Operating Systems

(Sistemas Operativos)

I/O Devices

University of Minho

2024-2025

What will we learn?

Persistence

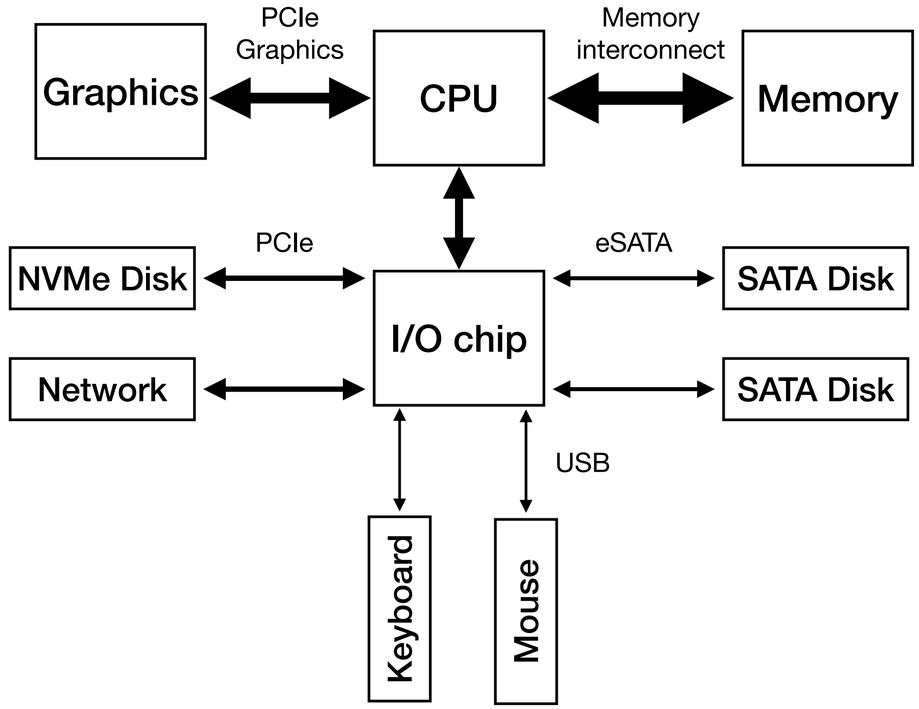
- Memory is volatile, meaning that data is lost upon a reboot (or crash) of the computer. Disk devices (e.g., HDDs, SSDs) allow programs to store their data persistently
  - How does the OS abstract the access (API) to the low-level device drivers of these disks?
  - How does the OS make disk operations efficient when multiple programs use the same device simultaneously?
  - We typically access our data using a file-based abstraction (files, folders, …). How does the OS enable such abstraction?

- We will study the properties and design of disk devices to answer these questions

I/O Devices

Brief overview of an intel-based architecture

- **CPU** and **memory** are connected by the **fastest BUS**
  - A **high-speed BUS** is also used for **graphic cards**
- **CPU** connects to an **I/O chip** through a **Direct Media Interface (DMI)**
  - **PCIe** is the fastest I/O BUS, used for network and modern storage disks (e.g., NVMe SSDs)
  - **eSATA** is used by more traditional disks (SATA SSDs, HDDs)
  - The **USB** BUS is used for low-performance devices such as keyboards, mouses, external drives …

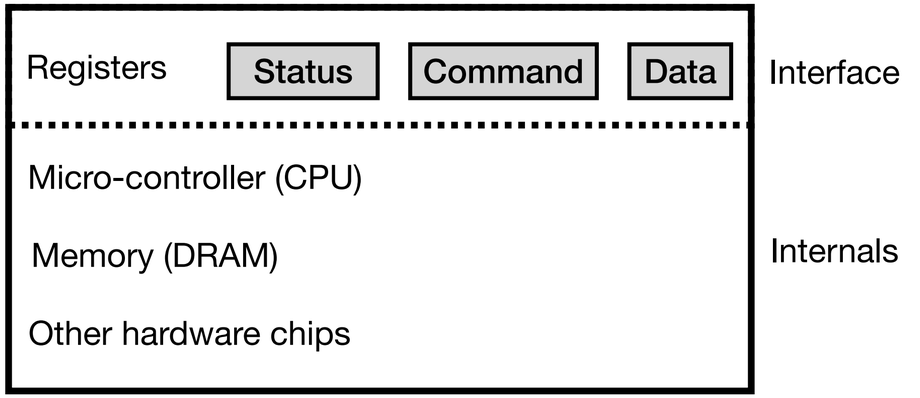


Modern system architecture (Intel-based)

I/O Devices

A canonical device

- A device has two main components
  - The **interface** presented to the system
  - Its **internal structure,** such as the hardware chips (e.g., simple CPU, memory, …) and firmware (i.e., the software that runs in the device)
- In a canonical device (not a real device)
  - The device is controlled through three registers that are read and/or written by the OS
    - **Status:** holds the current status of the device
    - **Command:** holds commands to be executed by the device
    - **Data:** holds data to be written to or read from the device



Canonical Device

I/O Devices

A canonical device

- A *simple protocol* to interact with a disk device would be
  1. The OS waits until the device is ready by **repeatedly reading the status register**. We call this **polling** the device (i.e., repeatedly asking the device)
  2. The OS writes some bytes in the **data register.** When the CPU is responsible for copying data to this register we refer to this approach as **programmed I/O (PIO)**
  3. The OS writes the command register to let the device know that data is present and should persist it
  4. Finally, the OS waits for the request to finish by **polling** again the device (status register)
- For the OS to interact with the device, two approaches are possible
  - **Specific I/O instructions** to write to the **registers** are supported by the OS
  - **Memory-mapped I/O** is used, i.e., the device registers are mapped as these were loaded into memory regions, and the OS uses these as regular memory

I/O Devices

Polling vs interrups

- Polling seems inefficient, it wastes CPU cycles waiting for the device to complete its activity
  - Remember the **interrupts**? The device raises a **hardware interrupt** when the request is ready meanwhile, the OS (and the CPU) can do other useful tasks
- However, using interrupts may not always be the best decision (strange, right?)
  - If the **device is very fast** (i.e., it answers back under one (or few) poll requests), the performance cost of **context switching** and **handling interrupts** may be spared[1]
- When the **speed of the device** is **unknown** or **varies over time,** a hybrid approach is used
  - Do some poll requests, but if the device is slow in serving the request, go for interrupts
- Devices supporting parallel requests often **coalesce** (batch) multiple interrupts before notifying the OS (i.e., to avoid the cost of handling these independently)
  - Trade-off between **throughput** and **latency** (a typical trade-off when requests are batched)

[1]Indeed, modern kernel-bypass I/O frameworks resort to polling to achieve low latency (e.g., the SPDK framework for NVMe SSDs)

I/O Devices

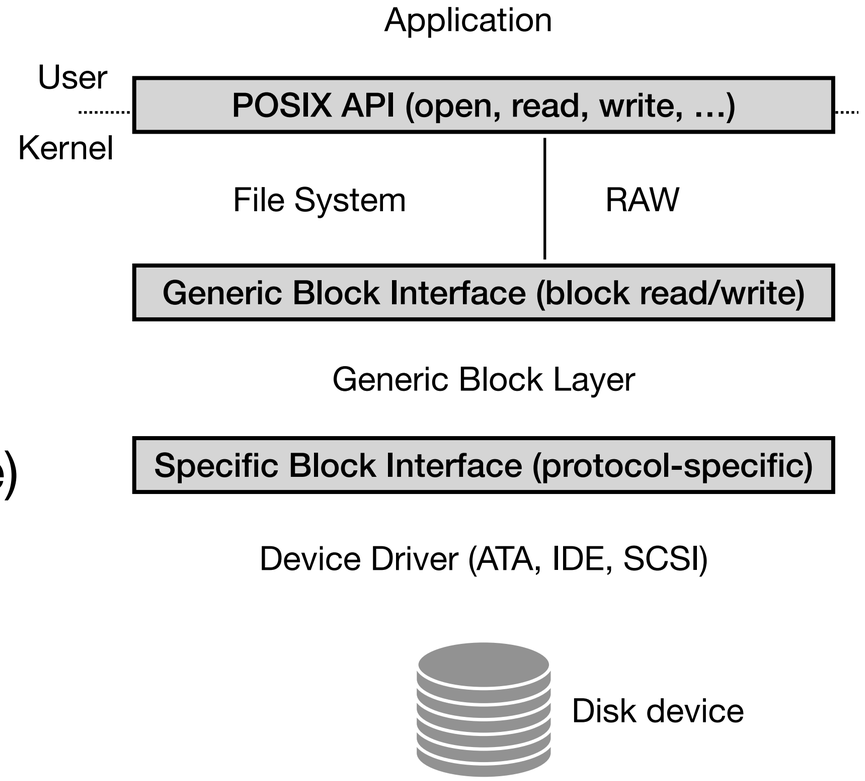
Direct Memory Access (DMA)

- Using **programmed I/O (PIO)** forces the CPU to stop other tasks just to copy data from memory to the I/O device (e.g., the device's register)
- The **DMA** engine is a specific device that can orchestrate data between the I/O devices and main memory with little CPU intervention
- The OS just tells the DMA device(s) where data is to be copied from and to, and then the CPU is free to do other tasks

Fitting Into The OS

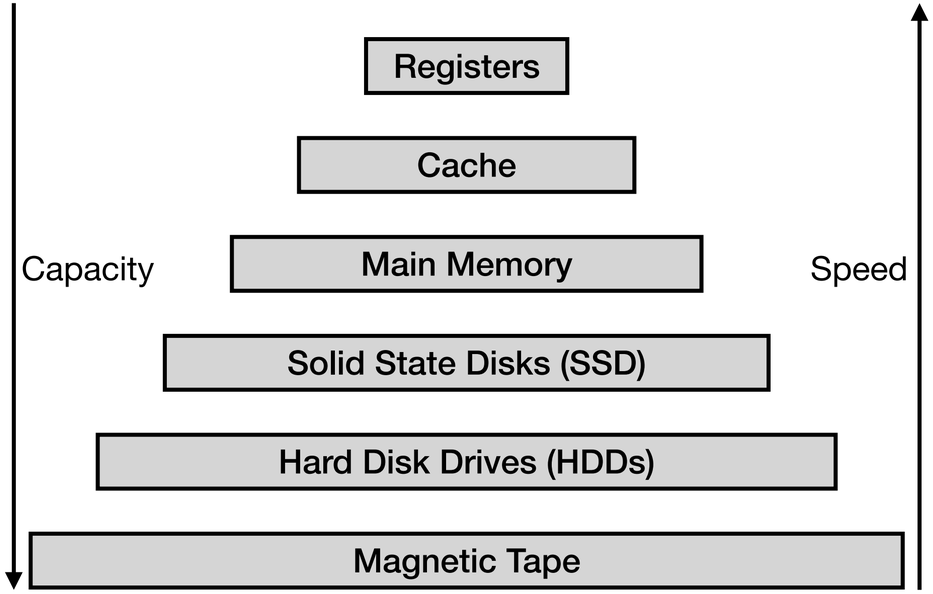
Device driver

- I/O devices support different interfaces (e.g., ATA, IDE, SCSI). How can the OS abstract these interfaces?
  - Solution: the **device driver abstraction**
- Example: A simplified Linux File System stack
  - **Applications** access a file system or a RAW block device, exposing the POSIX API
  - The **file system** (and applications using a raw block device) only see the generic interface exposed by the block layer (i.e., block read and write requests)
  - The **block layer** routes requests to the right **device driver**, which knows how to translate these to protocol-specific writes and reads at the I/O device



Memory Hierarchy
Revisited

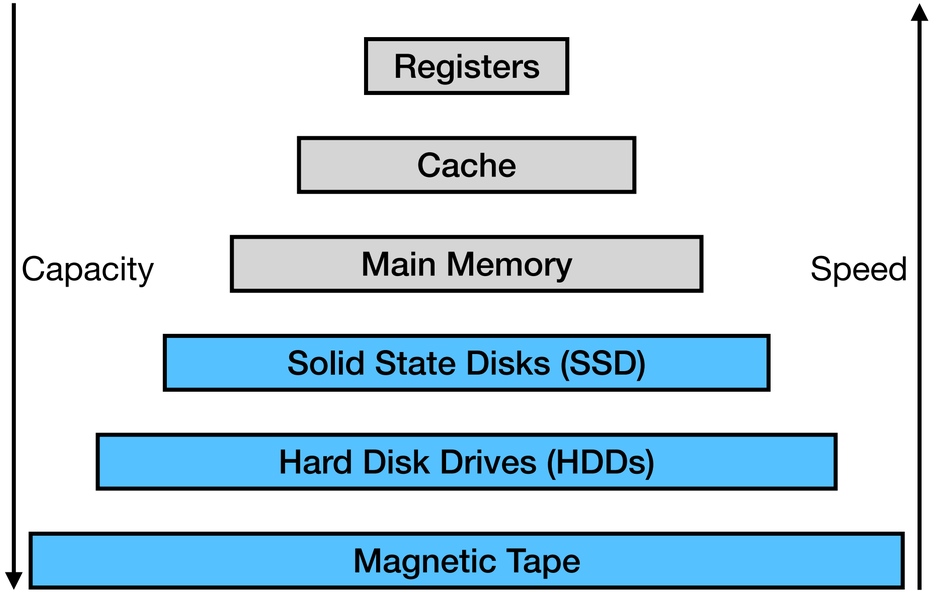
- CPU registers, caches and main memory have the fastest access times (in this order) but are **volatile**
- **Persistent** devices such as SSDs, HDDs and Tapes have slower access speeds (in this order)
- **Trade-offs:**
  - As speed increases, the cost per byte is also higher
  - To make systems affordable (and due to the laws of physics), the amount of available **fast memory** is smaller than the amount of slower **capacity** memory



Persistent Storage Devices

Which disk should I pick?

- Given the sheer amount of persistent storage devices, which one(s) should we pick?
- To answer this question, we must
  - Understand the strengths and drawbacks of each device
    - I.e., performance, cost, capacity, …
  - Know the **workload** (i.e., the type of requests) that devices will be serving
- Let's start with the latter



Storage Workloads

Archival data

- Data is stored for archival purposes. Useful for data that is rarely accessed but may be relevant in the future
  - **Throughput** (i.e., number of operations done per unit of time) is favored over latency (i.e., time required to serve each operation)
    - Large amounts of data must be written/read efficiently
  - **Sequential** and **append-only** workloads
    - Archival data (e.g., files) is written and read sequentially
    - Archival data is usually append-only
      (i.e., **write-once** data with no in-place updates)

Storage Workloads

Backup data

- Data backups of *fresh* data. Useful for information that is still in use and may be accessed frequently in the near future
  - **Throughput** is still favored over latency
    - Large amounts of data must be written/read efficiently
  - **Sequential** workloads mainly (random accesses may also happen)
    - Sometimes one may want to retrieve/update specific parts of backup data (e.g, specific files or even small portions of these)
  - **Data can now be updated** (typically in a sporadic fashion)
    - In some cases, **only diffs** (modified data) **are stored** across backups

Storage Workloads

Primary Storage[1] (not only RAM!)

- Storage support for databases, data analytics, AI frameworks, VMs ...
  - **High-throughput** and/or **low-latency** are now desirable
    - Large amounts of data may be written/read (throughput)
    - Small-sized writes/reads must be done efficiently (latency)
  - **Sequential** and **random** workloads
    - Content (e.g., of files) may be partially accessed and out of order
  - **Data-** and **metadata-intensive** workloads
    - E.g., frequent access to the content (data) of files but also to their metadata
  - Data is expected to be **updated frequently**

[1]Definition taken from: Paulo, J and Pereira, J. 2014. ***A Survey and Classification of Storage Deduplication Systems***. ACM Computing Surveys

Persistent Storage Devices

And the main workloads these target…

- **Tape**
  - Used for archival data
  - Reliable and cheap
  - Limited support for random accesses and in-place updates
- **HDD**
  - Used for archival, backup, and (still in some cases) primary data
  - Still cheap, with support for random accesses and in-place updates
- **SSD** (includes SATA/NVMe SSDs)
  - Used for backup and primary data
  - More expensive than HDDs but faster, specially for random accesses
- **Persistent Memory**[1]
  - Used for primary storage (mainly used as cache[2])
  - Speed closer to RAM for sequential and random workloads, but expensive

[1] Intel Persistent Memory is now discontinued

[2] Don’t forget that RAM (main memory) is also used as cache for persistent data :)

Hard Disk Drives (HDDs)

Case study

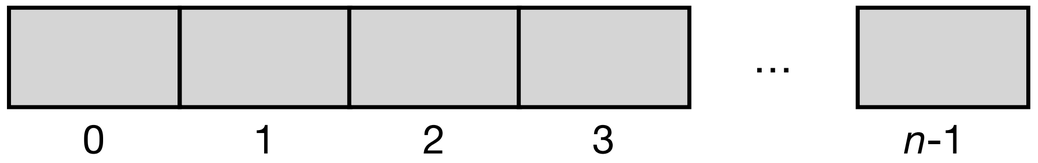
- Why study modern HDDs?
  - These have been the main form of persistent data storage for decades
  - They shaped the design of widely used file systems
- Questions
  - How do HDDs store data, and what is their interface?
  - How is data laid out and accessed on these disks?
  - How does disk scheduling improve performance?

Hard Disk Drives (HDDs)

Interface

- Composed of sectors (typically 512-byte blocks) that can be read or written
  - For a disk with *n* sectors, these are numbered from 0 to *n*-1
  - A disk can be seen as an array of sectors (0 to *n*-1 is the disk's **address space**)
- **Multi-sector operations** are possible
  (e.g., many file systems read or write 4KB (or more) at a time)
  - However, the drive manufacturers only guarantee that a single 512-byte write is **atomic** (i.e., it will either complete in its entirety or it won't complete at all)
  - If a power loss occurs, only a portion of a larger write may be completed (**torn write**)

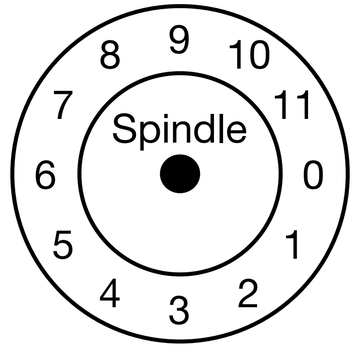
Logical representation of a disk
(an array of sectors 0 to *n*-1)



Hard Disk Drives (HDDs)

Geometry of an HDD

- The **platter** is a circular hard surface on which data is stored
  - Each platter has two sides, each of which is called a **surface**
- Data is encoded on each surface in concentric circles of sectors named **tracks**
  - A single surface contains thousands and thousands of tracks!
- A drive may have one or more platters bound around a **spindle**. The spindle is connected to a motor that rotates the platters at a given fixed rate
  - The rate for modern drives is in the 7200 to 15000 **rotations per minute (RPM)**
  - Assuming a 10000 RPM, a single rotation takes 6 milliseconds
- The **disk head** is used to read or write a specific sector (one head per surface)
  - The **disk arm** moves across the surface to position the head on the right track

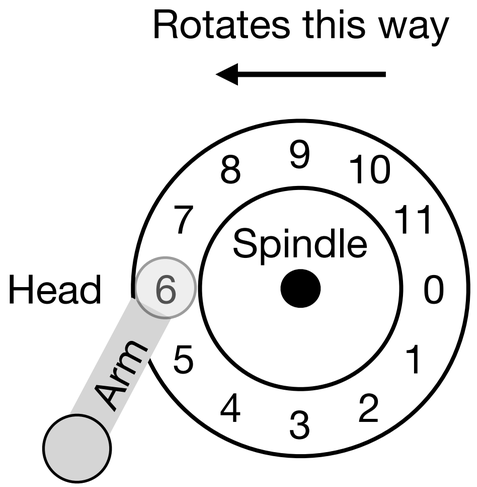


A disk with just a single track

Geometry of HDDs

A single track disk

- **Example:** a disk with a single track (sectors 0 to 12)
  - The disk head, attached to the end of the arm, is placed at sector 6
  - The platter rotates counter-clockwise
- A request to **sector 0** is done
  - One must wait for sector 0 to be under the disk head (i.e., wait for the platter to rotate)
  - If a full disk rotation takes *R*, for this example, one must wait half of *R* (**rotational delay**) for sector 0 to be under the disk head
- What sector request has the highest rotational delay for this example?

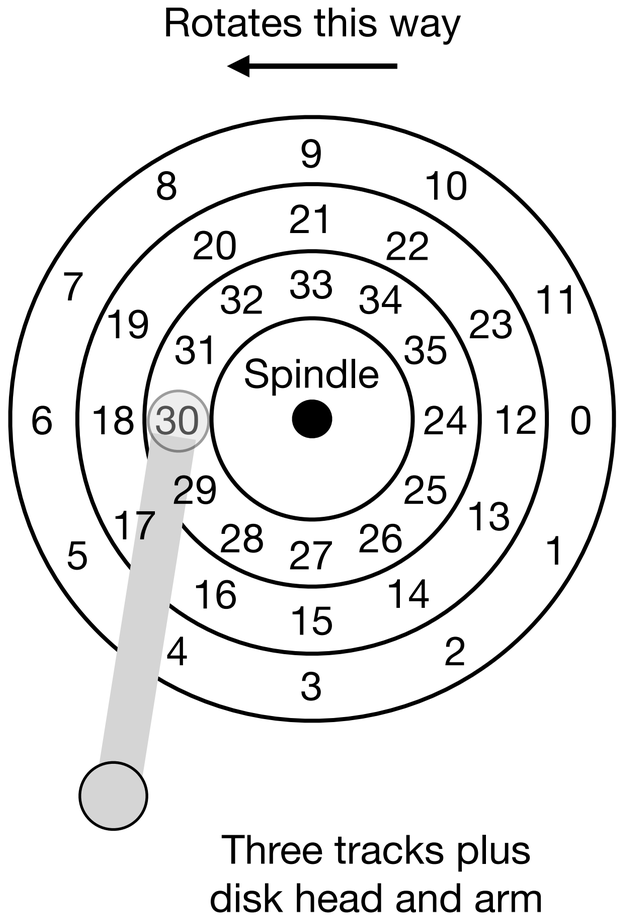


A single track plus disk head and arm

Geometry of HDDs

A multiple track disk

- **Example:** disk with 3 tracks, each with 12 sectors
  - The disk head is placed at sector 30 (innermost track)
  - The platter rotates counter-clockwise
- A request to **sector 11** is done
  1. The arm must first move to the outermost track (**seek** phase)

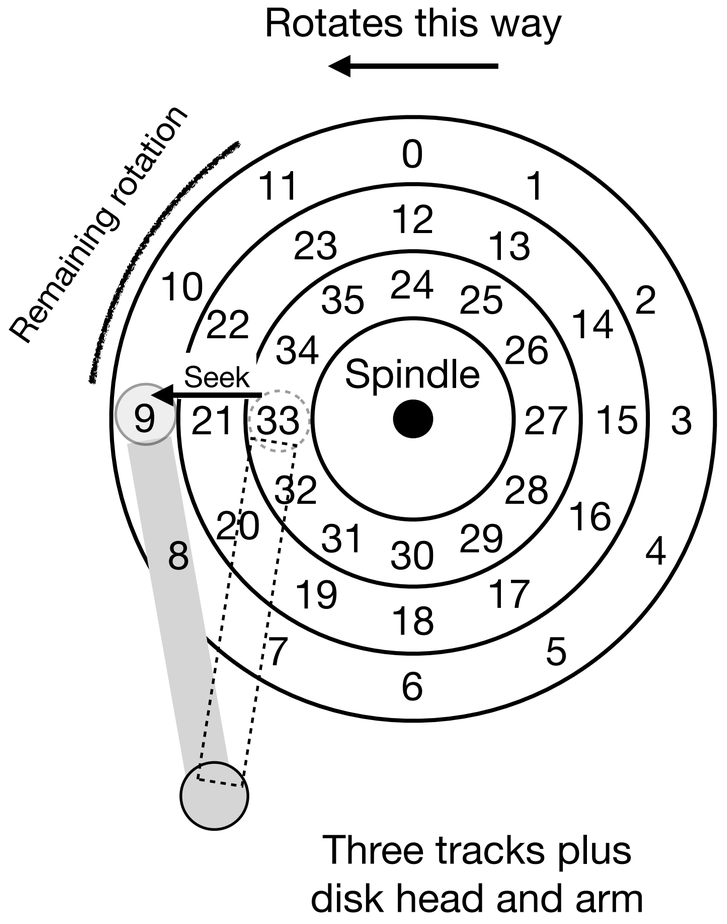


Three tracks plus
disk head and arm

Geometry of HDDs

A multiple track disk

- **Example:** disk with 3 tracks, each with 12 sectors
  - The disk head is placed at sector 30 (innermost track)
  - The platter rotates counter-clockwise
- A request to **sector 11** is done
  1. The arm must first move to the outermost track (**seek** phase)
  2. When the seek is done, the platter has rotated (e.g., the head is passing through sector 9). One must wait for sector 11 to be under the head (**rotation** phase)

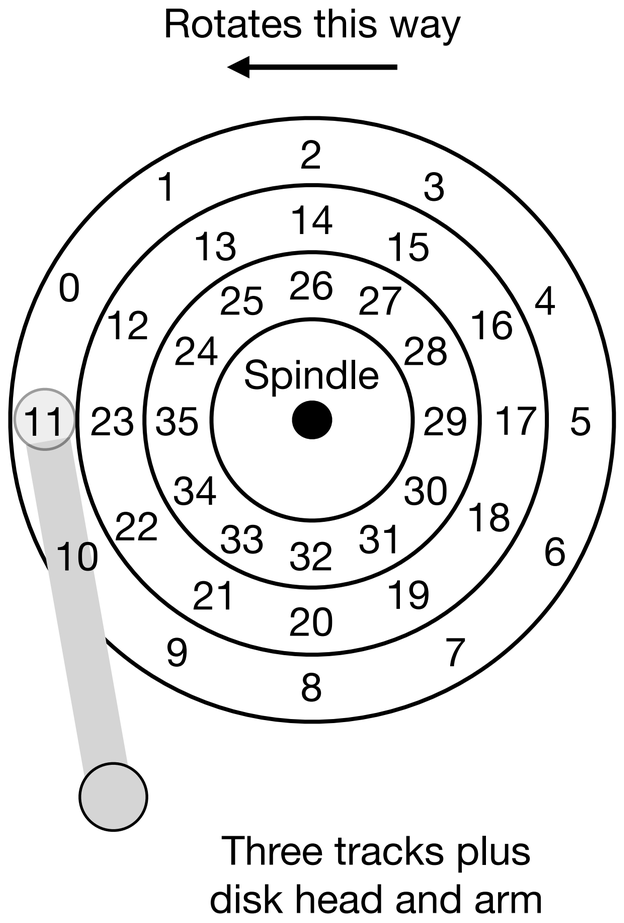


Three tracks plus disk head and arm

Geometry of HDDs

A multiple track disk

- **Example:** disk with 3 tracks, each with 12 sectors
  - The disk head is placed at sector 30 (innermost track)
  - The platter rotates counter-clockwise
- A request to **sector 11** is done
  1. The arm must first move to the outermost track (**seek** phase)
  2. When the seek is done, the platter has rotated (e.g., the head is passing through sector 9). One must wait for sector 11 to be under the head (**rotation** phase)
  3. When sector 11 is under the head, data can be read from or written to disk (**transfer** phase)

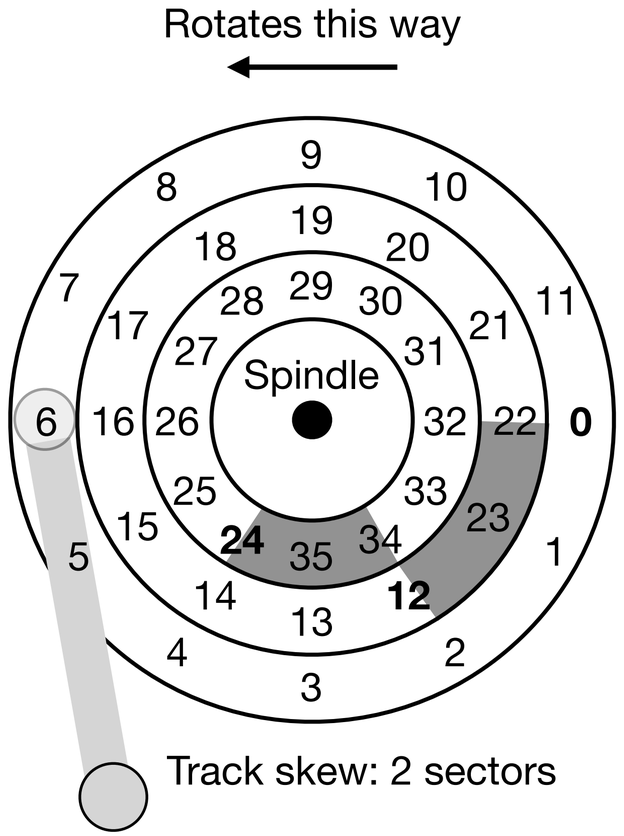


Three tracks plus disk head and arm

Geometry of HDDs

Some considerations

- To ensure that sequential operations are properly served when crossing track boundaries, sectors of tracks may be **skewed**
  - **Compensates seek time**, i.e. when the head reaches the right track, the desired sector has not rotated under the head yet
- Outer tracks tend to have more sectors than inner ones (more space available) and can be organized into zones (**multi-zoned** disk drives)
- Disks usually have a **small cache** (e.g., 8, 16, 32 MB) to hold data read from or written to disk
  - **Important:** When should the disk acknowledge the completion of a write request?
    - When it is on the cache (**write-back**) or when it has actually been written on the disk platter (**write-through**)?



Hard Disk Drives (HDDs)

I/O time and rate

- The I/O time to read/write a given sector ($T_{I/O}$) depends on
  - the time to **move the disk arm** to the right track (T_{seek})
  - the time to **place the disk head** under the right sector ($T_{rotation}$)
  - the time to **read/write** (transfer) the sector ($T_{transfer}$)

$$T_{I/O} = T_{seek} + T_{rotation} + T_{transfer}$$

- The rate of I/O (i.e., bytes served per unit of time - $R_{I/O}$) is calculated by **dividing** the **amount of data to transfer** ($Transfer_{size}$) by the **time it took** ($T_{I/O}$)
  - E.g., If a 1MB transfer requires 10 ms of I/O time (0.010 seconds), then our disk provides a rate of 100 MB/s for requests of 1MB

$$R_{I/O} = Transfer_{size} / T_{I/O} = 1 / 0.010 = 100\ MB/s$$

I/O Time and Rate Analysis

Based on the workload…

- I/O time and rates vary according to the workloads the disks are serving
- **Sequential workloads:** Reading a large number of contiguous sectors per I/O request (e.g., 1MB) only requires a single seek and rotation before a long transfer time
  - I.e., When the transfer of sector *n* is done, the disk head is now under sector *n*+1 (assuming the sectors are on the same track…)
- **Random workloads:** Issuing small I/O operations (e.g., 4KB) to non-contiguous sectors will require a seek, rotation, and transfer per request
- **Conclusion:** Whenever possible, transfer data to and from disks sequentially. When such is not possible, do random accesses with large payloads (block sizes)
  - See **Chapter 37.4** - Remzi H. Arpaci-Dusseau, Andrea C. Arpaci-Dusseau. **Operating Systems: Three Easy Pieces**. Arpaci-Dusseau Books, 2018 for a more extensive analysis with two real disk examples!

Hard Disk Drives (HDDs)

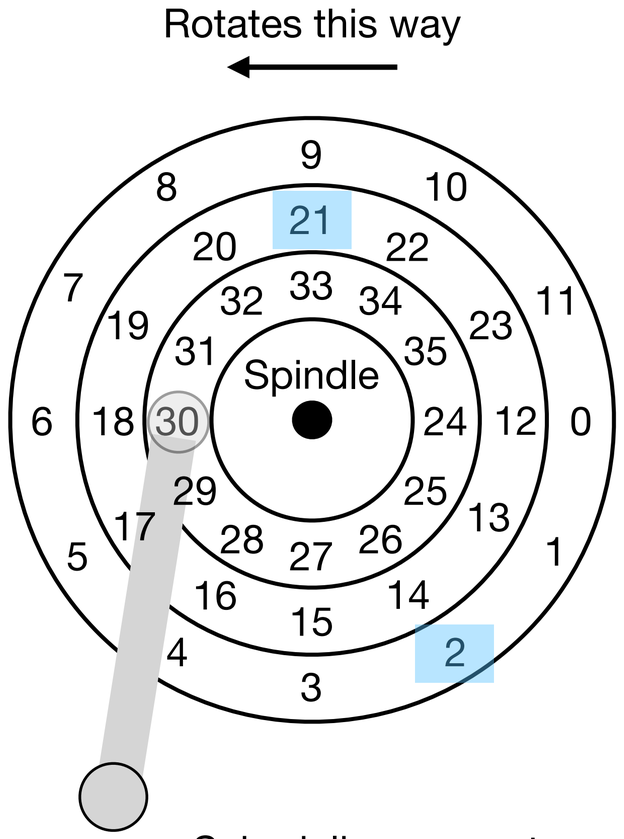
Disk scheduling

- Due to the high cost of disk I/O, deciding how a set of I/O requests are asked to the disk plays an important role in performance
  - This is a task for the **disk scheduler**
- **Question:** When the disk scheduler has several I/O requests waiting to be served (on an I/O queue), which one(s) should be done first?
- **Good news!**
  - Unlike in process scheduling, where the duration of each job is unknown, in disk scheduling, we can estimate the seek time and rotational delay of a given request, thus, having an idea of the time required to serve it
  - Let's look at possible scheduling policies

Scheduling Policies for HDDs

Shortest Seek time First (SSTF)

- **Selection criteria:** Serve first the requests for the nearest track (shortest seek time)
- **Example:** Head on sector 30 (inner track), with pending requests for sectors 2 and 21
  - **Decision:** First serve sector 21 (middle track) and then 2 (outer track)
- **Implementation issue:** The drive geometry (disposition of sectors across tracks) is not known by the OS (it just sees an array of sectors)
  - **Solution:** The OS uses a **nearest-block-first (NBF)** policy to schedule requests. Nearest blocks in our sector array will correspond to near tracks at the disk geometry
- **Main Drawback:** SSTF suffers from **starvation**
  - Imagine a steady stream of requests coming to the inner track. Requests from other tracks will never be served!

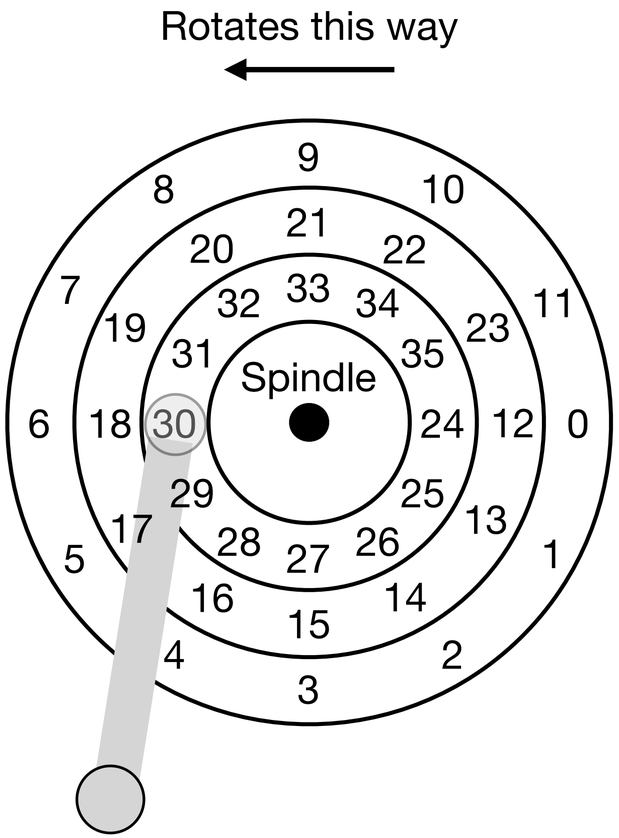


Scheduling requests to sectors 2 and 21

Scheduling Policies for HDDs

Elevator (aka SCAN or C-SCAN)

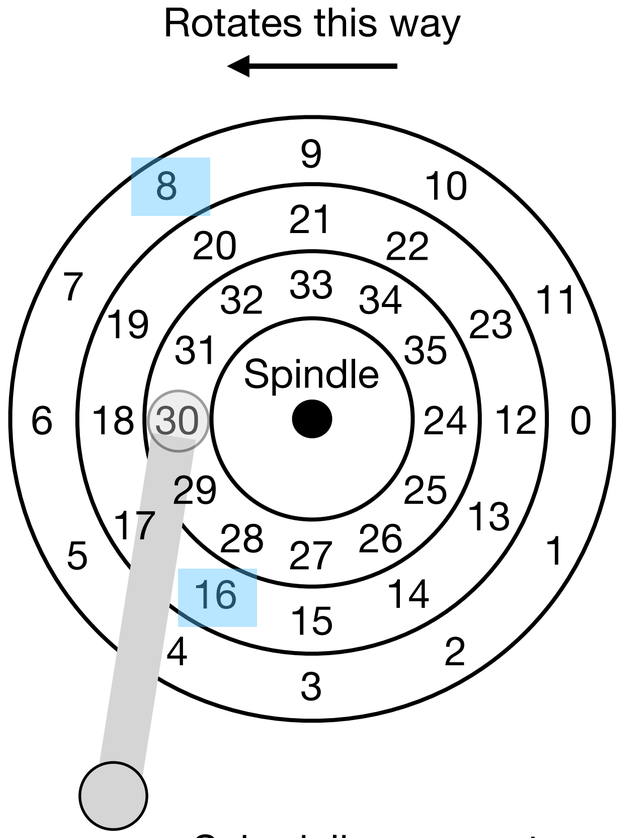
- **SCAN:** Move across the disk from inner to outer tracks (or the other way around) i.e., do a disk **sweep**
  - If a request comes for a sector that has already been passed through on this sweep, queue it for the next sweep (**avoids starvation of far-away requests**)
  - In SCAN, sweeps are done in both directions (e.g., sweep from inner to outer tracks, then sweep from outer to inner tracks, and then repeat…)
- **Interesting fact:** SCAN is also known as the **elevator algorithm** due to its sweep strategy. Imagine going from floor 10 to 1, somebody getting in at floor 3 and pressing 4, and the elevator going to the 4th floor because it is closer…
- **Circular SCAN (C-SCAN):** Sweeps disk tracks in a single direction (e.g., sweep from outer to inner tracks, reset, and start again at the outer tracks)
  - SCAN sweeps tracks back and forth, favoring middle tracks (after serving the outer track, it passes through the middle track twice before returning to the outer one)
  - C-SCAN, a variant of SCAN, is a bit **more fair for inner and outer tracks**



Scheduling Policies for HDDs

Shortest Positioning Time First (SPTF)

- So far, our algorithms **only consider seek time** (ignoring rotation time)
  - **Example:** Head on sector 30, with requests pending for sectors 16 and 8
  - Which request should be serviced first? It depends…
- If **seek time is high** and **rotation time is small**, then SSTF is a nice approach (i.e., **sector 16** should be served first)
- What if **rotation time is high** and **seek time is small**?
  - Serving **sector 8** first would be faster in terms of disk arm and head positioning. Serving sector 16 first would require almost a full disk rotation for the head to be over this sector
- Seek and rotation times are roughly equivalent in modern disks, making sense to consider both to choose the **shortest positioning time**
  - As the OS may not have the necessary knowledge to implement the SPTF algorithm (i.e., the position of the head, track boundaries, …), this strategy is usually implemented in the disk drive (at the **drive's controller**)



Scheduling requests to sectors 8 and 16

Scheduling Policies for HDDs

Final remarks

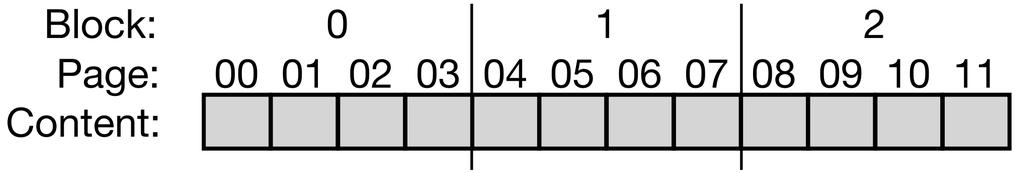
- In older systems, disk scheduling would be done by the OS. In modern systems, the OS picks a few requests and lets the disk controller decide the best way to schedule them (e.g., by following SPTF)
  - Nonetheless, even in modern systems, the OS implements important optimizations
- **I/O merging:** Requests to contiguous sectors/blocks[1] are typically merged by the OS before being sent to disk
  - **Important** to submit less requests to disk, better leverage its I/O bandwidth, and make disk workloads more sequential
- Therefore, in many cases, the OS waits (queues) some I/O requests from programs (i.e., it follows a **non-work-conserving** approach) before issuing these to the disk. Not issuing requests immediately (i.e., not following a **work-conserving** approach), allows for scheduling and merging optimizations
  - The tricky part, as usual, is deciding how many requests to queue before sending them to the disk

[1]A block can be seen as group of one or more contiguous sectors

Solid State Drives (SSDs)

Brief overview

- After decades of dominance of HDDs, a new form of persistent storage gained traction, the **Solid-State Drives (SSDs)**
  - The technology underlying these is known as **flash**, more specifically NAND-based flash
- SSDs have no mechanical of moving parts, as in HDDs. Bits are stored on transistors
  - No arm seek time or head rotation time!
  - No mechanical errors (e.g., head crashes)
- The disk is organized into flash chips, each containing two different-sized units
  - **Blocks** (also called **erasure blocks**), which are typically of size 128KB or 256KB
  - Inside each block, there are multiple **pages**[1], each with a few KBs in size (e.g., 4KB)



A simple flash chip with 3 blocks, each with 4 pages

[1] Do not confuse the term page here with memory pages. CS people tend to use the same term, i.e., page, block, …, for different contexts

Solid State Drives (SSDs)

Interesting characteristics and challenges

- Flash chips allow **reading single pages very efficiently**, regardless of the position of the page on the device and the location of the previous request
  - **Good for random read workloads** (unlike HDDs)!
- Before writing/updating a page, the **full block** containing that page must be **erased**
  - **Expensive** operation **performance-wise** that erases the content of all pages at the block (i.e., the content of valid pages must be copied somewhere else before erasing their block)
- Once a block has been erased, the corresponding pages can be **programmed** (i.e., written with new content)
  - More efficient (time-wise) than erasing a block, but less efficient than reading a page

Solid State Drives (SSDs)

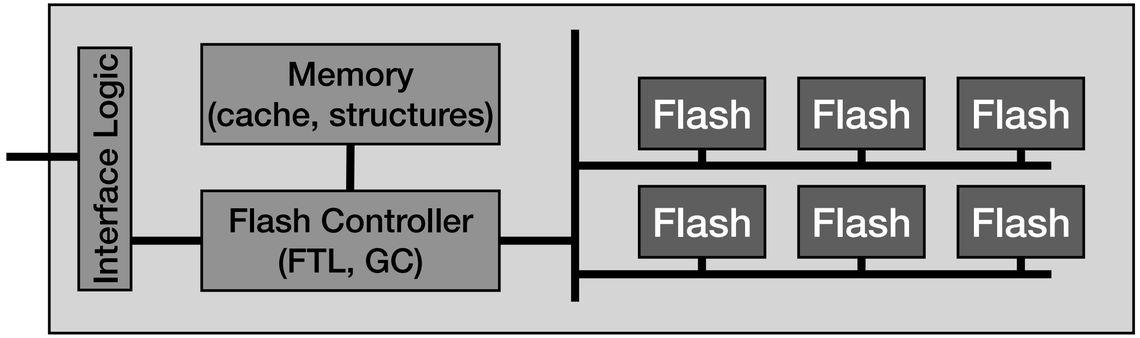
Wear out and disturbance

- Erasing a block also **wears out** the lifespan of the SSD drive
  - Each block can only be erased and programmed a certain amount of times (e.g., 10000 times)
  - This challenge motivated several **wear leveling algorithms** that spread erase/program operations (i.e., write load) across all the blocks of the device evenly to extend its lifespan
- Also, sometimes, when programming (writing to) a page within a flash chip, neighboring pages may have some bits flipped, a phenomenon known as **disturbance**

Solid State Drives (SSDs)

Providing the disk abstraction

- How are multiple flash chips provided as a traditional storage device?
  - The SSD device offers a standard block (sector) interface atop the flash chips
- The **Flash Translation Layer (FTL)** maps **logical addresses** (device interface) to the corresponding **physical blocks and pages** (flash chips)
- The **Garbage Collector (GC)** tracks **unused pages** and **erases blocks**
- **Data structures** for the FTL and GC, along with **data caches**, are kept on volatile memory (SRAM) at the device



Logical Diagram of a Flash-based SSD

Summary
HDDs vs SSDs

- When compared to HDDs, SSDs
  - Excel in **random write and random read** performance
  - Provide better performance for **sequential workloads** (although the difference is smaller)
- Remember that performance gains typically increase **monetary costs**
- Although in SSDs, the gap between sequential and random workloads' performance is smaller, it still pays off (especially for I/O-demanding applications) to **reduce random I/O** operations

Data Integrity and Protection

Disk failure modes

- Disks can **fail**. When such happens, it can lead to data **unavailability** and even to data **loss**
  Let's quickly go through common disk failure modes
- **Fail-stop model:** Either the disk is working, or it fails completely
- **Fail-partial model:** The disk fails partially, i.e., some sectors[1] are not accessible. Two types of single-block[1] failures are common and worthy to consider
  - **Latent sector errors (LSEs):** A sector (or a group of these) is damaged (e.g., head disk crash, cosmic rays)
    - In-disk **error correcting codes (ECC)** can be used to determine if sectors are damaged and, in some cases, to fix them
    - Disks have **spare sectors** that are used to replace bad sectors. Done by the **disk controller**, transparently to the OS!
  - **Corruption:** In some cases, sectors (blocks) may become corrupt in a way not detectable by the disk
    - **Silent faults** are tricky as the disk may return corrupt data to the program
    - **Checksums** (small digest of a block[2]), typically stored along with the block's content, can be used to detect data corruption (i.e., the checksum of previously stored data must match the one from the content being read)

[1] Since these failures and protection mechanisms are applicable at different layers of the I/O stack, we use the terms block and sector interchangeably

[2] Different algorithms can be used to calculate checksums such as XOR, CRC, …

Data Integrity and Protection

Disk failure modes

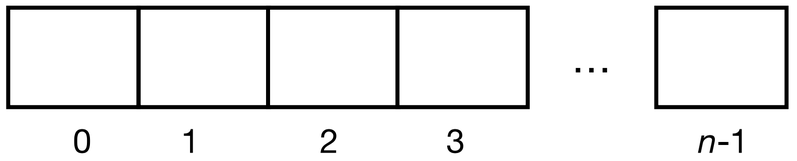
- **Misdirected write:** Content is written to the wrong disk offset (e.g., due to a bug in the disk controller)
  - One can add the physical location of the block to the checksum
- **Lost writes:** The disk tells the OS that a write is done, but in fact the content was never persisted
  - One can read each block after writing it to be sure, but this has a high-performance penalty
  - Some systems store the checksum in a different disk location (i.e., instead of storing it together with the sector/block). The probability of having a bad confirmation from the disk for two independent writes is smaller than for a single one…
- **Checksumming** is an important feature for data resilience. Indeed, critical systems typically **scrub** their disks, validating all the checksums of stored content to detect, at an early stage, potential data corruption
  - However, checksumming introduces **space overhead** (additional disk space for storing checksums) and **processing overhead** (checksums must be calculated when data is written/read)
- Finally, when a disk fully or partially fails, what can we do to recover back our data?
  - One needs **redundancy**!

Redundant Array of Inexpensive Disks (RAID)

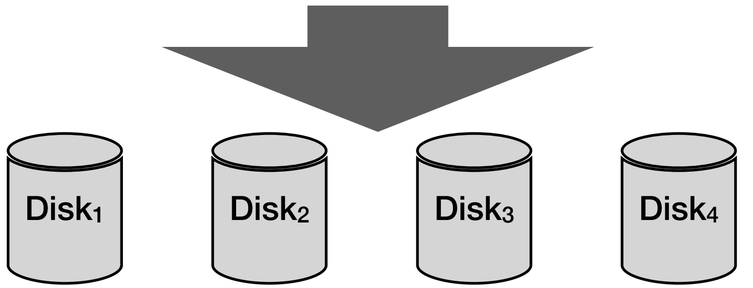
Technique and abstraction

- **RAID:** Technique that uses multiple disks in concert to build a faster, bigger and more reliable disk system
- **Externally**, a RAID system looks like a single disk, i.e., a group of blocks one can read and write - **transparency**!
- **Internally**, it is a complex system that requires multiple disks and a controller (with volatile memory and a processor). RAID can be implemented through hardware or software

Logical representation of a RAID disk system
(an array of blocks 0 to *n*-1)



Physical representation of a RAID disk system
(multiple disks persisting the content of blocks)



Redundant Array of Inexpensive Disks (RAID)

RAID Level 0: Striping

- **RAID level 0:** Blocks (e.g., 4KB) are spread across several disks in a round-robin fashion
  - Blocks in the same row form a **stripe** (in our example, blocks 0, 1, 2, and 3 are in the same stripe)
- Advantages
  - **More capacity** (4x increase in storage space in our example)
  - **Enhanced parallelism** (performance) when accessing contiguous chunks (i.e., a stripe)
- Challenges / disadvantages
  - Smaller blocks lead to better parallelism, but small disk accesses are costly (remember the HDD slides?) **Finding the right block size** for striping is not always easy
  - RAID 0 **does not provide reliability**. If one disk fails, data is lost

Disk 0	Disk 1	Disk 2	Disk 3	
0	1	2	3	stripe
4	5	6	7	
8	9	10	11	
12	13	14	15	

Redundant Array of Inexpensive Disks (RAID)

RAID Level 1: Mirroring

- **RAID level 1:** Blocks are copied across multiple disks (it can be combined with RAID level 0)
- Advantages
  - **Resiliency**, i.e., if one disk fails, its blocks can be recovered from the mirror disk
  - Data can be read from multiple disks (possibility of having **read parallelism** and **load balancing** optimizations)
- Challenges/disadvantages
  - **Capacity-wise**, one does not take advantage of having more disks
    (In our example, we only have the capacity of one disk to store data but require two for redundancy)
  - **Writes** must be **done at all mirrored disks**. If done in parallel, it can take as much time as the slowest disk
- How can one **reduce the extra capacity** needed by RAID 1 while **maintaining resiliency**?

Disk 0	Disk 1
0	0
1	1
2	2
3	3

mirror

RAID 1 over two disks

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4	Disk 5	Disk 6	Disk 7
0	0	1	1	2	2	3	3
4	4	5	5	6	6	7	7
8	8	9	9	10	10	11	11
12	12	13	13	14	14	15	15

RAID 1+0 (stripe of mirrors)

Redundant Array of Inexpensive Disks (RAID)

RAID Level 4: Parity Block

- **RAID level 4:** Adds a parity disk. This disk holds a parity block (calculated with XOR) for each stripe of blocks
- Advantages
  - **Tolerates 1 disk failure**, i.e., if one of the disks fails, the parity block can be used to reconstruct the stripe
  - **Read request parallelism** is possible across all disks (except for the parity one)
  - Only requires **1 additional disk** (less storage space overhead than RAID 1+0)
- Challenges / Disadvantages
  - Every time a block is updated, the parity block for the stripe must be recalculated (which may include reading the content of other blocks at the stripe). Thus, the **parity disk is a performance bottleneck**!

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4	
0	1	2	3	P0	
4	5	6	7	P1	Parity blocks
8	9	10	11	P2	
12	13	14	15	P3	

Redundant Array of Inexpensive Disks (RAID)

RAID Level 5: Rotating Parity

- **RAID level 5:** Spreads parity blocks across the RAID disks
- Advantages
  - As in RAID 4, it **tolerates 1 disk failure** with **a single extra disk**
  - Performance-wise, **read requests can take advantage of all disks** (as parity blocks are spread across disks[1])
  - Spreading the parity blocks also **alleviates the parity recalculation bottleneck**
- Challenges / disadvantages
  - It **does not fully eliminate the overhead of parity recalculation** (but avoids rewriting parity blocks always at the same disk)
  - **Harder to implement** and only tolerates a single disk failure. There are other variants of RAID (e.g., RAID level 6) to tolerate multiple disk faults

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
5	6	7	P1	4
10	11	P2	8	9
15	P3	12	13	14
P4	16	17	18	19

Parity blocks

[1] Note that parity blocks still cannot be used to serve read requests. But since now blocks are spread across the 5 disks, the aggregated read bandwidth increases

More Information

- **Chapters 36, 37, 38, 44 and 45** - Remzi H. Arpaci-Dusseau, Andrea C. Arpaci-Dusseau. **Operating Systems: Three Easy Pieces.** Arpaci-Dusseau Books, 2018.
- Avi Silberschatz, Peter Baer Galvin, Greg Gagne. **Operating System Concepts (10. ed)**. John Wiley & Sons, 2018.

Questions?