CSCI 3753
Operating Systems

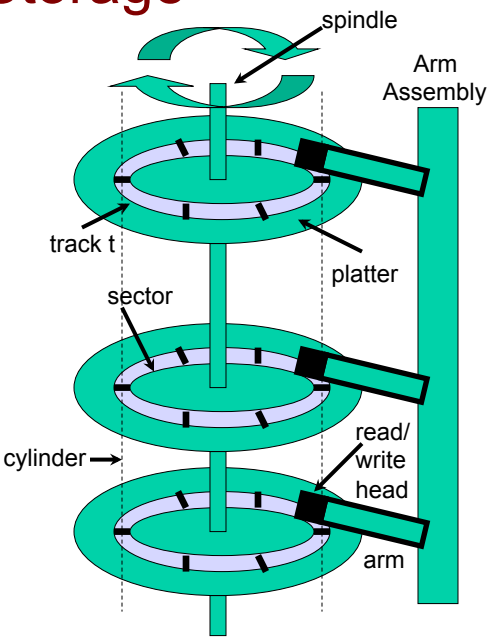
Mass-Storage Structure

Chapters 10

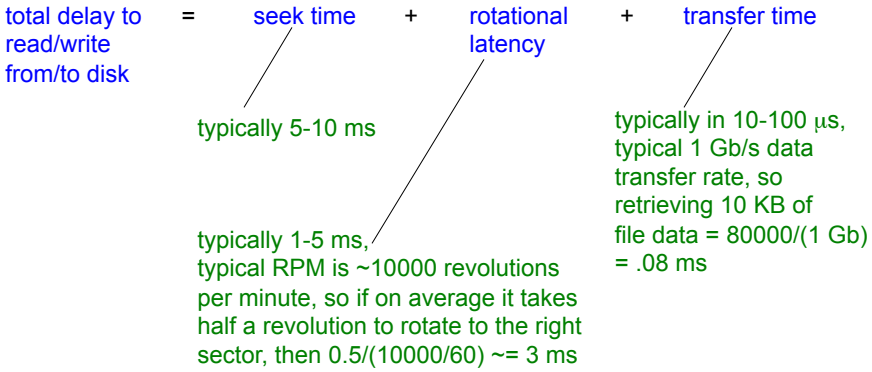
Lecture Notes By
Shivakant Mishra
Computer Science, CU-Boulder
Last Update: 04/10/14

Disk Storage

- Files and file systems are stored in non-volatile permanent/secondary storage, e.g. magnetic disks (or solid-state/flash drives)
- 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*
- read/write 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*



Disk Access Latency



- 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
- If the R/W head is 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
  - if initial R/W head position = 53, and given requests 98, 183, 37, 122, 14, 124, 65, 67, then SSTF services requests in the following order:
    - 65, 67, 37 (closer than 98), 14, 98, 122, 124, 183
    - total number of cylinders traversed = 236, this is about 1/3 of FCFS!
  - 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
    - similar to shortest job first process scheduling, suffers the same problem of starvation, i.e. while the disk head is positioned at 37, a series of nearby requests may come into the queue for 39, 35, 38, ... SSTF will then continue to service requests near the current position, while requests far away can be starved for service

SCAN/LOOK Disk Scheduling

- Disk R/W head moves in one direction, say from innermost to outermost track service requests on the way until there are no more requests in that direction, then reverses direction, then reverses..., sweeping across the disk in alternate directions
  - Analogous to how an elevator operates
  - Starvation free solution, and handles dynamic arrival of new requests, and simple to implement
- Given initial direction up from current head position = 53, 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→37→14
  - total # cylinders traversed = 130 up + 169 down = 299 cylinders

C-SCAN/C-LOOK Disk Scheduling

- Problem with SCAN: 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
- Solution: 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
  - In the example, C-SCAN would scan 53→65→67→98→122→124→183→14→37 for a total of 322 tracks

Disk Scheduling

- Most of the previous algorithms focused on reducing seek times. There are other algorithms for reducing rotational latency, usually embedded in the disk controller.

What is Flash Memory?

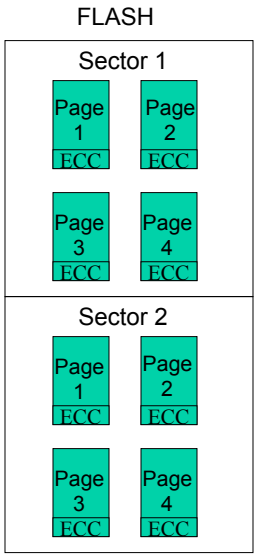
- Flash memory is a type of EEPROM – bits are stored in an electronic chip, not on a magnet medium like a disk
  - Programmed and erased electrically – no mechanical parts
  - Non-volatile storage, i.e. "permanent" with caveats
- Primarily used in solid state drives (SSDs), memory cards, USB flash drives, mobile smartphones, digital cameras, etc.
- Flash's name is due to its electronic circuit design, which can inexpensively erase a large amount of solid state memory quickly, like a "flash" of light
- Advantages vs. disk: much faster access (lower latency), more resistant to kinetic shock (& intense pressure, water immersion etc.), more compact, lighter, lower power (typically 1/3-1/2 of disks), …
- Disadvantages vs. disk: more costly per byte, limited lifetime, erases are costly, …

NOR and NAND Flash Memory

- NAND flash
  - Smaller memory cell area (less expensive)
  - Word accessible (large granularity, 100-1000 bits/word, good for secondary storage where files don't mind being read out in large chunks/words)
  - Slower random byte access (have to read a word)
  - Short erasing (~2 ms) and programming times (~5 MB/sec sustained write speed)
  - Longer write lifetime
- NOR flash
  - Larger memory cell area
  - Byte accessible (good for ROM-like program storage, where instructions need to be read out on a byte-size granularity)
  - Faster random byte access
  - Longer erasing (~750 ms) and programming times (~.2 MB/sec sustained write speed)
  - Shorter write lifetime
- iPhone has both multi-GB NAND flash for multimedia file storage and multi-MB NOR flash for code/app storage

Organizing Flash Memory

- Flash memory is typically divided into blocks or sectors
  - There can be many pages per block
    - e.g. 64 pages per block, 16 blocks per flash, and 4 KB per block => 4 MB of flash
  - The last 12-16 bytes of each page is reserved for error detection/correction (ECC) and bookkeeping – the flash file system can put information in this space



Operations on Flash Memory

- Typical operations:
  - Reads and writes occur on a sub-page level granularity, i.e. you could read a byte (NOR) or word (NAND), and under certain conditions can write a byte (NOR) or word (NAND)
  - Erases occur on a sector level granularity only – this is a key property and drawback to flash memory, i.e. the price we pay for quickly erasing a large amount of information in a "flash" is that it can be done only in large chunks or sectors
- Most operations proceed as normal
  - To read a byte or word of data, specify which page you want to read from and what offset in that page to start reading
  - To write a byte or word of data on a clean page (not been written to since the last erase operation), specify which page you want to write to and what offset in that page to start writing

Characteristics of Flash Memory

- *Rewriting* over memory (a portion of a page) that has already been written to earlier usually requires an extra block/sector erasure step
  - This is unlike magnetic disks, where you can rewrite over any block of disk that has already been written to simply be changing the magnetization of the bits on disk
  - Example: a flash page P contains files F1 and F2. F1 is deleted, and its space in P is deallocated. If we want to rewrite over the space formerly occupied by F1, we have to first erase that space. Since erases can only happen on a sector granularity, then we have to erase all the other pages in the sector in addition to P.

Characteristics of Flash Memory

Rewriting over memory (continued)

- If we also want to preserve the existing information in the other pages in the same sector, and also F2 in P,
  1. we'd have to read the entire sector into memory, rewrite in RAM over the space corresponding to F1,
     - Or read entire sector except for F1's pages into a clean sector S, then write into F1's page locations
  2. erase the entire sector on flash
  3. copy the modified sector from RAM or sector S back to flash.
- Thus *rewrites* to previously written parts of pages require a sector erase, which is very slow.

Characteristics of Flash Memory

- Memory wear - Flash memory has a limited write lifetime
  - This is unlike magnetic disk, which can be rewritten indefinitely
  - NOR flash memory can withstand 100,000 write-erase cycles before becoming unreliable
  - NAND flash memory can withstand 1,000,000 write-erase cycles

Wear Leveling Solution

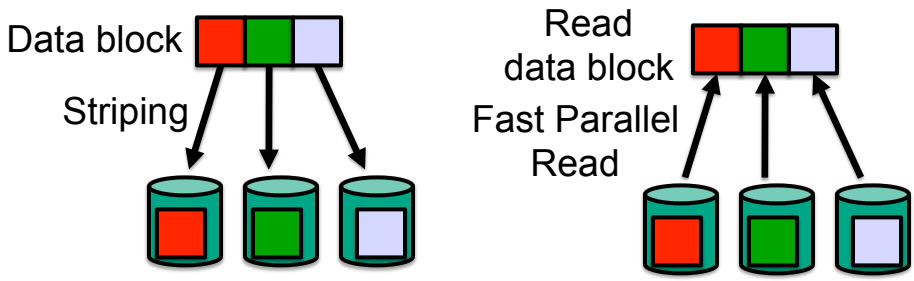
- To extend the life of flash memory chips, write and erase operations are spread out over the entire flash memory
  - Keep a count in 12-16 byte trailer of each page of how many writes have been made to that page, and choose the free page with the lowest write count
  - Randomly scattering data over the entire flash memory span.
  - Many of today's flash chips implement wear leveling in hardware

RAID

- Redundant Array of Inexpensive Disks
- Disks are cheap these days.
- Attaching an array of disks to a computer brings several advantages:
  - Faster read/write access to data by having multiple reads/writes in parallel. Data is *striped* across different disks, e.g. each bit of a byte is striped onto a different disk, or each byte out of an 8-byte word is striped to a different disk
  - Better fault tolerance/reliability – if one disk fails, a copy of the data could be stored on another disk – redundancy to the rescue!

RAID

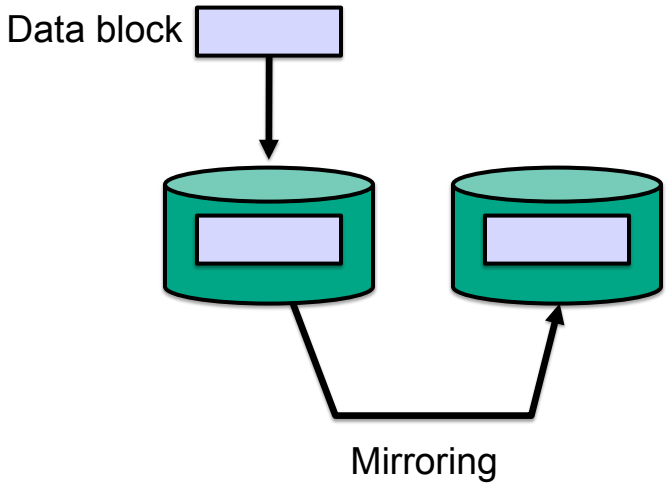
- RAID has different levels with increasing redundancy
  - RAID0 = data striping with no redundancy



Also fast parallel writes

RAID

- RAID1 = mirror each disk



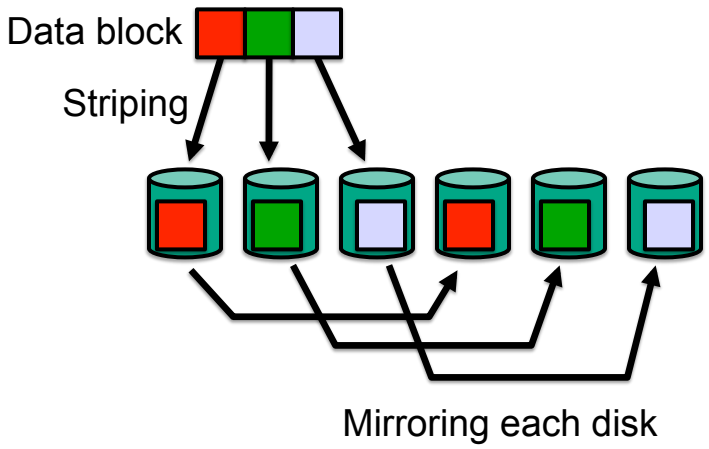
Get redundancy, but not parallel performance speedup of RAID0. Hence, combine RAID0 with RAID1 (next slide)

Can get limited read speedup by submitting the read to all mirrors, and taking the 1st data result

Twice the delay on writes, but OS can mask this by delaying writes

RAID

– RAID1+0, aka "RAID 10"

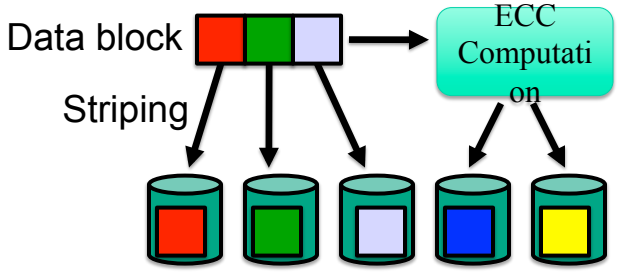


Combinations:

RAID 1+0 = mirror each disk then stripe. Any two disks may fail, and the data can still be retrieved, unless the two disks mirror the same data.

RAID

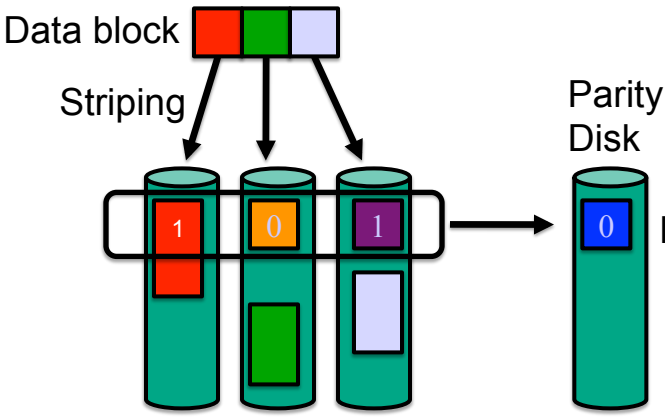
– RAID2 = put Error Correction Code (ECC) bits on other disks



Error correction codes are more compact than just copying the data, and provide strong statistical guarantees against error, e.g. a crashed disk's lost data can be corrected with the redundant data stored in the ECC disks

RAID

– RAID3 = bit-interleaved parity: for each bit at the same location on different disks, compute a parity bit, that is stored on a parity disk

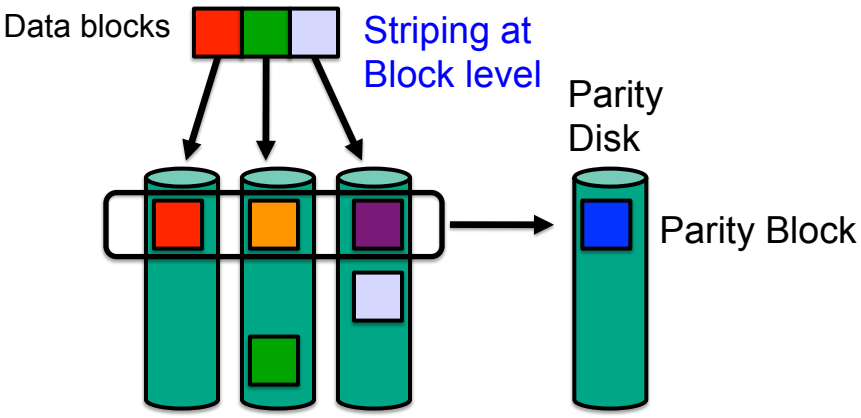


Only 1 extra disk, so less overhead than RAID2

Parity calculation can slow writes, so implement parity calculation in hardware

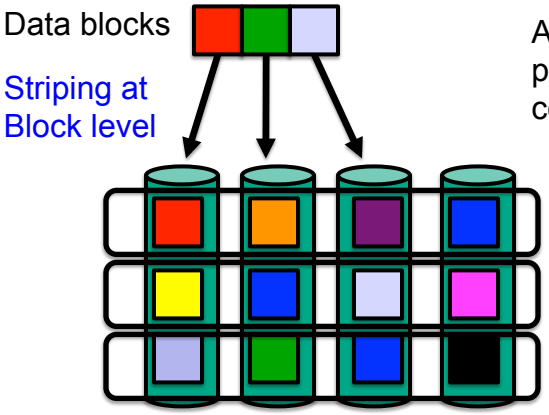
RAID

– RAID4 = block-interleaved parity: for each block at the same location on different disks, compute a parity block, that is stored on a parity disk



RAID

– RAID5 = block-interleaved *distributed* parity: spread the parity blocks to different disks, so every disk has both data and parity blocks



Avoids overuse of RAID4's parity disk, and is the most common parity RAID system

= Parity Blocks

RAID6 replaces parity blocks with stronger ECC

RAID Implementation

Three methods: Software Based, Firmware Based, Hardware Based

1. Software Based: Plug in disks to your computer bus and implement RAID management in software in OS kernel
   - The RAID software layer sits above the device drivers
   - Cost effective and easy to implement, but not as efficient
   - In Linux, use the mdadm package to manage RAID arrays
   - Cannot boot from RAID

RAID Implementation

2. Firmware/Driver Based: Plug in disks to your computer's I/O bus, and an intelligent hardware RAID controller recognizes them and integrates them into a RAID system automatically
   - Standard disk controller chips with special firmware and drivers
   - RAID instructions are stored in the firmware of the device
   - Can boot from RAID: During startup/boot, the RAID is essentially kick-started by the firmware
3. Hardware Based: Plug in disks to a RAID array, which is a stand-alone hardware unit with a RAID controller that looks like one physical device, e.g. SCSI, to your computer bus
   - File system doesn't have to know about RAID to use and benefit from this RAID disk array
   - Expensive but highly efficient