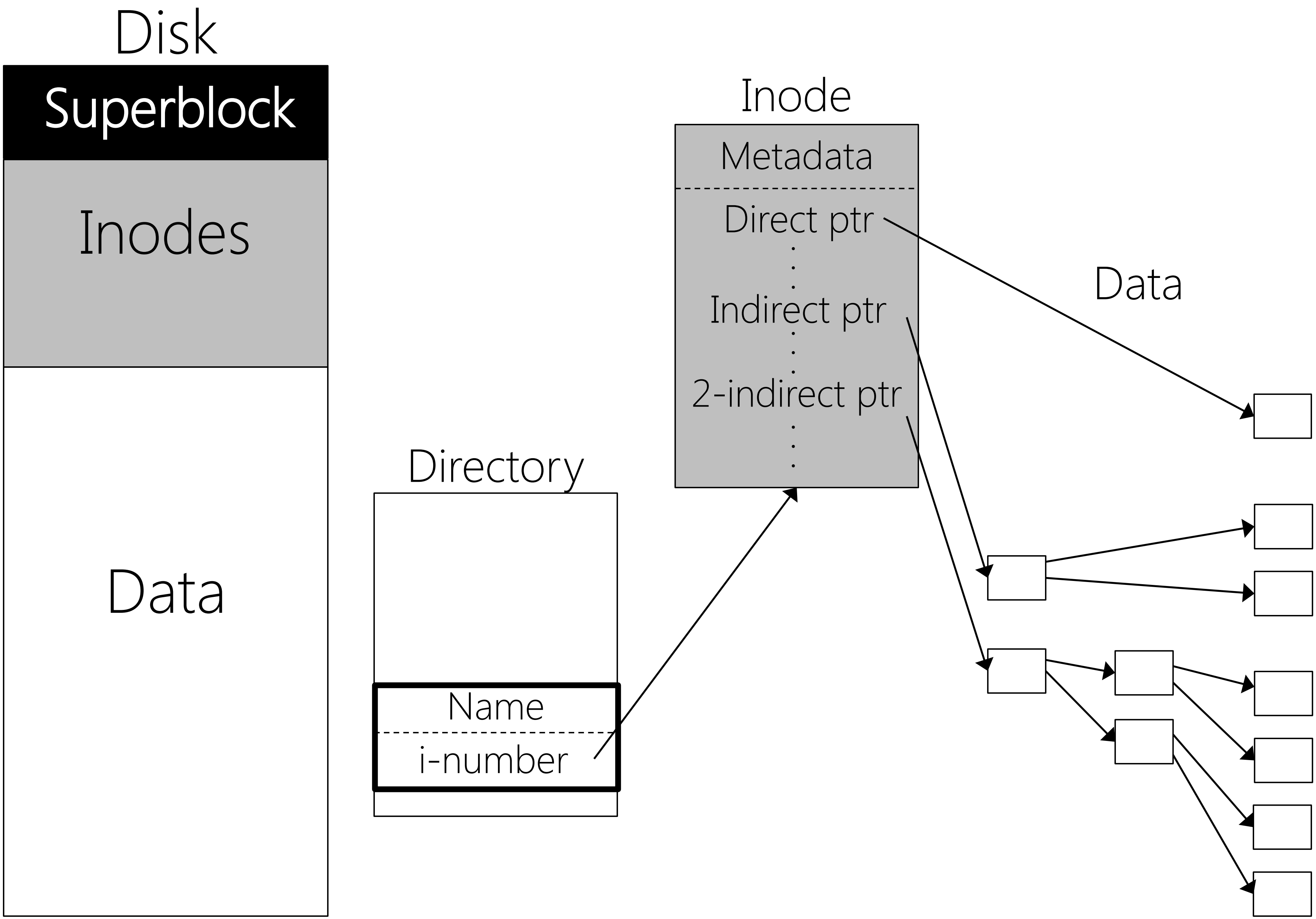
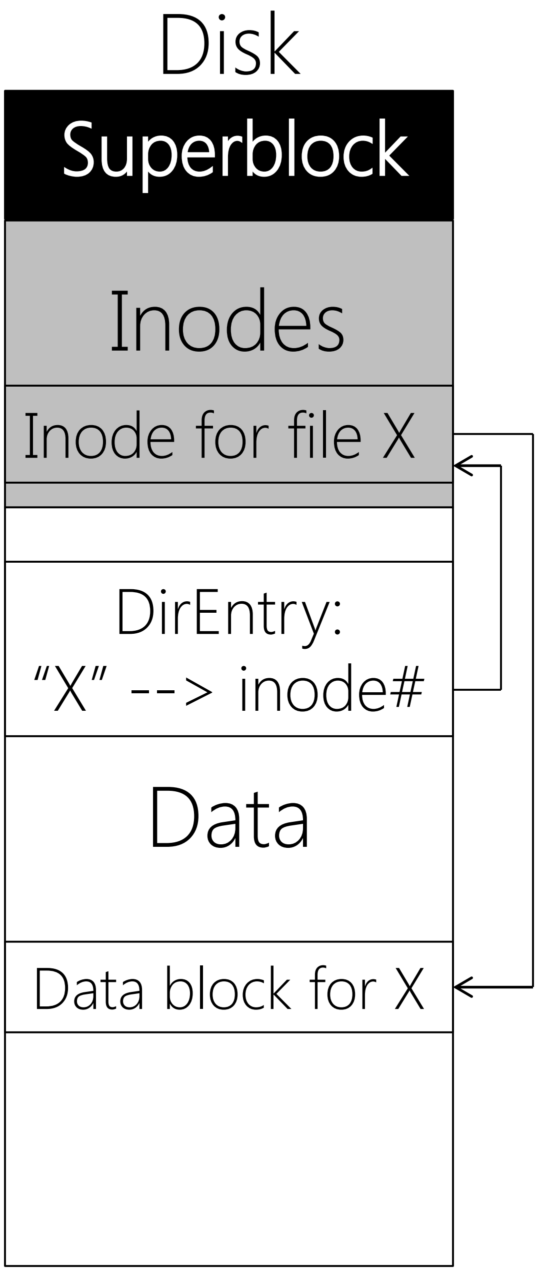
FFS: The Fast File System
-and-
The Magical World of SSDs

The Original, Not-Fast Unix Filesystem



The Original, Not-Fast Unix Filesystem

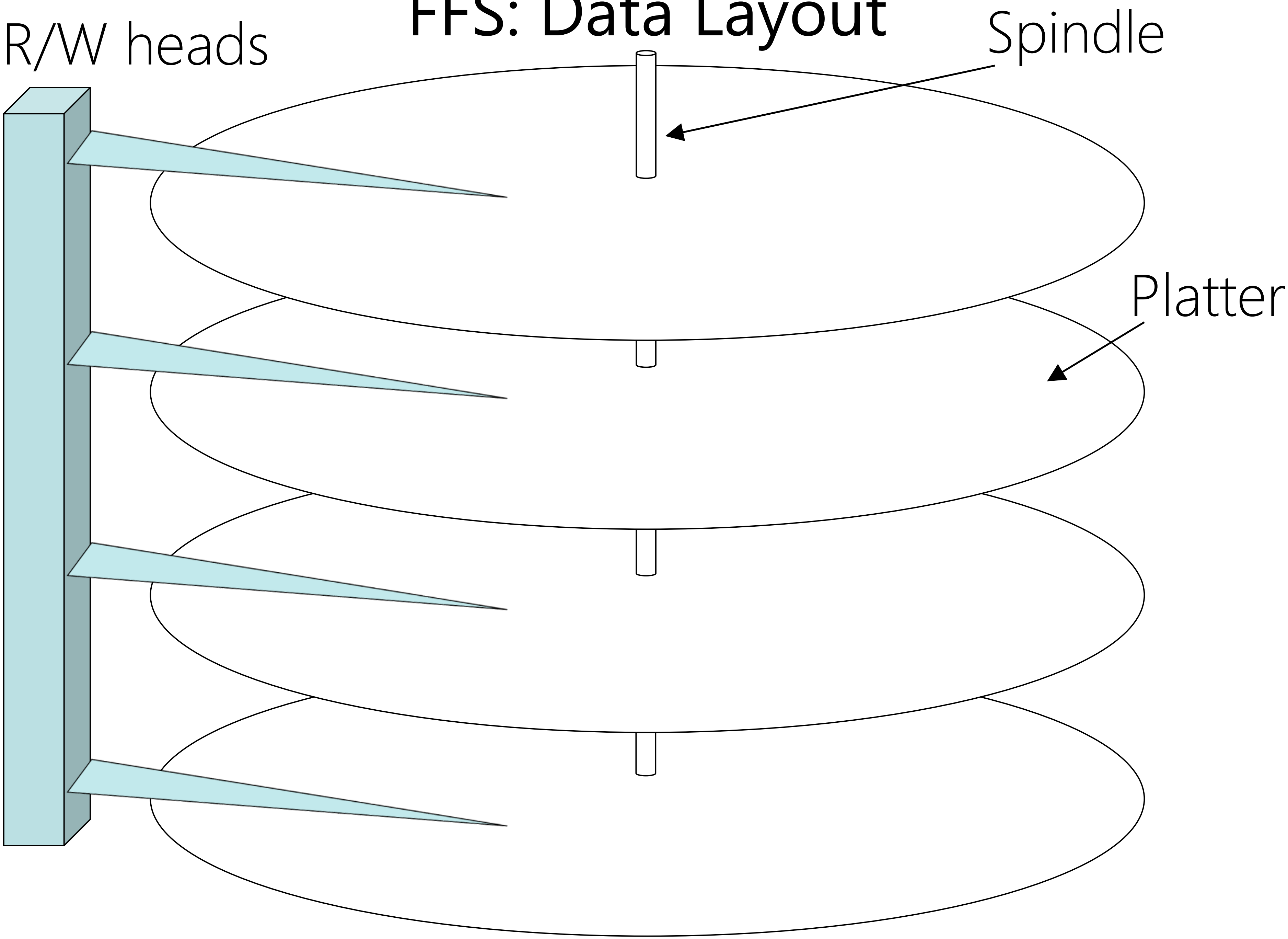


- Design: Disk is treated like a linear array of bytes
- Problem: Data access incurs mechanical delays!
  - Accessing a file's inode and then a data block requires two seeks
  - Block allocation wasn't clever, so files in the same directory were often far apart
  - Block size was 512 bytes, increasing penalty for poor block allocation (more disk seeks!)
- Result: File system only provided 4% of the sequential disk bandwidth!

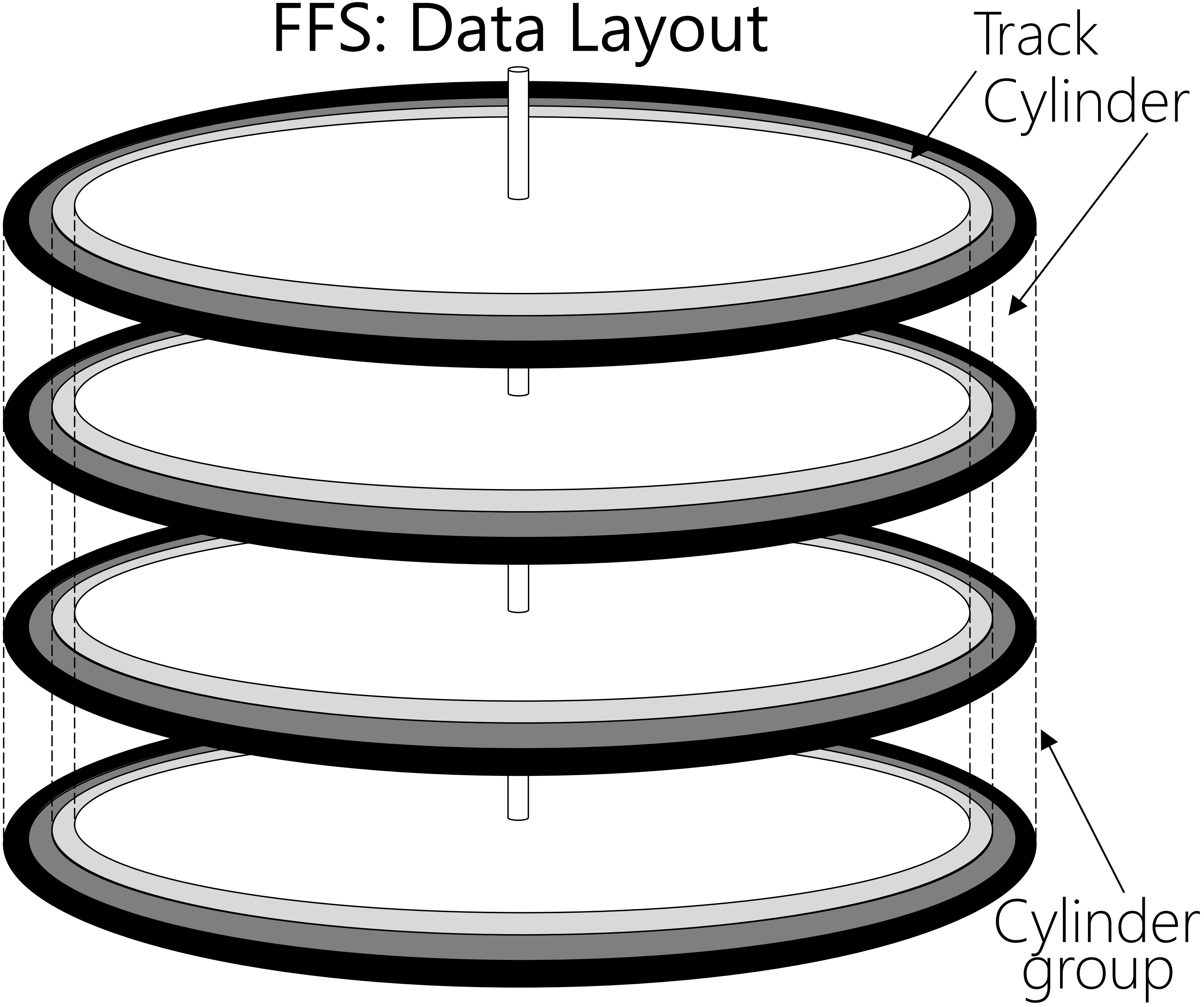
FFS: The Fast File System

- Goal: Keep the same file system abstractions (e.g., open(), read()), but improve the performance of the implementation
- First idea: Increase block size from 512 bytes to 4096 bytes
  - Increases min(bytes_returned_per_seek) --> decreases number of seeks
  - 8x as much data covered by direct blocks --> fewer indirect block accesses --> decreases number of seeks
- Second idea: Disk-aware file layout
  - Consider disk geometry and mechanical delays when deciding where to put files
  - Keep related things next to each other to reduce seeks

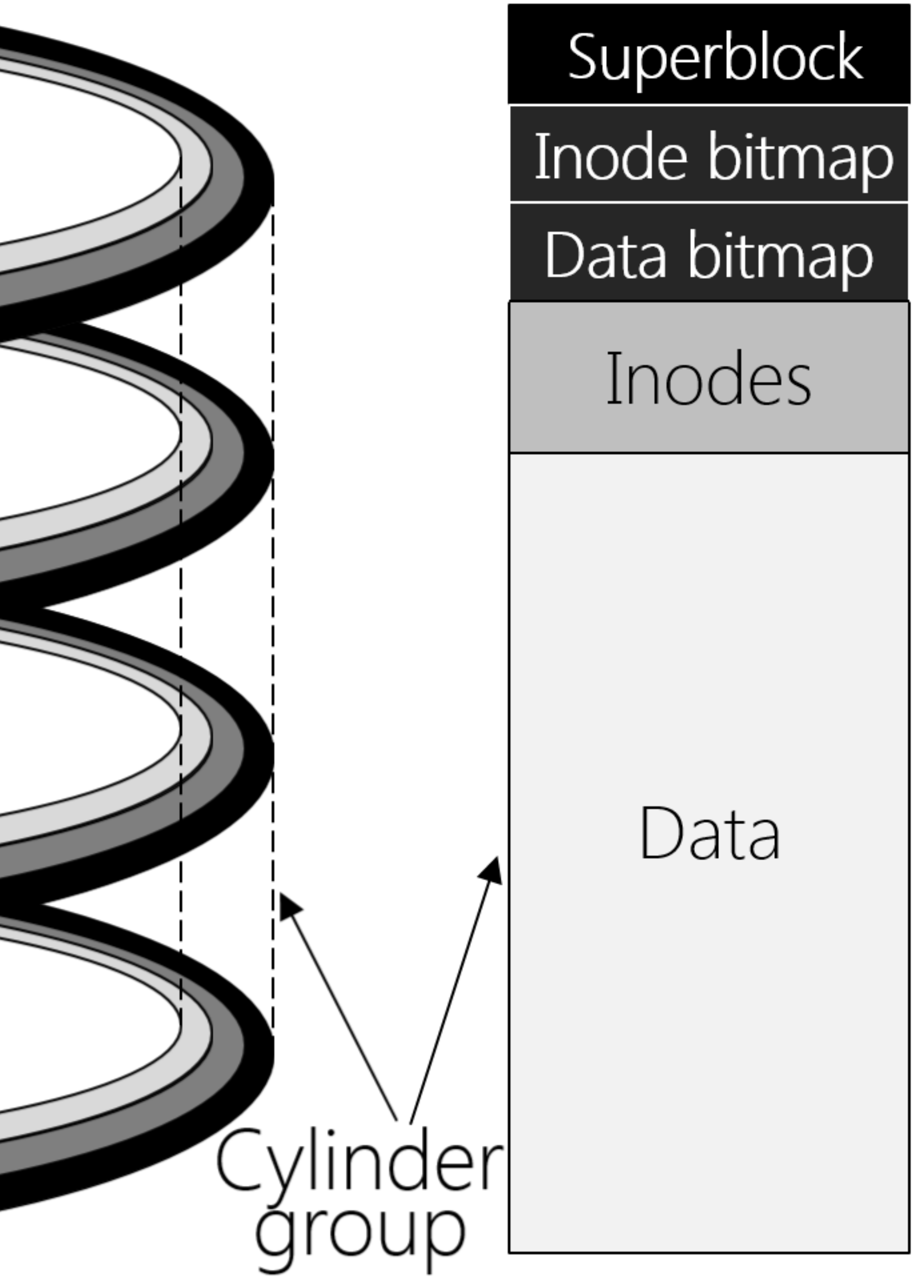
FFS: Data Layout



FFS: Data Layout

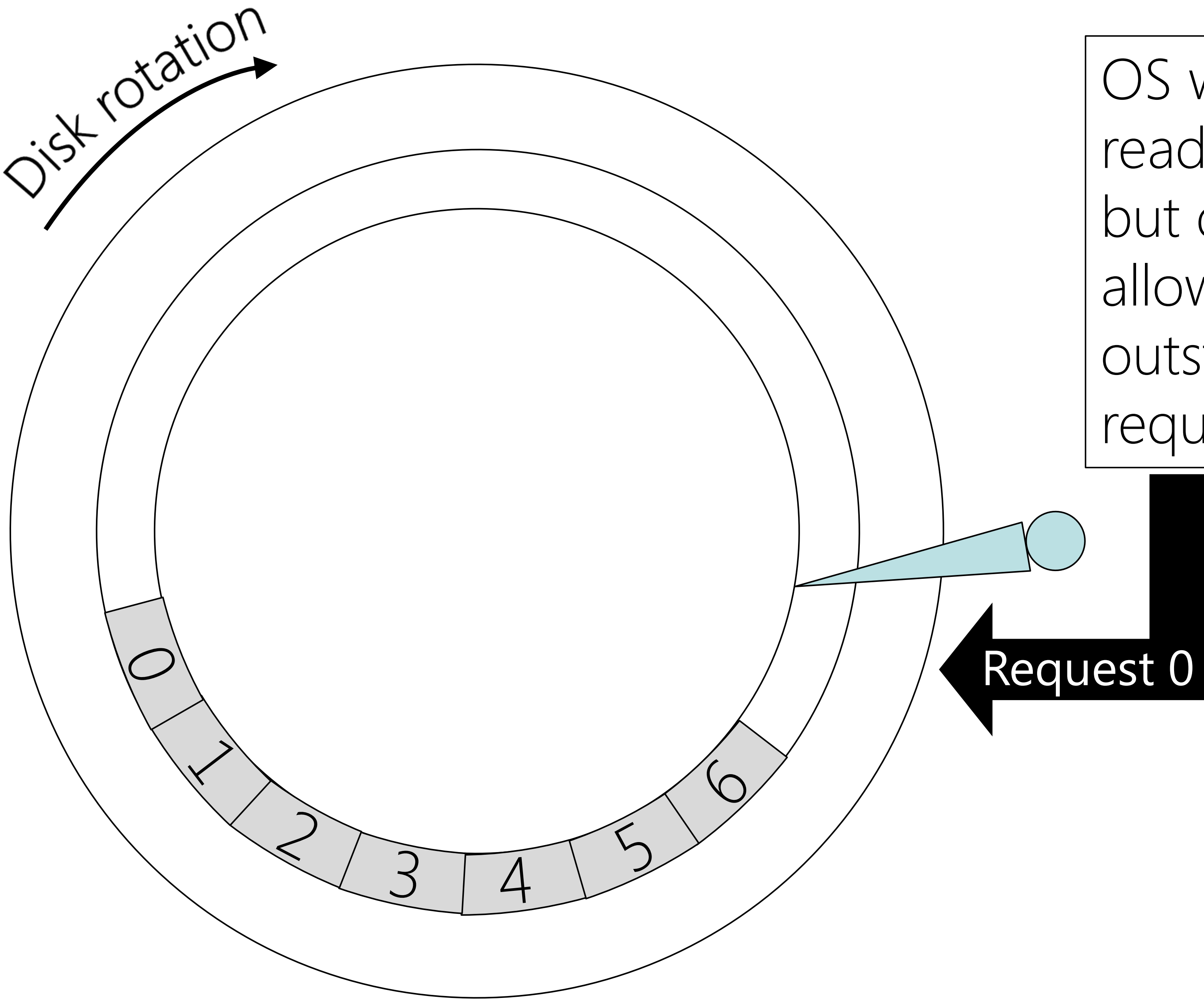


FFS: Data Layout



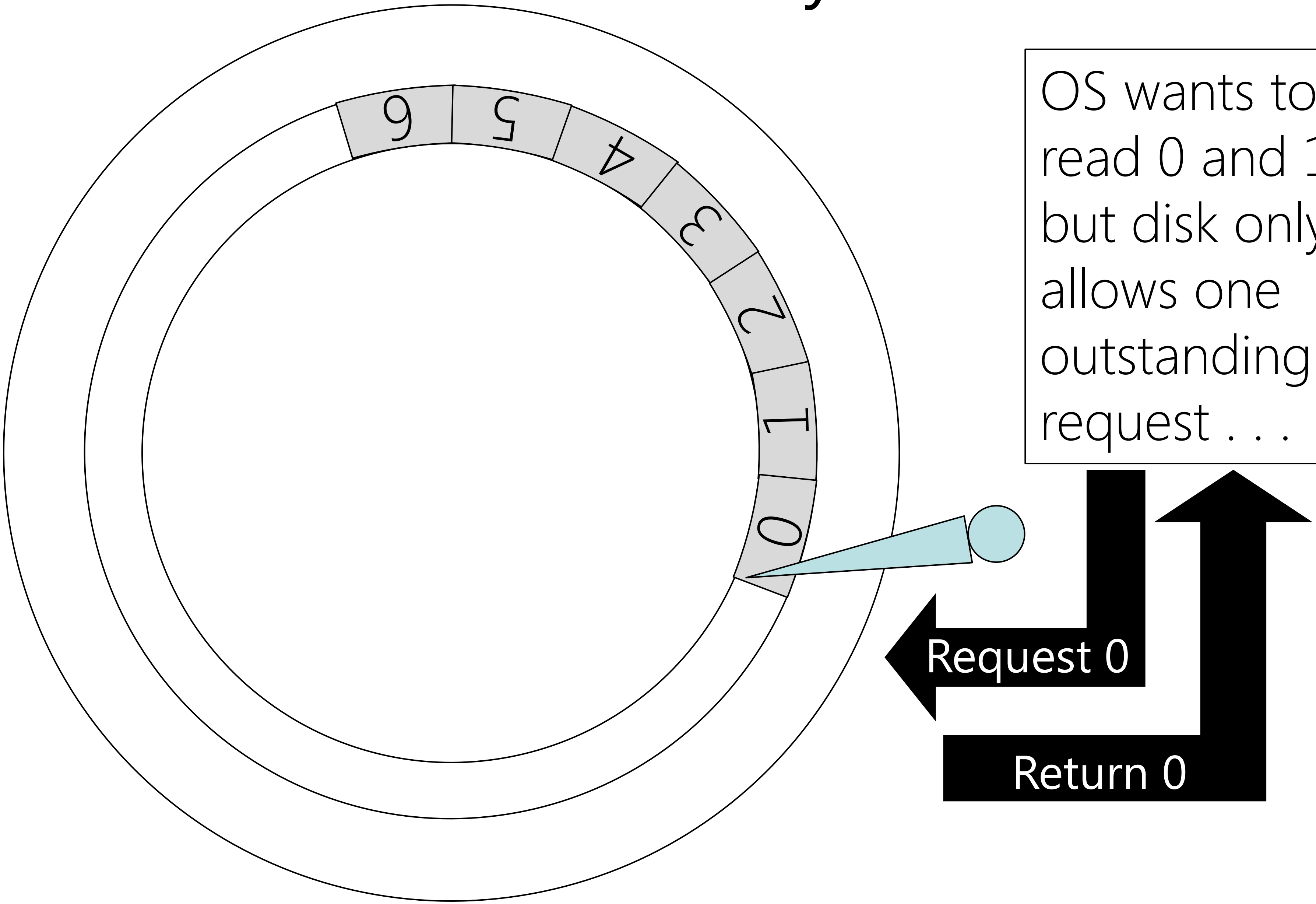
- Directory allocation: Use a cylinder group with few allocated directories and many free inodes
- File allocation: Allocate file inodes in cylinder group of parent directory; allocate file data blocks in cylinder group of file inode
- Allocation policies driven by expectation of temporal locality
    - Files in the same directory will be accessed together (e.g., source code compilation, a browser's web cache)
    - Providing spatial locality for data with temporal locality decreases disk seeks!

FFS: Data Layout



OS wants to read 0 and 1, but disk only allows one outstanding request . . .

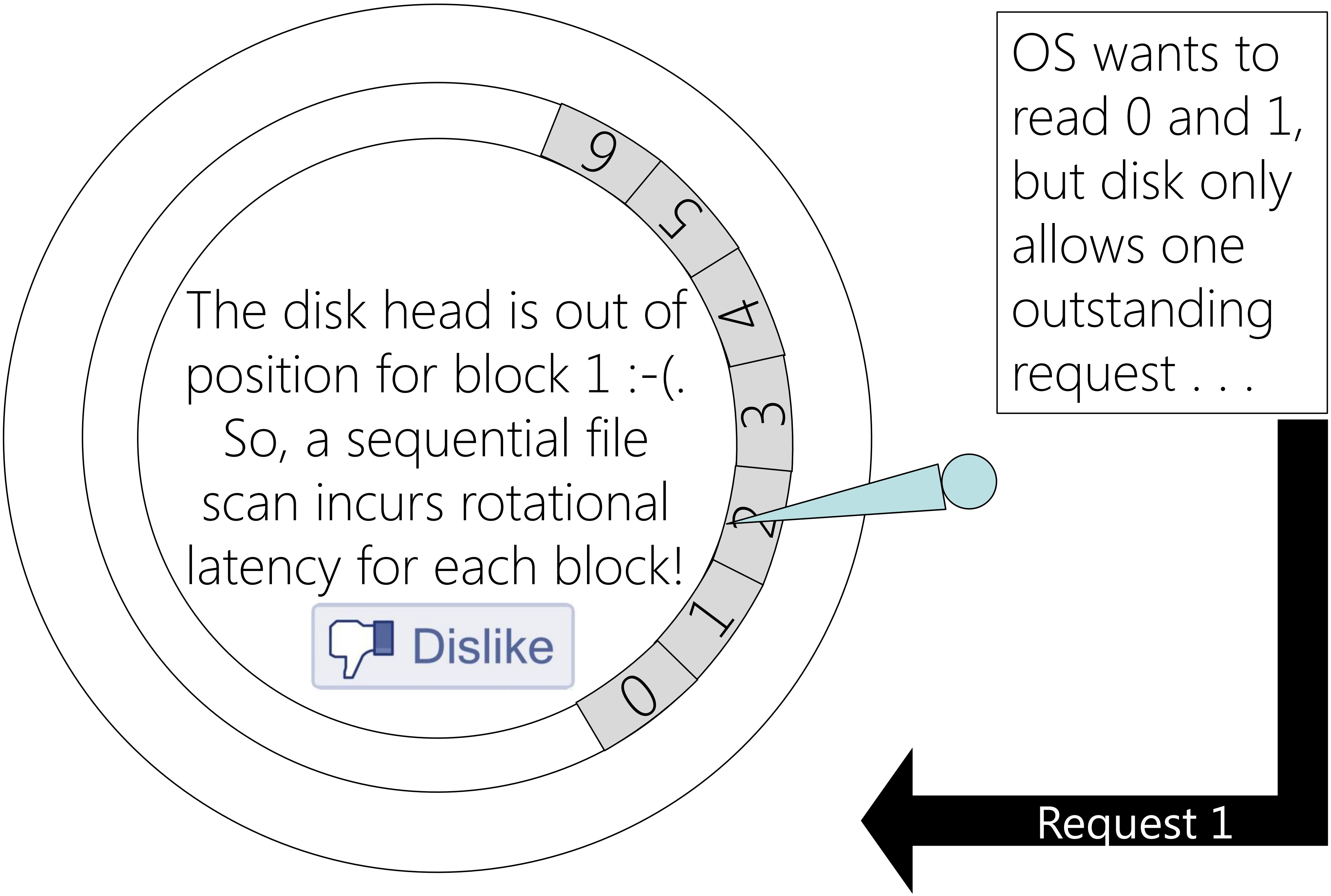
FFS: Data Layout



OS wants to read 0 and 1, but disk only allows one outstanding request . . .

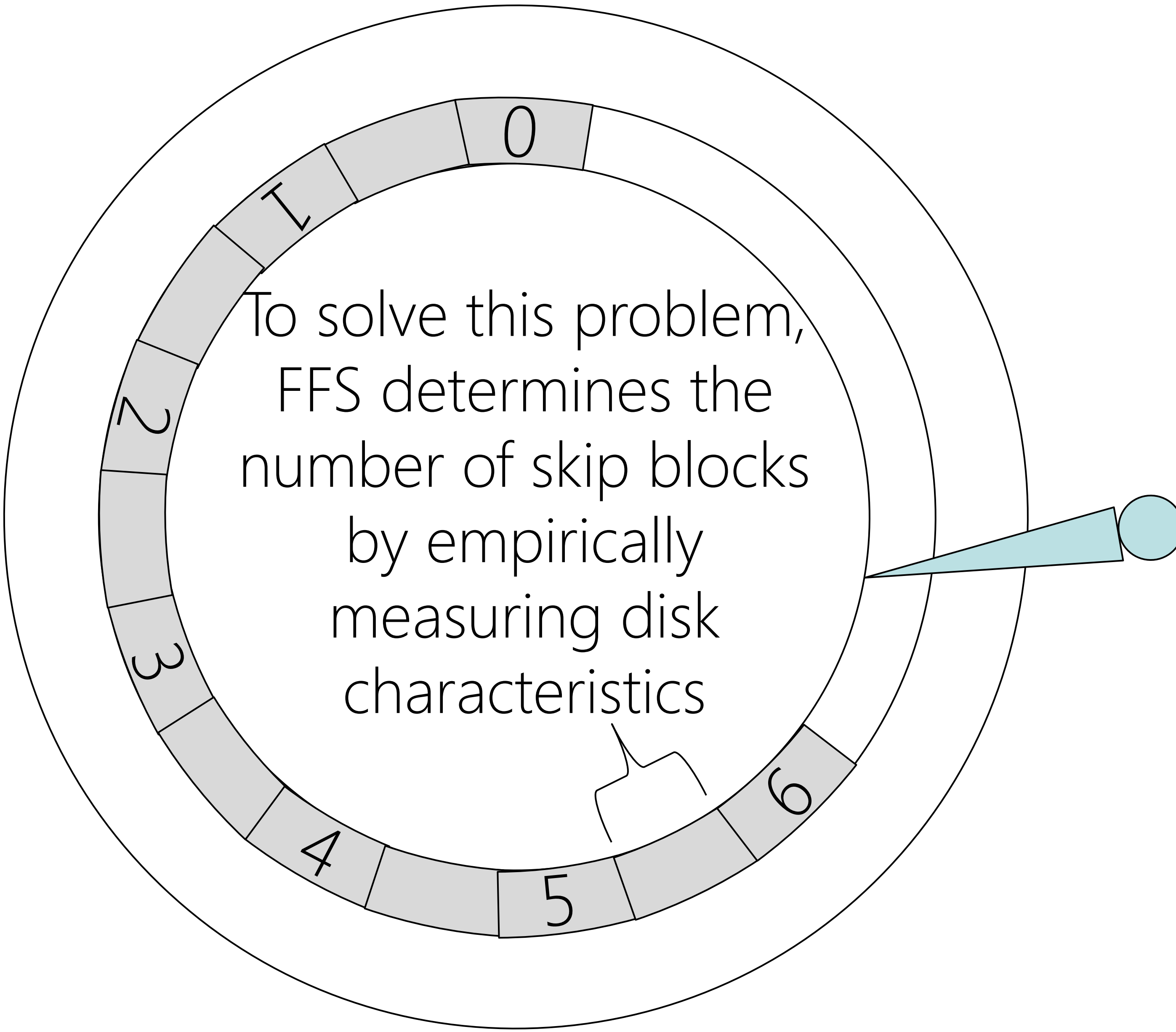
FFS: Data Layout

OS wants to read 0 and 1, but disk only allows one outstanding request . . .



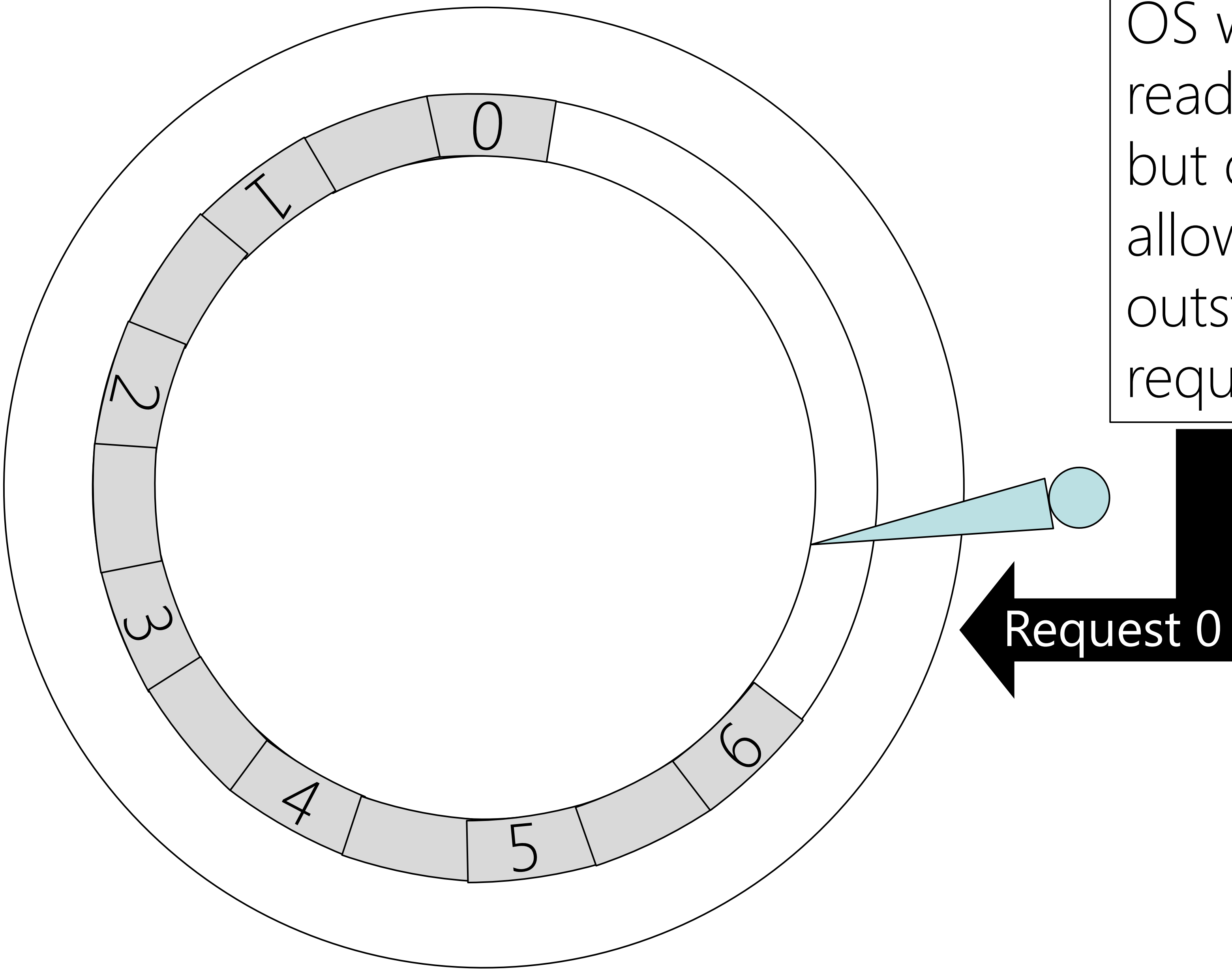
The disk head is out of position for block 1 :-(. So, a sequential file scan incurs rotational latency for each block!

FFS: Data Layout



OS wants to read 0 and 1, but disk only allows one outstanding request . . .

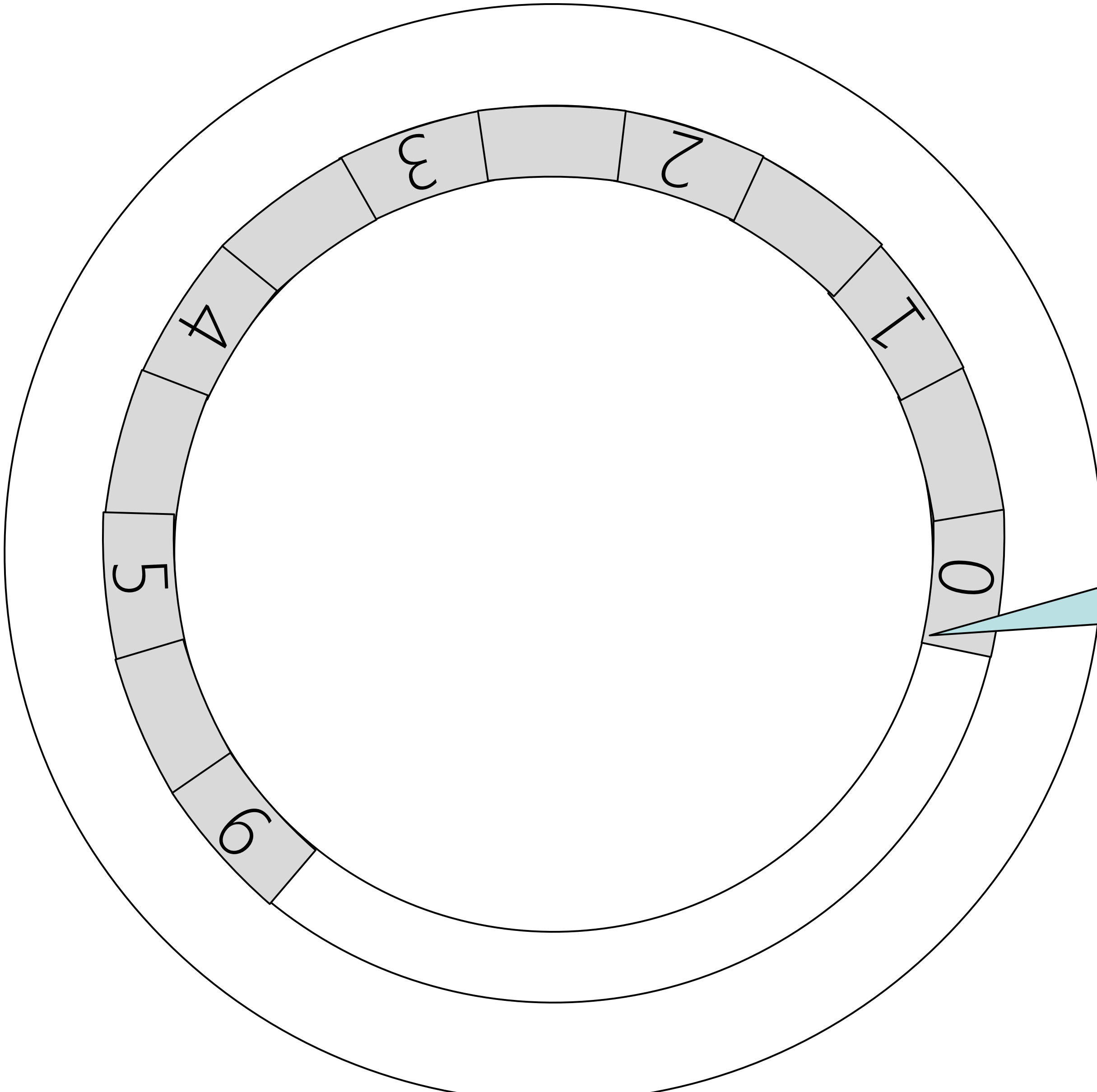
FFS: Data Layout

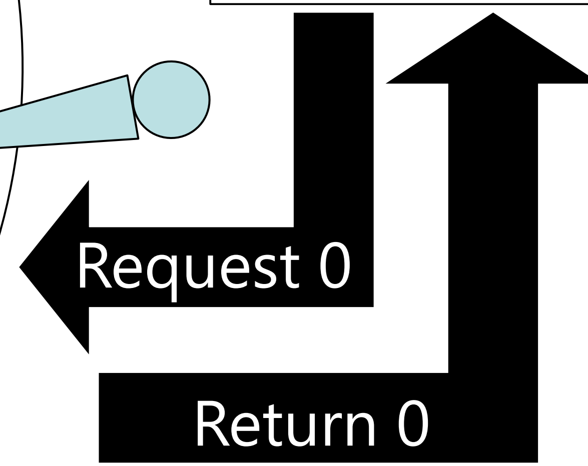


OS wants to read 0 and 1, but disk only allows one outstanding request . . .

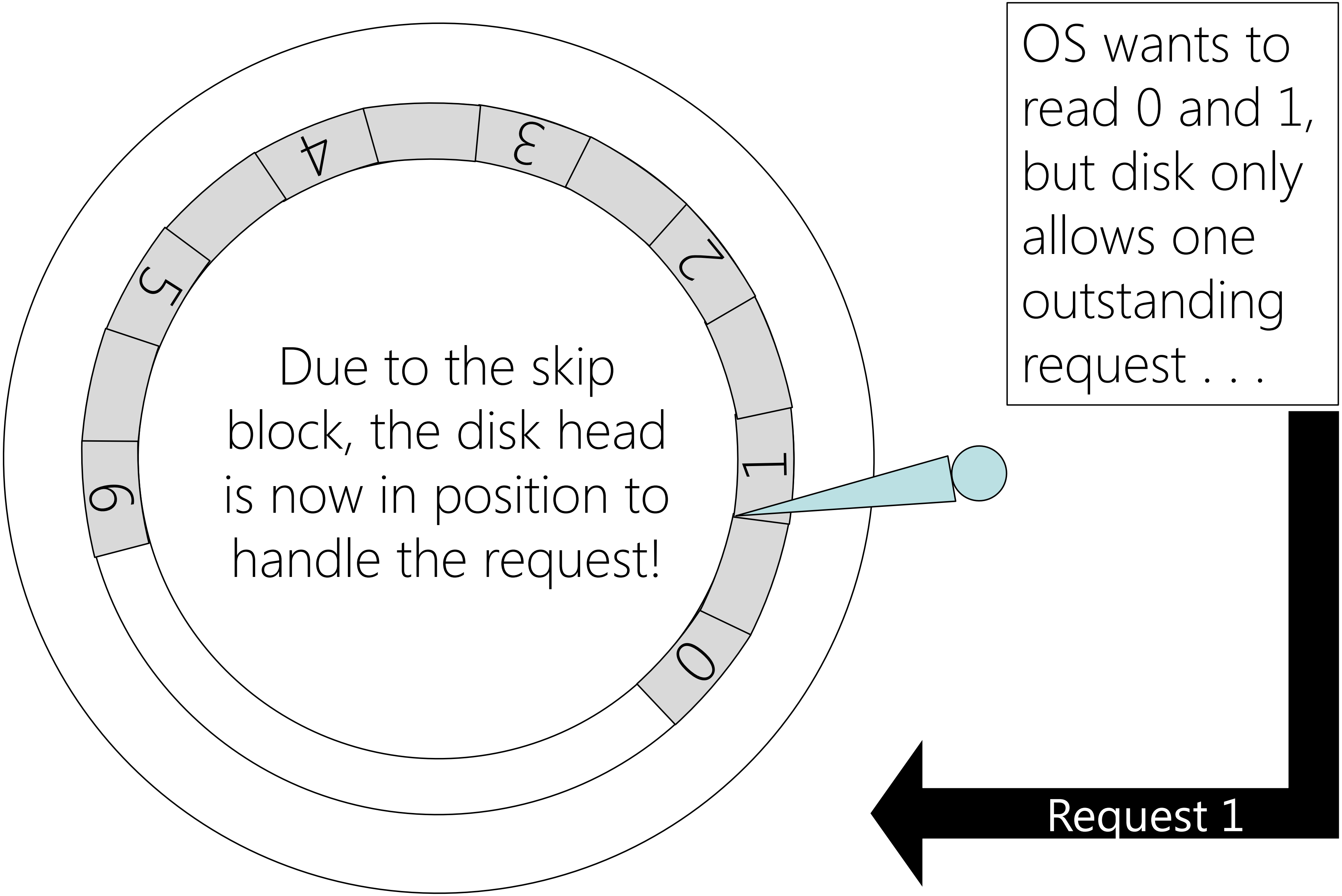
FFS: Data Layout

OS wants to read 0 and 1, but disk only allows one outstanding request . . .



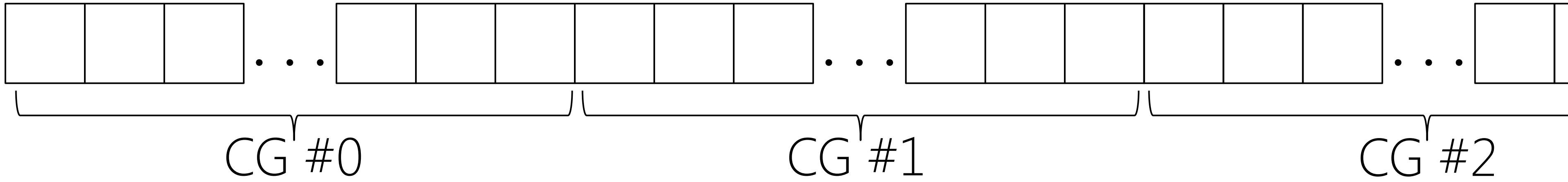


FFS: Data Layout



Block Placement Tricks: Still A Good Idea?

- Modern disks are more powerful than FFS-era disks
  - Use hardware-based track buffer to cache entire track during the read of a single sector
  - Buffer writes, and batch multiple sequential writes into single one
  - Keep a small reserve of “extra” physical sectors so that bad sectors can be avoided (disk implements a virtual-to-physical mapping!)
- Modern disks don’t expose many details about geometry
  - Only guarantee that sectors with similar sector numbers are probably “close” to each other w.r.t. access time
  - So, modern file systems use “block groups” instead of “cylinder groups”

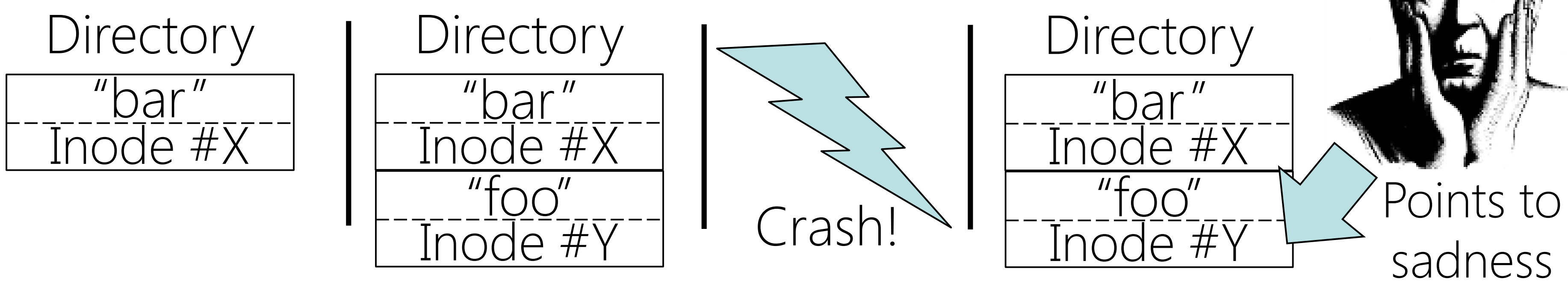


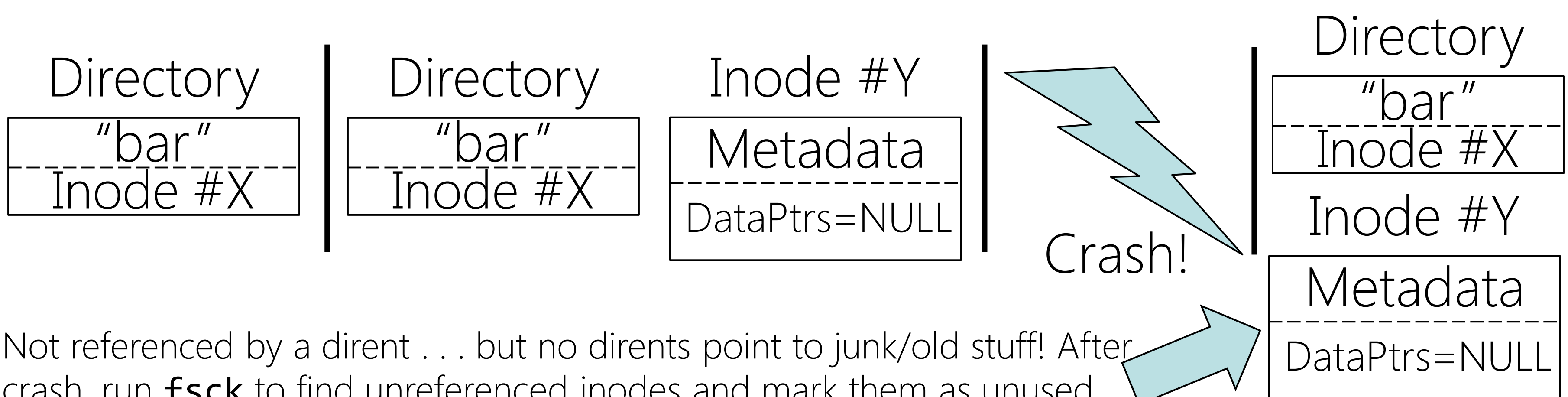
Ensuring Consistency After Crashes

- Q: What happens to on-disk structures after an OS crash, a hard reboot, or a power outage?
- A: What would Gallant do? He would ensure that the file system recovers to a reasonable state.
  - Some data loss is usually ok . . .
  - . . . but it's NOT ok to have an unmountable file system!
  - There's a trade-off between performance and data loss

Crash Consistency: Creating a New File

- To create a new file "foo", you need to:
  1. Update inode bitmap to allocate a new inode
  2. Write the new inode for "foo" to disk
  3. Write an updated version of the directory that points to the new inode
- The order of the writes makes a difference! Suppose (1) has completed . . . how should we order (2) and (3)?





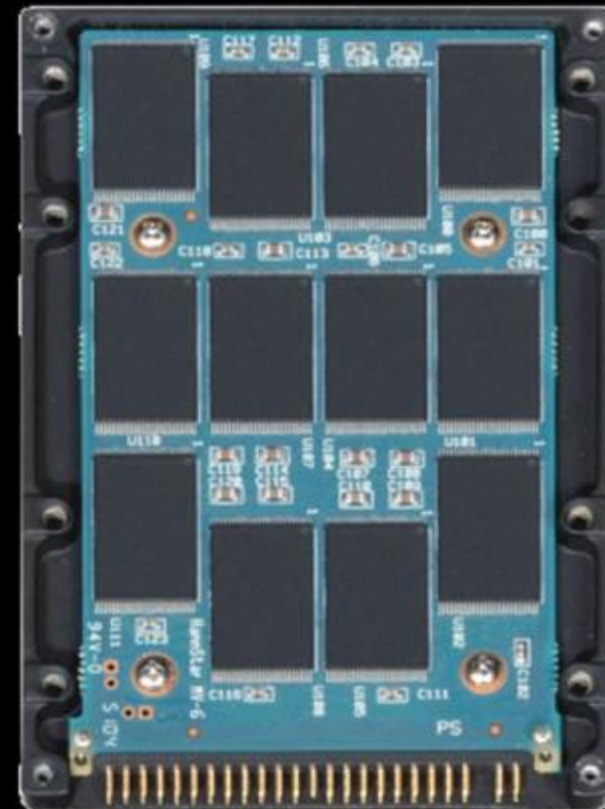
Not referenced by a dirent . . . but no dirents point to junk/old stuff! After crash, run **`fsck`** to find unreferenced inodes and mark them as unused.

Crash Consistency Using Synchronous Writes

- For a file system operation that requires multiple ordered writes, wait for each write to hit the disk before issuing the next one
  - Ex: On file create(), issue write to the inode, wait for it to complete, then issue write to the directory
- Good: File system will be left in a consistent state after crash
- Bad: fsck is slow (it has to make multiple passes over metadata)
- Bad: Synchronous writes make the file system slow
  - We'd like to be able to issue IOs immediately, and have multiple IOs in-flight at any given time: provides the disk with maximum ability to reorder writes for performance
  - However, reordering for performance may violate the desired consistency semantics

MARGO WILL SLAY THE CONSISTENCY DRAGON

VS



HDD

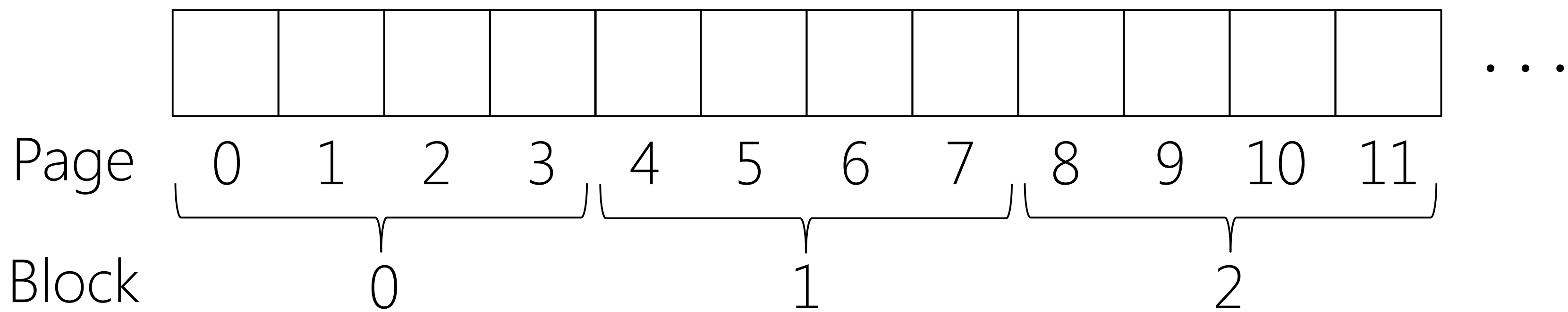
SSD

Solid-state Storage Devices (SSDs)

- Unlike hard drives, SSDs have no mechanical parts
  - SSDs use transistors (just like DRAM), but SSD data persists when the power goes out
  - NAND-based flash is the most popular technology, so we'll focus on it
- High-level takeaways
  1. SSDs have a higher $/bit than hard drives, but better performance (no mechanical delays!)
  2. SSDs handle writes in a strange way; this has implications for file system design

Solid-state Storage Devices (SSDs)

- An SSD contains blocks made of pages
  - A page is a few KB in size (e.g., 4 KB)
  - A block contains several pages, is usually 128 KB or 256 KB

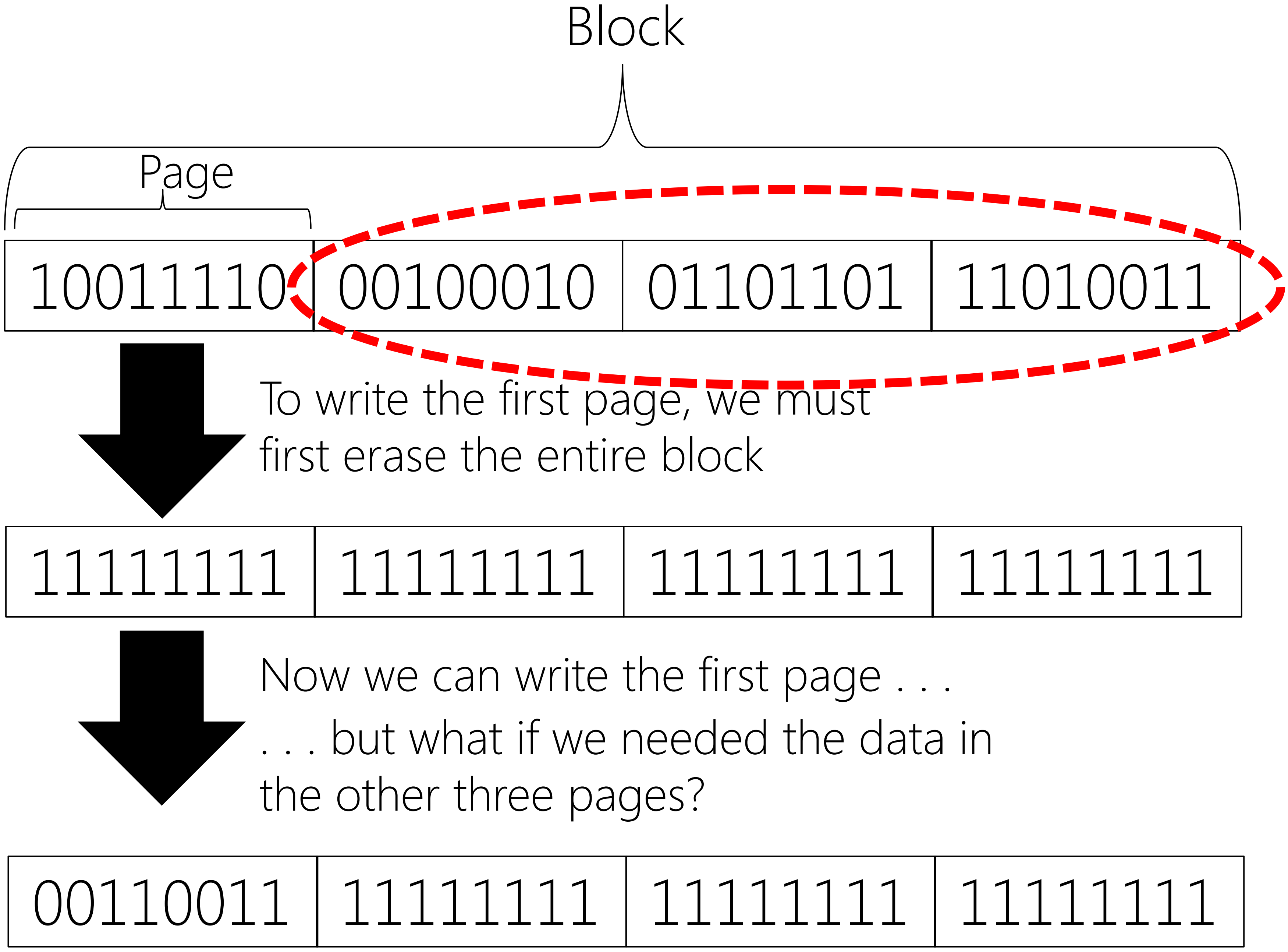


- To write a single page, YOU MUST ERASE THE ENTIRE BLOCK FIRST
- A block is likely to fail after a certain number of erases (~1000 for slowest-but-highest-density flash, ~100,000 for fastest-but-lowest-density flash)

SSD Operations (Latency)

- Read a page: Retrieve contents of entire page (e.g., 4 KB)
  - Cost is 25—75 microseconds
  - Cost is independent of page number, prior request offsets
- Erase a block: Resets each page in the block to all 1s
  - Cost is 1.5—4.5 milliseconds
  - Much more expensive than reading!
  - Allows each page to be written
- Program (i.e., write) a page: Change selected 1s to 0s
  - Cost is 200—1400 microseconds
  - Faster than erasing a block, but slower than reading a page

Hard disk: 4—10ms avg. seek latency
2—7ms avg. rotational latency


Block
Page
10011110
00100010
01101101
11010011
To write the first page, we must
first erase the entire block
11111111
11111111
11111111
11111111
Now we can write the first page . . .
. . . but what if we needed the data in
the other three pages?
00110011
11111111
11111111
11111111

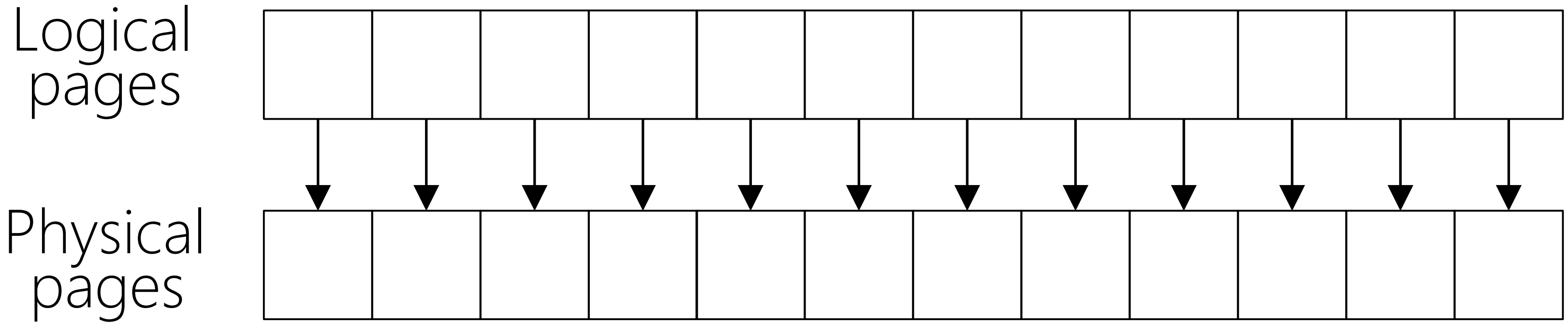
Flash Translation Layer (FTL)

- Goal 1: Translate reads/writes to logical blocks into reads/erases/programs on physical pages+blocks
  - Allows SSDs to export the simple “block interface” that hard disks have traditionally exported
  - Hides write-induced copying and garbage collection from applications
- Goal 2: Reduce write amplification (i.e., the amount of extra copying needed to deal with block-level erases)
- Goal 3: Implement wear leveling (i.e., distribute writes equally to all blocks, to avoid fast failures of a “hot” block)

- FTL is typically implemented in hardware in the SSD, but is implemented in software for some SSDs

FTL Approach #1: Direct Mapping

- Have a 1-1 correspondence between logical pages and physical pages



- Reading a page is straightforward
- Writing a page is trickier:
  - Read the entire physical block into memory
  - Update the relevant page in the in-memory block
  - Erase the entire physical block
  - Program the entire physical block using the new block value

Sadness #1: Write amplification

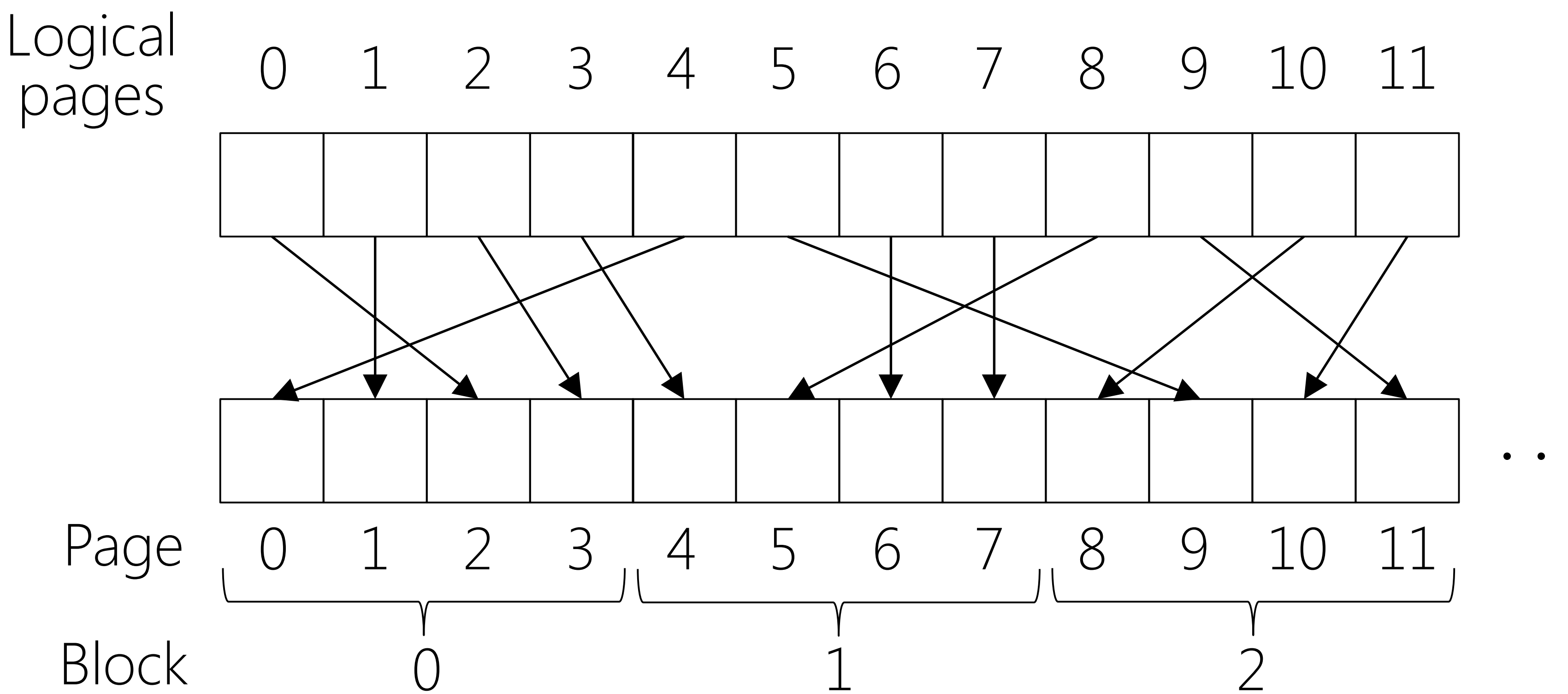
- Writing a single page requires reading and writing an entire block

Sadness #2: Poor reliability

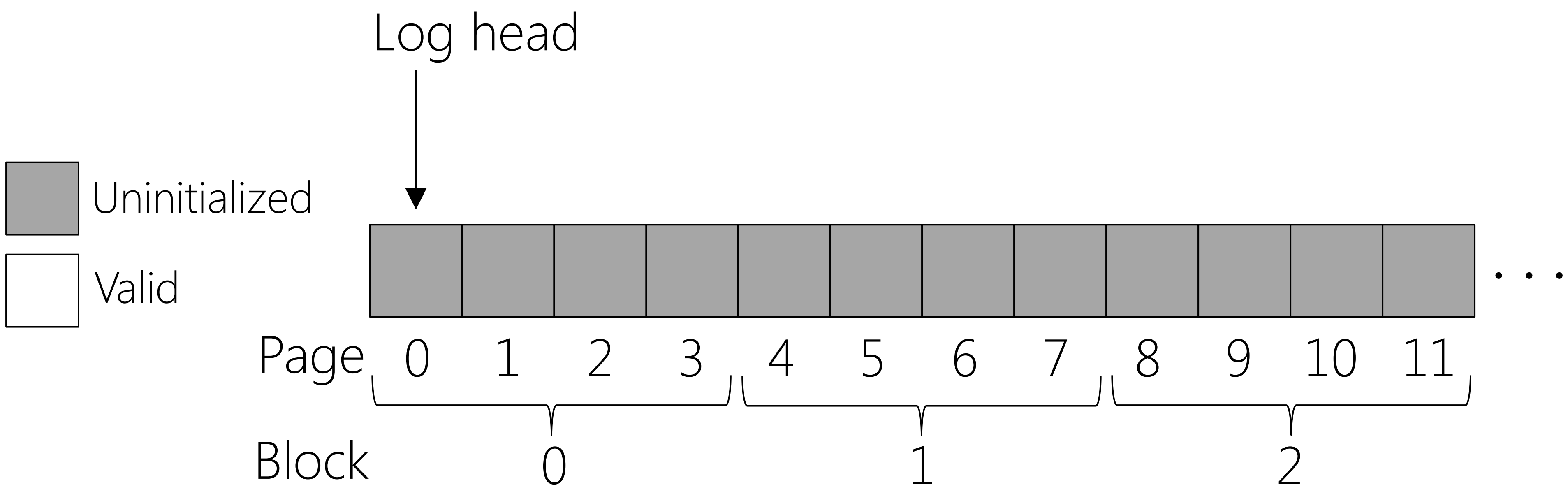
- If the same logical block is repeatedly written, its physical block will quickly fail
- Particularly unfortunate for logical metadata blocks

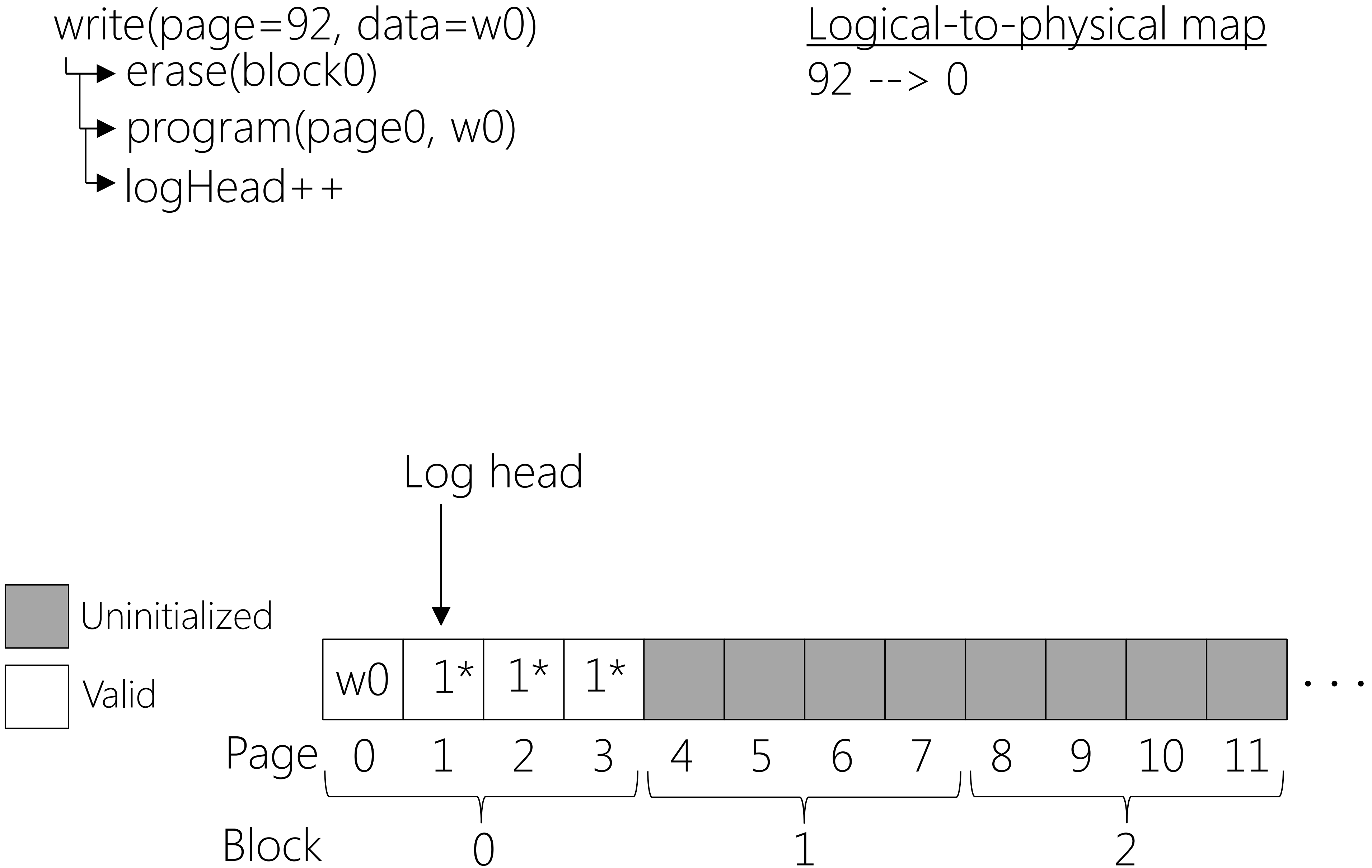
FTL Approach #2: Log-based mapping

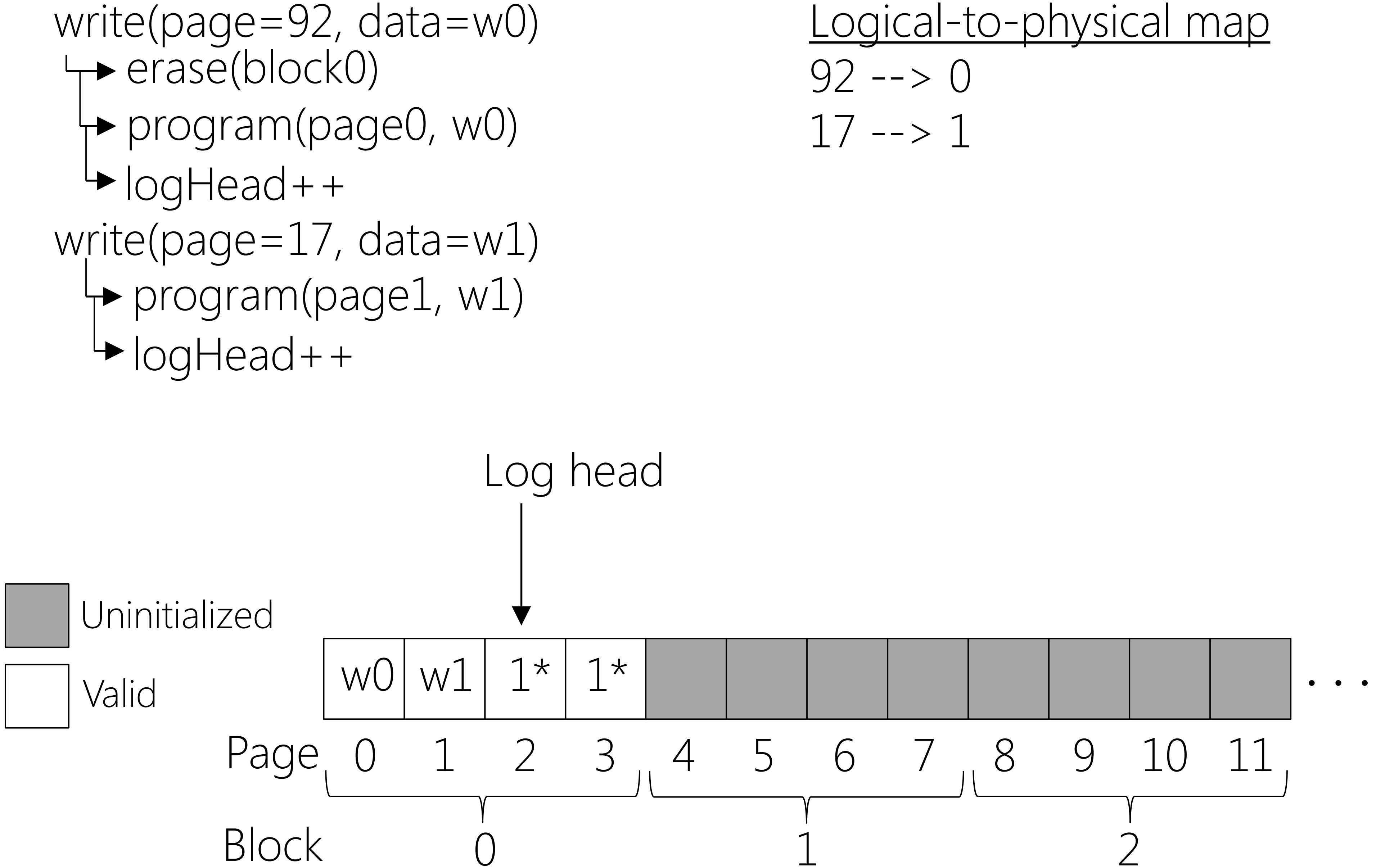
- Basic idea: Treat the physical blocks like a log
  - Send data in each page-to-write to the end of the log
  - Maintain a mapping between logical pages and the corresponding physical pages in the SSD



Logical-to-physical map




write(page=92, data=w0)
erase(block0)
program(page0, w0)
logHead++
Logical-to-physical map
92 --> 0
Log head
Uninitialized
Valid
w0
1*
1*
1*
Page 0 1 2 3 4 5 6 7 8 9 10 11
Block 0 1 2


write(page=92, data=w0)
erase(block0)
program(page0, w0)
logHead++
write(page=17, data=w1)
program(page1, w1)
logHead++
Logical-to-physical map
92 --> 0
17 --> 1
Log head
Uninitialized
Valid
w0
w1
1*
1*
Page
0
1
2
3
4
5
6
7
8
9
10
11
Block
0
1
2

write(page=92, data=w0)
- erase(block0)
- program(page0, w0)
- logHead++

write(page=17, data=w1)
- program(page1, w1)
- logHead++

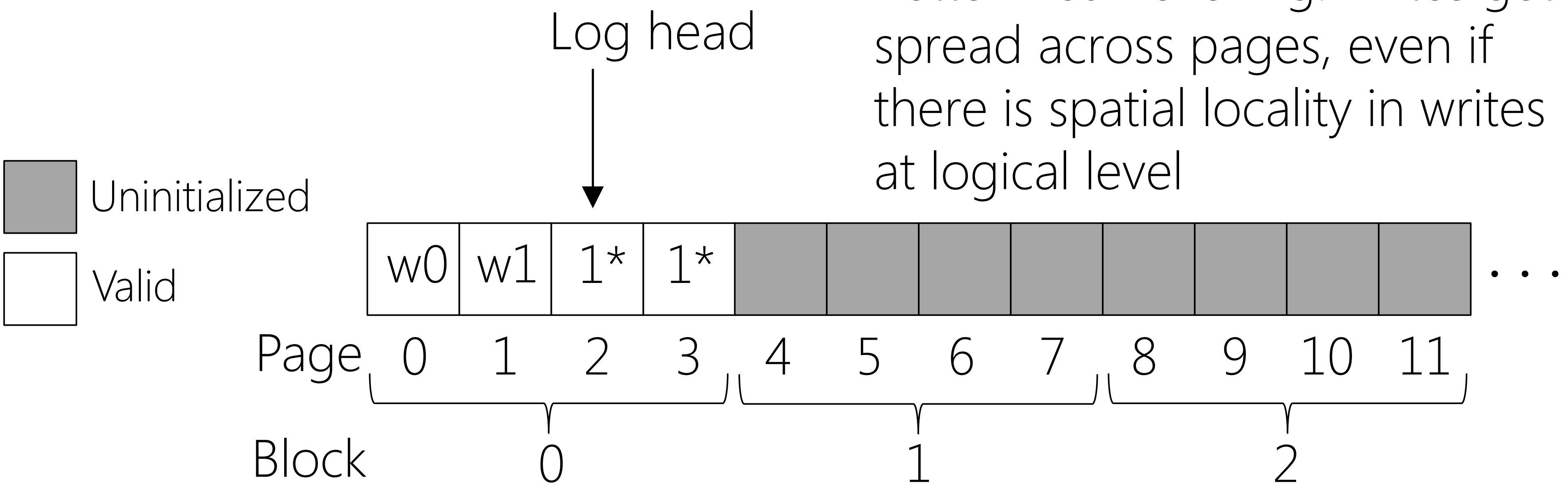
Logical-to-physical map

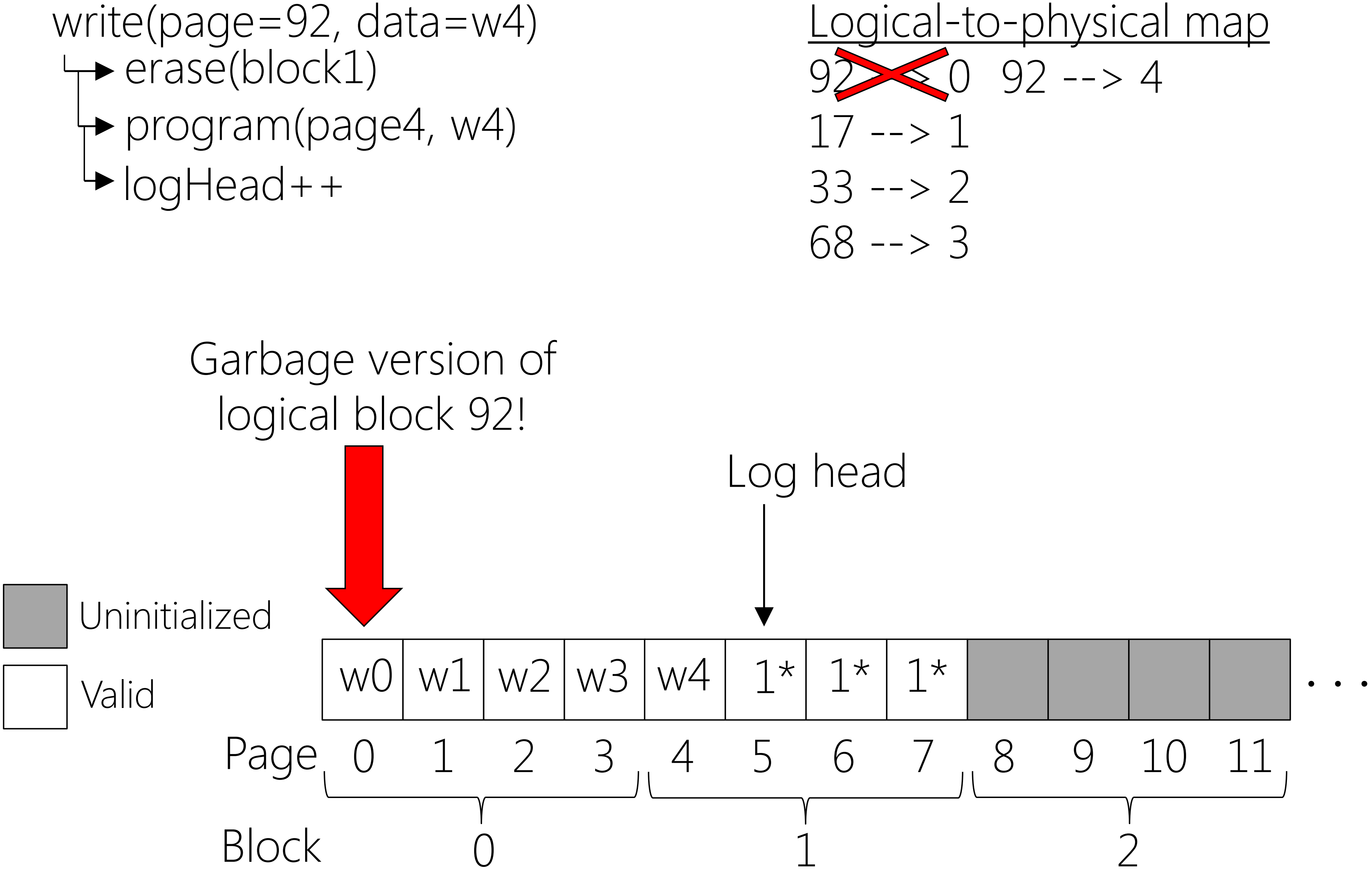
92 --> 0

17 --> 1

Advantages w.r.t. direct mapping

- Avoids expensive read-modify-write behavior
- Better wear levelling: writes get spread across pages, even if there is spatial locality in writes at logical level





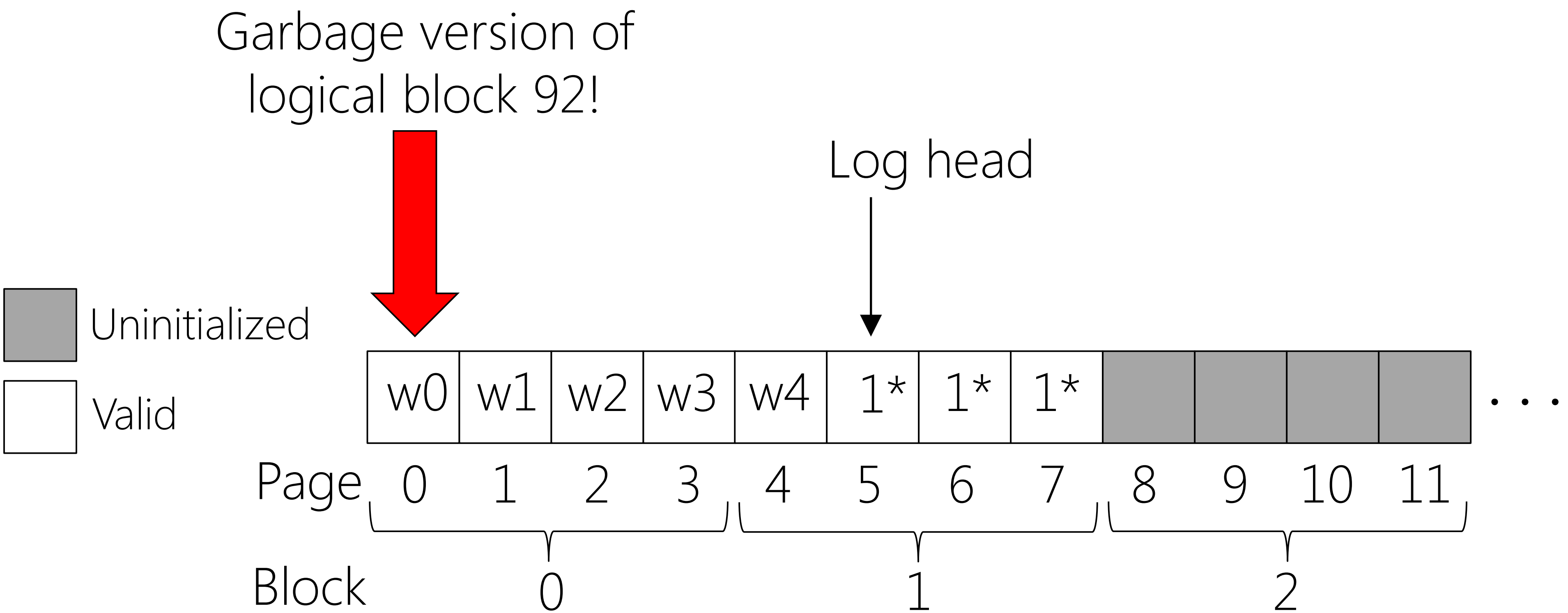
write(page=92, data=w4)
erase(block1)
program(page4, w4)
logHead++
Logical-to-physical map
92 --> 0
92 --> 4
17 --> 1
33 --> 2
68 --> 3
Garbage version of logical block 92!
Log head
Uninitialized
Valid
w0
w1
w2
w3
w4
1*
1*
1*
Page
0
1
2
3
4
5
6
7
8
9
10
11
Block
0
1
2

At some point, FTL must:

- Read all pages in physical block 0
- Write out the second, third, and fourth pages to the end of the log
- Update logical-to-physical map

Logical-to-physical map

~~92 --> 0~~ 92 --> 4
17 --> 1
33 --> 2
68 --> 3



Trash Day Is The Worst Day

- Garbage collection requires extra read+write traffic
- Overprovisioning makes GC less painful
  - SSD exposes a logical page space that is smaller than the physical page space
  - By keeping extra, “hidden” pages around, the SSD tries to defer GC to a background task (thus removing GC from critical path of a write)
- SSD will occasionally shuffle live (i.e., non-garbage) blocks that never get overwritten
  - Enforces wear levelling

SSDs versus Hard Drives (Throughput)

Device	Random Reads (MB/s)	Random Writes (MB/s)	Sequential Reads (MB/s)	Sequential Writes (MB/s)
Samsung 840 Pro SSD	103	287	421	384
Seagate 600 SSD	84	252	424	374
Intel 335 SSD	39	222	344	354
Seagate Savio 15K.3 HD	2	2	223	223

Dollars per storage bit: Hard drives are 10x cheaper!

Source: "Flash-based SSDs" chapter of "Operating Systems: Three Easy Pieces" by the Arpaci-Dusseaus.