Operating Systems

Input/Output, HDDs, SSDs

Thomas Ropars

thomas.ropars@univ-grenoble-alpes.fr

2023

References

The content of this lecture is inspired by:

- The lecture notes of Prof. David Mazières.
- *Operating Systems: Three Easy Pieces* by R. Arpaci-Dusseau and A. Arpaci-Dusseau

Other references:

- *Modern Operating Systems* by A. Tanenbaum
- *Operating System Concepts* by A. Silberschatz et al.

In this lecture

The mechanisms involved in the interactions between the OS and the I/O devices

- Polling vs Interrupts
- Programmed I/O vs Direct Memory Access
- Drivers

The characteristics of Hard Disk Drives and the associated challenges

- The hardware
- Scheduling of disks I/O

A glimpse on Solid State Drives based on Flash Memory

I/O: an important topic

Motivation

- Without I/O, computing is useless.
- It is the main purpose of most programs. (eg, editing a file, browsing web pages)

All kinds of I/O devices

- mouse/keyboard
- disk/cdrom/usb stick
- network card
- screen/printer

A hardware/software infrastructure is required to interact with all these devices.

Agenda

Interacting with an I/O device

Drivers

Basic Geometry of a disk

Scheduling disk I/O

Flash-based SSDs

Agenda

Interacting with an I/O device

Drivers

Basic Geometry of a disk

Scheduling disk I/O

Flash-based SSDs

The global picture

Communication between the processor/OS and a device

Synchronization/interactions – 2 main approaches:

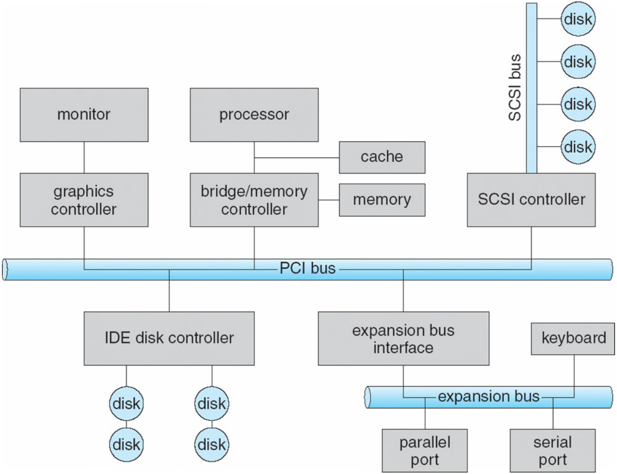
- Polling
- Interrupts

Data transfers

- Communication through a Bus
- Two main techniques:
  - Programmed I/O
  - Direct Memory Access

The I/O Bus

Figure by Silberschatz et al



Controller = collection of electronics that operates a bus or a peripheral device

The I/O Bus

A bus is a communication system interconnecting several devices.

A hierarchical architecture

- A general I/O bus (PCI).
  - Connects the processor-memory subsystem to higher performance devices (video card, network card, etc.)
- One or several peripheral buses to connect other devices (USB, SATA)
  - Connects to disks, keyboard/mouse, etc.

Why hierarchical?

- Performance: performance decreases with the length of the bus
- Cost: designing a highly efficient bus is costly (and not useful to all devices)

Interactions between the OS and an I/O device

Device interface

Exposes some registers managed by the controller:

- Status: Used by the device to inform about its status
- Command: Used by the OS to control the device
- Data: For data exchanges between the OS and the device

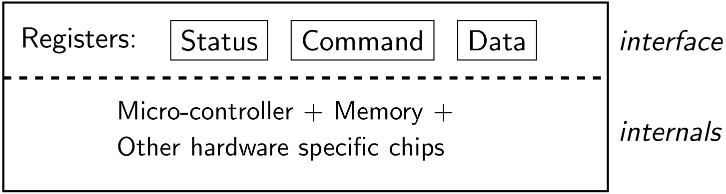


Figure: A canonical device

Interactions between the OS and an I/O device

A typical execution – Executing a command on a device

1. The OS uses the status register to detect when the device is not *BUSY*.
2. The OS writes a chunk into the data register and sets the command register.
   - The controller sets the status to *BUSY*.
3. The controller reads the command and the data register, and launch the execution of the command.
4. The OS detects when the command has been executed based on the status register.
   - The controller clears the command and resets its *BUSY* status once the command has been executed. It set its status to *ERROR* if needed.

Interactions between the OS and an I/O device

Questions

- How does the OS checks when the device is done?
- How are data transferred between the processor and the device?

Polling and Programmed I/0

The basic solution

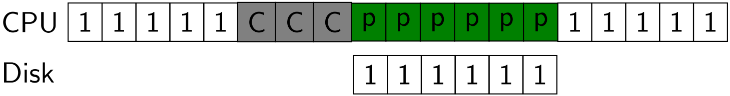


Figure: '1': doing work for process 1; '*C*': Copying data; '*p*': Polling

- Polling: The OS repeatedly checks the status of the device
- Programmed I/O: the main processor is involved in the data transfer

Synchronization based on interrupts

About interrupts

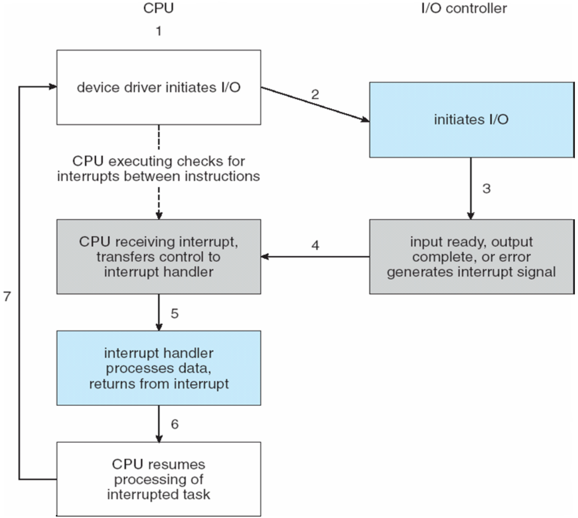
- Hardware mechanism that allows a device to interrupt the processor
  - The device controller can raise an interrupt
- The CPU hardware has a wire called the interrupt-request line (several in practice)
  - The CPU senses it after executing every instruction

Handling interrupts

- The CPU catches the interrupt and dispatches it to the interrupt handler
- The interrupt handler determines the cause of the interrupt and performs the necessary processing
- After running the handler, the CPU is restored to the execution state prior to the interrupt.

How do interrupts work? [1]

Figure by Silberschatz et al



[1] A description about the implementation of interrupts (together with references) is provided at the end of the slides

Execution with interrupts

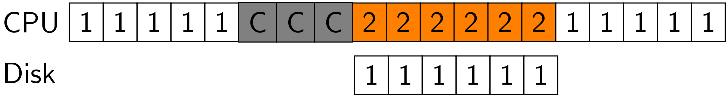


Figure: '1': Doing work for process 1; '*C*': Copying data; '2': Doing work for process 2

Polling vs Interrupts

- Using interrupts allows putting process 1 to *sleep* until the I/O is completed.
  - The scheduler can schedule another process on the CPU (avoids wasting CPU cycles)
  - It would be difficult to predict when to poll in the future.
- Polling can be efficient if the device is ready very rapidly

Interrupts are not always better than polling

Hybrid approach

- Handling an interrupt is costly (hundreds of cycles)
- Hybrid approach: The best of both world
  - Start by polling
  - If the device is not ready, put calling process to wait and schedule another process

Interrupts are not always better than polling

Hybrid approach

- Handling an interrupt is costly (hundreds of cycles)
- Hybrid approach: The best of both world
  - Start by polling
  - If the device is not ready, put calling process to wait and schedule another process

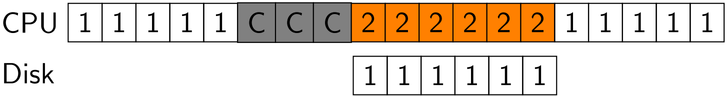
Problem of livelock

- The processor receives so many interrupts that it only processes interrupts and never allows a user-level process to run
  - Example: A network interface receives a lot of messages and sends an interrupt for each of them.
- Solutions:
  - Use polling
  - Interrupt coalescing: wait before sending interrupts until several requests have been completed

Improving data transfer performance

Execution with interrupts and PIO

(C = copy of a single word)



Problem with programmed I/O

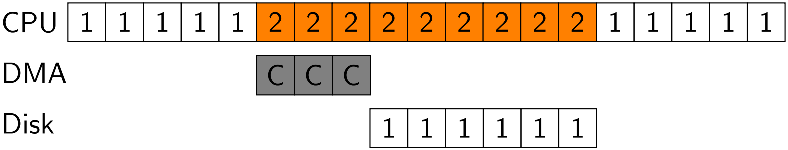
- The processor wastes CPU cycles for every word
- What if a large amount of data has to be output to the device?

Direct Memory Access (DMA)

Direct Memory Access engine

A DMA engine is a specific device that orchestrates data transfer between memory and I/O devices without CPU intervention.

- The OS writes a command to the DMA engine with the source address, the destination address and the amount of data to transfer.
- The DMA engine sends an interrupt to the CPU when the transfer is done.



Agenda

The problem

Context

- We would like the OS to be as general as possible (work on any hardware)
- Each device can have a very specific interface

An example: a file system

We would like to open a file but it could be stored on different I/0 devices:

- A disk (different kinds)
- A USB stick
- A CD

Drivers

Keywords

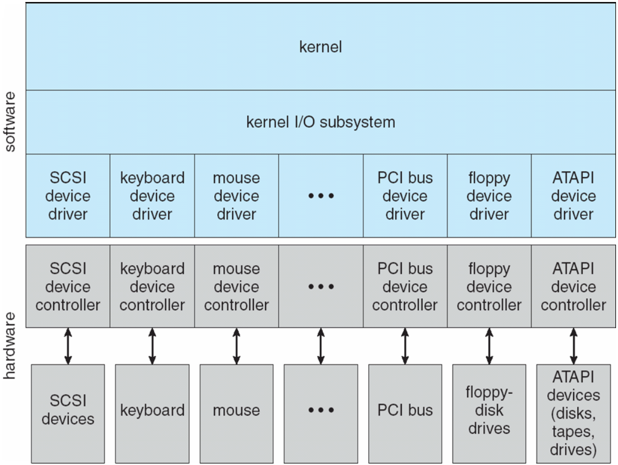
- Abstraction
- Encapsulation
- Software layering

A piece of software must know in detail how a device works: this is the Device Driver.

- The driver exposes a generic interface to the rest of the OS.
- Any new device should come with a driver that implements (at least part of) the standard I/0 interface to be usable.

Drivers

Figure by Silberschatz et al



About drivers

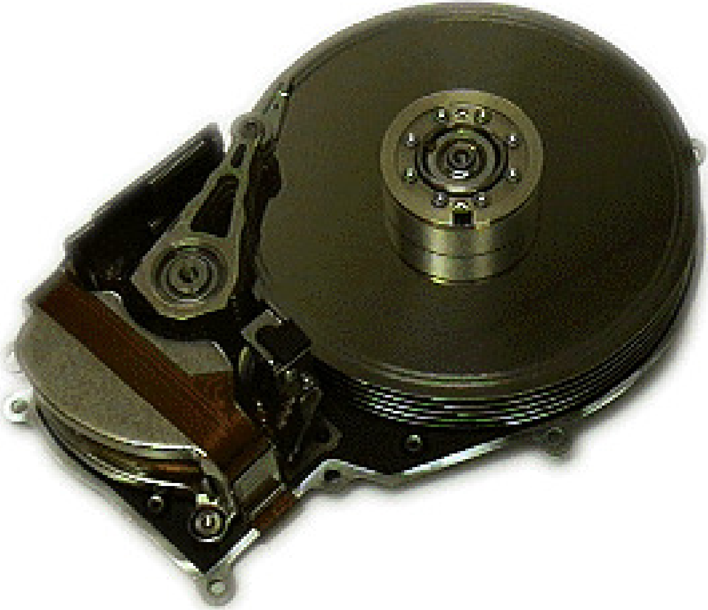
Drawbacks

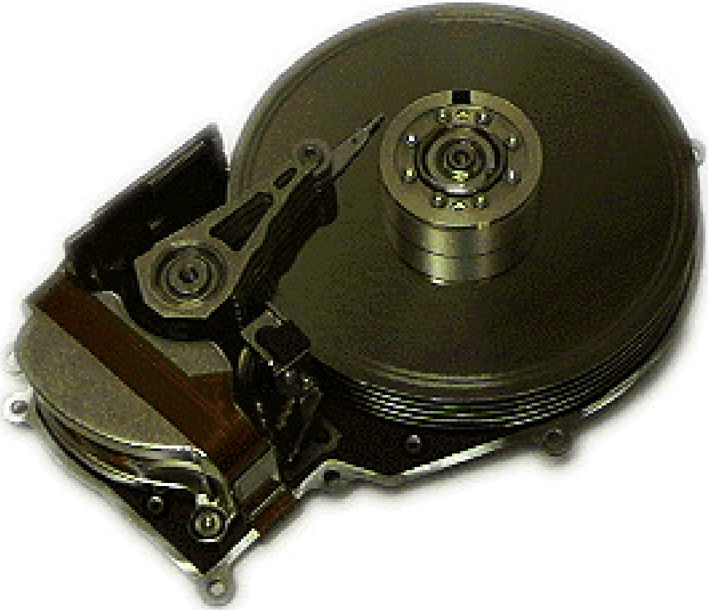
- The generic approach might prevent from taking advantage of advanced features of the hardware
- Example: SCSI devices provide rich error reporting. The Linux I/O interface only reports generic I/O errors.

In the kernel

- In 2001, drivers were accounting for 70% of the kernel code
- Of course it is not all active at the same time
- Many bugs are in the drivers

The case of Hard Disk Drives





Agenda

Storage on a magnetic platter

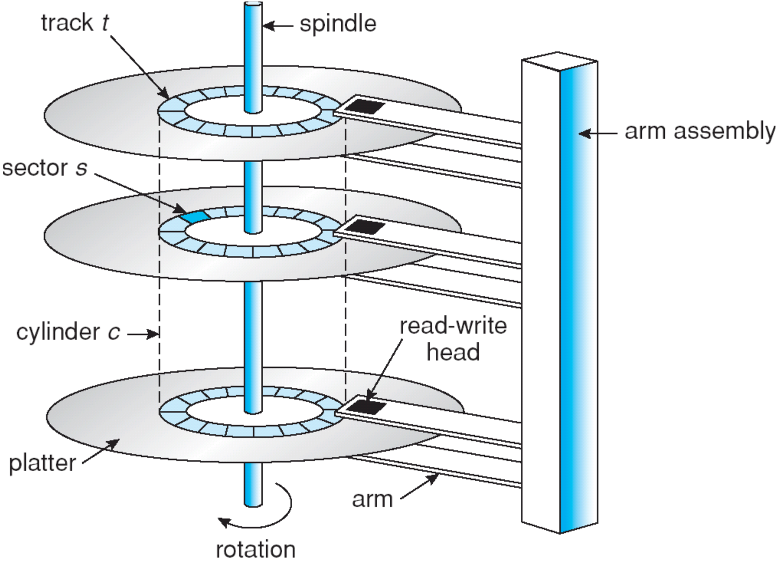
- Platter: a circular hard surface on which data is stored persistently by inducing magnetic changes to it.
  - A disk may have one or multiple platters.
- Surface: One side of a platter
  - Data is encoded on each surface
- Tracks: A surface is divided into concentric tracks.
  - Many thousands of tracks on a surface
  - Hundreds of tracks fit into the width of a human hair
- Cylinder: A stack of tracks of fixed radius is a cylinder

Storage on a magnetic platter

- Head/Arm: Reading or writing is accomplished by a disk head attached to a disk arm.
  - One head per surface
  - Heads record and sense data along tracks
  - Generally only one head is active at a time
- Sector: A track is divided into 512-byte blocks called sectors
  - Sectors are numbered from 0 to $n-1$ (n-sector disk)
  - Multi-sectors operations are possible (eg, update 4 Mb at a time)
  - A sector is the granularity for atomic operations.

Cylinders, tracks, & sectors

Figure by Silberschatz et al



Accessing a sectors: Seeks

A seek is the action of moving the head from its current track to the track containing the target sector.

4 phases

- Acceleration: accelerate arm to max speed or half-way point
- Coasting: move at max speed (for long seeks)
- Slowdown: stops arm near destination
- Settle: adjusts head to actual desired track
  - Is a costly operation (0.5 to 2 ms)
  - The hard drive must be certain to find the right track!

Accessing some sectors

Other delays:

- Rotational delay: Time for the target sector to pass under the disk head.
  - Rotating speed of modern disks: 7,200 RPM to 15,000 RPM (RPM= rotations per minute)
- Transfer time: Time for I/O to take place.

$$\text{I/O Time} = \text{Seek time} + \text{Rotational delay} + \text{Transfer time}$$

About performance

Comments about performance

- Accessing sectors that are close is faster
- Accessing contiguous sectors is faster than random access

Cache

Disks may use a cache to improve observed performance

- Read and cache consecutive sectors
- Caching writes can be dangerous (breaks atomicity)

Agenda

Interacting with an I/O device

Drivers

Basic Geometry of a disk

Scheduling disk I/O

Flash-based SSDs

Context

The OS should decide in which order to execute I/O on the disk to optimize performance

- Contiguous accesses are better
- Try to avoid long seeks.

Differences with process scheduling

Context

The OS should decide in which order to execute I/O on the disk to optimize performance

- Contiguous accesses are better
- Try to avoid long seeks.

Differences with process scheduling

- It is possible to estimate seek time and rotational delay (the "*future*").
- A strategy similar to SJF can be applied!

First Come First Served (FCFS)

Process disk requests in the order they are received

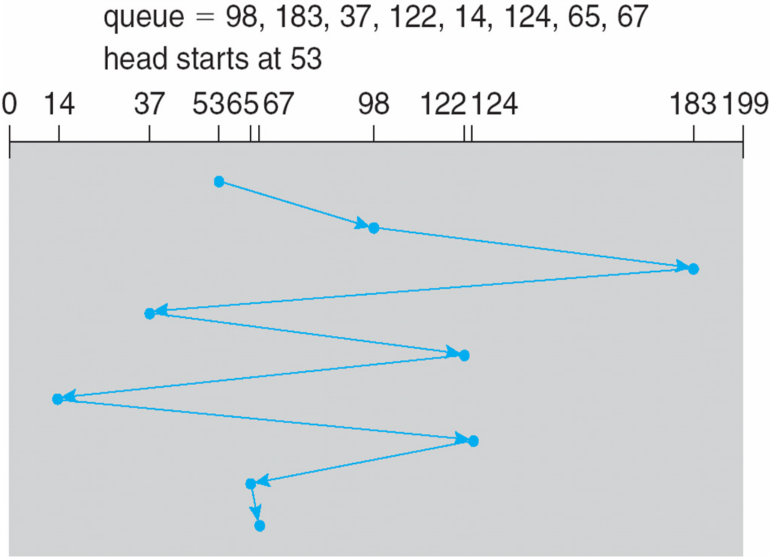
Advantages

- Easy to implement
- Good fairness

Disadvantages

- Cannot exploit locality of requests
- Increases average latency, decreases throughput

FCFS example [1]



[1] The numbers are track ids

Shortest seek time first (SSTF)

Always pick request with shortest seek time

Advantages

- Exploits locality of disk requests
- Higher throughput

Disadvantages

Shortest seek time first (SSTF)

Always pick request with shortest seek time

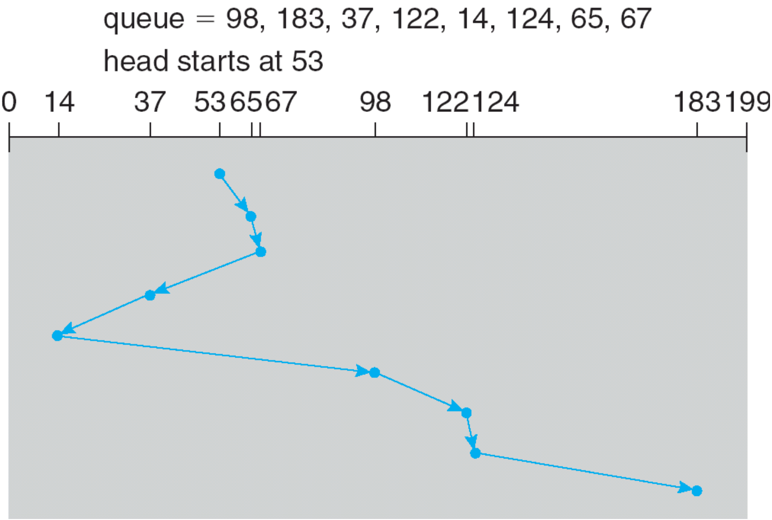
Advantages

- Exploits locality of disk requests
- Higher throughput

Disadvantages

- Starvation (some aging strategy could be used to fix the problem)
- The OS does not always know what request will be the fastest
  - The OS does not have direct access to the disk geometry (position of the sectors)

SSTF example



“Elevator” scheduling (SCAN)

Sweep across disk, servicing all requests passed

- Like SSTF, but next seek must be in same direction
- Different variants:
  - Switch directions only if no further requests (SCAN)
  - Back to first track when no further requests (Circular-SCAN)

Advantages

- Takes advantage of locality
- Bounded waiting

Disadvantages

“Elevator” scheduling (SCAN)

Sweep across disk, servicing all requests passed

- Like SSTF, but next seek must be in same direction
- Different variants:
  - Switch directions only if no further requests (SCAN)
  - Back to first track when no further requests (Circular-SCAN)

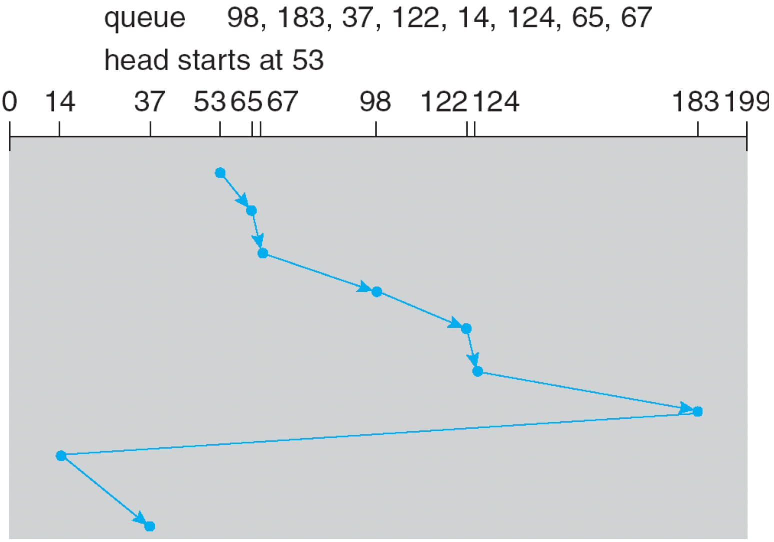
Advantages

- Takes advantage of locality
- Bounded waiting

Disadvantages

- Might miss locality SSTF could exploit

CSCAN example



More on scheduling

- Some strategies try to mix SSTF and SCAN
  - VSCAN(r): Apply SSTF but with a weight *r* to account for the direction

- All presented strategies only take into account seek time
  - Rotational delay might be as important as seek time
  - SPTF (Shortest Positioning Time First) tries to do this
  - However rotational delay is hard to evaluate at the OS level

Scheduling with modern disks

Features of modern disks

- Disks can accommodate multiple outstanding requests
  - The OS can send multiple requests to the disk without waiting for completion
- Disks include sophisticated schedulers
  - They can implement SPTF accurately!
- Disks can also do I/O merging
  - Wait for multiple I/O requests to try to merge consecutive ones in a single multi-blocks request

Interactions with the OS

- The OS issues a few request (tries to select best from its point of view)
- The disk applies advanced scheduling to those requests

Agenda

Flash memory

NAND-Based Flash

- Transistor storing one or multiple bits
- Single-level Cell
  - Store one bit per cell
  - Fast – high endurance – expensive
  - Industrial usage
- Multi-level cell
  - Store several bits per cell (eg, 3)
  - Slower – lower endurance – cheaper
  - Used in USB keys and SSDs

Flash chips structure

- Chips are organized in banks
- Banks are divided in blocks (eg, 256 KB)
- Blocks are divided in pages (eg, 4 KB)

Operations on data

Reading

- Granularity: a page
- Performance: **10s of microseconds**
  - 2 order of magnitude faster than rotating disks

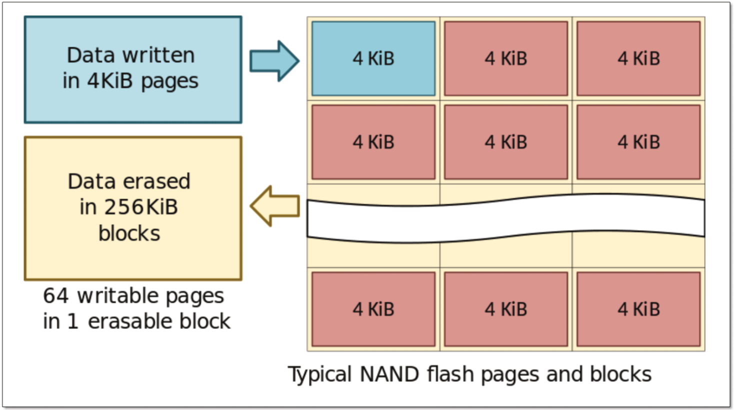
Writing

Writing requires **erasing a block** before writing (programming) a page.

- Erasing a block
  - Destroys the content of the block by setting all bits to 1
  - Requires copying first the data that should not be lost
  - Performance: A few milliseconds
- Programming a page
  - Setting some bits to 0 by writing a page
  - Performance: 100s of microseconds

Reading and writing to flash memory

Figure by D. Nosachev



Challenges associated with Flash memory

Write performance

- Overwriting a page is costly and complex
- Need to minimize the write amplification
  - The ratio between the size of *logical* writes and *physical* writes.

Wear out

- The number of times a block can be programmed/erased is limited ($O(10000)$)
  - Extra charge is accumulate in the cells on erase operation
  - When the charge is too high, it becomes impossible to differentiate between 0 and 1.
- Need for wear leveling
  - Ensure that all blocks wear out more or less at the same time

From Flash to Flash-based SSDs

Solid-state drive (SSD) = A device that store data persistently using integrated circuits without any involvement of moving mechanical parts.

Basic description

- Offers 512-byte sector read/write operations based on addresses (classical storage device interface)
- A SSD includes:
  - Some number of flash chips
    - Accessing multiple chips in parallel increases performance
  - Some amount of volatile memory
  - Control logic to orchestrate device operations
    - Implements a flash translation layer

Flash translation layer

- Transforms logical operations into internal flash operations

Implementation of FTL

A log structure

- Creation of a log: On a logical write of a block[1], the block is appended to the end of the log
  - Limited write amplification
  - Good wear-leveling
- A mapping table stores the address of the logical blocks
  - Stored in volatile memory (with enough info in flash memory to reconstruct it)
- Garbage collection is needed
  - Complex and costly operation
  - Find garbage pages and reclaim the dead blocks
    - Might require copying valid pages

[1] A logical block typically corresponds to a physical page

References for this lecture

- *Operating Systems: Three Easy Pieces* by R. Arpaci-Dusseau and A. Arpaci-Dusseau
  - Chapter 36: I/O devices
  - Chapter 37: Hard Disk Drives
  - Chapter 44: Flash-based SSDs

- *Operating System Concepts* by A. Silberschatz et al.
  - Chapter 13: I/0 systems

Additional Slides

How do interrupts work?[1]

- The controller raises an interrupt
  - The CPU hardware has a wire called the interrupt-request line (inf fact multiple IRQs)
  - The CPU senses it after executing every instruction
- The CPU catches the interrupt and dispatches it to the interrupt handler
  - The CPU performs a state save and jumps to the interrupt handler routine at a fixed address in memory.
- The handler clears the interrupt by servicing the device
  - The interrupt handler determines the cause of the interrupt and performs the necessary processing
  - New interrupts on the line are ignored while the handler is running
    - Tasks executed inside an interrupt handler should be small.
  - After running the handler, the CPU is restored to the execution state prior to the interrupt.

[1] To know more: `https://www.safaribooksonline.com/library/view/understanding-the-linux/0596005652/ch04s06.html`

How to select the proper interrupt handler?

Basic solution

Check all devices to find which one is ready.

- Problem: there can be many devices to check.

Interrupt dispatching

- Several interrupt lines are available that correspond to offsets in a table called the interrupt vector
  - Each entry in the vector contains a pointer to an interrupt handler
- Problem: The host might include more devices than the number of entries in the vector
  - Use interrupt chaining (ie, each entry points to a list of handlers)

More on interrupts

Masking and priorities

- Some interrupts are maskable (handling can be deferred), some are not (eg, errors).
- Priorities between interrupts can be defined
  - A high-priority interrupt can preempt the execution of a low-priority interrupt