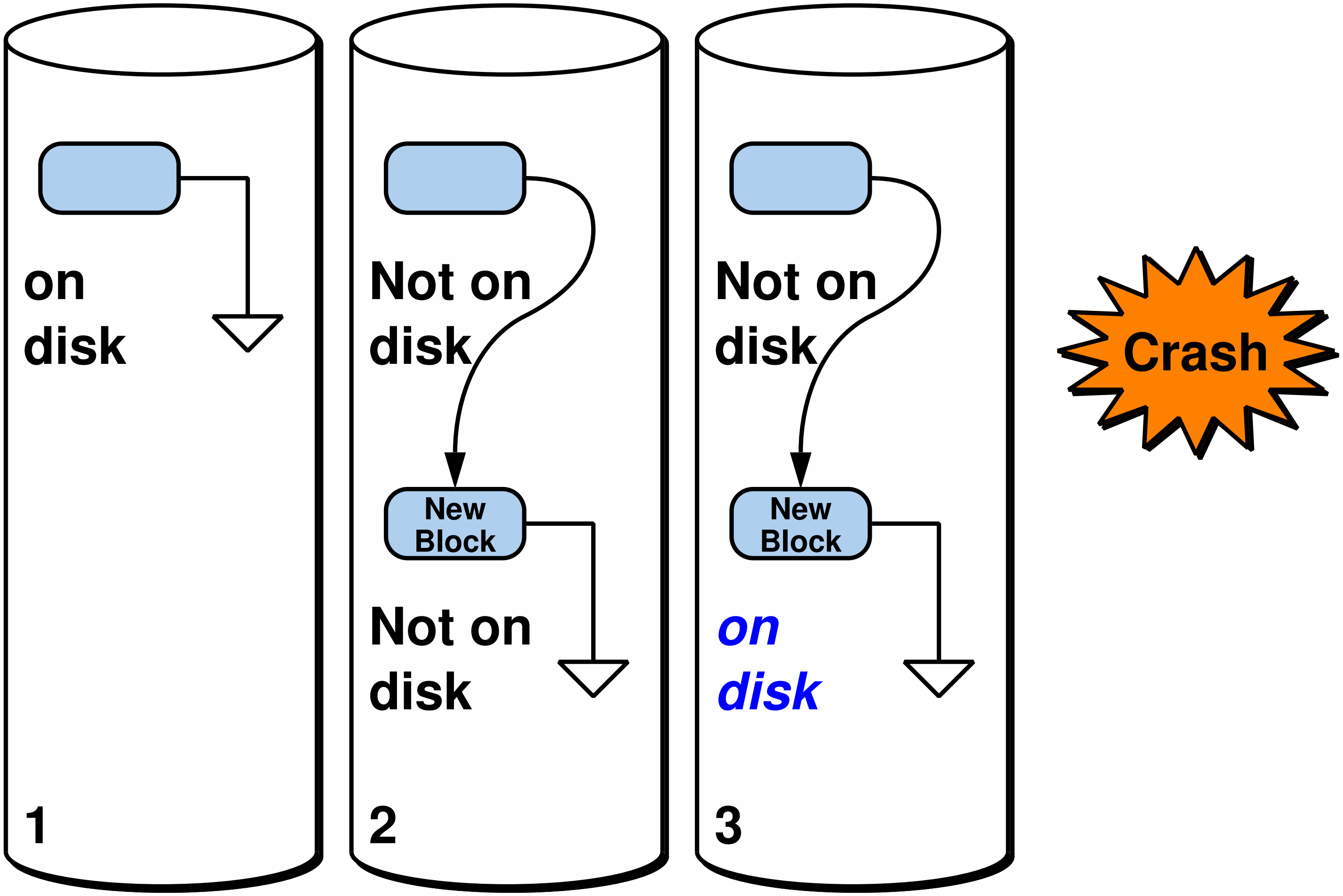
- How to go from 1 to 4 ***atomically***?
  - write the "new block" first

File-System Consistency (1)



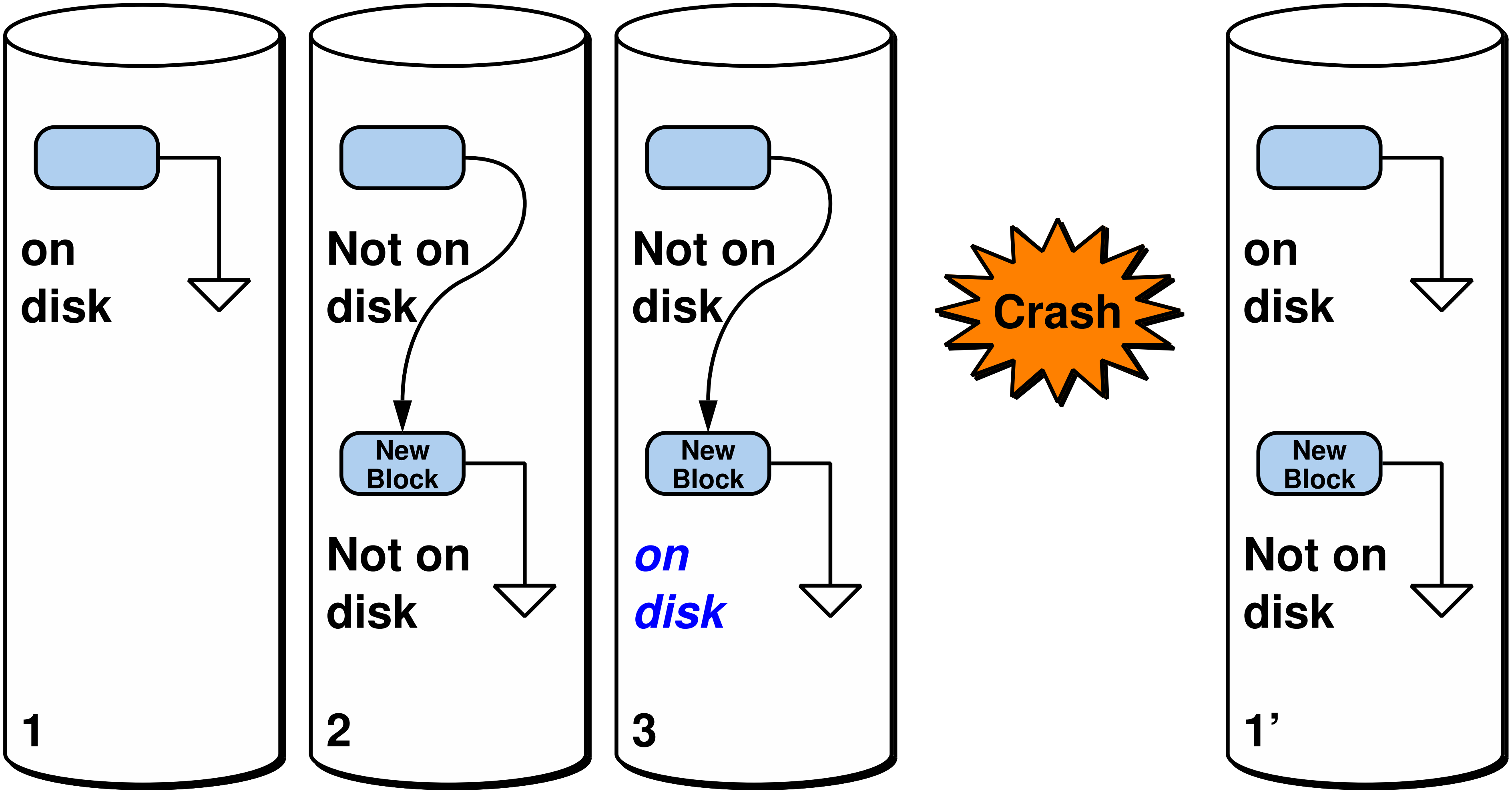
- How to go from 1 to 4 ***atomically***?
  - write the "new block" first
  - then write new values into the old block

File-System Consistency (1)



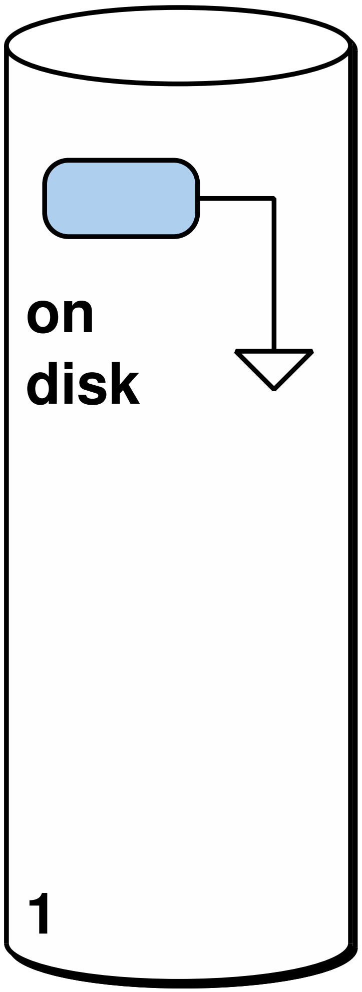
- If crash occurs before the modified old block is written to the disk

File-System Consistency (1)



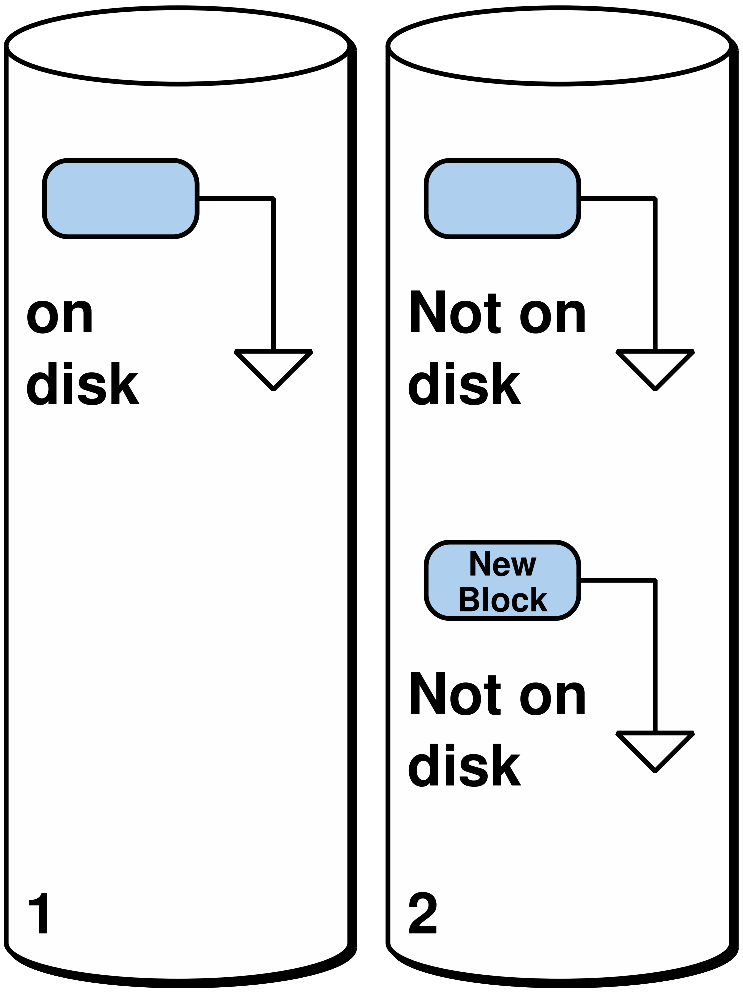
- If crash occurs before the modified old block is written to the disk
  - is this okay?

File-System Consistency (2)



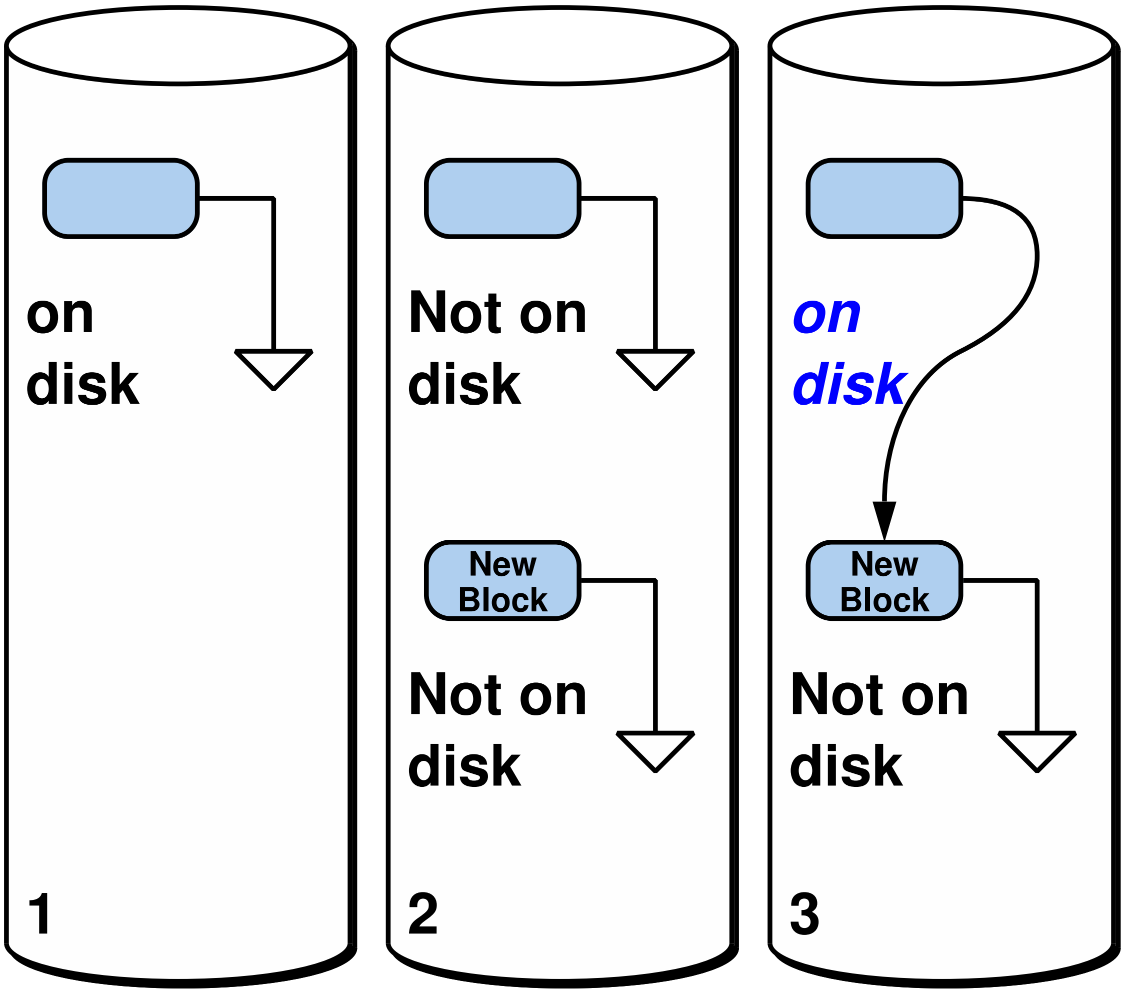
- Problem: in S5FS and FFS, the *disk update thread* can sequence disk writes *in any order*
  - it may use an *elevator algorithm*

File-System Consistency (2)



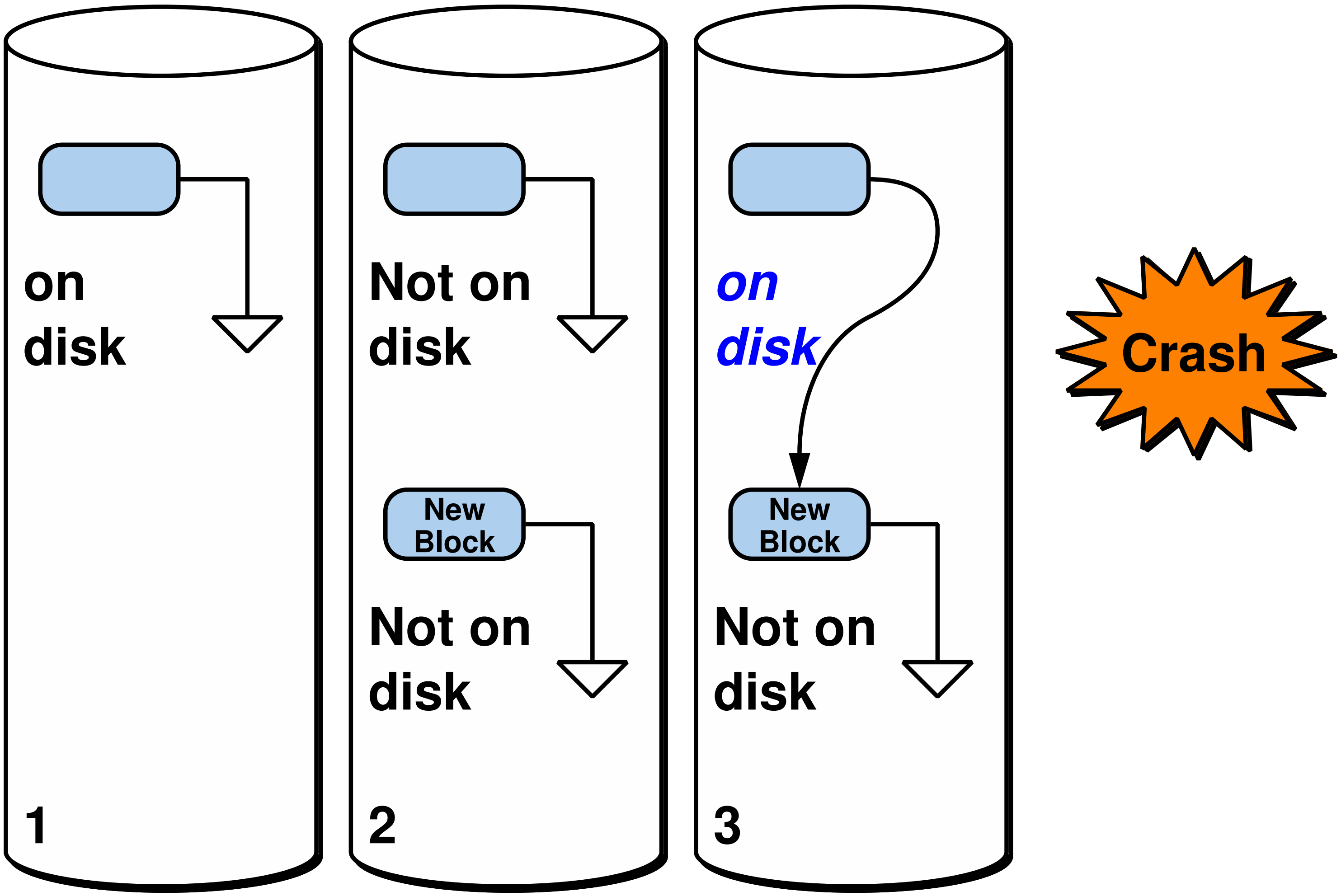
- Problem: in S5FS and FFS, the *disk update thread* can sequence disk writes *in any order*

File-System Consistency (2)



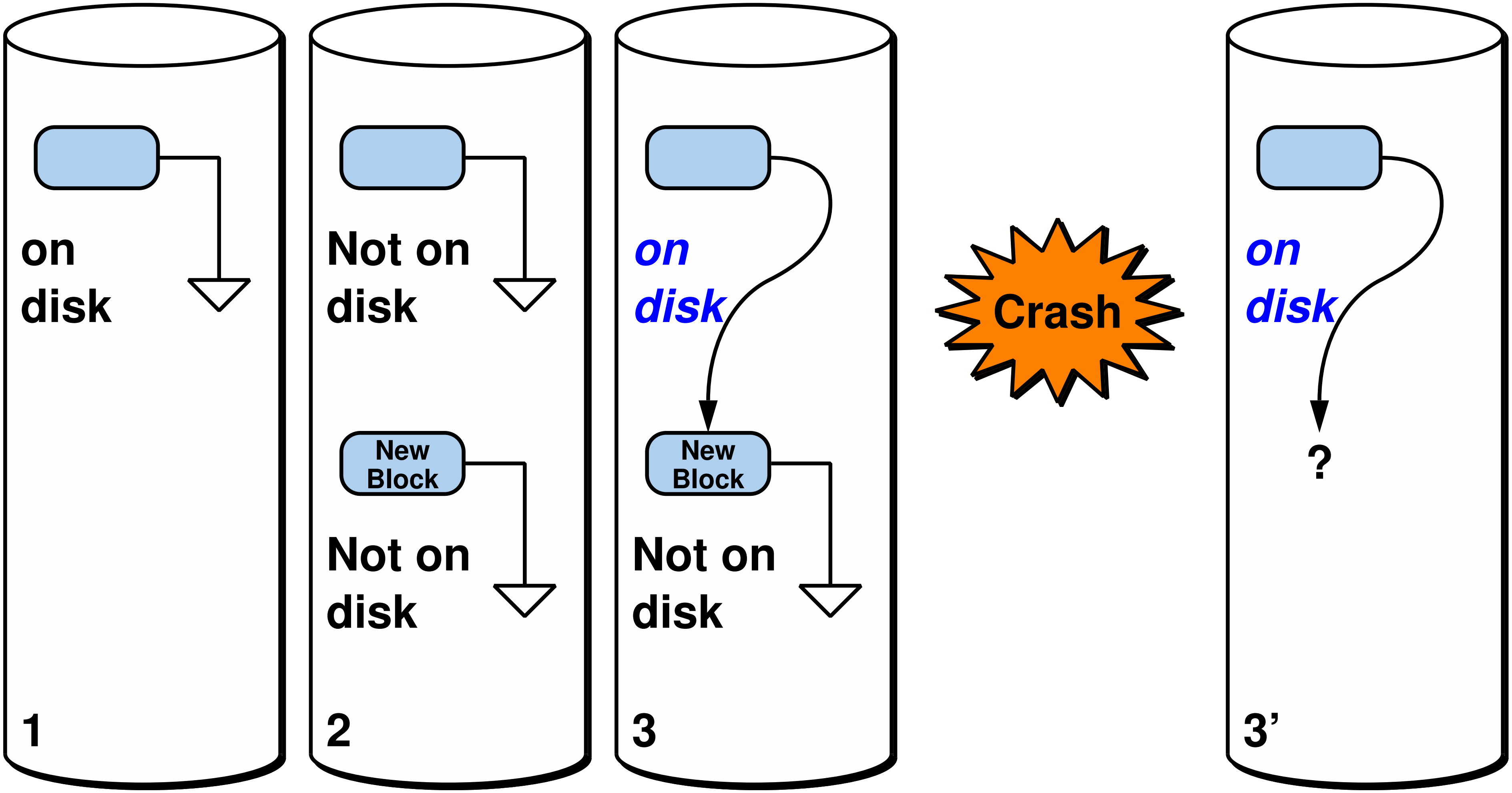
- Problem: in S5FS and FFS, the *disk update thread* can sequence disk writes *in any order*
  - what if it writes new values into the old block first?

File-System Consistency (2)



- Problem: in S5FS and FFS, the *disk update thread* can sequence disk writes *in any order*
  - what if it writes new values into the old block first?

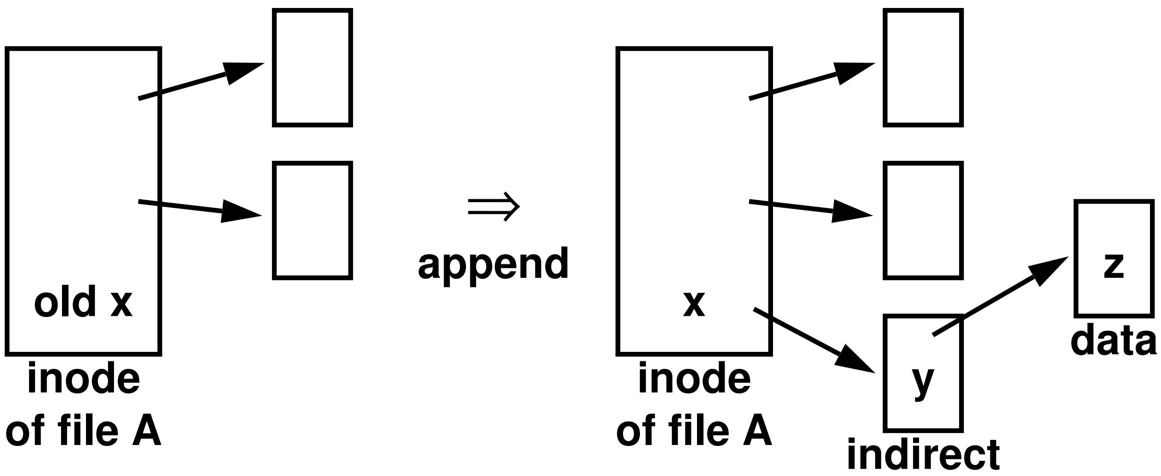
File-System Consistency (2)



- Problem: in S5FS and FFS, the *disk update thread* can sequence disk writes *in any order*
  - what if it writes new values into the old block first?

A More Realistic Example

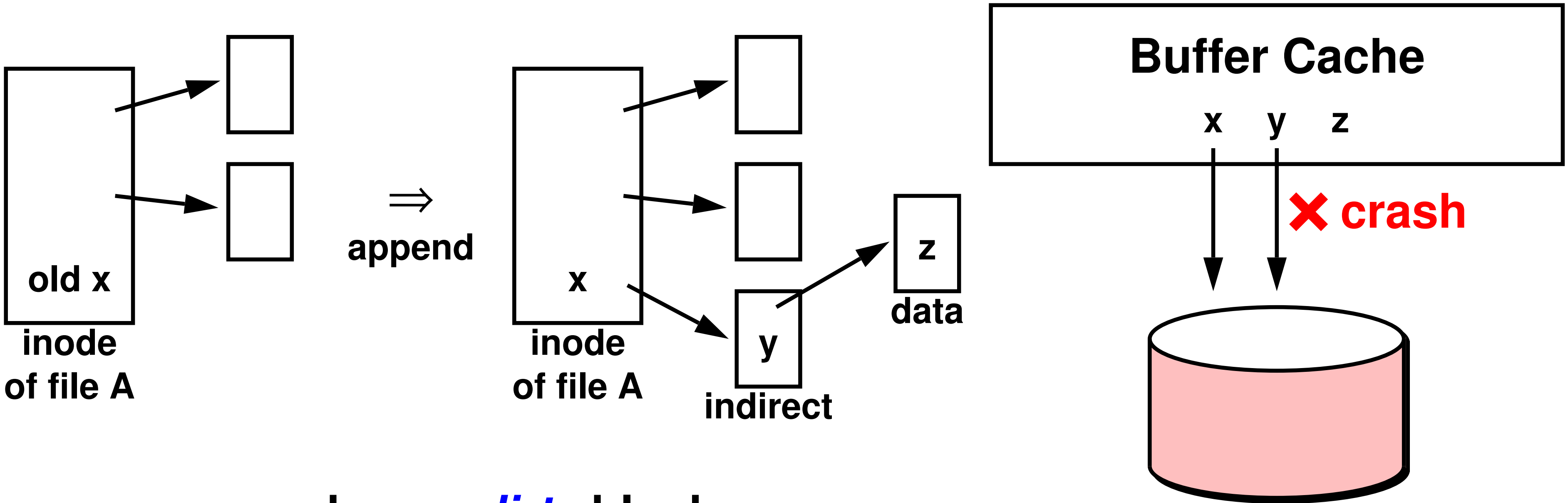
- Let's say that you are appending to 10KB file A



  - x, y, and z are *dirty* blocks

A More Realistic Example

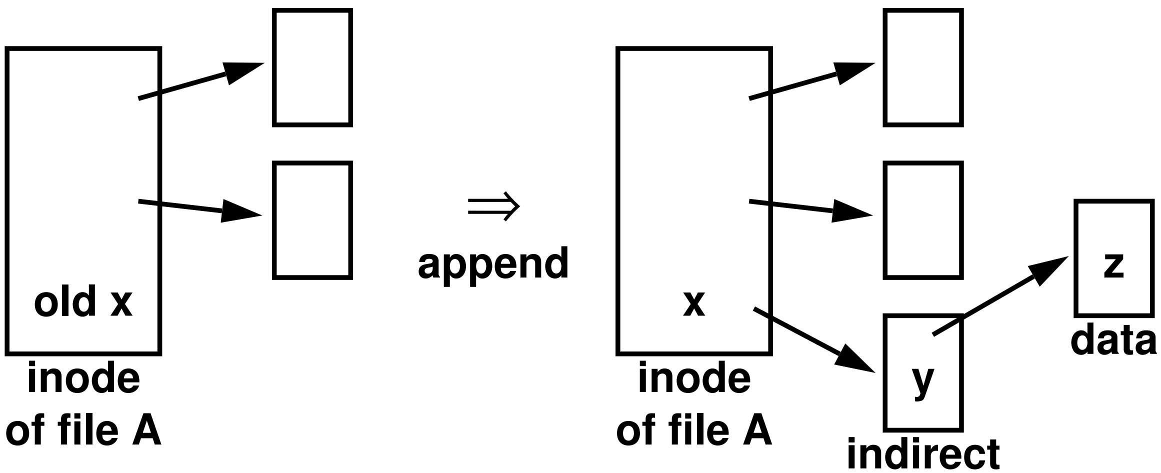
- Let's say that you are appending to 10KB file A

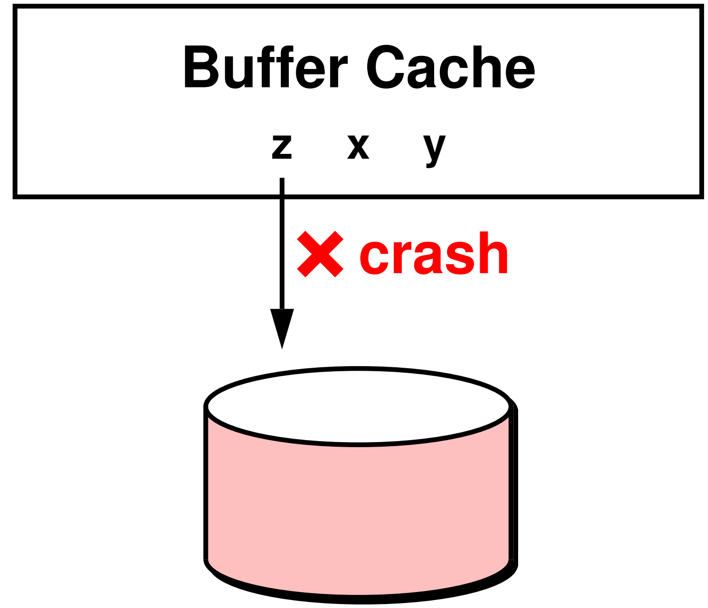


- x, y, and z are *dirty* blocks
- the buffer cache does not know about the relationship among blocks x, y, and z
- techniques like locking (i.e., lock the disk or file system so that it cannot crash when it's locked) won't work
- it's obvious that the solution is to *make the disk update thread aware* of the relationship among these blocks
  - but how? there are different approaches

A More Realistic Example

- Let's say that you are appending to 10KB file A



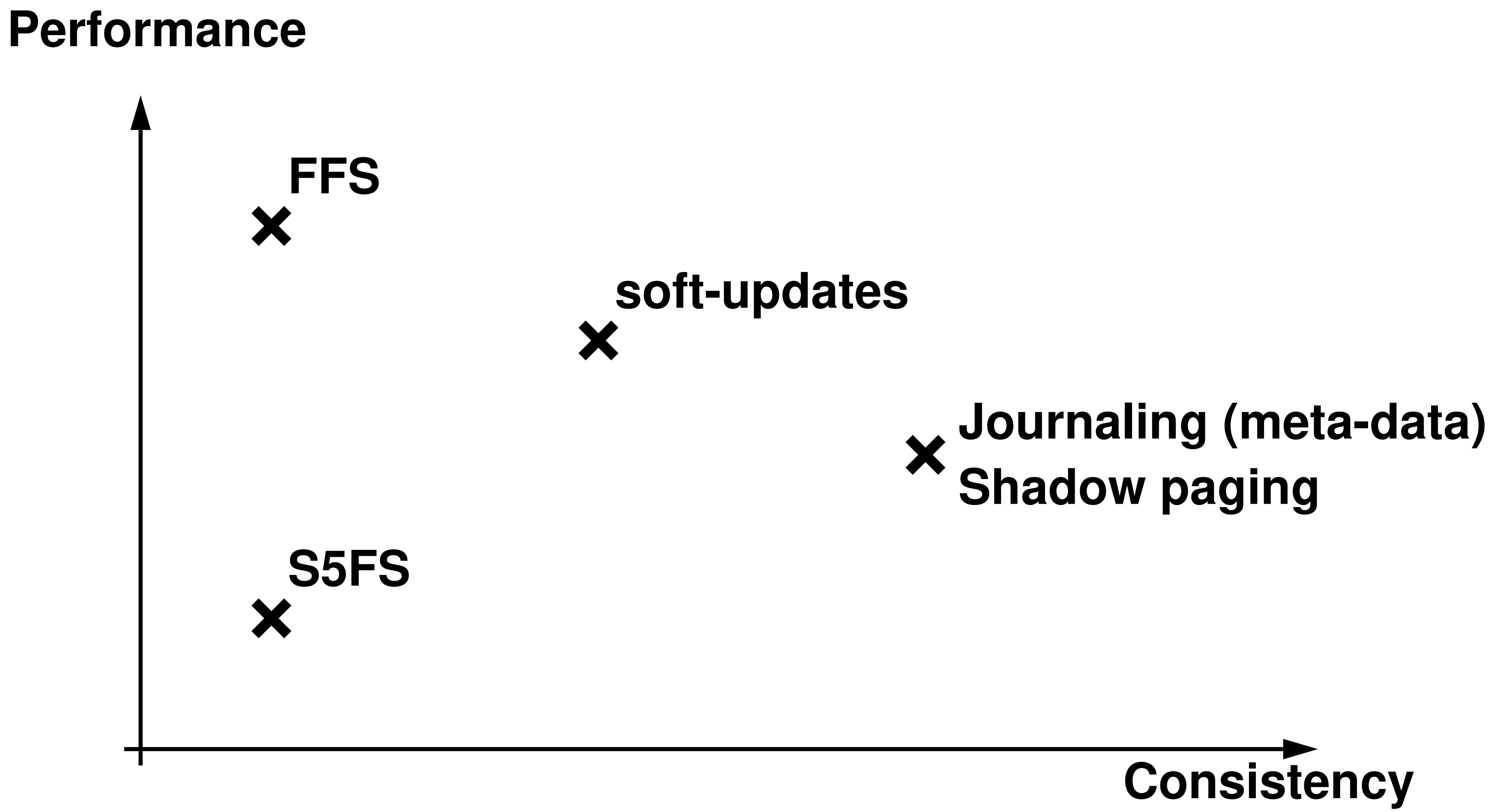


- x, y, and z are *dirty* blocks
- what about this order and crash timing?
  - what about other combinations?
- does order matter?
- should order matter?

How to Cope ...

- **Don't crash**
  - **not realistic**
- **Perform multi-step disk updates in an order such that disk is always consistent, i.e., the *consistency-preserving approach***
- **Perform multi-step disk updates as *transactions*, i.e., implemented so that either all steps take effect or none do**

How to Cope ...

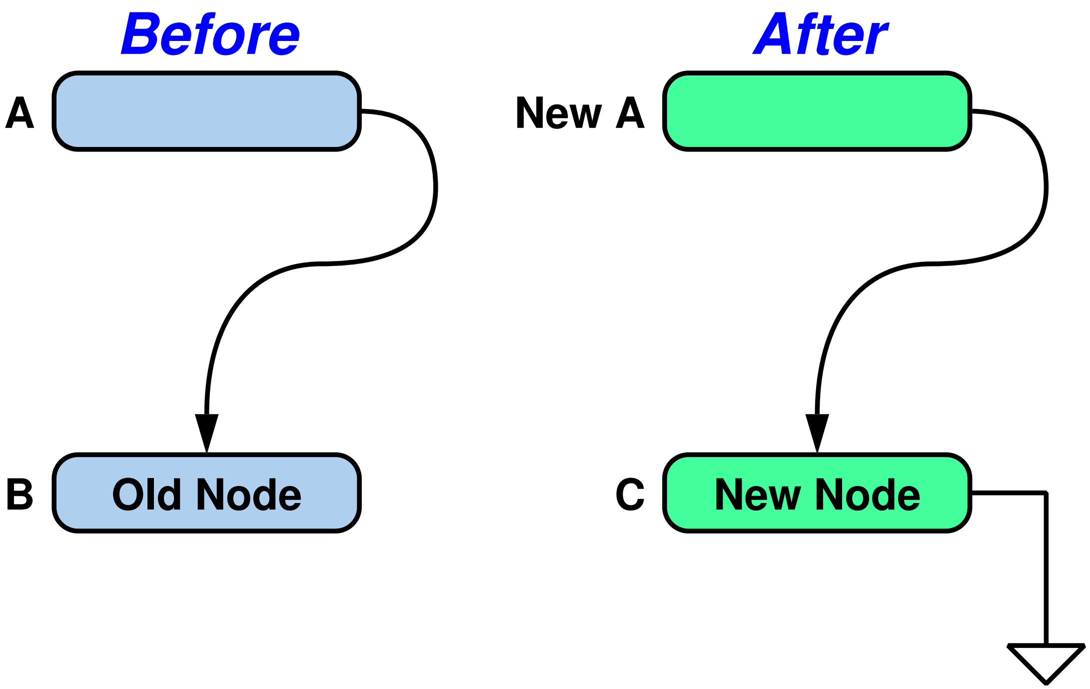


- soft-update provides *recoverable consistency*
- journaling and shadow paging provide *transactional consistency*

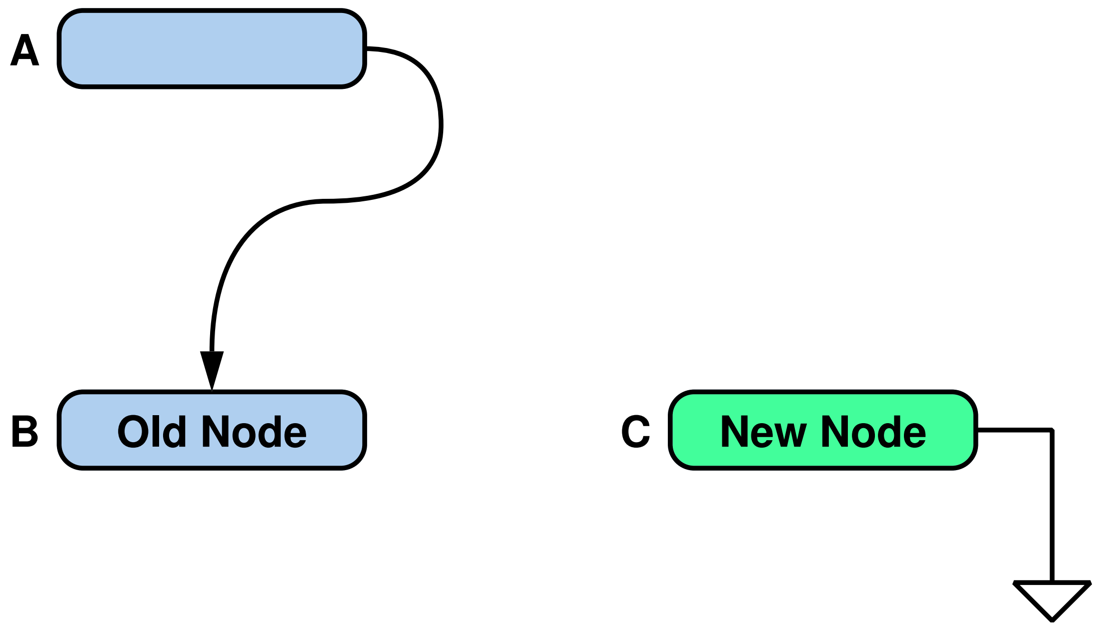
Soft Update

- **Main idea**
  - *order disk operations* to *preserve meta-data consistency*
    - "innocuous inconsistency" is considered ok
  - synchronous write can be slow
    - use data structure to *describe dependencies* and pass the data structure to disk update task

Maintaining Consistency

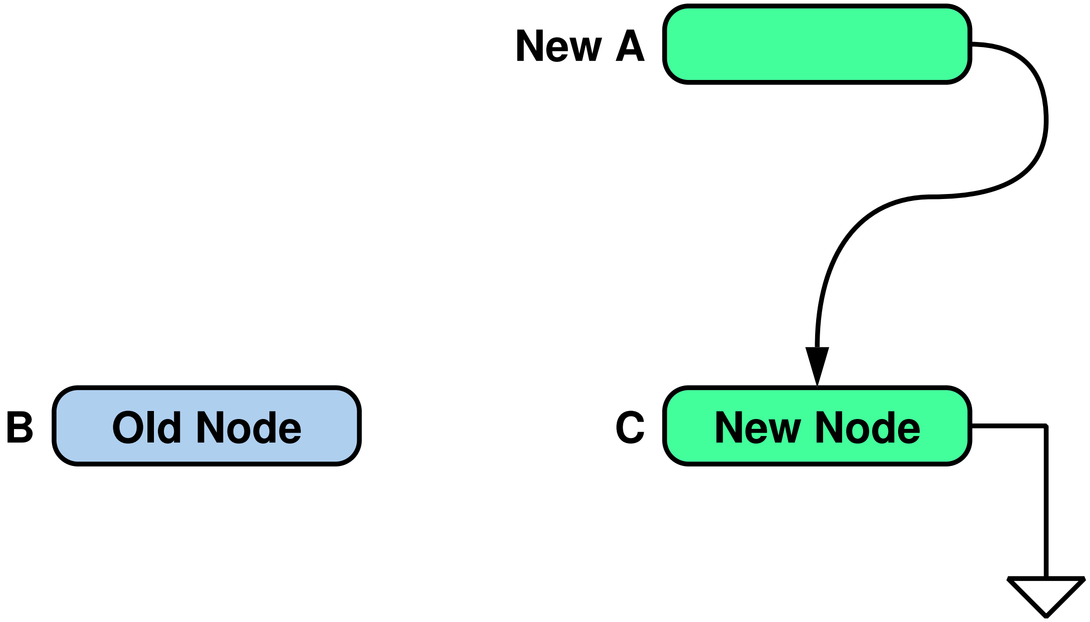


Maintaining Consistency



1) In AFS, write this ***synchronously*** to disk (like write-through)

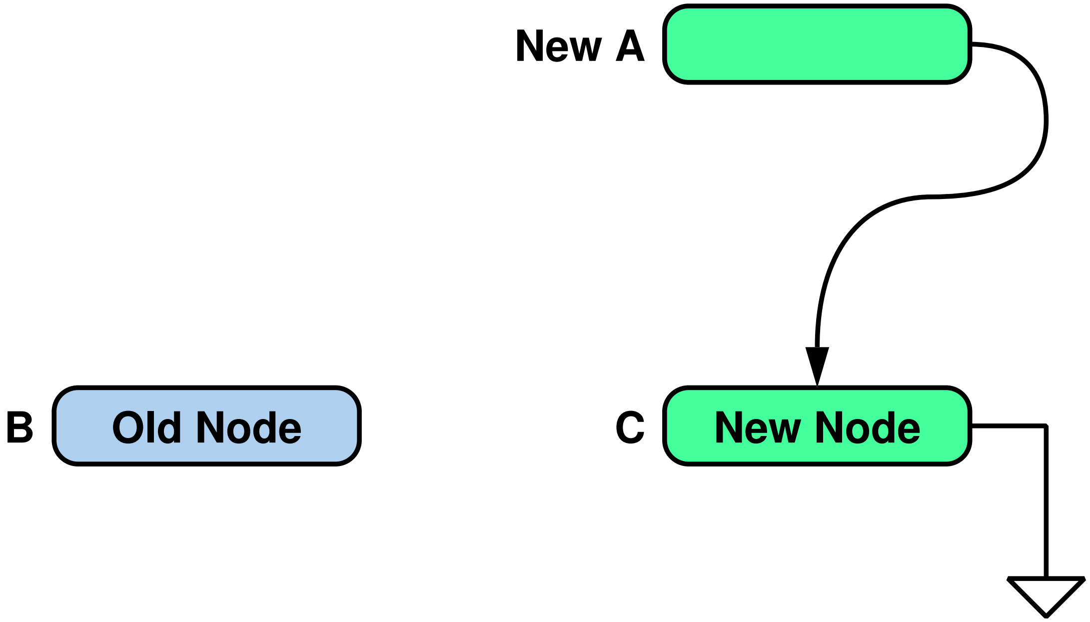
Maintaining Consistency



2) Then write this ***asynchronously*** via the cache (i.e., send to disk update task)

1) In AFS, write this ***synchronously*** to disk (like write-through)

Maintaining Consistency

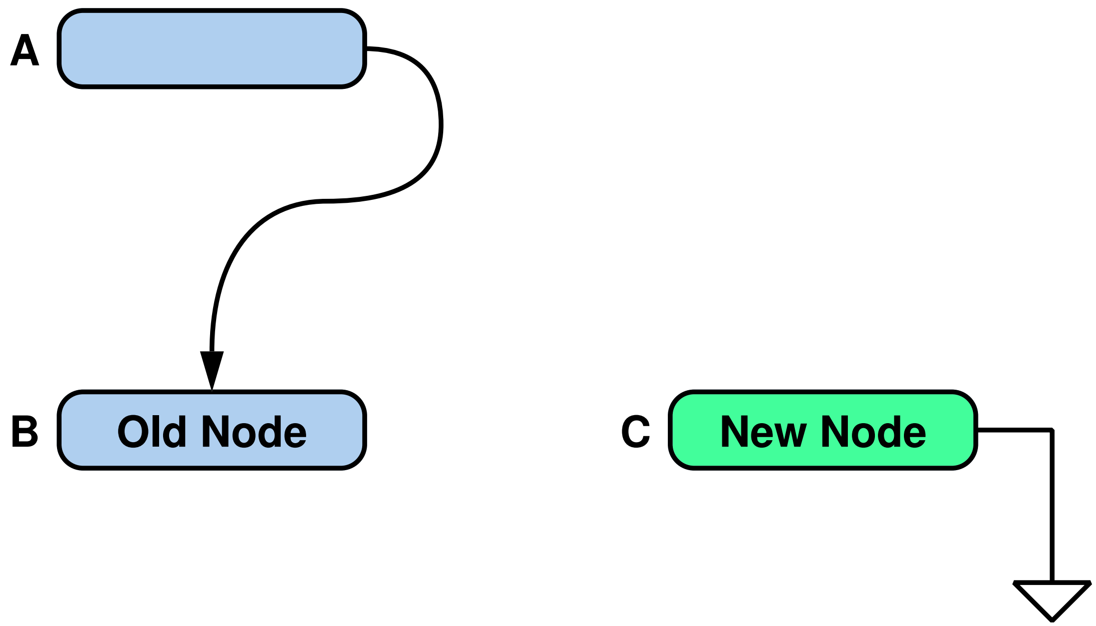


2) Then write this *asynchronously* via the cache (i.e., send to disk update task)

1) In AFS, write this *synchronously* to disk (like write-through)

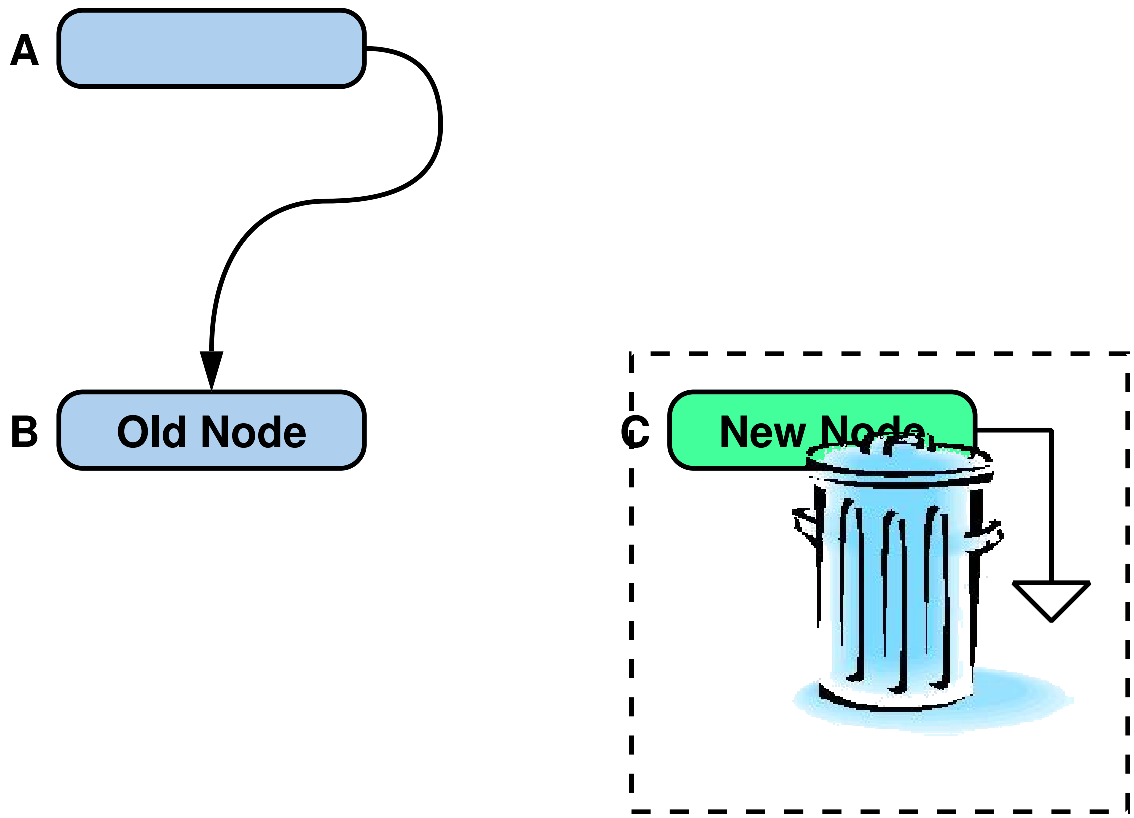
- If crash happens before (2) is performed but after (1) is performed
  - what would it look like?

Innocuous Inconsistency



- If crash happens before (2) is performed but after (1) is performed
  - what would it look like?

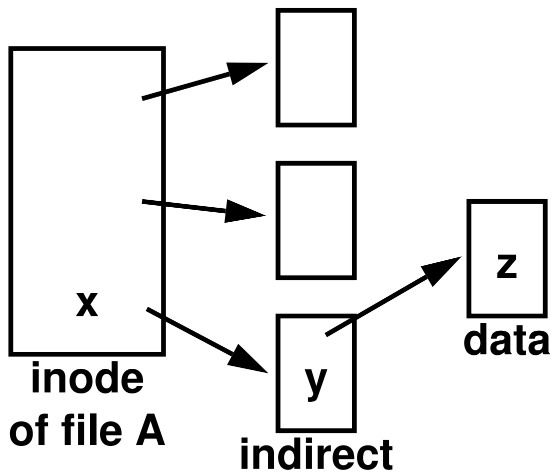
Innocuous Inconsistency

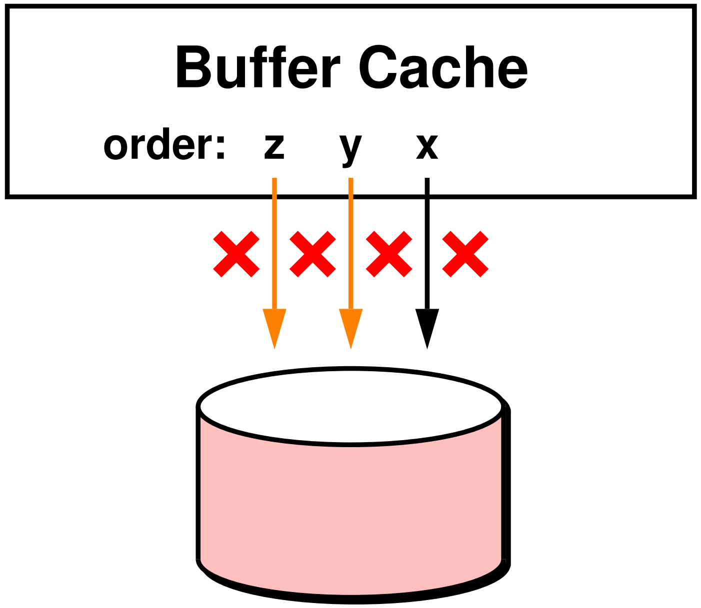


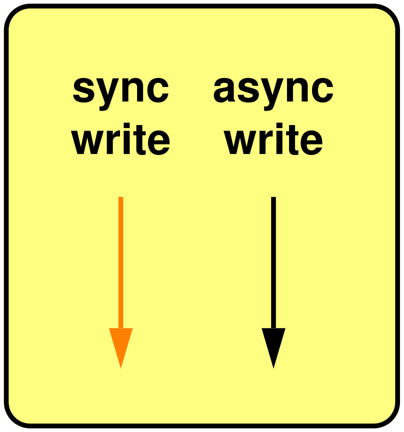
- ***Innocuous inconsistency*** **is** ***acceptable***
  - **although need to** ***reclaim lost disk blocks***
    - **e.g., in FFS, use a "disk scavenger" to find all these blocks and add them to** `/lost+found`

Back To The Example

- Let's say that you are appending to file A (y and z are new blocks)

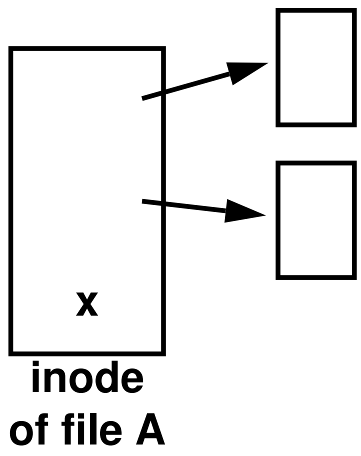


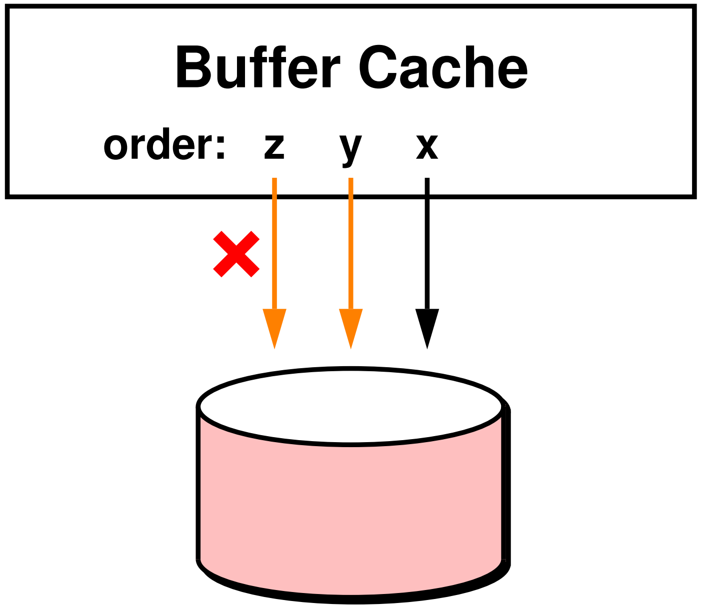


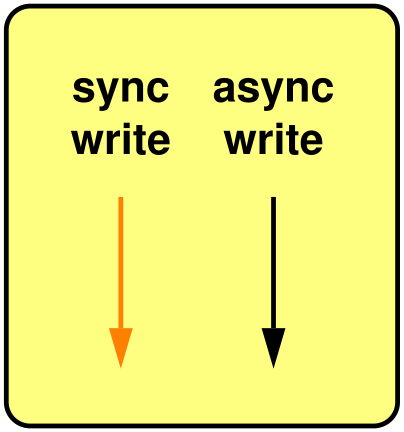


Back To The Example

- Let’s say that you are appending to file A (y and z are new blocks)

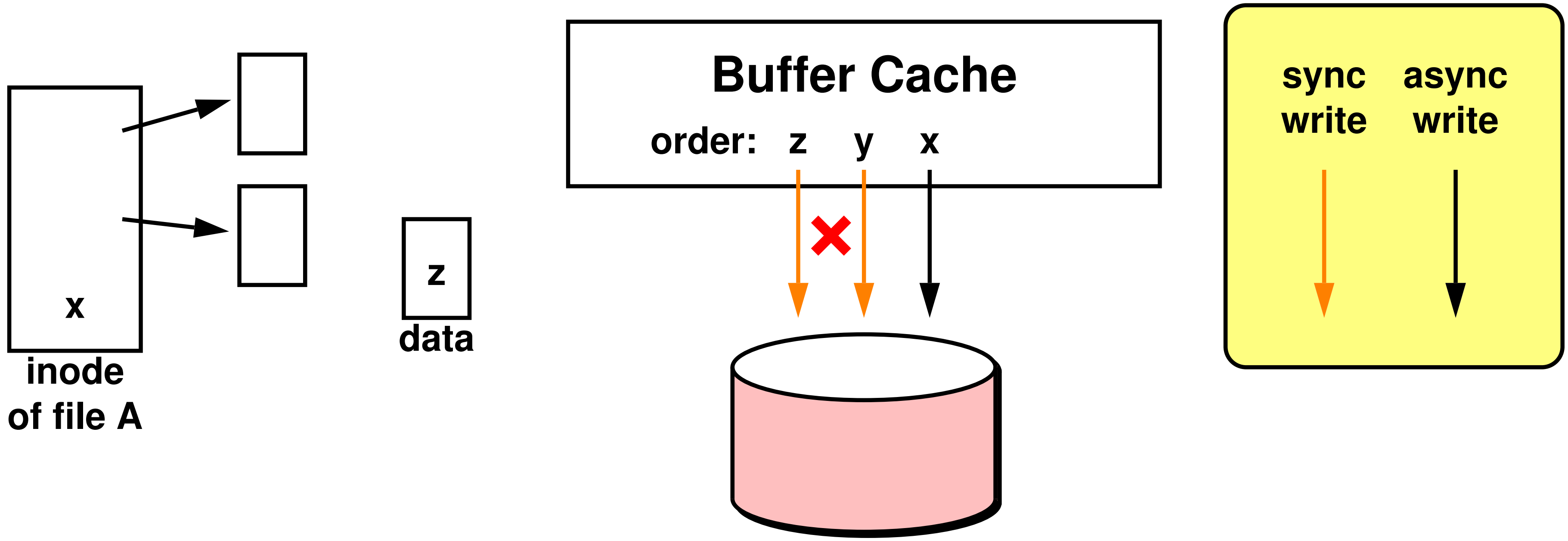






Back To The Example

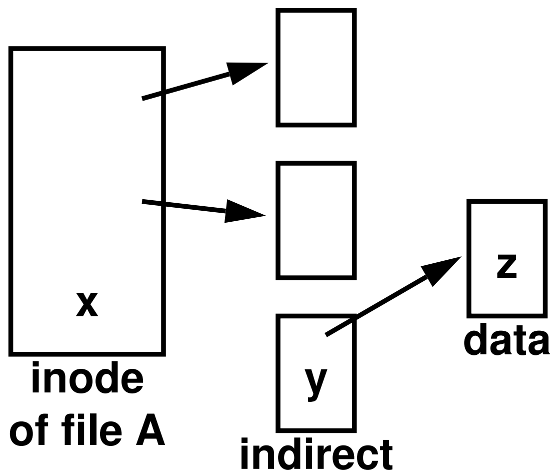
- Let's say that you are appending to file A (y and z are new blocks)

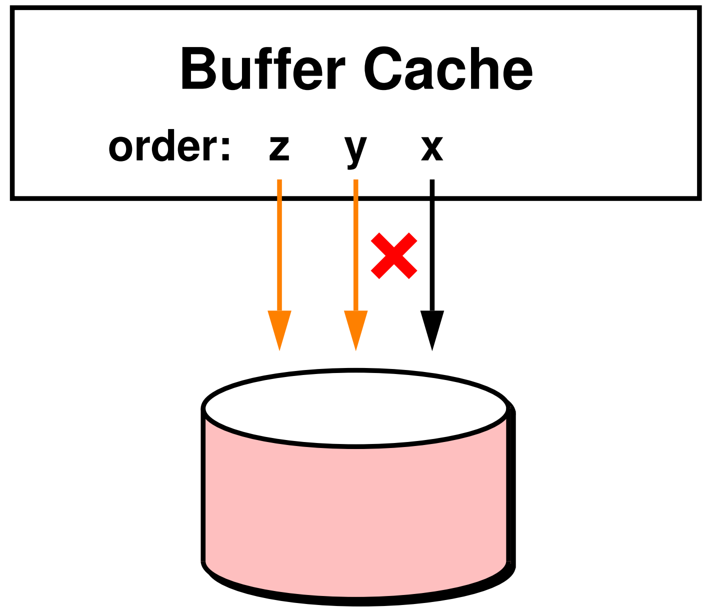


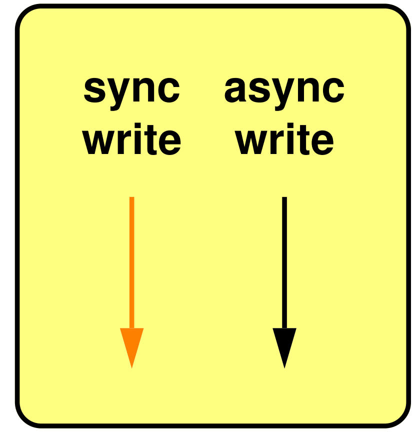
- is this bad?
  - how bad is it?

Back To The Example

- Let's say that you are appending to file A (y and z are new blocks)



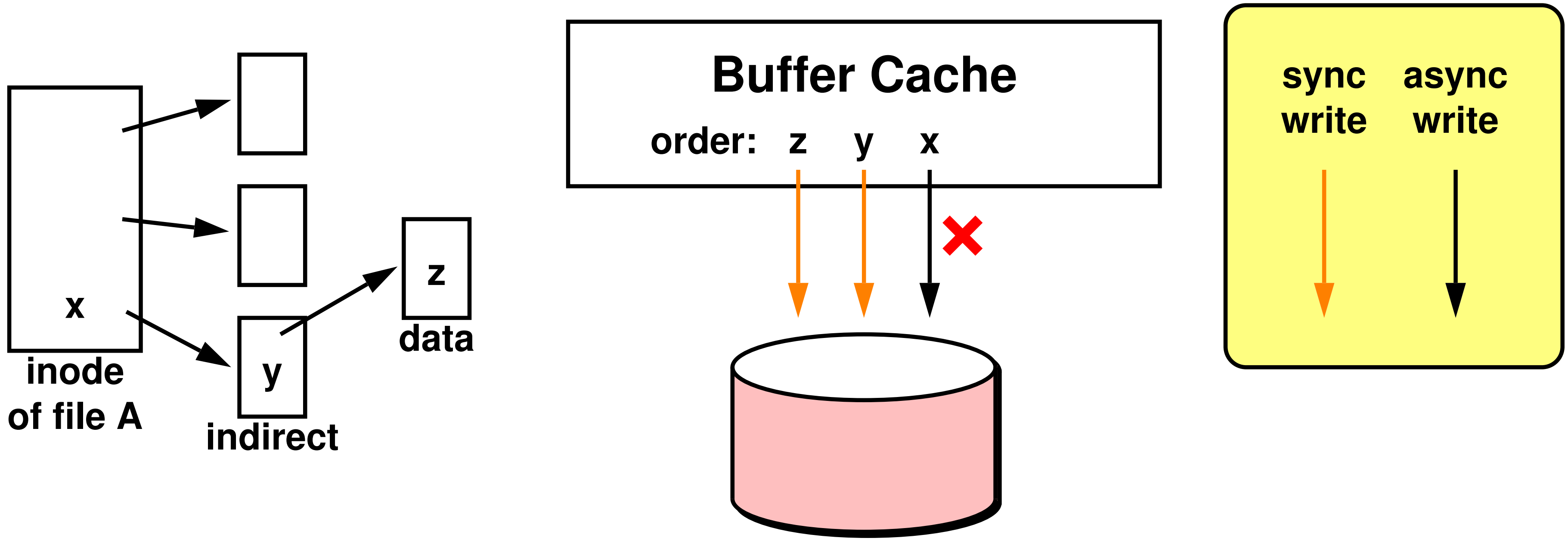




- is this bad?
  - how bad is it?

Back To The Example

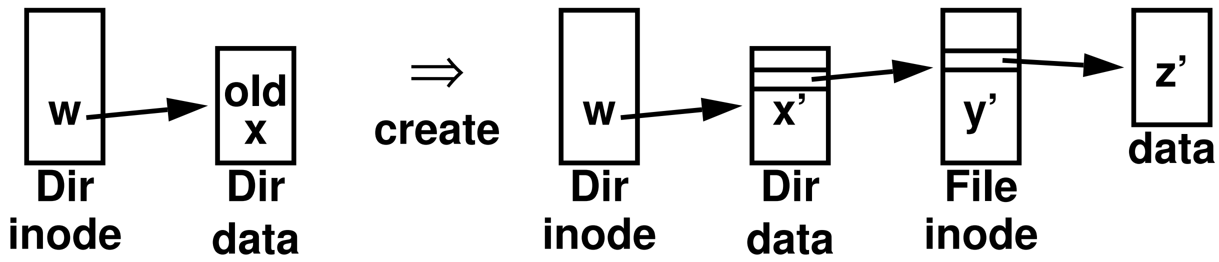
- Let's say that you are appending to file A (y and z are new blocks)

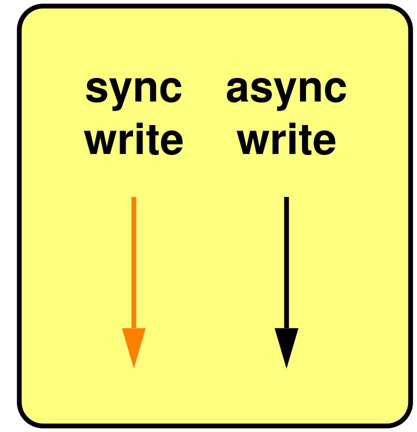


  - is this bad?
    - no
    - although this is slow because it requires 2 synchronous writes
      - failure is uncommon

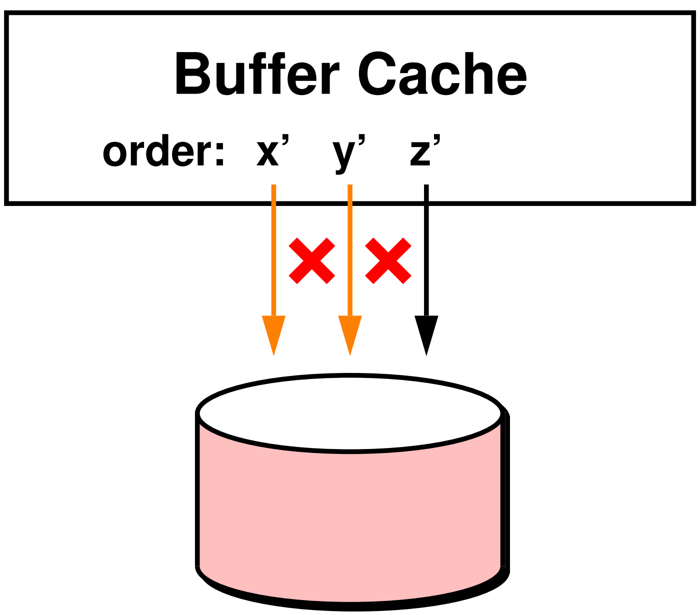
Soft Update Example 2

- Create a new file with one data block

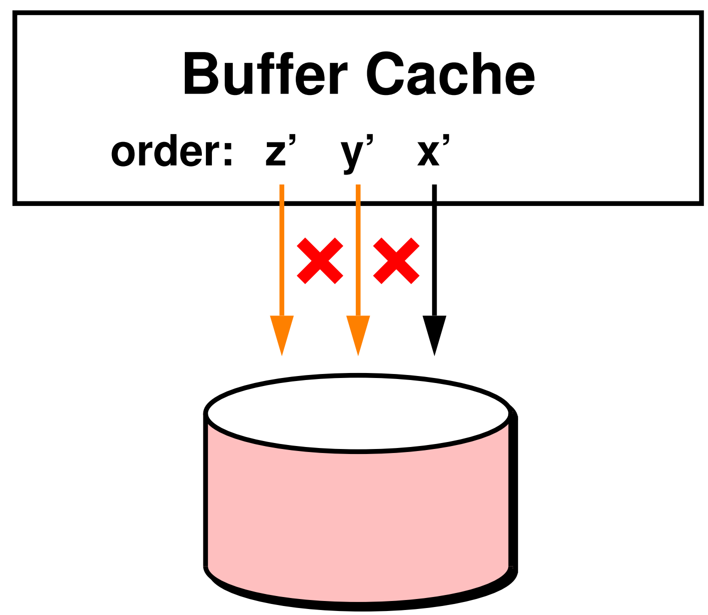




Choice 1

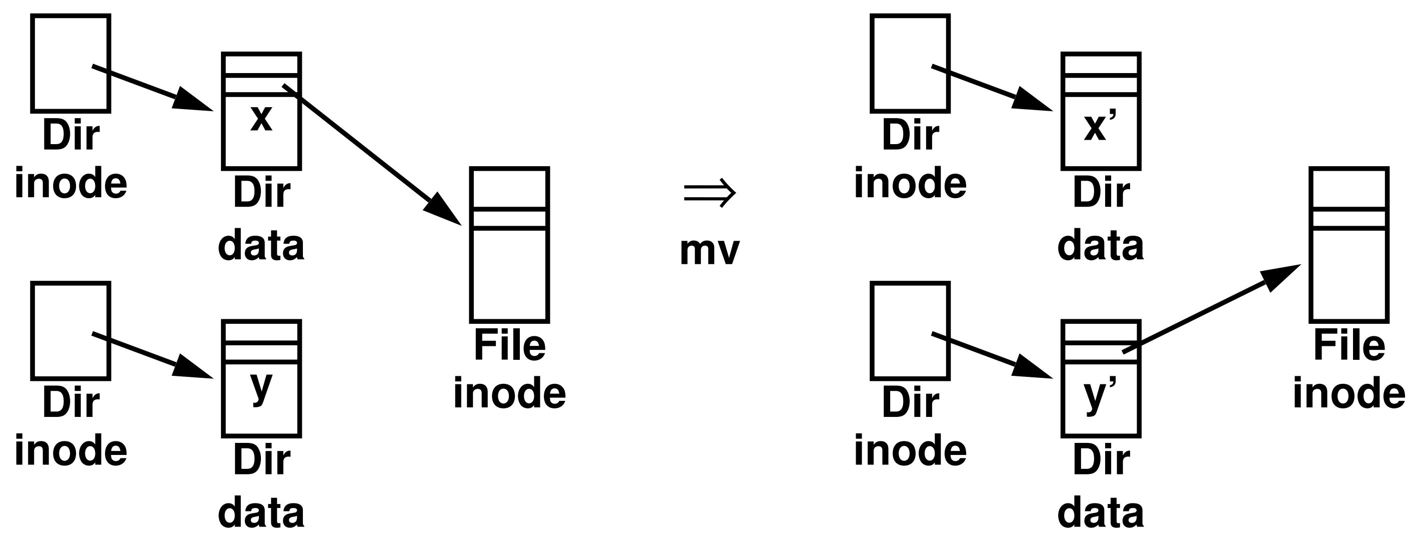


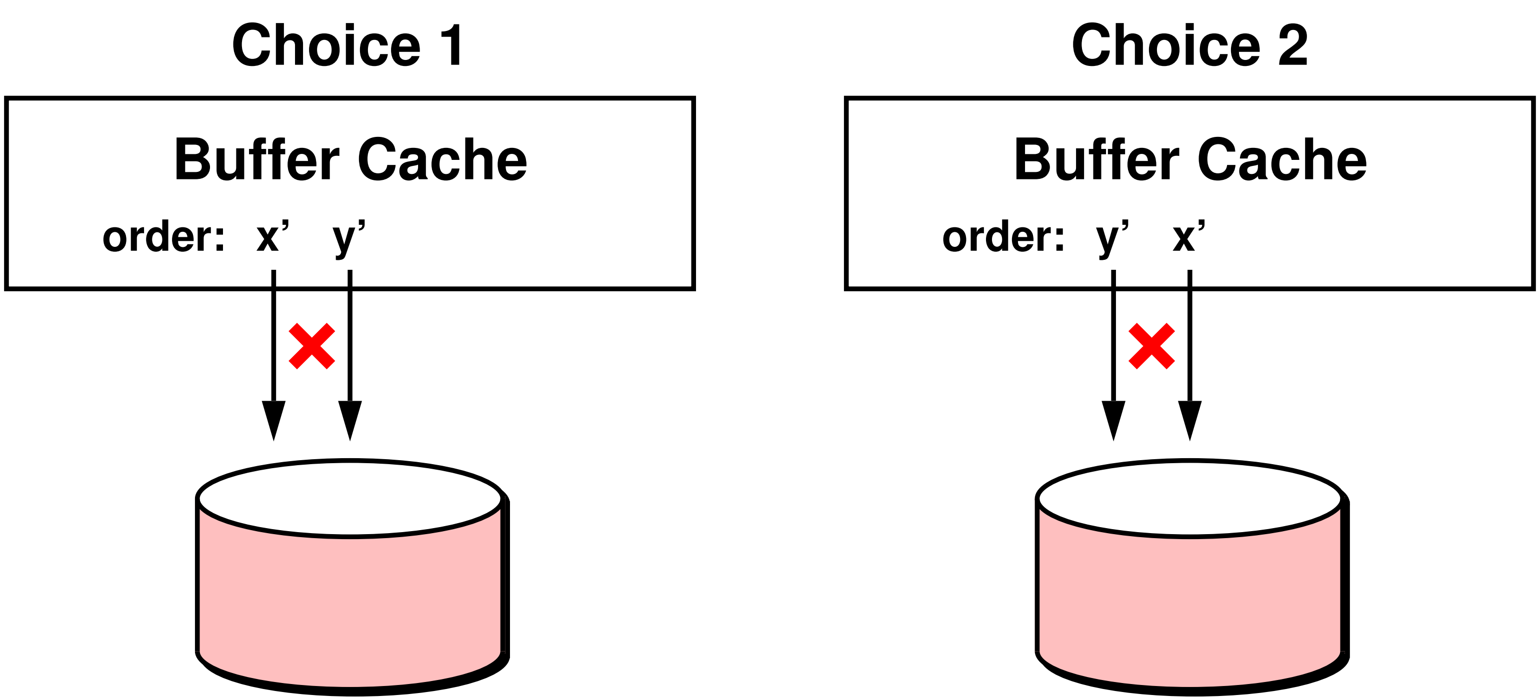
Choice 2



Soft Update Example 3

- Move a file

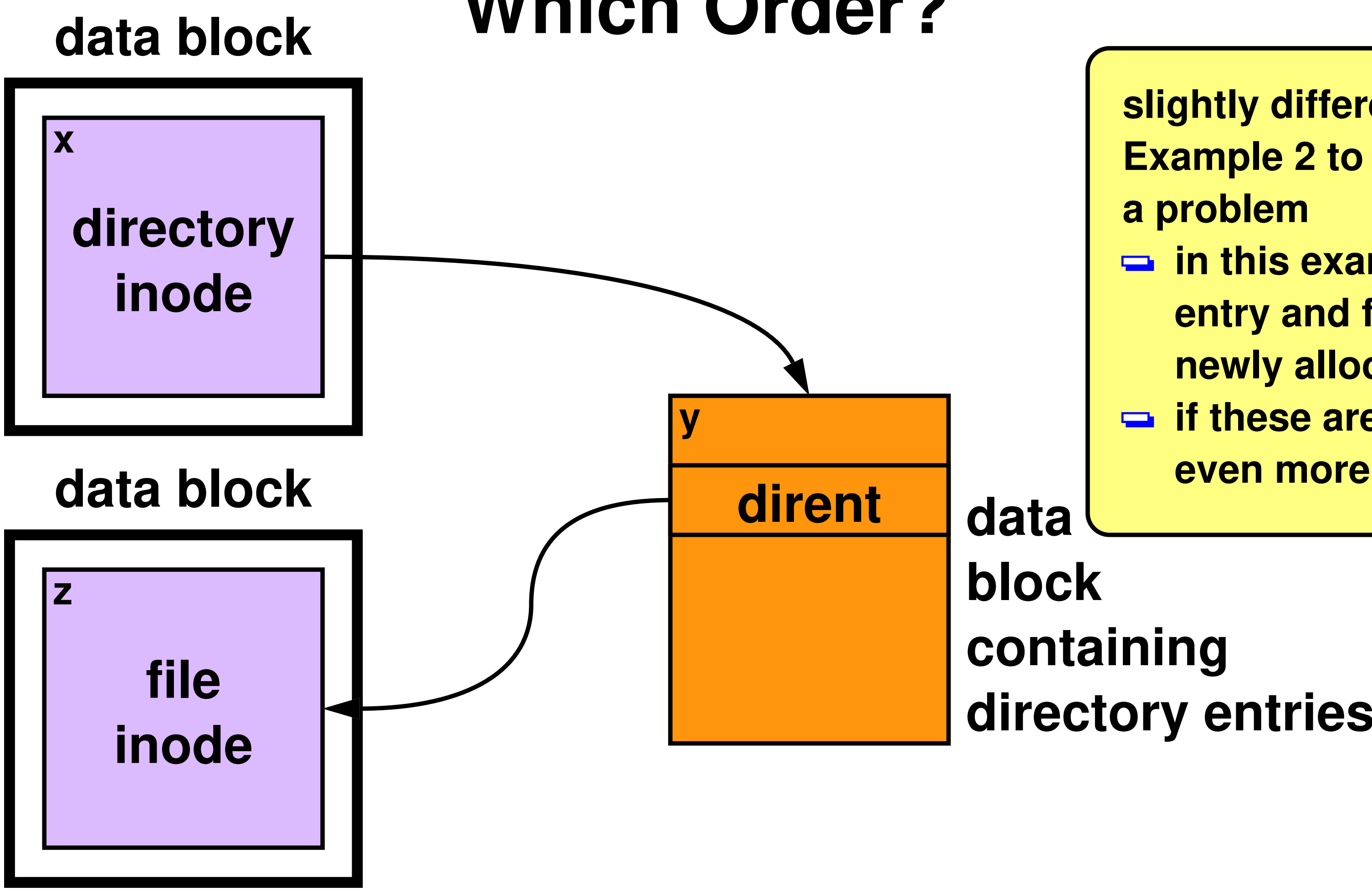




Soft Update

- An implementation of the consistency-preserving approach
  - the idea is simple:
    - update cache in an order that maintains consistency
    - write cache contents to disk in same order in which cache was updated
  - reality isn't
    - (assuming speed is important)

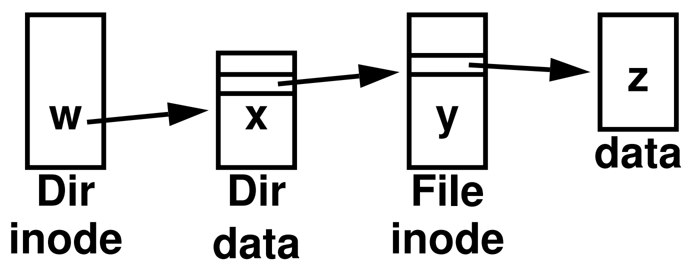
Which Order?



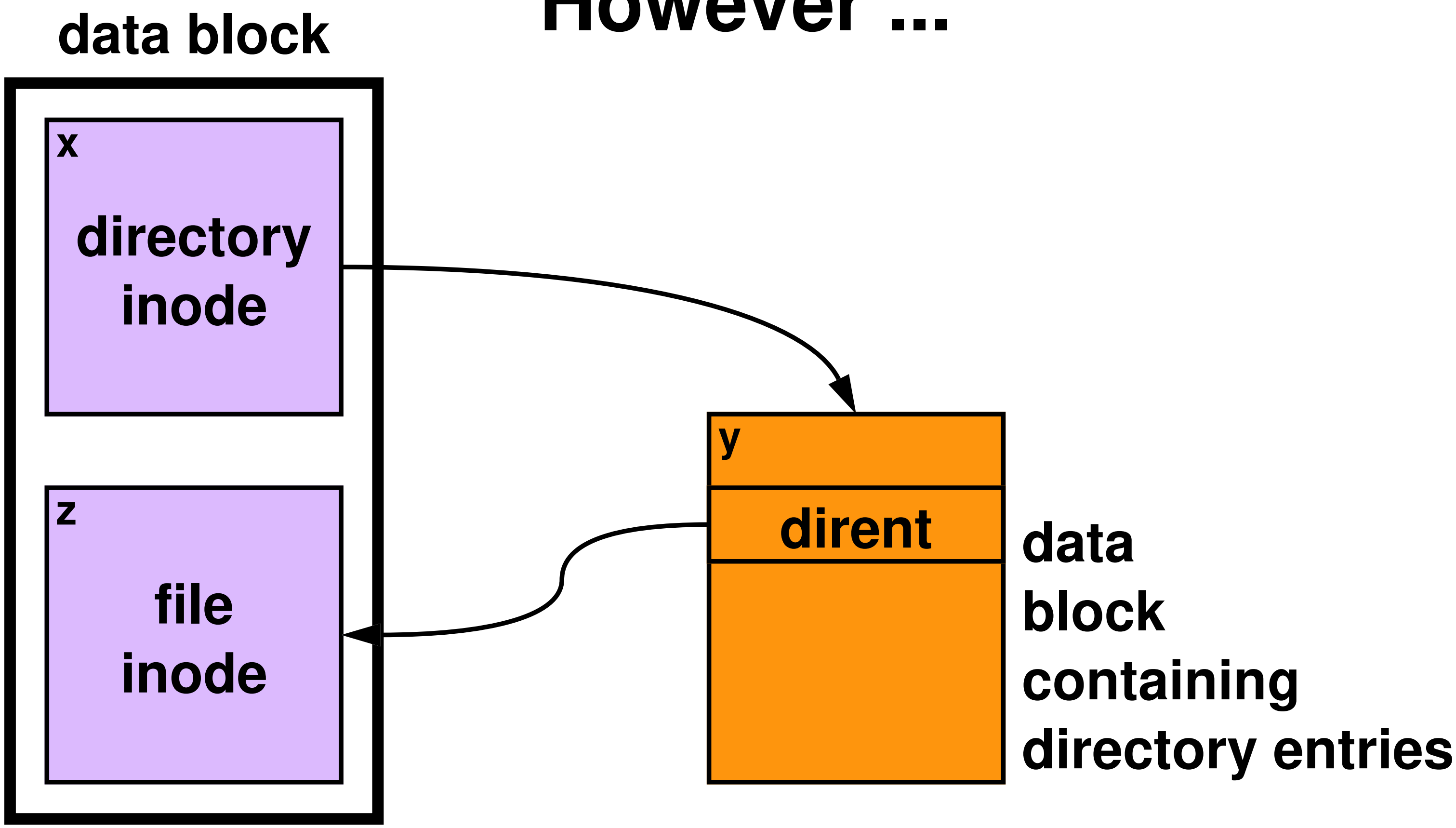
slightly different from Example 2 to illustrate a problem

- in this example, directory entry and file inode are newly allocated blocks
- if these are existing blocks even more complicated

This looks like Example 2 before with simple dependencies, but...

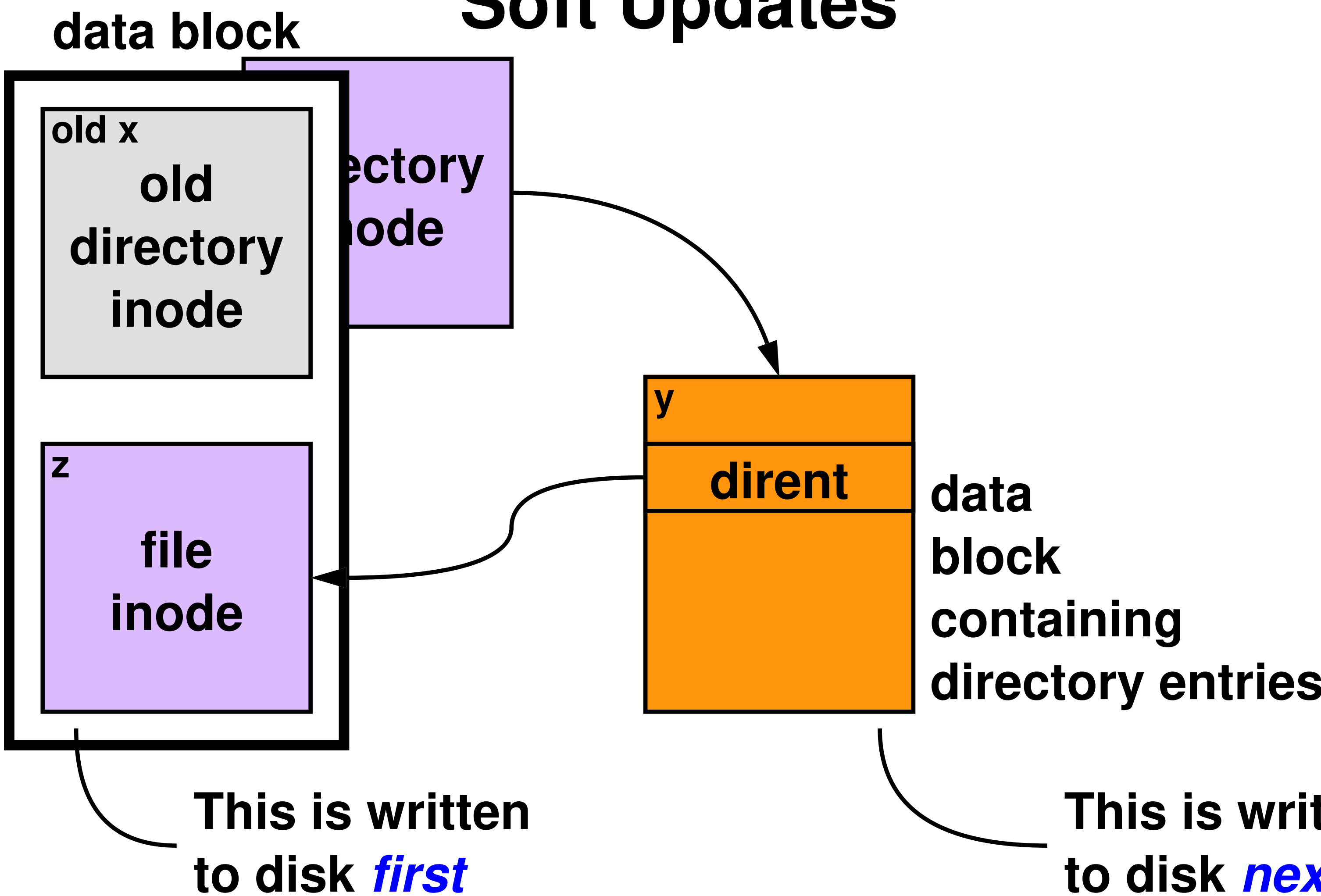


However ...



- This looks like Example 2 before with simple dependencies, but...
  - in reality, in order to *reduce the number of disk writes*, multiple objects can be *packed* into a disk block
  - *primary problem* with soft update: *circular dependency*

Soft Updates



- This looks like Example 2 before with simple dependencies, but...
  - breaking circular dependency
    - 3 steps, with 2 synchronous writes
    - slow

Soft Updates in Practice

- Implemented for FFS in 1994
- Used in FreeBSD's FFS
  - improves performance (over FFS with synchronous writes)
  - disk updates may be *many seconds* behind cache updates
  - need to *reclaim lost disk blocks* as background activity after the system restarts