Secondary Storage

Chapters: 11 (11.6, 11.7, 11.8 are optional)

# Overview

* The Memory Hierarchy
* Magnetic Disks Drives (HDD)
  + Technology
* – Performance
  + Scheduling
  + Scheduling Algorithms
* Solid-state Drives (SSD)
  + Read/write
  + SSD vs HDD
* Technology Update

# Traditional Secondary Storage

* Main memory can be accessed **directly from the CPU**. It uses **byte storage**
* Secondary storage uses **block access**. The CPU receives a block of code at a time to process.
* Characteristics:
  + **Large**: 500 - 4000GB and more (HDD)
  + **Cheap**: 0.035gbp/GB (HDD)
  + **Slow**: millisecond (HDD)
  + **Persistent**: data survives power loss
  + **Fail** **rarely**
    - Drive dies
* Most used block size: 512 bytes of 4kB.

# Magnetic Disks: DescriptionDiagram Description automatically generated

* Uses a **magnetic disk** that spins and is read by a read/write head.
* Each disk arranged into **consecutive circles** or **circular tracks**, split into **sectors**.
* Each sector has fixed size and is smallest unit of transfer.
* Data stored using magnetic field
* Initially **physical geometry**  was used for addressing
  + “**head** go to **cylinder** x to **sector** y (**CHS**)
* More modern systems use **logical block addressing** (**LBA**)
  + Hard disk is treated as single large device, simply counting existing blocks starting at 0.
  + Then the device controller handles the conversion into a CHS address.

## Chart Description automatically generatedDisk Performance

* Depends on:
  + **Seek time**: moving the disk arm to the correct cylinder. This depends on how fast the arm can move
  + **Rotation** (**latency**): waiting for the rector to rotate under the head. This depends on rotation rate
  + **Transfer time**: time taken to transfer data from surface of disk to controller. Depends on density of bytes on disk.
* OS attempts to **minimise** these costs, specifically **seek** and **rotation**.

## Software Performance

* To reduce seeking, the OS can:
  + increase file block size
  + co-locate “related” items which could be:
    - Blocks of the same file
    - Data and metadata for a file
* The OS can also keep data and metadata in memory to reduce physical disk access. This **avoids** **slow disk accesses** but could potentially waste valuable memory.
* The OS can also fetch blocks into memory before requested. This would **hide** the **slow disk access**.

## Performance vs. Block Scheduling

* A picture containing diagram

  Description automatically generatedTo reduce the seeking time, we use **block scheduling**
* Multiple applications request disk accesses. The OS will:
  + maintains the **requests queues**
  + generate **transfer commands** to/from the disk(s) i.e. it is the software that communicates with the device drivers.
* To **reduce** application **waiting time**:
  + OS modifies **order** of block requests queued waiting on the disk
    - This is traditionally based on cylinder number
* There are multiple **block-scheduling** algorithms implemented in the **OS**, including:
  + FCFS (first come first served i.e. no scheduling)
  + SSTF (shortest seek time first)
  + SCAN (elevator algorithm)
  + C-SCAN (typewriter)
* Layers:
  + Application (user)
  + Virtual File System (VFS, kernel)
  + Block layer with queues and scheduling algorithm
  + Device driver
* In order for the head to navigate to the right cylinder, an LBA will be converted into a CHS.

### FCFS

* Chart, line chart

  Description automatically generatedFirst come first served
* Example:
  + queue: 98, 183, 37, 122, 14, 124, 65, 67
  + head starts at 53
* Clearly high seeking time.
* Reasonable when load is high, but long waiting time for long request queues

### SSTF

* Graphical user interface, chart, line chart

  Description automatically generatedShortest seek time first
* Example:
  + queue: 98, 183, 37, 122, 14, 124, 65, 67
  + head starts at 53
* This minimises **arm movement** and **maximises request rate**
* Does however lead to clustered blocks and favours middle of disk.

### SCAN

* Chart, line chart

  Description automatically generatedDisk starts **at one end** of disk and moves to the other, servicing requests until it gets to the other side of the disk. Then **reverses direction** and continues**.**
* Example:
  + queue: 98, 183, 37, 122, 14, 124, 65, 67
  + head starts at 53
* Otherwise known as **elevator algorithm** aselevators have the same algorithm.
* Works well when requests are uniformly dense, but requests at one end will need to wait for head to process requests up to spindle and back to edge

### C-SCAN

* Chart, line chart

  Description automatically generatedSimilar to SCAN, except instead of reversing the head returns to original end and begins servicing again.
* Provides a more **uniform wait time**.
* Example:
  + queue: 98, 183, 37, 122, 14, 124, 65, 67
  + head starts at 53

## Selecting A Disk Scheduling Algorithm

* When there is **one request** all algorithms behave like FCFS
* SCAN and C-SCAN perform better for systems with **heavy load** on the disk (less starvation)
* Generally performance depends on **numbers** and **types** of requests
* Request for disk can be influenced by:
  + File-allocation method
  + Metadata layout
* The block scheduling algorithm is a **module** of the OS, so it can be easily replaced.
* Linux uses:
  + **Deadline**: variation of C-SCAN with two queues
  + **NOOP**: variation of FCFS
  + **CFQ**: uses concept of timeslices

# Solid-State Drives

* No moving drives: use chips.
* Diagram

  Description automatically generatedEssentially a minicomputer with an embedded processer and flash controllers.
* SSD handles its own geometry
* Different technologies: NOR, NAND, 3D XPoint, Memristor etc.
* Multiple interfaces USB, SATA, mSATA, NVMe (M.2, PCIe) etc.
* Laptops work better with SSD as there are no moving parts

## SSD Reads

* Unit of read is a **page**, typically 4kB
* COTS SSD handles
  + Approximately 100k reads per second
  + 10-100 microseconds latency
    - This is 50x-1000x better than magnetic disks
  + 50-5000 MB/s read throughput:
    - At least 1-10x better than magnetic disks
* Read access time is (mostly) **independent** of device geometry, so block scheduler is unecesary.

## SSD Writes

* Unit of write is also a page
  + Less writes/s than reads/s
  + Higher write latency than read latency
  + Lower throughput than read
* Flash media **must be erased** before it can be written, HDDs can just overwrite
* Unit of erase is a block, typically 64-256 pages
  + Takes approx. 1ms to erase a block
  + Can only be erased a certain number of times before unusable (typically 10,000-1,000,000 times)
  + This results in **write amplification**: a phenomenon that happens when writing data to a storage media.
    - The original amount of the write is multiplied – due to several factors – into a disproportional amount of data on the storage device.
* To extend lifetime, we can use firmware called **Flash Translation Layer** (**FTL**).
* This erases the data transparently, and handles **wear levelling**
* Wear Levelling arranges the data so that write/erase cycles are distributed evenly among all blocks in the SSD.
* This is obviously important due to the limited number of times that a block in a SSD can be overwritten.

# SSD vs HDD

* **Capacity**:
  + Flash SSDcost min£1/GB, 1 TB costs approx. £100
  + 1 TB hard drive costs £35: **much cheaper**
* **Energy efficiency**
  + SSD typically more efficient due to lack of mechanical moving parts
    - SSD uses 1-2 watts of power
    - HDD uses approx. 10 watts of power
* Physical Resistance
  + SSD has no moving parts
  + HDD cannot work correctly if subject to physical accelerations and movements

# Technology Update - NOT IN EXAM

* Everybody desires:
  + **Faster** (secondary) storage, as fast as main memory
  + **Larger** main memory, as large as (secondary) storage
  + **Persistent**: not losing our data
* New invention: **non-volatile memory** (NVM)/**Persistent Memory** (PM, PMem)
* “New RAM” that is also persistent
* Same form of RAM and it is plugged into the address bus, but it can keep data.
* Uses some form of a buffer