Housekeeping (Lecture 16 - 10/21/2013)

- Kernel #1 due at 11:45pm this Friday, 10/25/2013
  - if you have code from a previous semester, be very careful and *not copy any code from it*
    - it's best if you just get rid of it
- If you modify additional files, make sure you include them in your submission
  - you would need to change your top-level Makefile in this case
  - you should be able to get the assignment to work without having to do this, but it's okay that you have to change other files
- *Grading guidelines* is the only way we will grade
  - make sure you have tried everything there
- After submission, make sure you *Verify Your Kernel Submission*
- *Midterm exam* coverage posted
  - see the News section of the class web page

6.1 The Basics of File Systems

- UNIX’s S5FS
- Disk Architecture
- *Problems with S5FS*
- Improving Performance
- Dynamic Inodes

S5FS on Rhinopias (A Marketing Disaster ...)

- Rhinopias's maximum transfer speed?
  - 63.9 MB/sec

- S5FS's average transfer speed on Rhinopias?
  - average seek time:
    - < 4 milliseconds (say 2)
  - average rotational latency:
    - ~3 milliseconds
  - per-sector transfer time:
    - negligible
  - time/sector: 5 milliseconds
  - transfer time: 102.4 KB/sec (.16% of maximum)

6.1 The Basics of File Systems

- UNIX’s S5FS
- Disk Architecture
- Problems with S5FS
- *Improving Performance*
- Dynamic Inodes

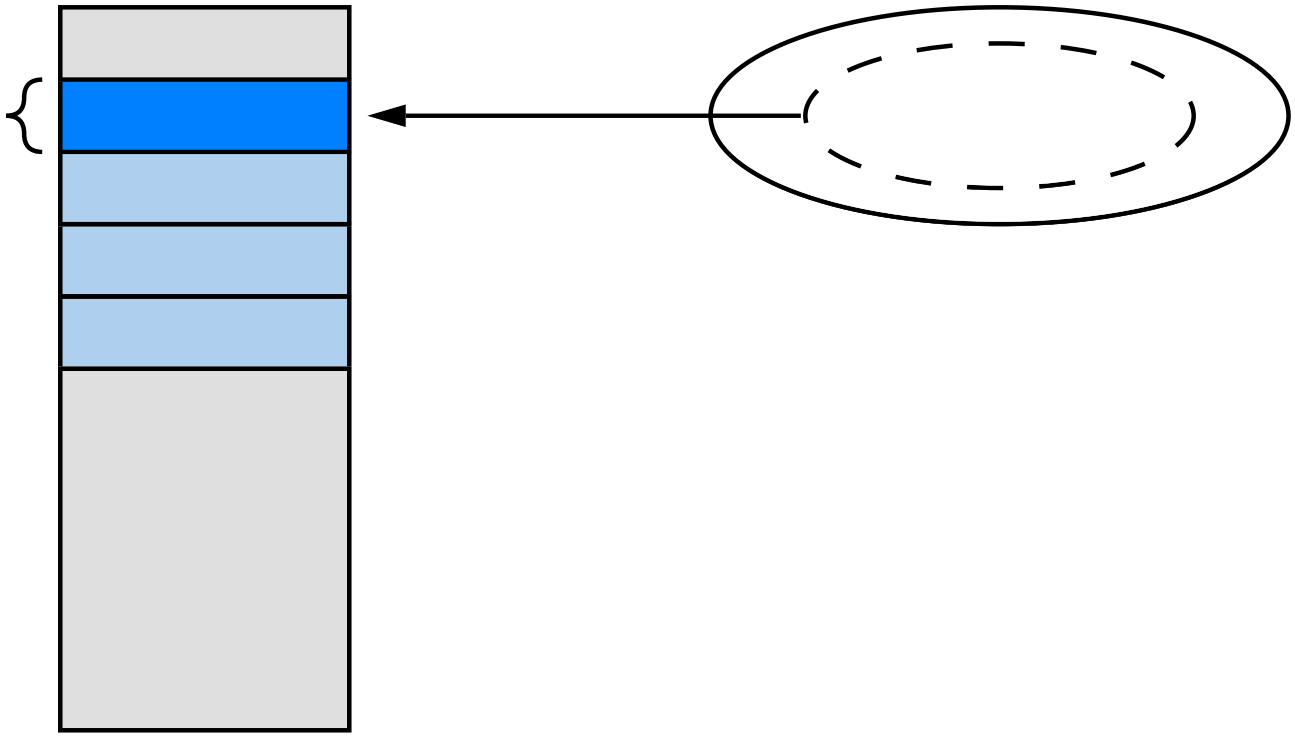
What to Do About It?

- Hardware
  - employ pre-fetch buffer
    - filled by hardware with what’s underneath head
    - helps reads a bit; doesn’t help writes
- Software
  - better on-disk data structures
    - increase block size
    - minimize seek time
    - reduce rotational latency

FFS

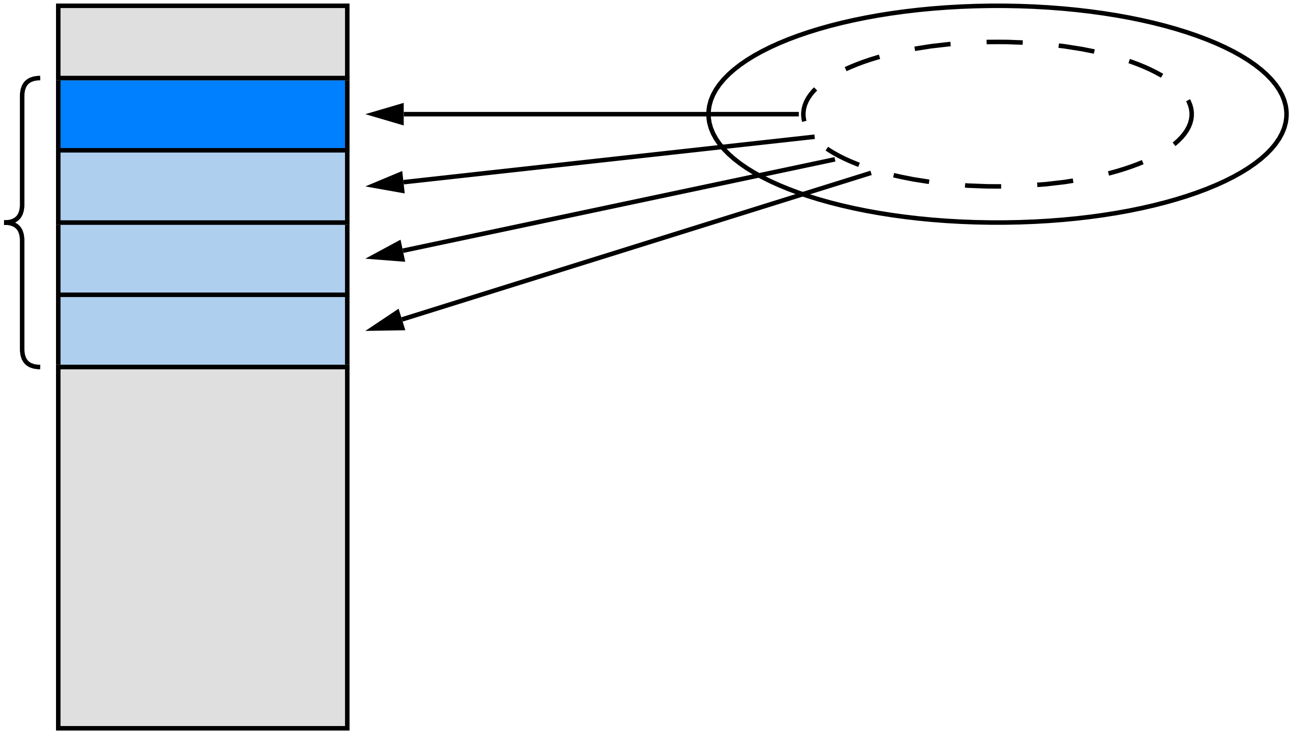
- Better on-disk organization
- Longer component names in directories
- Retains disk map of S5FS

Larger Block Size



Not just this

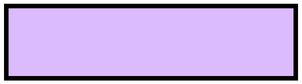
Larger Block Size



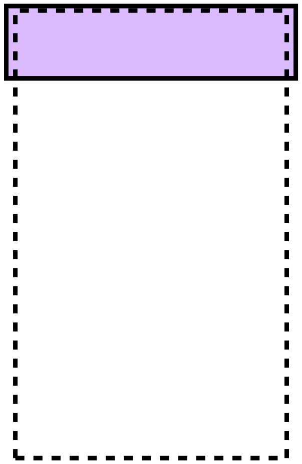
But all this

The Down Side ...

Smaller
Block Size



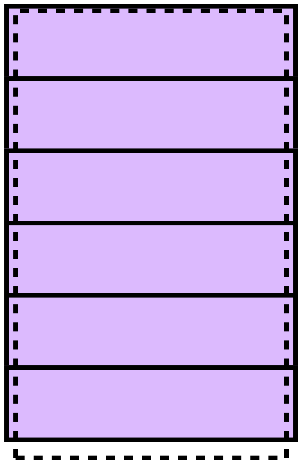
The Down Side ...



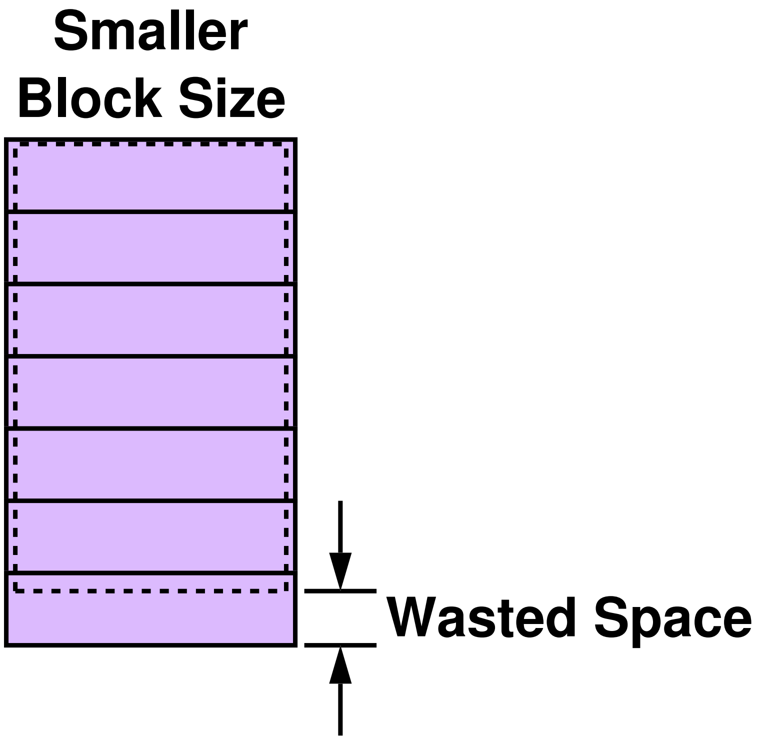
Smaller Block Size

The Down Side ...

Smaller
Block Size

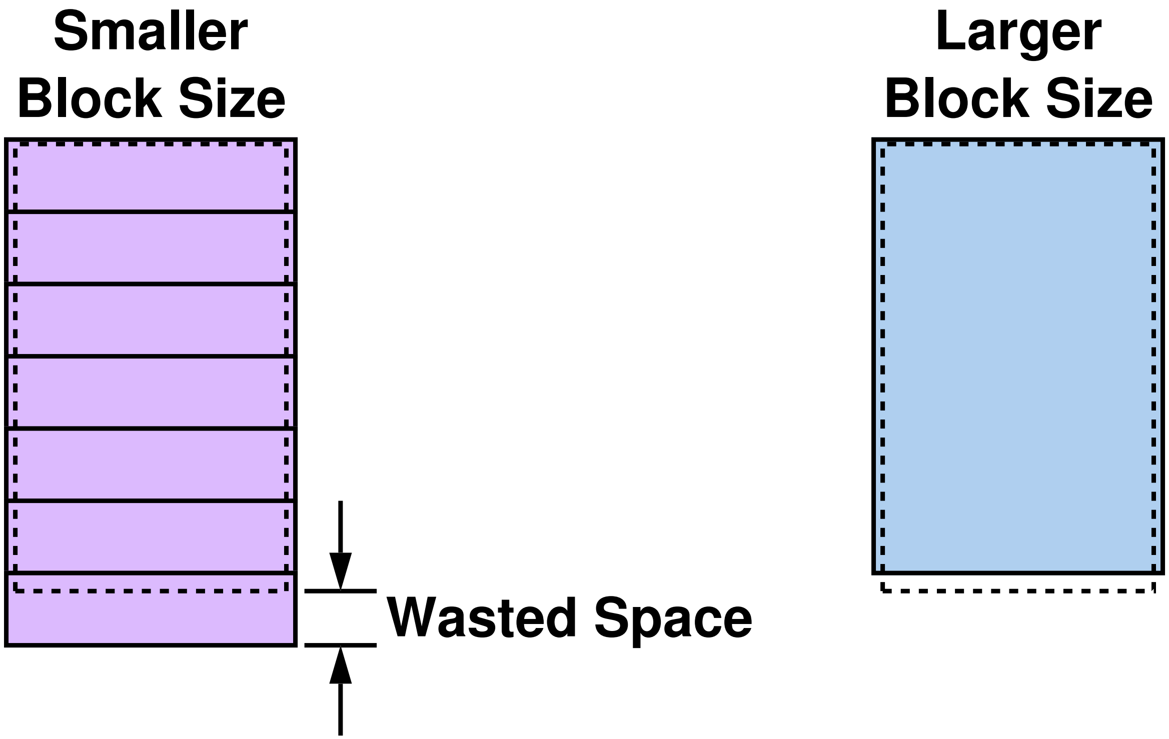


The Down Side ...

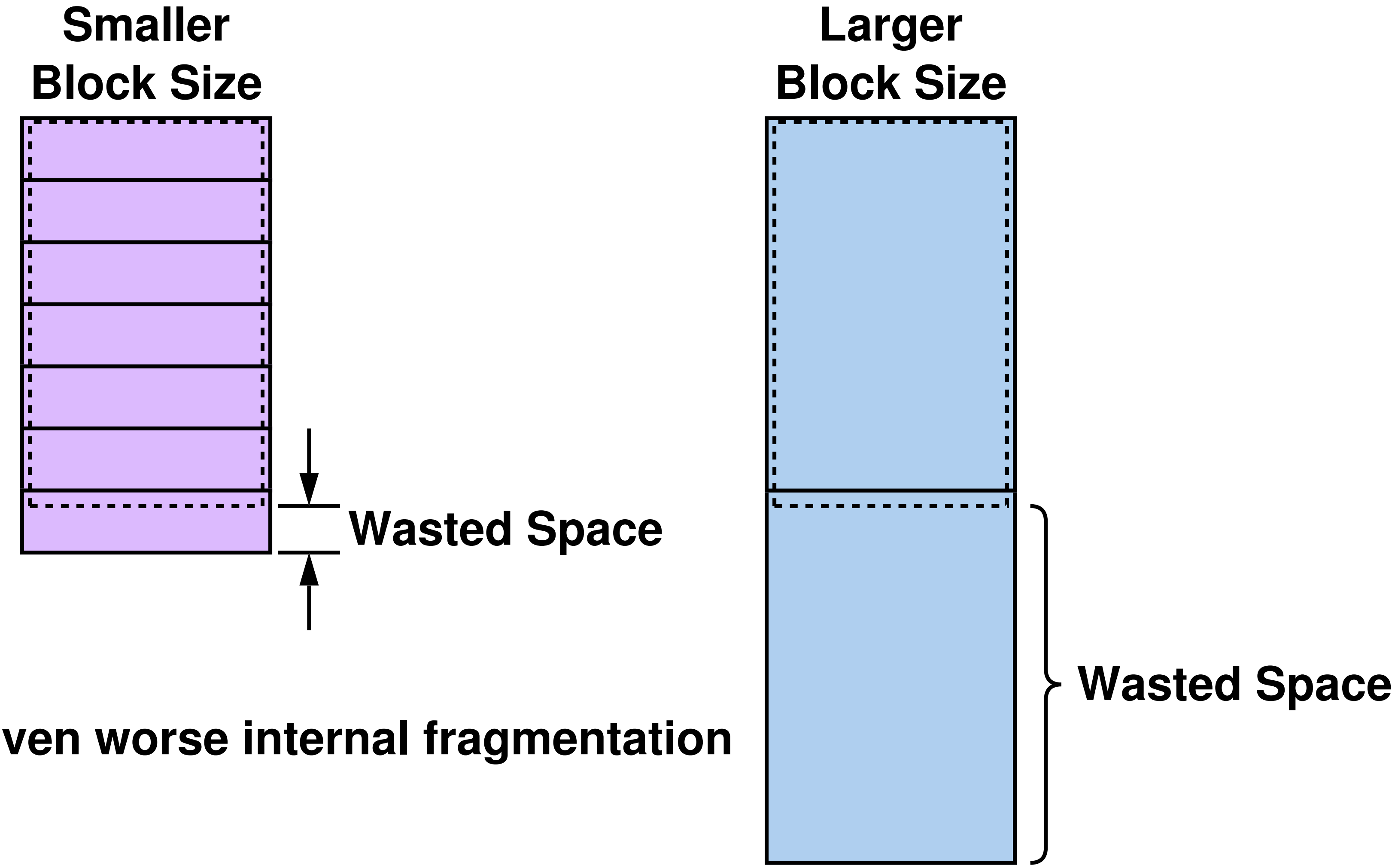


- internal fragmentation

The Down Side ...

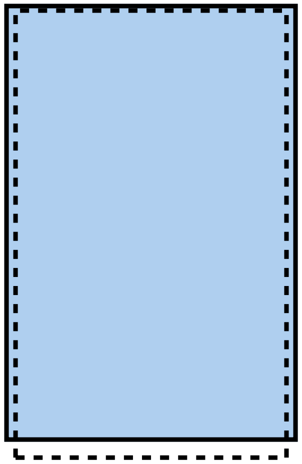


The Down Side ...

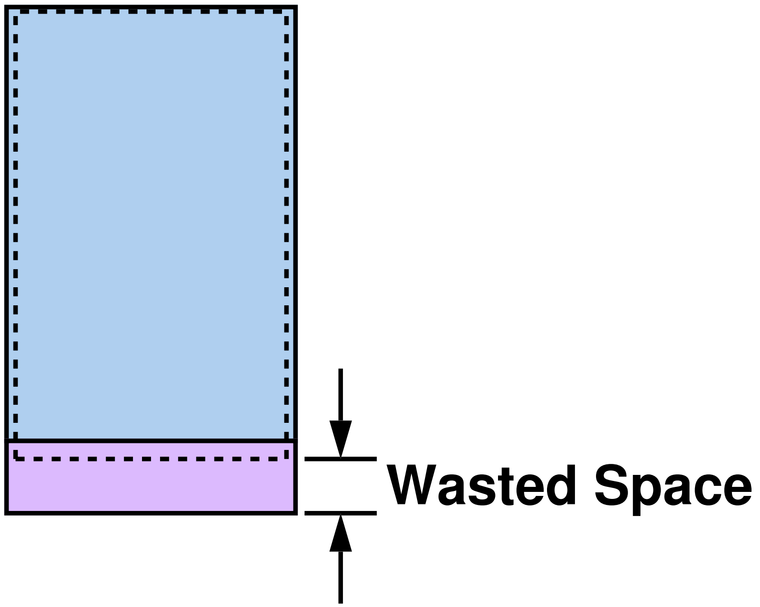


- even worse internal fragmentation

Two Block Sizes ...



Two Block Sizes ...

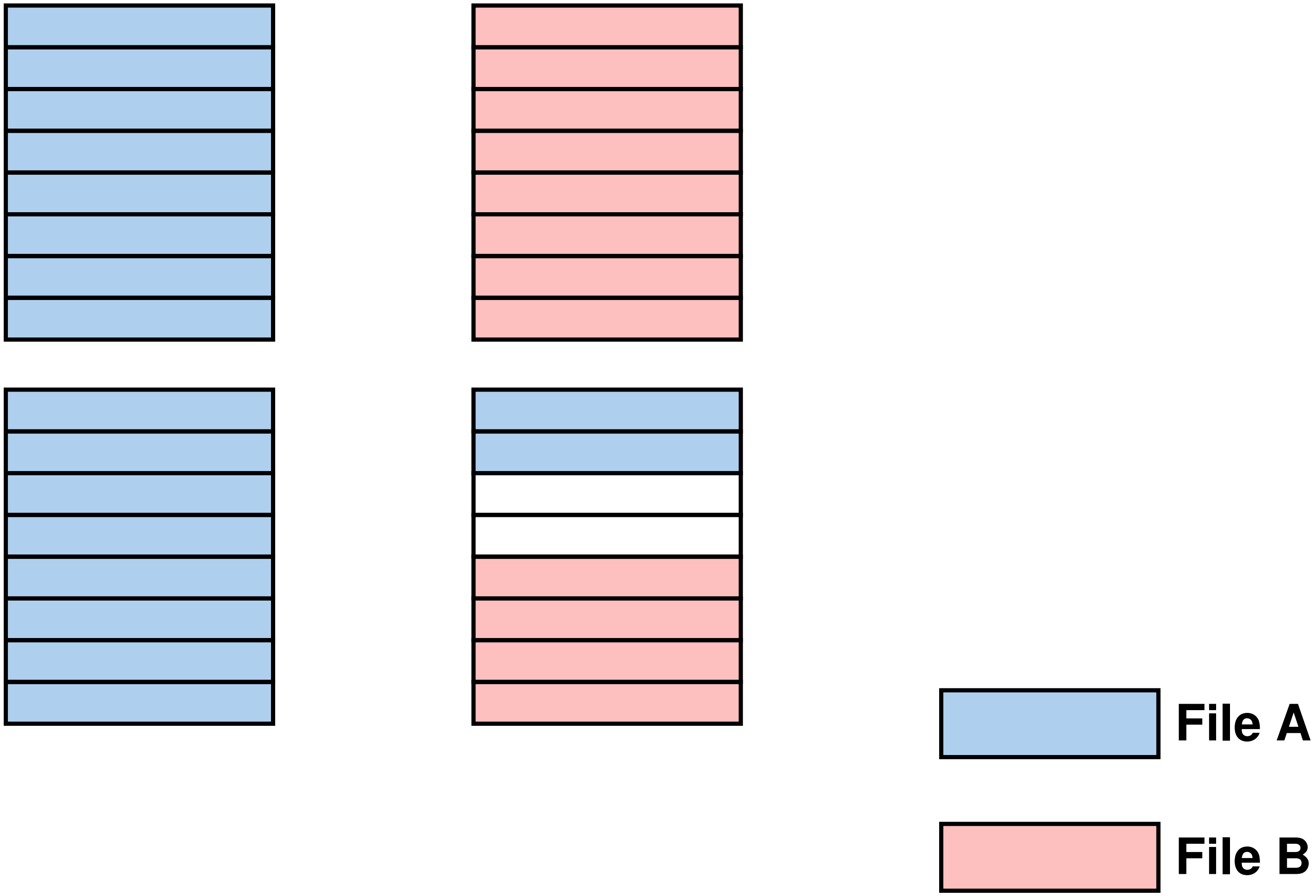


- e.g., 16KB blocks and 1KB fragments
- best of both worlds

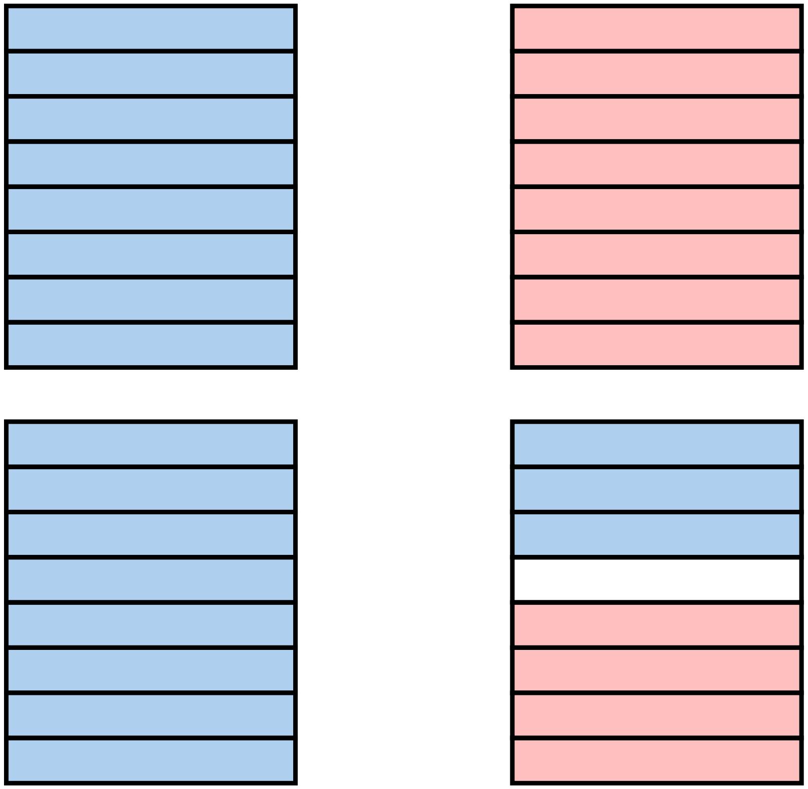
Rules

- File-system blocks may be split into fragments that can be independently assigned to files
  - fragments assigned to a file must be contiguous and in order
- The number of fragments per block (1, 2, 4, or 8) is fixed for each file system
- Allocation in fragments may only be done on what would be the last block of a file, and only for small files

Use of Fragments (1)



Use of Fragments (2)

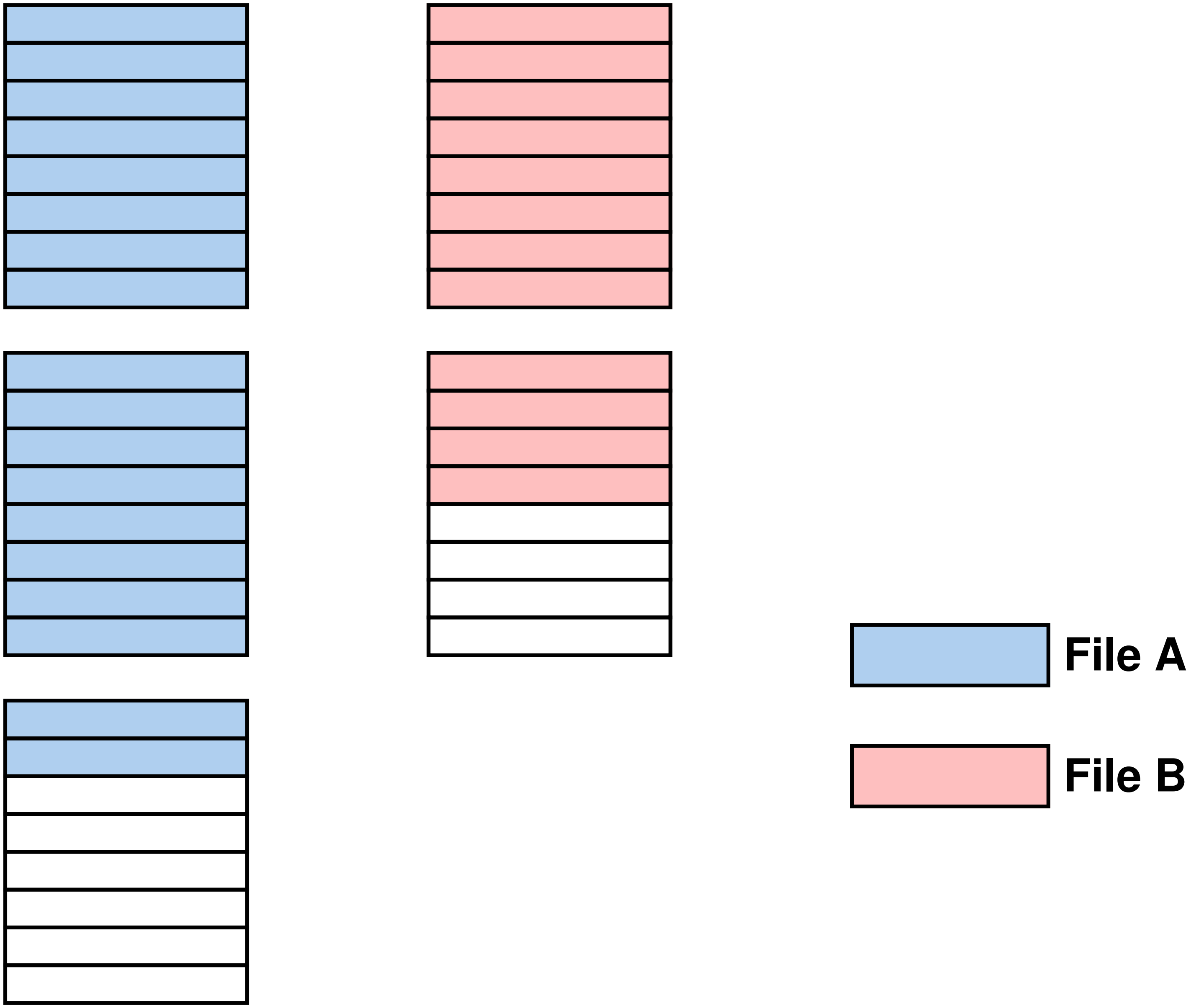


🟦 File A

🟥 File B

- A can grow by 2 segments

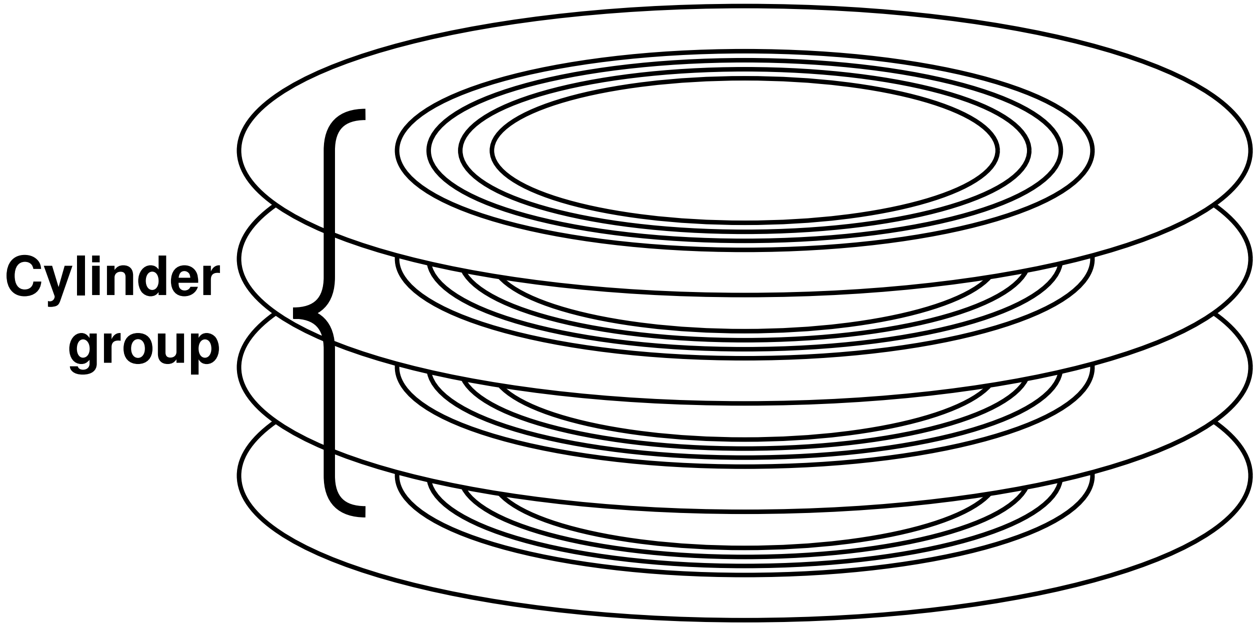
Use of Fragments (3)



Minimizing Seek Time

- Keep related things close to one another
- Separate unrelated things

Cylinder Groups

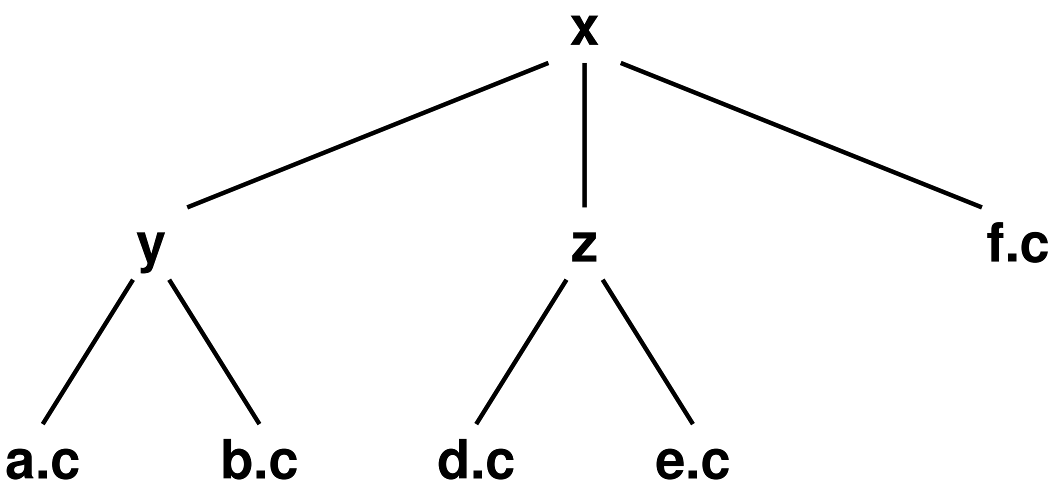


- recall that *seeking* to the *next sector* is much *faster*

Minimizing Seek Time

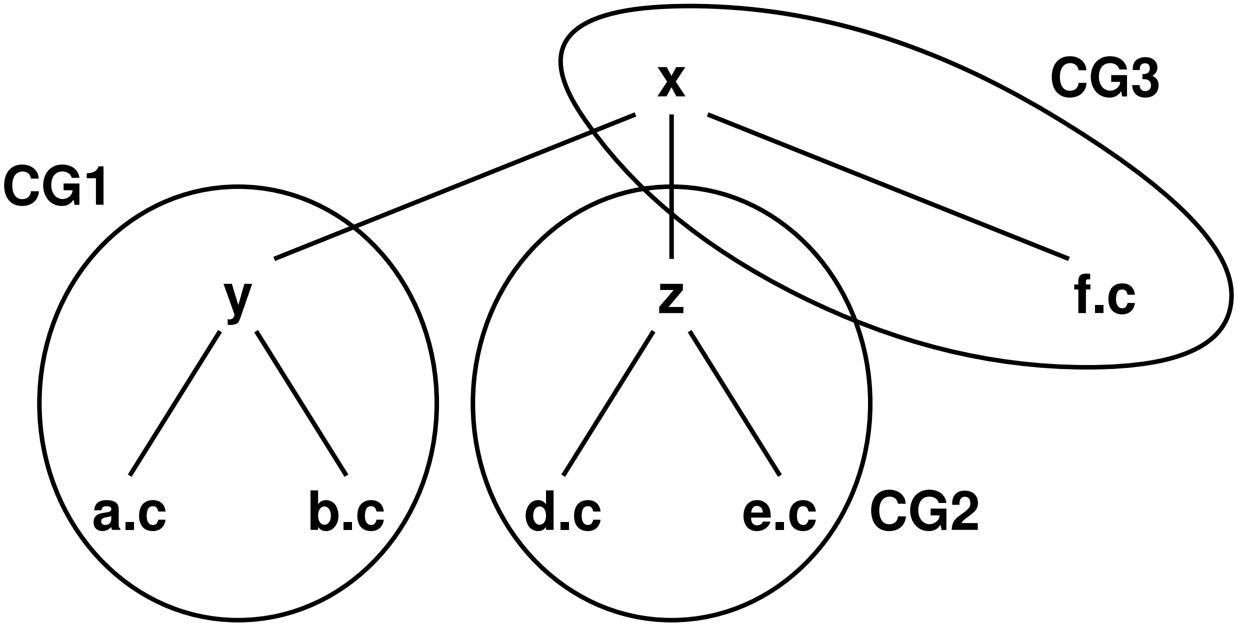
- **The practice:**
  - attempt to put new inodes in the same cylinder group as their directories
  - put inodes for new directories in cylinder groups with "lots" of free space
  - put the beginning of a file (first 10KB, i.e., direct blocks) in the inode's cylinder group
  - put additional portions of the file (each 2MB) in cylinder groups with "lots" of free space

Locality Of File Access



- if access "d.c", likely to access "e.c"

Locality Of File Access

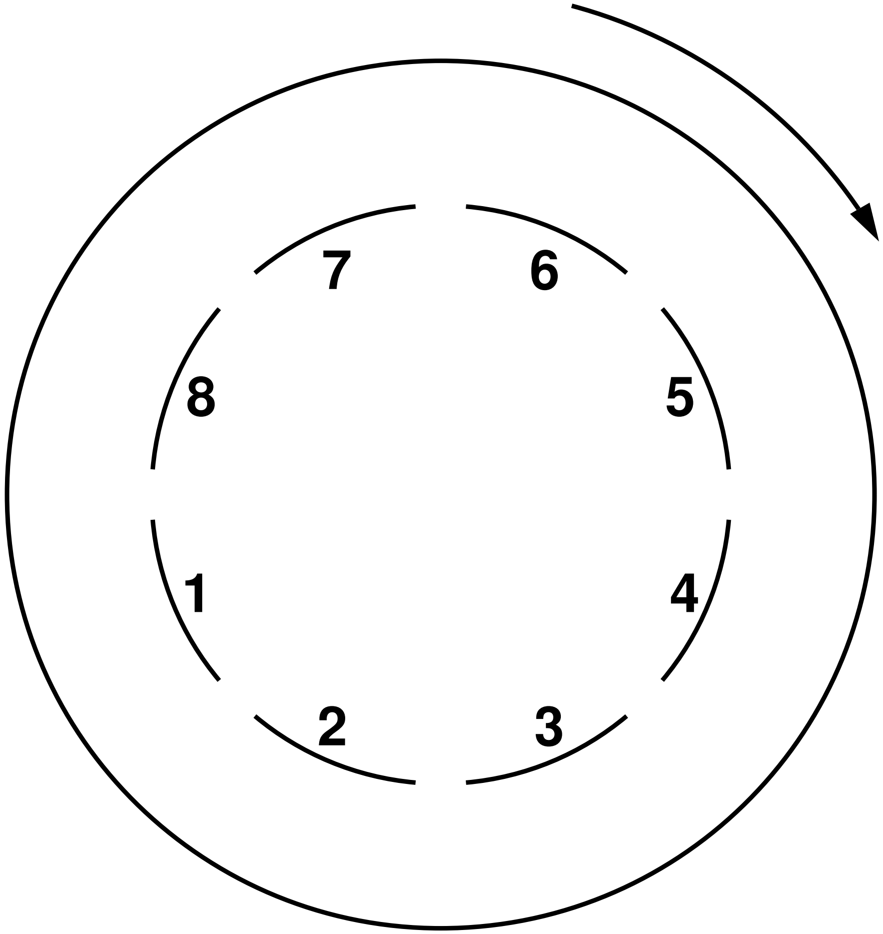


- if access "d.c", likely to access "e.c"

How Are We Doing?

- Configure Rhinopias with 20 cylinders per group
  - 2-MB file fits entirely within one cylinder group
  - average seek time within cylinder group is ~.3 milliseconds
  - average rotational delay still 3 milliseconds
  - .12 milliseconds required for disk head to pass over 8KB block
  - 3.42 milliseconds for each block
  - 2.4 million bytes/second average transfer time
  - 20-fold improvement
  - 3.7% of maximum possible

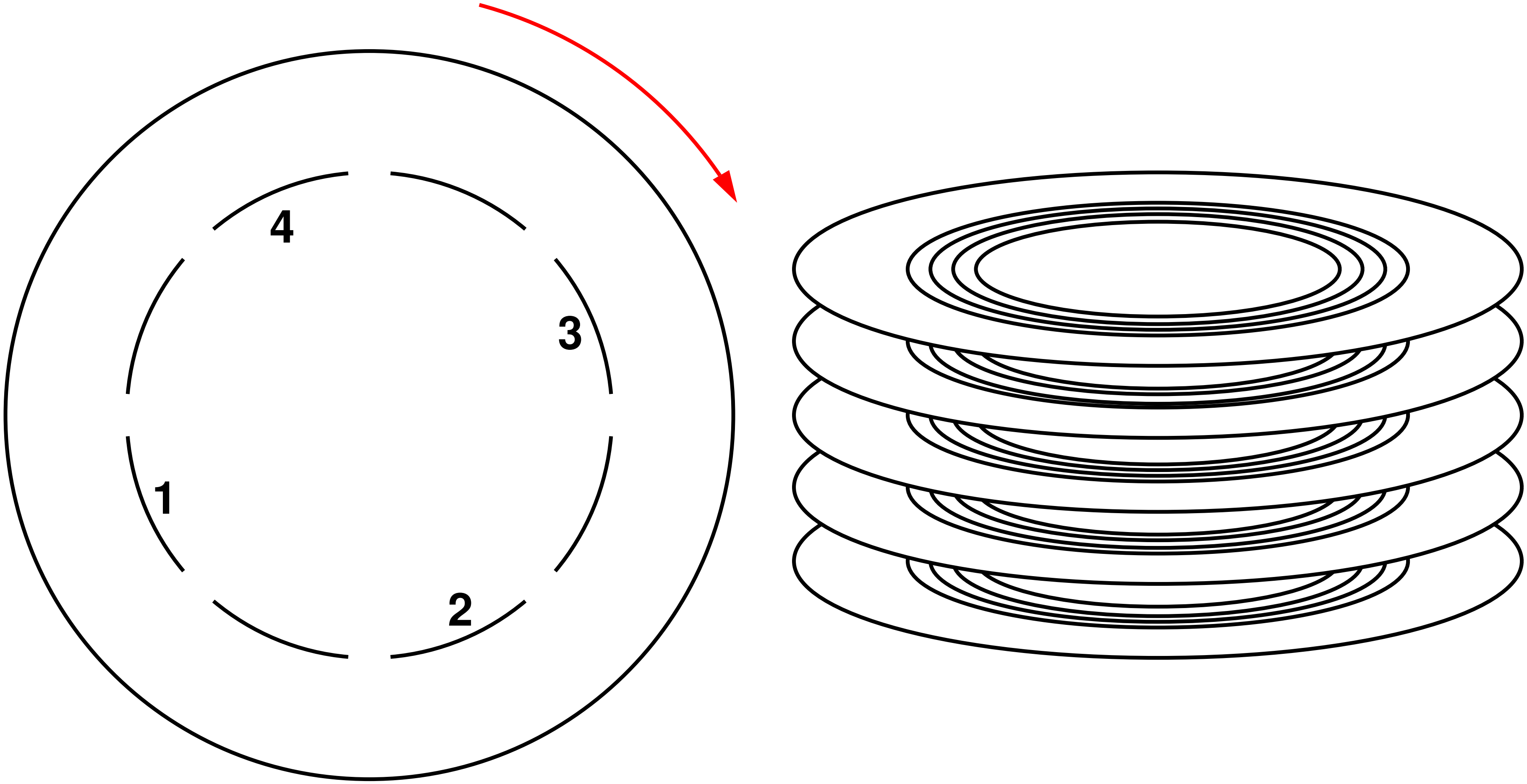
Minimizing Latency (1)



Numbers

- Rhinopias spins at 10,000 RPM
  - 6 milliseconds/revolution
- 100 microseconds required to service disk-completion interrupt and start next operation
  - typical of early 1980s
- Each block takes 120 microseconds to traverse disk head
- Reading successive blocks is expensive!

Minimizing Latency (2)



- Block interleaving

How’re We Doing Now? (part 1)

- Time to read successive blocks (two-way interleaving):
  - after request for second block is issued, must wait 20 microseconds for the beginning of the block to rotate under disk head
  - factor of 300 improvement!

How’re We Doing Now? (part 2)

- Same setup as before
  - 2-MB file within one cylinder group
  - actually fits in one cylinder
  - block interleaving employed: every other block is skipped
  - .3-millisecond seek to that cylinder
  - 3-millisecond rotational delay for first block
  - 50 blocks/track, but 25 read in each revolution
  - 10.24 revolutions required to read all of file
  - 32.4 MB/second (50% of maximum possible)

Further Improvements?

- S5FS: 0.16% of capacity
- FFS without block interleaving: 3.8% of capacity
- FFS with block interleaving: 50% of capacity
- What next?

Larger Transfer Units

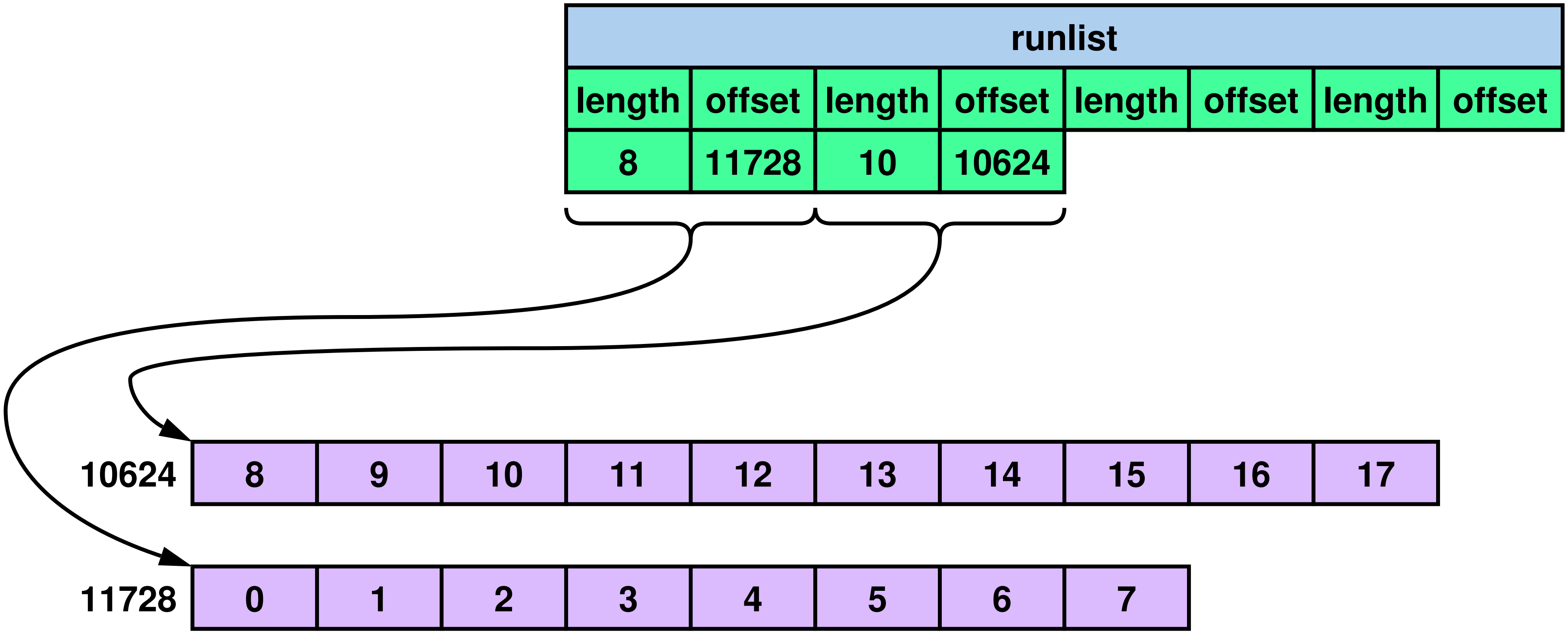
- Allocate in whole tracks or cylinders
  - too much wasted space
- Allocate in blocks, but group them together
  - transfer many at once

Block Clustering

- Allocate space in blocks, eight at a time
- Linux's Ext2 (an FFS clone):
  - allocate eight blocks at a time
  - extra space is available to other files if there is a shortage of space
- FFS on Solaris (~1990)
  - delay disk-space allocation until:
    - 8 blocks are ready to be written
    - or the file is closed

Extents

Windows



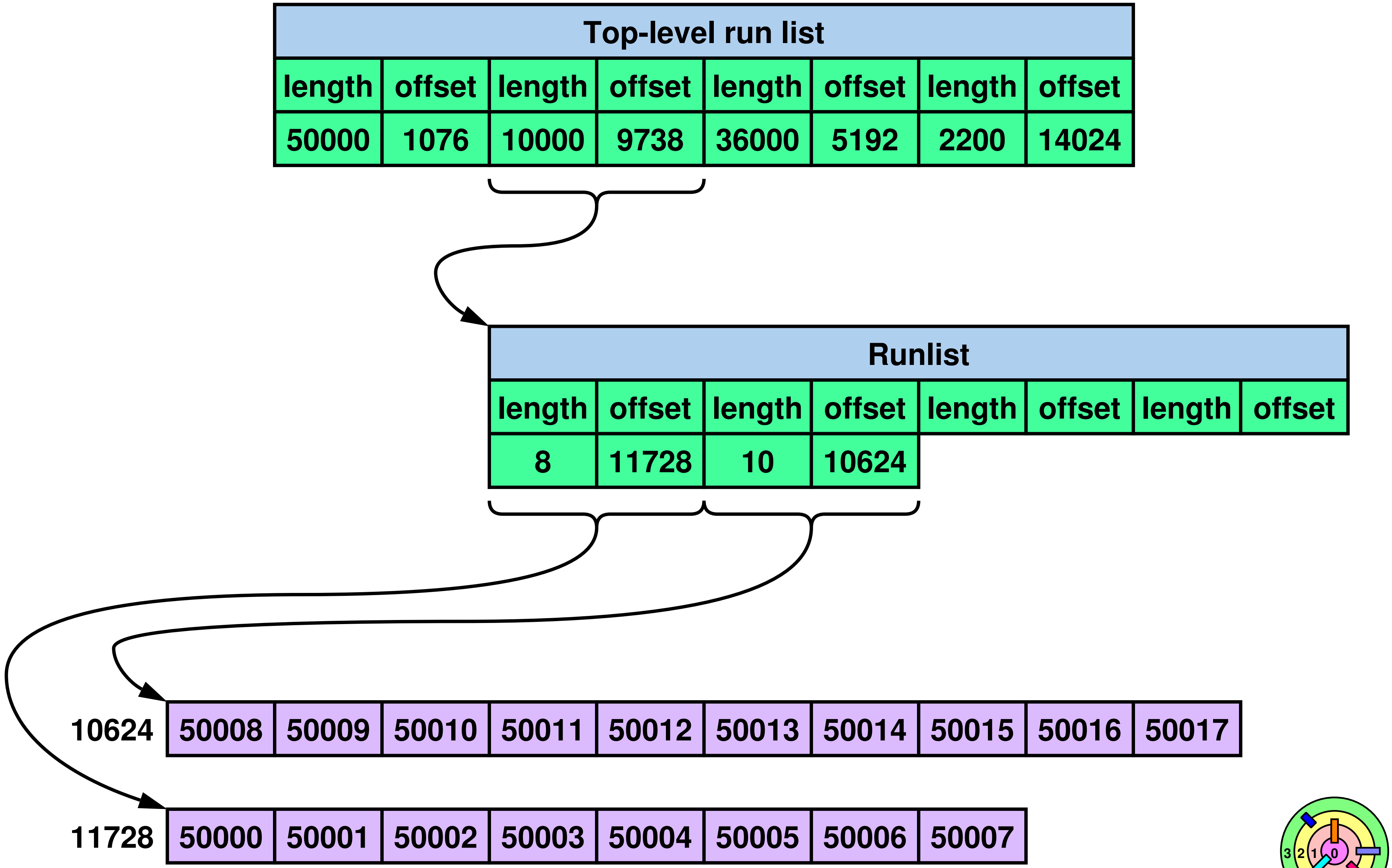
runlist							
length	offset	length	offset	length	offset	length	offset
8	11728	10	10624				

10624	8	9	10	11	12	13	14	15	16	17
11728	0	1	2	3	4	5	6	7		

Problems with Extents

- 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
  - solution: multiple levels

Extents in NTFS



Top-level run list

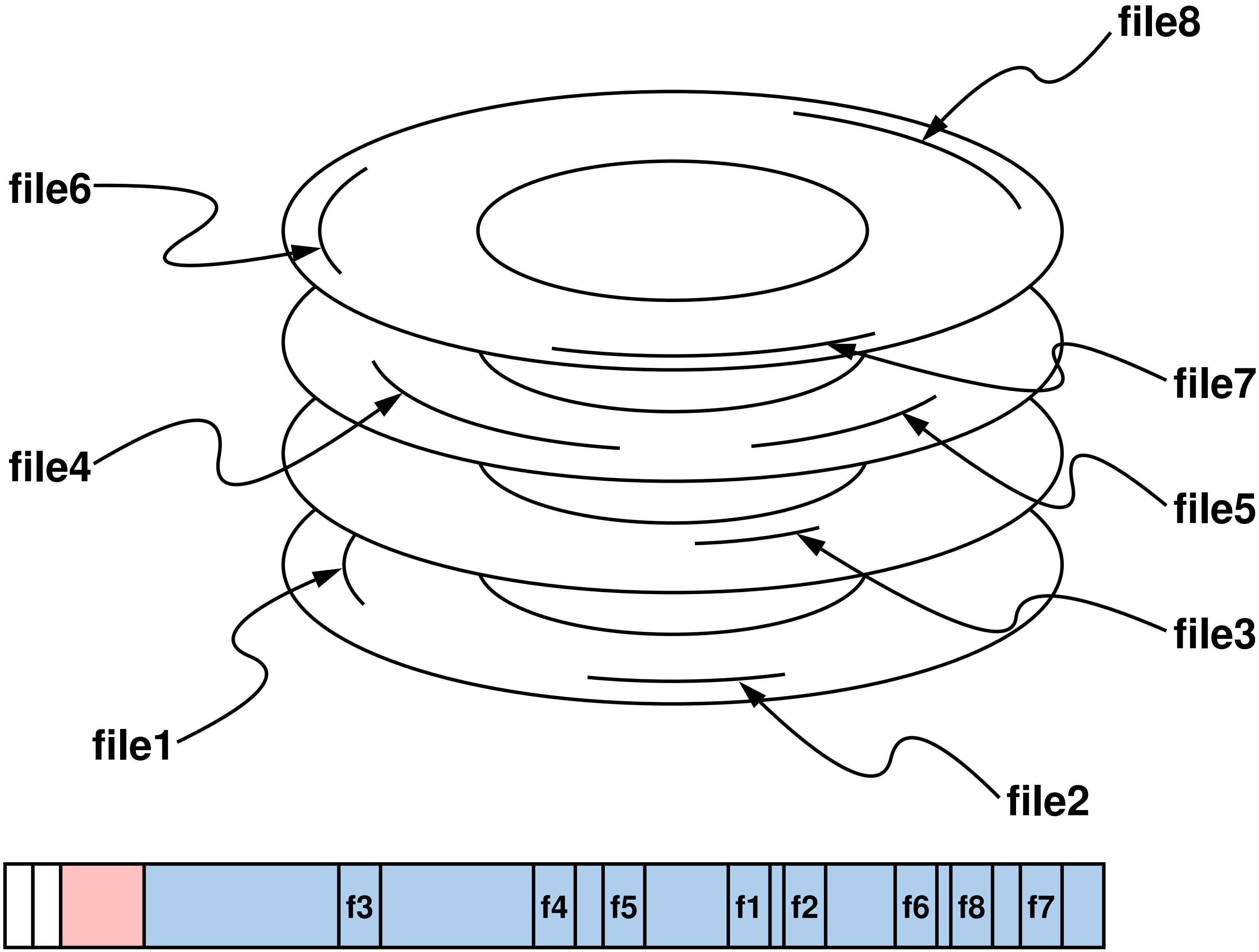
length	offset	length	offset	length	offset	length	offset
50000	1076	10000	9738	36000	5192	2200	14024

Runlist

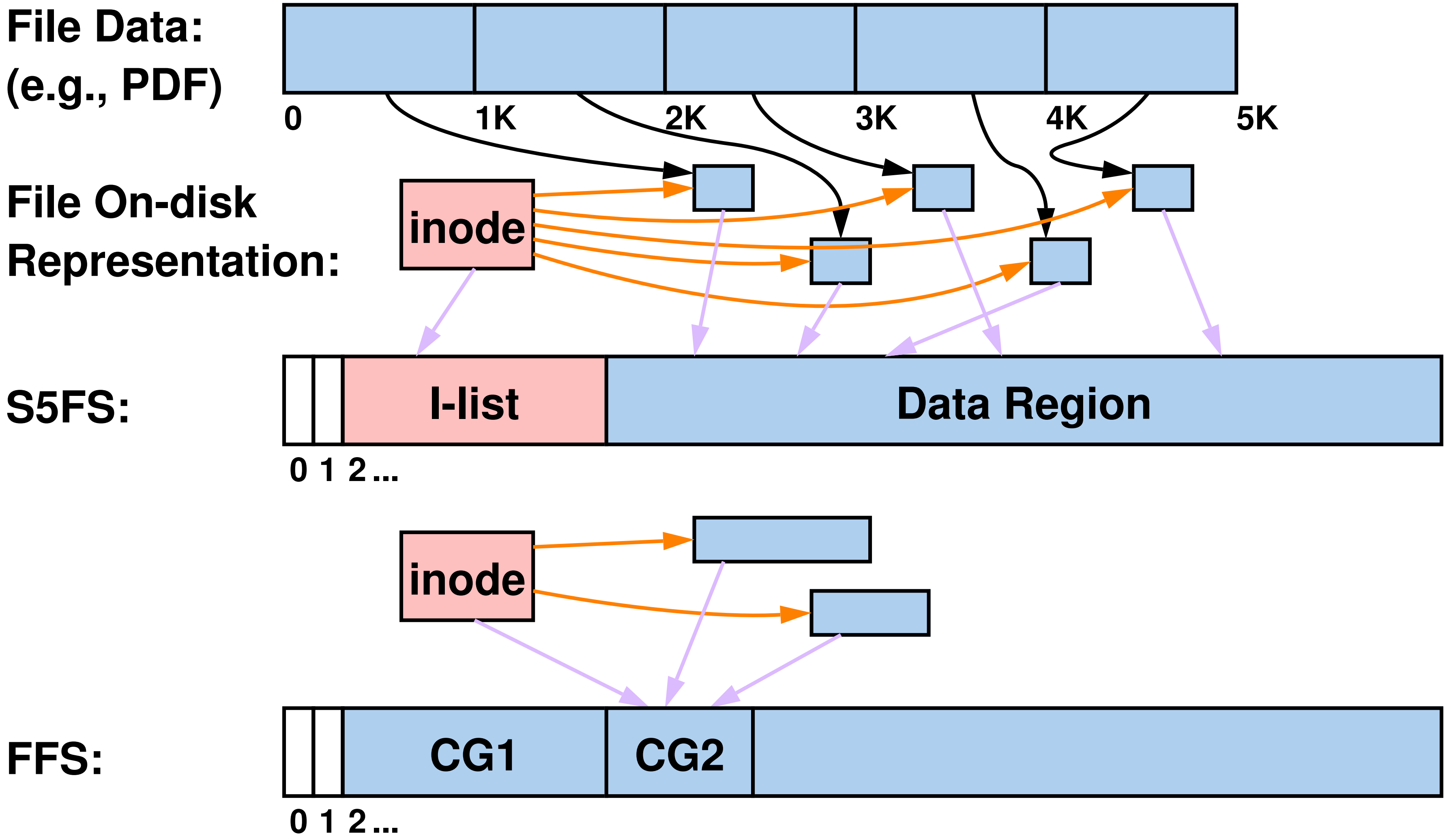
length	offset	length	offset	length	offset	length	offset
8	11728	10	10624				

10624	50008	50009	50010	50011	50012	50013	50014	50015	50016	50017
11728	50000	50001	50002	50003	50004	50005	50006	50007		

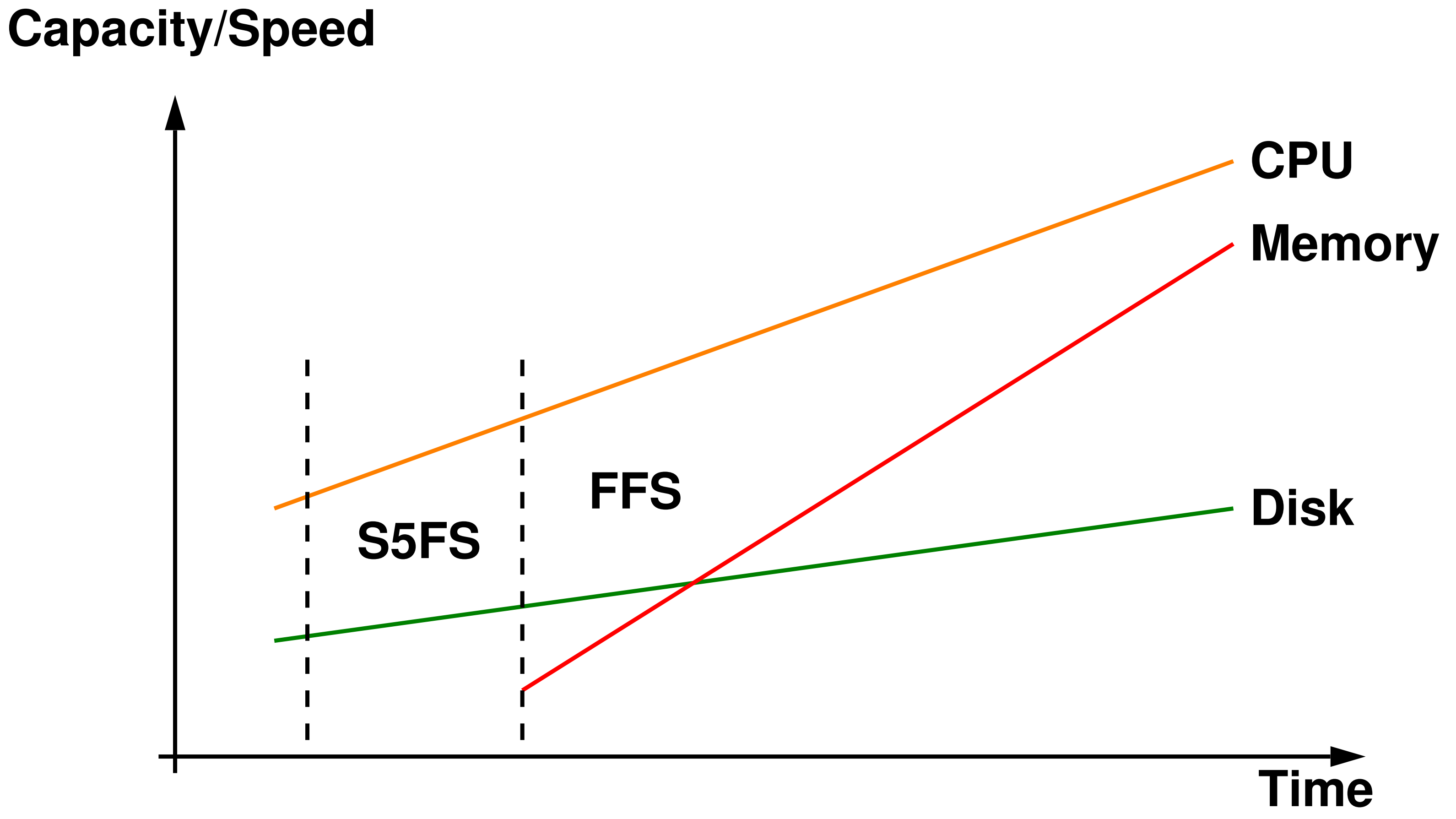
Are We There Yet?



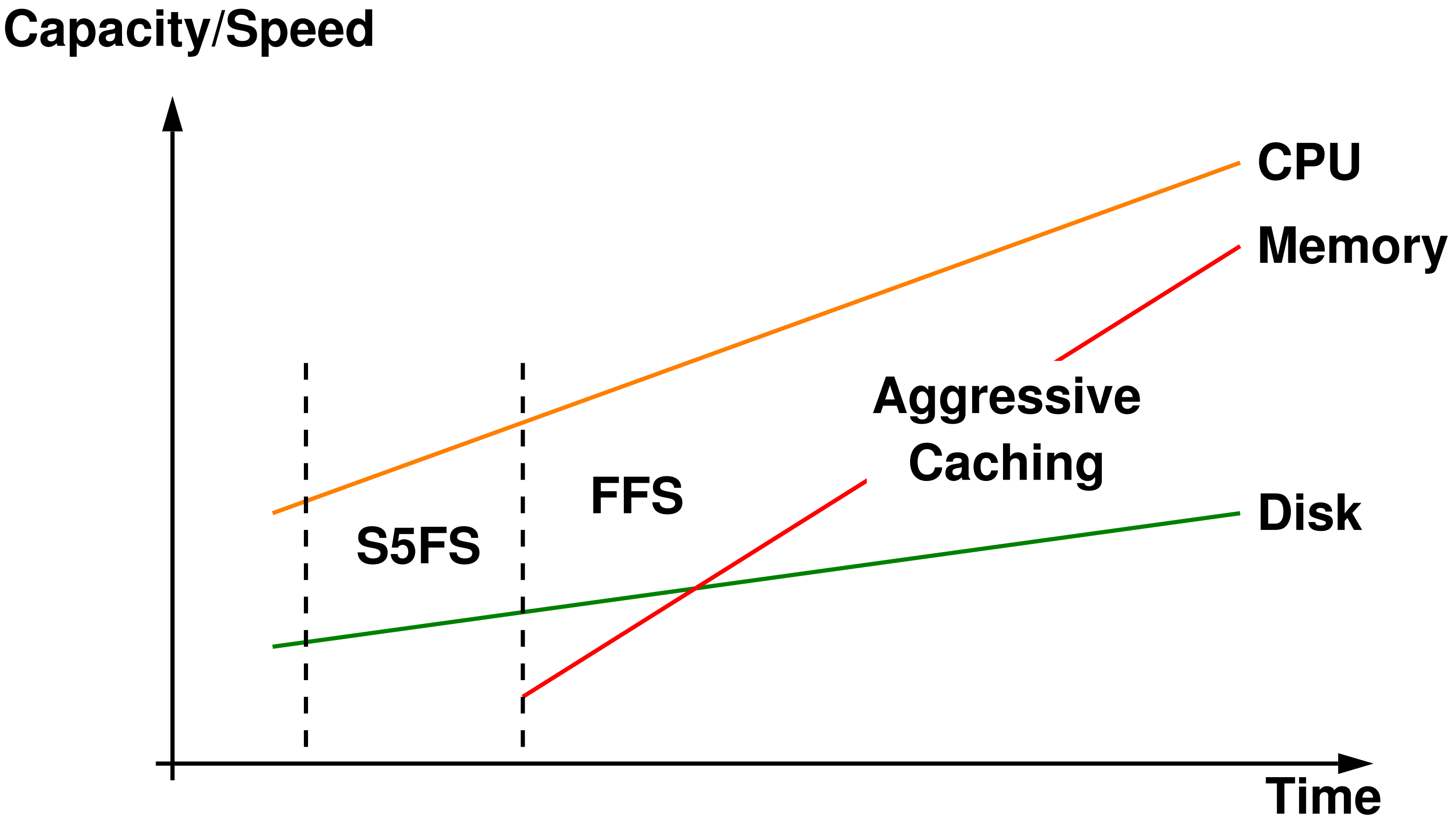
S5FS & FFS Data Placement Example



CPU, Memory, Disk Speeds Over Time



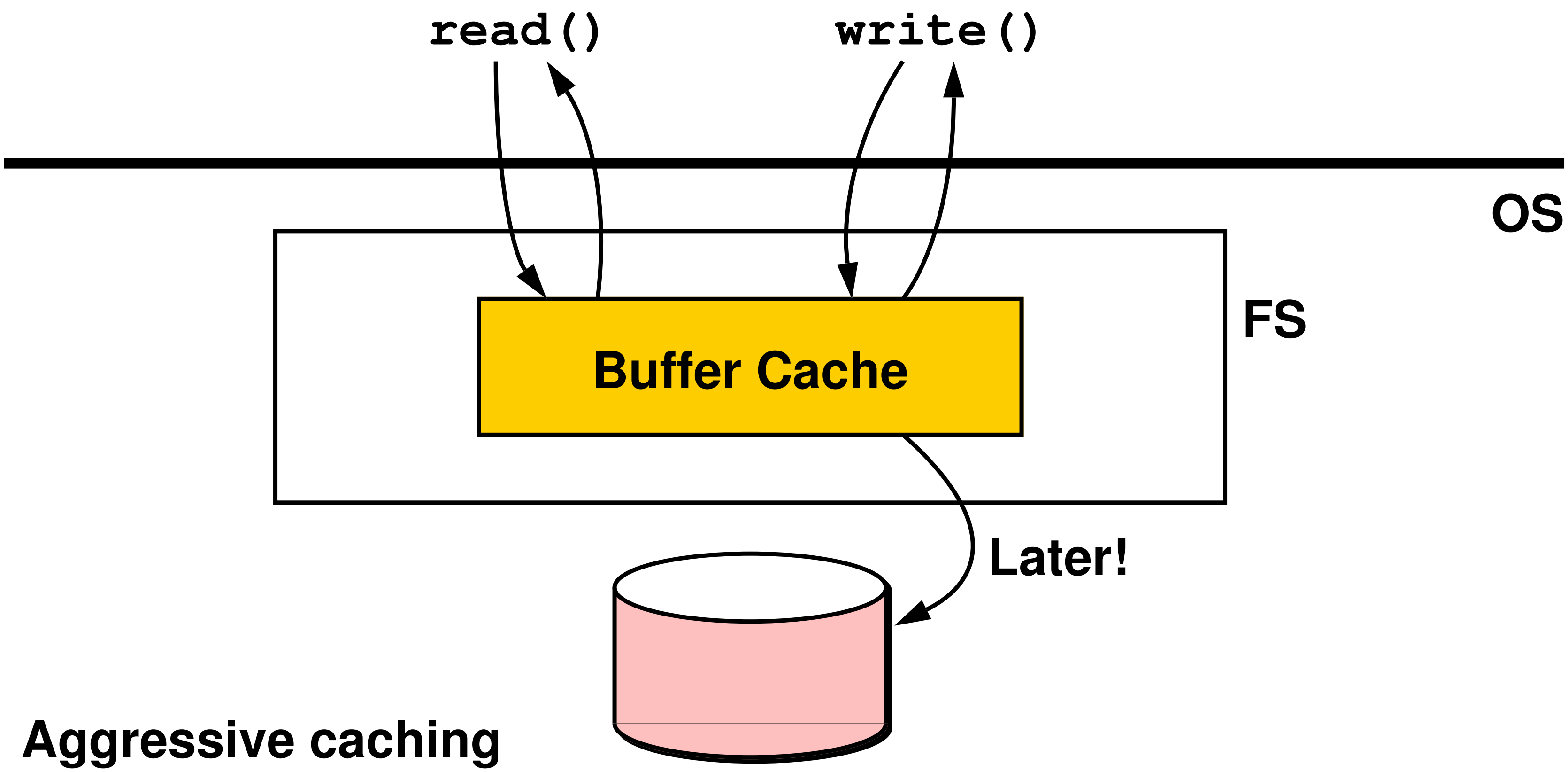
CPU, Memory, Disk Speeds Over Time



A Different Approach

- We have lots of primary memory
  - enough to cache all commonly used files
- Read time from disk doesn’t matter
- Time for writes does matter

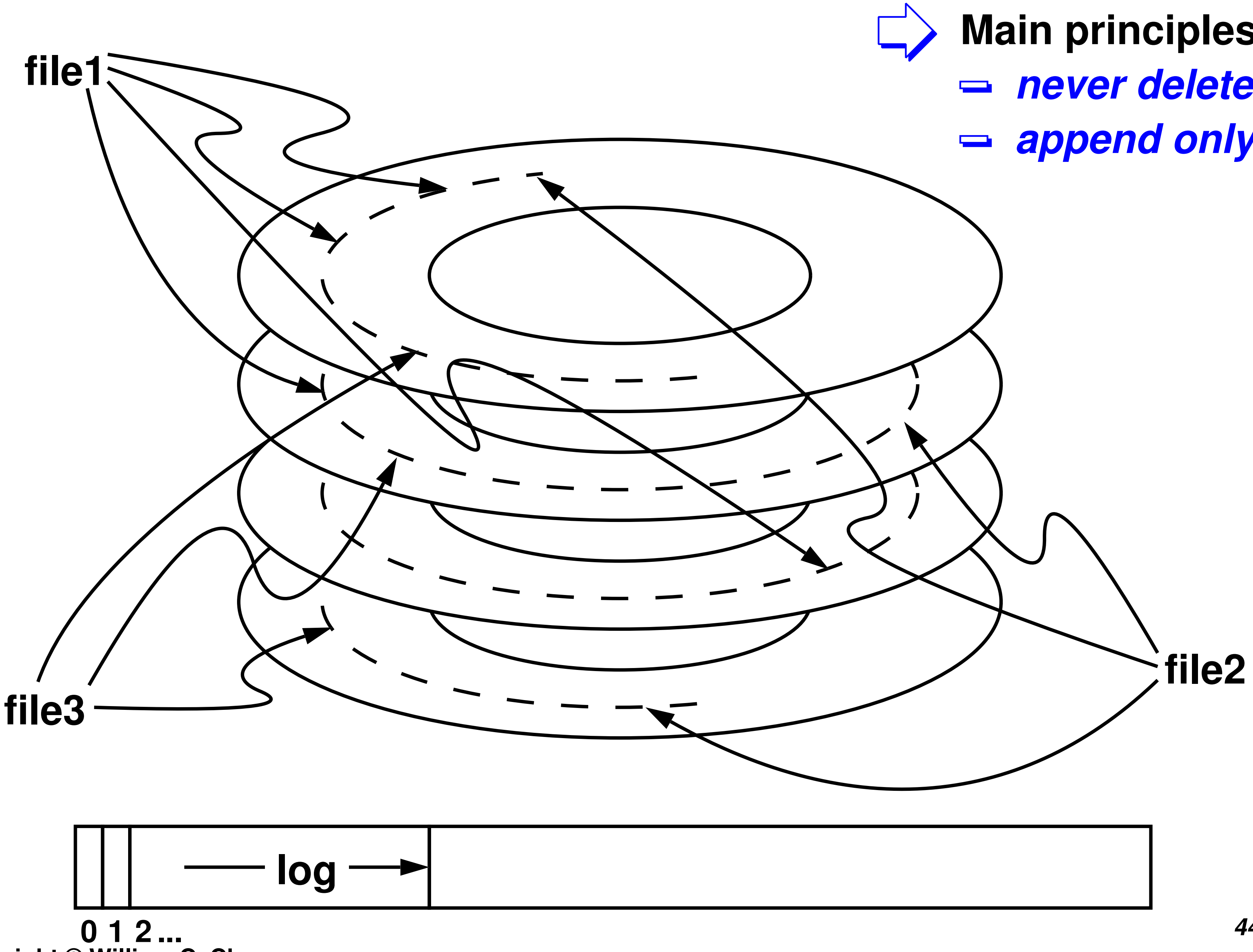
The Buffer Cache



- Aggressive caching
  - most read and write will have a *cache hit*
  - writes to the disk can wait, may be for quite a while
    - longer the wait, higher the *risk*
  - file system optimized for *writing!*
    - how? you organize the disk as a very long *log*

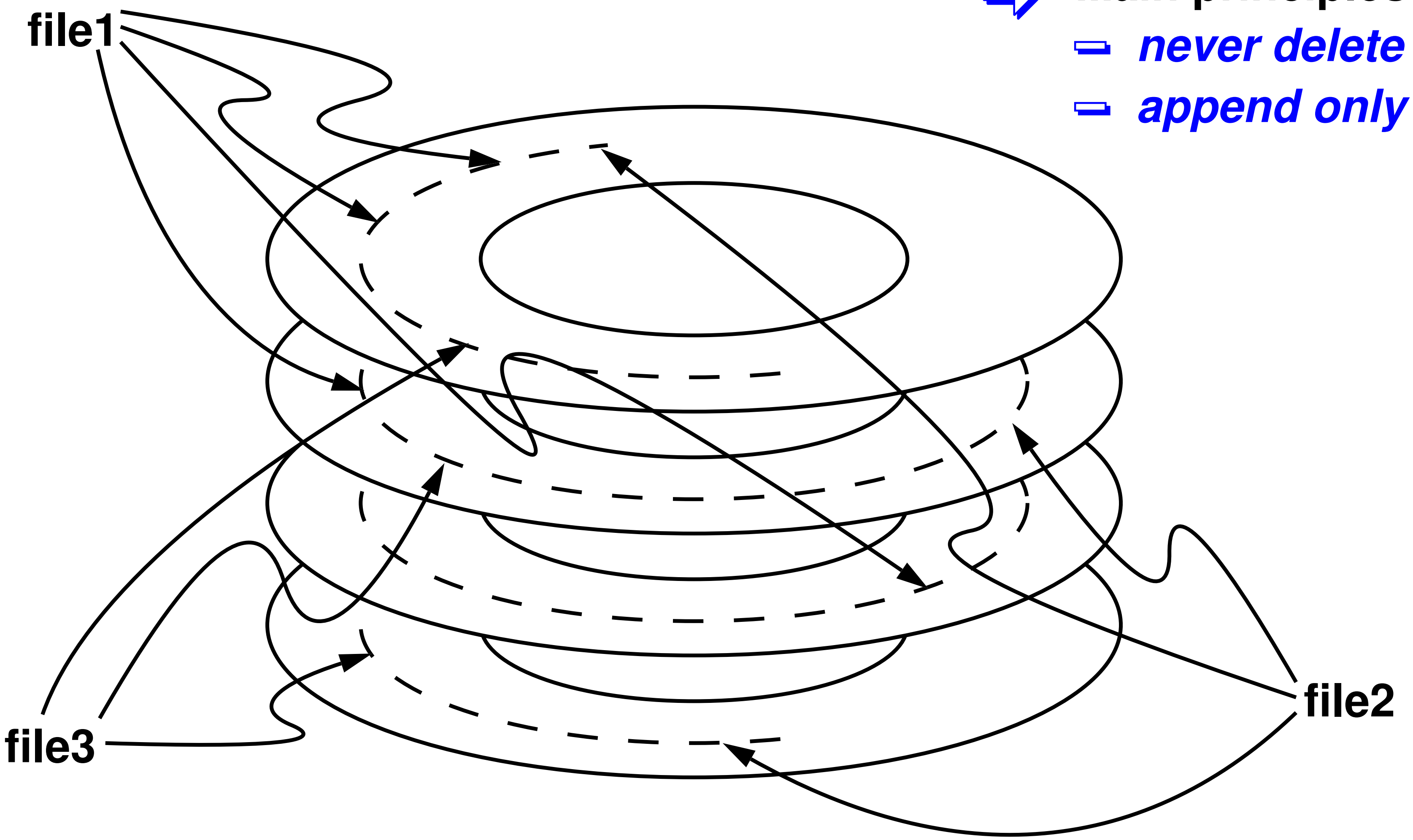
Log-Structured File Systems

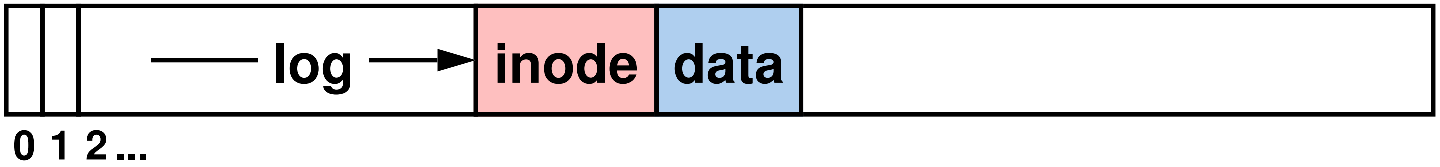
- Main principles
  - *never delete*
  - *append only*



Log-Structured File Systems

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



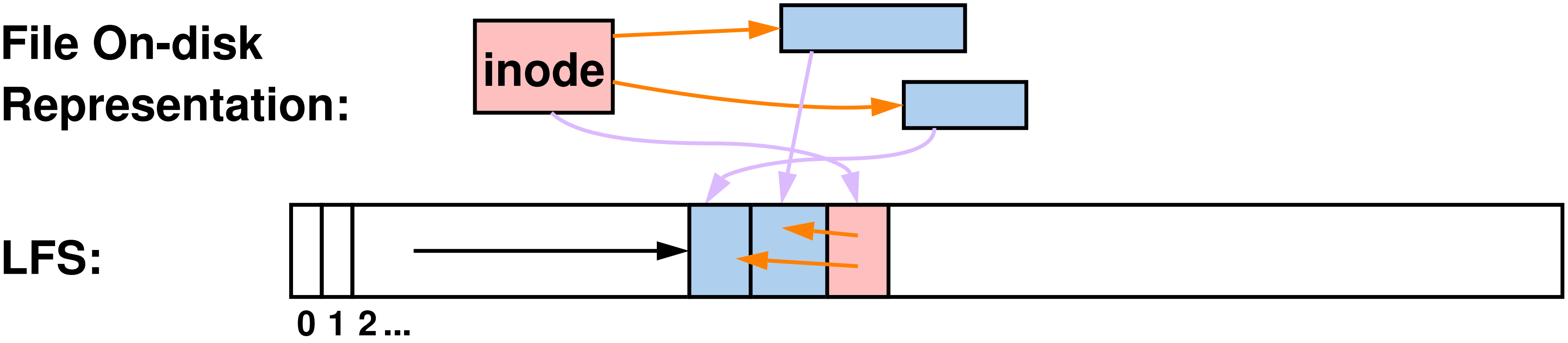


Log-Structured File Systems

- How does "never delete" and "always append" help with performance?
  - minimize seek latency
  - 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

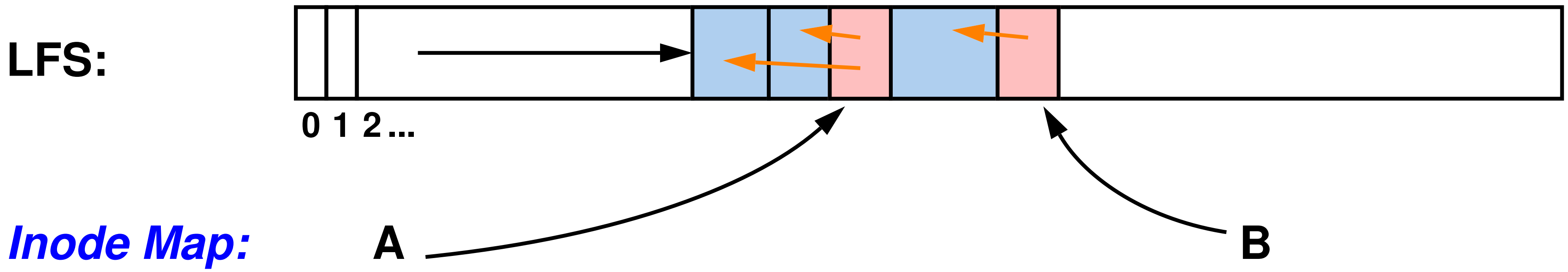
LFS Data Placement Example



LFS Data Placement Example

- What happens if you want to modify the file?
  - how does "append-only" really work?

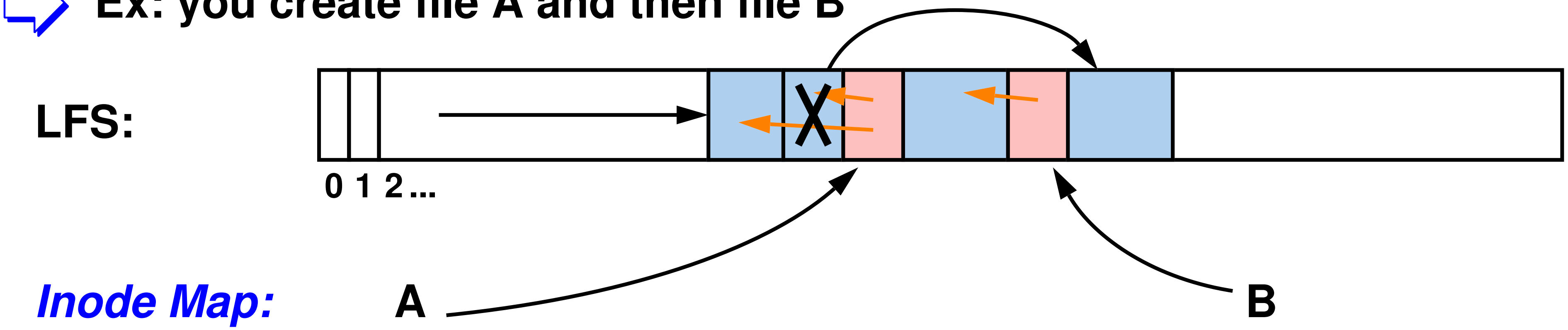
- Ex: you create file A and then file B



  - you modify file A, e.g., append to the last block of file A
  - the new file will be referred as A'

LFS Data Placement Example

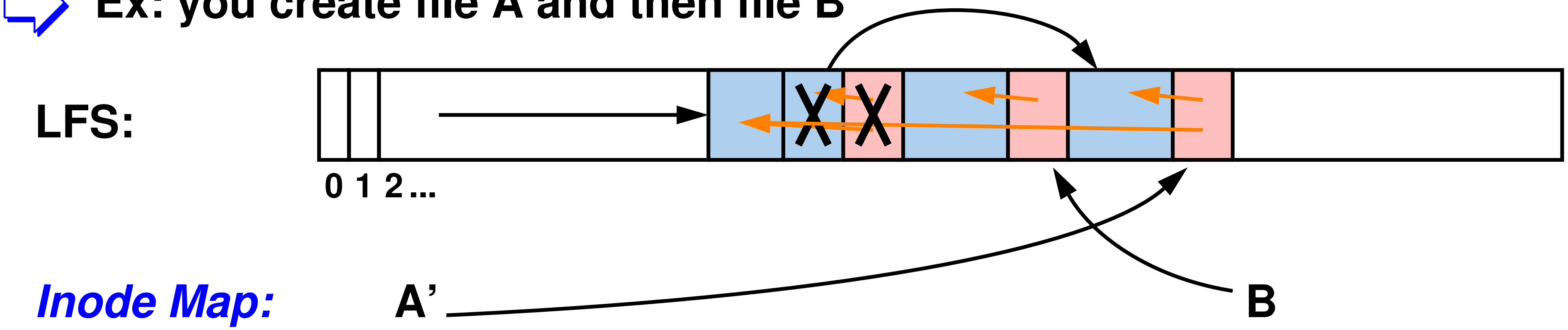
- What happens if you want to modify the file?
  - how does "append-only" really work?
- Ex: you create file A and then file B



  - you modify file A, e.g., append to the last block of file A
  - the new file will be referred as A’

LFS Data Placement Example

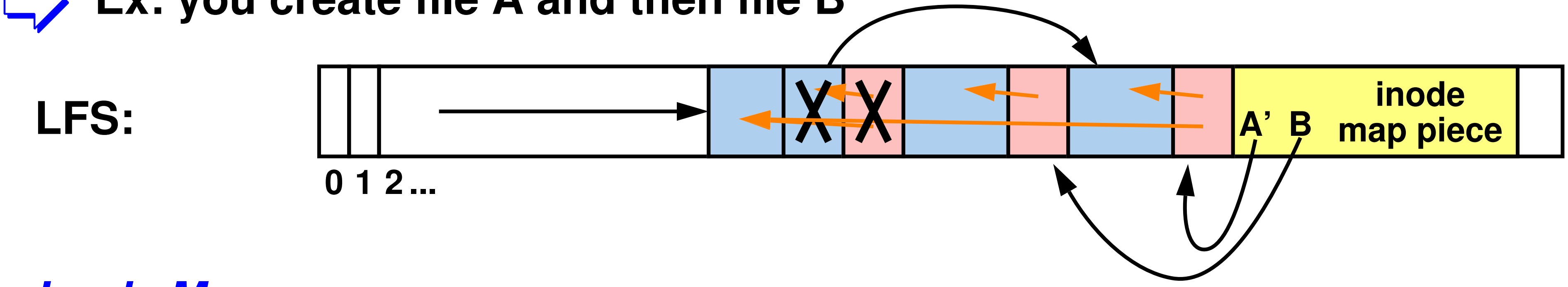
- What happens if you want to modify the file?
  - how does "append-only" really work?
- Ex: you create file A and then file B



  - you modify file A, e.g., append to the last block of file A
  - the new file will be referred as A’
    - the inode has changed as well

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 new file will be referred as A’
  - the inode has changed as well
- a *piece* of the *inode map* is appended to the log
  - 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

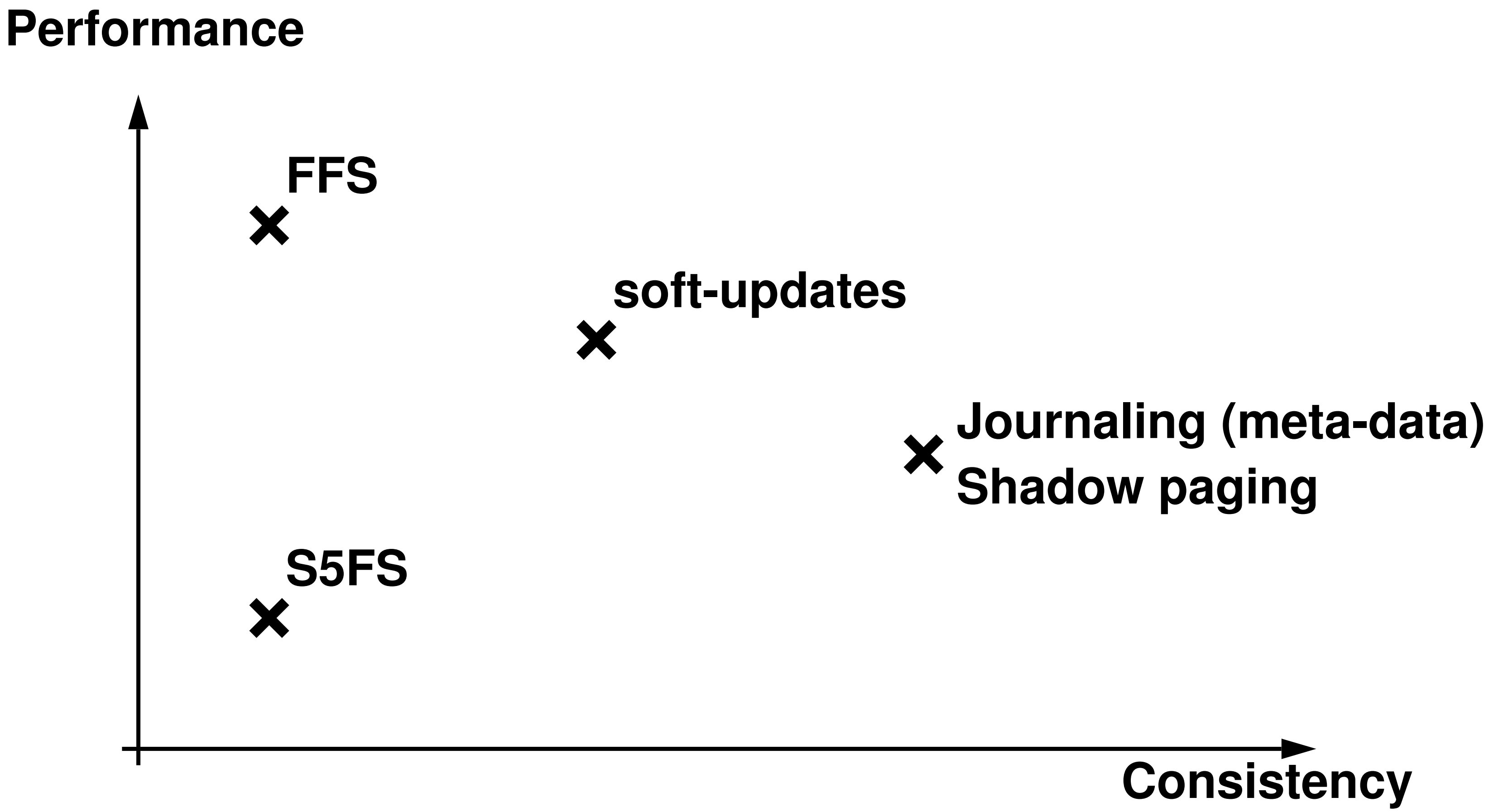
- *Inode Map* cached in primary memory
  - indexed by inode number
  - points to inode on disk
  - written out to disk in pieces as updated
  - *checkpoint file* contains locations of pieces
    - written to disk occasionally
    - two copies: current and previous

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

6.2 Crash Resiliency

- What Goes Wrong
- Dealing with Crashes

Overveiw

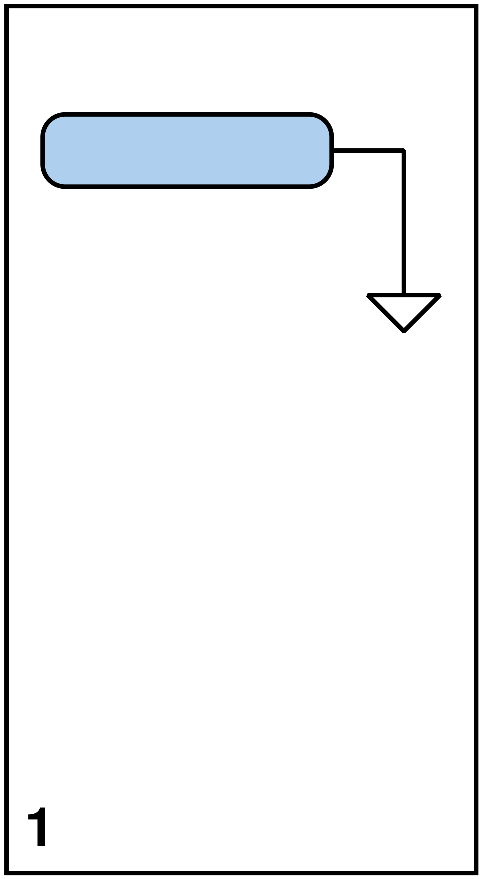


- soft-update provides *recoverable consistency*
- journaling and shadow paging provide *transactional consistency*

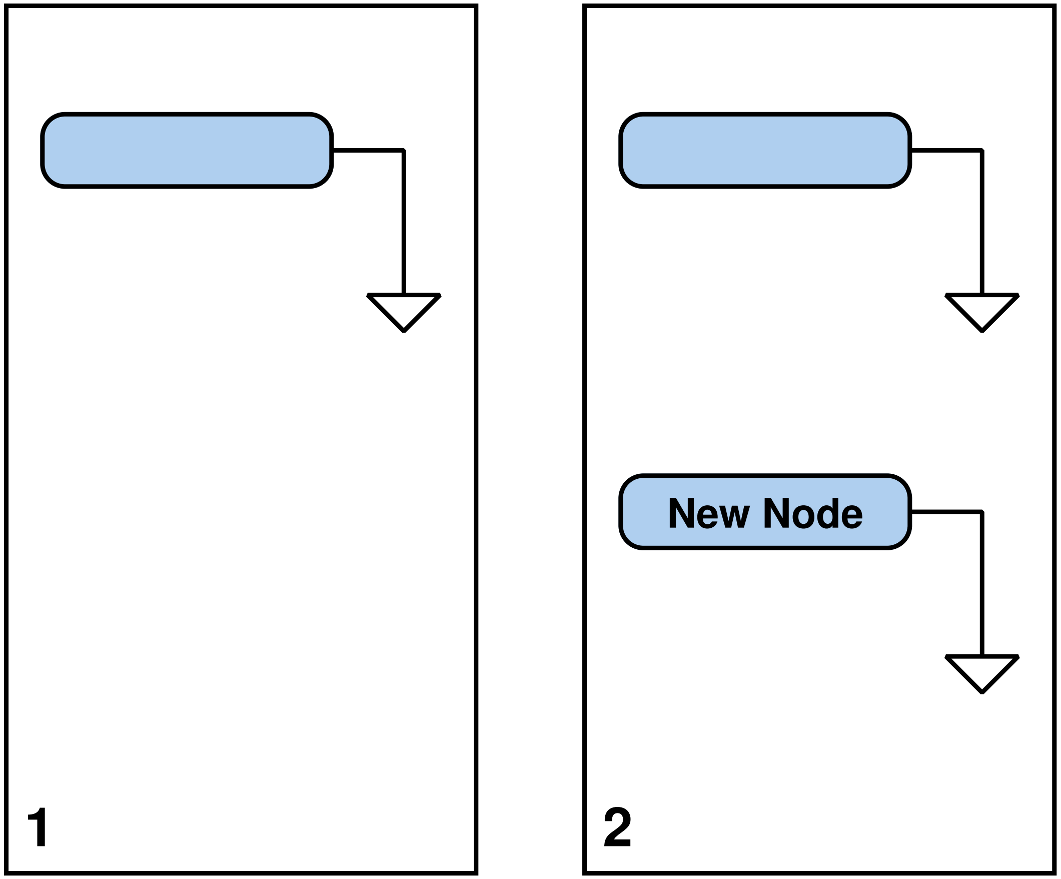
In the Event of a Crash ...

- 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 ...

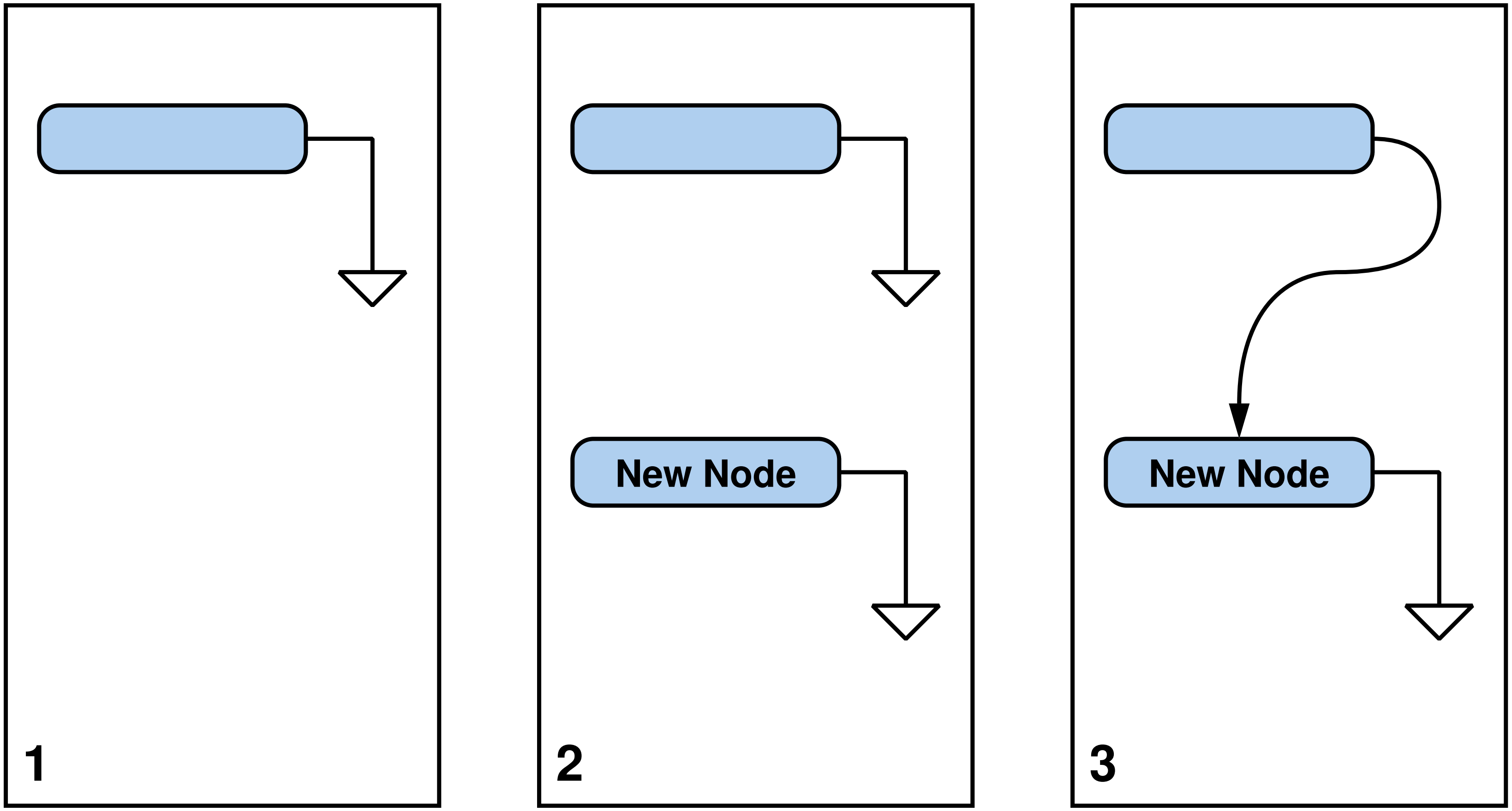
File-System Consistency (1)



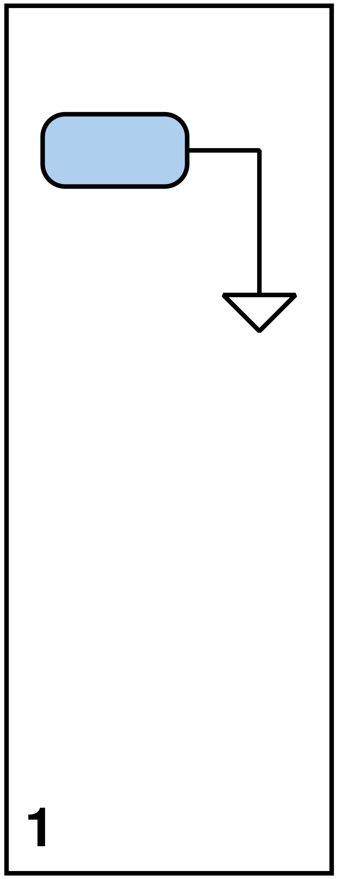
File-System Consistency (1)



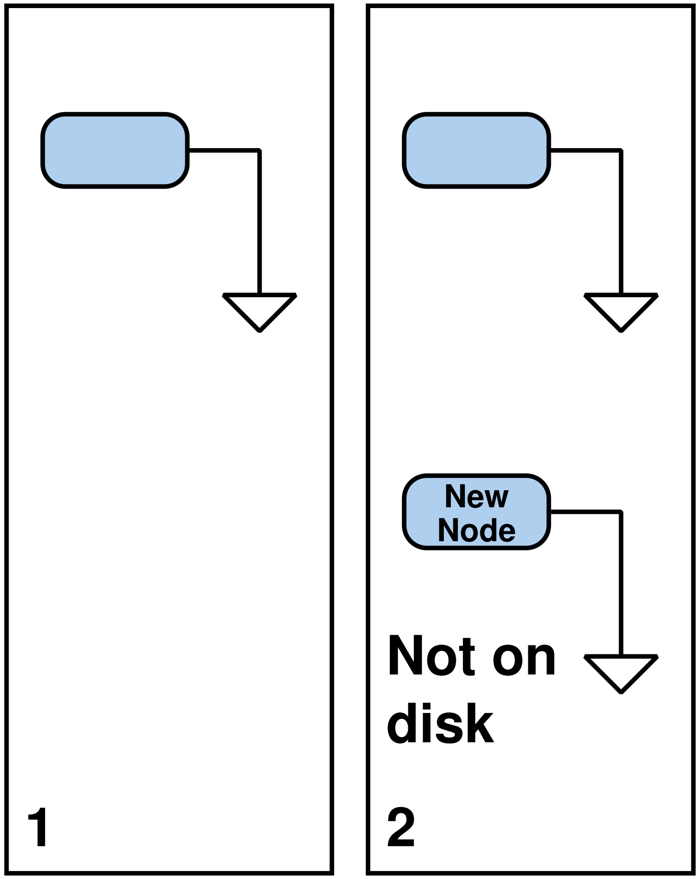
File-System Consistency (1)



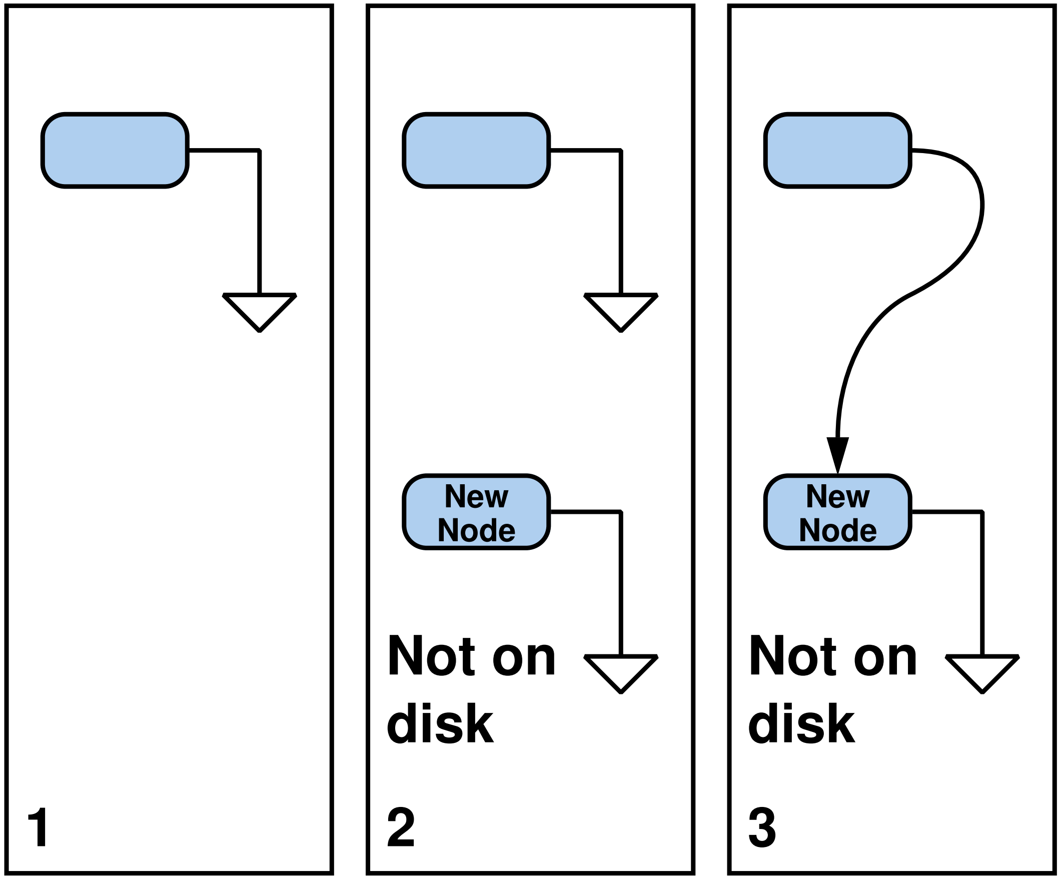
File-System Consistency (2)



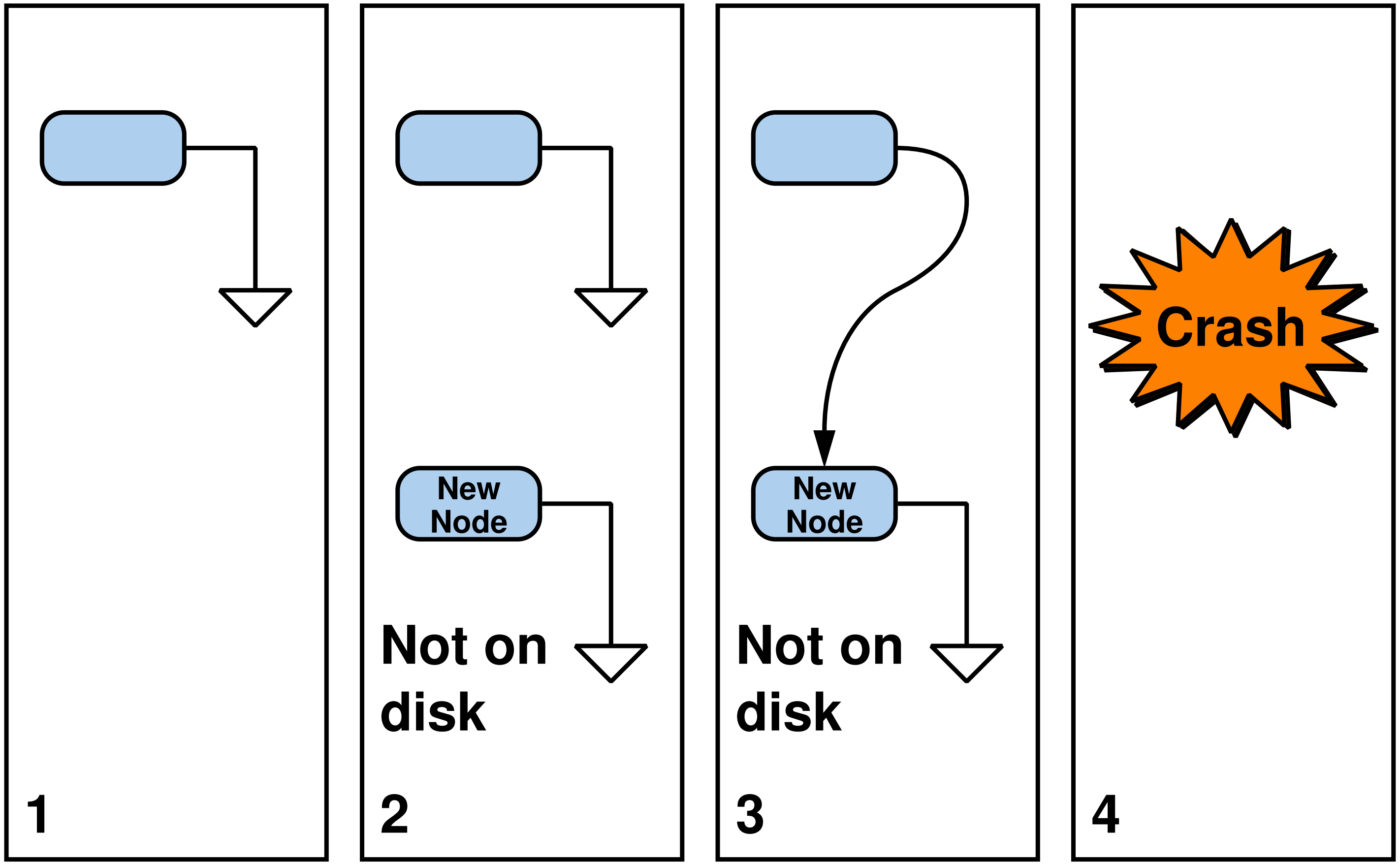
File-System Consistency (2)



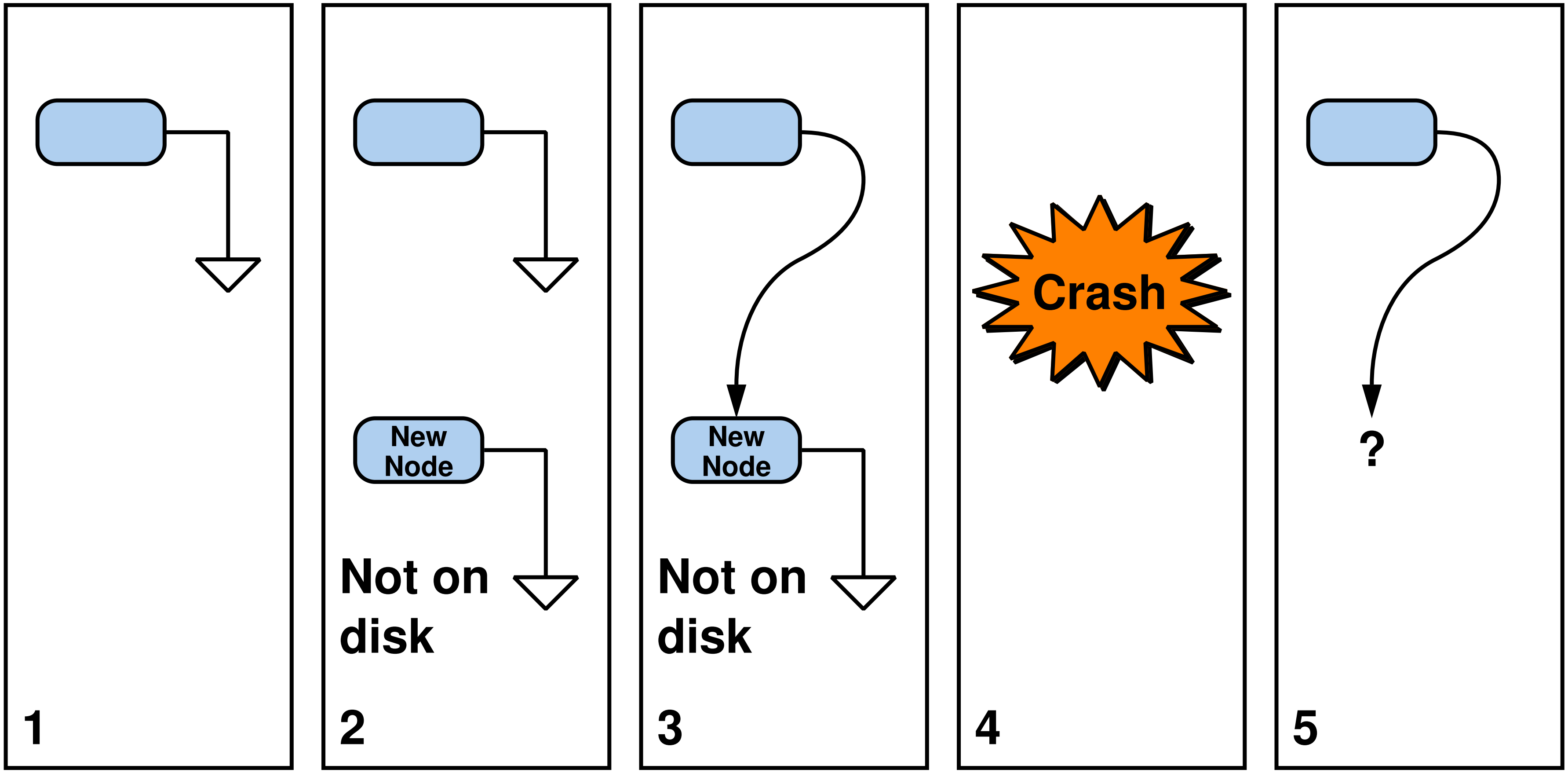
File-System Consistency (2)



File-System Consistency (2)

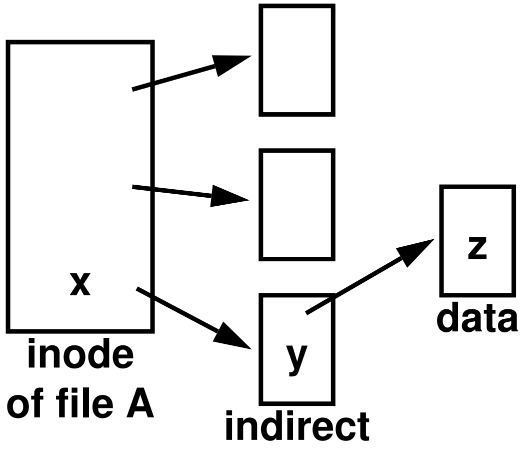


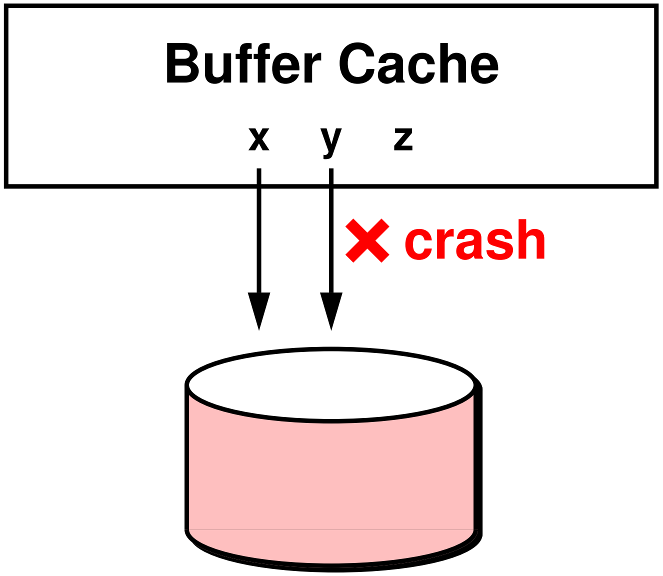
File-System Consistency (2)



A More Realistic Example

- Let’s say that you are appending to file A

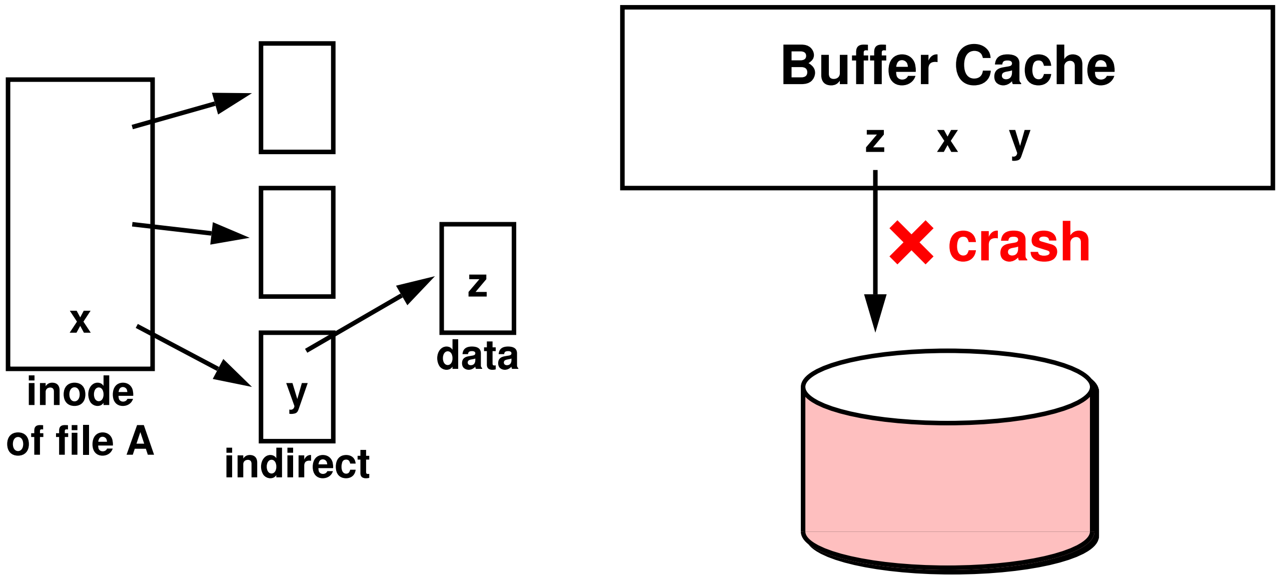




  - the buffer cache does not know about the relationship between blocks x, y, and z
  - techniques like locking (i.e., lock the disk so that it cannot crash when it’s locked) won’t work

A More Realistic Example

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



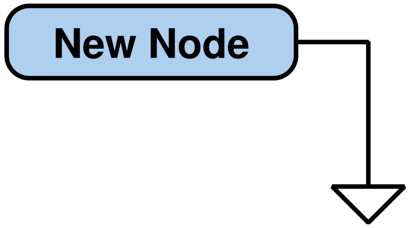
  - what about this order and crash timing?
    - what about other combinations?

How to Cope ...

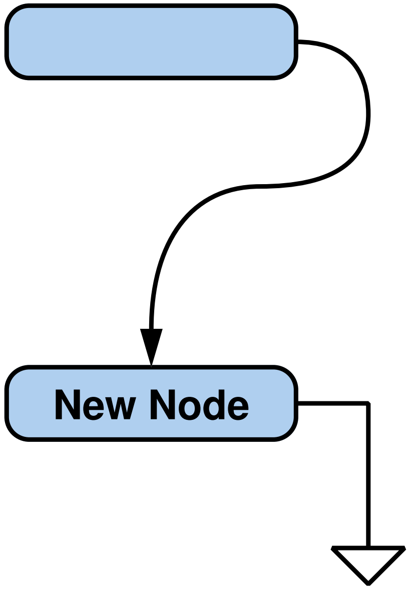
- Don't crash
- Perform multi-step disk updates in an order such that disk is always consistent, i.e., the *consistency-preserving approach*
- Perform multi-step disk updates as *transactions*, i.e., implemented so that either all steps take effect or none do

Maintaining Consistency

1) Write this synchronously to disk



Maintaining Consistency

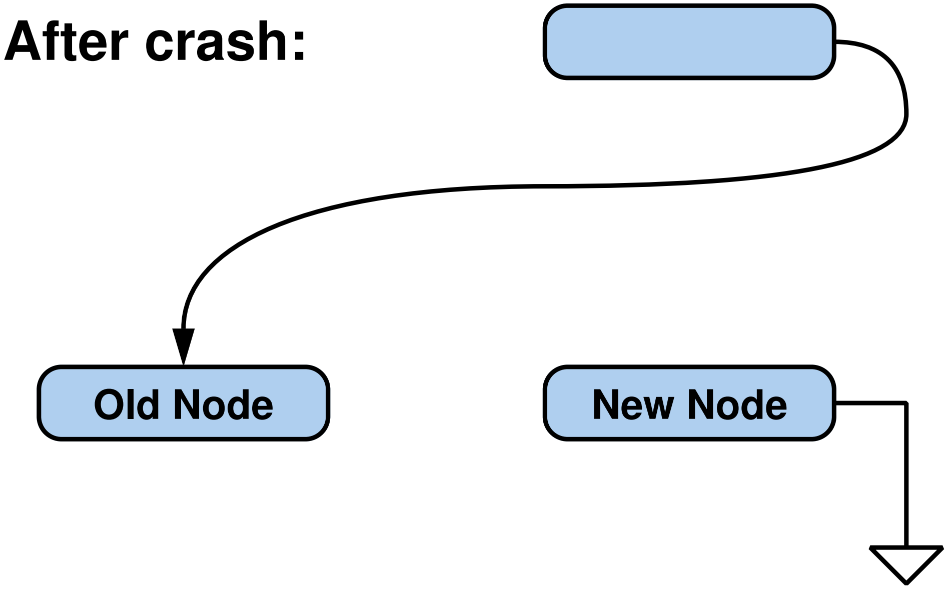


2) Then write this asynchronously via the cache

1) Write this synchronously to disk

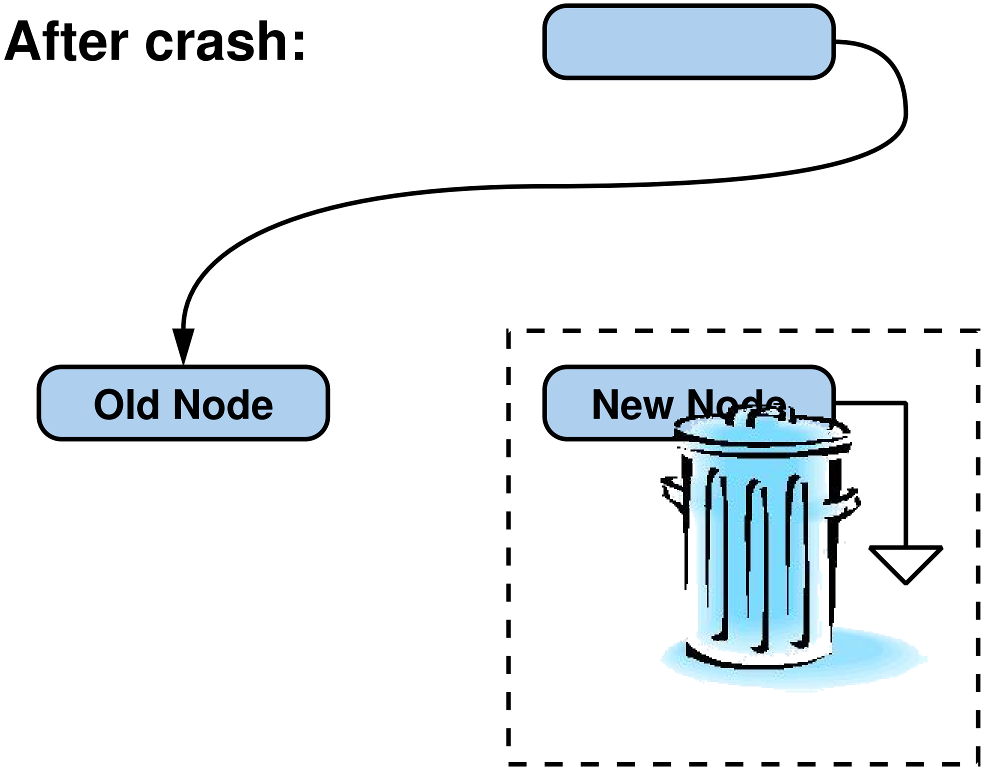
Innocuous Inconsistency

After crash:



Innocuous Inconsistency

After crash:

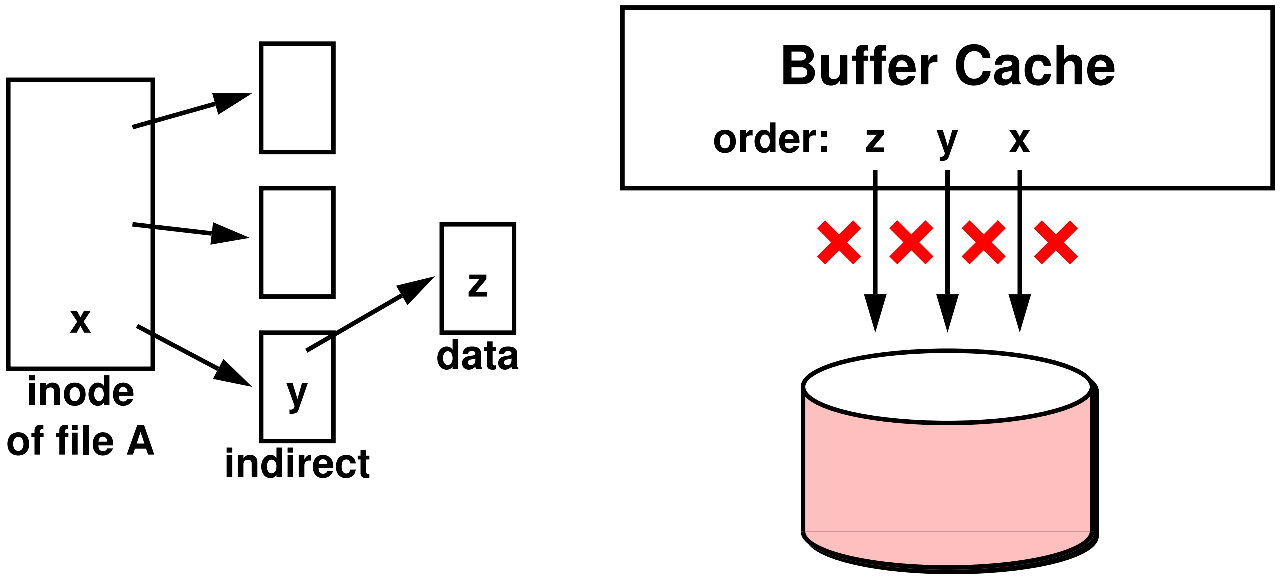


Soft Updates

- Main idea
  - order disk operations to preserve meta-data consistency
    - innocuous inconsistency is considered ok

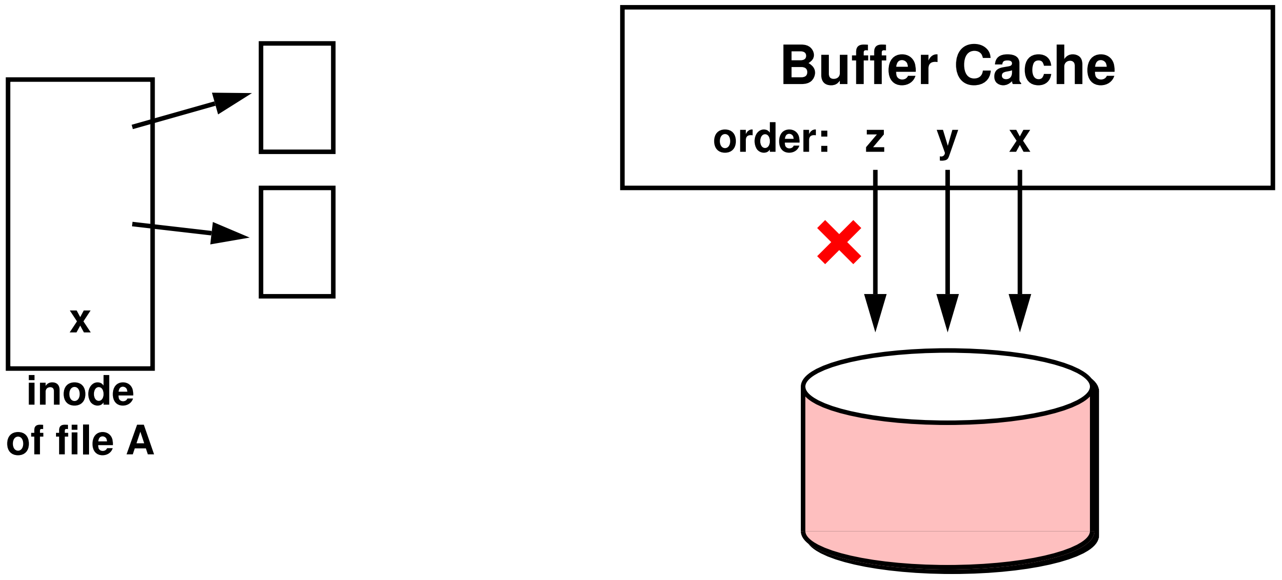
Back To The Example

- Let’s say that you are appending to file A



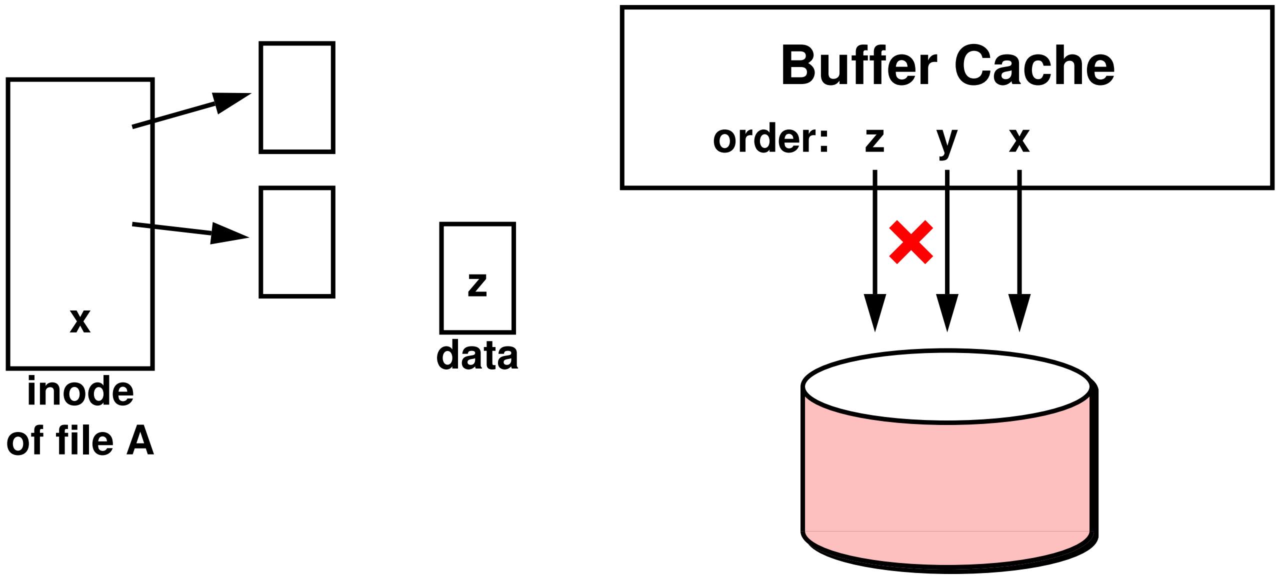
Back To The Example

- Let’s say that you are appending to file A



Back To The Example

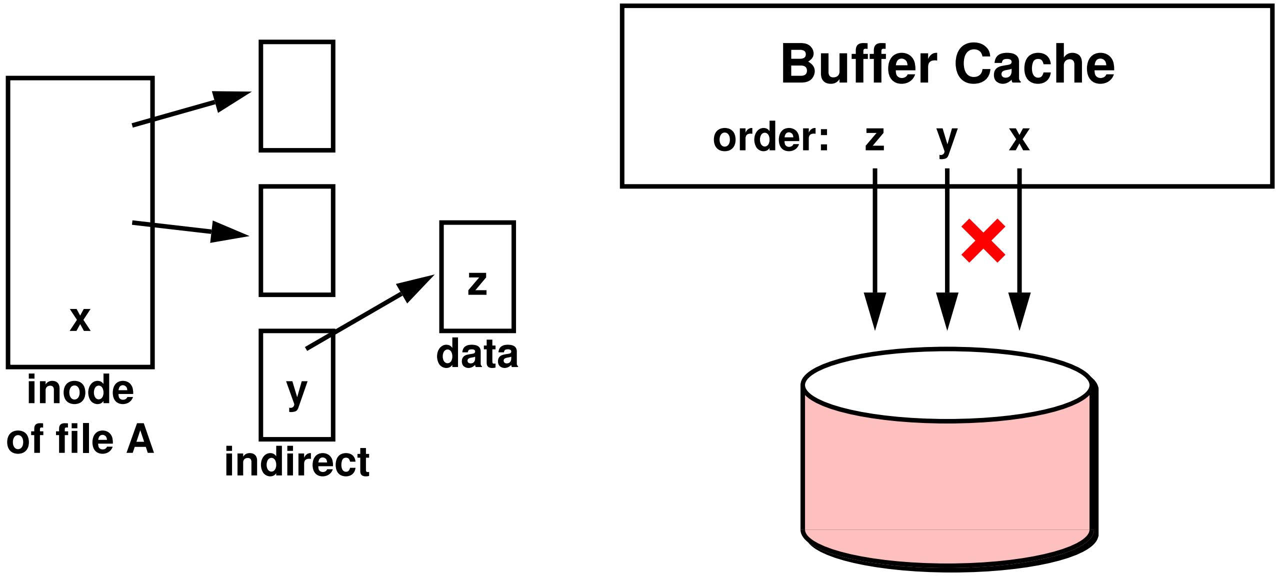
- Let’s say that you are appending to file A



  - is this bad?
    - how bad is it?

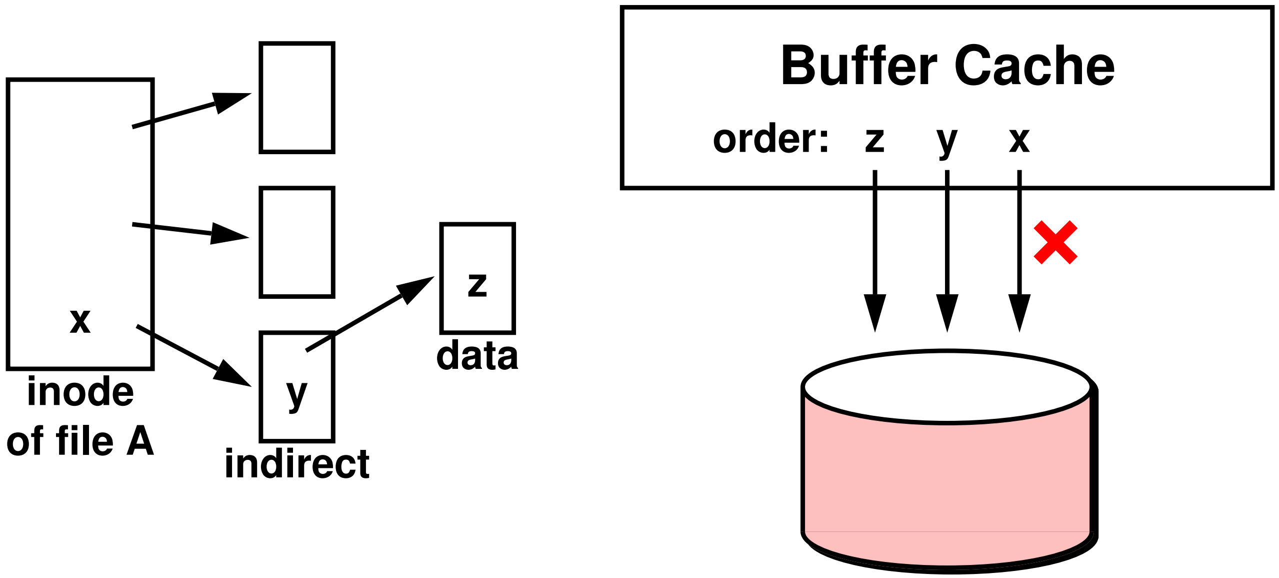
Back To The Example

- Let’s say that you are appending to file A



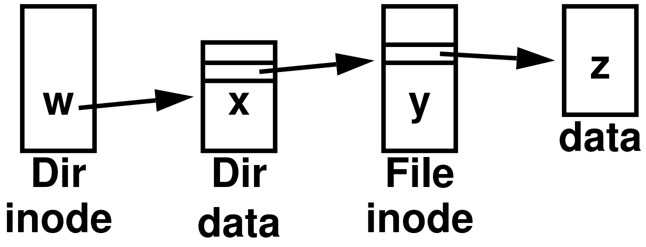
Back To The Example

- Let’s say that you are appending to file A

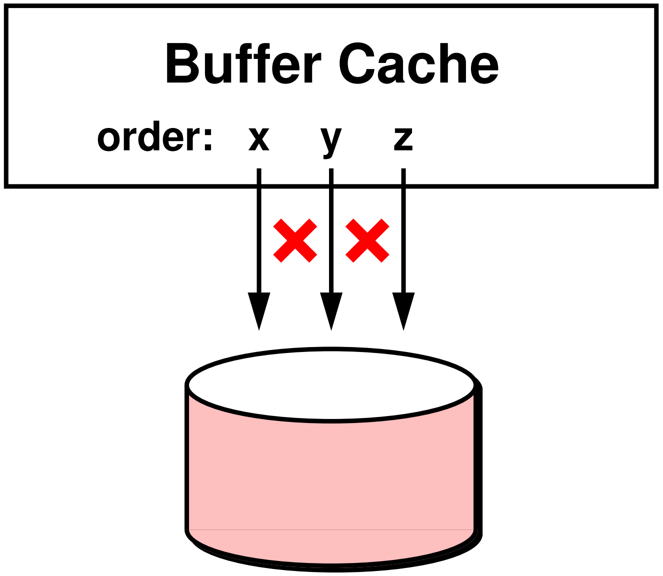


Soft Update Example 2

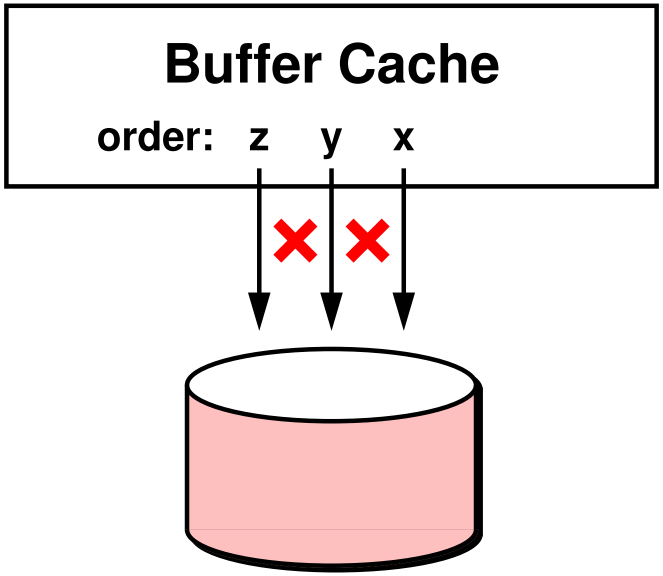
- Create a new file with one data block



Choice 1

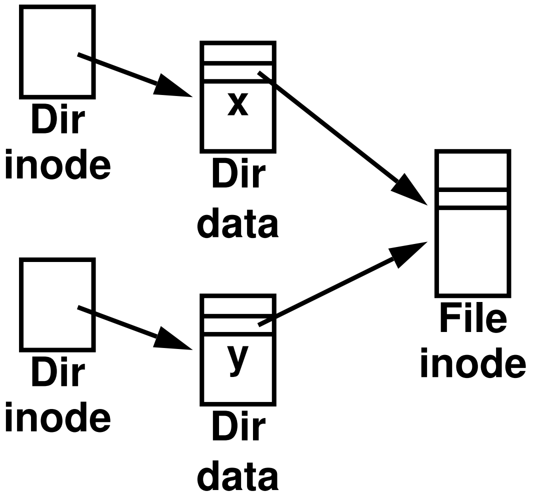


Choice 2



Soft Update Example 3

- Move a file



Dir inode

x

Dir data

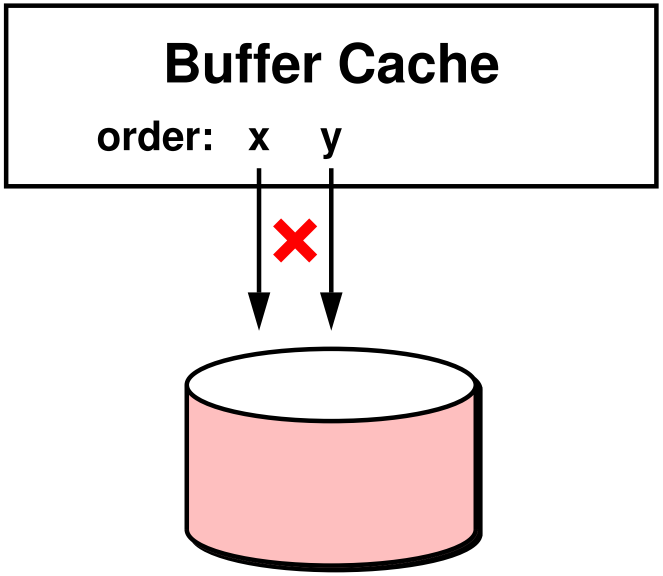
Dir inode

y

Dir data

File inode

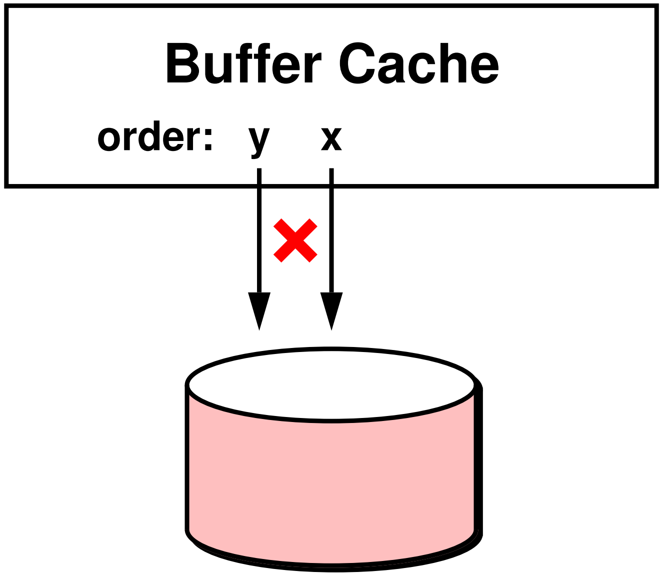
Choice 1



Buffer Cache

order: x y

Choice 2



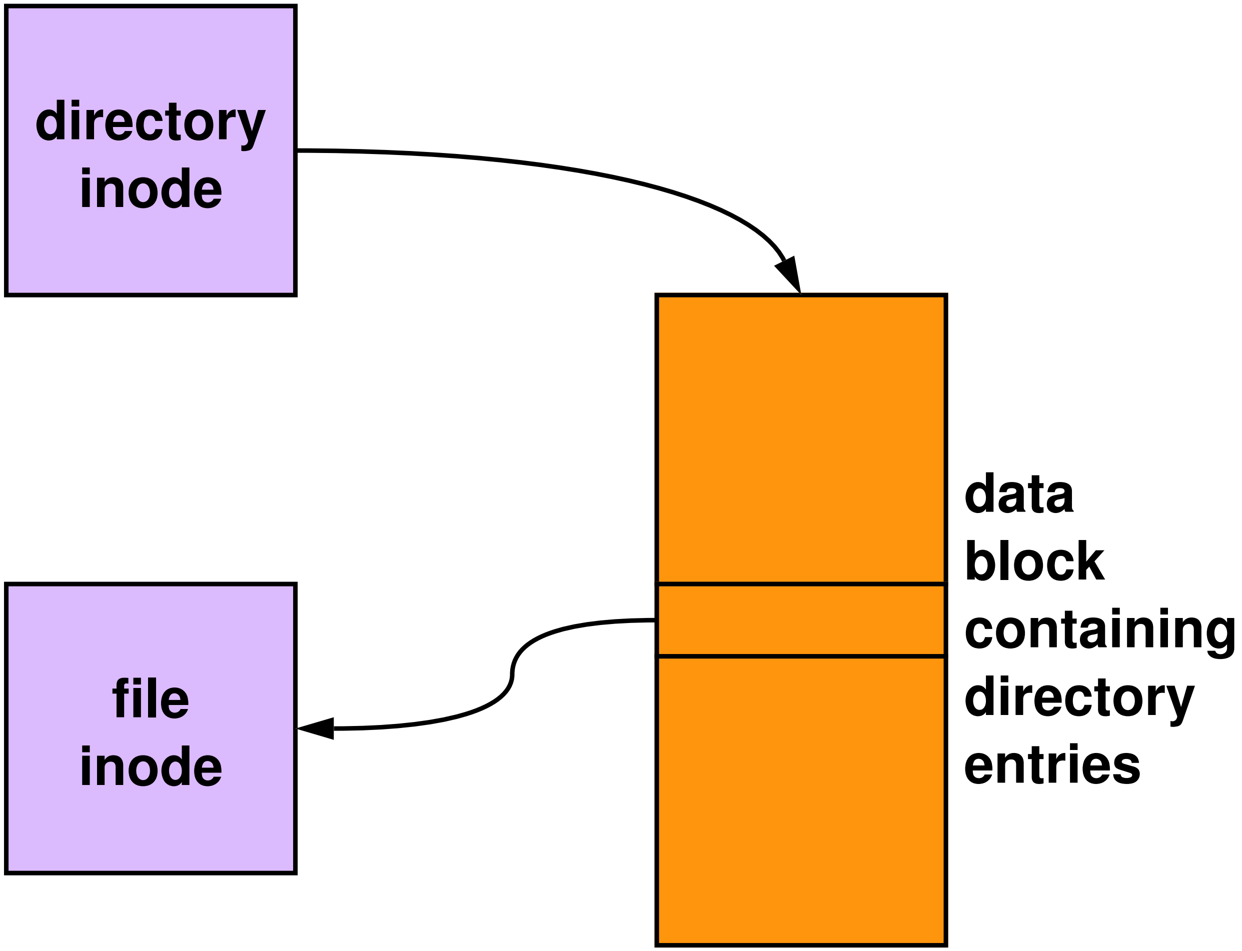
Buffer Cache

order: y x

Soft Update

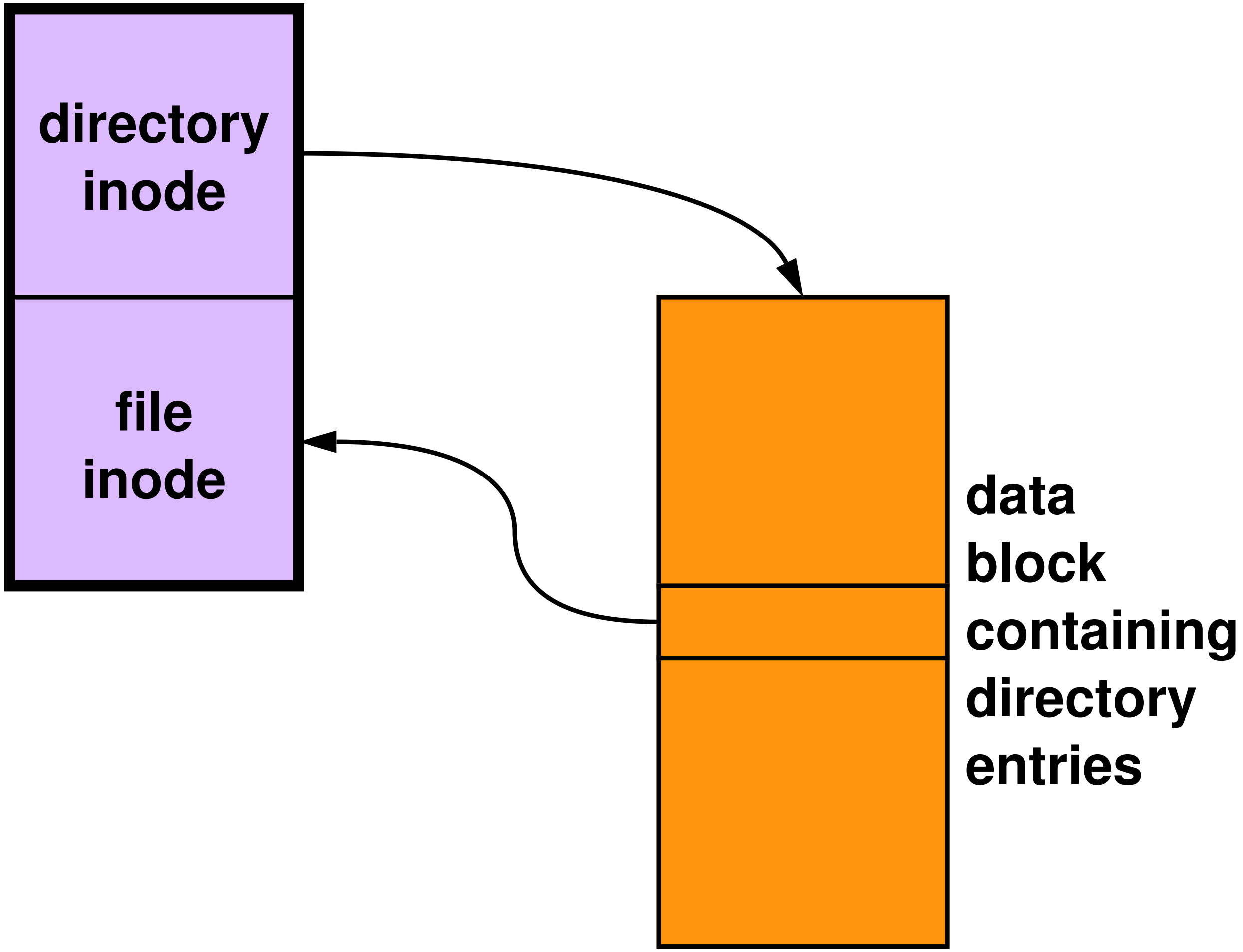
- **An implementation of the consistency-preserving approach**
  - **the idea is simple:**
    - **update cache in an order that maintains consistency**
    - **write cache contents to disk in same order in which cache was updated**
  - **isn't, because reality is more complicated**
    - **(assuming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

Which Order?



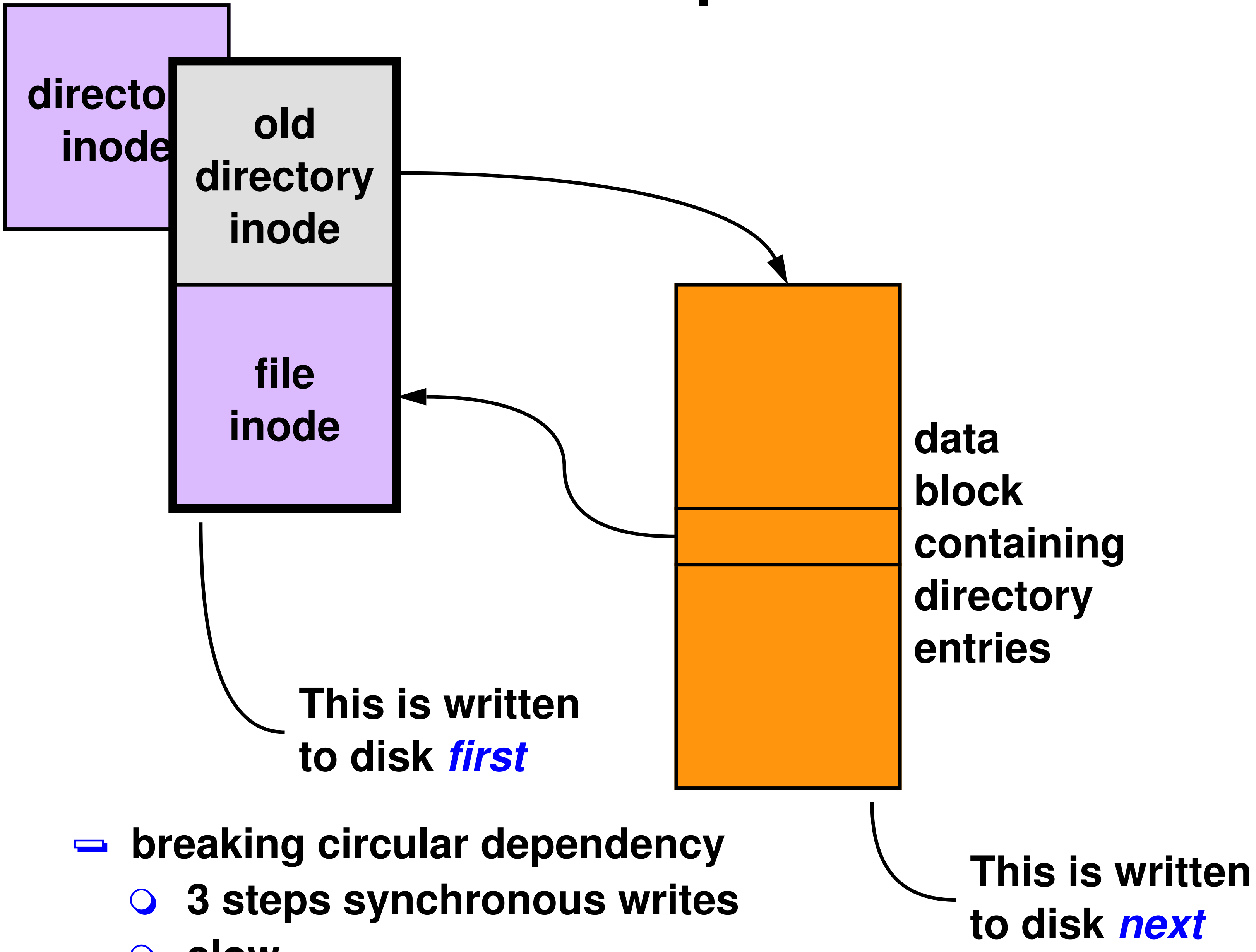
- this is easy

However ...



- circular dependency
  - in reality, in order to save the number of disk writes, multiple objects can be packed into a disk block

Soft Updates



- breaking circular dependency
  - 3 steps 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**