CSE 506: Operating Systems

Disk Scheduling
Key to Disk Performance

- Don’t access the disk
  - Whenever possible

- Cache contents in memory
  - Most accesses hit in the block cache

- **Prefetch** blocks into block cache (a.k.a. *read-ahead*)
  - When OS accesses disk, get next few blocks too
    - Keep them in block cache
  - If access hits on prefetched block
    - Read the next few blocks in the background
  - Avoids *demand* access to the disk
Caching + Throughput

• Most reads and writes to disk are asynchronous
  – Dirty data can be buffered and written at OS’s leisure
  – Most reads hit in block cache – read-ahead works

• How to optimally order pending disk I/O requests?
  – Hint: it isn’t first-come, first-served
Another view of disk accesses

• Between block cache and disk, there is a queue
  – All disk requests wait in this queue
  – Requests are a tuple of (block #, read/write, buffer addr)

• Requests can be reordered
  – To achieve best performance across all requests

• What reordering heuristic to use? If any?
  – Heuristic is called the **IO Scheduler**
A simple disk model

• Disks are slow
  – Moving parts much slower than circuits
  – Flash storage is faster
    • Still multiple orders of magnitude slower than memory

• Programming interface: simple array of sectors (blocks)

• Physical disk layout:
  – Concentric “cylinders” of blocks on a platter
    • Two tracks, one on each side
  – E.g., sectors 0-9 on innermost track, 10-19 on next track, etc.
  – Disk arm (with heads attached) moves between tracks
  – Platter rotates under disk head to align w/ requested sector
Each block on a sector

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

Disk Head reads at granularity of entire sector
Disk Model

Concentric tracks

Gap between 7 and 8 accounts for seek time

Disk head seeks to different tracks

Gap between 7 and 8 accounts for seek time
Many Tracks

Disk Arm
Several (~4) Platters

- Platters spin together at same speed
- Each platter has two heads; All heads seek together on arm
3 key latencies

- I/O delay: Time to read/write a sector
- Rotational delay: Time for track to rotate under head
  - Note: disk rotates continuously at constant speed
- Seek delay: Time to move disk arm to cylinder
Greedy IO Scheduler

• Op. latency is function of arm and cylinder position
• Each request changes these values
• Idea: build a model of the disk
  – Use delay values from measurement or manuals
  – Use math to evaluate latency of each pending request
  – Greedy algorithm: always select lowest latency
Problem with Greedy?

• “Far” requests will starve
• Disk head may just hover around the “middle” tracks
Elevator Algorithms (SCAN)

• Arm moves in continuous “sweeps” in and out
  – Reorder requests within a sweep
    • Closest block in direction of travel is next to be read
    • Request that was just passed has to wait for sweep to return

• Prevents starvation
  – Sectors “inside” or “outside” serviced after bounded time

• Reasonably good throughput
  – Sort requests to minimize seek latency

• Simple to code
  – Programming model hides low-level details
Elevator Algorithms (C-SCAN)

• SCAN is not fair
  – Cylinders in the middle get serviced ~twice as often
    • Likely to be handled when arm travels in either direction

• Only perform ops when moving in one direction
  – Once the end is reached, quickly go to the beginning

• More fair
  – But probably lower average performance
Pluggable 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
- Prioritize requests closer to expiration
  - Constrains reordering to ensure forward progress
- Good for real-time applications
Anticipatory Scheduler

• Idea: Try to anticipate locality of requests
• If process P issue bursts of requests for close blocks
  – If a request from P arrives
    • Hold request in queue for a while
    • Hope that more “nearby” requests come in
  – Eventually, schedule all pending requests at once
    • Coalesce adjacent requests
Optimizations at Cross-purposes

- The disk itself does some optimizations
  - Caching
    - Disks have their own caches
    - And do their own read-ahead
  - Reordering requests internally
    - Disk protocols (e.g., SATA) allow many outstanding commands
      - Can’t assume that requests are serviced in-order
  - Bad sectors can be remapped to “spares”
    - Problem: disk arm flailing on an old disk