

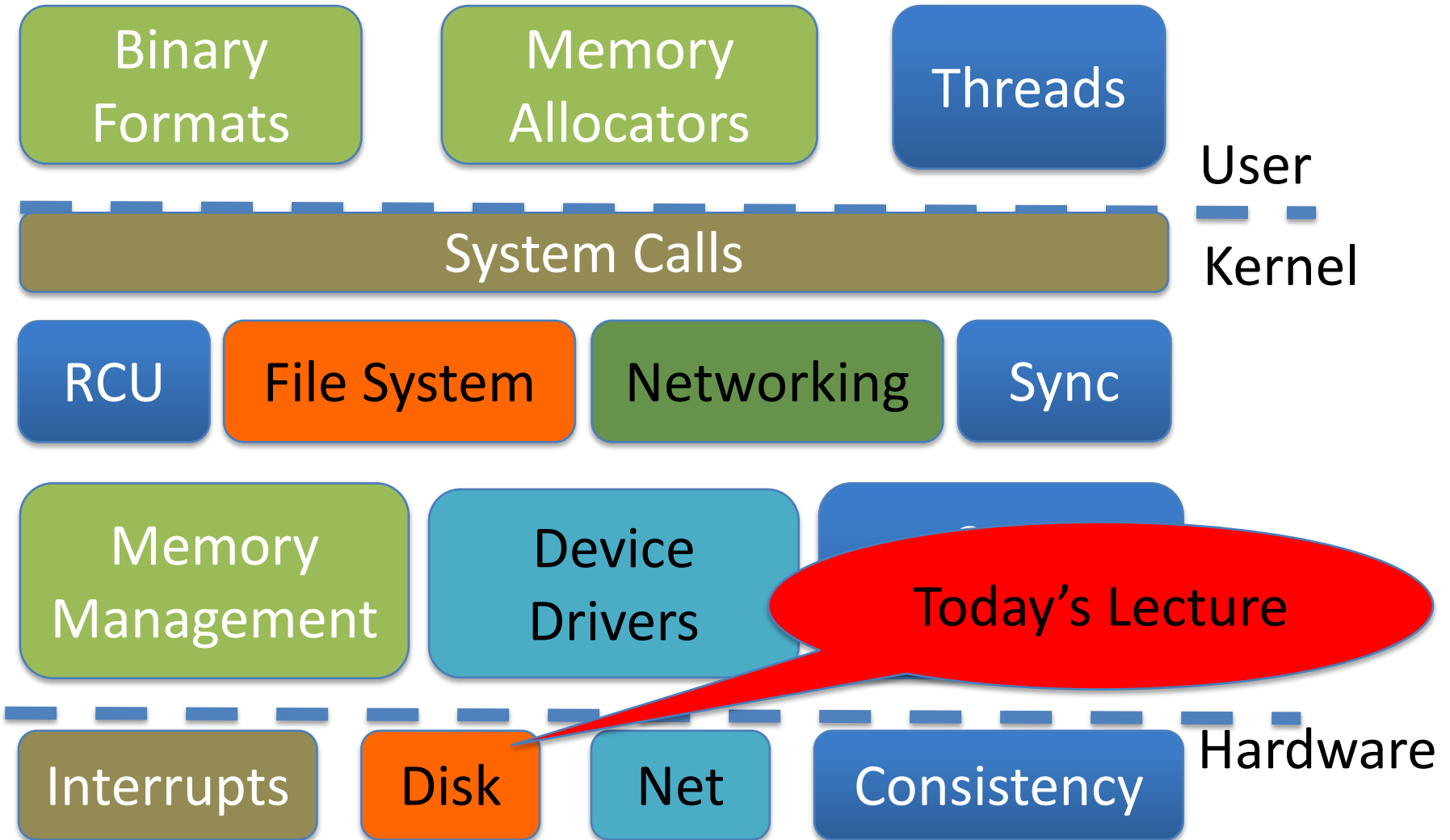


Block Device Scheduling

Don Porter



Logical Diagram





Quick Recap

- CPU Scheduling
 - Balance competing concerns with heuristics
 - What were some goals?
 - No perfect solution
- Today: Block device scheduling
 - How different from the CPU?
 - Focus primarily on a traditional hard drive
 - Extend to new storage media

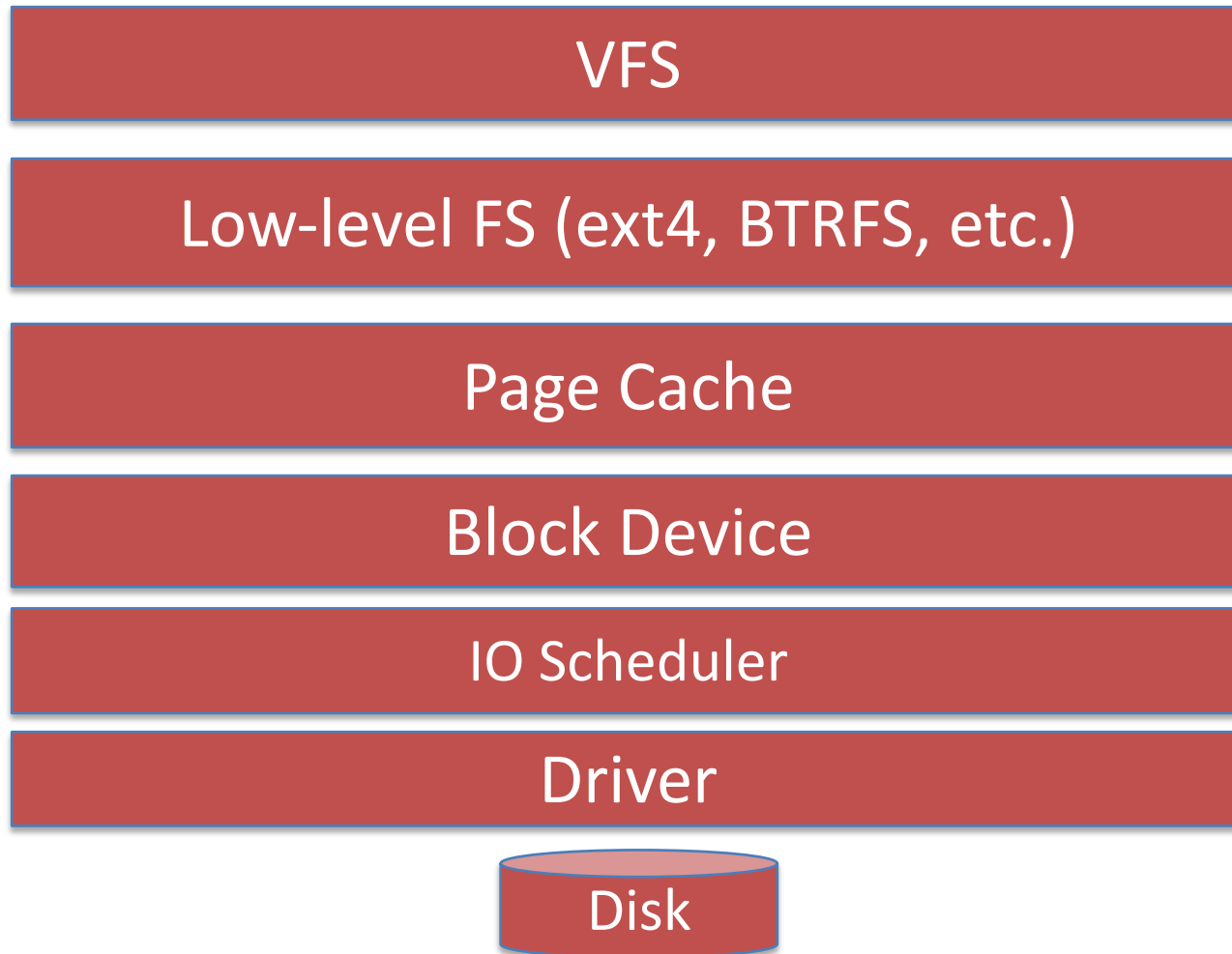


Block device goals

- Throughput
- Latency
- Safety – file system can be recovered after a crash
- Fairness – surprisingly, very little attention is given to storage access fairness
 - Hard problem – solutions usually just prevent starvation
 - Disk quotas for space fairness



Big Picture





OS Model of a Block Dev.

- Simple array of sectors
 - Sectors are usually 512 or 4k bytes

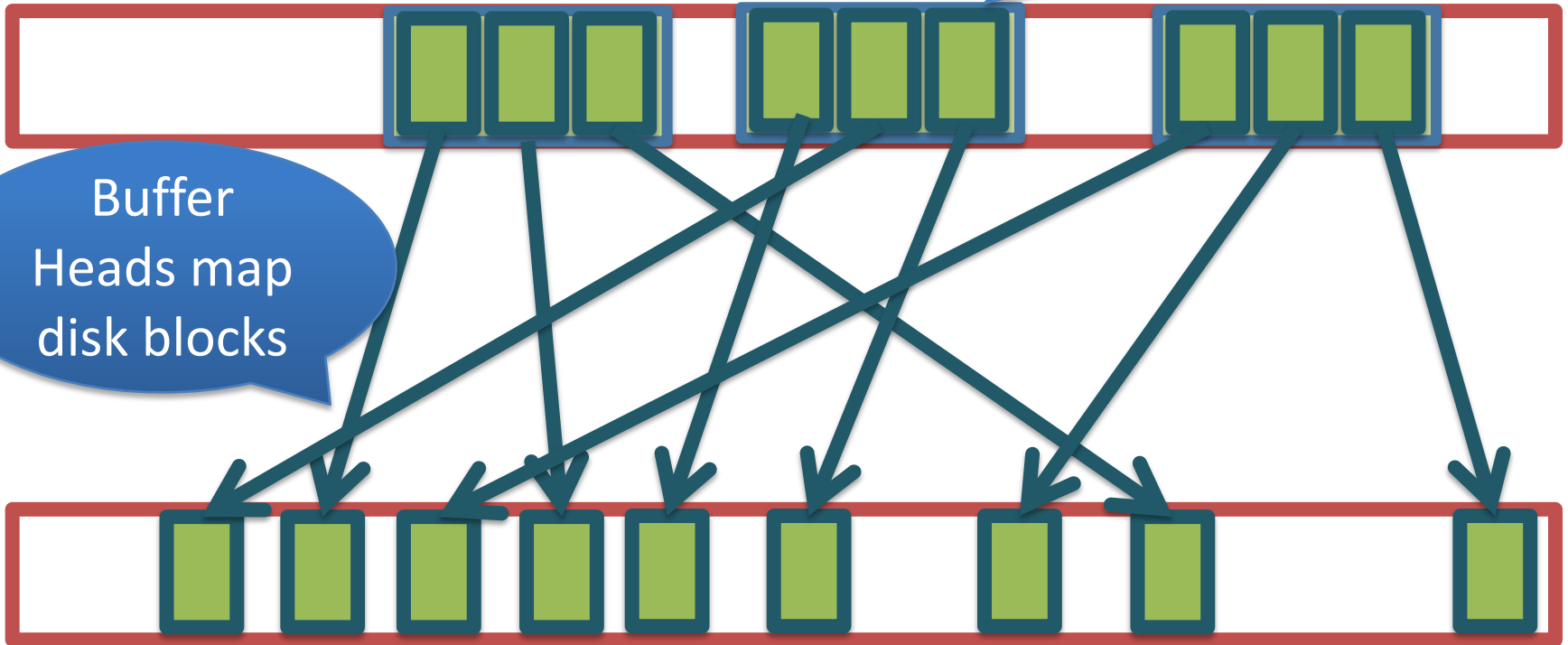


Recall: Page Cache

Page Cache

Page (blue) w/ 3
buffer heads (green)

Buffer
Heads map
disk blocks



Block Dev



Caching

- Obviously, the number 1 trick in the OS designer's toolbox is caching disk contents in RAM
 - Remember the page cache?
- Latency – can be hidden by pre-reading data into RAM
 - And keeping any free RAM full of disk contents
 - Doesn't help synchronous reads (that miss in RAM cache) or synchronous writes



Caching + throughput

- Assume that most reads and writes to disk are asynchronous
 - Dirty data can be buffered and written at OS's leisure
 - Most reads hit in RAM cache – most disk reads are read-ahead optimizations
- Key problem: How to optimally order pending disk I/O requests?
 - Hint: it isn't first-come, first-served



Another view of the problem

- Between page cache and disk, you have a queue of pending requests
- Requests are a tuple of (block #, read/write, buffer addr)
- You can reorder these as you like to improve throughput
- What reordering heuristic to use? If any?
- Heuristic is called the **IO Scheduler**



A simple disk model

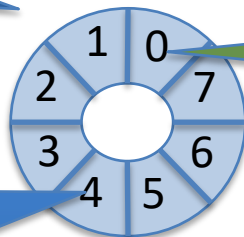
- Disks are slow. Why?
 - Moving parts \ll circuits
- Programming interface: simple array of sectors (blocks)
- Physical layout:
 - Concentric circular “tracks” of blocks on a platter
 - E.g., sectors 0-9 on innermost track, 10-19 on next track, etc.
 - Disk arm moves between tracks
 - Platter rotates under disk head to align w/ requested sector



Disk Model

Each block on
a sector

Disk spins at a
constant speed.
Sectors rotate
underneath head.

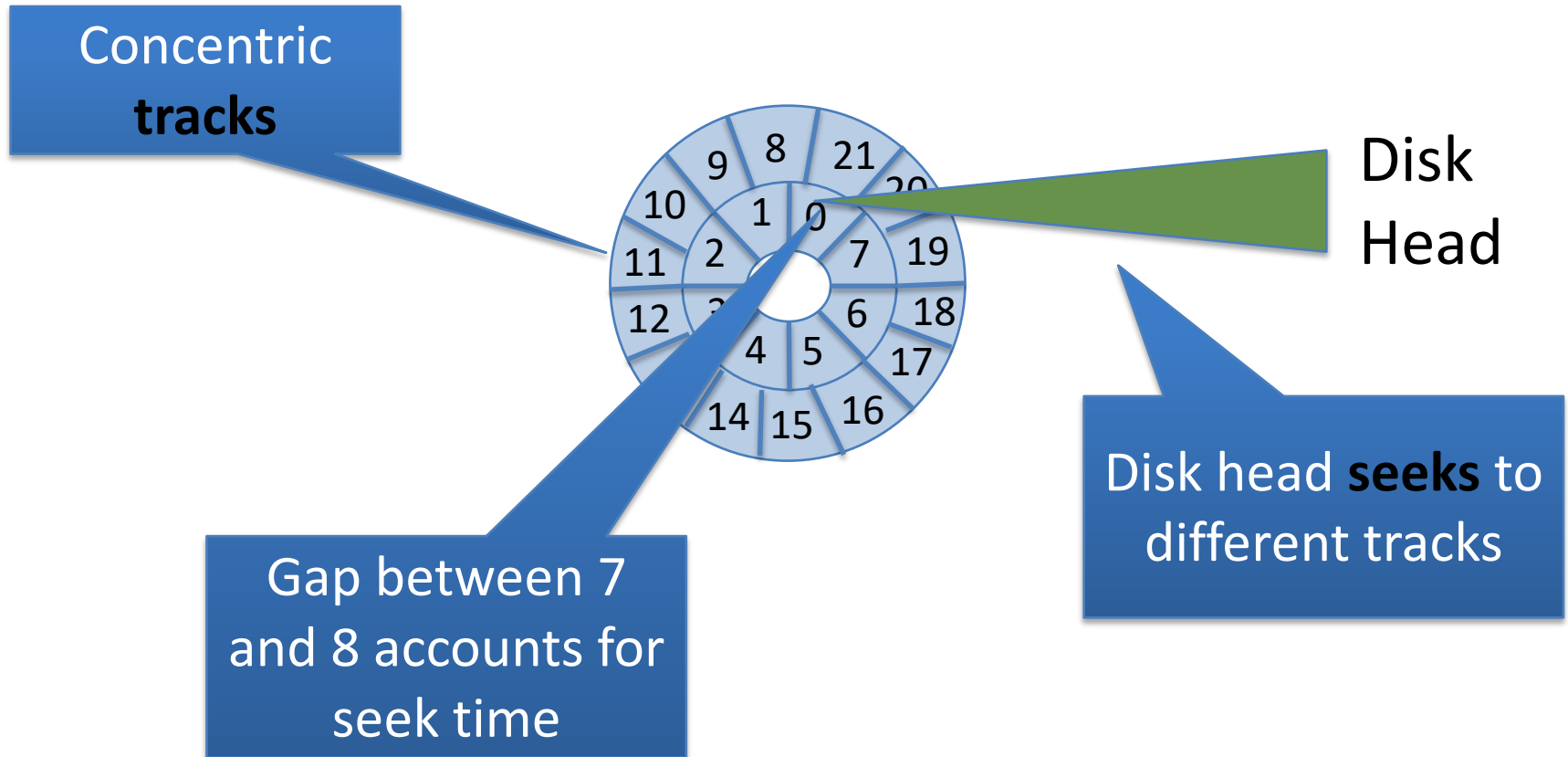


Disk
Head

Disk Head
reads at
granularity of
entire sector

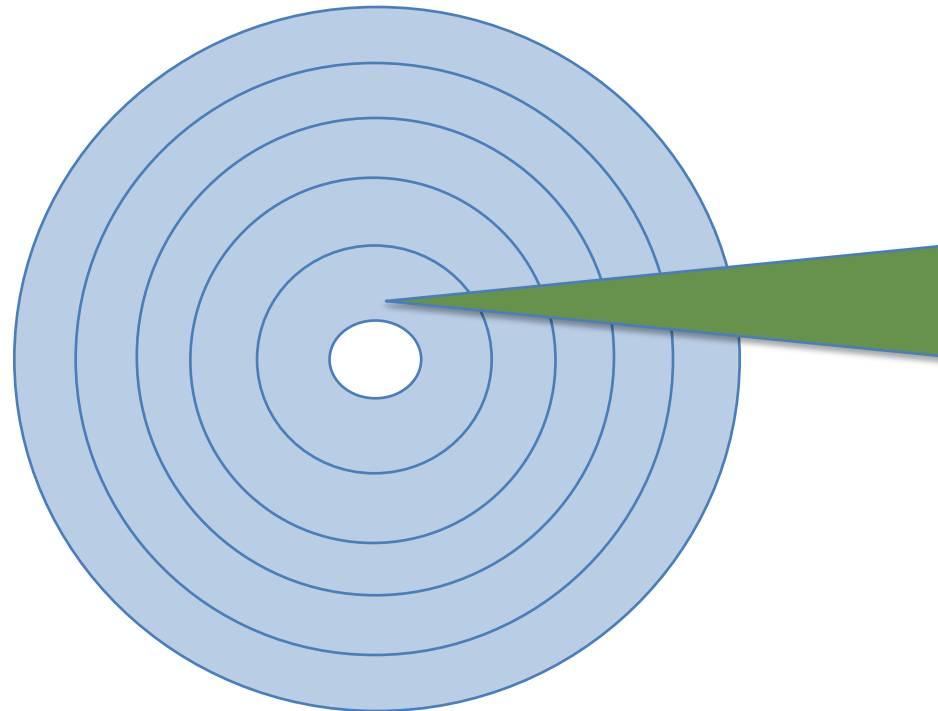


Disk Model





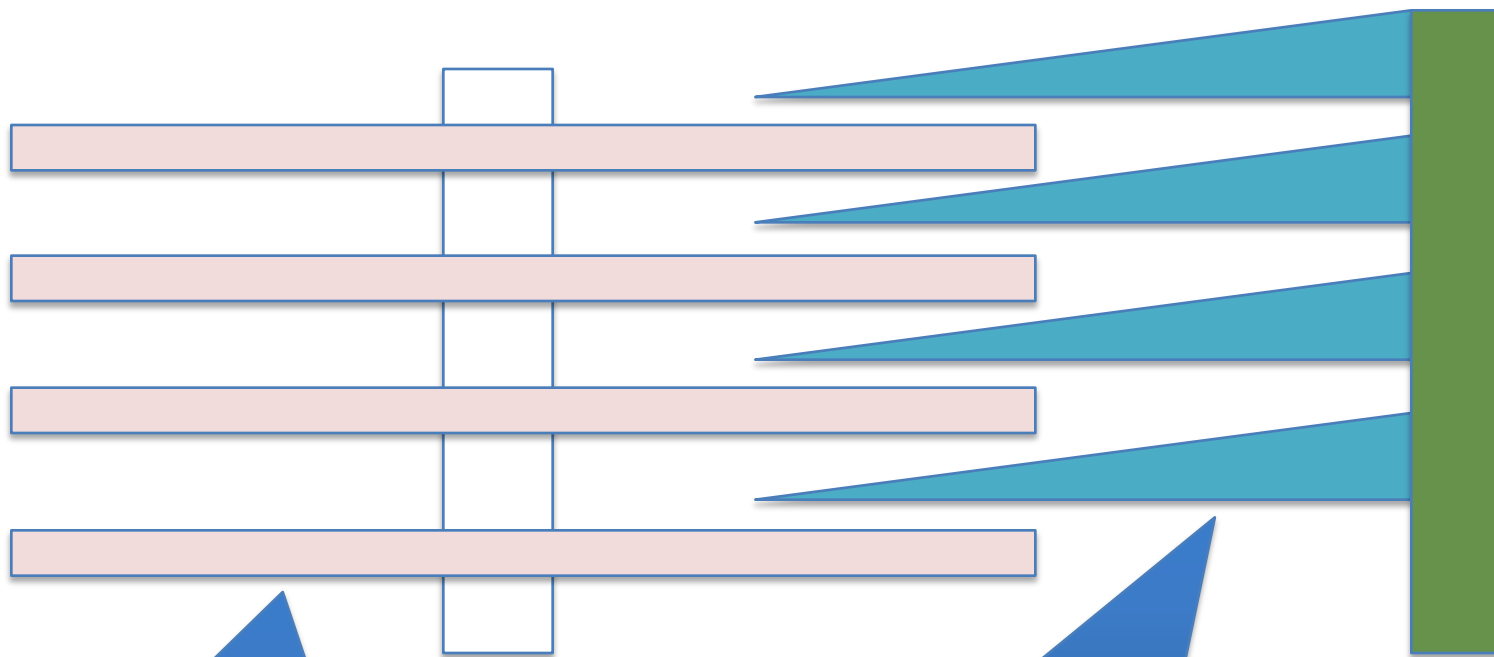
Many Tracks



Disk
Head



Several (~4) Platters



Platters spin together at same speed

Each platter has a head;
All heads seek together



Implications of multiple platters

- Blocks actually striped across platters
- Example:
 - Sector 0 on platter 0
 - Sector 1 on platter 1 at same position
 - Sector 2 on platter 2, Sec. 3 on Plat. 3 also at same position
 - 4 heads can read all 4 sectors simultaneously



3 key latencies

- I/O delay: time it takes to read/write a sector
- Rotational delay: time the disk head waits for the platter to rotate desired sector under it
 - Note: disk rotates continuously at constant speed
- Seek delay: time the disk arm takes to move to a different track



Observations

- Latency of a given operation is a function of current disk arm and platter position
- Each request changes these values
- Idea: build a model of the disk
 - Maybe use delay values from measurement or manuals
 - Use simple math to evaluate latency of each pending request
 - Greedy algorithm: always select lowest latency



Example formula

- s = seek latency, in time/track
- r = rotational latency, in time/sector
- i = I/O latency, in seconds

- $\text{Time} = (\Delta\text{tracks} * s) + (\Delta\text{sectors} * r) + i$
- Note: $\Delta\text{sectors}$ must factor in position after seek is finished. Why?



Problem with greedy?

- “Far” requests will starve
- Disk head may just hover around the “middle” tracks



Elevator Algorithm

- Require disk arm to move in continuous “sweeps” in and out
- Reorder requests within a sweep
 - Ex: If disk arm is moving “out,” reorder requests between the current track and the outside of disk in ascending order (by block number)
 - A request for a sector the arm has already passed must be ordered after the outermost request, in descending order



Elevator Algo, pt. 2

- This approach prevents starvation
 - Sectors at “inside” or “outside” get service after a bounded time
- Reasonably good throughput
 - Sort requests to minimize seek latency
 - Can get hit with rotational latency pathologies (How?)
- Simple to code up!
 - Programming model hides low-level details; difficult to do fine-grained optimizations in practice



Modular Schedulers

- Linux allows the disk scheduler to be replaced
 - Just like the CPU scheduler
- Can choose a different heuristic that favors:
 - Fairness
 - Real-time constraints
 - Performance



Complete Fairness Queue (CFQ)

- Idea: Add a second layer of queues (one per process)
 - Round-robin promote them to the “real” queue
- Goal: Fairly distribute disk bandwidth among tasks
- Problems?
 - Overall throughput likely reduced
 - Ping-pong disk head around



Deadline Scheduler

- Associate expiration times with requests
- As requests get close to expiration, make sure they are deployed
 - Constrains reordering to ensure some forward progress
- Good for real-time applications



Anticipatory Scheduler

- Idea: Try to anticipate locality of requests
 - If process P tends to issue bursts of requests for close disk blocks,
 - When you see a request from P, hold the request in the disk queue for a while
 - See if more “nearby” requests come in
 - Then schedule all the requests at once
 - And coalesce adjacent requests



Optimizations at Cross-purposes

- The disk itself does some optimizations:
 - Caching
 - Write requests can sit in a volatile cache for longer than expected
 - Reordering requests internally
 - Can't assume that requests are serviced in-order
 - Dependent operations must wait until first finishes
 - Bad sectors can be remapped to “spares”
 - Problem: disk arm flailing on an old disk



A note on safety

- In Linux, and other OSes, the I/O scheduler can reorder requests arbitrarily
- It is the file system's job to keep unsafe I/O requests out of the scheduling queues
 - Or issue barriers in the queue



Dangerous I/Os

- What can make an I/O request unsafe?
 - File system bookkeeping has invariants on disk
 - Example: Inodes point to file data blocks; data blocks are also marked as free in a bitmap
 - Updates must uphold these invariants
 - Ex: Write an update to the inode, then the bitmap
 - What if the system crashes between writes?
 - Block can end up in two files!!!



3 Simple Rules

(Courtesy of Ganger and McKusick, “Soft Updates” paper)

- Never write a pointer to a structure until it has been initialized
 - Ex: Don’t write a directory entry to disk until the inode has been written to disk
- Never reuse a resource before nullifying all pointers to it
 - Ex: Before re-allocating a block to a file, write an update to the inode that references it
- Never reset the last pointer to a live resource before a new pointer has been set
 - Ex: Renaming a file – write the new directory entry before the old one (better 2 links than none)



A note on safety

- It is the file system's job to keep unsafe I/O requests out of the scheduling queues
- While these constraints are simple, enforcing them in the average file system is surprisingly difficult
 - Journaling helps by creating a log of what you are in the middle of doing, which can be replayed
 - (Simpler) Constraint: Journal updates must go to disk before FS updates



Disks aren't everything

- Flash is increasing in popularity
 - Different types with slight variations (NAND, NOR, etc)
- No moving parts – who cares about block ordering anymore?
- Can only write to a block of flash ~100k times
 - Can read as much as you want



More in a Flash

- Flash reads are generally fast, writes are more expensive
- Prefetching has little benefit
- Queuing optimizations can take longer than a read
- New issue: wear leveling – need to evenly distribute writes
 - Flash devices usually have a custom, log-structured FS
 - Group random writes



Even newer hotness

- Byte-addressible, persistent RAMs (BPRAM)
 - Optane, Phase-Change Memory (PCM), Memristors, etc.
- Splits the difference between RAM and flash:
 - Byte-granularity writes (vs. blocks)
 - Fast reads, slower, high-energy writes
 - Doesn't need energy to hold state (DRAM refresh)
 - Wear an issue (bytes get stuck at last value)
- Intel shipping first Optane prototypes, more to come...



Important research topic

- Most work on optimizing storage accessed is tailored to hard drives
- These heuristics are not easily adapted to new media
- Future systems will have a mix of disks, flash, PRAM, DRAM
- Does it even make sense to treat them all the same?



Summary

- Performance characteristics of disks, flash, BPRAM
- Disk scheduling heuristics
- Safety constraints for file systems