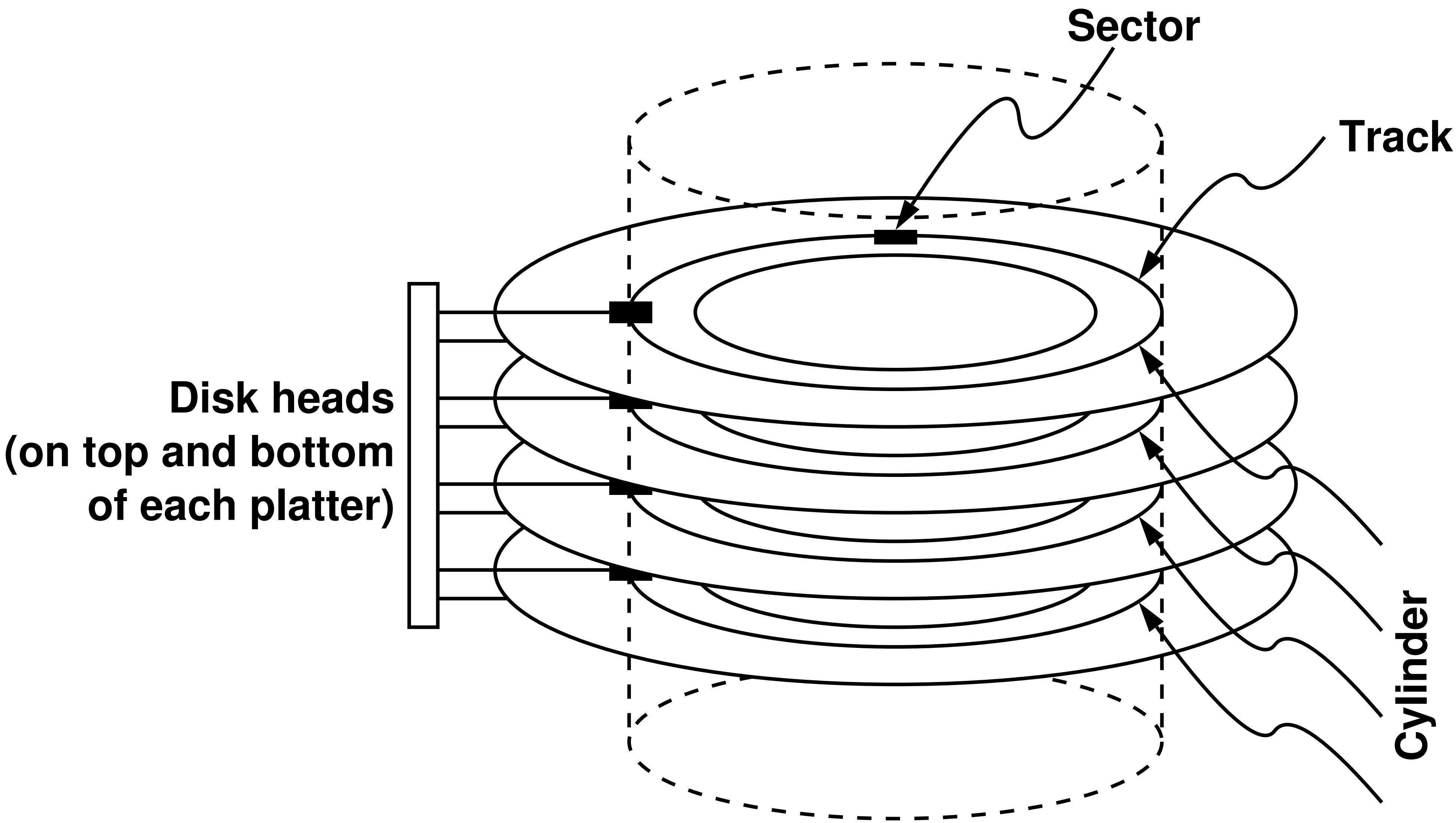
6.1 The Basics of File Systems

- UNIX's S5FS
- Disk Architecture
- *Problems with S5FS*
- Improving Performance

Disk Architecture



- Smallest addressable unit is a *sector*
  - disk address = (*head/surface#*, *cylinder/track#*, *sector#*)

Rhinopias Disk Drive

Rotation speed	10,000 RPM
Number of surfaces	8
Sector size	512 bytes
Sectors/track	500-1000; 750 average
Tracks/surface	100,000
Storage capacity	307.2 billion bytes
Average seek time	4 milliseconds
One-track seek time	.2 milliseconds
Maximum seek time	10 milliseconds

- ***Disk access time:*** time to copy a sector from disk to controller
  - *access time = seek time + rotational latency + data transfer time*
    - some people would use the term "response time" to mean "access time", but we should avoid the use of the term "response time"

S5FS on Rhinopias (A Marketing Disaster ...)

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

6.1 The Basics of File Systems

- UNIX's S5FS
- Disk Architecture
- Problems with S5FS
- *Improving Performance*

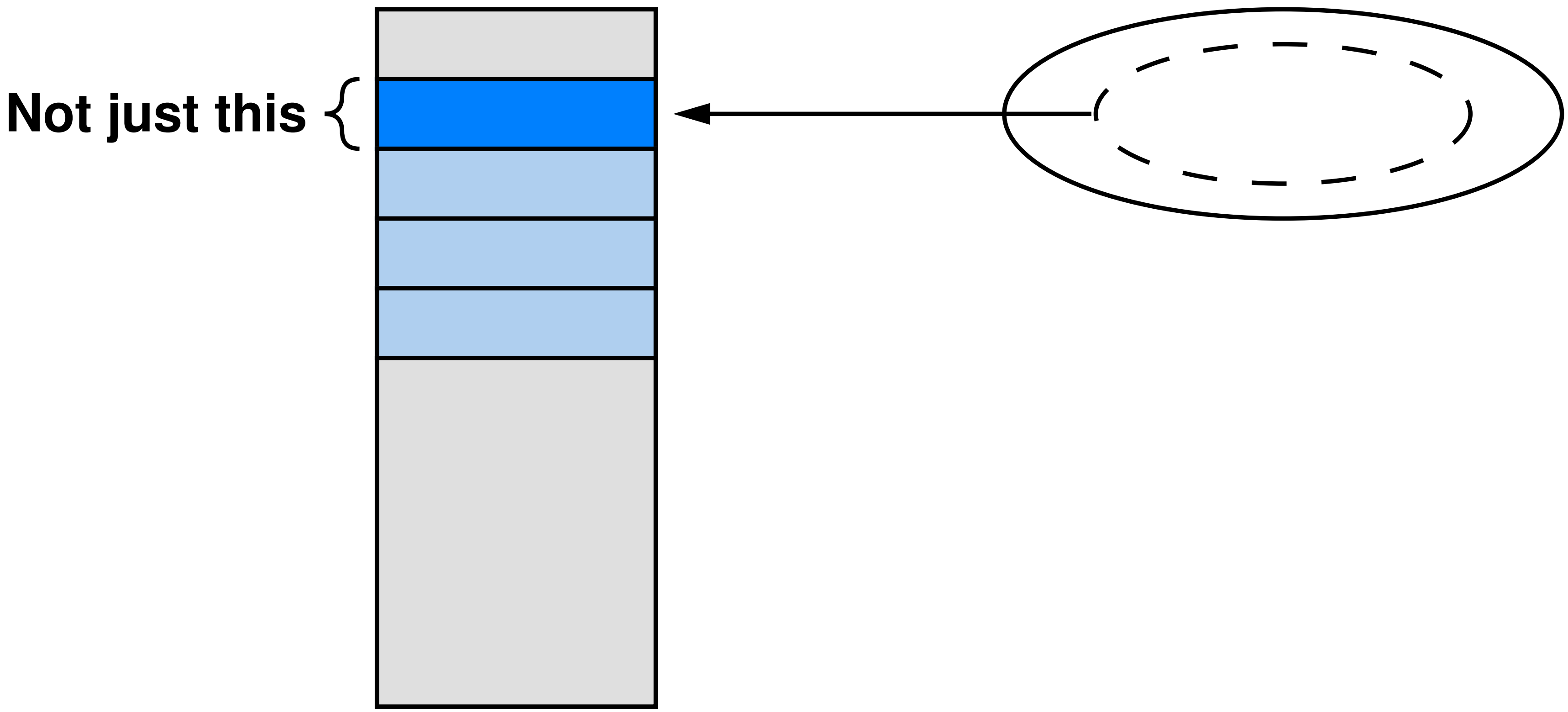
What to Do About It?

- **Hardware**
  - employ pre-fetch buffer in disk controller
    - filled by hardware with what's underneath disk 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

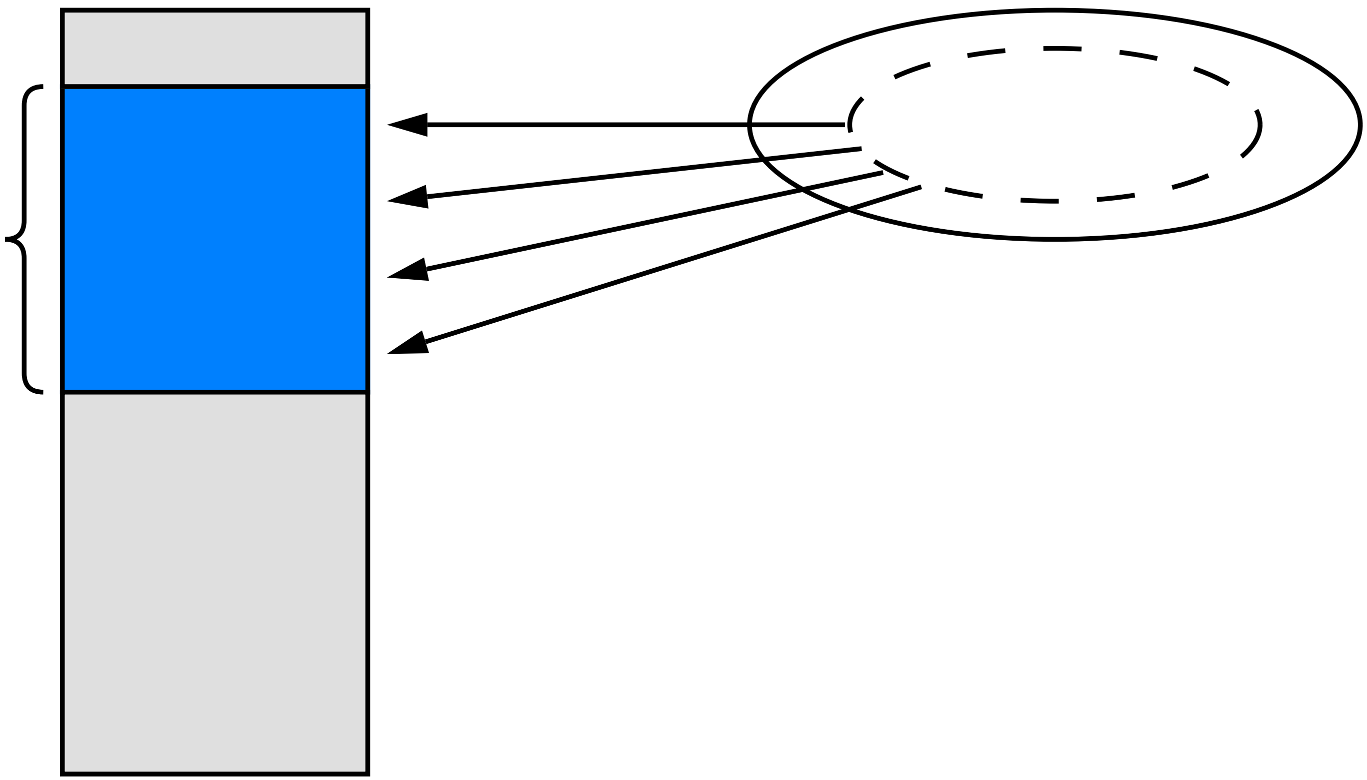
- Better on-disk organization
- Longer component names in directories
- Retains disk map of S5FS

Larger Block Size



Larger Block Size

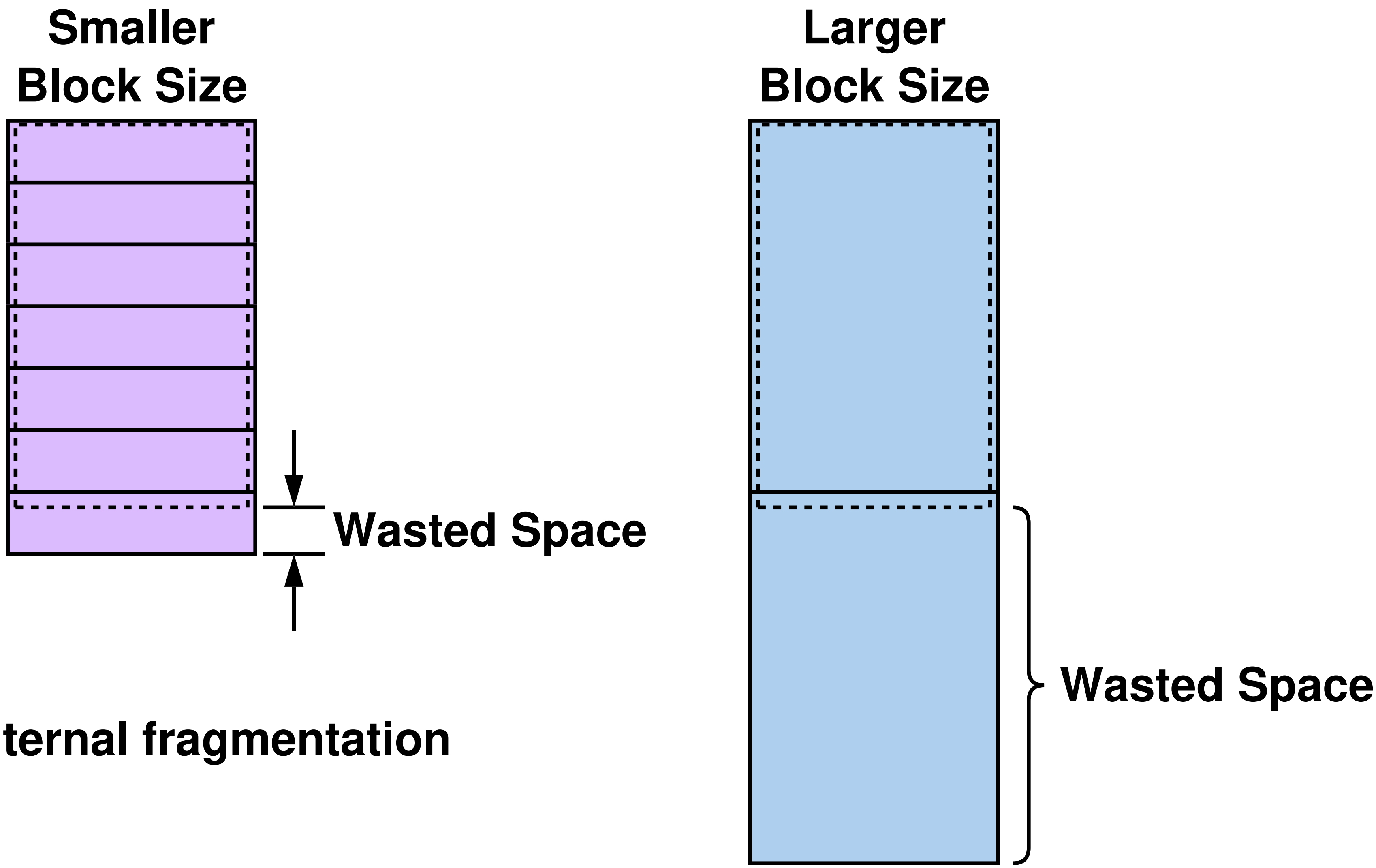
But all this



- *O(n) = O(n/4)*, why bother?
  - reading ***consecutive sectors*** is faster
    - much better than the hardware solution of pre-fetching
  - to improve file system performance, you need to reduce the number of times you go to the disk

The Down Side ...

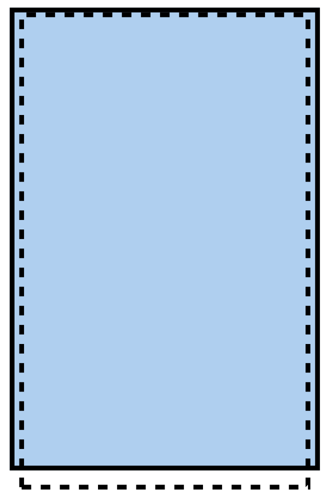
The Down Side ...



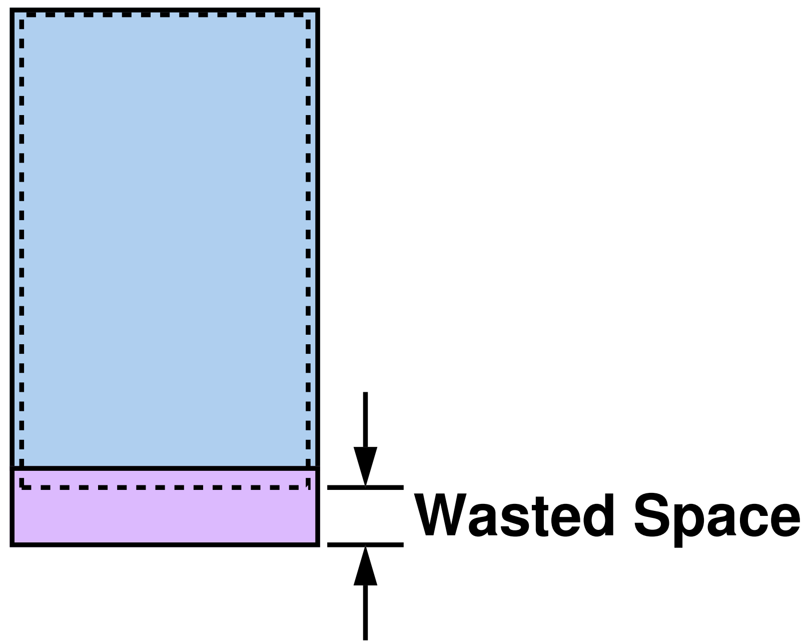
- internal fragmentation

- even worse internal fragmentation

Two Block Sizes ...



Two Block Sizes ...

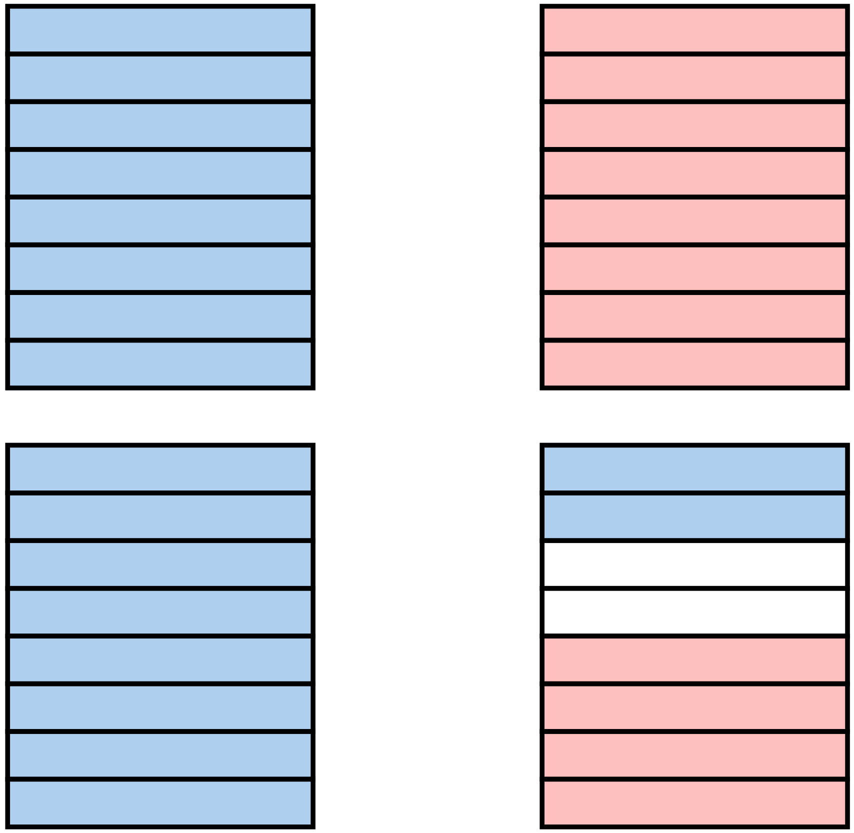


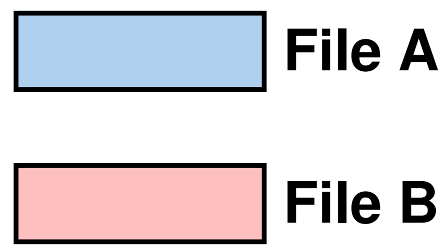
- e.g., 16KB blocks and 1KB fragments
- best of both worlds
  - but there is no "free lunch", so what's the cost?

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

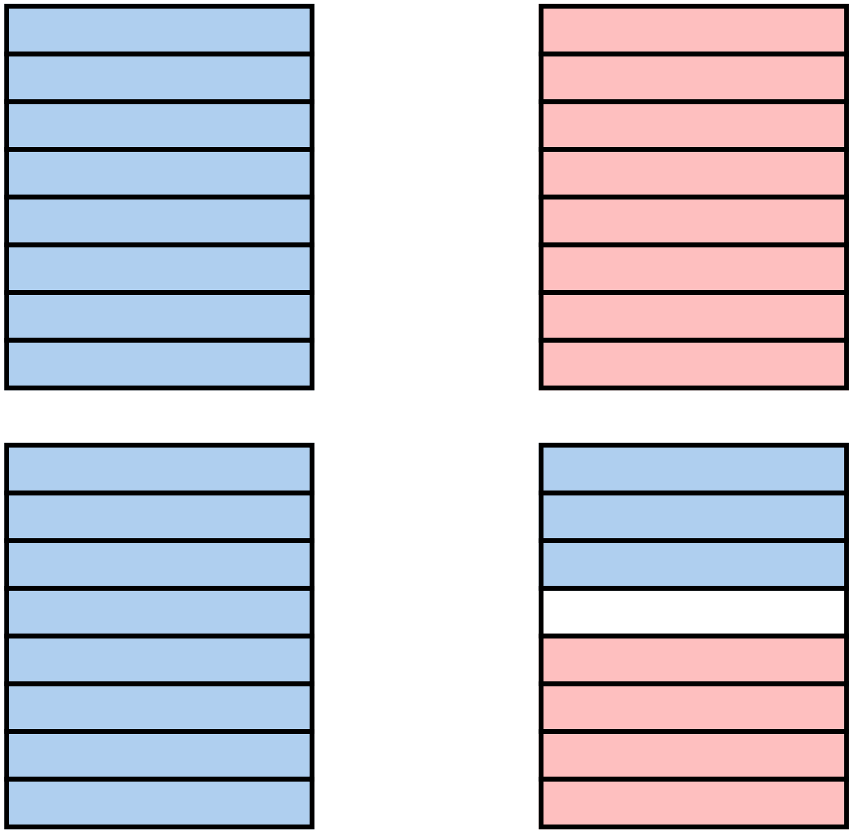
Use of Fragments (1)

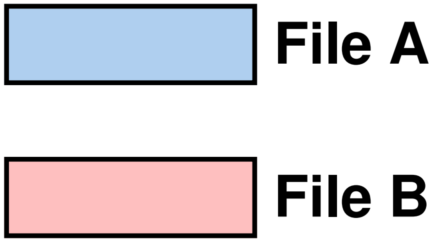




- can save even more space

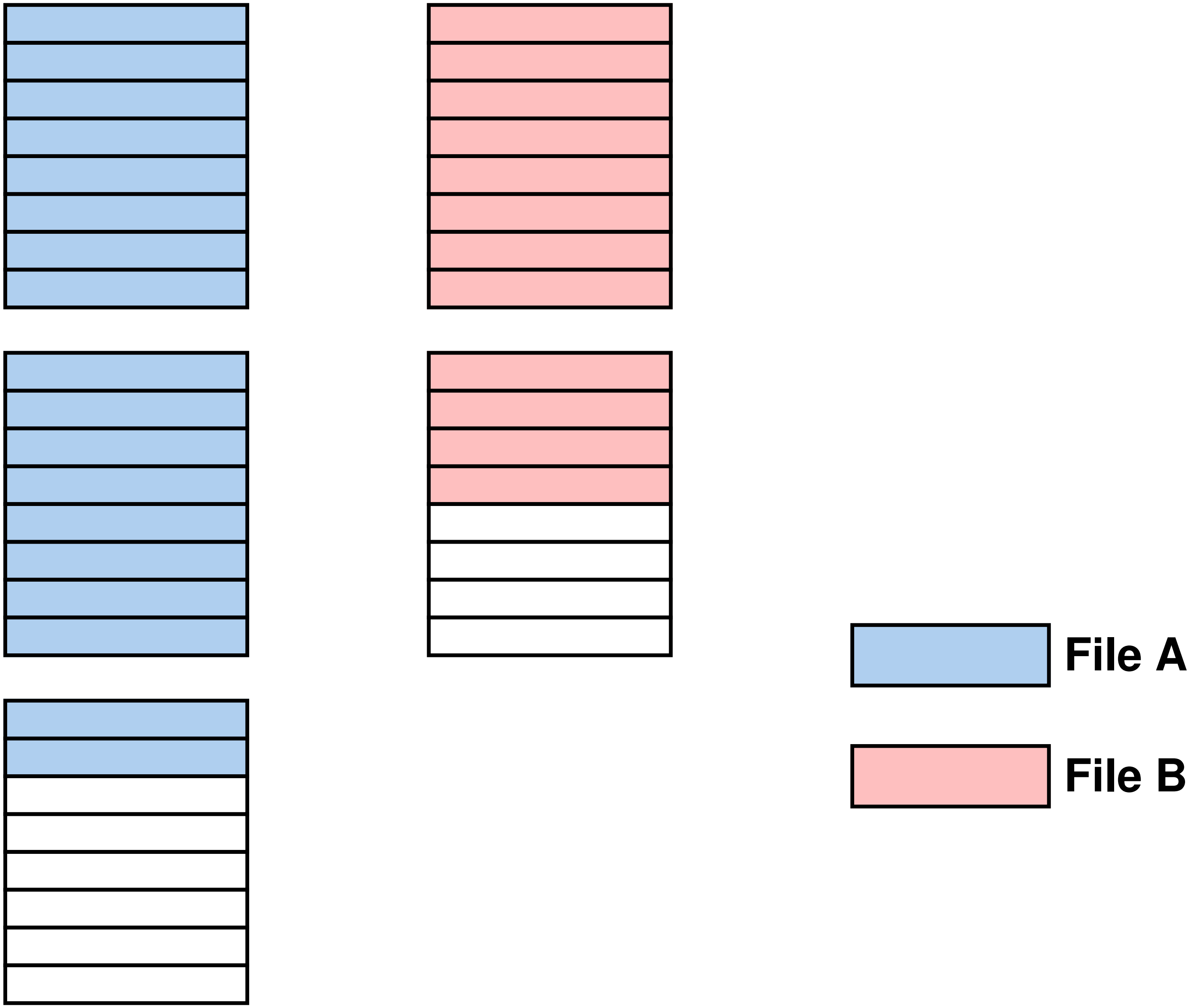
Use of Fragments (2)





- A can grow by 2 segments

Use of Fragments (3)

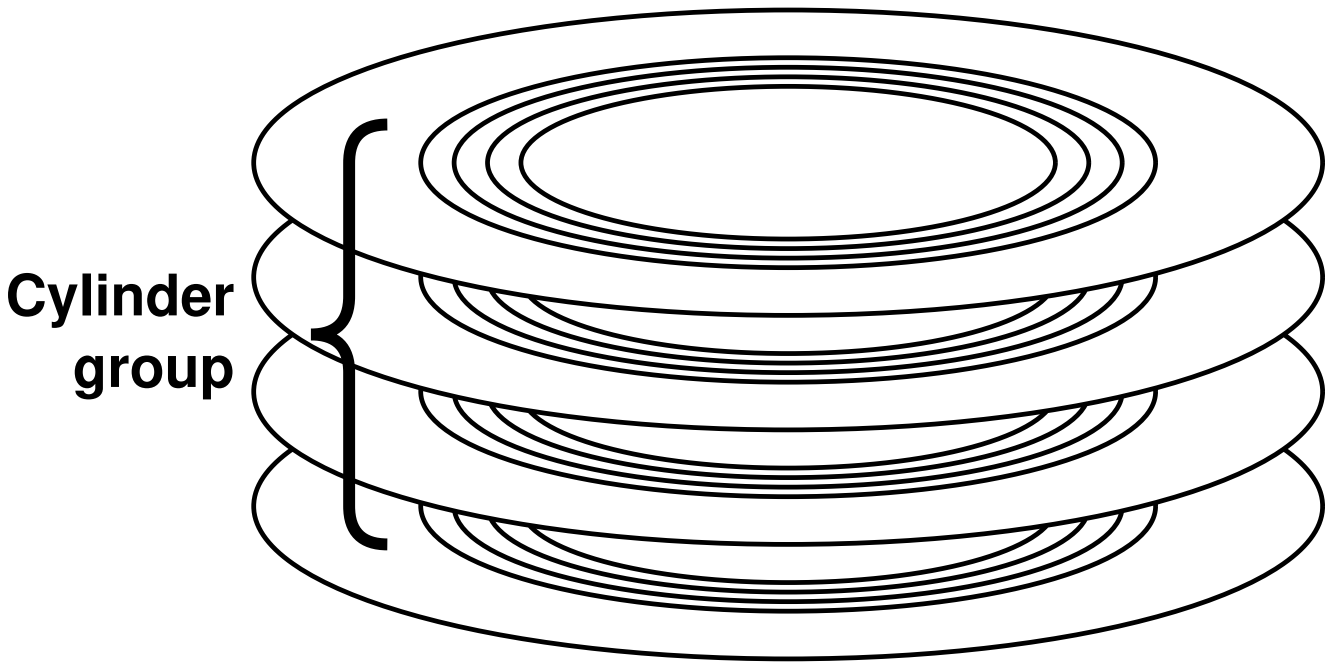


Minimizing Seek Time

Rotation speed	10,000 RPM
Number of surfaces	8
Sector size	512 bytes
Sectors/track	500-1000; 750 average
Tracks/surface	100,000
Storage capacity	307.2 billion bytes
Average seek time	*4 milliseconds*
One-track seek time	*.2 milliseconds*
Maximum seek time	10 milliseconds

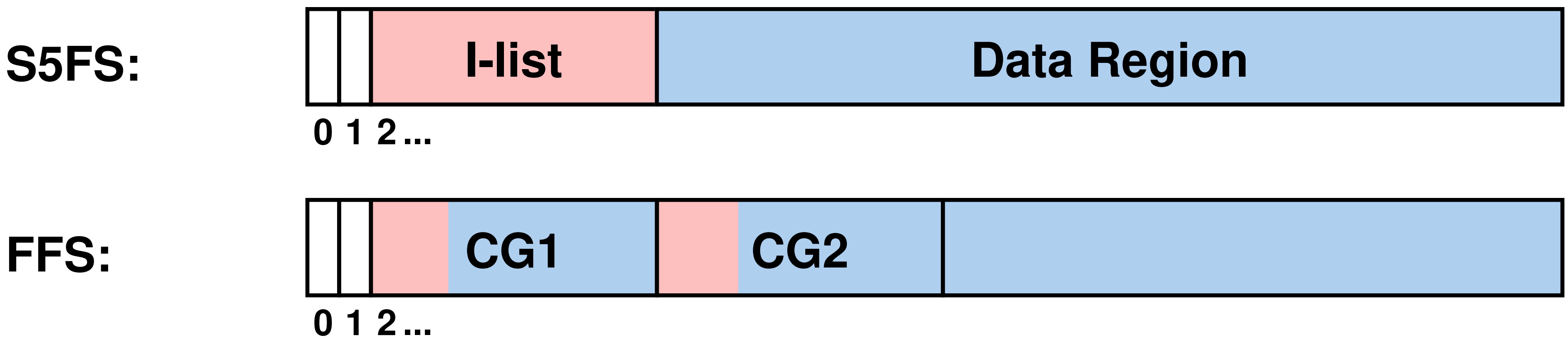
- Keep related things close to one another
- Separate unrelated things

Cylinder Groups



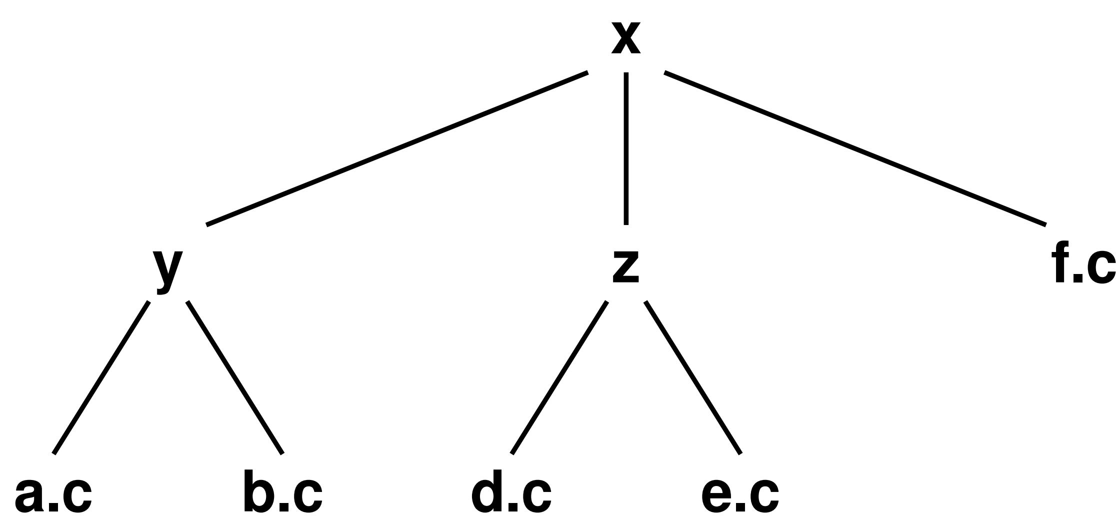
- recall that *seeking* to the *next cyliner/track* is much *faster* than seeking to a random track

Minimizing Seek Time



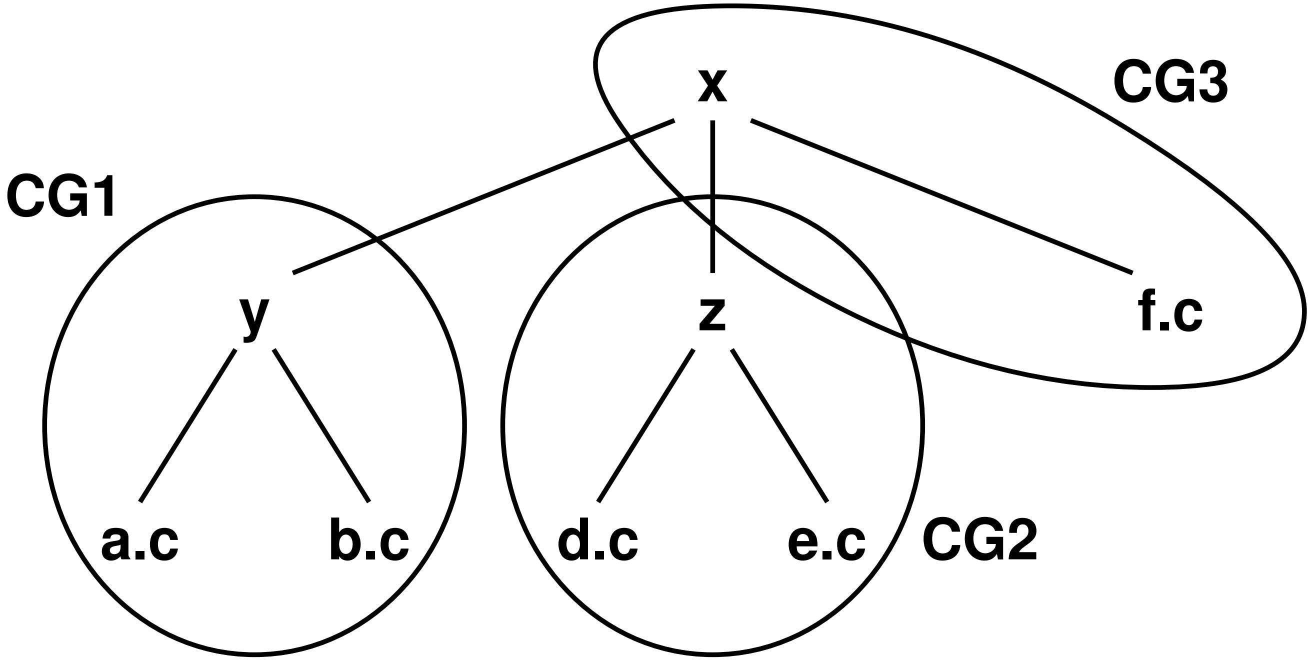
- Complication: which cylinder group to create a file?
  - the practice: use heuristics and not go for the "optimal solution"
    - attempt to put new inodes in the same cylinder group as their directories
    - put inodes for new directories in cylinder groups with "lots" of free space
    - put the 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

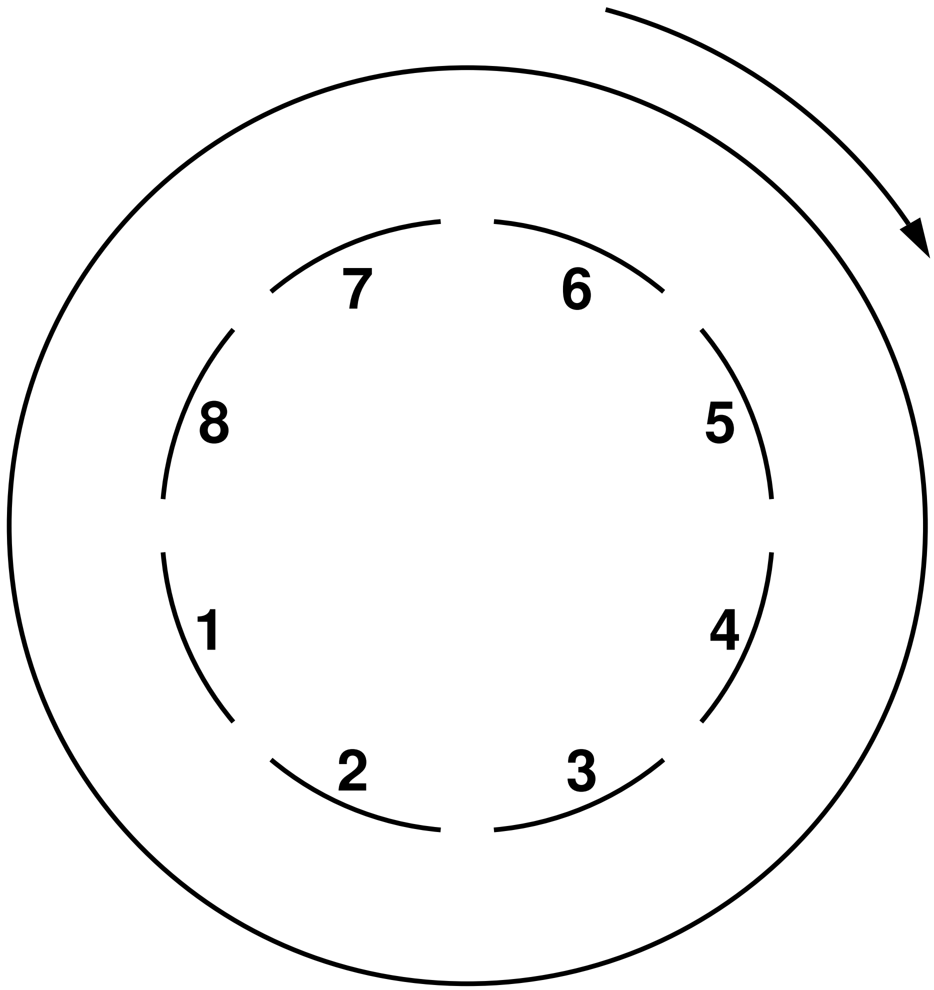


- if access "d.c", likely to access "e.c"

How Are We Doing? (Part 1)

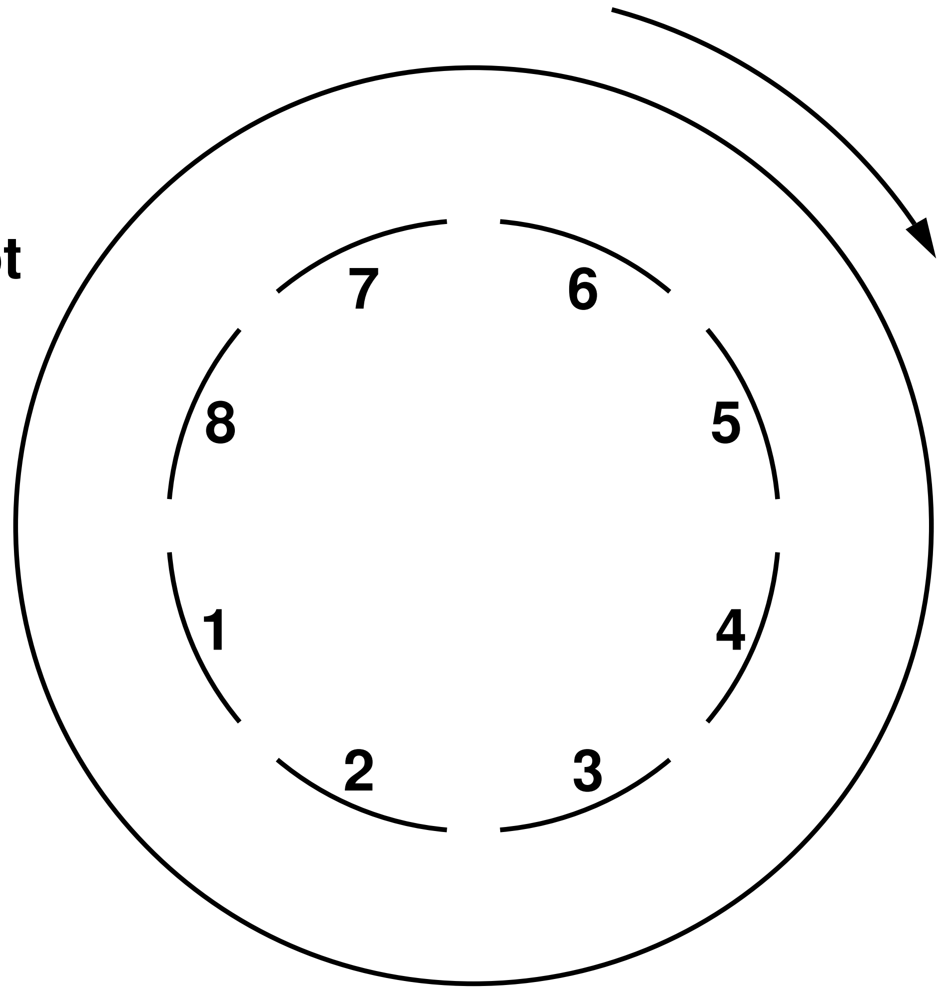
- Configure Rhinopias with 20 cylinders per group
  - 2-MB file fits entirely within one cylinder group
  - average seek time within cylinder group is ~.3 milliseconds
  - average rotational delay still 3 milliseconds
  - .12 milliseconds required for disk head to pass over 8KB block
  - 3.42 milliseconds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

Minimizing Latency

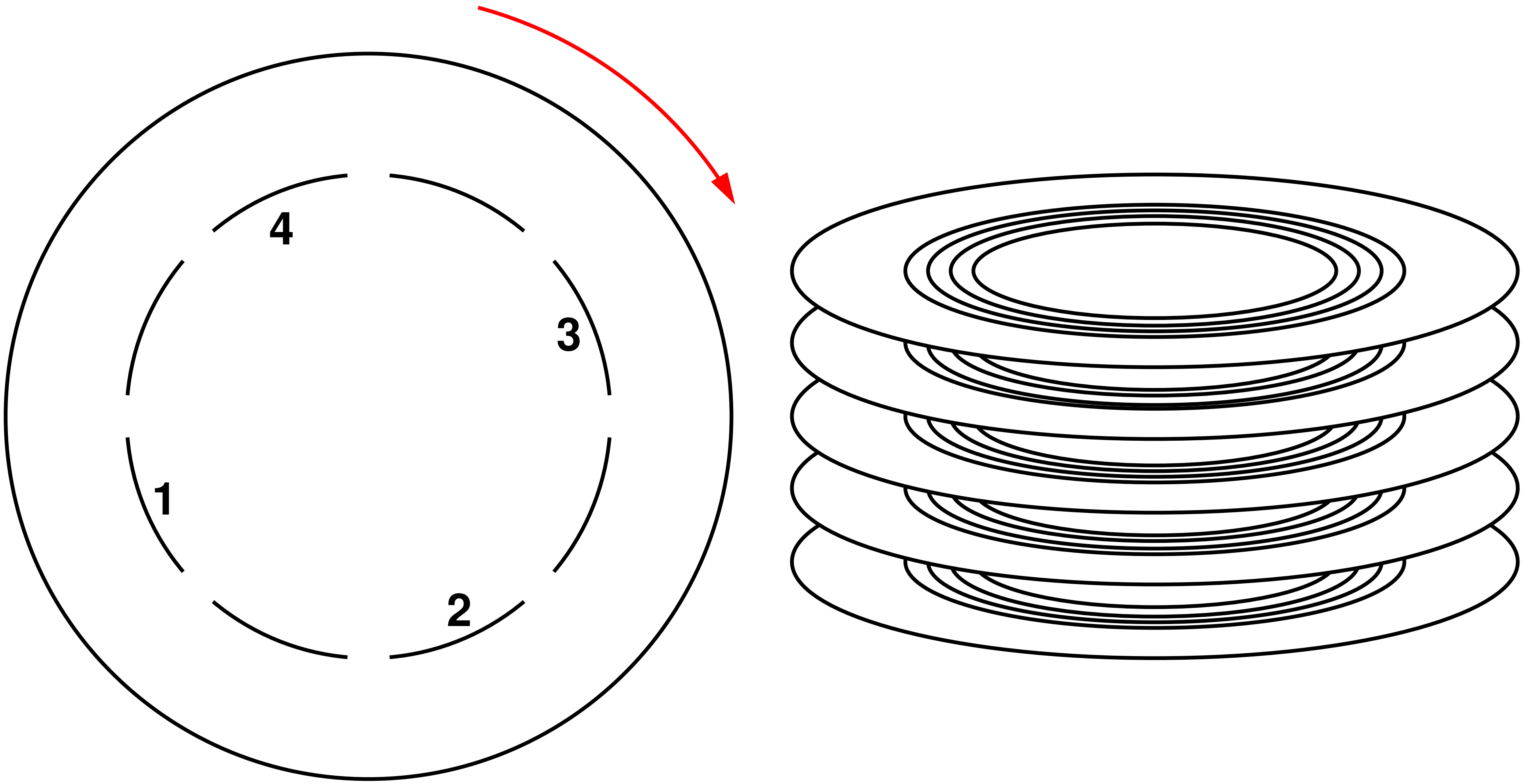


Numbers

- 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



- Block interleaving

How're We Doing Now? (Part 2)

- **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 15 improvement***
    - **together with other improvements, overall, a factor of 300 improvement**

How’re We Doing Now? (Altogether)

- 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
  - factor of 20 improvement
  - reached 3.8% of capacity
- FFS with block interleaving
  - another factor of 15 improvement
  - reached 50% of capacity
- Can we reach 100% of capacity?

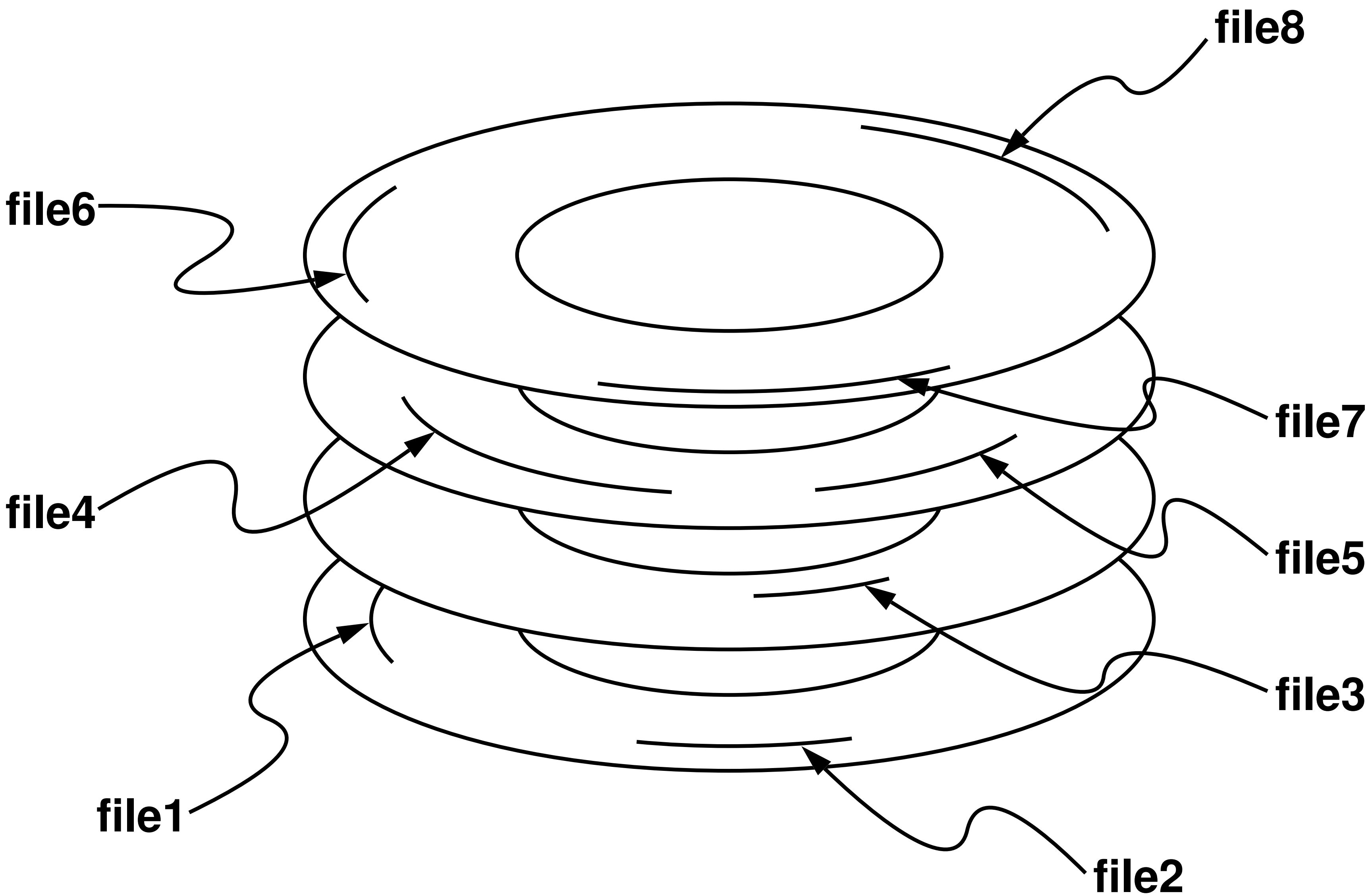
Larger Transfer Units

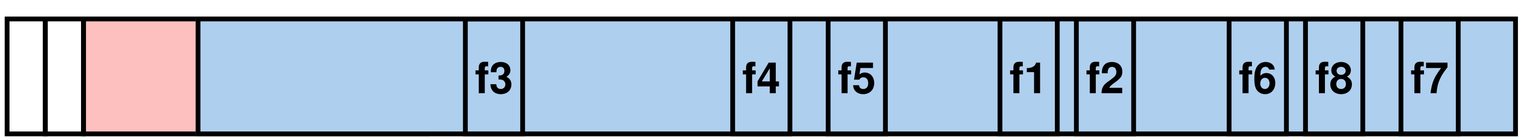
- Allocate in whole tracks or cylinders
  - too much wasted space
- Allocate in blocks, but group them together
  - transfer many at once

Block Clustering

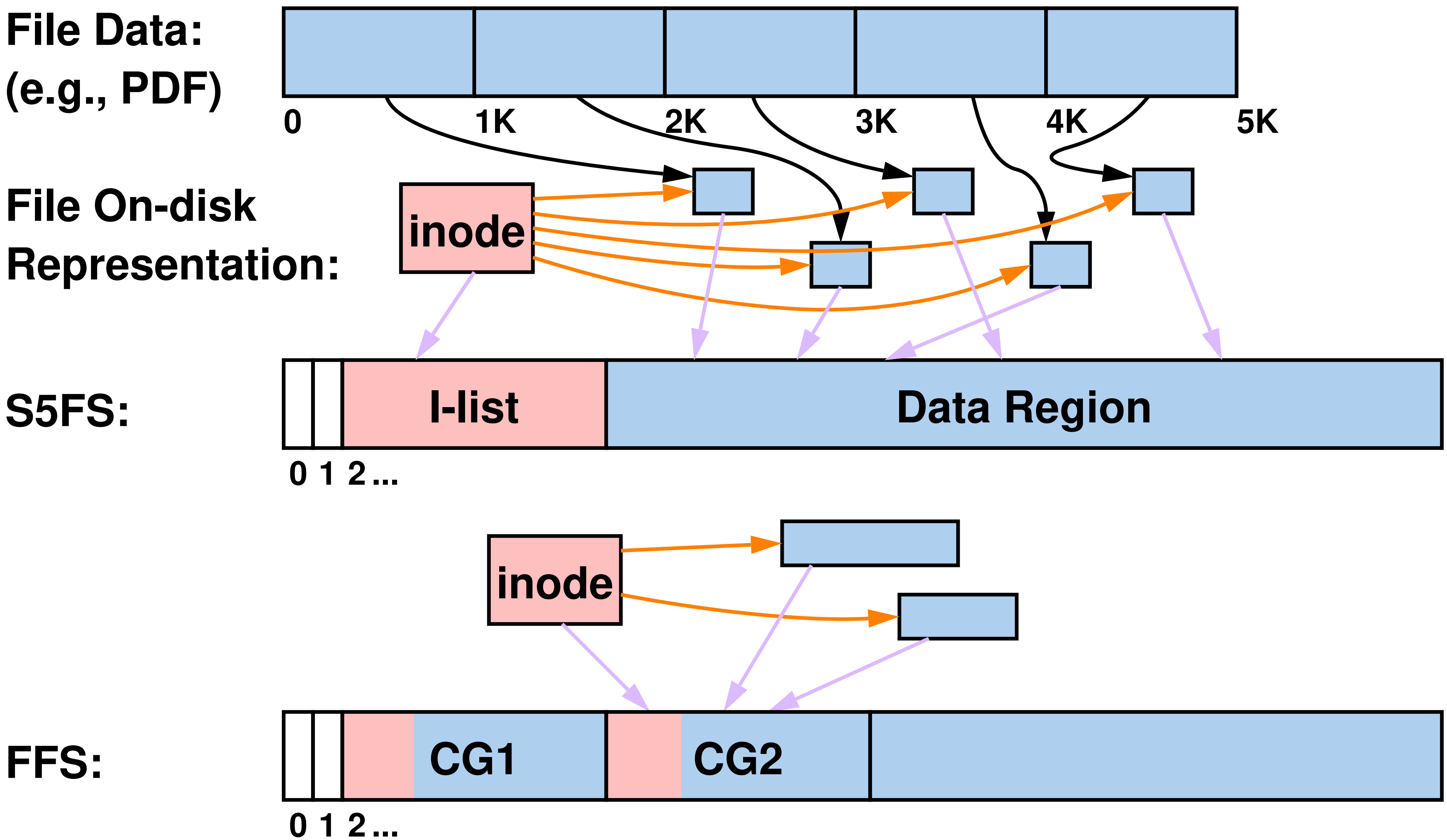
- Allocate space in blocks, eight at a time
- Linux's Ext2 (an FFS clone):
  - allocate eight blocks at a time
  - extra space is available to other files if there is a shortage of space
- FFS on Solaris (~1990)
  - delay disk-space allocation until:
    - 8 blocks are ready to be written
    - or the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

Can We Get To 100% Of Disk Transfer Capacity?



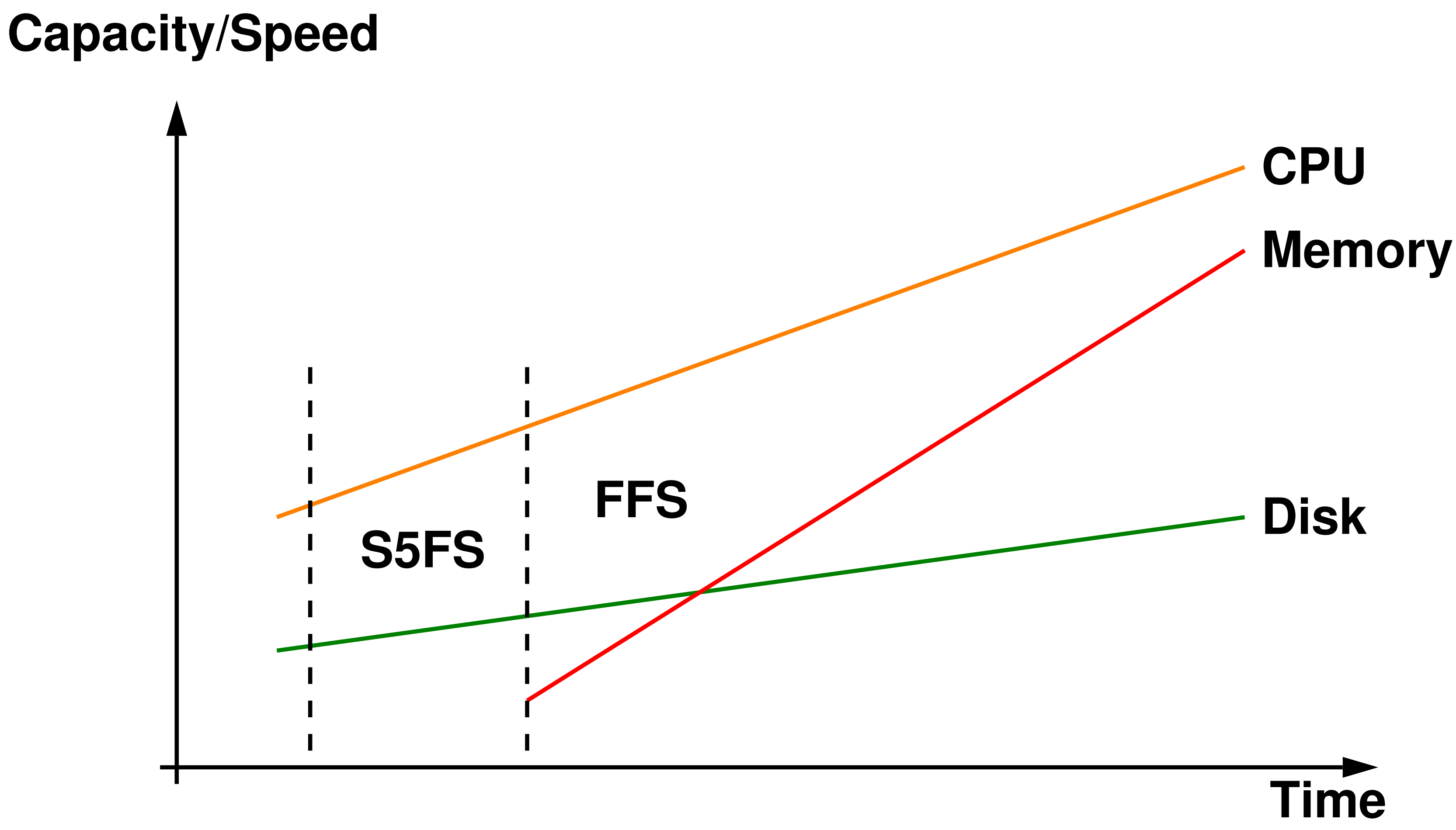


What A File Look Like in S5FS & FFS



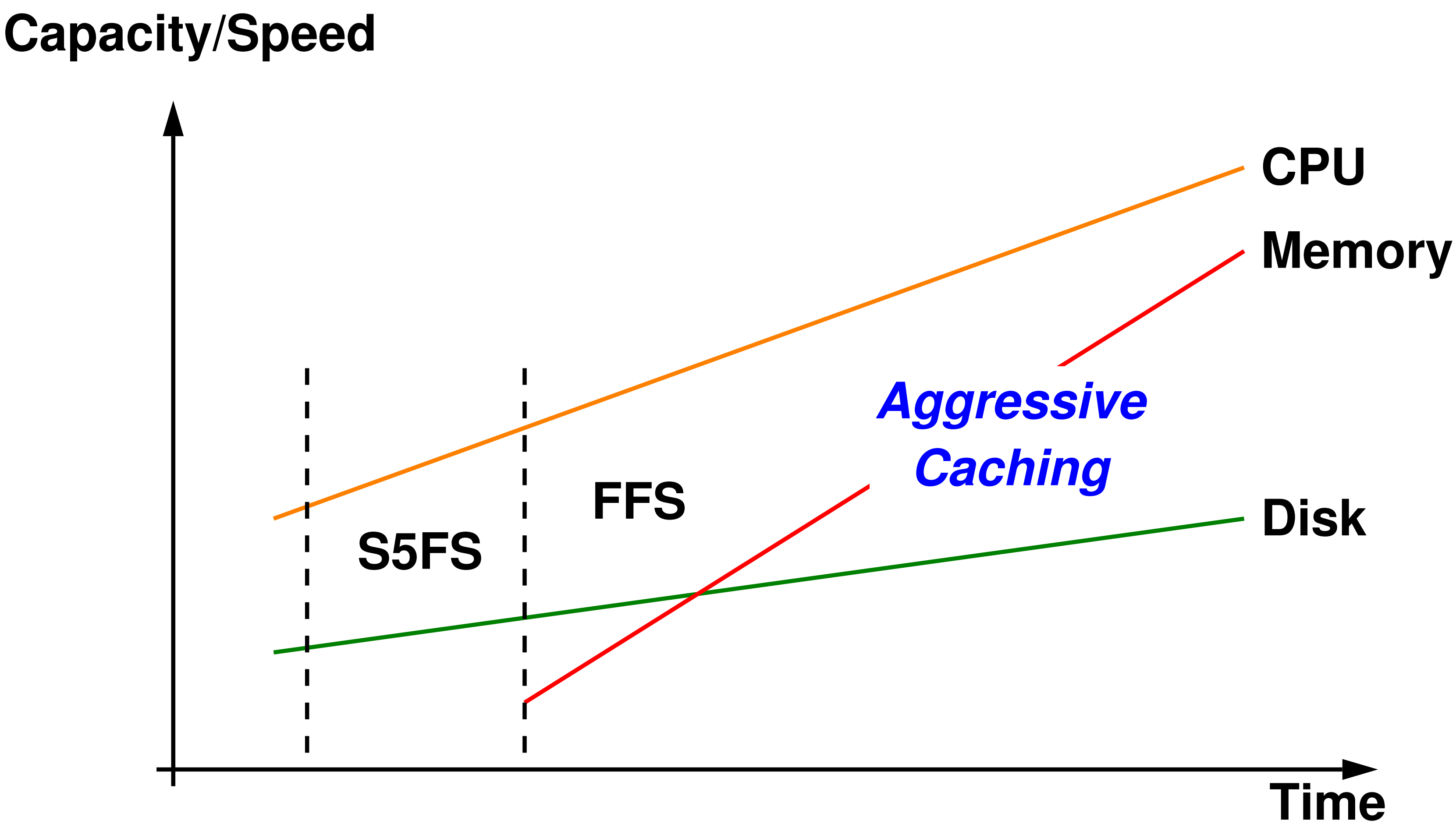
- Using all the tricks in FFS, we can only get to 50% of the disk transfer capacity

CPU, Memory, Disk Speeds Over Time



- this figure is not drawn to scale

CPU, Memory, Disk Speeds Over Time

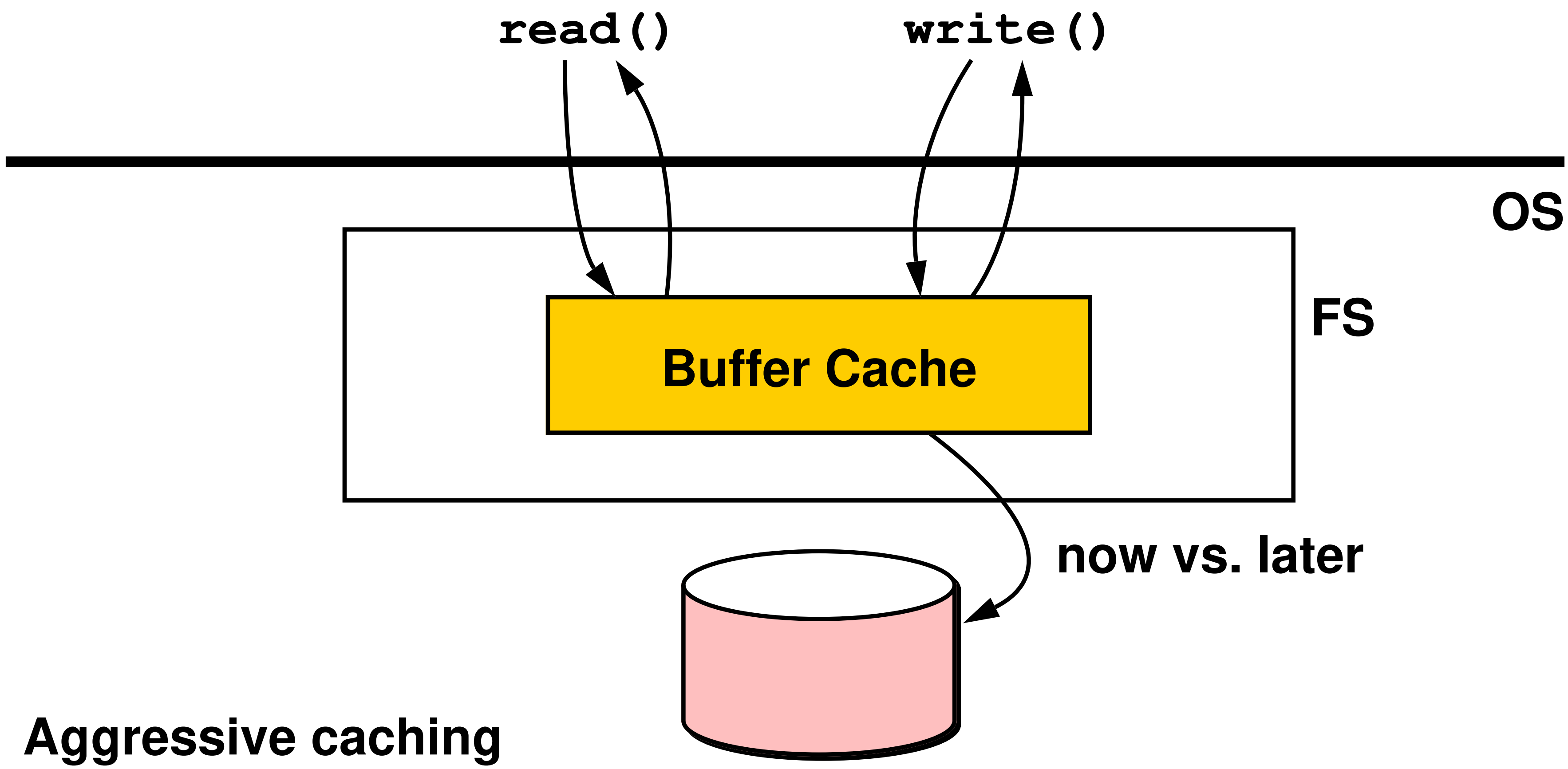


- this figure is not drawn to scale

A Different Approach

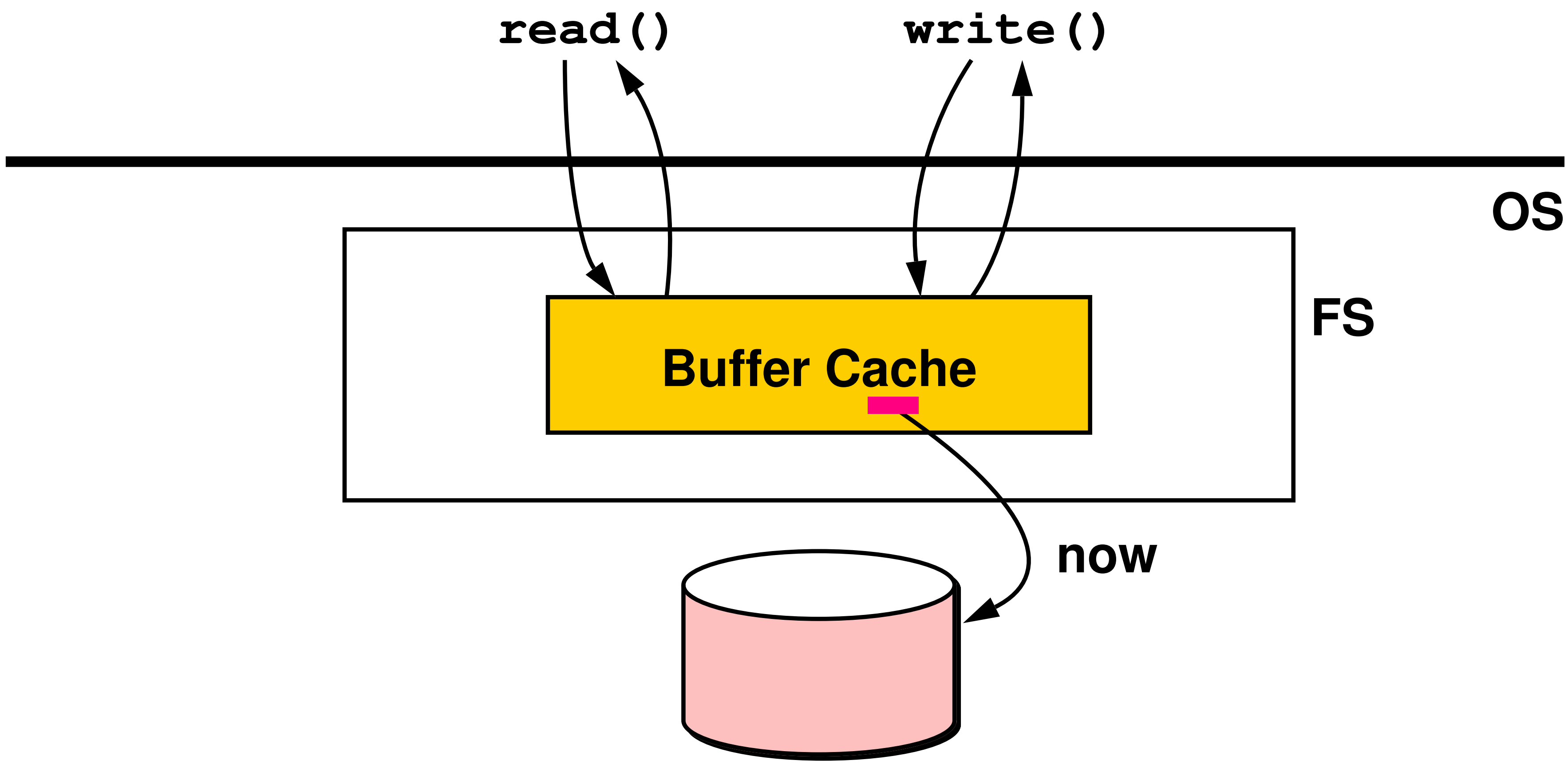
- We have lots of primary memory
  - cache the entire disk in memory
    - if disk is too big, need to cache intelligently
    - may be enough to cache only files being accesses
- Read time from disk doesn't matter
  - once a disk block is brought into memory, keep it in memory
    - next read of this block will cost you 0 in disk access time
  - with a high hit rate, performance can get > 100% of disk capacity
    - e.g., with a 90% hit rate, the disk will appear to be 10 times faster; with a 99% hit rate, the disk will appear to be 100 times faster
- Time for writes does matter

The Buffer Cache



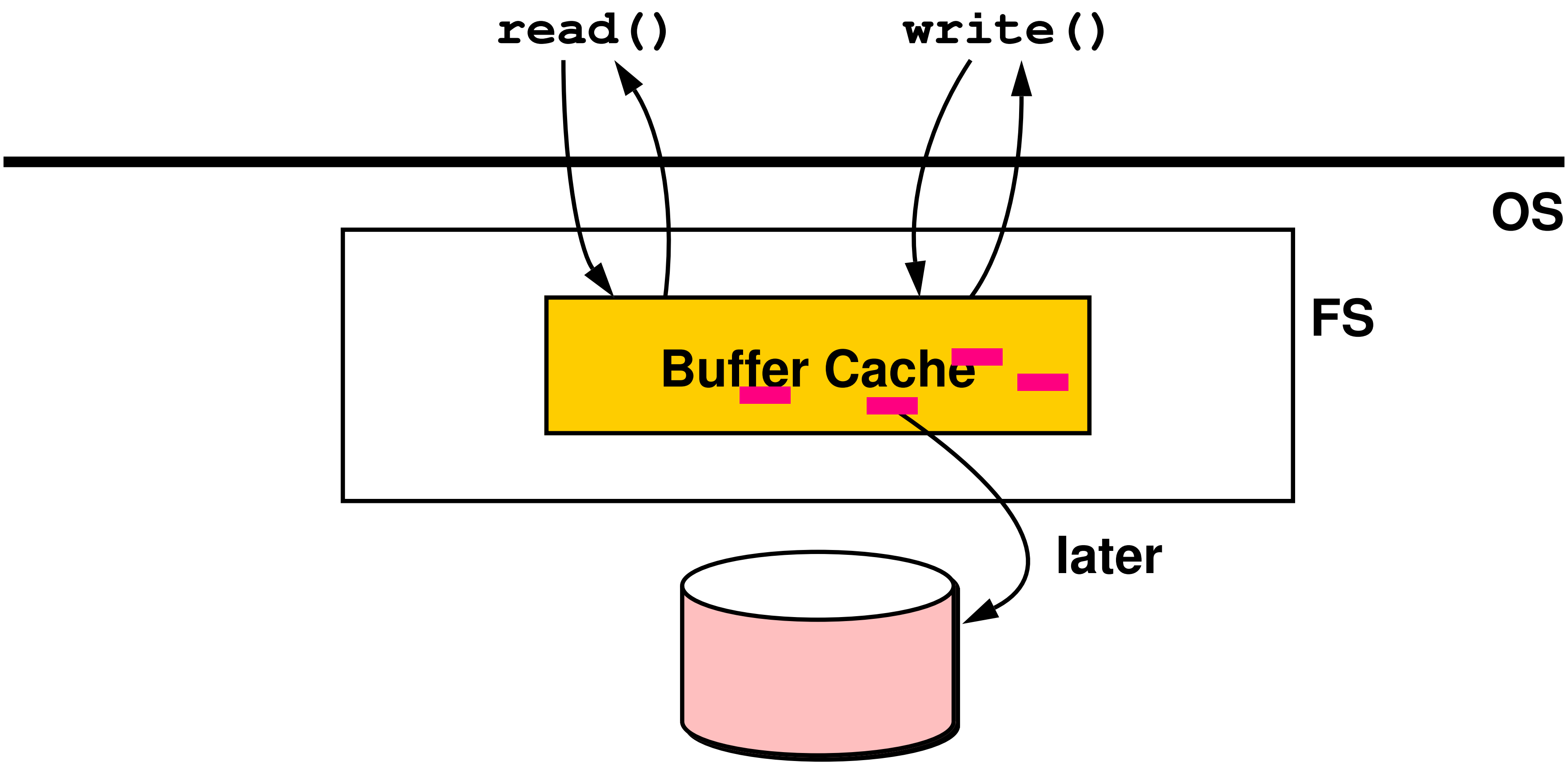
- Aggressive caching
  - most read and write will have a *cache hit*
    - can perform > 100% of the disk transfer capacity for reads
    - if buffer cache is used, read performance no longer an issue
  - for writes, need to update the disk
    - *write-through* vs. *write-back*

The Buffer Cache



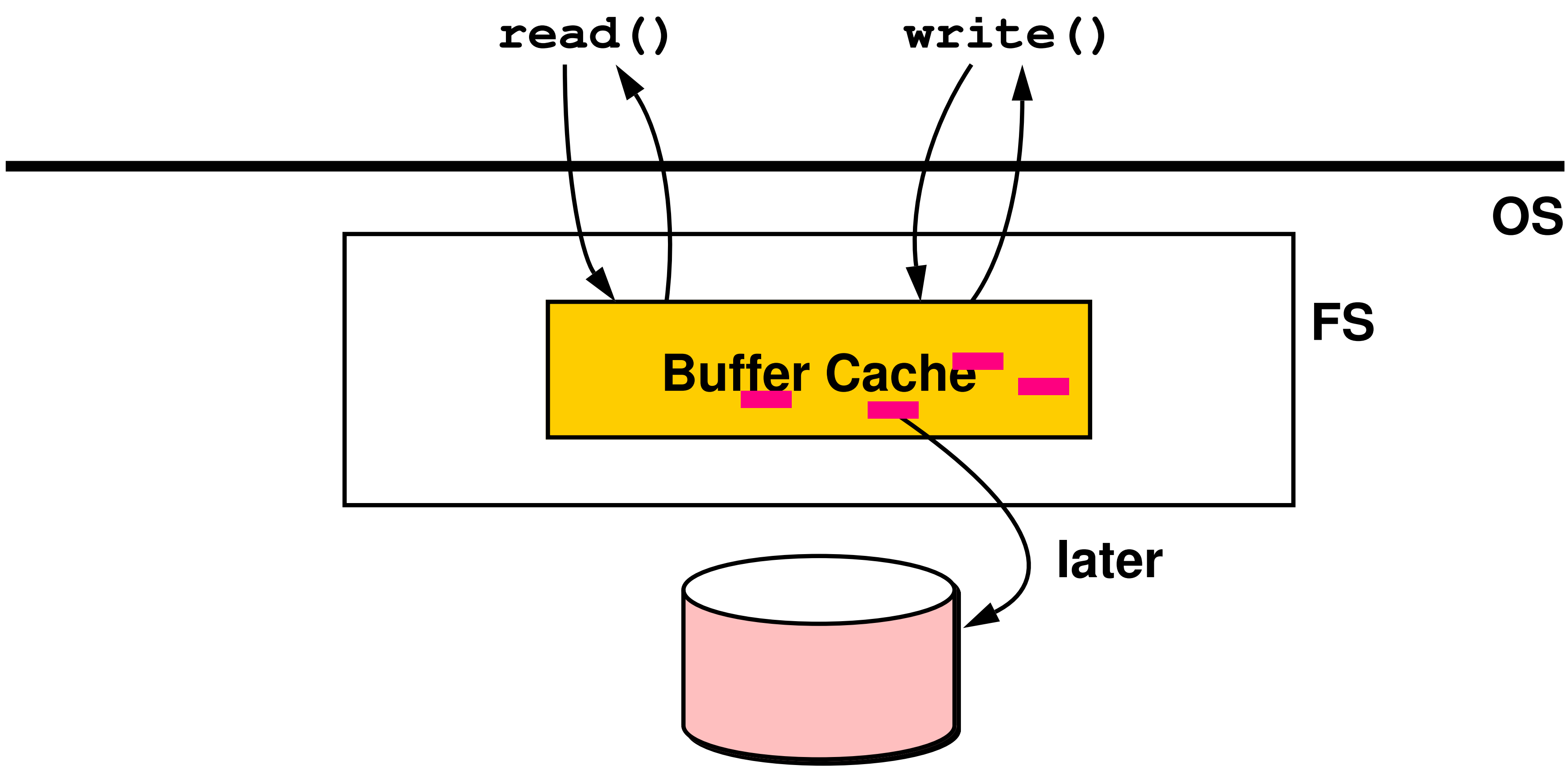
- Problems with *write-through*
  - slow

The Buffer Cache



- Problems with *write-back*
  - writes to the disk can wait, may be for quite a while
    - longer the wait, higher the *risk*

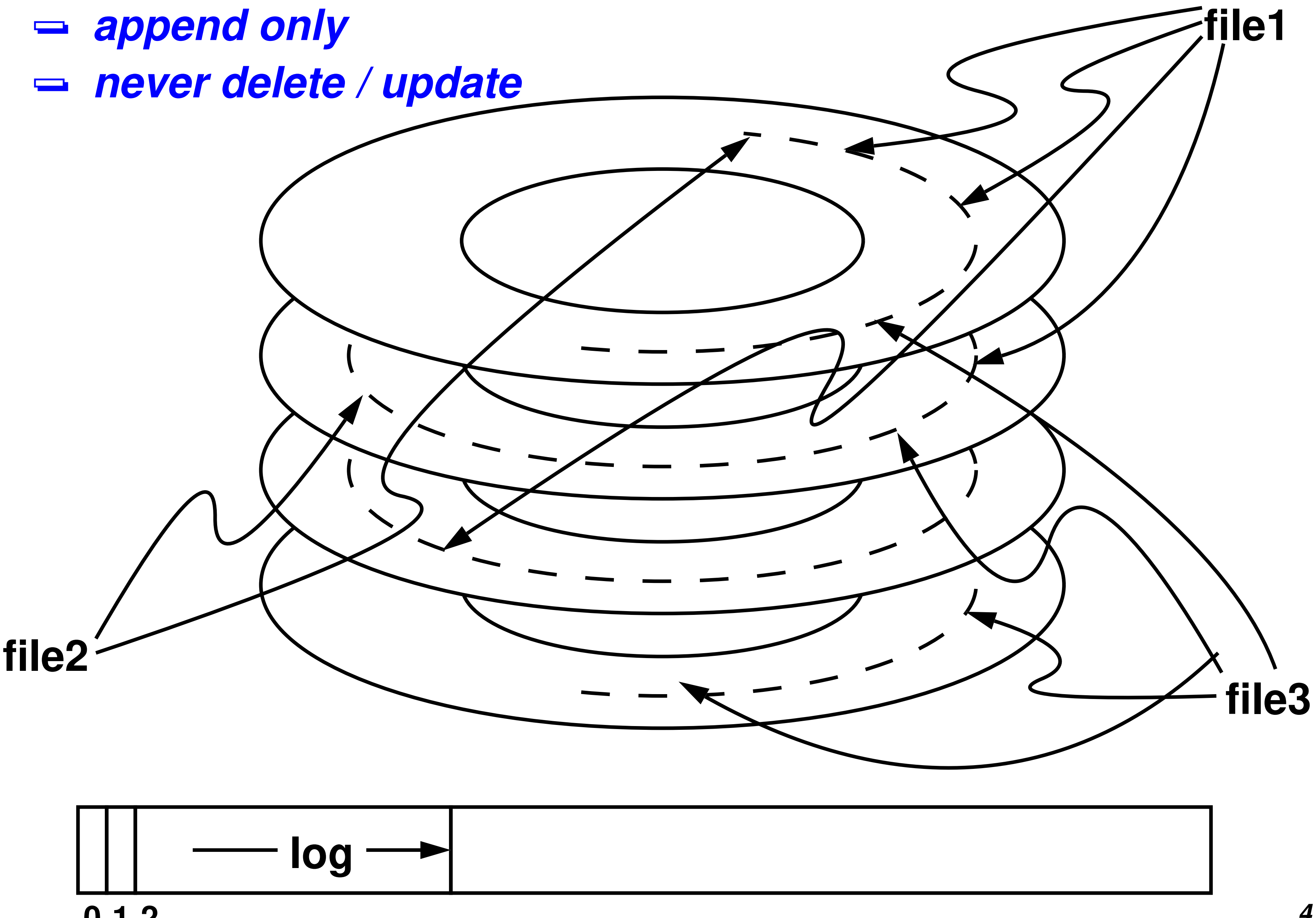
The Buffer Cache



- Need a file system optimized for *writing*!
  - how?
    - you organize the disk as a very long *log*
    - also need to address the "risk factor"

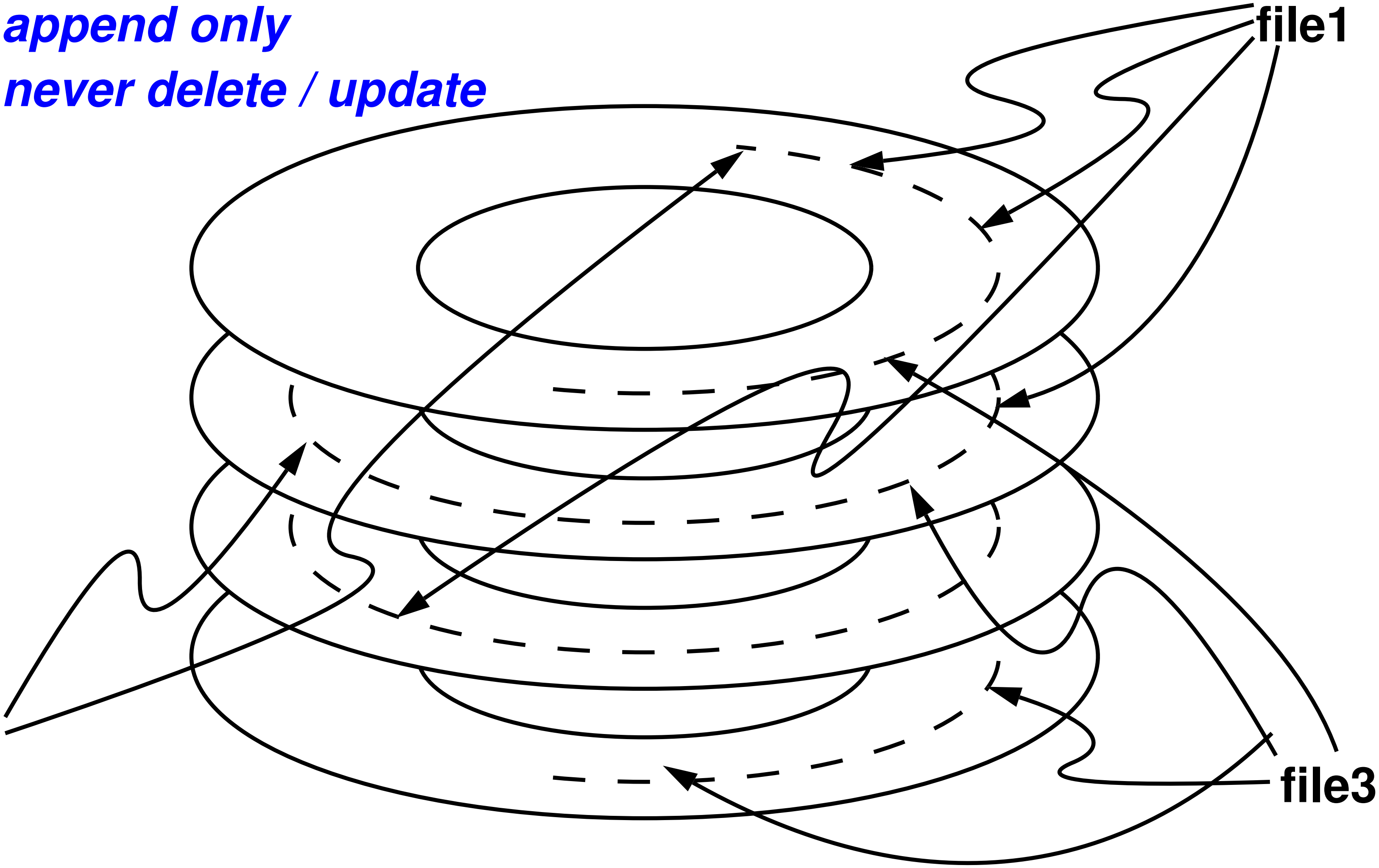
Log-Structured File Systems

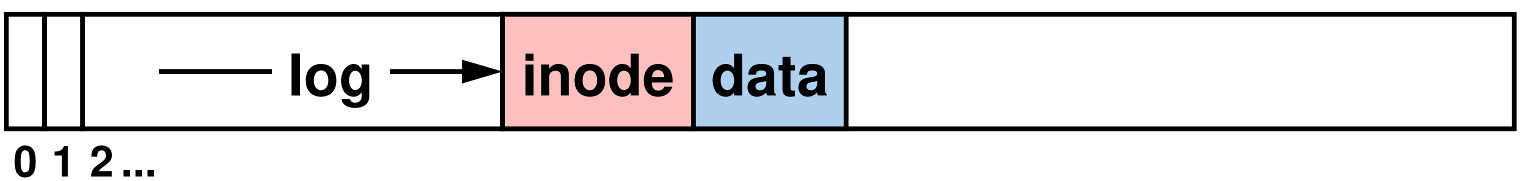
- **Main principles**
  - *append only*
  - *never delete / update*



Log-Structured File Systems

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



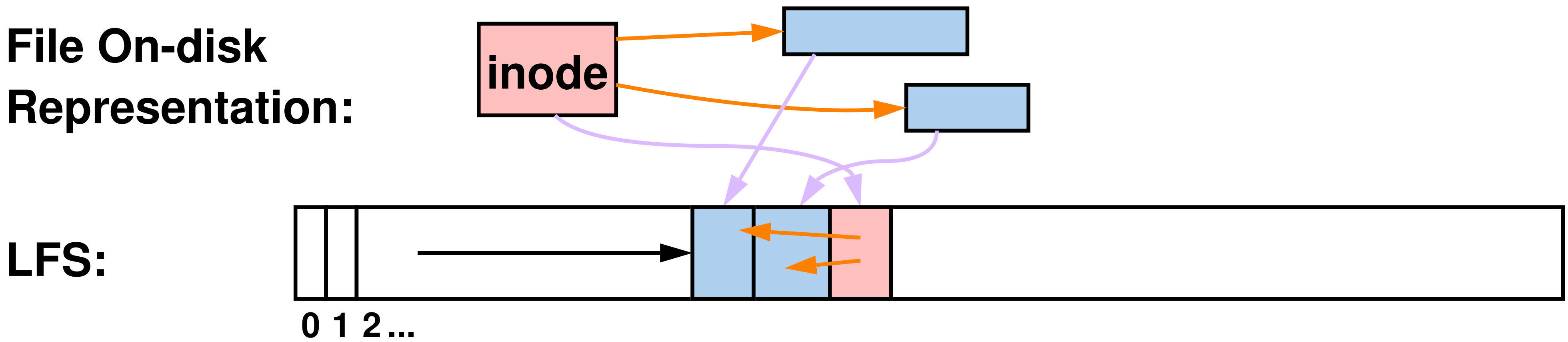


Log-Structured File Systems

- 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

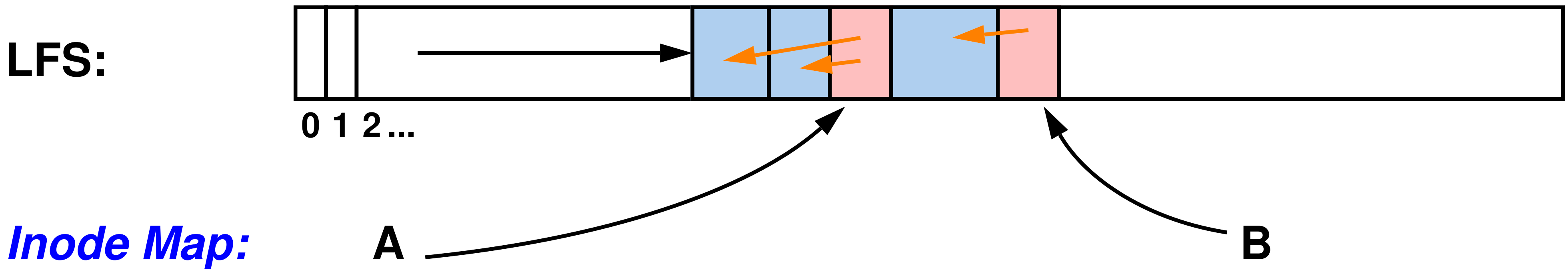
- *Sprite FS* (a log-structured file system)
  - through batching, a single, long write can write out everything

LFS Data Placement Example



LFS Data Placement Example

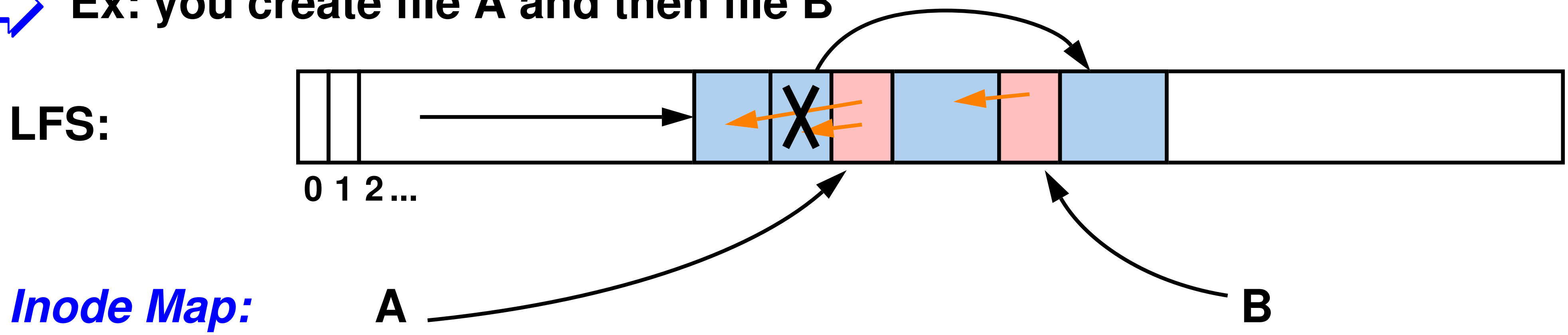
- What happens if you want to modify the file?
  - how does "append-only" really work?
- Ex: you create file A and then file B



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

LFS Data Placement Example

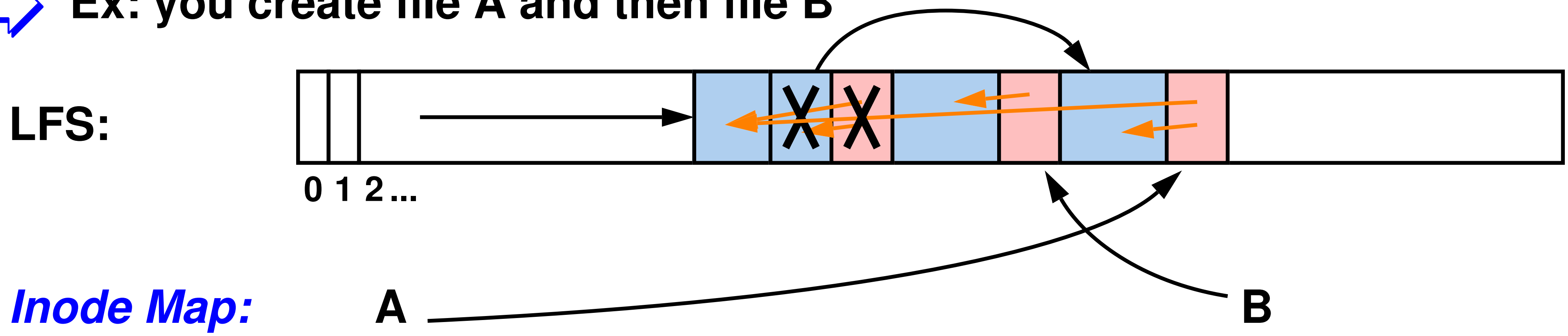
- What happens if you want to modify the file?
  - how does "append-only" really work?
- Ex: you create file A and then file B



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

LFS Data Placement Example

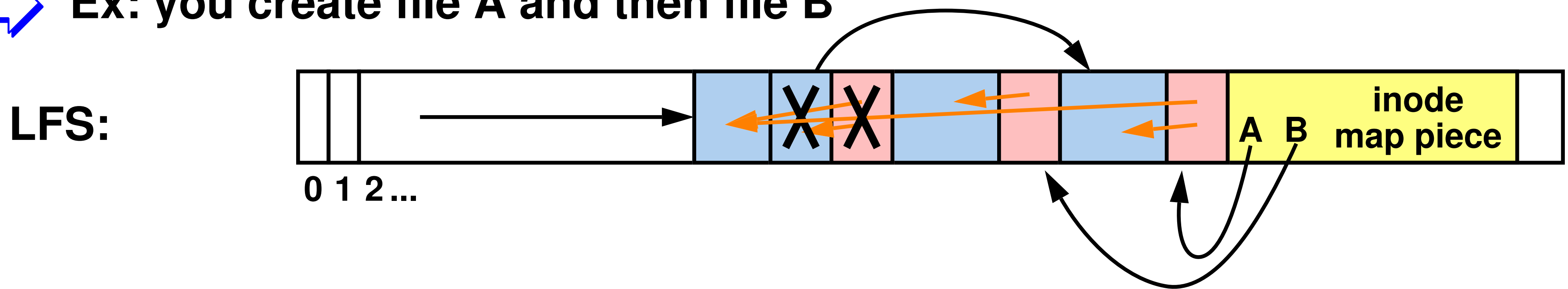
- What happens if you want to modify the file?
  - how does "append-only" really work?
- Ex: you create file A and then file B



  - 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

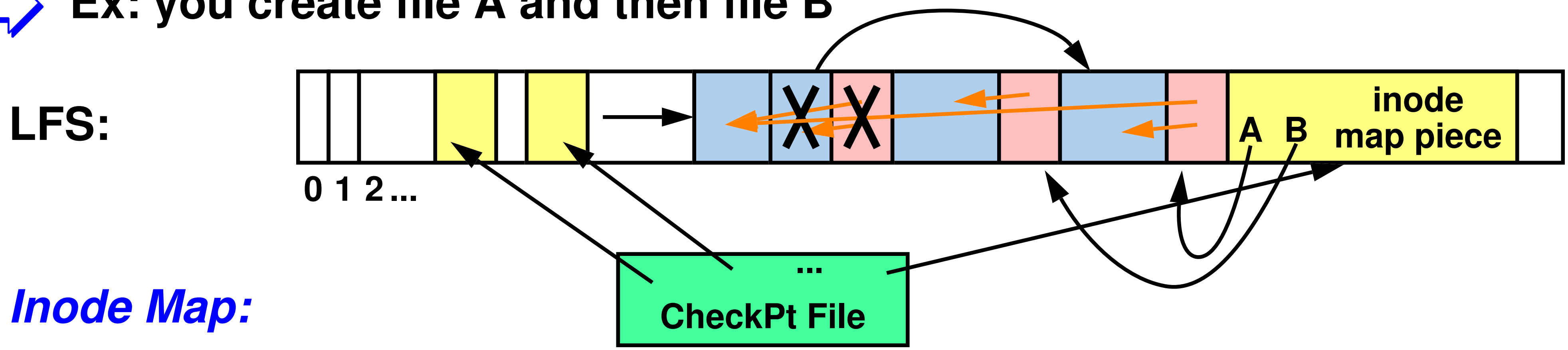


Inode Map:

- 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

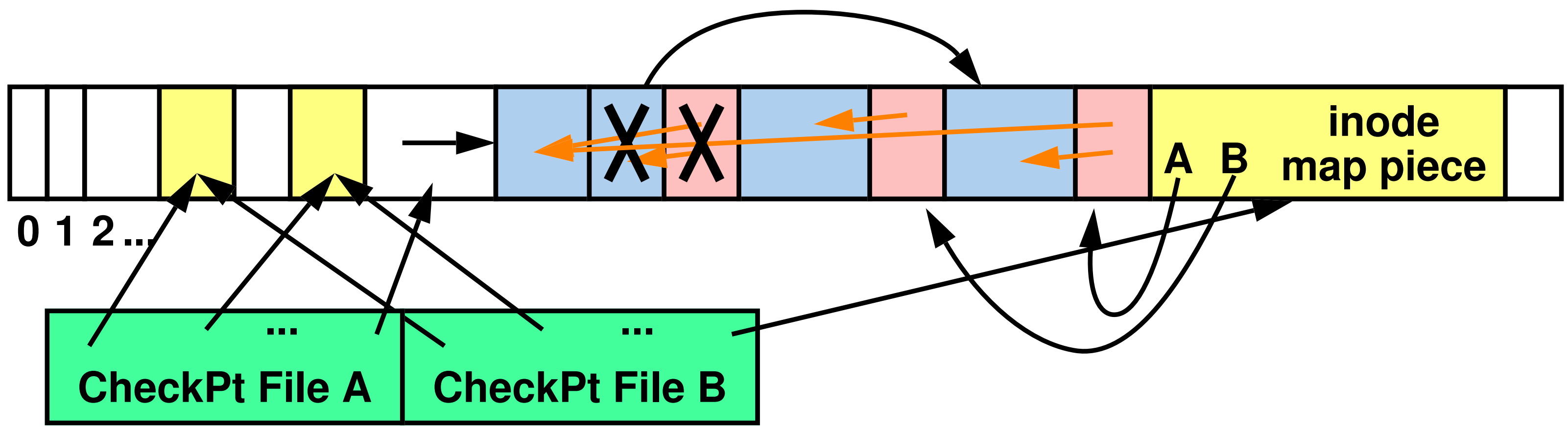


Inode Map:

- 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**

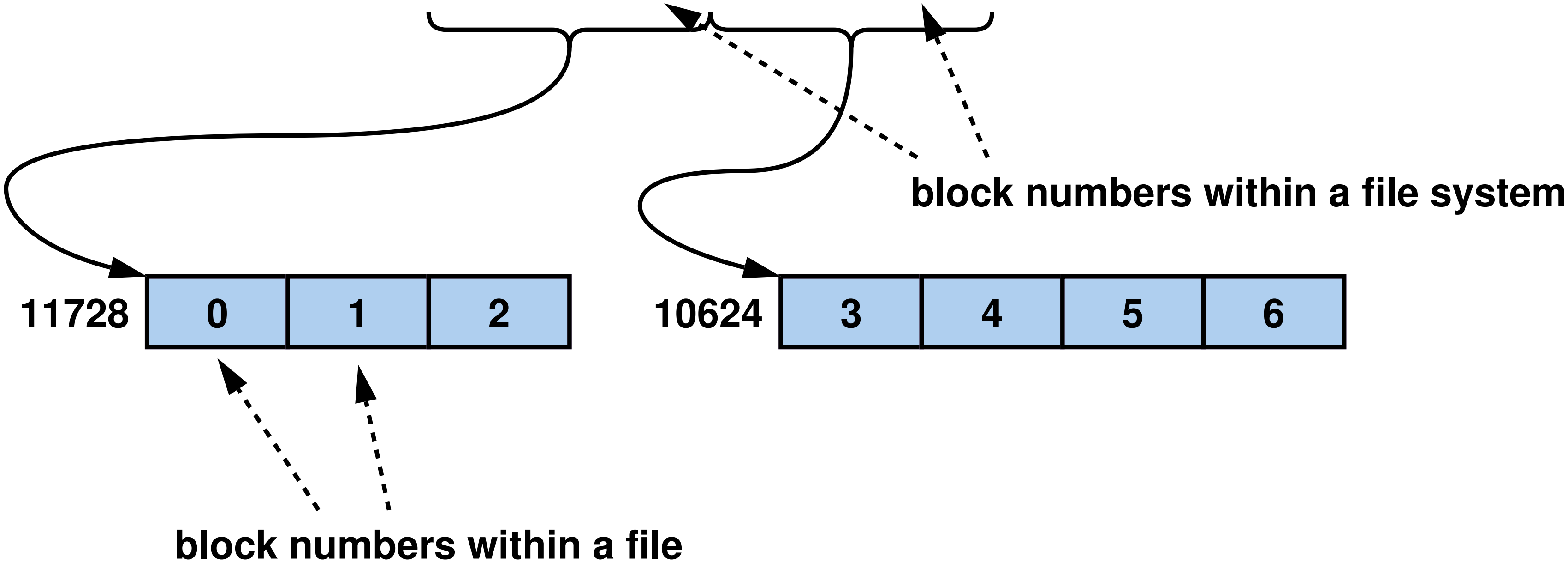
LFS Summary

- **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

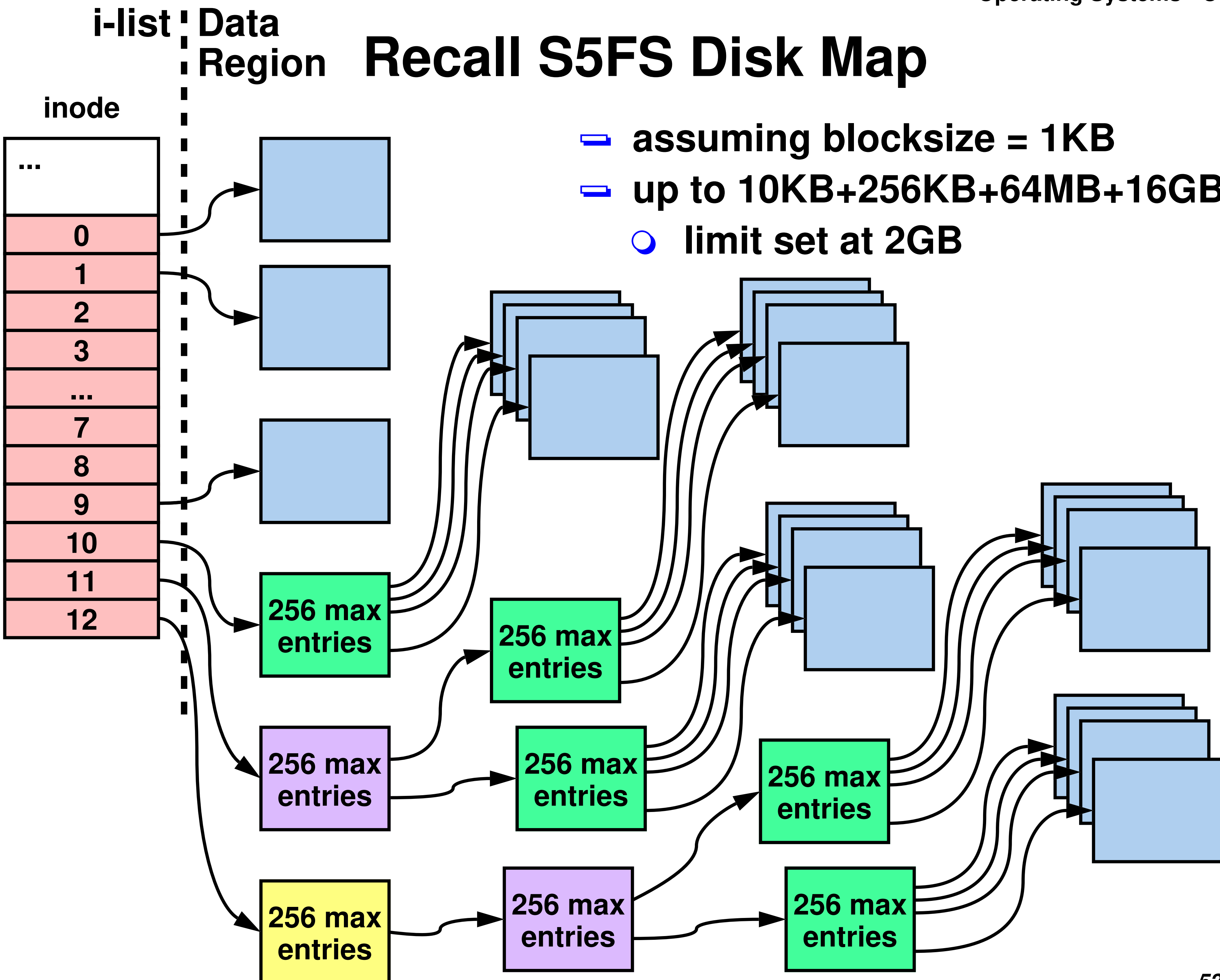
- Windows' equivalent of *disk map* in S5FS is *extent*
  - an *extent* is a *list* of *runs* (consecutive disk blocks)

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

Top-level runlist							
length	offset	length	offset	length	offset	length	offset
84	9738	132	1076	98	124	...	...

9738

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

11728

0	1	2

10624

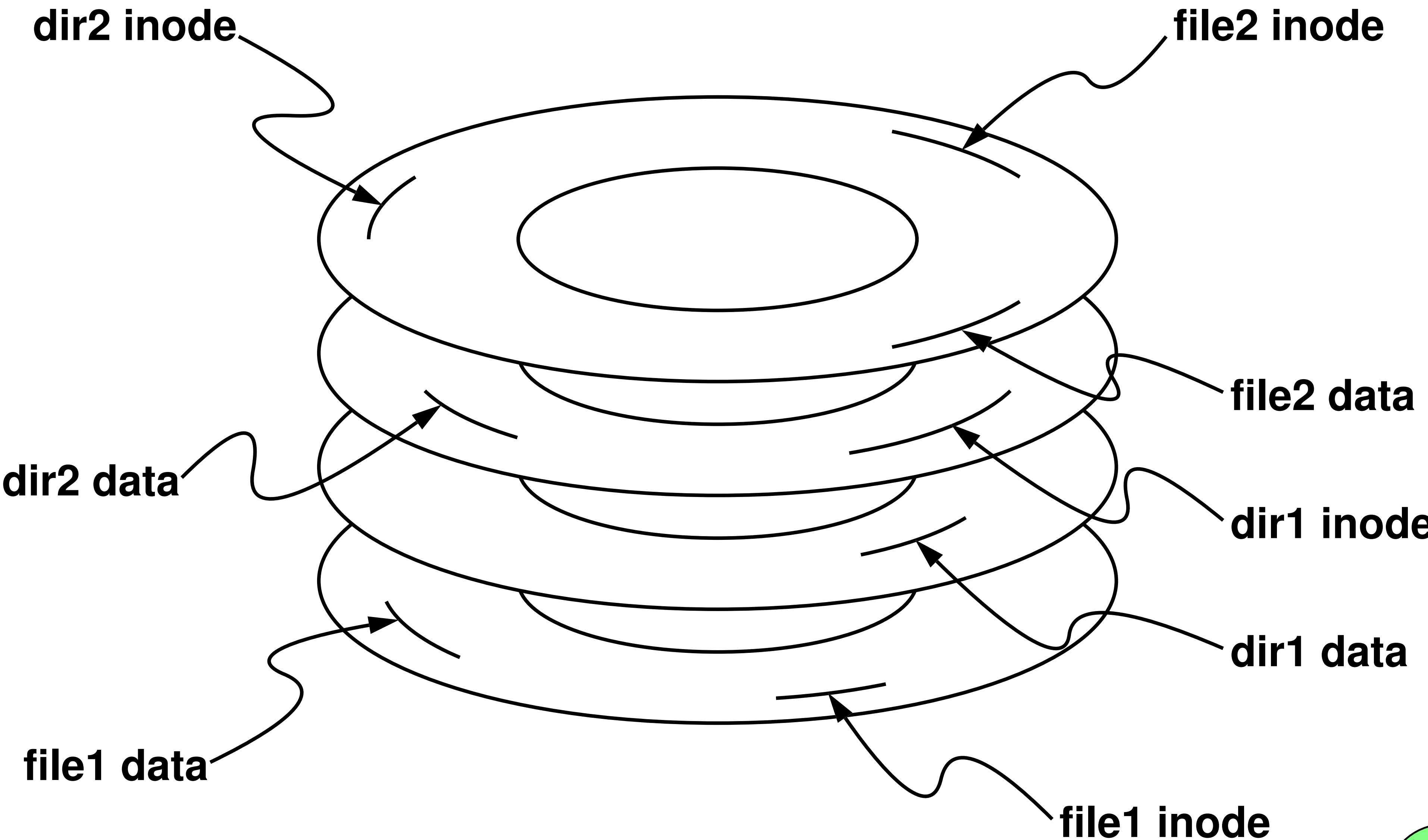
3	4	5	6

Extra Slides

Example

- We create two single-block files
  - dir1/file1
  - dir2/file2
- FFS
  - allocate and initialize inode for file1 and write it to disk
  - update dir1 to refer to it (and update dir1 inode)
  - write data to file1
    - allocate disk block
    - fill it with data and write to disk
    - update inode
  - six writes, plus six more for the other file
    - seek and rotational delays

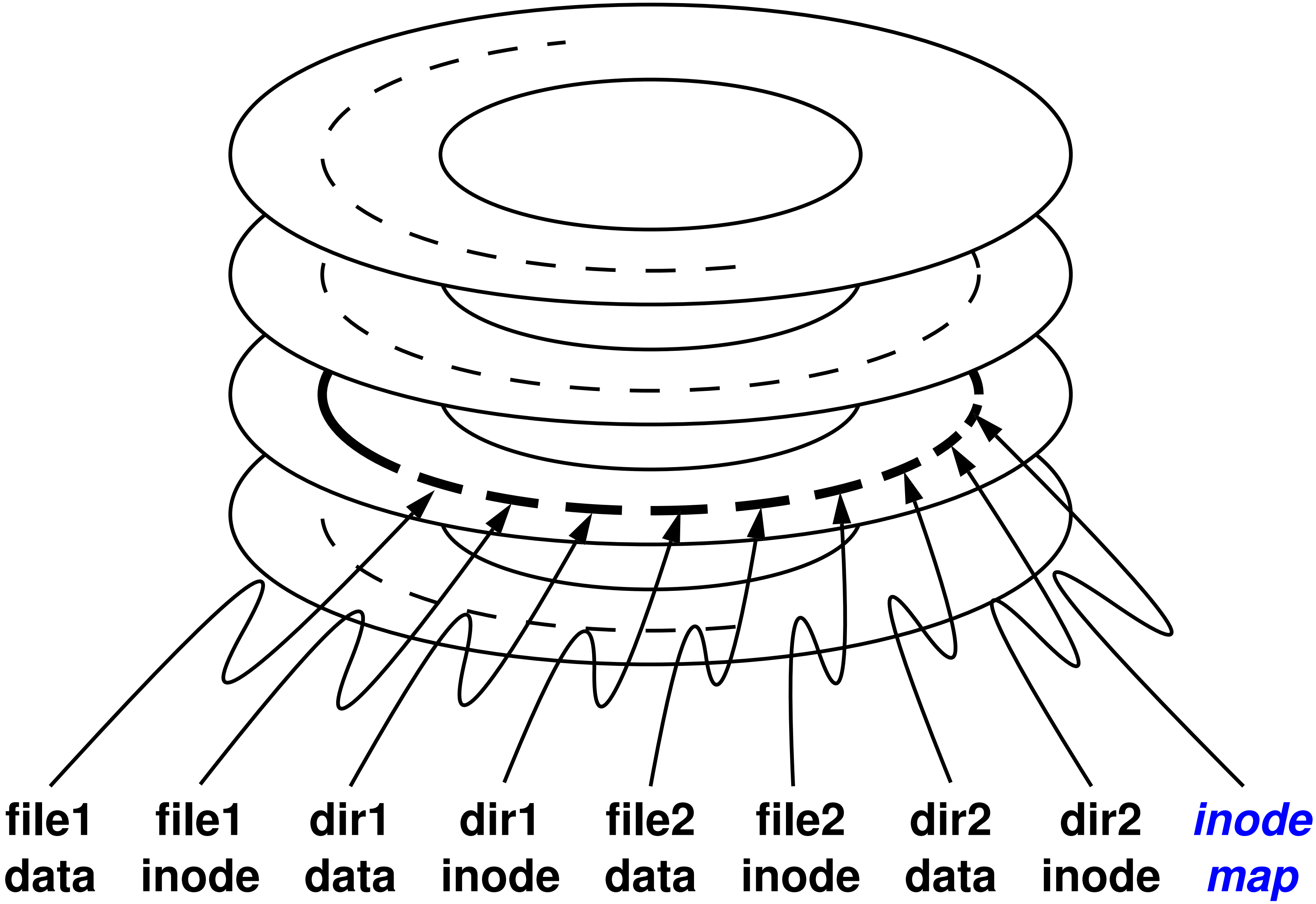
FFS Picture


dir2 inode
file2 inode
file2 data
dir2 data
dir1 inode
dir1 data
file1 data
file1 inode

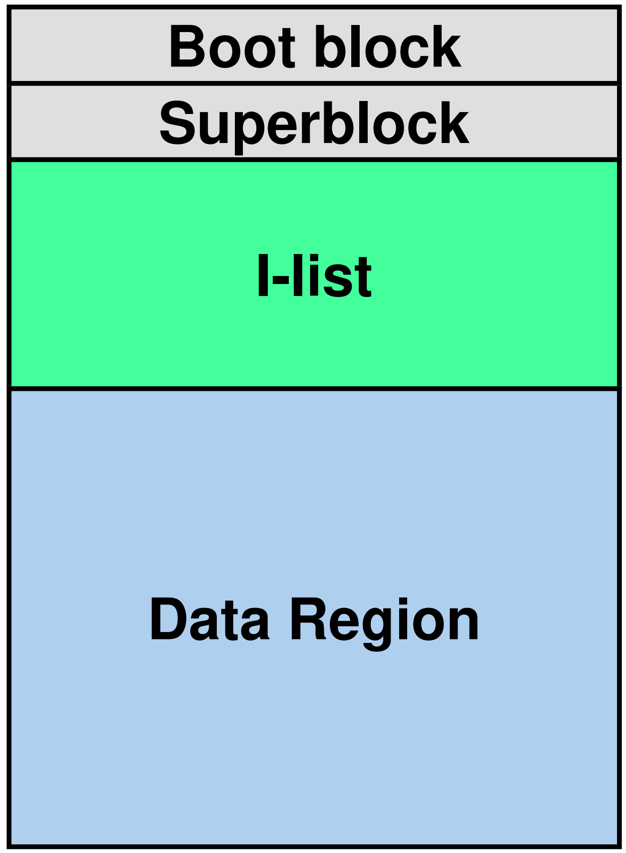
Example (Continued)

- Sprite (a log-structured file system)
  - one single, long write does everything

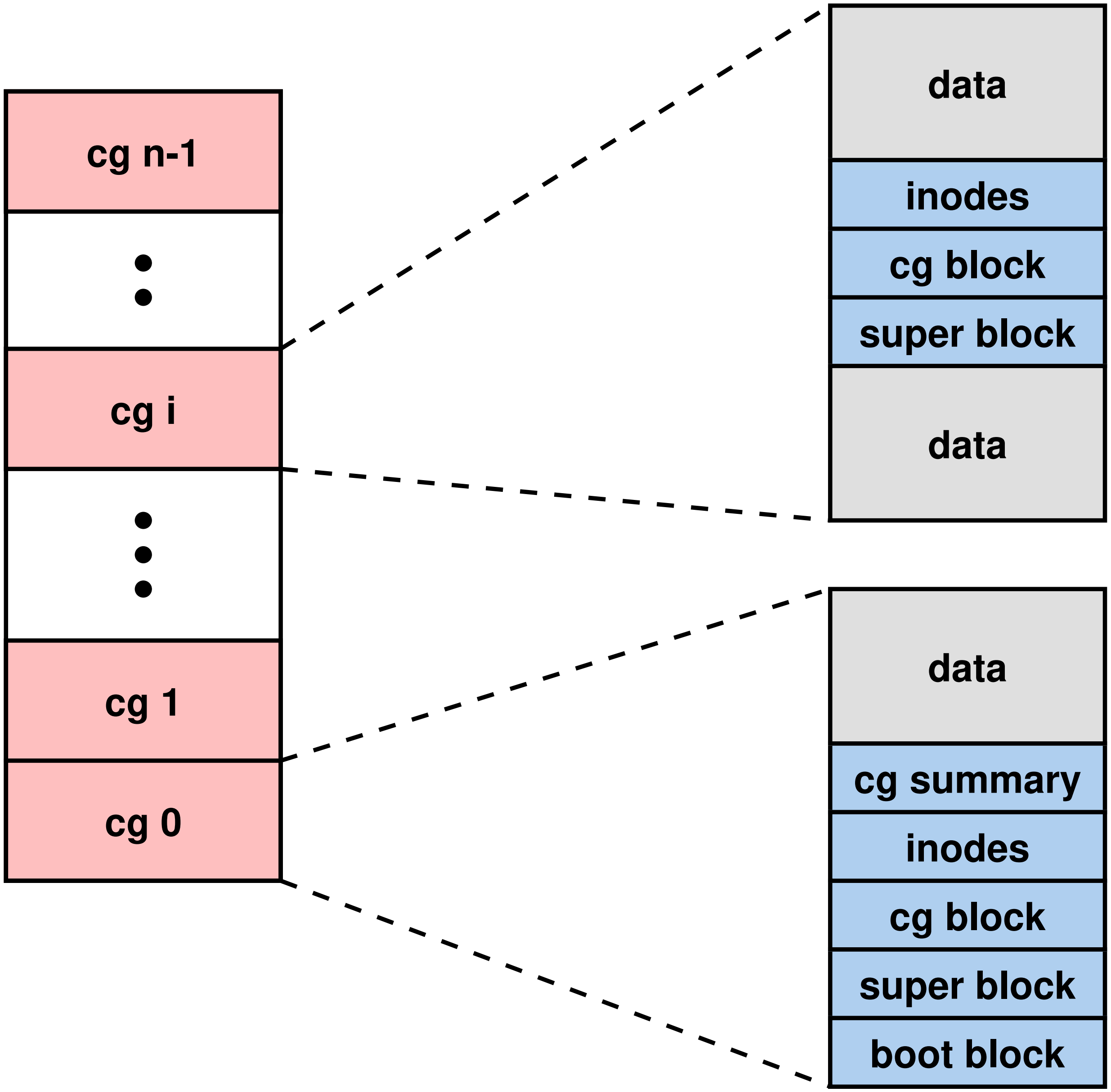
Sprite Picture



S5FS Layouts



FFS Layout


cg n-1
cg i
cg 1
cg 0
data
inodes
cg block
super block
data
data
cg summary
inodes
cg block
super block
boot block

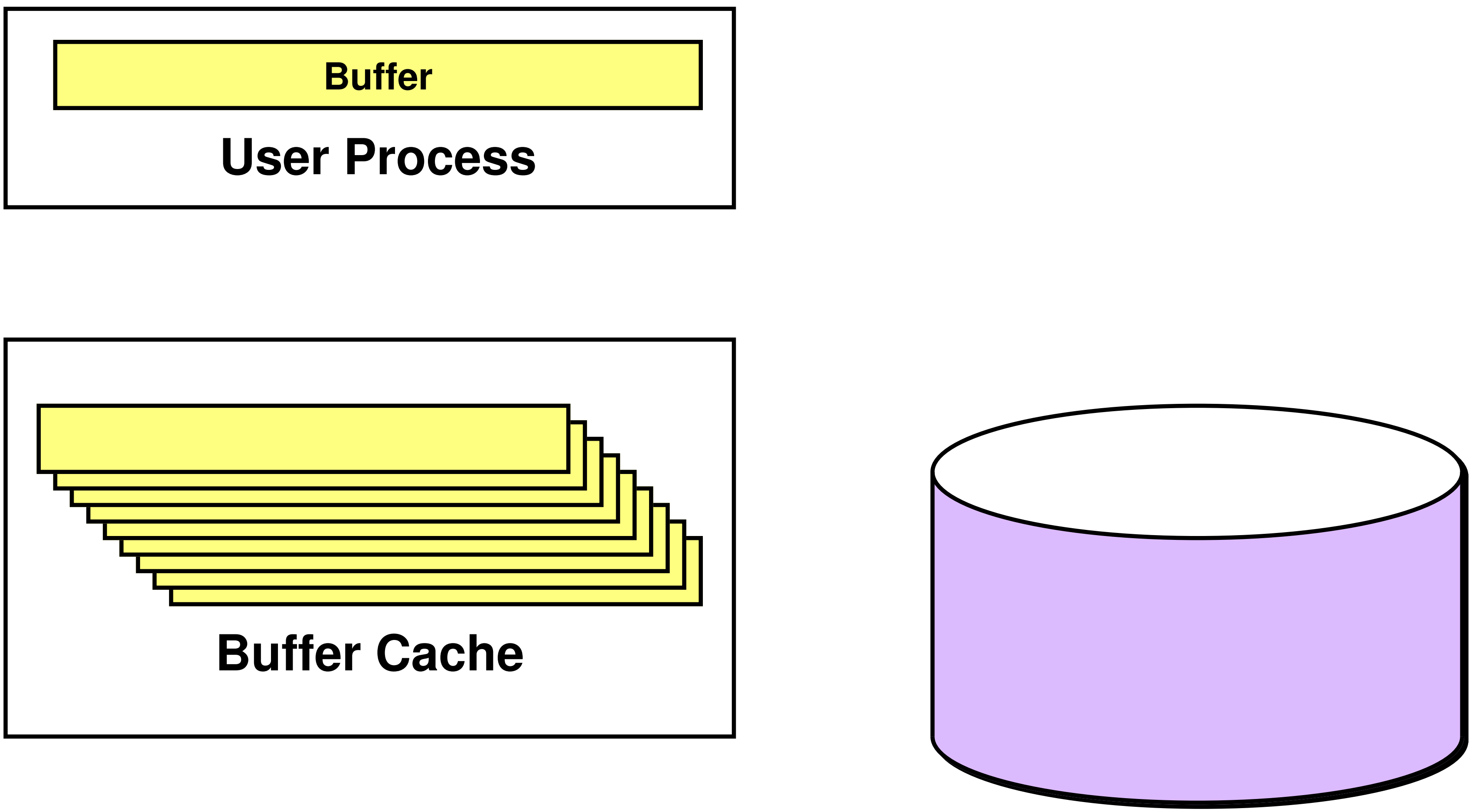
6.1 The Basics of File Systems

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

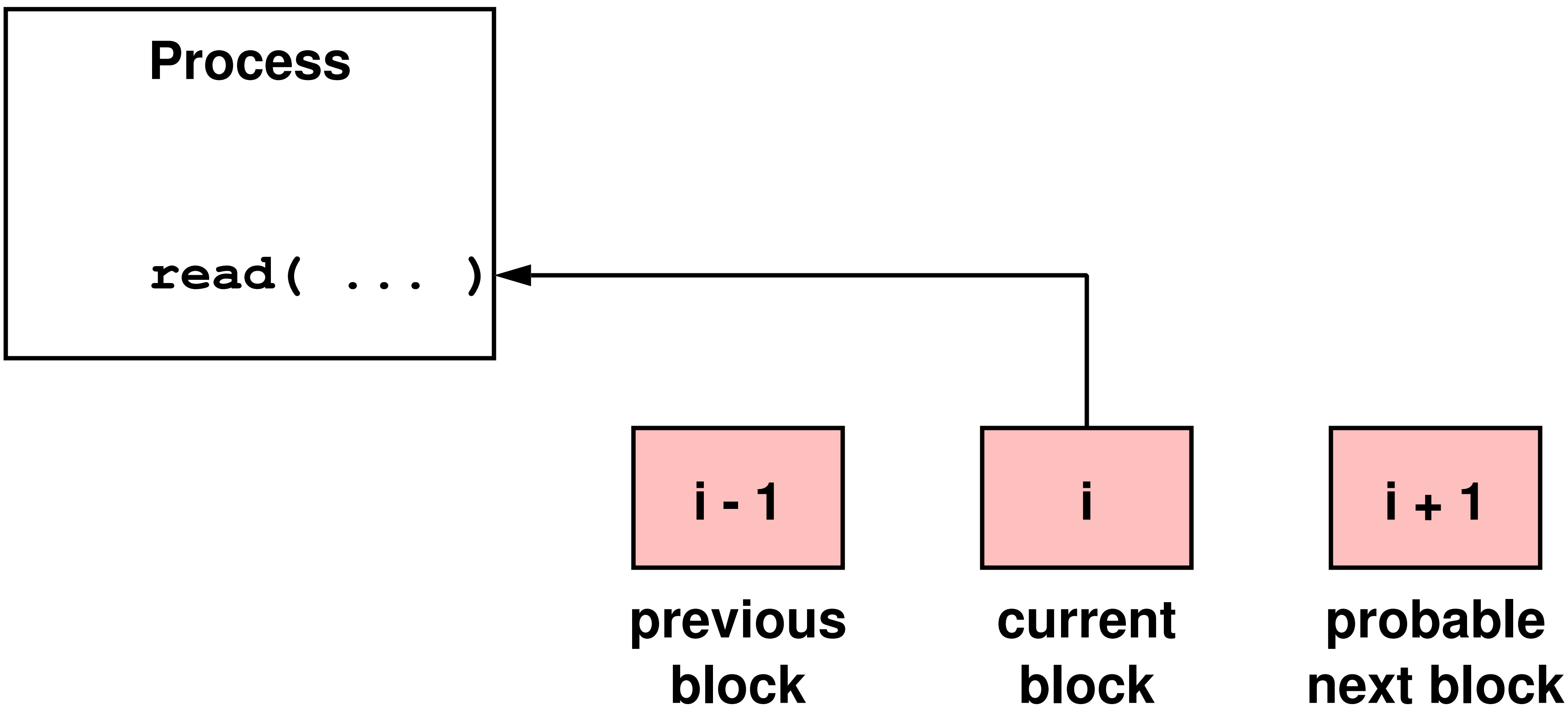
NTFS Master File Table

MFT
MFT Mirror
Log
Volume Info
Attribute Definitions
Root Directory
Free-Space Bitmap
Boot File
Bad-Cluster File
Quota Info
Expansion entries
User File 0
User File 1

The Buffer Cache



Multi-Buffered I/O



Maintaining the Cache