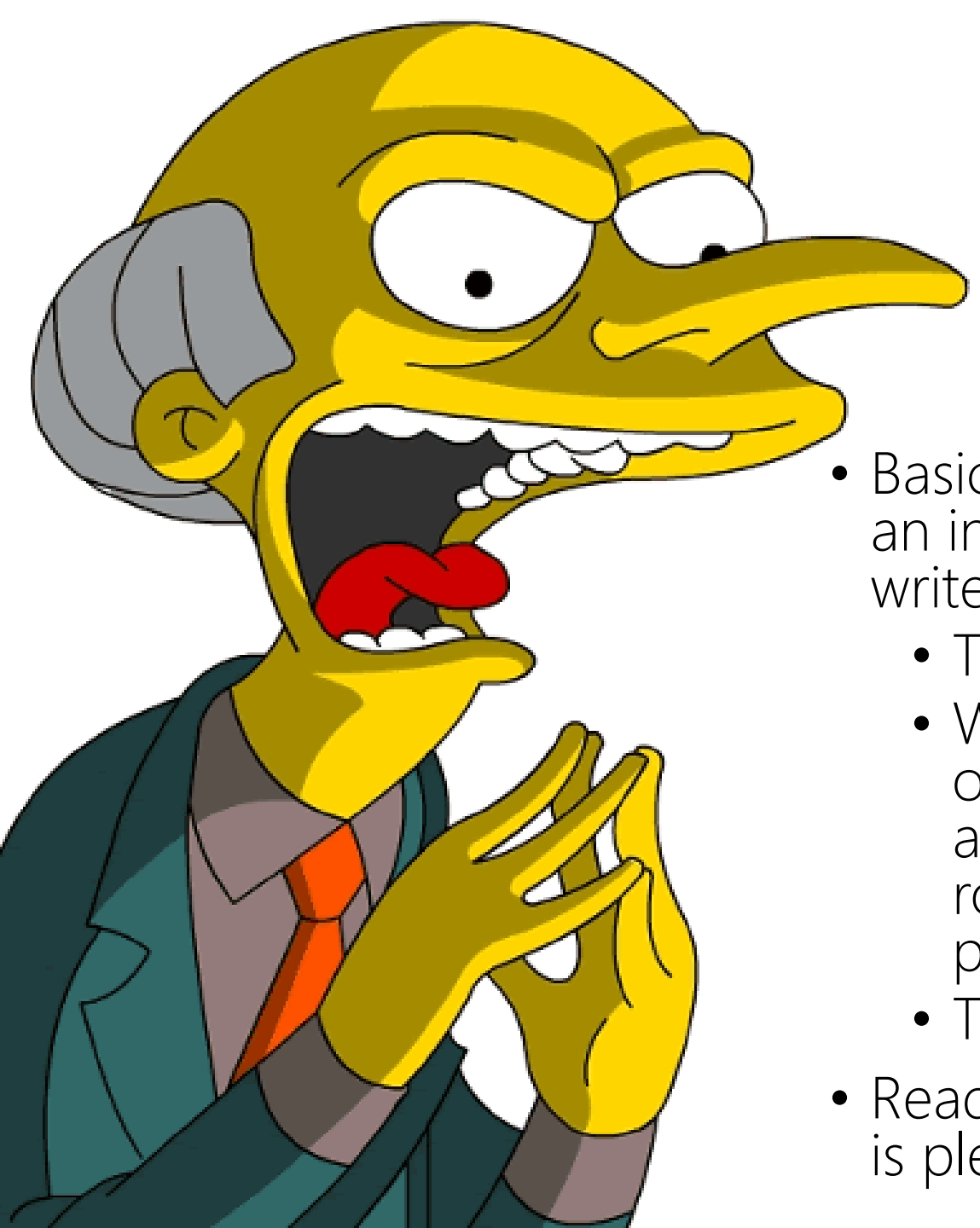
Log-structured File Systems

CS 161: Lecture 15

4/6/17

Motivation

- In the 90s, people realized that physical RAM was getting larger
  - So, buffer caches were getting larger
  - So, once data was read from disk, reads increasingly hit in the buffer cache!
  - Result: file system performance is largely determined by how efficiently you handle writes
- As physical RAM got larger, sequential IOs continued to be much faster than random IOs
  - FFS tried to avoid seeks and rotational delays by allocating related files and directories in the same block group
    - However, related files and directories still weren't directly next to each other
    - So, FFS still paid (short) seek costs and rotational delays
  - Journaling file systems (e.g., ext3, NTFS) write sequentially to the journal, but asynchronous checkpoints still require head movements away from the journal
- So, state-of-the-art file systems couldn't write at full sequential disk speeds



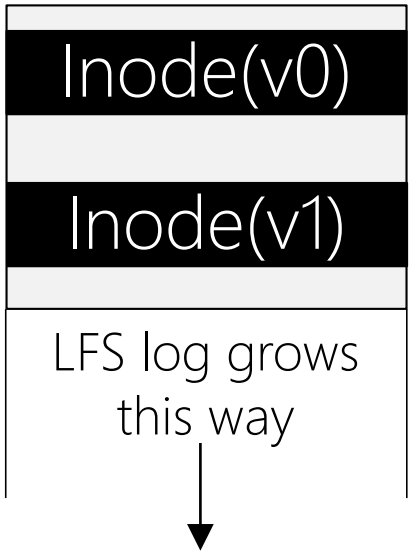
MAKE THE ENTIRE FILE SYSTEM A LOG

YES, JUST A LOG

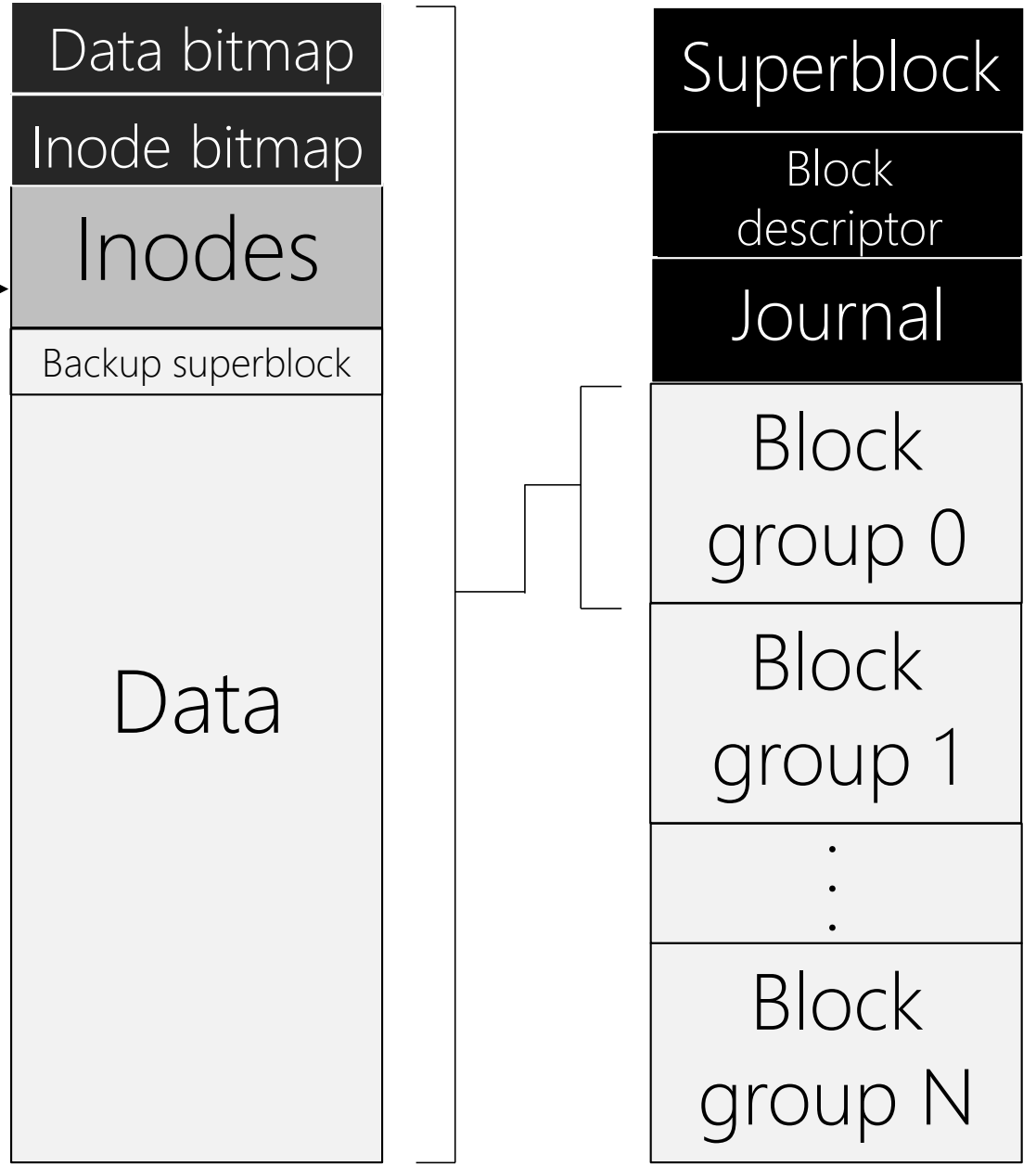
- Basic idea: Buffer all writes (data + metadata) using an in-memory segment; once the segment is full, write the segment to a log
  - The segment write is a sequential write, so its fast
  - We write one large segment (e.g., 4 MB) instead of a bunch of block-sized chunks to ensure that, at worst, we pay only one seek and then no rotational latencies (instead of one seek and possibly many rotational latencies)
  - There are no in-place writes!
- Reads still require random seeks, but physical RAM is plentiful, so buffer cache hit rate should be high

Finding Inodes

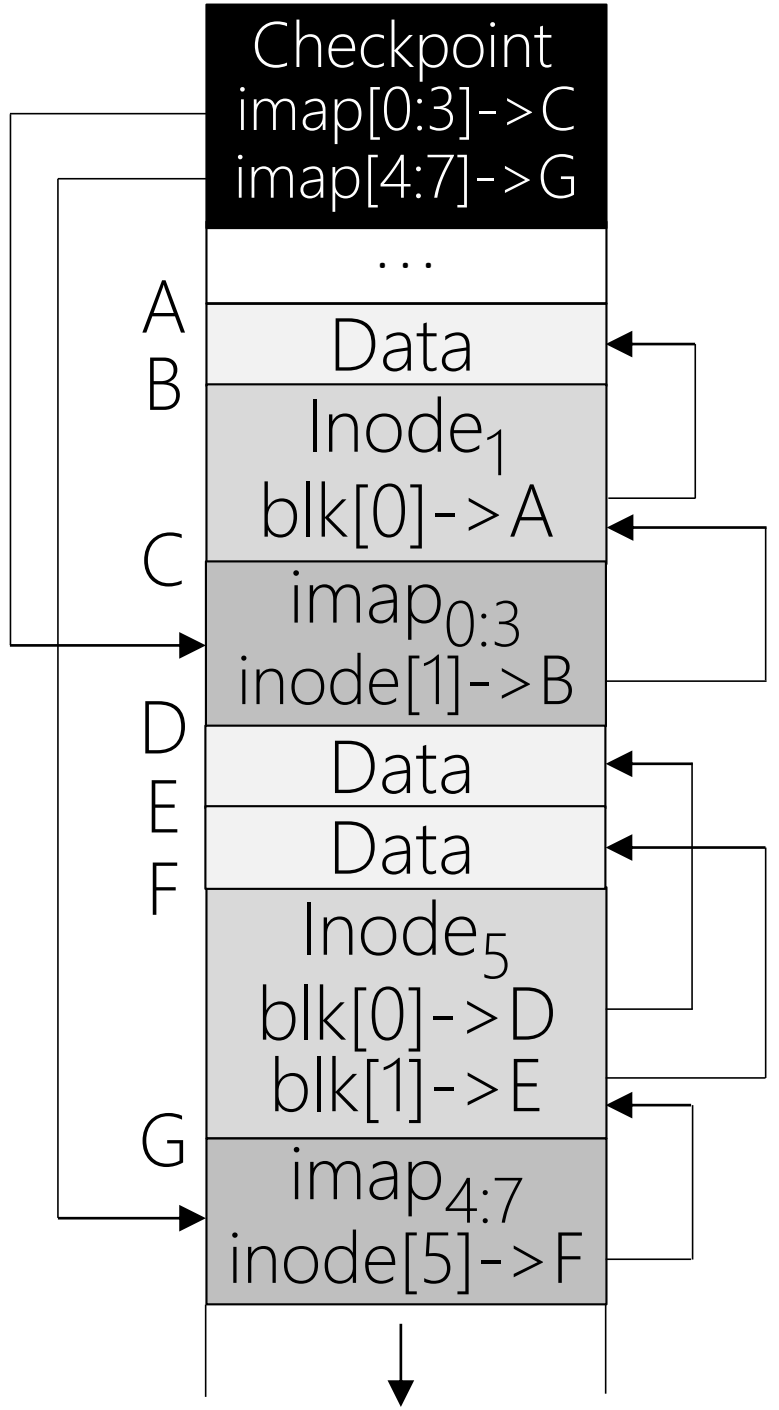
- In a standard file system (e.g., ext3), inodes are statically allocated during file system initialization, and live in a fixed location
- With LFS, there are no in-place updates, so how do we determine the place on disk that has the *last* (i.e., most recent) version of an inode?



- Another challenge in LFS: How do we find and deallocate old versions of data blocks and inodes?
- LFS solves these problems using an inode map which translates inode numbers to on-disk inode locations

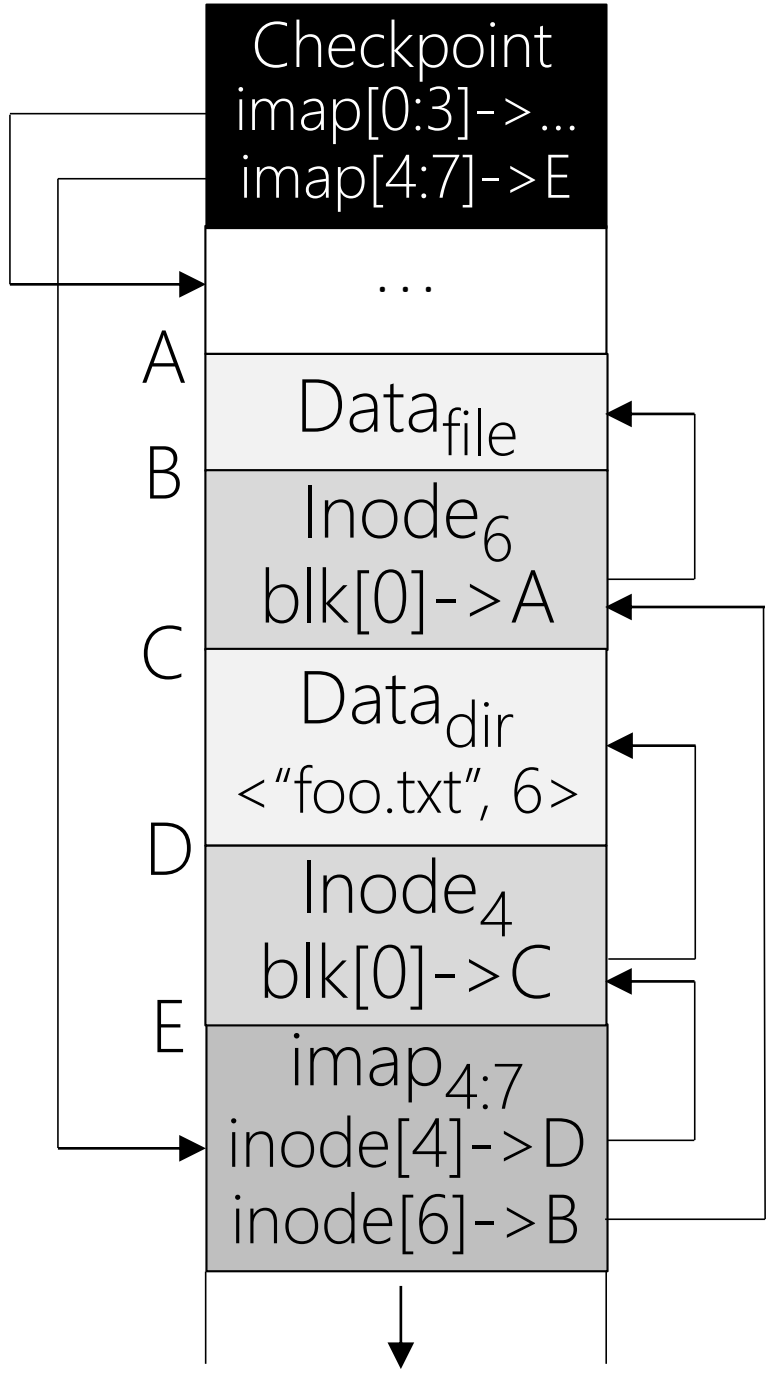


- Trick 1: LFS caches the entire inode map in memory
  - Thus, determining the on-disk location for an inode doesn't require a disk seek
  - However, LFS does need to write map updates to disk to allow for bootstrapping after a crash/reboot
- Trick 2: LFS breaks the inode map into small, logical pieces
  - When LFS must write an updated inode to the log, LFS also writes the piece of the inode map that is associated with the inode
  - Periodically (e.g., once every 30 seconds), LFS updates a fixed location called the checkpoint region
    - The checkpoint region contains pointers to the current versions of the inode pieces
    - During a clean shutdown, LFS flushes the entire in-memory inode map to the checkpoint region
    - After a clean reboot, LFS reads the checkpoint region to create the in-memory inode map



LFS: Directories

- LFS represents a directory as a file that contains <name,inode> pairs
  - This is the same representation used by traditional Unix file systems like ext3
  - However, LFS directories (like LFS inodes) are not updated in place
- Ex: creating a new file foo.txt and then adding a data block to it
- Note that LFS always writes a thing that is pointed-to before a pointer to that thing
  - This will be useful during crash recovery . . .

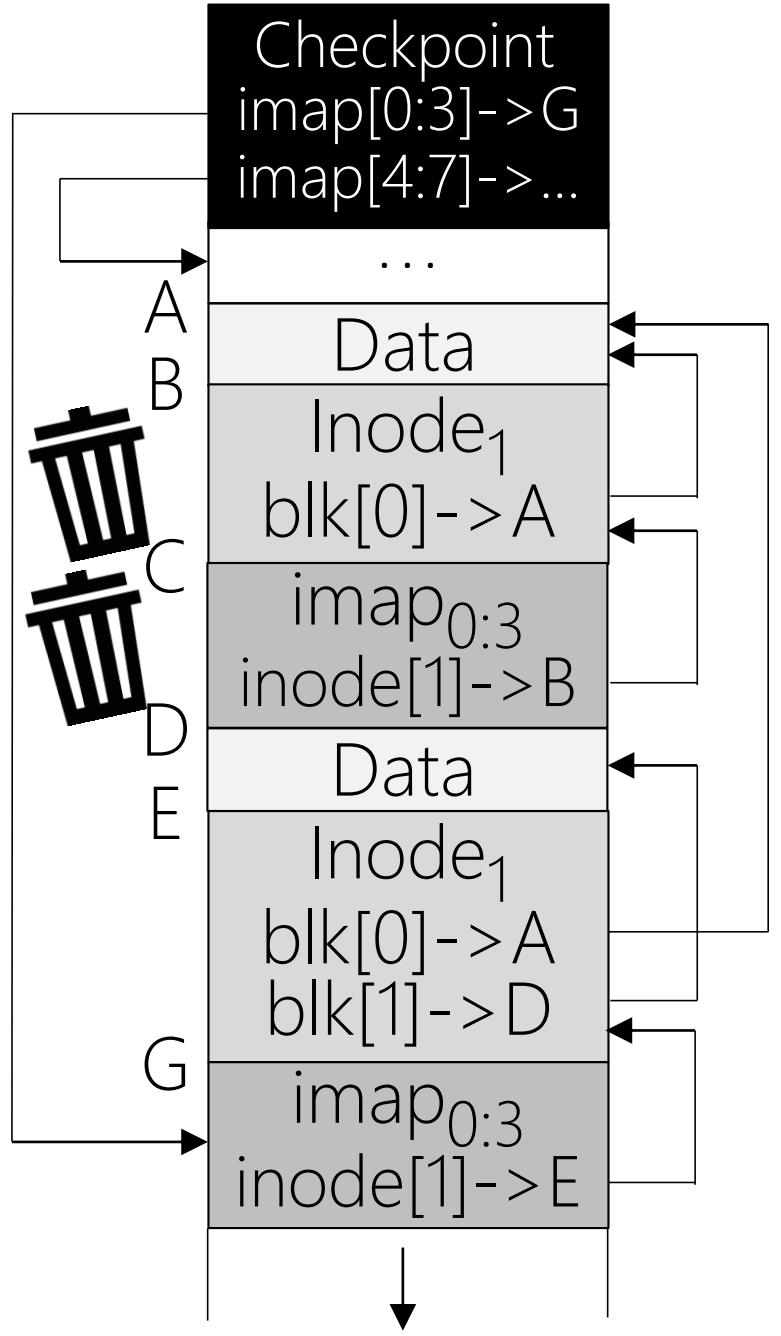


LFS: Reading File Data For A File With Inode Number i

- Assuming that the inode map is cached in memory, LFS requires the same number of disk reads as in a traditional Unix file system
  - Consult the inode map to determine the disk address for inode i
  - Read the inode into memory if it isn't already in the buffer cache, and find the appropriate data pointer
    - You may need to read one or more indirect blocks
  - Read the data block that is referenced by the data pointer

LFS: Garbage Collection

- As the file system handles new writes, older parts of the journal become stale
  - Ex: Appending a new block to a file invalidates the file's old inode (and the associated imap chunk)
  - Keeping stale data could actually be useful if we wanted to implement a versioning file system!
  - Regardless, stale data will eventually need to be garbage collected
- LFS performs garbage collecting at a segment granularity
  - The cleaner reads X partially invalid segments, and writes the live data to Y<X segments (which will be totally valid)
  - This approach keeps GC reads and writes sequential!



LFS: Identifying Garbage

- To assist with the garbage collection of data blocks, LFS includes a "segment summary" at the beginning of each segment it writes
  - For each data block in the segment, the summary describes:
    - the inode number N of the inode that owns the block
    - the offset F of the block inside the inode's data region (e.g., "this block is at byte offset 4096")
  - Using the summary, the cleaner determines whether a data block is live by:
    - reading inode N (either from the buffer cache or from disk via the inode map)
    - finding the appropriate data pointer for offset F, and seeing whether that pointer points to the data block in the segment being examined
  - When live data blocks are written to a new segment, the cleaner must also write new versions of the corresponding inodes and imap entries
- The inode map indicates which inodes are live, and each on-disk inode struct contains the inode number
  - So, when the cleaner finds inodes in an old segment, the cleaner can check the inode map to see whether the inodes should be written to a new segment

LFS: How Exactly Should We Implement GC?

- Scheduling garbage collection turns out to be tricky
  - Could only run the cleaner when free space is low
    - Good: No garbage collection when space is plentiful!
    - Bad: The disk has finite size, so eventually space will run out, and then the cleaner will have to run frequently (possibly blocking writes from normal processes until free segments are available)
  - Could run the cleaner at regular intervals, but then the cleaner will regularly contend with normal processes for disk bandwidth
- Determining how to cluster live data in compacted segments is also tricky
  - Place data from the same directory in the same segment, to improve spatial locality for subsequent reads?
  - Place data of the same age in the same segment, to create "cold" segments that won't need further compaction, and "hot" segments that will quickly fill up with stale data and will release a lot of free space upon compaction?

The LFS Flame Wars of the 1990s

- In two papers (one in 1993, one in 1995) Seltzer compared FFS to a BSD port of Sprite's LFS, finding that:
  - LFS is great for workloads with:
    - frequent small writes
    - read patterns that are amenable to hitting in the buffer cache
    - enough idle time for cleaning to run without hurting foreground tasks
  - LFS is not great when:
    - the disk is full (because the cleaner must read many segments just to find a little free space)
    - writes are too random (because dead space will be spread evenly throughout the segments, forcing the cleaner to read many segments to free space)
- Ousterhout (who wrote Sprite LFS) claimed that:
  - BSD LFS was poorly implemented and had performance bugs
  - The benchmarks used to evaluated BSD LFS were unfair (e.g., the compilation benchmark was CPU bound and doesn't provide much insight into file system behavior; the transaction processing workload contains a pathological number of random writes)
  - FFS fragmentation can hurt performance just as much as LFS cleaning

LFS: Crash Recovery

- Remember that the on-disk checkpoint region contains pointers to all of the on-disk imap pieces
- LFS updates the checkpoint region every 30 seconds
  - After a crash, LFS can regenerate the in-memory map by reading the checkpoint region
  - To ensure that the checkpoint region is updated atomically, LFS keeps two copies of it in two different portions of the disk
    - LFS alternates which location receives the current checkpoint
    - To detect interrupted checkpoints, LFS writes a timestamp, the pointers to the imap fragments, and then the same timestamp; the checkpoint is only valid if both timestamps match
  - This approach works, but we lose the last 30 seconds of write data :-(

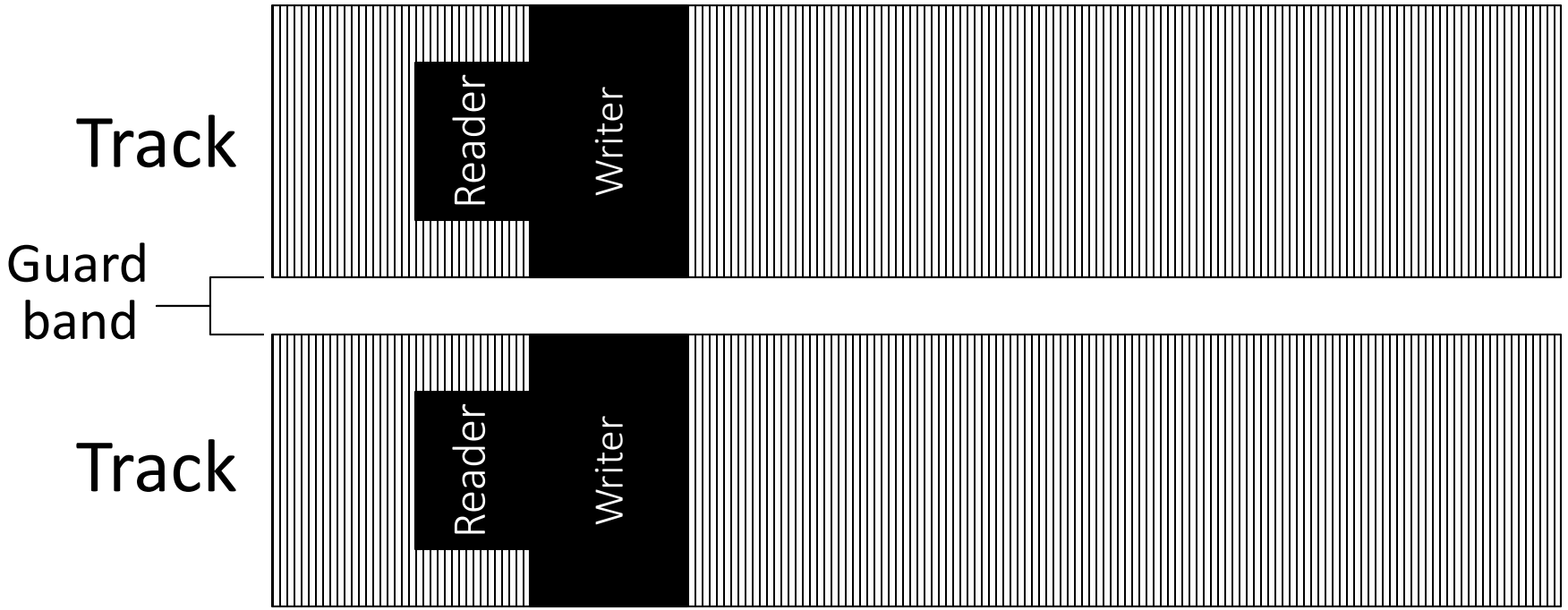
LFS: Crash Recovery

- To lose less data, LFS stores a small amount of additional information:
  (1) each checkpoint includes the current log head position
  (2) each checkpoint also includes the order of the on-disk segments in the log
  (3) each segment summary includes the location of any inodes in the segment
- So, the full LFS recovery protocol is to:
  (1) read the checkpoint region to initialize the in-memory inode map
  (2) roll forward through the segments, starting from the segment mentioned in the checkpoint region, using the discovered inodes to update the in-memory inode map (and thereby incorporate those inodes' data blocks into the recovered file system)


Shingled Drives

Adding More Storage Capacity To Disks

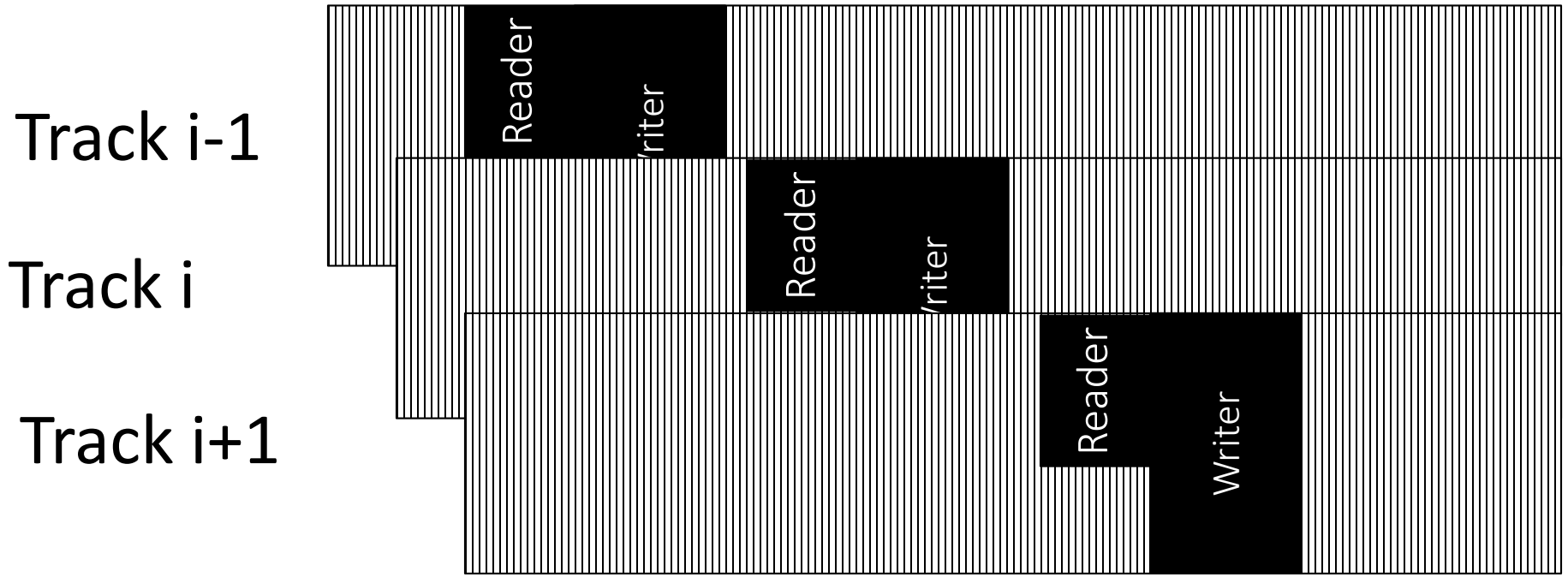
- A disk head has two components: a read head and a write head
- Standard approach: perpendicular magnetic recording
  - Write head sets the magnetic polarity of a bit "up" or "down"
  - Read head retrieves the polarity of a bit
  - Each track is separated by a guard band



- Could try to increase capacity by making write heads smaller . . .
  - . . . but smaller write field isn't strong enough to flip bits :-(

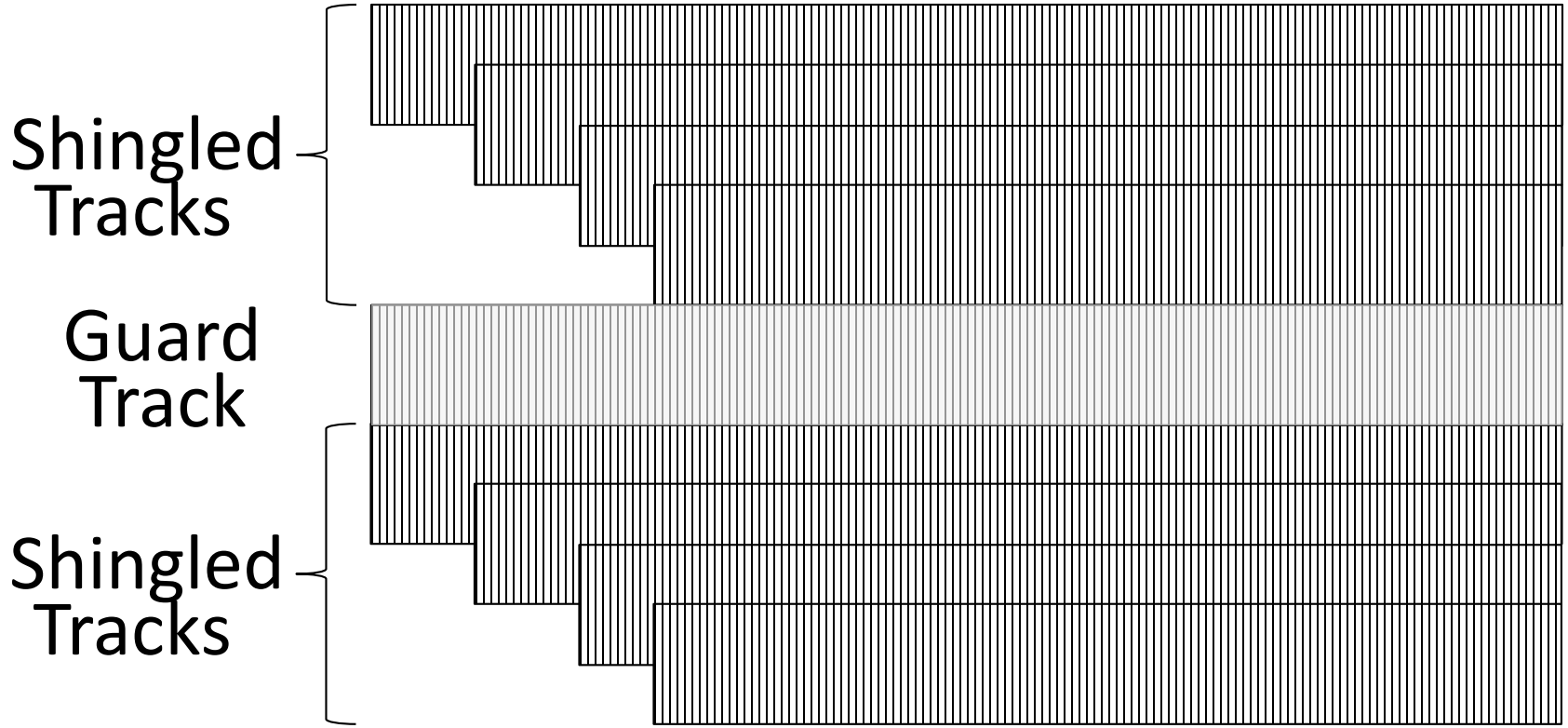
Shingled Magnetic Recording

- Idea: Overlap tracks!
  - Good: Increases capacity
  - Bad: Writes to track i overwrite part of track i-1
    - Data from track i-1 can still be read
    - However, if you write new data to track i-1, you invalidate track i's data :-(
    - So . . . write the disk sequentially and then never update?



Shingled Magnetic Recording

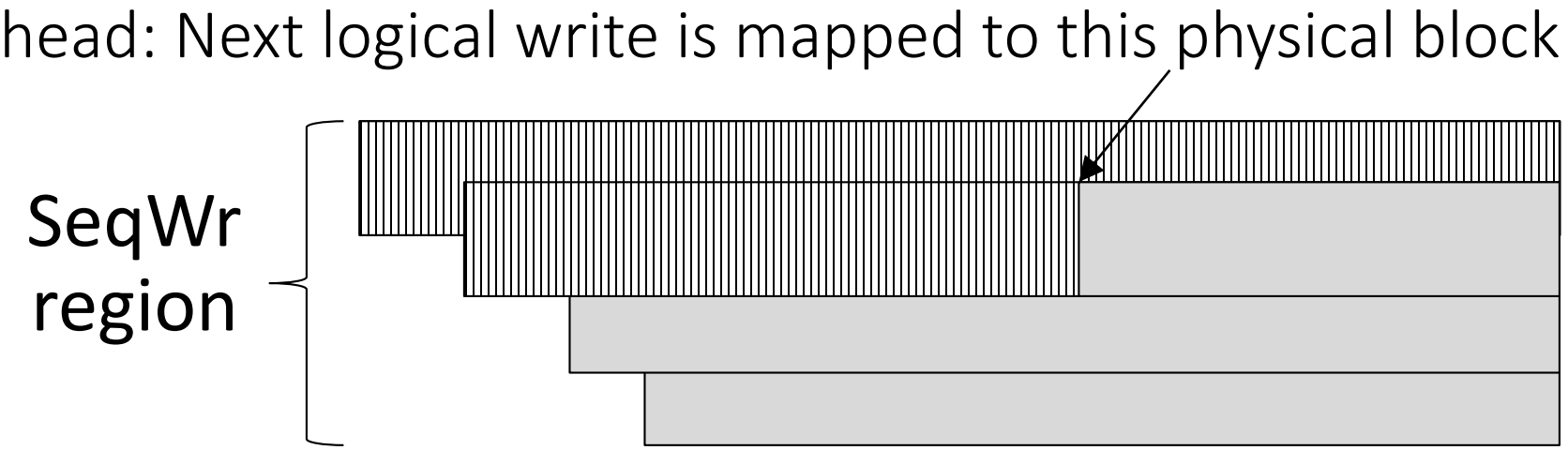
- Use guard regions to create "sequential write regions"
  - Each region should be written sequentially . . .
  - . . . but you can use some regions as scratch/temporary write space without overwriting other regions



- Guard tracks provide some normal r/w storage, but sequential writes are still a good idea!

SMR: Enforcing Sequential Writes

- Use a log-based storage layout + "shingle translation layer"



- "Shingle translation layer" can be implemented by the drive firmware or by the file system
- Log-structured file systems work well with shingled drives, since almost all writes are sequential ones
  - Log-structured file systems also work well with SSDs, which need to spread writes across all blocks to implement wear levelling
- As always, beware of garbage collection overhead!

Summary: The File Systems That We Have Loved

- The original Unix file system had a simple design, but was slow
  - Data layout did not provide spatial locality for directories and files with temporal locality
  - Only provided 4% of the sequential disk bandwidth
- FFS leveraged knowledge of disk geometry to improve performance
  - To reduce seeks:
    - Related files and directories are stored in the same cylinder group
    - Block size increased from 512 bytes to 4KB
  - To minimize rotational latency, FFS used "skip sectors"
- However, FFS had slow failure recovery due to the horrifying multi-pass nature of fsck

Summary: The File Systems That We Have Loved

- Journaling file systems use write-ahead logging to make crash recovery faster
  - ext3 performs redo logging of physical blocks
  - NTFS performs redo+undo logging of operations
  - The journal converts random writes into sequential ones, but the file system must eventually issue random writes to perform checkpoints
- LFS turns the entire file system into a log
  - Assumes that the buffer cache will handle most reads, so making writes fast is the most important thing
  - The log turns all writes (random or sequential, large or small) into large sequential writes
  - Fast recovery
  - Requires garbage collection, which (depending on the workload) may eliminate benefits from making all writes sequential