Thrashing and Storage Devices

ECE 469, March 24

Aravind Machiry

RECAP: Page Replacement Algorithms

- Optimal

- FIFO
- Random

- Approximate LRU (NRU)

- FIFO with 2nd chance
- Clock: a simple FIFO with 2nd chance
- Enhanced FIFO with 2nd chance

Whose pages should be replaced?

- Global replacement:
  - All pages from all processes are lumped into a single replacement pool
  - Most flexibility, least (performance) isolation

- Local replacement
  - Per-process replacement:
    - Each process has a separate pool of pages
  - Per-user replacement:
    - Lump all processes for a given user into a single pool
- In local replacement, must have a mechanism for (slowly) changing the allocations to each pool

Why we need paging?Handling low memory

- Suppose you have 8GB of main memory

- Can you run a program that its program size is 16GB?
    - Yes, you can load them part by part
    - This is because we do not use all of data at the same time

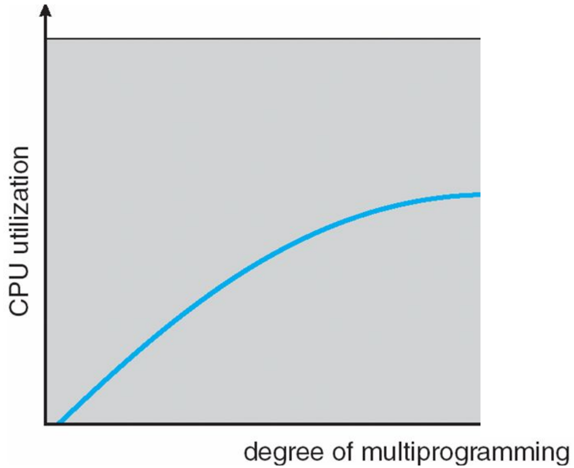
- Can your OS do this execution seamlessly to your application?

Why we need paging?Efficient use of memory!

- Process exhibit locality - not all pages of a process need to be in memory!

- Bringing in only required pages allows us to execute multiple processes seamlessly:
    - Increases CPU utilization

Increasing multiprogramming increases CPU utilization!!?



What happens when there is not enough physical memory?

- Suppose many processes are making frequent references to 50 pages, memory has 49

- Assuming LRU
  - Each time one page is brought in, another page, whose content will soon be referenced, is thrown out

- What is the average memory access time?
- The system is spending most of its time paging!

- The progress of programs makes it look like "*memory access is as slow as disk*", rather than "*disk being as fast as memory*"

Thrashing!!

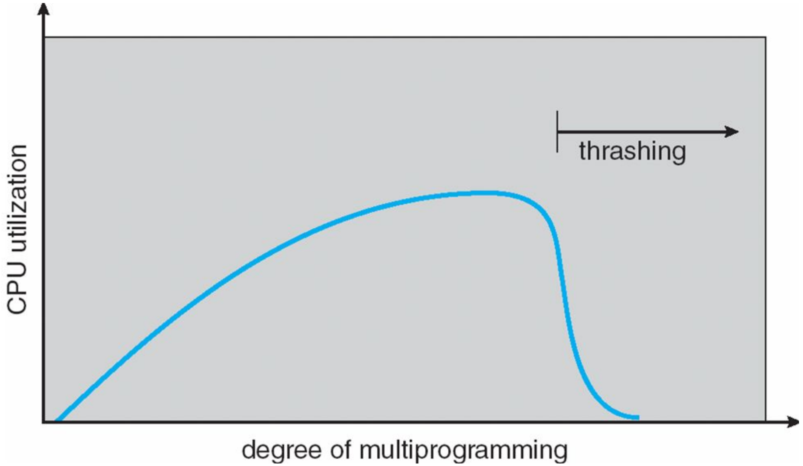
- **Thrashing** ≡ a process is busy swapping pages in and out

Thrashing can lead to vicious cycle

- If a process does not have “enough” pages, the page-fault rate is very high. This leads to:
  - low CPU utilization
  - OS thinks that it needs to increase the degree of multiprogramming (actual behavior of early paging systems)
  - another process added to the system
  - page fault rate goes even higher

Vicious Cycle

Thrashing!!



What causes Thrashing!?

- The system does not know it has taken more work than it can handle
- Virtual memory bites back!
- Mitigating Thrashing:
  - Run fewer programs.
  - Dropping or degrading a course if taking too many than you can handle ☺️

Demand Paging and Thrashing!?

- Why does demand paging work?
  - Data reference exhibits locality

- Why does thrashing occur?
  - Σ size of locality > total memory size

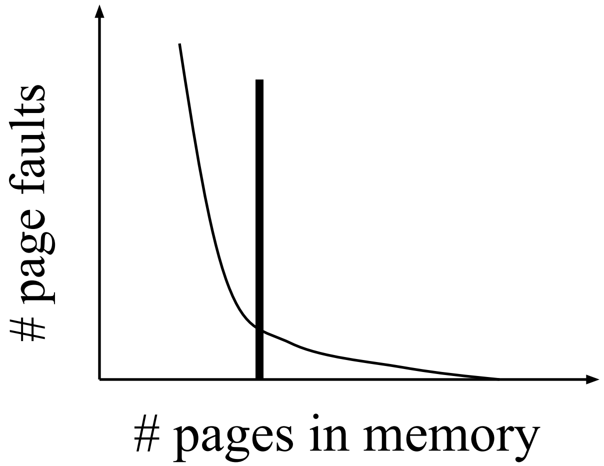
Intuitively, what to do about thrashing?

- If a single process's locality too large for memory, what can OS do?
  - e.g., pin most data (hotter data) in memory, sacrifice the rest

- If the problem arises from the sum of several processes?
  - Figure out how much memory each process needs – "locality"
  - What can we do?
    - Can limit effects of thrashing using local replacement
    - Or, bring a process' working set before running it
    - Or, wait till there is enough memory for a process's need

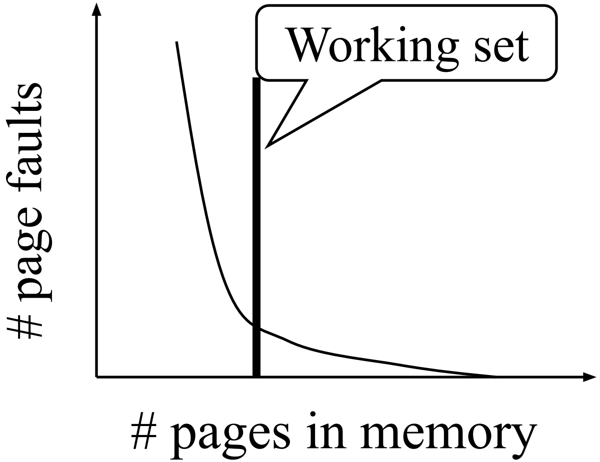
Key Observation

- Locality in memory references
  - Spatial and temporal
- Want to keep a set of pages in memory that would avoid a lot of page faults
  - "Hot" pages
- Can we formalize it?



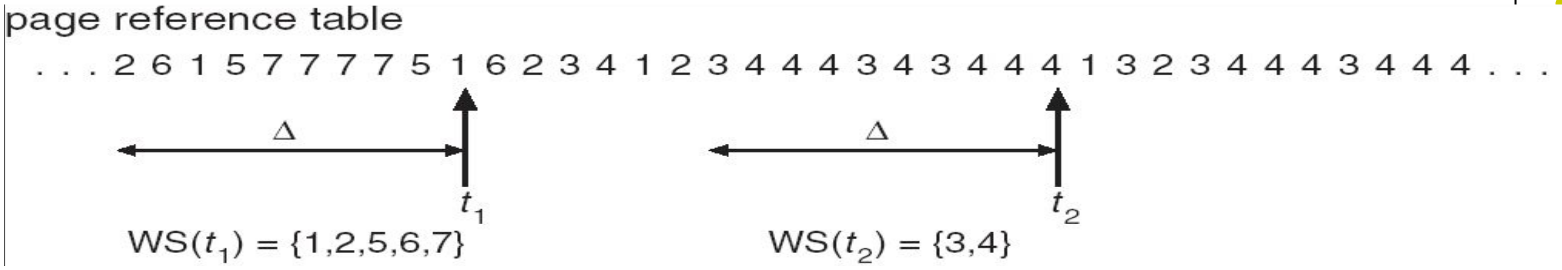
Working Set Model – by Peter Denning (Purdue CS head, 79-83)

- An informal definition:
  - Working set: The collection of pages that a process is working within a time interval, and which must thus be resident if the process is to avoid thrashing



- But how to turn the concept/theory into practical solutions?
  1. Capture the working set
  2. Influence the scheduler or replacement algorithm

Working Sets

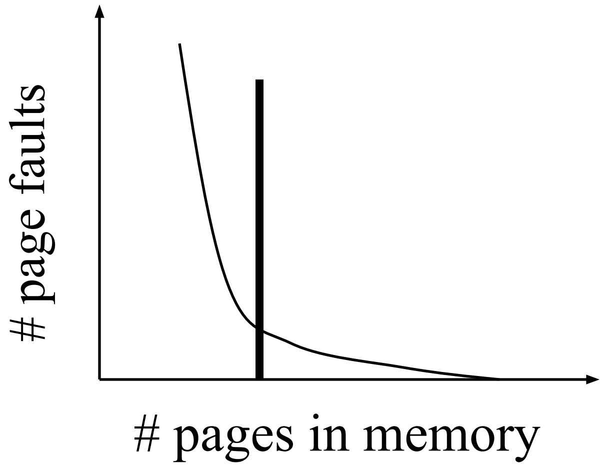


- The <u>working set size</u> is *num of* pages in the working set
  - the number of pages touched in the interval [t-Δ+1..t].
- The working set size changes with program locality.
  - during periods of poor locality, you reference more pages.
  - Within that period of time, you will have a larger working set size.
- Goal: keep WS for each process in memory.

Working Set Model

- Usage idea: use recent needs of a process to predict its future needs
  - Choose Δ, the WS parameter
  - At any given time, all pages referenced by a process in its last Δ seconds comprise its working set
  - Don't execute a process unless there is enough memory to fit its working set

- Needs a companion replacement algorithm



Working Set Replacement Algorithm

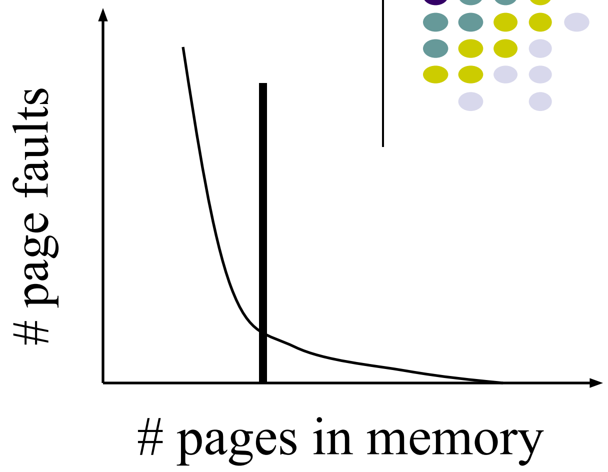
- Main idea
  - *Take advantage of reference bits*
  - *Variation of FIFO with 2nd chance*
- An algorithm (assume reference bit)
  - On a page fault, scan through all pages of the process
  - If the reference bit is 1, clear the bit, record the current time for the page
  - If the reference bit is 0, check the “last use time”
    - If the page has not been used within Δ, replace the page
    - Otherwise, go to the next page

Working Set Clock Algorithm (assume reference bit + modified bit)

- Upon page fault, follow the clock hand
- If the reference bit is 1, set reference bit to 0, set the current time for the page and go to the next
- If the reference bit is 0, check “last use time”
  - If page used within Δ, go to the next
  - If page not used within Δ and modify bit is 1
    - Schedule the page for page out (then reset modify bit) and go to the next
  - If page not used within Δ and modified bit is 0
    - Replace this page

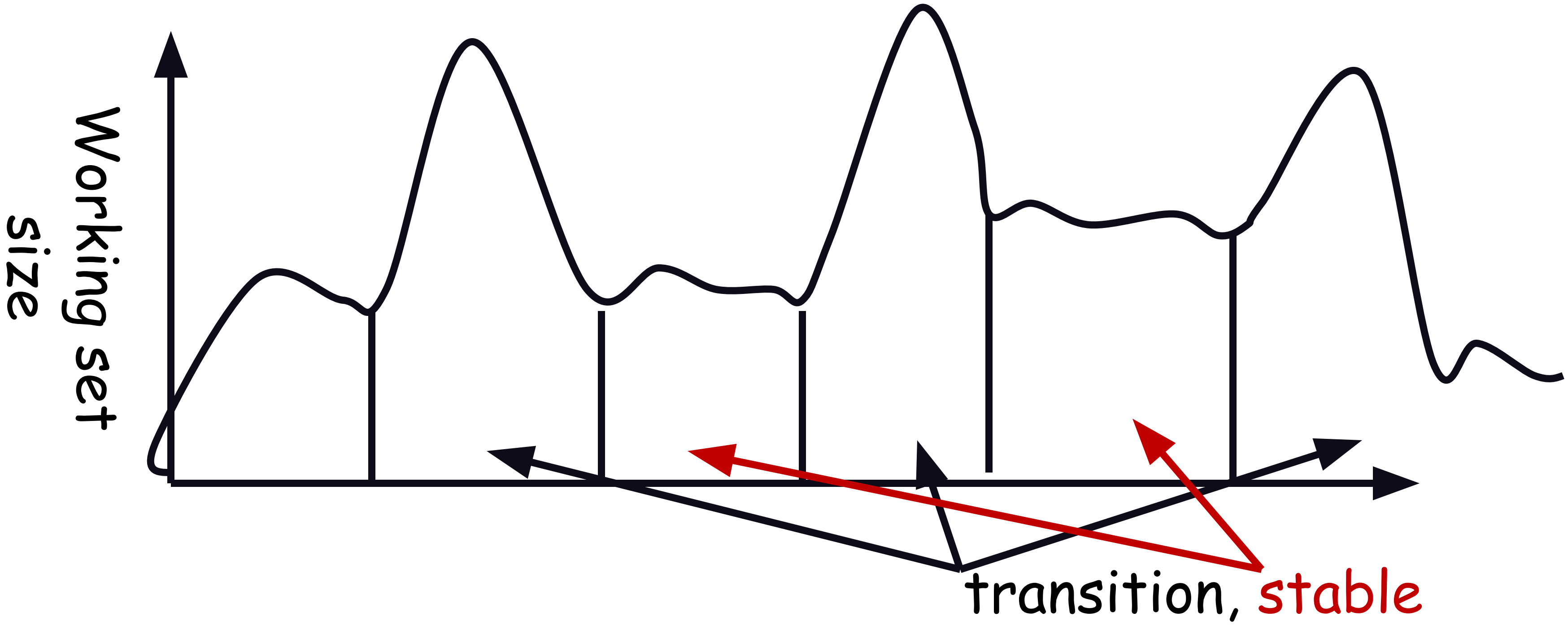
Challenges with WS algorithm implementation

- What should Δ be?
  - What if it is too large?
  - What if it is too small?



- How many jobs need to be scheduled in order to keep CPU busy?
  - Too few □ cannot keep CPU busy if all doing I/O
  - Too many □ their WS may exceed memory

Working Sets in Real World



More Challenges with WS algorithm implementation

- Working set isn't static
- There often isn't a single "working set"
  - e.g., Multiple plateaus in previous curve (L1 $, L2 $, etc)
  - Program coding style affects working set
  - e.g., matrix multiply
- Working set is often hard to measure
  - What's the working set of an interactive program?
  - How to calculate WS if pages are shared?

Storage Devices

- Devices used to store data

Storage Technologies

- Tapes
- Magnetic Disks
- Flash Memory

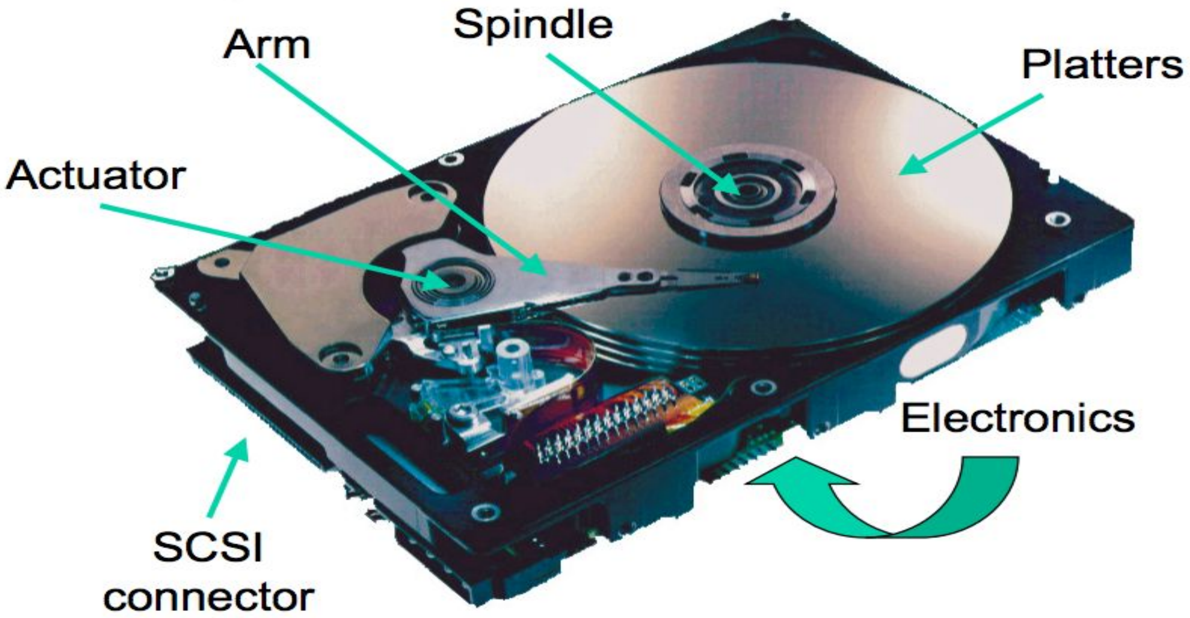
Tapes

- Low-cost, highly-reliable storage.
- Slow access: ~30MB Per Second.

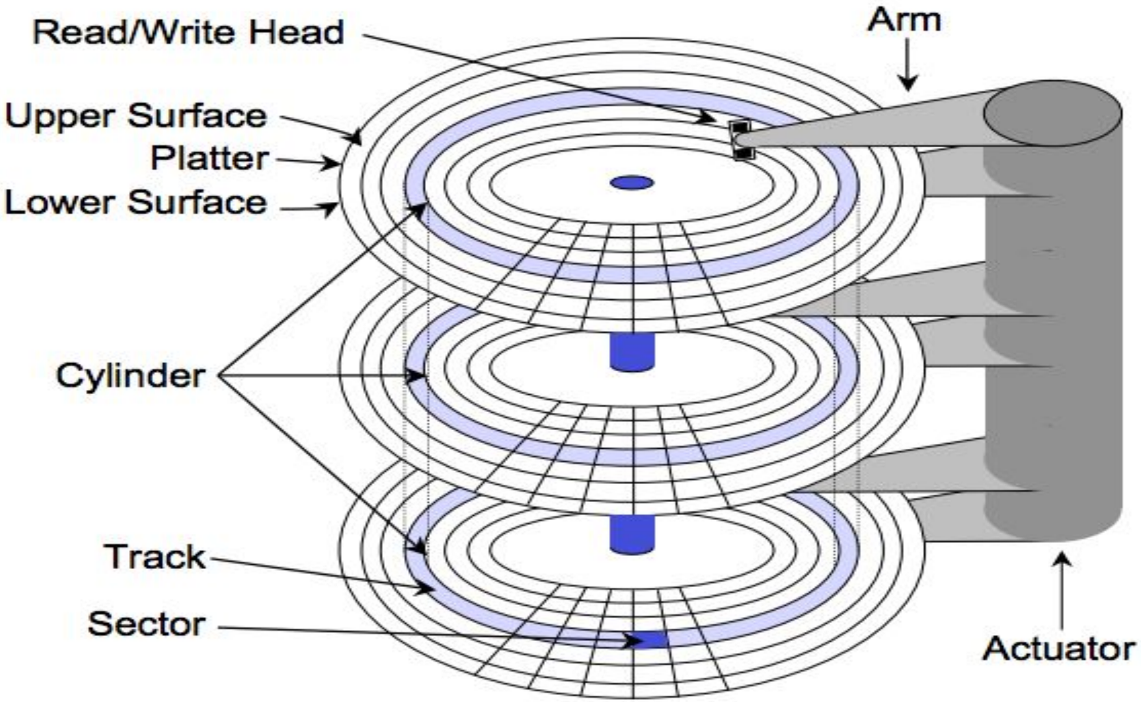
Magnetic Disks

- De Facto standard for storage (not any more).
- Medium access: ~150MB Per Second.
- Relatively high failure rate (vs Tape).

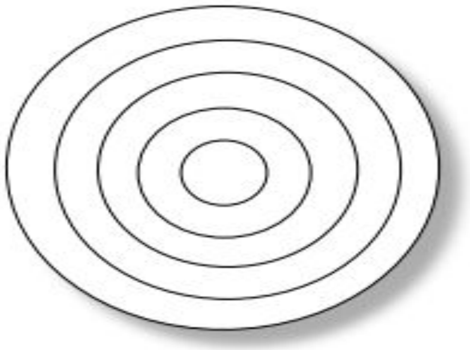
Magnetic Disk



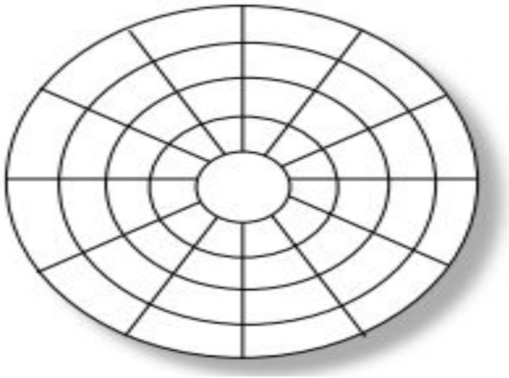
Disk Components



Surface Organized into Tracks

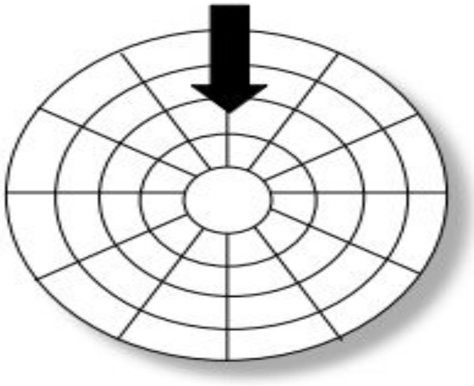


Tracks Broken into Sectors

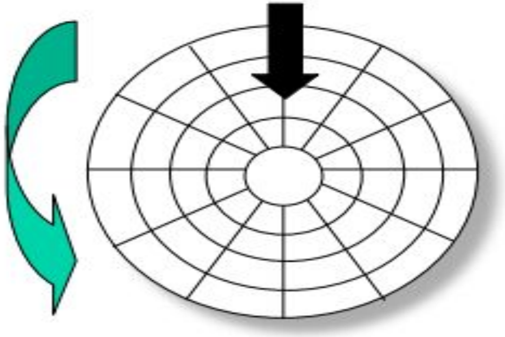


- Disk accesses in the granularity of a sector (usually 512KB)
- This I/O interface is called block I/O interface

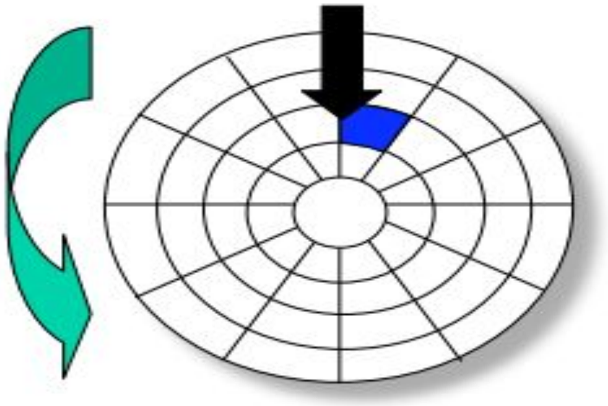
Disk Head Position



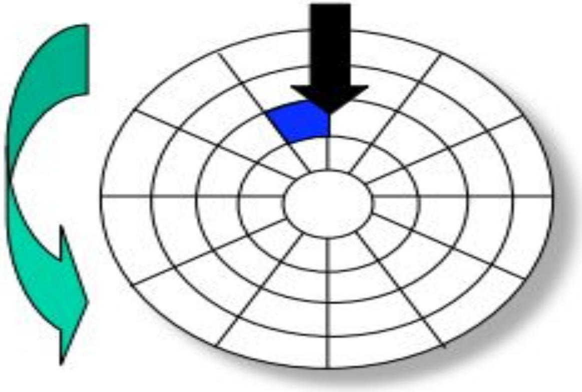
Rotation is Counterclockwise



About to Read Blue Sector

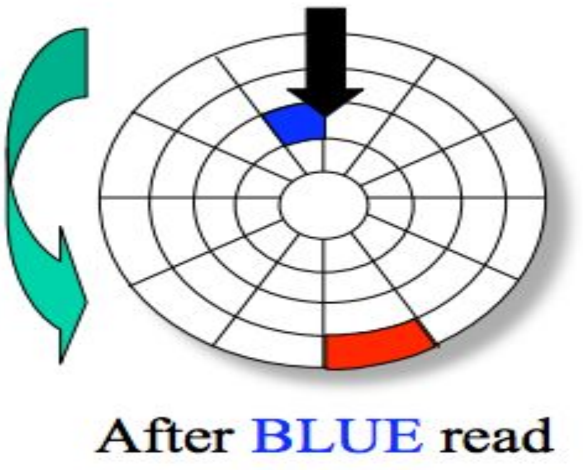


After Reading Blue Sector

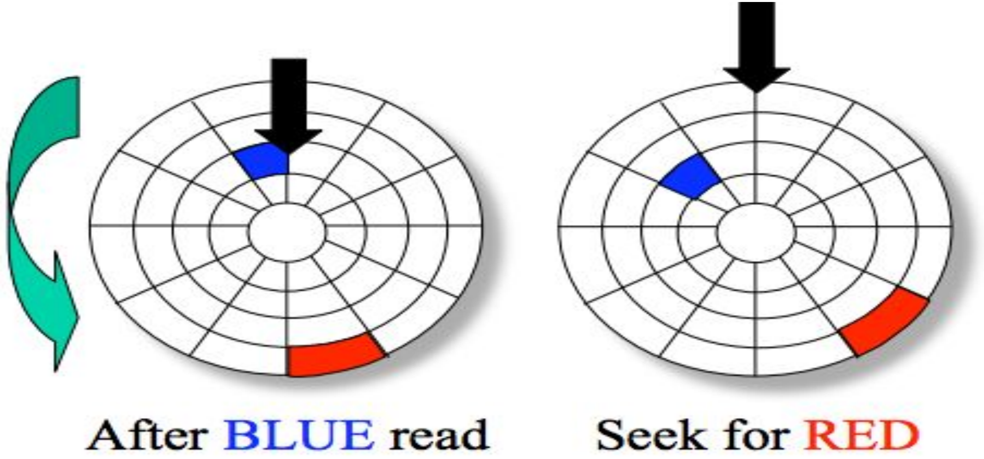


After BLUE read

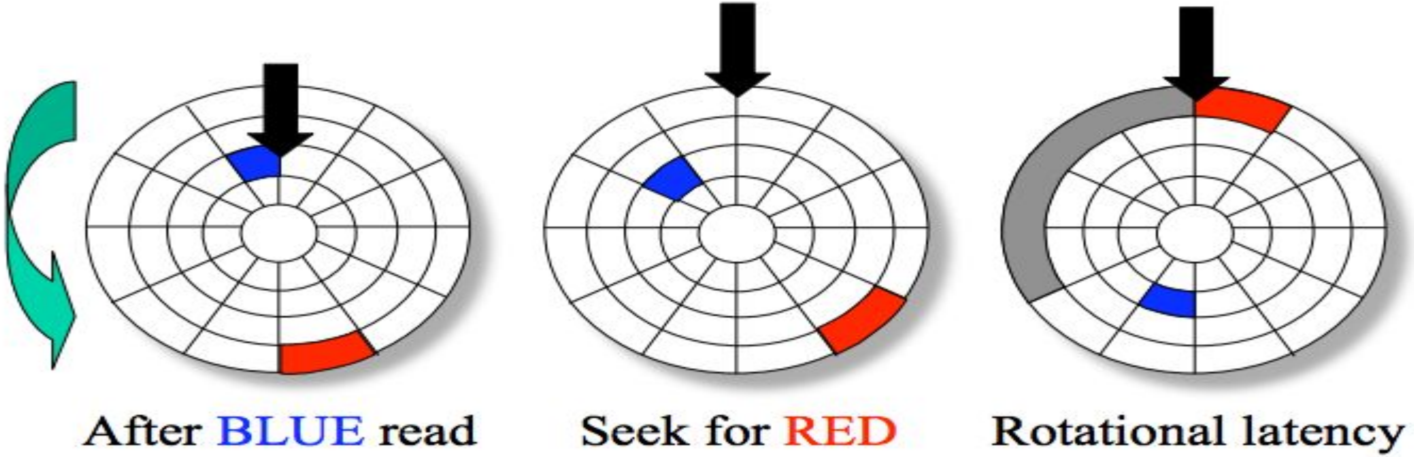
Red Request Scheduled Next



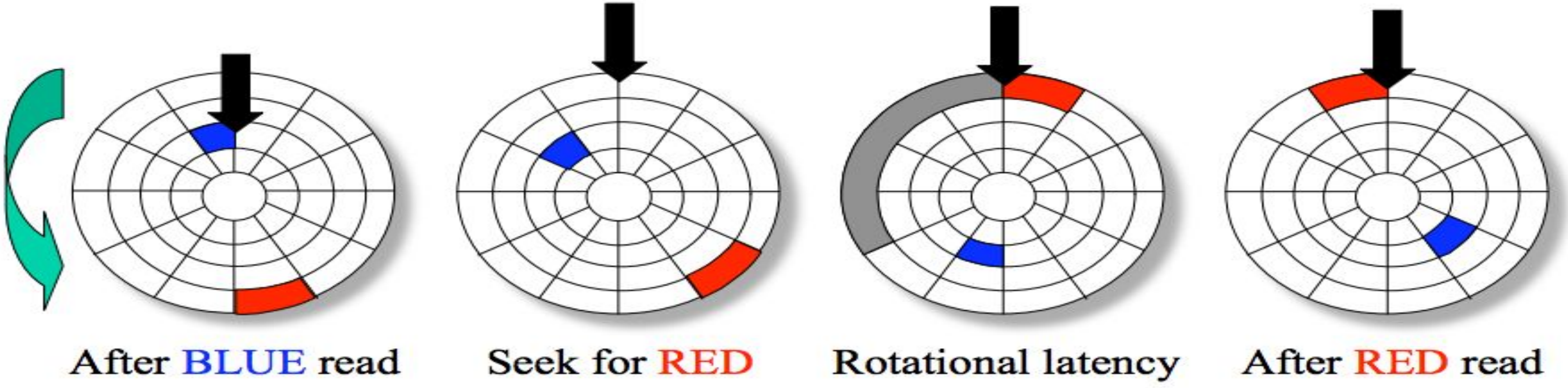
Seek to Red's Track



Seek to Red Sector to Reach Head



Read Red Sector



Real numbers for Modern Disks

- # of platters: 1-8

– 2-16 surfaces for data

- # of tracks per surface: 10s of 1000s

– same thing as # of cylinders

- # sectors per track: 200-1000

– so, 100-500KB

- # of bytes per sector: usually 512

– can be chosen by OS for some disks

Response Time for Disks

- Access time: (service time for a disk access)
  - Once command is received, how long it takes to get the data to OS.
    - Seek + Rotation + Transfer

- Response time:
  - Commands may be queued!
    - Queue time + Access time

Seek Time

- Time required to move head over desired track
  - Physically moving the head, not electronic => slow!

- A seek has up to four components
  - – accelerate
  - – coast at max velocity
  - – decelerate
  - – settle onto correct track

- Seek time depends on workload
  - For random workloads, longer seek time

Rotational Latency

- Time required for the first desired sector to reach head
- Depends on rotation speed
  – measured in Rotations Per Minute (RPMs)
- Computing average rotational latency
  – for almost all workloads, we can safely assume that there is an equal likelihood of landing on any sector of the track
  – this gives equal probability of each rotational latency
    - from 0 sectors to N-1 sectors
  – thus, average rotational latency is time for 1/2 revolution
  – e.g., for 7200 RPM
    - one rotation = 60s / 7200 = 8.33 ms
    - average rotational latency = 4.16 ms

Modern Disk Performance Characteristics

- Seek times: 0.5-15ms, depending on distance
  - average 5-6ms
  - improving at 7-10% per year

- Rotation speeds: 5600-15000 RPMs
  - improving at 7-10% per year

Disk failures

- Disks fail more often....
  - When continuously powered-on
  - With heavy workloads
  - Under high temperatures

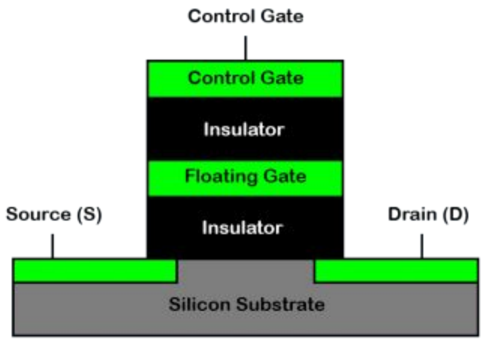
How disks fail?

- How do disks fail?
  - Whole disk can stop working (e.g., motor dies, firmware errors)
  - Transient problem (cable disconnected, firmware errors)
  - Individual sectors can fail (e.g., head crash or scratch)
    - Data can be corrupted or block not readable/writable

Fixing disk errors

- Disks can internally fix some sector problems
  - ECC (error correction code): Detect/correct bit flips
  - Retry sector reads and writes: Try 20-30 different offset and timing combinations for heads
  - Remap sectors: Do not use bad sectors in future
    - How does this impact performance contract??

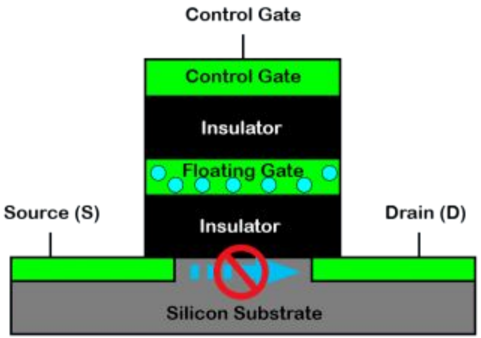
Flash Memory: NAND Cell



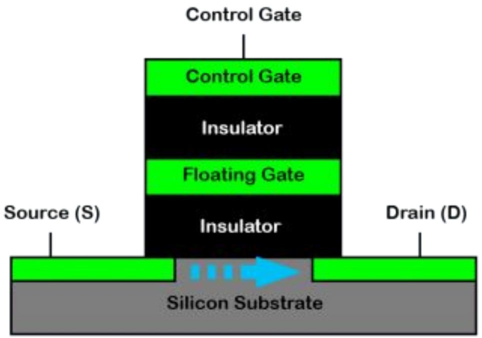
https://www.cactus-tech.com/resources/blog/details/solid-state-drive-primer-1-the-basic-nand-flash-cell/

Flash Memory: NAND Cell

Reading



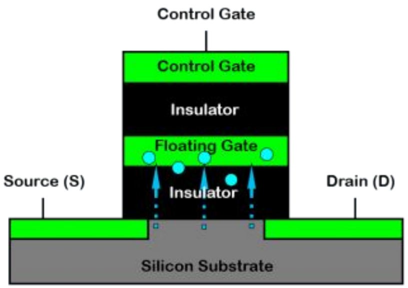
0



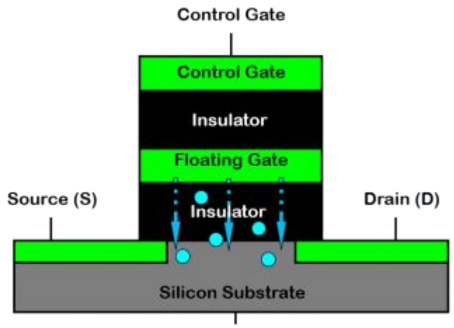
1

Flash Memory: NAND Cell

Writing

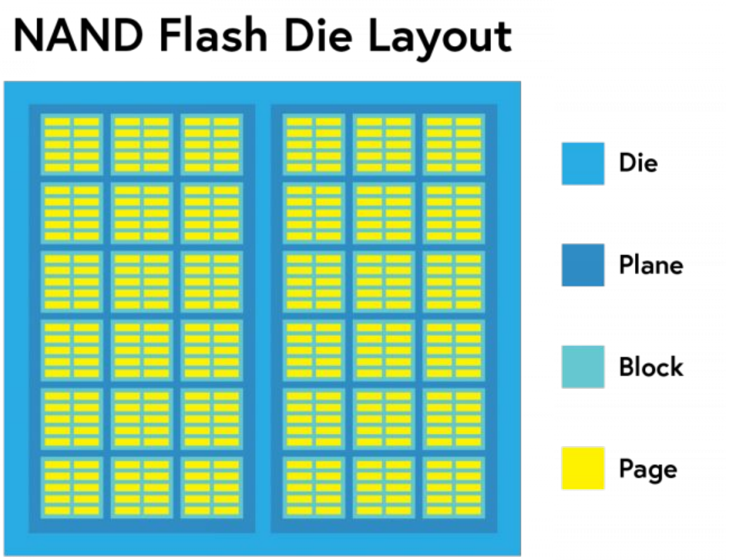


0



1

Flash Memory



Flash Memory

- Page Size: 512 - 4K bytes.

- Write needs complete erasure:
  - Erase before write.
    - Cannot go cell-wise from 0 -> 1

  - Any cells that have been set to 0 (written to) by programming can only be reset to 1 by erasing the entire block.

- Writes >> Reads

Flash Memory : SSD

Samsung SSD 860 EVO 1TB 2.5 Inch SATA III Internal SSD (MZ-76E1T0B/AM)

Add to Cart

★★★★★ (62922)

109^{99}

✓prime

Amazon.com

mac, windows

78 Gb per second

Internal Solid State Drive

1 TB

2.50 inches

SATA 6.0 Gb/s

PC, Mac

3.94 x 2.76 x 0.27 inches

1.80 ounces

2018

SSD : Flash Translation Layer

- Maps logical blocks to real blocks.
- Hides erase before write.
- Maintains free blocks.