BTI1341 – Operating Systems

Part 3: Persistence – 10) I/O Devices

Autumn 2025-26 `master@352a46f (20250901-155000)`

P. Mainini / E. Benoist / C. Fuhrer / L. Ith

Berner Fachhochschule | Haute école spécialisée bernoise | Bern University of Applied Sciences

Outline

- I/O Devices
- Device Drivers
- Hard Disks
- Appendix

I/O Devices

Introduction

So far, we were mostly concerned about efficiently using the CPU and memory (RAM). In this final part, we look at persistence :

- How to persist (i.e. permanently store) data?
- How to handle failures (computer crashes, disk failures, ...)?
- How to store and find data?

For this, a *basic* understanding of I/O devices is required first:

- What are I/O devices?
- How can they be accessed and used by the OS?
- What are potential issues and how can they be solved?

Bus Hierarchies

I/O devices are typically attached to some sort of bus or interconnect , e.g.:

- Memory bus
- General I/O bus (often a PCI derivative)
- Peripheral buses
  - SATA, SCSI
  - USB
- etc.

The need for different technologies comes from physics: Faster signals require shorter wires and are more affected by connectors.

Over the years, many different types of buses have been used and disappeared, while development still continues.

System Architecture I

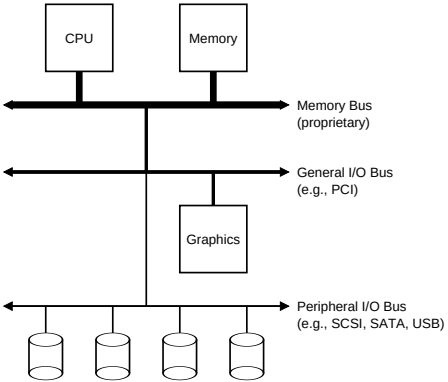

Figure: Generic System Architecture

Courtesy of [ADAD18]

System Architecture II

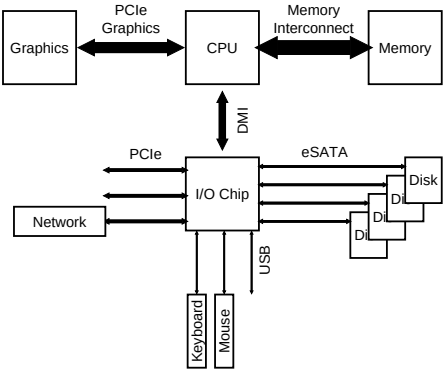

Figure: Intel Z270 Chip Set (Kaby Lake, 2017)

Courtesy of [ADAD18]

Bus Data Rates

The following table lists some typical bus data rates:

Bus/Device	Data rate
Keyboard	10 bytes/sec
Mouse	100 bytes/sec
Scanner (300 dpi)	1 MB/sec
4x Blu-ray disc	18 MB/sec
802.11n Wifi	37.5 MB/sec
USB 2	60 MB/sec
Gigabit Ethernet	125 MB/sec
SATA 3 disk	600 MB/sec
USB 3	625 MB/sec
PCIe 3.0 (1 lane)	985 MB/sec
USB 4	5 GB/sec

Sources: [TB14], Wikipedia

A Typical I/O Device

In the following, we assume a typical I/O device, consisting of:

- Hardware interface
  - Electrical and mechanical specifications etc.
  - *A defined protocol for interaction*
- Internal structure
  - Hardware implementation
  - *Not relevant from an OS point of view*

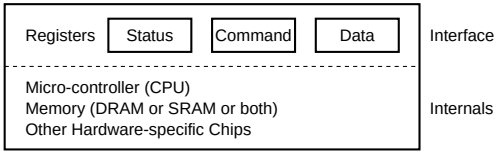

Figure: A Typical Device

Courtesy of [ADAD18]

The Hardware Interface

The hardware interface of a device consists of a series of registers , through which the OS can interact with it. We distinguish three kinds of registers:

- Status registers
  Can be read to learn about the current state of a device. Typically contain multiple flags (bits), representing various aspects of the device's state.
- Command registers
  Can be written for passing commands to the device, i.e. actions it should carry out next.
- Data registers
  Typically, a command needs or returns some data. Reading and/or writing this data uses one or more data register(s).

The Interface Protocol

In general, interacting with a device follows a protocol, which looks more or less the same for most types of devices (pseudocode):

```
while(true) {
    while(STATUS_REGISTER == BUSY)
      ;   // wait for the device to become ready

    read data from DATA_REGISTER
    // or
    write data to DATA_REGISTER

    write command to COMMAND_REGISTER
}
```

Programmed I/O (PIO)

The protocol from the previous slide is known as polling : It *repeatedly* reads `STATUS_REGISTER` to learn about the state of the device.

Occasionally, a data exchange takes place by reading/writing data registers and telling the device what to do next. *This requires work by the CPU.*

If CPU interaction is required for I/O, we call it programmed I/O (PIO) .

Clearly, such a protocol is inefficient and wasting CPU cycles while:[1]

1. *Waiting and querying* the device until it is ready.
2. *Transferring data* to/from the device (possibly in many chunks).

[1] Consider that devices are orders of magnitude slower than the CPU.

I/O Using Interrupts I

The first issue can be solved using interrupts , which allow for overlapping of I/O and computation:

1. Send command (and data) to the device.
2. Put the calling process to *sleep* and switch to another one.
3. Later, when the device has finished the operation, it generates an *interrupt*.
4. The OS interrupt handler is triggered and the result may be returned to the process.

⚠ Note that this is not always a better solution:

- *For very fast devices*, polling may still be a better solution (avoids context switches).
- Bursts of interrupts may lead to livelocks .

I/O Using Interrupts II

A graphical comparison between polling and using interrupts can be seen in the following two figures:

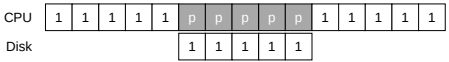

Figure: Disk Access Using Polling

Courtesy of [ADAD18]

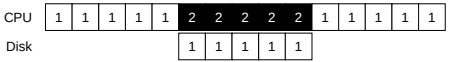

Figure: Disk Access Using Interrupts

Courtesy of [ADAD18]

Direct Memory Access (DMA) I

To solve the second issue, Direct Memory Access (DMA) has been introduced.

Instead of having the CPU copying data around, a *DMA controller* (hardware) can be programmed with memory locations and devices to which to read from and write data to. The DMA controller then handles all the work and raises an interrupt when done.

Example: *Without DMA*, the CPU handles data transfer from/to disk for process 1. This prevents process 2 from running:

Direct Memory Access (DMA) II

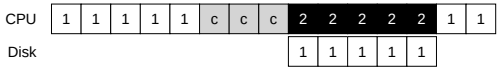

Figure: CPU Transferring Data

Courtesy of [ADAD18]

Direct Memory Access (DMA) III

With DMA, the DMA controller takes care of transferring data, which enables the CPU to run process 2 meanwhile:

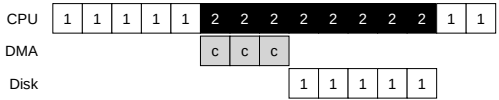

Figure: Data Transferred by the DMA Controller

Courtesy of [ADAD18]

Device Interaction Methods

So far, the question of *how to access the hardware interface registers* has not been discussed. There are two important methods:

- Explicit I/O instructions
  The CPU has specific instructions for dealing with hardware I/O registers (e.g. "`in`" and "`out`" on x86). Typically, they make use of *I/O ports* to address a device.

- Memory-mapped I/O
  I/O registers are made available as *memory locations* (similar to memory-mapped file I/O seen earlier). Regular load/store CPU instructions (e.g. "`mov`") can then be used to access them.

Both approaches are in use today.

Device Drivers

Motivation

Due to the large amount of different I/O devices, a *layer of abstraction* is required:

- Different device types are used for the same purpose. Example: A file system works independently of the underlying disk type (e.g. SATA or a SCSI).
- Implementation highly depends on the hardware interface provided by a device.

Device drivers are used for implementing such abstractions.[2] Today, they often represent the larger portion of the OS kernel code!

[2] On the downside, an abstraction can only provide a generic interface. This might hide additional device features.

Abstraction Layers

OSes typically include multiple layers of abstraction :

Hardware interface → device driver → generic interface → API (e.g. POSIX) → user space.

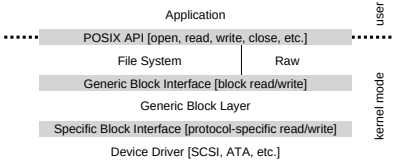

Figure: Example: Abstraction Layers of a File System Stack.

Courtesy of [ADAD18]

Common *generic interfaces* are block devices and raw devices (sometimes also called character devices).

I/O Subsystems

Drivers are generally organized in the I/O subsystem of an OS kernel. It contains all the driver code and provides common interfaces.

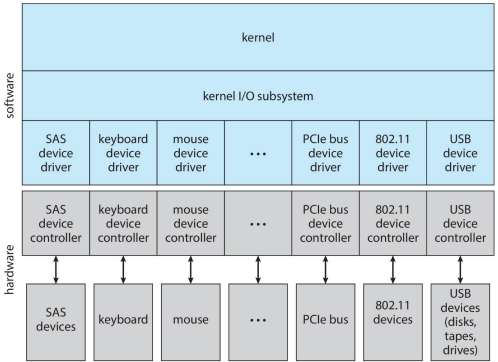

Figure: Overview of I/O Subsystems

Source: [SGG19]

Hard Disks

Introduction

Hard Disk Drives (HDD) are the prevalent form of persistent data storage since decades. To better understand file system details in the following lessons, their functioning principle is now described.

In this lesson, only “classical” hard disk drives with *rotating, mechanical disks* are discussed. NVM storage , e.g. Solid State Drives (SSD) , which became more and more important in recent years will be discussed later in the course.

On the next slide, the first HDD (shipped in 1956), can be seen: The *IBM Model 350*. It stores roughly 3.75 MiB (using 6-bit characters), weighs over 1 ton, and could only be leased at 750$ per month ([com]).

The First Hard Disk Drive

Figure: 1956: Shipment of the First Hard Disk

Source: [com]

HDD Basics

HDDs consist of a large number of 512-byte blocks (or sectors), which can be *addressed individually*.[3] These are numbered $0 \ldots n-1$ on a disk with n blocks, thus the disk address space goes from 0 to $n-1$.[4] A single block is guaranteed to be *written atomically* (e.g. in case of power failure).

Internally, a HDD has multiple rotating disks (called "platters"), fixed on a fast rotating spindle (e.g. 10'000 RPM, or one rotation every 6 ms). On every disk side, blocks are encoded in concentric tracks. A disk head attached to an arm moves across the tracks to read and write data.

A schematic view of these components is given in the figure on the next slide.

[3] Today, HDDs also use block sizes of 4096 bytes ("advanced format").

[4] So-called Logical Block Addressing (LBA).

HDD Basics: Components

One side of a HDD platter with three tracks, 35 blocks (sectors) and a disk head.

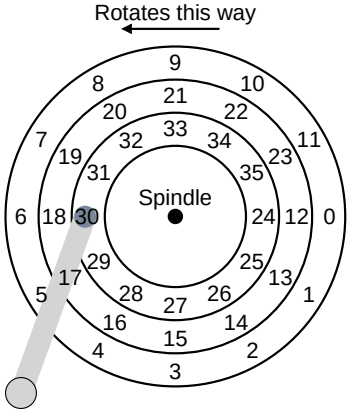

Figure: HDD Components

Courtesy of [ADAD18]

Disk Latency

Compared to RAM, HDDs are orders of magnitude slower due to various internal delays.

HDD delays, as for the HDD given on Slide 27, with the head starting at block 30:

- Seek Time: T_{seek}
  The time it takes to position the disk arm, e.g. to access block 6, which is two tracks away.
- Rotational Delay: T_{rotation}
  The time it takes until the desired block has been rotated under the disk head. E.g accessing block 24 takes half the time of a single disk rotation.
- Transfer Time: T_{transfer}
  The time it takes to actually read or write a given block.

I/O Time and Rate

Two important measures for HDDs are I/O time and I/O rate:

Definition

We define I/O time, $T_{\text{I/O}}$ as follows:

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

Furthermore, the I/O rate, $R_{\text{I/O}}$, which gives the amount of data *transferred per time* unit (typically seconds) is defined as follows:

$$R_{\text{I/O}} = \frac{Size_{\text{transfer}}}{T_{\text{I/O}}}$$

Where $Size_{\text{transfer}}$ is the amount of data transferred.

Examples: I/O Time and Rate I

	Cheetah	Barracuda
Capacity	300 GiB	1 TiB
RPM	15'000	7'200
Average Seek	4 ms	9 ms
Max Transfer	125 MiB/s	105 MiB/s
Platters	4	4
Cache	16 MiB	16/32 MiB
Interface	SCSI	SATA

⚠ Attention: HDD vendors typically use *decimal units* (MB $= 1000^2$, GB $= 1000^3$, etc.), while technically *binary units* (MiB $= 1024^2$, GiB $= 1024^3$, etc.) would be correct! Due to this, HDD sizes sold are larger than what can effectively be used...

Examples: I/O Time and Rate II

Assuming a **random workload** for the *Cheetah* with 4 KiB reads to random locations:

- $T_{\text{seek}} = 4$ ms – taken from the datasheet (average!)
- $T_{\text{rotation}} = 2$ ms – assuming *half a rotation* in average
  (15000 RPM $= 250$ RPS $= \frac{1}{250}$ sec per rotation)
- $T_{\text{transfer}} \approx 31\mu s$ (microseconds!)
  (4096 bytes / 125 MiB per sec)

Thus, $T_{\text{I/O}} \approx 6$ ms and $R_{\text{I/O}} \approx 0.65$ MiB/s (4096 bytes / 6 ms)

For the *Barracuda*:
$T_{\text{I/O}} \approx 13$ ms and $R_{\text{I/O}} \approx 0.30$ MiB/s (4096 bytes / 13 ms)

Examples: I/O Time and Rate III

Assuming a **sequential workload** of 100 MiB, with only a single seek and rotation:

$T_{\text{I/O}} = 806$ ms (*Cheetah*) and $T_{\text{I/O}} \approx 965$ ms (*Barracuda*).

Note that $R_{\text{I/O}}$ almost equals maximum transfer rate!

Summary:

	Cheetah	Barracuda
$R_{I/O}$ Random	0.65 MiB/s	0.30 MiB/s
$R_{I/O}$ Sequential	124.07 MiB/s	103.63 MiB/s

👉 Sequential transfers are $\sim$ 190 / 345 times faster!

Caching

Modern HDDs all contain an on-disk cache , which holds data read from or to be written to disk. This provides more flexibility for the on-disk controller chip, e.g.:

- To read more than requested (e.g. a full track instead of a single block).
- To optimize writing blocks located on different tracks.

When writing to disk, there are two options:

1. **Write back caching (immediate reporting)**
   Acknowledge a write as soon as the data is in the cache. Faster, but can be *dangerous if a specific write order is required* by the OS.
2. **Write through caching**
   Only acknowledge when the data has been written to disk.

Track Skew

Track skew , which is used by many disks, ensures that no unnecessary rotational delays occur during *sequential accesses over multiple tracks.*

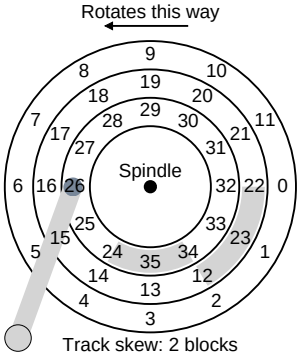

Figure: Track Skew

Courtesy of [ADAD18]

Disk Scheduling

As I/O is costly in general, and there is optimization potential (i.e. reducing seek time / rotational delay), an OS typically performs some sort of disk scheduling .

In the past, there was an advantage compared to process scheduling: When disk properties were known, the length of a job could be computed and SJF (shortest job first) was possible.

Today, modern HDDs do not expose their internals to the OS (e.g. due to caching), and the OS has less influence. It will typically schedule a small amount of disk requests at a time and try to guess the best case; the final scheduling then occurs in the disk.

The OS also performs I/O merging by keeping requests to close blocks together.

Shortest Seek Time First (SSTF)

A straight-forward approach is Shortest-Seek-(Time)-First (SSF or SSTF):[5]

1. Order disk requests by track.
2. Pick requests on the nearest track first.

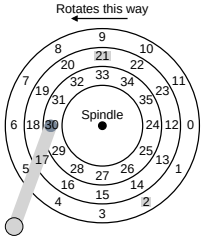

Figure: SSTF Example: Schedule 21 and 2

Courtesy of [ADAD18]

[5] Simpler first-come, first-serve (FCFS/FIFO) scheduling performs badly as it leads to a lot of seeking.

SSTF Issues

SSTF has two main issues:

1. The OS knows nothing about disk geometry (it only sees an array of blocks). Solution: Use Nearest-Block-First (NBF) scheduling.
2. *It is prone to starvation!*

Example of starvation for Slide 36:

1. Blocks 21 (middle track) and 2 (outer track) are scheduled.
2. A long stream of requests for blocks on the inner track occurs (e.g. 24-30).
3. During those requests, blocks 21 and 2 will not be served!

Elevator Algorithms (SCAN, C-SCAN, F-Scan)

To solve starvation, so called elevator algorithms have been developed.[6] They simply "scan" from one end to the other in some form by moving the disk head:

SCAN Moves to one side of the disk, then changes direction and moves to the other side.

F-SCAN Same as SCAN, but *freezes the queue* of requests during a sweep. This prevents starvation of far-away requests.

C-SCAN Only sweep in *one direction* (e.g. from outside to inside). This provides more uniform wait times.

[6] Think of the disk head as elevator and the requests as floors to service…

Shortest Positioning First (SPTF)

Neither elevator algorithms nor SSTF are real SJF algorithms: they do not consider rotational delay!

Assuming that seek- and rotation times are *roughly equal*, it might make sense to *service blocks on tracks further away first!*
This is what Shortest-Positioning-Time-First (SPTF) does.[7]

However, as this requires detailled knowledge about drive state (e.g. current position of the disk head), it can in general only be implemented in the disk controller itself.

[7] Aka Shortest-Access-Time-First (SATF) .

Seek Time vs. Rotational Delay

Which request to schedule first? It depends on seek time vs. rotation delay:

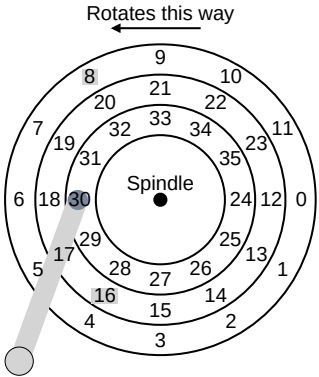

Figure: SPFT Example: Schedule 8 or 16 first?

Courtesy of [ADAD18]

Scheduling Example

Comparison of FCFS and SCAN scheduling for queue 29, 67, 34, 85, 20.

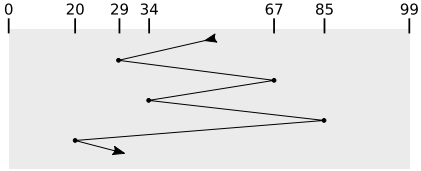

Figure: FCFS and SCAN Scheduling

Own work, based on [SGG19]

Linux Disk Scheduling

The Linux kernel supports different disk schedulers, the most important ones are:

Deadline Has separate queues for read and write, prioritizing reads. Essentially uses C-SCAN and tries to further reduce starvation.

CFQ Maintains queues with different priorities: real time, best effort and idle. Tries to minimize seek time using historical data.

NOOP Basic FIFO scheduler. Used in the past for NVM storage devices (e.g. SSDs).[8]

General information can be found at [linb], individual schedulers are documented in [lina].

[8] Kernels $\geq$ v3.13 use the `blk-mq` framework, bypassing traditional disk schedulers ([blk]).

Appendix

Bibliography I

[ADAD18] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, 1.00 ed., Arpaci-Dusseau Books, August 2018, Available online: `http://ostep.org`.

[blk] *Linux multi-queue block IO queueing mechanism (blk-mq)*, `https://www.thomas-krenn.com/en/wiki/Linux_Multi-Queue_Block_IO_Queueing_Mechanism_(blk-mq)`.

[com] *1956: First commercial hard disk drive shipped*, `https://www.computerhistory.org/storageengine/first-commercial-hard-disk-drive-shipped/`.

[lina] *Linux kernel, block documentation*, `https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/Documentation/block`.

[linb] *Linux kernel, switching-sched.txt*, `https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/Documentation/block/switching-sched.rst`.

[SGG19] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne, *Operating system concepts*, 10th ed., Wiley Publishing, 2019.

[TB14] Andrew S. Tanenbaum and Herbert Bos, *Modern operating systems*, 4th ed., Prentice Hall Press, USA, 2014.