EECS-343 Operating Systems
Lecture 18:
Log-structured File Systems

Steve Tarzia

Winter 2018

Northwestern

Diagrams taken from "Three Easy Pieces" book

Announcements

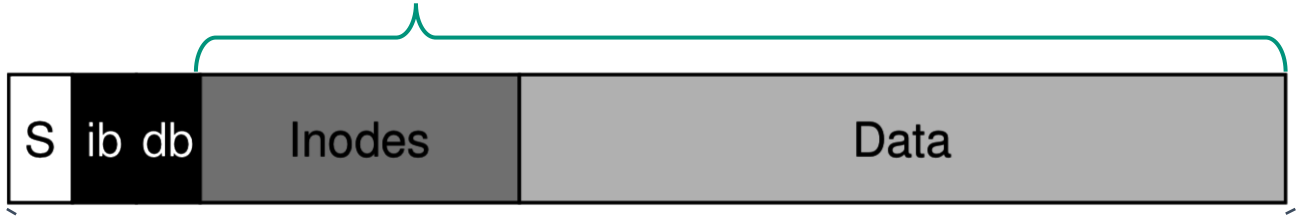
- Project 4 is due on Wednesday.
- HW4 due on Friday
- Final exam is this coming Monday, June 10th, 3-5pm
    - Final will be cumulative
    - There will be some code-reading questions
    - More questions will be objective, and fewer will be "essays."
    - Open book, open notes
    - No passing notes or books during the exam

Last Lecture – Storage Layer Interactions

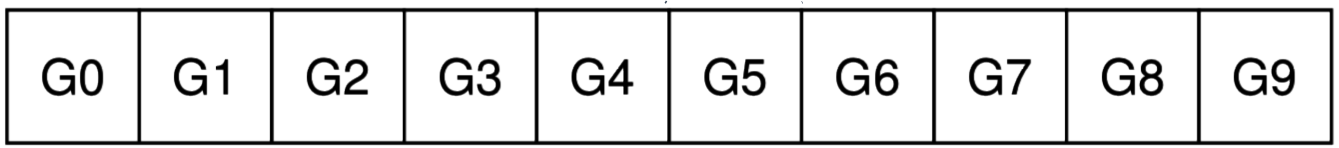
- Showed layered design of xv6 storage system
- Implementation of each layer uses only the layer(s) directly below
  - Must provide an API suitable for implementing the layer(s) directly above
  - Deeper layer are hidden.
- **`defs.h`** makes a subset of kernel functions in each file "public."
- Linux has a virtual file system (VFS) layer that allows multiple filesystems to coexist in one machine.

Fast File System reduces seek times

- Recall that a read/write accesses four or more disk blocks
  - In classic Unix/xv6 FS, long ***seeks*** are needed to move between inodes and data blocks:



- Fast File System and its descendants, like ext2 & ext3, divide the disk into ***block groups***, each arranged like a miniature filesystem:



  - If possible, put a file's inode and its data blocks (and parent and siblings) in the same block group. This is ***locality*** again – putting related things nearby.

Throughput problems in traditional filesystems

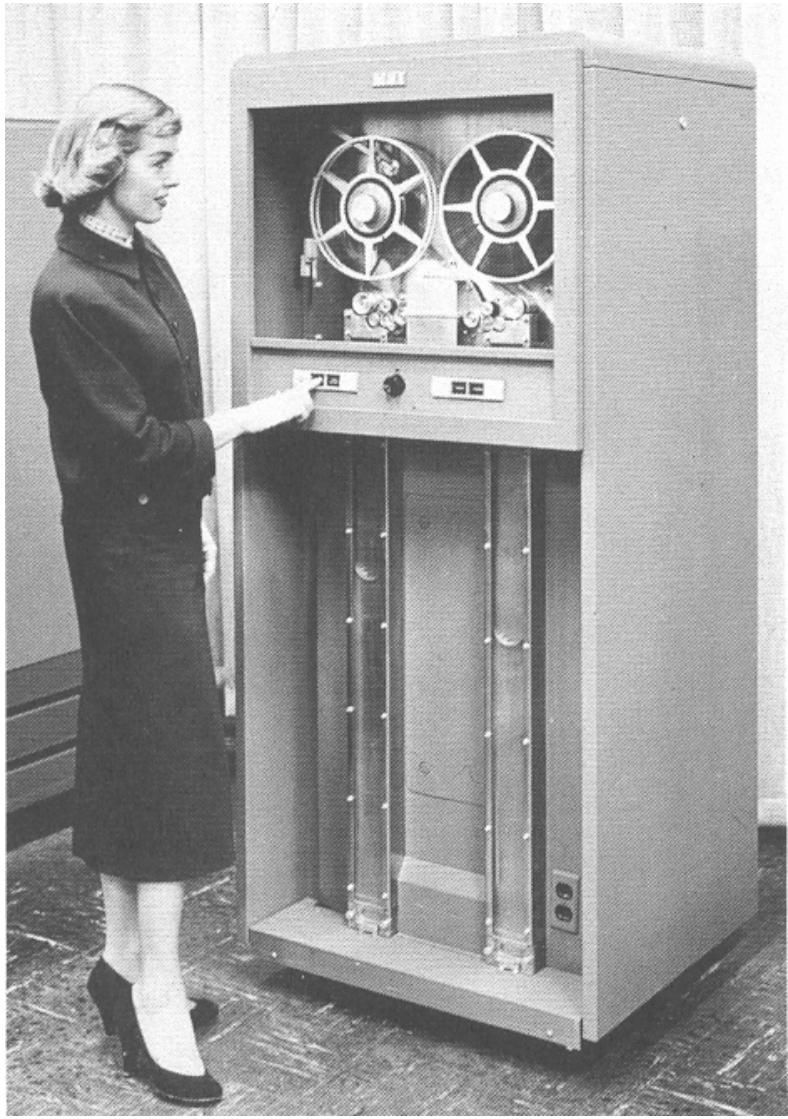
- Usually we think of a disk abstractly as an ***array*** of data sectors.
- But ***sequential*** reads/write are much faster than random accesses.
  - xv6 FS is ignorant of this. It just implements a buffer cache layer to reduce repeated disk I/Os to the ***same locations***.
- Caching does not help if you need to **write lots of new data**.
- If writing **one big file**, we can get a large sequence of contiguous data blocks.
- But, if writing many **small files**, good performance is very difficult to achieve.
  - FFS block groups reduce seek length.
  - Delaying and batching requests allows the disk firmware to reorder them.
  - But we are still doing a lot of seeks. 😔

Log-structured Filesystems (LFS)

- A radically different filesystem design.

  How would you design a filesystem for:
  - A tape drive, with very, very slow seek times?
    - ~one minute to rewind through the entire tape
    - 500 inches/second. 1500-3000 foot tape reel.
  - A pen and paper notebook?
- Try to do every write sequentially. But how?
- Idea is to treat the filesystem like a ***log***.
  - A ***log*** is a sequence of events, new ones at the end.
- When data changes, don't bother going back to edit the original, just store new copy at the end.
  - In the simplest case, assume infinite capacity.

Tape drive circa 1953:



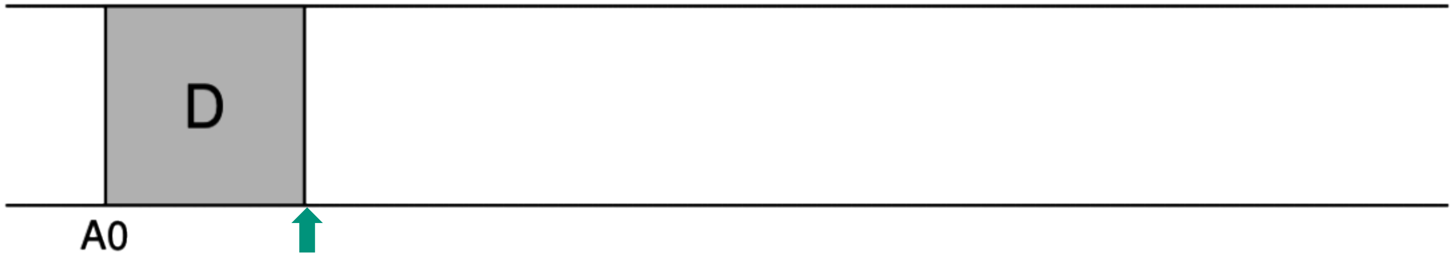
http://www.columbia.edu/cu/computinghistory/701-tape.html

LFS is optimized for write performance

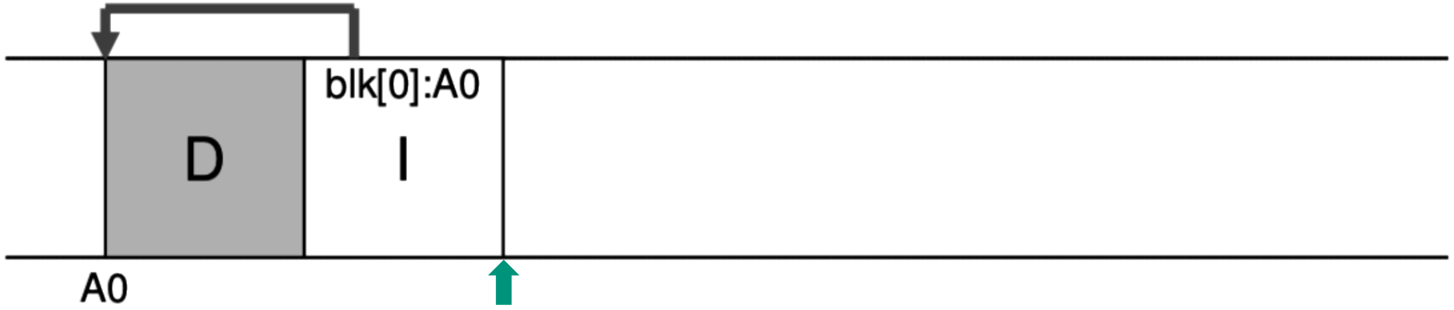
- Traditional filesystems access many different parts of disk during reads and writes, but buffer cache is meant to fix this problem.
  - However, caching only helps with repeated access to the same disk block.
- It turns out in real-world workloads:
  1. Reads of the same location are often repeated, but…
  2. Disk space is cheap, so programs often write lots of data, even if most is never used again.
- Caching helps with #1 but not with #2.

Just keep writing to the end, ***sequentially***

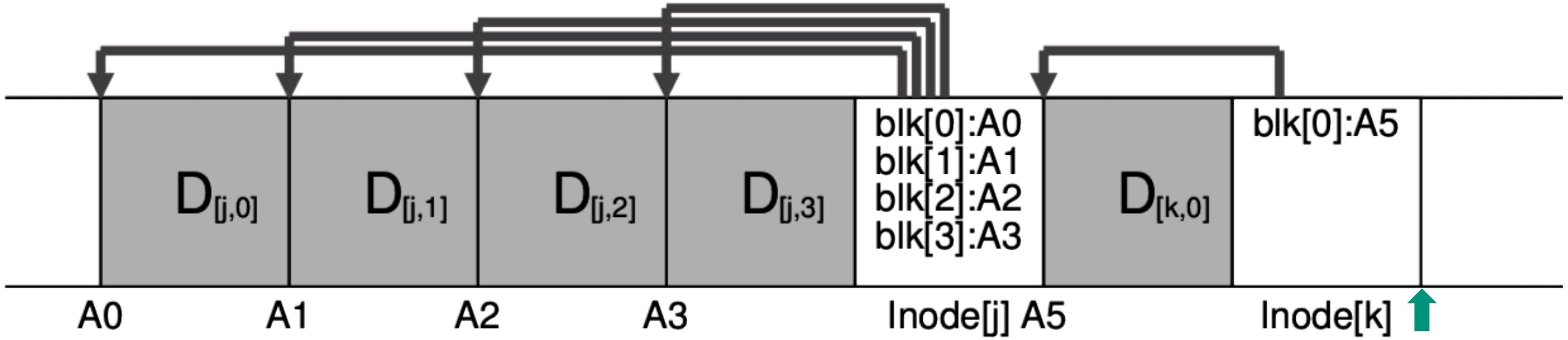
- LFS still has inodes and data blocks, it just ***places*** them differently.
- **Always write to the end**. For example, when writing a small file:
  - Write data block:



  - Then write the inode:



Write two files

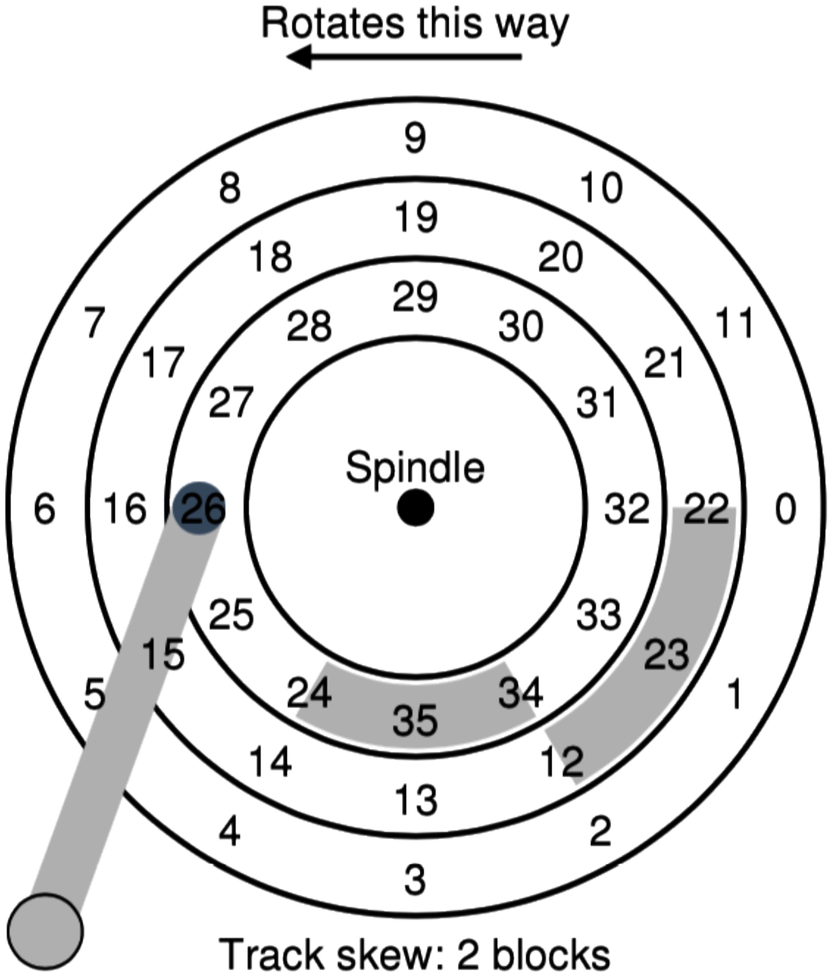


- As always, we write the data blocks before inodes to minimize the impact of an interruption/crash.

- **Note**: this picture assumes that we open the file and write a large chunk of data all in one big operation.
- This can be achieved by delaying the writes to disk with a buffering/caching layer.

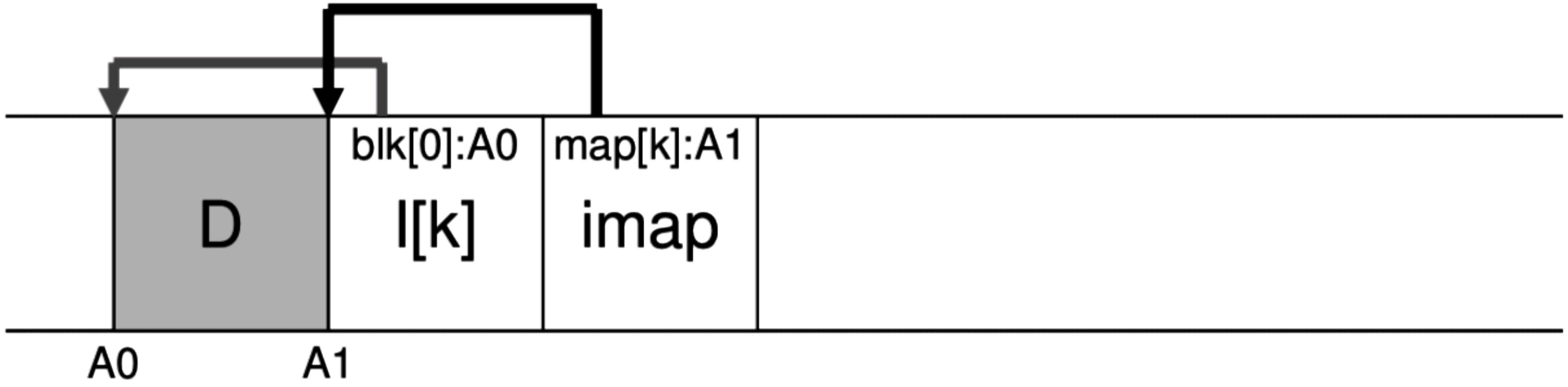
Writing in large segments reduces rotational delays

- Even though we are writing to sequentially increasing locations, there is still the possibility of a long *rotation delay* if the requests are not issued together.
  - For example, writing sectors 27, 28, 29 can be very fast in the best case, but only if we are ready to write 28 immediately after 27.
  - A small delay between **write***(27)* and **write***(28)* might make us to wait for a full disk rotation.
- So, LFS ***buffers writes*** and sends them in large batches (few MB) called ***segments***.
  - Goal is to balance rotation & seek delay with segment data transfer time.



Inode map tracks inodes within a segment

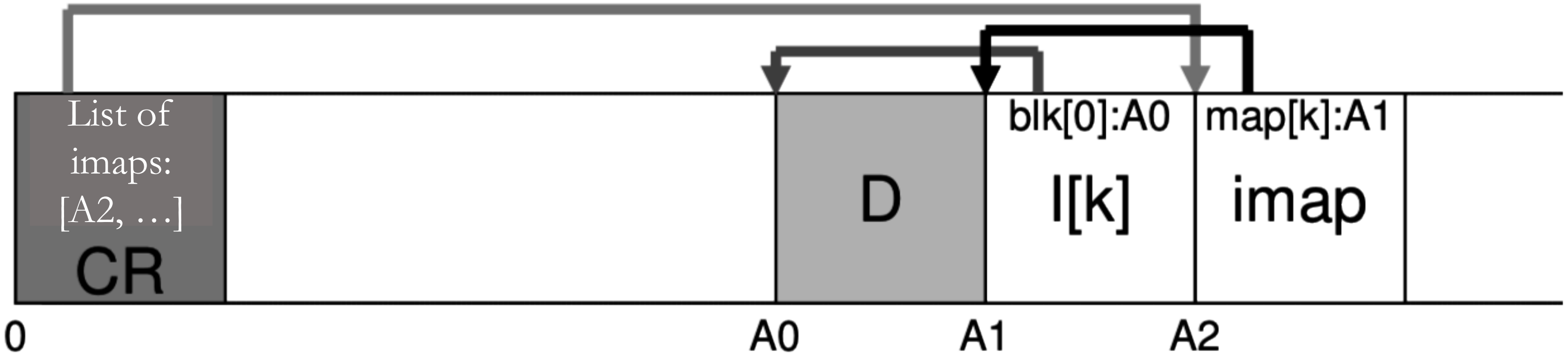
- Formerly, inode numbers could be used to find inode struct in an array.
- LFS makes it more difficult to find inodes. They are placed in arbitrary locations on disk. Now how do we find a particular inode?



- Each segment has an ***inode map*** giving address of each of its inodes.
- Recall that segments are large, and can contain hundreds of inodes.
- But this is not a complete solution, because there are many segments and many inode maps, themselves in random locations on disk.

Two levels of indirection to find inodes

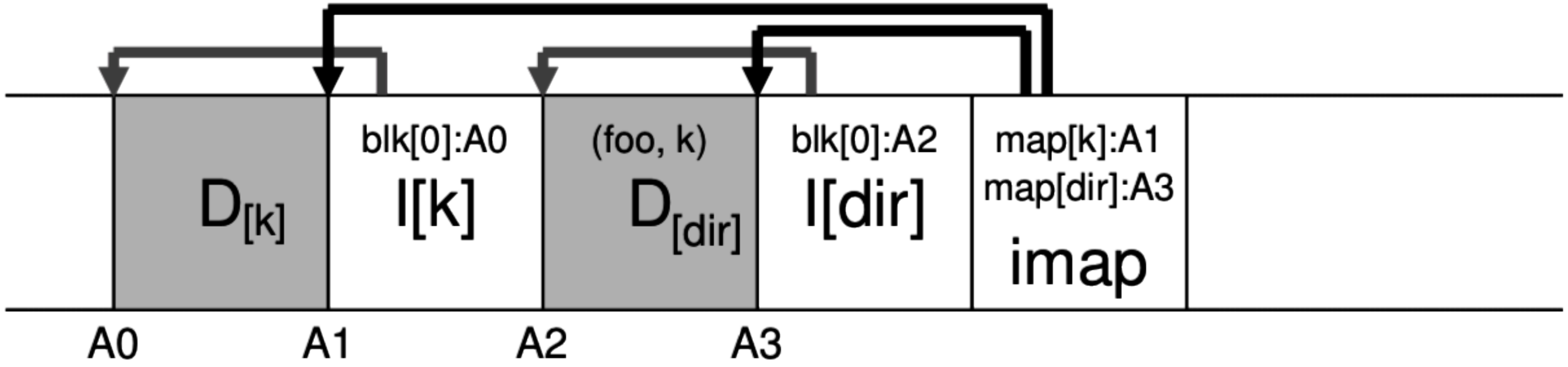
- At the beginning of the disk, store a ***checkpoint region***, which just points to all the ***valid*** inode maps on disk:



- The i-map is distributed throughout the disk in all the *valid* segments.
  - Finding an inode involves looking at the entire imap. This could be slow, but in practice we should be able to keep then entire ipap cached in memory.
  - Checkpoint region keeps a *persistent* record of the distributed imap.
- Infrequently (~30 seconds), seek to the beginning of the disk to *flush* the in-memory cache of the checkpoint region.

A segment with a file and a directory

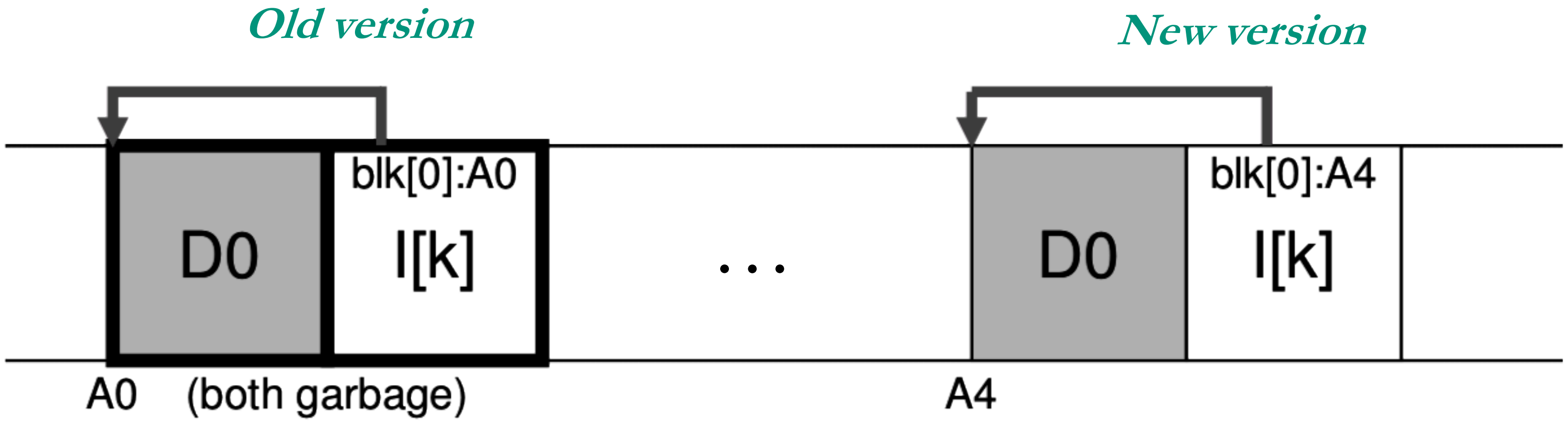
- Directories are also stored in the same way:



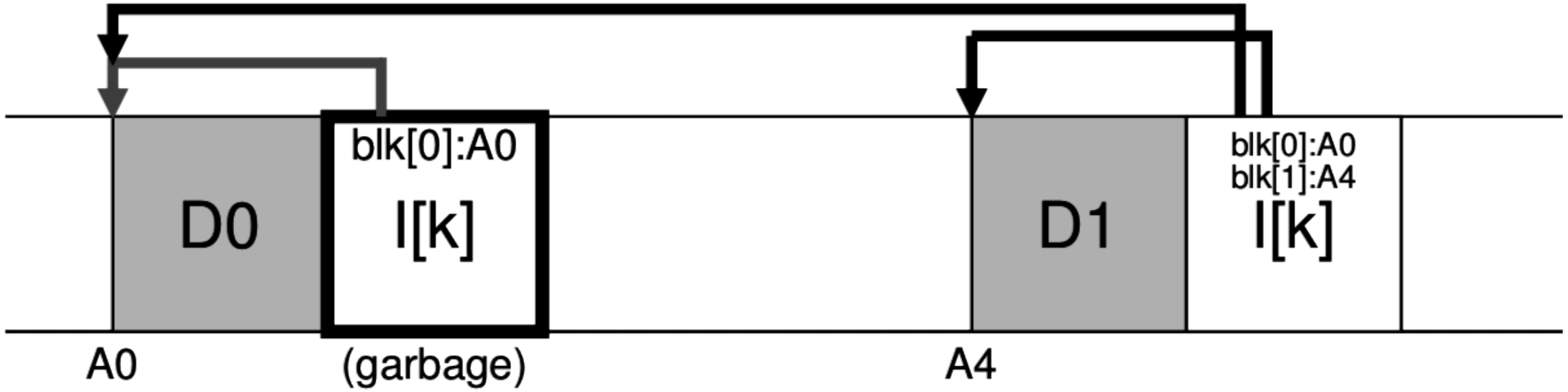
- Notice that the directory lists the <filename, inode#>, as usual.
- This inode # does not tell us where to find the file inode.
  - Must check the in-memory inode map to find the associated disk block.
  - At boot time, read the checkpoint region and the distributed inode map.

Never go back to modify existing data

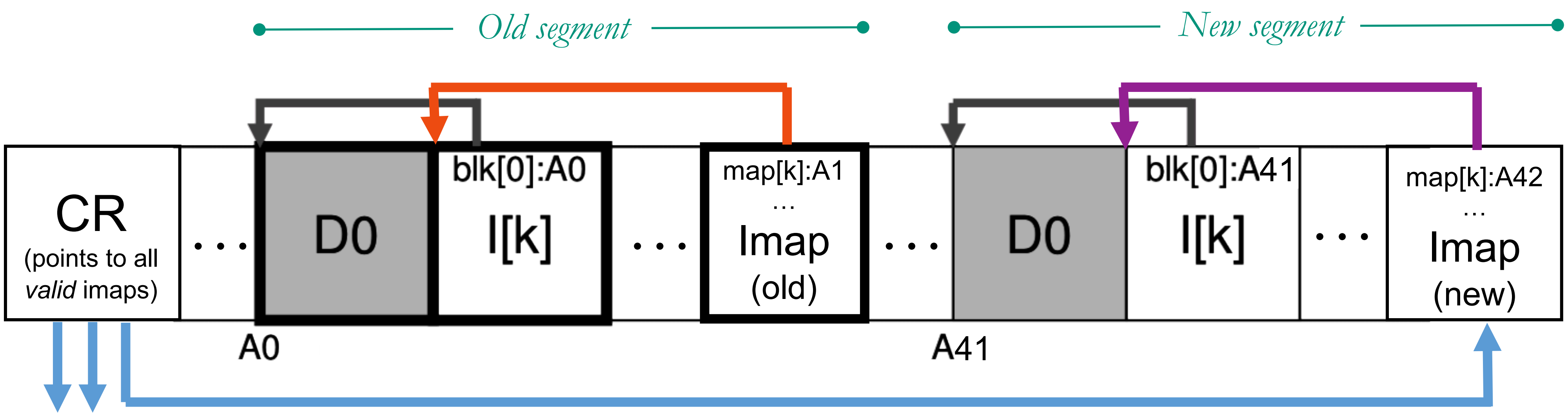
Always write a new copy of the entire block. If editing data:



If appending data to a file, the inode is edited:

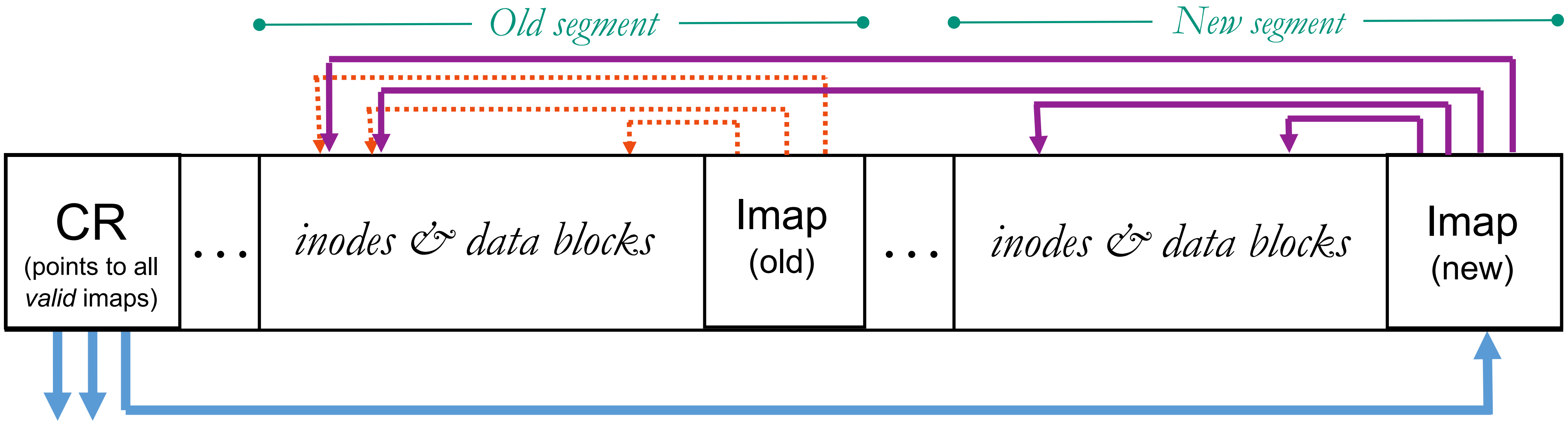


Pointing to the new version *(after file edit)*



- Old data still exists on the disk, but the in-memory i-map and its persistent copy in the checkpoint region do not refer to it.
- If we disk space is infinite, that's good enough.
- If we save an old version of the checkpoint region, it can be used to view and old ***snapshot*** of the filesystem!
  - A filesystem that preserves old snapshots is a ***versioning file system***.

Rewriting inode maps



- A segment holds many inodes, declared in one i-map.
  - Checkpoint region points to i-maps that must be ***entirely valid***
- So, when editing a file, not only must the file be rewritten (the data block(s) and inode), but the ***segment's i-map must be rewritten.***

Intermission

- *Show previous slide*

Pen and paper example

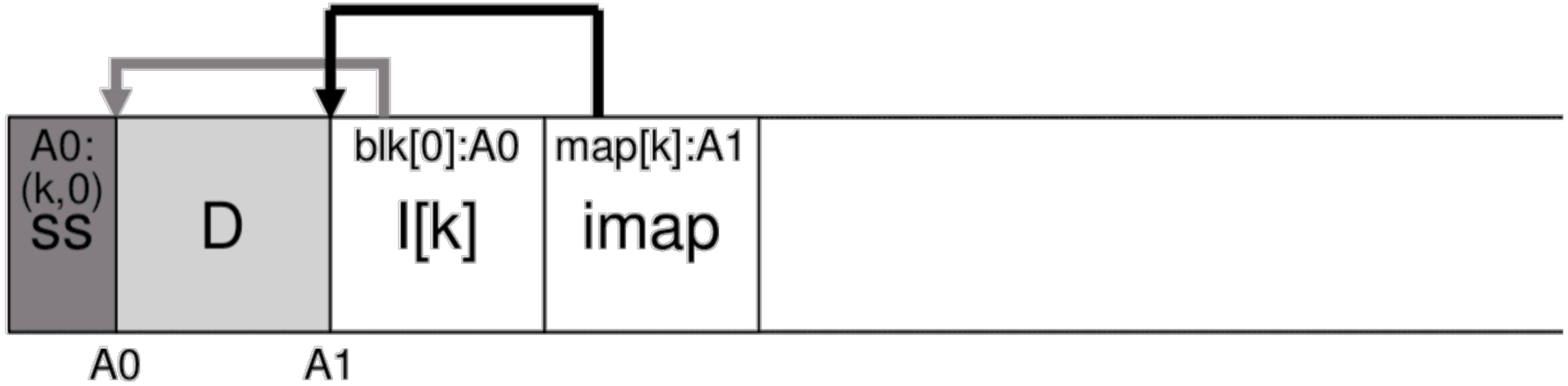
Appending to a file

Disks actually have finite size

- Cannot write sequentially forever
- Cannot keep old versions of data around indefinitely
- Eventually need to ***garbage collect*** segments with free space
  - Actually, we want to free *full-sized* segments.
  - If we encounter a segment that is partially filled, then free the full segment and rewrite a ***compacted*** version of the segment at the end of the log.
- After reaching the end of the disk, restart the writing at the beginning of the disk, but only write to the "holes" left by the garbage collector.
- Garbage collector periodically scans through the disk, perhaps during idle time.
  - But how can GC decide whether which blocks are *live* or *dead*?

Backward pointers aid garbage collection

- The distributed i-map tells us directly whether an *inode* is live or dead.
- *Data blocks* are more difficult to classify
  - Naïve approach is to examine every inode on the disk, looking for a reference to the block; but this is way too slow.
  - We want some kind of backward reference from data block to inode.



- Solution: add a ***segment summary block*** indicating the inode number and block offset for each data block in the segment.
  - Check whether inode listed in SSB still refers to the data block.

Final LFS notes

- The main idea of a log-structured filesystem is also called ***copy on write*** and ***shadow paging*** (in DBs).
    - This is different than "copy on write" of process memory when forking.
    - However that other kind of CoW can also be implemented for file copies.
- ZFS, Btrfs, and new Apple FS are log-structures filesystems.
- Also used in Linux LiveCDs to make a read-only disk appear writeable (as long as you have enough space in RAM for the writes).
- How to make LFS ***reads*** fast?
    - Writes are naturally sequential, but reads can involve lots of seeks.
    - As always, batch them together so they can be reordered to minimize seeks.

Data integrity

- With trillions of bits stored on a disk, it's very possible that one will be flipped due to a hardware malfunction, radiation, etc. ("bit rot")
- ***Checksum*** is a standard way to detect data corruption:
  - It's a mathematical function that produces a small summary of the data
    - Different checksum functions can be used: CRC, MD5, SHA1
  - Result has fixed length, but input can be of arbitrarily-large size
  - It's essentially a **hash** function: same input always gives same output.
  - Cannot be perfect, due to pidgeonhole principle
    - Sometime two different inputs will produce the same output, so not all errors are detectable.
- Store data or metadata checksums in a filesystem to detect corruptions.
  - ZFS and Btrfs do this.

Networked File Systems

- These are used:
  - When users must access their documents from multiple machines.
  - In huge systems, specialized server hardware is dedicated to storage and different hardware is used for computing tasks.
- Networked filesystem can be served by one machine,
- Or, for "big data" problems, multiple machines can be clustered to form a ***distributed/parallel filesystem***.
- Machines in the Wilkinson Lab and T-Lab mount home directories through ***NFS***, a Unix protocol for networked file systems.
  - Windows has something similar, called ***SMB/CIFS***.
- Quest supercomputer at NU has a 3.5PB GPFS parallel filesystem.

Why move storage further away?

- Surely, networking adds some latency and complexity.

But:

- Modern networks are fast.
- It's faster to access a neighbor's RAM than to access your own disk!
- Fault tolerance requires an array of disks (eg., RAID5) which may not fit in your client machine.
- Can manage *backups* centrally, rather than relying on client users/HW.
- ***Pooling*** storage leads to less wasted space.
- Can ***deduplicate*** shared data.
- Allows access of same data from multiple devices.

Etc., etc., …

Recap - Log-structured File System

- Tries to make all writes ***sequential***, at the end of the disk (at first).
- ***Never edit*** data blocks or inodes, just write new copies and stop referring to the old versions. Inodes are scattered throughout the disk.
- ***Checkpoint region*** points to distributed inode map, to find inodes.
  - CR is the only thing that is always written in a well-known location.
  - Using an old version of the checkpoint region lets us see the filesystem as it looked in the past. LFS can be extended easily to become a *versioned file system*.
- ***Garbage collector*** occasionally scans FS to ***compact*** segments with old, unused versions of blocks.
- Restart from start of disk after reaching the end, filling in holes.