Anatomy of a disk

- **Stack of magnetic platters**
  - Rotate together on a central spindle @3,600-15,000 RPM
  - Drives speed drifts slowly over time
  - Can’t predict rotational position after 100-200 revolutions

- **Disk arm assembly**
  - Arms rotate around pivot, all move together
  - Pivot offers some resistance to linear shocks
  - Arms contain disk heads—one for each recording surface
  - Heads read and write data to platters
Storage on a magnetic platter

- Platters divided into concentric *tracks*
- A stack of tracks of fixed radius is a *cylinder*
- Heads record and sense data along cylinders
  - Significant fractions of encoded stream for error correction
- Generally only one head active at a time
  - Disks usually have one set of read-write circuitry
  - Must worry about cross-talk between channels
  - Hard to keep multiple heads exactly aligned
Disk positioning system

- Move head to specific track and keep it there
  - Resist physical socks, imperfect tracks, etc.

- A *seek* consists of up to four phases:
  - *speedup*–accelerate arm to max speed or half way point
  - *coast*–at max speed (for long seeks)
  - *slowdown*–stops arm near destination
  - *settle*–adjusts head to actual desired track

- Very short seeks dominated by settle time (~1 ms)

- Short (200-400 cyl.) seeks dominated by speedup
  - Accelerations of 40g
Seek details

- Head switches comparable to short seeks
  - May also require head adjustment
  - Settles take longer for writes than reads

- Disk keeps table of pivot motor power
  - Maps seek distance to power and time
  - Disk interpolates over entries in table
  - Table set by periodic “thermal recalibration”
  - 500 ms recalibration every 25 min, bad for AV

- “Average seek time” quoted can be many things
  - Time to seek 1/3 disk, 1/3 time to seek whole disk,
Sectors

- **Disk interface presents linear array of sectors**
  - Generally 512 bytes, written atomically

- **Disk maps logical sector #s to physical sectors**
  - *Zoning*—puts more sectors on longer tracks
  - *Track skewing*—sector 0 pos. varies for sequential I/O
  - *Sparing*—flawed sectors remapped elsewhere

- **OS doesn’t know logical to physical sector mapping**
  - Larger logical sector # difference means larger seek
  - Highly non-linear relationship (*and* depends on zone)
  - OS has no info on rotational positions
  - Can empirically build table to estimate times
Disk interface

- Controls hardware, mediates access

- Computer, disk often connected by bus (e.g., SCSI)
  - Multiple devices may contend for bus
  - SCSI devices can disconnect during requests (+200 μs)

- Command queuing: Give disk multiple requests
  - Disk can schedule them using rotational information

- Disk cache used for read-ahead
  - Otherwise, sequential reads would incur whole revolution
  - Cross track boundaries? Can’t stop a head-switch

- Some disks support write caching
  - But data not stable—not suitable for all requests
Scheduling: First come first served (FCFS)

- Process disk requests in the order they are received

- Advantages
  - 
  - 

- Disadvantages
  - 
  - 
Scheduling: First come first served (FCFS)

- Process disk requests in the order they are received

- **Advantages**
  - Easy to implement
  - Good fairness

- **Disadvantages**
  - Cannot exploit request locality
  - Increases average latency, decreasing throughput
Shortest positioning time first (SPTF)

- Always pick request with shortest seek time

- Advantages
  - 
  - 
  - 

- Disadvantages
  - 
  - 
  - 

- Improvement
  - 
  - 
  - 
Shortest positioning time first (SPTF)

• Always pick request with shortest seek time

• Advantages
  - Exploits locality of disk requests
  - Higher throughput

• Disadvantages
  - Starvation
  - Don’t always know what request will be fastest

• Improvement: Aged SPTF
  - Give older requests higher priority
  - Adjust “effective” seek time with weighting factor:
    \[ T_{\text{eff}} = T_{\text{pos}} - W \cdot T_{\text{wait}} \]
“Elevator” scheduling (SCAN)

- Sweep across disk, servicing all requests passed
  - Like SPTF, but next seek must be in same direction
  - Switch directions only if no further requests

- Advantages
  -
  -

- Disadvantages
  -
  -

- Variant
“Elevator” scheduling (SCAN)

- Sweep across disk, servicing all requests passed
  - Like SPTF, but next seek must be in same direction
  - Switch directions only if no further requests

- Advantages
  - Takes advantage of locality
  - Bounded waiting

- Disadvantages
  - Cylinders in the middle get better service
  - Might miss locality SPTF could exploit

- CSCAN: Only sweep in one direction
  Very commonly used algorithm in Unix
VSCAN(r)

- **Continuum between SPTF and SCAN**
  - Like SPTF, but slightly uses “effective” positioning time
    If request in same direction as previous seek: \( T_{\text{eff}} = T_{\text{pos}} \)
    Otherwise: \( T_{\text{eff}} = T_{\text{pos}} + r \cdot T_{\text{max}} \)
  - when \( r = 0 \), get SPTF, when \( r = 1 \), get SCAN
  - E.g., \( r = 0.2 \) works well

- **Advantages and disadvantages**
  - Those of SPTF and SCAN, depending on how \( r \) is set
Proportional scheduling

• **Goal:** Prioritize processes
  - More important tasks should get more resources
  - Fix a target ratio for resource utilization, e.g., 2:1

• **Generally implemented using feedback**
  - Track difference between desired and actual usage
  - Actual will fluctuate from desired over time
  - Weight scheduler with difference

• **Example:** Background thread scans DB for statistics
  - Want more throughput for critical transactions
Review: The different Unix contexts

- User-level (processes that must be scheduled)
- Kernel "top half"
  - System call, page fault handler, kernel-only process, etc.
- Software interrupt
- Device interrupt
- Timer interrupt (hardclock)
- Context switch code
Transitions between contexts

- **User** → **top half**: syscall, page fault
- **User/top half** → **device/timer interrupt**: hardware
- **Top half** → **user/context switch**: return
  - `want_resched` variable causes scheduler to be invoked
  - Can be set in device context—e.g., completed disk I/O makes process runnable
- **Top half** → **context switch**: sleep
- **Context switch** → **user/top half**
Top/bottom half synchronization

- Top half kernel procedures can mask interrupts
  
  ```c
  int x = splhigh();
  /* ... */
  splx(x);
  ```

- splhigh disables all interrupts, but also splnet, splbio, splsoftnet, ...

- Masking interrupts in hardware can be expensive
  
  - Optimistic implementation – set mask flag on splhigh, check interrupted flag on splx
Process scheduling

- **Goal: High throughput**
  - Minimize context switches to avoid wasting CPU, TLB misses, cache misses, even page faults.

- **Goal: Low latency**
  - People typing at editors want fast response
  - Network services can be latency-bound, not CPU-bound

- **BSD time quantum: 1/10 sec (since ~1980)**
  - Empirically longest tolerable latency
  - Computers now faster, but job queues also shorter
Multilevel feedback queues (BSD)

• Every runnable proc. on one of 32 run queues
  - Kernel runs proc. on highest-priority non-empty queue
  - Round-robin among processes on same queue

• Process priorities dynamically computed
  - Processes moved between queues to reflect priority changes
  - If a proc. gets higher priority than running proc., run it

• Idea: Favor interactive jobs that use less CPU
Process priority

- \texttt{p\_nice} – user-settable weighting factor

- \texttt{p\_estcpu} – per-process estimated CPU usage
  - Incremented whenever timer interrupt found proc. running
  - Decayed every second while process runnable

\[
\text{p\_estcpu} \leftarrow \left( \frac{2 \cdot \text{load}}{2 \cdot \text{load} + 1} \right) \text{p\_estcpu} + \text{p\_nice}
\]

- Run queue determined by \texttt{p\_usrpri}/4

\[
\text{p\_usrpri} \leftarrow 50 + \left( \frac{\text{p\_estcpu}}{4} \right) + 2 \cdot \text{p\_nice}
\]

(value clipped if over 127)
Sleeping process increases priority

- \( p_{estcpu} \) not updated while asleep
  - Instead \( p_{slptime} \) keeps count of sleep time

- When process becomes runnable
  \[
p_{estcpu} \leftarrow \left( \frac{2 \cdot \text{load}}{2 \cdot \text{load} + 1} \right)^{p_{slptime}} \times p_{estcpu}
\]
  - Approximates decay ignoring nice and past loads
Discussion

• 10 people running vi have 1 sec latency?

• How do UNIX signals work?
  - What if signal arrives while process in “top half”

• Does UNIX kernel suffer from priority inversion?