Chapters 10, 11 and 12: Magnetic Disk Scheduling

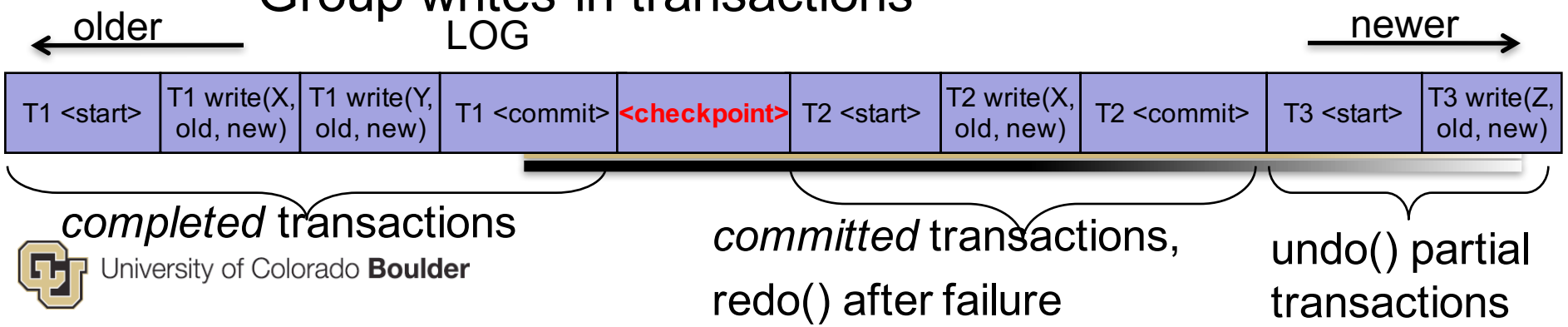
CSCI 3753 Operating Systems

Prof. Rick Han

University of Colorado **Boulder**

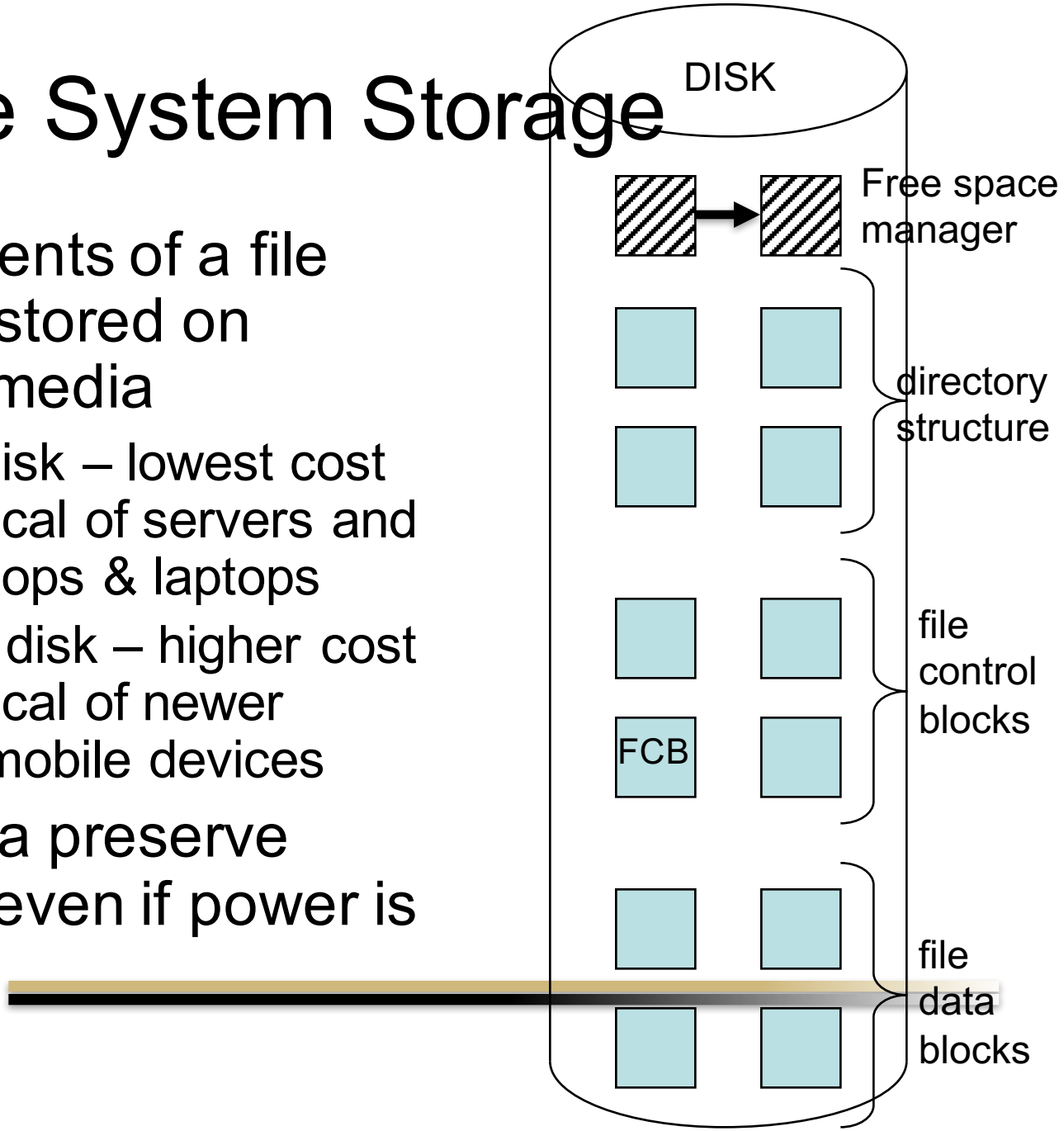
Recap

- Managing free space on disk in a file system
  - Linked list-based approaches
- File System Performance
  - Read ahead
  - Asynchronous Writes
- File System Reliability
  - Journaling/Log-Structured File Systems log each write *before* the write = write-ahead logging
  - Group writes in transactions



File System Storage

- All the elements of a file system are stored on permanent media
  - Magnetic disk – lowest cost per bit, typical of servers and older desktops & laptops
  - Solid state disk – higher cost per bit, typical of newer laptops & mobile devices
- These media preserve digital data even if power is lost

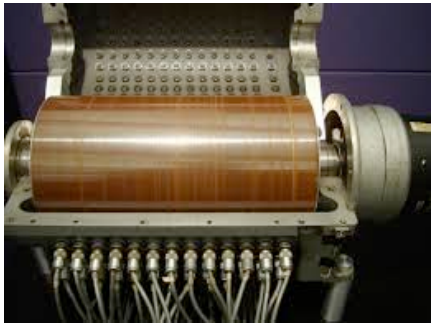


Magnetic Media

- Can set a bit to 1 or 0 by applying a strong enough magnetic field to a small region of a magnetic film
  - This region retains its magnetic sense even when power is removed
- To set or read a sequence of bits, e.g. file,
  - Either move the magnetic head over the magnetic media,
  - Or move/rotate the media under the (static) magnetic head – this is the easier approach

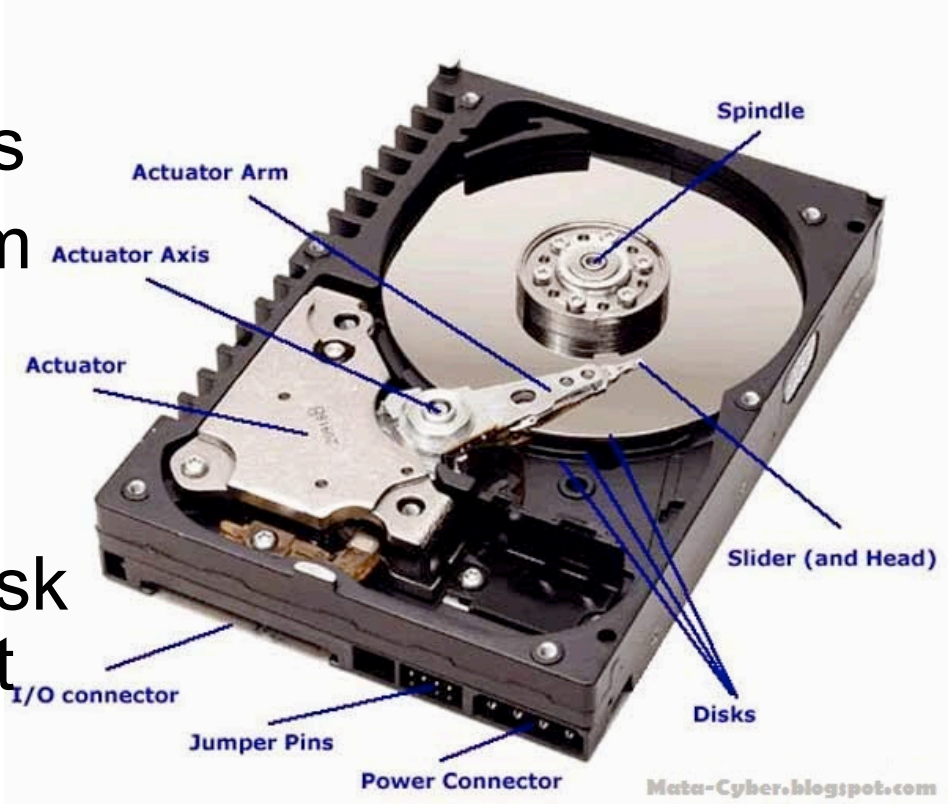
Magnetic Media

- Early magnetic media included
  - Rotating drums
    - Problem: 1) need multiple magnetic heads, 2) these are space inefficient since the interior of the drum is wasted space.
  - Magnetic tapes (including cassette tapes)
    - Problem: good for sequential access but very slow for random access



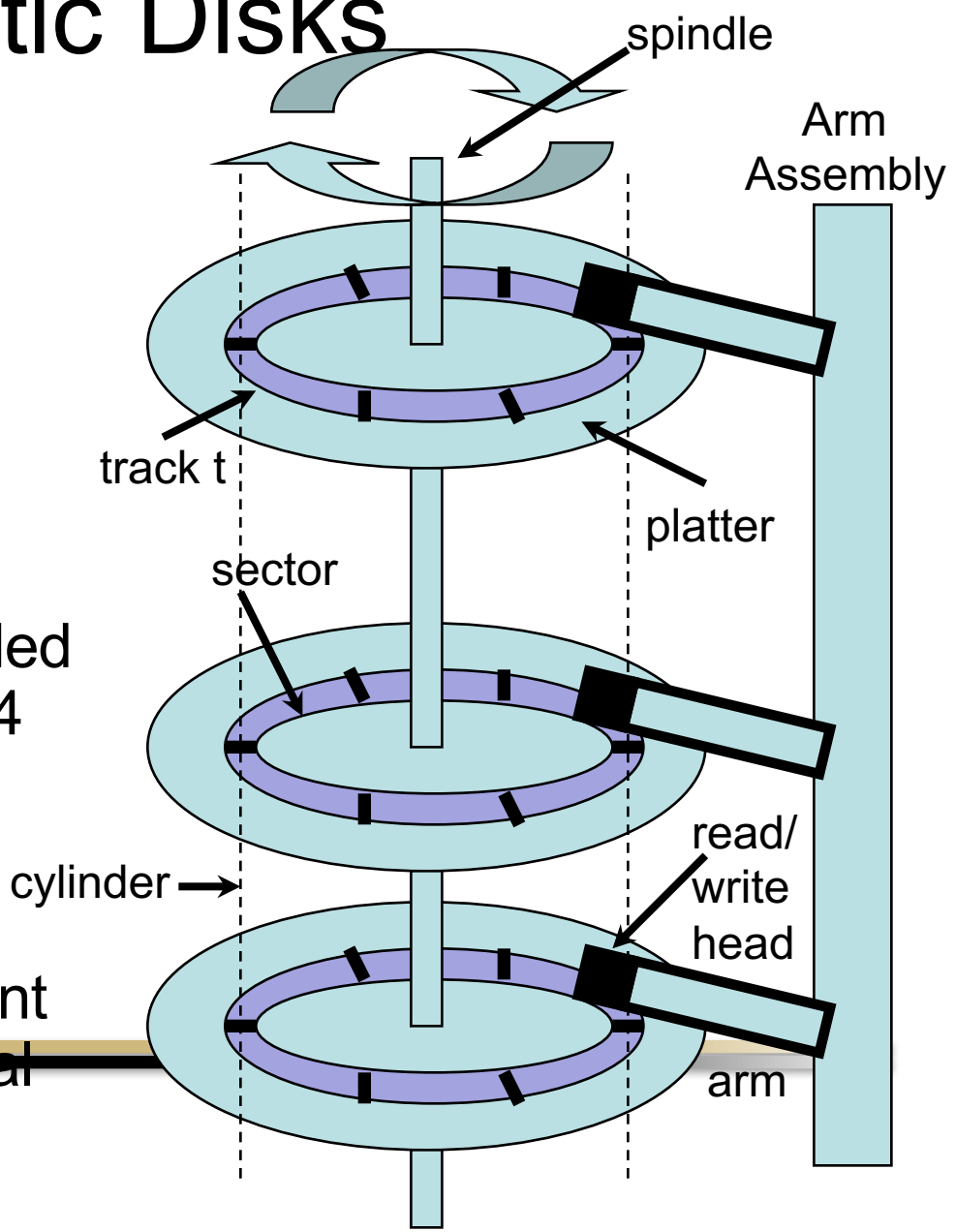
Magnetic Disks

- More space efficient than magnetic drums
- Support both random and sequential access, unlike magnetic tapes
- Array of magnetic disk platters increases bit storage at marginal cost in space and new disk heads



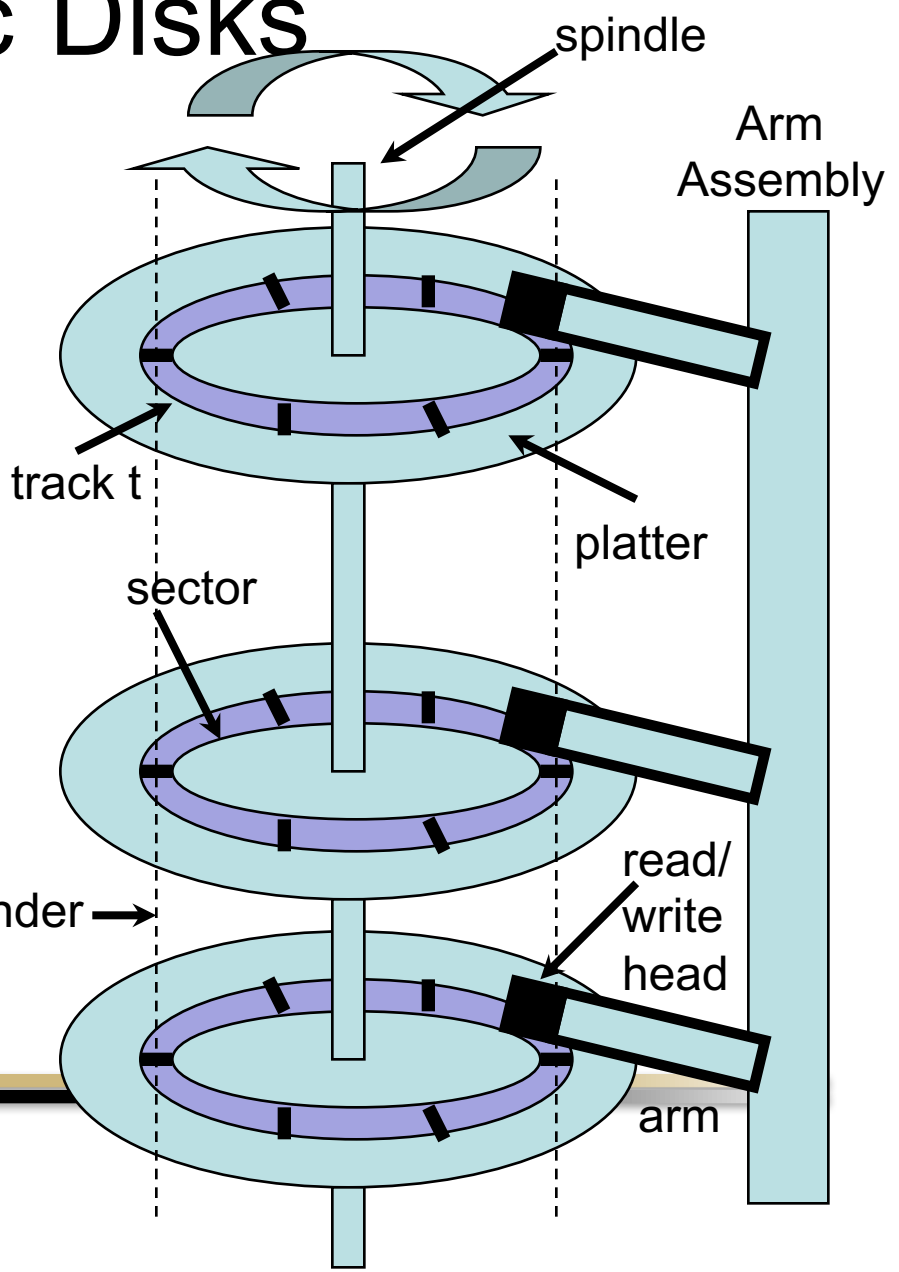
Magnetic Disks

- disk consists of an array of magnetic *platters*
  - each platter is subdivided into concentric *tracks*
  - each track is subdivided into *sectors* (512 B - 4 KB)
  - the collection of the same track (same radius) across different platters forms a logical *cylinder*



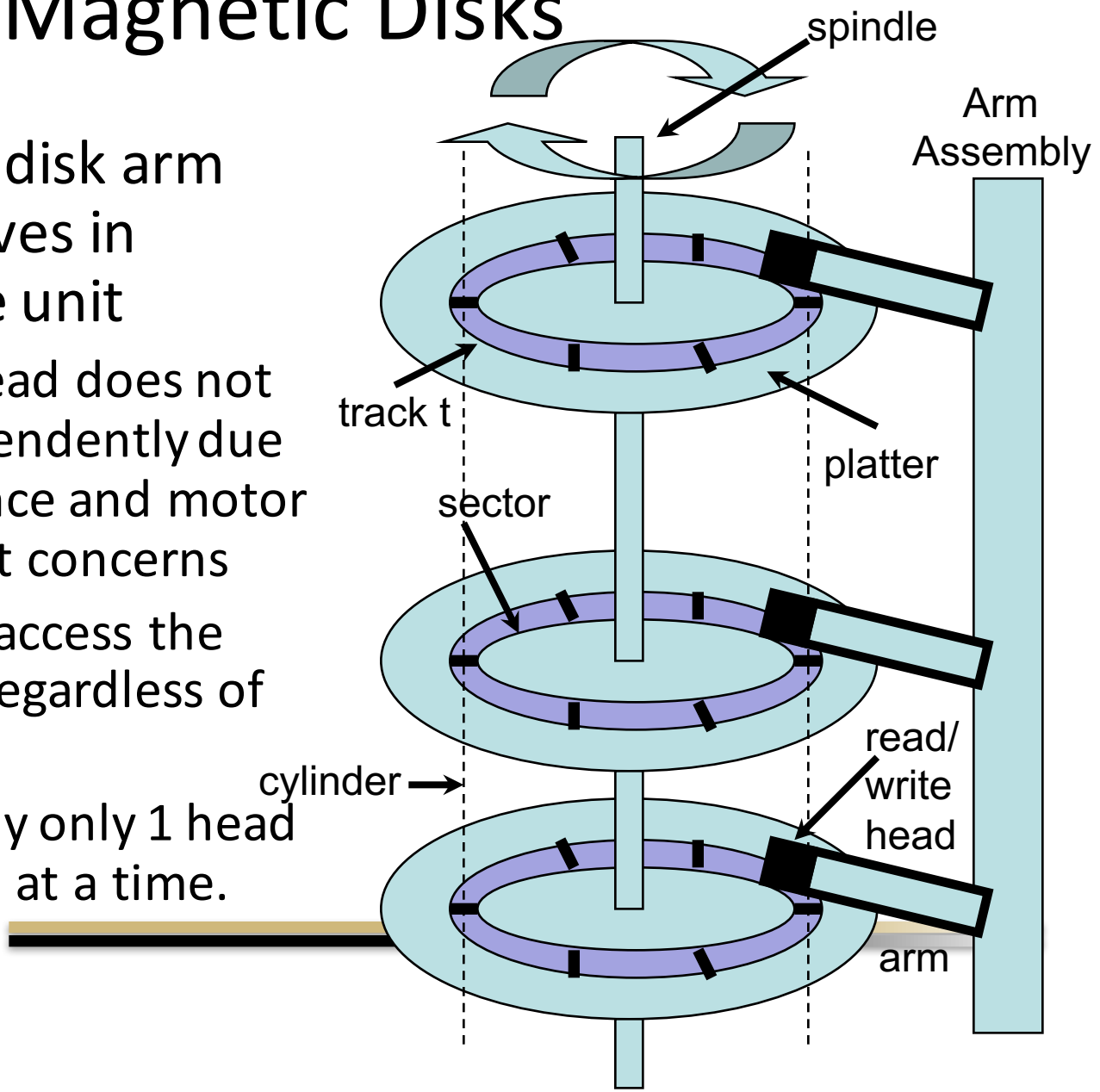
Magnetic Disks

- reading/writing from/to disk consists of
  - mechanically moving the arm to position the read/write head to the correct track
    - this delay is called the *seek time*
  - Mechanically rotating the disk so that the correct sector rotates under the R/W head
    - this is called *rotational latency*



Magnetic Disks

- Typically, the disk arm assembly moves in unison as one unit
  - Each R/W head does not move independently due to interference and motor actuator cost concerns
  - so all heads access the same track regardless of platter
  - Also, typically only 1 head reads/writes at a time.



Disk Access Latency

total delay to read/write from/to disk = seek time + rotational latency + transfer time

typically 5-10 ms

typically 1-5 ms, typical RPM is ~10000 revolutions per minute, so if on average it takes half a revolution to rotate to the right sector, then 0.5/(10000/60) ~= 3 ms

typically in 10-100 μs, typical 1 Gb/s data transfer rate, so retrieving 10 KB of file data = 80000/(1 Gb) = .08 ms

- thus, total disk access delay is often dominated by seek times and rotational latency
  - any technique that can reduce seek times and rotational latency is a big win in terms of improving disk access times
  - *disk scheduling* is designed to reduce seek times and rotational latency

Disk Scheduling

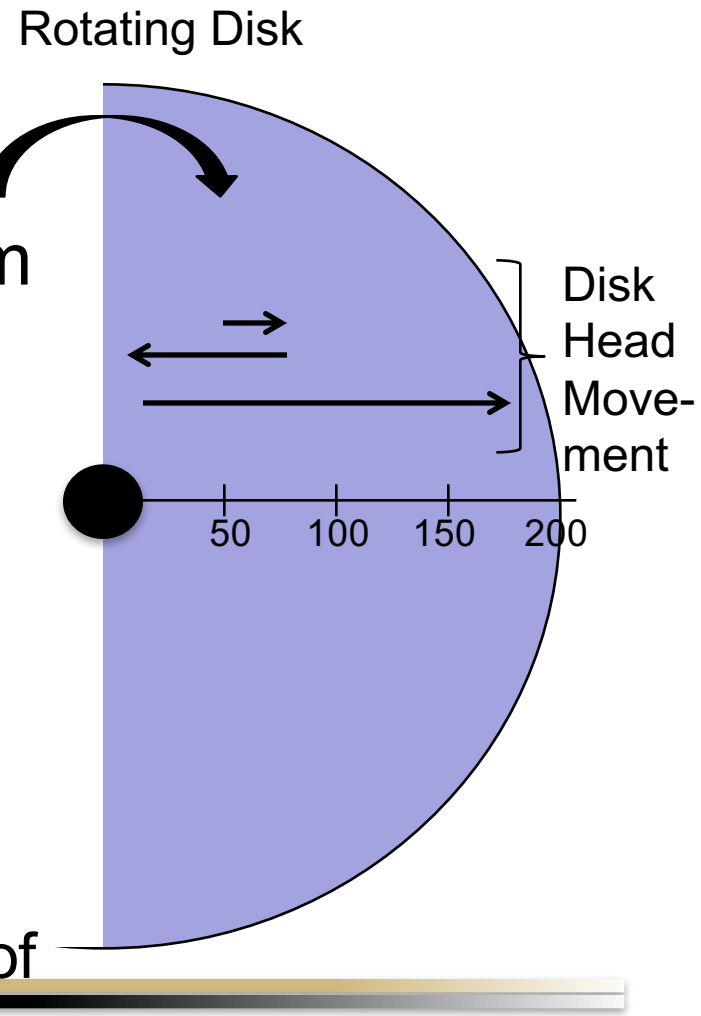
- The OS receives from various processes a sequence of reads and writes from/to disk
  - at any given time, these are stored in a queue
  - the OS can choose to intelligently serve or schedule these requests so as to minimize latency
  - suppose we are given a series of disk I/O requests for different disk blocks that reside on the following different cylinder/tracks in the following order:
    - 98, 183, 37, 122, 14, 124, 65, 67
    - Our goal: find an algorithm that minimizes seek time…

FCFS Disk Scheduling

- Schedule disk reads/writes in the order of their arrival
- Given the previous example, cylinders would be scheduled in the same order as the order of arrival
  - 98, 183, 37, 122, 14, 124, 65, 67
- Let the R/W head be initially at track 53
- then the total number of cylinders/tracks traversed by the disk head under FCFS would be |53-98| + |98-183| + |183-37| + |37-122| + ... = 640 cylinders
- Observation: disk R/W head would move less if 37 and 14 could be serviced together, similarly for 122 and 124
- Easy to implement, and no starvation, but can be very slow

SSTF Disk Scheduling

- SSTF (Shortest-Seek-Time-First) Scheduling selects the next request with the minimum seek time from the current R/W head position
  - R/W head starts @ 53, and has requests 98, 183, 37, 122, 14, 124, 65, 67
  - SSTF services requests as 65, 67, 37 (closer than 98), 14, 98, 122, 124, 183
  - 236 cylinders traversed = ~1/3 of FCFS!



SSTF Disk Scheduling

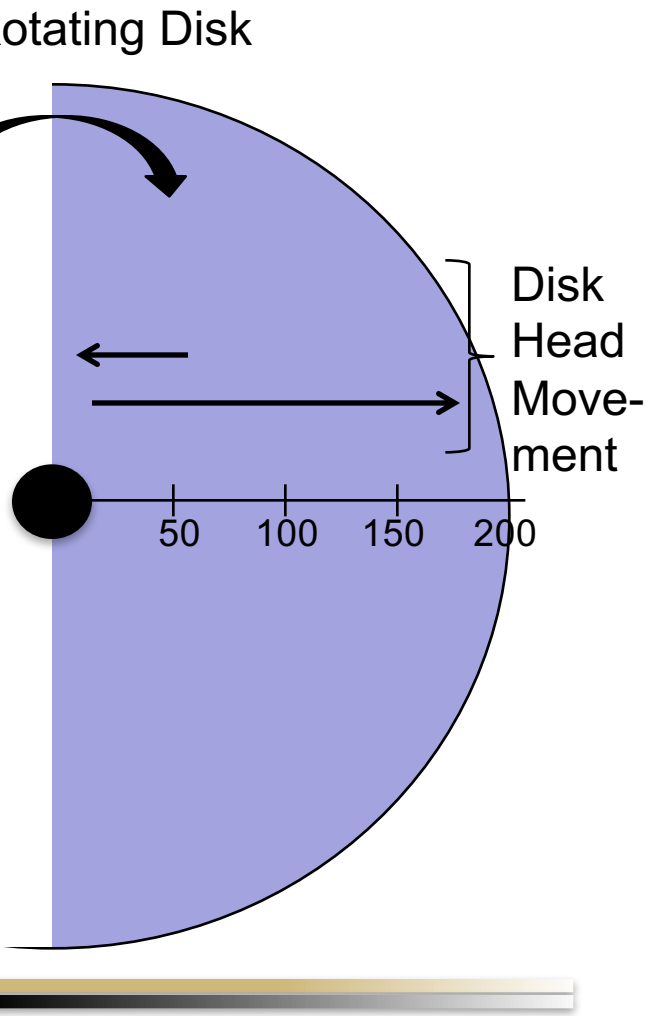
- Locally optimal, but why not globally optimal?
  - still not optimal - better to move disk head from 53 to 37 then 14 then 65 rather than 65 directly, because this would result in 208 cylinders
- Another drawback: starvation
  - while the disk head is positioned at 37, a series of new nearby requests may come into the queue for 39, 35, 38, ...
    - SSTF keeps servicing nearby requests, letting far away requests starve
  - similar to shortest job first process scheduling, which suffers the same problem of starvation

OPT Disk Scheduling

- An optimal approach would be to minimize back-and-forth movement of the disk head
  - Given a set of tracks, there will be two tracks that are the "edge" tracks (innermost and outermost)
  - Move the disk head from the current position to the closest edge track, and sweep through once (either innermost to outermost, or outermost to innermost)
  - this should minimize the overlap of back and forth and give the minimum # of tracks covered
    - There's only one overlap where the head revisits tracks, and it is of the smallest size

OPT Disk Scheduling

- Let initial direction be up from current head position = 53,
  - assume requests, 98, 183, 37, 122, 14, 124, 65, 67
- OPT services requests as follows:
  - 53→37→14→65→67→98→122→124→183
    - OPT thus ignores the initial direction and changes direction if need be
  - total # cylinders traversed = 39 down + 169 up = 208 cylinders



OPT Disk Scheduling

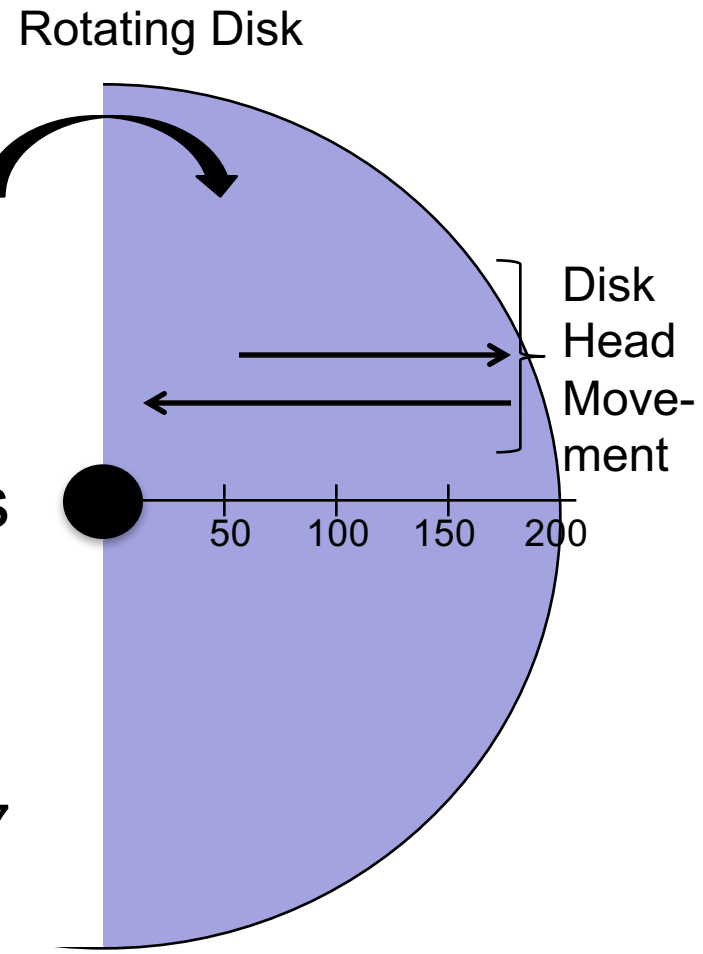
- The problem with OPT is that you have to recalculate every time there's a new request for a disk track
  - This can lead to disk head movement that moves back and forth, so though the algorithm is locally/piecewise optimal, it is no longer globally optimal
  - Moreover, recalculating the local optimum causes back and forth disk head movement could conceivably starve certain disk requests
- Best approximation to OPT is probably to just scan the disk in one direction, and then change the direction of scanning

SCAN Disk Scheduling

- disk R/W head moves in one direction from innermost to outermost track, then reverses direction, then reverses..., sweeping across the disk in alternate directions
  - analogous to an elevator that goes up all the way to the top floor, then goes down all the way to the bottom floor, then goes all the way up to the top,...
  - Advantages:
    - Approximates OPT
    - Starvation free solution
    - handles dynamic arrival of new requests
    - simple to implement

SCAN Disk Scheduling

- given initial direction is up from current head position = 53, and given 200 tracks, and given requests, 98, 183, 37, 122, 14, 124, 65, 67, then SCAN would service requests in the following order:
  - 53→65→67→98→122→124→183→200→37→14
  - total # cylinders traversed = 147 up + 186 down = 333 cylinders



SCAN Disk Scheduling

- Problem 1: SCAN only approximates OPT, and in a local time window may not be accurate
  - If initial direction were down from 53, SCAN = OPT
  - Since initial direction is up, SCAN winds up sweeping through many more tracks than OPT
    - There is significantly more overlap, as shown by the arrows for disk head movement, compared to the arrows in the OPT figure
- Problem 2: unnecessarily goes to extreme edges of disk even if no requests there
  - We'll see a solution to this later…

SCAN Disk Scheduling

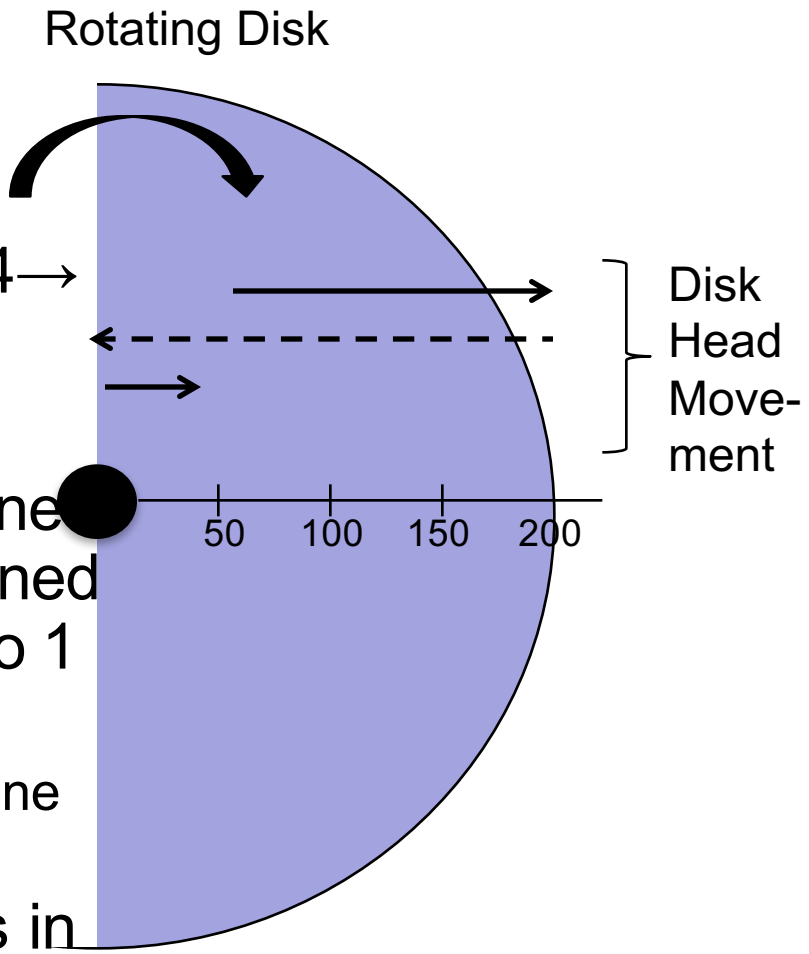
- Problem 3: it is unfair to the tracks on the edges of the disk (both inside and outside)
  - Writes to the edge tracks take the longest since after SCAN has reversed directions at one edge, the writes on the other edge always wait the longest to be served
  - After two full scans back and forth, middle tracks get serviced twice, edge tracks only once

C-SCAN Disk Scheduling

- Solution to Problem 3: treat disk as a circular queue of tracks
  - so when reaching one edge, move the R/W head all the way back to the other edge
  - and continue scanning in the same direction rather than reversing direction
  - this is called circular SCAN, or C-SCAN
  - After reaching an edge, writes at the opposite edge get served first – this is more fair

C-SCAN Disk Scheduling

- In the example, C-SCAN would scan:
  - 53→65→67→98→122→124→ 183→200→1→14→37
  - for a total of 384 tracks
  - Note how no scanning is done as the disk head is repositioned from 200 all the way down to 1 to ensure circularity
    - This is shown by the dashed line in the figure
  - Hence scanning only occurs in 1 direction



LOOK Disk Scheduling

- Solution to SCAN's Problem 2: don't travel to the extreme edge if no requests there
  - so look at the furthest request in the current direction and only move the disk head that far, then reverse direction
  - This is called the LOOK disk scheduling algorithm
  - No starvation in this algorithm either
- In the example, then LOOK would service requests in the following order:
  - 53→65→67→98→122→124→183→37→14
  - total # cylinders traversed = 130 up + 169 down = 299 cylinders

C-LOOK Disk Scheduling

- Improve LOOK by making it more fair, i.e. circular LOOK or C-LOOK
  - after only moving disk head to furthest request in current direction, then move disk head directly all the way back to the further request in the other direction, continue scanning in the same direction
  - C-LOOK would service requests in the following order:
    - 53→65→67→98→122→124→183→14→37
    - total # cylinders traversed = 130 up + 169 down + 23 up = 322 cylinders

Disk Scheduling in Practice

- In the previous examples, the # of cylinders traversed by SCAN, LOOK, and C-LOOK are much lower if the initial direction was down
  - We can't keep changing the direction, or can get starvation/unfairness. Preserving directionality of the scan is important for fairness, but we sacrifice some optimality/performance
- Either SSTF or LOOK are reasonable choices as default disk scheduling algorithms
- These algorithms for reducing seek times are typically embedded in the disk controller today
- There are other algorithms for reducing rotational latency, also embedded in the disk controller

File Layout and Disk Scheduling

- Linked (FAT) or indexed allocation schemes can spread a file's data across the disk, increasing seek times, while contiguous allocation minimizes seek times
  - When a file needs to allocate a data block, try to find a free sector that is close to where the rest of the file is laid out on disk, i.e. cluster your file on disk
- Opening a file for the first time requires searching the disk for the directory, file header, and file data – if these are all spread out, then seek times are large
  - Could store directories in the middle tracks, so at most half the disk is traversed to find the file header and data
  - Could store the file header near the file data, or near the directory

File Layout and Disk Scheduling

- Which disk block is chosen to allocate if there are many available free blocks?
  - Linux ext3fs tries to cluster the location and layout of disk blocks for file data around the file header/inode of the file. This will minimize mechanical disk latency.
  - Similarly, ext3fs tries to locate file inodes near their parent directory on disk
  - However, ext3fs spreads high-level parent directories evenly around disk to minimize fragmentation in any 1 area of disk
    - If too many directories and their references file headers are clustered in same place, then there's not enough room for the file data too, causing fragmentation of files spread across disk
    - Instead, subdivide directory hierarchy into pieces at a directory granularity. All file headers for a particular directory are clustered together. Also there should be enough space to allocate the data close to the associated file headers.
    - Different directories are spread across the disk.