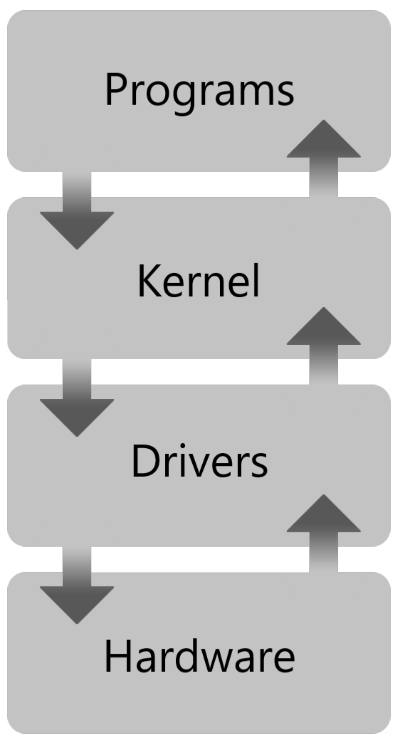
Operating Systems
Structure and Implementation

Daniel Huber, Nick Gilgen and Moray Yesilgüller

November 23, 2021

with supervision from Dr. G. Palfinger and Dr. C. Bersier



Maturaarbeit Kantonsschule Baden

Contents

1 Workprocess

1.1 Workflow

Our first step was deciding what our project should look like. We knew that we wanted to do something related to computer science, but we were still uncertain of the details. Rather spontaneously did we decide to make an operating system. The exact content of our matura project was decided after we talked to several different teachers. We appreciated their oppinions and we were warned of the adversities we would have to face if we chose to go through with it. The first challenge was to find find a teacher willing to oversee our project because we decided to write the operating system in 16-bit assembly. None of the teachers were familiar enough with assembly language, since it is an old and nowadays rarely used programming language. Nonetheless, we embraced the challenge after we talked to Dr. Günther Palfinger. He was kind enough to accept our request and oversee our project. After a few weeks Dr. Palfinger reviewed the contract we set up where we decided on the terms and conditions. The second challenge was getting to know the assembly language. We started the research on the programing language and computer science in general even before we finished the contract. It was rather tough in the beginning because assembly is not comparable to high-level languages like python which we picked up in school. Assembly language is much more intricate considering that it is closer to machine language which is the language understood by a computer. We had to invest approximately two months into research to comprehend the basics of the language. Afterwards we each chose a task and started researching these specific topics individually. We listed the websites we used under the chapter "Sources".

1.2 Tools

Before we began programming we installed a virtual machine that allowed us to run Linux even though our host operating systems were either Windows or macOS. By using a virtual machine we protected our hardware from our own mistakes. The command line is a powerful tool that can, if used incorrectly, break your software and even hardware. But thanks to the virual machine we were able to isolate our host operating system from our working environment which was especially important as we were new to using the command line. Furthermore Linux is commonly used by specialists meaning that there are a lot of guides and tutorials to help beginners. On top of that there are more and usually better, convenient, free and open-source applications. We then installed the Netwide Assembler or NASM for short which, as the name suggest, is an assembler. It translates the assembly source code into machine language so that the hardware can read and execute the program. Additionally we installed QEMU, a software that allowed us to emulate an entire computer system. QEMU enabled us to test our programs quickly and efficiently. To write the chapters and code we used neo vim which is a text editor focused on extensibility and usability. It can be an extremly powerful text editor if the user is experienced. While neo vim takes some getting used to it also provides the user with convenient shortcuts. Sometimes our code was faulty and we needed to debug it. For convience's sake we used an online debugger called GDB. This website provided us with quick and easy access to a reliable debugger. In order to save time and for simplicity and consitency we also used the make utility. We created a Makefile that contains instructions for building, emulating and debuging software. The command make executes the Makefile in the current directory and is followed by the name of the target(s). In our project the Makefile launches NASM, translating the source code, and then QEMU, emulating the operating system according to the parameters defined in the Makefile. We shared the Makefiles and everything else we made with git. The aforementioned is a distributed version control system that allowed us to manage our project quickly and efficiently. With git one can upload files to and download files from a repository. We used GitHub to store our files on a cloud so that we were able to work together on this project. Finally, we used Pandoc, a universal document converter to convert our written text for the matura from LaTeXinto a pdf. All of the previously mentioned programs are free and open-source.

1.3 Workflow

As soon as we installed the necessary software and got a grasp of the basics of assembly language we started programming. We chose easier tasks in the beginning so that we could warm up to this rather complicated language. Whenever we were done writing a program we began debugging it. Some bugs cost us a great amount of time which was often very frustrating as we lost up to three weeks because of a single bug. Fortunately, we never lost hope and always continued our research. A few times we had to rewrite our code because we were either stuck or unhappy with the structure. This helped a lot because we had already gained new insights by the time we started rewriting the program. After a while the tasks we worked on got more complicated and the research had to be more extensive. When we were not in the mood to work on the programs or do research we wrote chapters explaining the most important parts of an operating system. Besides that, we contributed to the work journal every week to keep Dr. Palfinger up to date. In the final stages of our matura project we finished writing the chapters and debugging our programs.

2 Booting

Whenever a computer starts up, there is a fixed set of instructions that have to be performed to initialize the system correctly. This routine is called a system startup. In IBM-PC compatible systems, the system startup is done by the Basic Input Output System (BIOS). The BIOS is firmware that is stored on the motherboard and is executed automatically. On newer systems there is more powerful but also more complex firmware available, namely the UEFI BIOS. We will cover the differences between the two later but they both handle the system initialisation. One of the firmware's main tasks is doing the Power-On Self-Test (POST), which checks the presence and the integrity of hardware components such as the processor, RAM, keyboard, graphic processings unit (GPU) as well as storage devices. If system critical components such as RAM or the processor are not present the PC speaker makes a beeping sound and turn off. After a successful POST the BIOS gathers some details about the hardware and write some basic information on the hardware into the Bios Data Area (BDA). Because the BIOS prepares the machine for an operating system (OS) it presents the hardware with details in a format that can be read by the OS. The BDA is a data structure that is located in memory at location 0x400-0x4FF and can be read as soon as the operating system has been loaded. Slightly newer BIOSes also write to the Extended BDA, where more modern hardware information can be found. Afterwards, the BIOS configures hardware such as the system clock to keep in sync with the time and sets up interrupt handlers (more information in chapter Interrupts). Finally, the BIOS searches all the attached storage devices for a bootloader sector. The search order can be set in the BIOS's settings. Identifying a bootloader sector of a storage device is done by comparing the 511. and 512. byte of the first sector of the storage medium with the values 0x55 and 0xAA respectively. These values were initially arbitrarily chosen by the original equipment manufacterers (OEM) as the identifying number for bootsectors. When the BIOS finds a valid bootloader sector it loads the entire sector into RAM at the fixed location 0x7C00. Yet again, this value was chosen randomly by the OEMs back in the 1970s but to this day IBM-PC compatible systems load the bootsector into that location. After a successfull load, the BIOS transfers control to the bootloader by telling the CPU to fetch instructions from location 0x7C00 [1].

[1] Bootprocedure, from: Oracle docshttps://docs.oracle.com/cd/E19683-01/817-3814/6mjcp0qjq/index.html, retrieved November 23, 2021

2.1 Bootloader

The bootloader has two important tasks. The first one is gathering more information about the RAM of the system. It determines whether there is enough RAM to load the OS and then if there is it loads the OS. This means that it accesses the storage devices and reads a number of sectors into memory using a BIOS interrupt. The number of sectors depends on the size of the OS on the storage device. A BIOS interrupt can be thought of as a function provided by the BIOS that the bootloader can call. There are multiple interrupts but only one of them can be used to read sectors into RAM. This interrupt has the number number 0x13 in hex or 19 in decimal and it is invoked with parameters that load x many sector on y many platters into the RAM address determined by a parameter. After a successful load the bootloader hands over the execution to the operating system, specifically the *kernel*. [2]

2.2 Kernel

The kernel is the most important part of the OS. It controls the hardware and it manages all *system resources*. System resources are both digital or physical components of a computer that are shared between multiple users and all their running programs. Examples of such system resources are:

- Processes; programs are started by other programs and users but they are managed by the kernel.
- CPU time; There might be a hundred processes running but there aren't a hundred processors available. The kernel will offer every process a *time slice*during which a process will be running on the CPU (core).
- Memory; We don't want a process to interfere with other processes' memory. It is a kernel's job to manage memory to avoid inter process interference. Memory that is managed by a kernel is called *Virtual Memory* (VM).
- Internet access; Taking care of internet packets is also one of the kernel's jobs.
- Direct hardware access; Direct hardware access is done by the kernel through device drivers. A program may ask the kernel for disk access and the kernel will either grant or refuse it. If the kernel grants access it will tell the driver to perform the hardware specific operation requested by the program.

There are many more, some of which will be explained later in this booklet. But one can clearly see that a kernel must take care of a lot of stuff. It is because of that reason that the kernel is the most important part of an operating system; as it ensures everything is working (more or less) correctly. It is important to point out that the kernel on its own is useless without other components of an operating system because there aren't any system resources to manage if nothing is being requested. The other components of an operating system are programs that create an environment that provides the user with utility and comfortability. Some examples include Graphical user interfaces (GUIs) or text user interfaces (TUIs) and a *shell*. A shell is a program that reads text-only commands and performs the requested commands. It is called *shell* because it is the outer layer of an operating system. Users interact with said outer layer and there are many different implementations of shells. Summarized, an operating system is made out of a kernel and many programs working on top of the kernel that provide an user experience. [3]
hard disk and a file. The Kernel will ensure that the hard disk can be read from and written to by programs just like it will also manage the reading and writing of a file. There are hundreds of different hard disk drives and they all work a bit differently. If every program had its own

[2]Booting in Operating system, from: Javapoint, November 23, 2021 https://www.javatpoint.com/booting-in-operating-system

[3]Bellevue Linux Users Group: Kernel Definition, November 23, 2021 http://www.linfo.org/kernel.html

2.3 UEFI

3 The CPU

The Processor (Central Processing Unit) is often refered to as the brain of a computer. On the lowest level, it is made of circuits with thousands or up to billions of transistors. In essence, a CPU reads and writes data to memory and performs operations on it based on the instructions it receives. There are many different architectures and processor families but this chapter will explain the most basic inner workings of a processor with a hypothetical model. We will upgrade our CPU on the run to introduce new features and instructions, but start off with a simple CPU that has 4 components:

- A Programm Counter (PC)
- Accumulator Register
- An Arithmetic Logic Unit (ALU)
- Random Access Memory (RAM)

The PC is a register that holds the address of the next instruction located in RAM. First, it is important to understand what a register is. A register is a circuit that can hold a binary value. The size of register is given in bits or in bytes. Our registers are 8 bits wide, meaning they can hold any value between 0 and 255. Registers can be read and written to and will retain their value until they are overwritten again. They are located on the CPU and most CPU operations are done *on* data in regsiters.
Secondly, it is important to understand that RAM can be addressed by means of *numeric values*, these values are also called a *memory address* and when the address is stored in a register, it is called a *pointer*. So the PC is a pointer to the next instruction in memory.
Now, the next question arises, what exactly is random access memory? RAM is made of many fields that can store one byte of data. They are similar to registers, in the sense that they hold data, a numeric value between 0 and 255, until they are overwritten again. In RAM, there is a large number of those fields while there may only be a few registers present. RAM takes the form of its own separate hardware, namely RAM sticks. RAM is used by the CPU while it is running, but once the computer is turned off, all the data present in RAM and in registers is deleted. One key feature of RAM is that it truly is random access, meaning there is no particular order in which data has to be stored or retrieved. Each one of those fields has an address and these fields can contain normal data or also instructions for the CPU itself.
The accumulator register is just a register that the CPU uses for temporary storage, before it is either stored in memory or overwritten. Many operations also just work on registers, if we for example add two numbers, one number has to be stored in a register but the other value can be stored in memory. In our hypothetical CPU there is only one register, namely the accumulator.
The next component is the ALU which is a complex circuit that performs binary operations. These operations range from arithmetic operations to bitwise operations of. All these operations are circuits etched in the ALU and a instruction that uses the ALU will execute as follow:
- Operands are loaded into the ALUs own registers. - Instruction triggers the correct arithmetic circuits to perform the operation on the registers in the ALU. - Result is then stored into the destination, our CPU defaults to storing it in the accumulator.
The ALU can be very complicated, depending on which arithmetic operations it supports. Common ones are addition, subtraction, division, multiplication as well as binary operations like xoring, anding and bitshifts. Our CPU will receive an ALU upgrade a bit later in this chapter.
Now that we have a basic understanding of the component, let's look at a possible instruction set (= All instructions a processor can understand and execute):

Instruction name	Argument	Encoding
LOAD	<NUMBER>	0x0F $<XX>$
LOADADDR	$<ADDRESS>$	0x0E $<XX>$
ADD	$<NUMBER>$	0x7C $<XX>$
STORE	$<ADDRESS>$	0xE1 $<XX>$

These instructions seems rather weird, haven't we all heard from a young age that computers work with 1s and 0s? That is obviously true, but a binary instruction can also be represented in a human readable form. This human readable form of the most basic CPU instructions is called *assembly language*. The table also features a column named encoding. Encoding refers the numeric value that the instruction has and is represented here as a hexadecimal number. In memory, the instruction is stored in its binary representation.
Lets look at a instruction located in memory. It is encodes it as follows: '0b00001111 0b00000101' which in hex is 0x0F 0x05 and in human readable consists of the mnemonic 'LOAD' and the operand '5'.
Forward on, we will only use the hex representation and the human readable representation because the binary one is cumbersome. Most instruction encodings are also numbers that were chosen pretty randomly with the only requirement being that the encodings don't collide. If both 'LOAD' and 'STORE' were encoded as '0F' the CPU would not know which operation to perform.
So in our case,
x0F is the instruction that tells the CPU to 'LOAD' a number in the accumulator. and the x05 is the number we want it to hold. When our CPU has finished executing the first instruction the accumulator holds the right value and the PC is incremented automatically to point to the next instruction, which is then fetched from memory. Here, the next instruction is the following:

```
\x7C \x03 or in human readable form: ADD 3
```

The $0x7C$ instruction triggers the ALU to perform an addition with the accumulator and the value that followed the $0x7C$ instruction, namely the $0x03$, which means that the value in the accumulator will be updated to 8. The PC is incremented again and the next instruction is fetched from memory: $0xE1$ $0x00$ which disassembles into 'STORE 0'. This instruction stores the 8 that is in the accumulator into the memory location $0x00$. This might be surprising, but 0 is a valid memory address. That small field in RAM now holds the data which is saved for later.
So far we have familiarized ourselves with a cycle called the *fetch, decode, execute cycle*. It describes the steps taken for an instruction to be run by the CPU. Fetching is done by retrieving the instruction from RAM at address held in the PC. Engineers have realized that having instructions in memory that must be repeated take up a lot of memory and memory was very limited in the early days. [4]

3.1 Memory

Memory refers to a system or device that is able to store data for immediate use. Compared to permanent storage memory offers a faster access to data at the cost of very limited storage capacity. At the beginning of computer science memory storage was very ineffective. Thousands of small vacuum tubes were needed for simple decimal calculations. There are several different memory storage mediums and memory types, each with their own benefits and drawbacks. The use of memory is determined by the purpose of the data in memory. There are three different segments of a computers memory. The fastest segment is cache memory, followed by primary memory and lastly secondary memory. While secondary memory is the slowest segment it is also usually the one with the highest memory size. Peripheral storage devices such as hard disks, cds, dvds and floppy disks are part of secondary memory. The data in secondary memory is only accessible through I/O ports

[4] CPU, from: Learncomputerscienceonline, November 23, 2021
https://www.learncomputerscienceonline.com/what-are-cpu-registers/

making it slower than the other segments. Primary memory refers to the memory that can be accessed by the CPU. Main memory, cache memory and CPU registers are all part of primary memory. However, the fastest types of memory are also the most expensive types and the ones with the smallest data capacity. Memory nowadays is implemented as semiconductor memory. The data is stored in memory cells, where each can hold one bit of data. Semiconductor memory is seperated into two types of memory:

3.1.1 Volatile memory

Volatile memory refers to memory that requires power to store data. The data stored in volatile memory devices is either lost or stored somewhere else when the computer shuts down. Examples for volatile memory are DRAM (dynamic random-access memory) and SRAM (static random-access memory). Both have their advantages. DRAM uses only one transistor per bit which means that it is cheaper and takes up less space on the RAM sticks but is more difficult to control and needs to be regularly refreshed to keep the data stored. SRAM on the other hand does not lose the data as long as it is powered and is simpler for interfacing and control but uses six transistors per bit. Using only SRAM would be much more expensive and unnecessary for certain tasks, where the hardware cannot send a response within nanoseconds. DRAM is mostly used for desktop system memory. In contrast, SRAM is used for cache memory. The cache is seperated into two to three levels. L1 is the first level of cache memory and is located on the cores of the CPU and not the RAM like DRAM. The size of a level 1 cache can range between only 2 KB and 64 KB. Processor calculations can be as fast as nanoseconds, which is why the data in memory needs to be accessible in such a short time. L2 is the second level of cache memory and it is present inside or outside of the CPU. The size of memory ranges from 256 KB to 512 KB. Level 2 is not as fast as level 1 but is still faster than primary memory thanks to a high-speed bus that connects the cache to one or two cores of the CPU. L3 is the third level cache which is always outside of the processor. All cores share access to level 3 caches which enhances performance of level 1 and level 2 cache. The size of memory is between 1 MB and 8 MB.

3.1.2 Non-volatile memory

Non-volatile memory can retain the stored data without a supply of power. ROM (read-only memory) is a well-known example of non-volatile memory storage, as well as peripheral storage devices such as hard disks, floppy disks, cds and dvds. Non-volatile memory usually handles secondary storage and long-term storage. This type of memory is once again divided into two categories:
-Electrically addressed systems such as ROM, which are generally fast but are expensive and have a limited capacity.
-Mechanically addressed systems are cheaper per bit but are slower. tubes were needed for simple decimal calculations. [5]

3.2 Jumps and subroutines

Let's assume that a program should run forever. Our primitive CPU lacks such a feature because it would require an infinite amount of RAM and the PC is a *register* that can only address 255 instructions. The PC also points to the *next instruction*. What if our CPU allowed operations on the PC register just like it does with the accumulator? Depicted below is an example of a dissasembly. The numbers on the left side display the address of the memory that is being disassembled and on the right is the instruction mnemonic. "‘ 0x0 -¿ LOAD 0 0x2 -¿ ADD 5 0x4 -¿ JMP 2 "‘ The first instruction should be familiar to most readers. Shortly after starting the program

[5] Memory, from: Javapoint, November 23, 2021
https://www.javatpoint.com/volatile-memory

the ‘LOAD‘ instruction is executed and the PC incrememted. The second instruction should also be mundane. The CPU executes the instruction and again increments the PC by 2. There is a new instruction in our instruction set. ‘JMP‘ is similar to ‘LOAD‘ but instead of moving a value to the accumulator it moves it to the PC. Because PC holds the value 2, the proccessor will fetch the next instruction from address 2. The CPU will execute the instructiom ‘ADD 2‘ and prepare itself for the next instruction by fetching it. This would be the instruction at address 4. The processor would run the instruction and end up executing the 2 instructions (‘ADD 5‘ and ‘JMP 2‘) over and over again. This short program will run forever, adding 5 to the accumulator until it *overflows*. This means that the result that should be present in a register is altered because the result can not be represented in the register size. In practice this means that a 8-bit register can hold maximum value of 255. Whenever a mathematical opertion results in a number greater than 255, such as 0b11111111 + 1 then the value will be 0 inside the register instead of 256, which requires more than 8 bits.

3.3 Conditional branching

If we want a program to run forever, jumps are more than enough. Readers familiar with programming languages should know about ‘if‘ and ‘else‘ statements. They are used in programming to express under what condition a certain code block should be run. They are translated to CPU instructions in the following format:

C	Assembly

if (<condition>) {
        // first code block
}
else {
        // second code block
}
// more instructions...
	cmp <value>
	jne elseblock
	        // first code block
	jmp done
	elseblock:
	        // second code block
	done:
	// more instructions...

The relevant instructions are ‘cmp‘ and ‘jne‘. A processor that supports conditional branching needs another register, specificly a *status register*. A status register contains bits that each signify a single condition. Our status reguster is 2 bits wide and those bits are the *Carry Flag* and the *Zero Flag*. These flags are extremely crucial to conditional branching. The ‘if‘ statement in the programming language is followed by a *condition*, such as a *comparison*. A comparison can be $xsmallerthan3$ or $yequalto0$. They result in a either true or false value. In CPU language, the ‘cmp‘ instruction performs a subtraction between two values, in our case between the accumulator and the value 7. The result of the subtraction is not stored in the accumulator but in the status register. The result is not the numeric value that results from the subtraction but rather the status. If the compared values are equal, the subtraction results in in zero and the Zero Flag bit in the status register will be set to 1. If the first compared value is larger than the second one, the subtraction would result in a positive value. A positive value sets the Carry Flag to 1 and the Zero Flag 0. If the first value is smaller than the second value, the subtraction results in a negative value and the Carry Flag and the Zero Flag are set to 0 . [6]

3.4 Negative numbers and two’s complement

Undrstanding how computers represent positive integers is not very abstracted to the way humans represent numbers, with the exception of a representation in *base 2* instead of *base 10*. Humans denote negative numbers with a minus in front of the numeric character resulting in such a notation:

[6]Jumps, from: Infoscinstitute, November 23, 2021
https://resources.infosecinstitute.com/topic/conditionals-and-jump-instructions/

-6 or -3. One might think that combining the *absolute value* of a number together with a *sign-bit* should suffice for a CPU express negative numbers. Integers -9 and 9 would look like 0b10001001 and 0b10001001 respectively. It is certainly a valid option and engineers have produced CPUs that internally use this type of binary representation for integers. However, this is not the internal integer representation used in processors today. There is a weird issue that can occur after a mathematical operations resulting in zero. Instead of there being one zero there are actually two zeros now; 0 and -0. This is why modern processors use a different notation. Instead of only using a single sign-bit, all the bits are flipped and act as "extended" sign bits and then 1 is subtracted from the number. This means that 0b11111111 is -1 and 0b10000000 is -128. This also voids the issue of multiple zeroes and means that a 8 bit register can contain the numbers between -128 and 127. A programer can also declare that he wishes to use an *unsigned value*. The compiler will generate machine code that treats the value as purely positive, no matter the sign bits. [7]

3.5 Modern processors

So far our CPU is capable of basic arithmetic operations, conditional branching and working with both signed and unsigned data i.e. integers. This is more or less all what CPUs had to do to get work done in the early days of computing. With the creation of *integrated circuits*, CPUs were mass produced and became a lot cheaper. This meant that a computer system hwas made of a CPU and some other, smaller and cheaper co-processors. To this day these co-processors come soldered on the motherboard and relieve the CPU (note the **Central** *Processing Unit*) from some work. Before we follow up un *what* the CPU and co-processors do together we must understand *how* they work together. These processors must be able to communicate with each other. This happens either through *memory mapped I/O* or through **I/O pins**. Every processor needs RAM to get its instructions and store and retrieve data from. In some systems, multiple processors share the same RAM and these processors comunicate with each other over certain memory regions. The processors would read and write to those memory regions and communicate with each other, but it is important that all processors know what the address of said memory areas is and that there are no *data races* that cause memory corruption because multiple processors read or write simultaniously to the same memory address. Memory mapped I/O is used in CPU cores. Every core in a CPU can be seen as an individual processor and all these cores share the same RAM i.e. they communicate with each other over memory. Many processors on a motherboard do not necessarily share RAM and memory mapped I/O is not an option. I/O pins are a fantastic way for comminucations between both processors and hardware.For this, a processor must have new instructions, namely *in <pin number>* and *out <pinnumber>*. These two commands *send the data* in the accumulator to the specified pin serially and *read a byte* from the specified pin number. Most readers should have seen a CPU and noticed the hundreds of pins that are sticking out of the CPU. Some of those pins are reserved for $\mathbf{V}_{in}$ and ground but almost all other pins are for **I/O**.

[7] Negative Numbers, from: Binaryhakka, November 23, 2021
https://binaryhakka.blogspot.com/2020/03/assembly-language-negative-numbers.html

4 Interrupts

One of the key features of modern processors is the ability to support *interrupts*. Unlike many other processor features interrupts are not an arithmetic operation but rather the ability of a processor to *respond to a event asynchronously*. Events are unexpected changes that are signaled to the kernel. This chapter will explain in greater detail what they are and how they work in low level.

4.1 The different types of interrupts

Gone are the days where the only instructions CPUs were capable of executing were purely arithmetic. The engineers behind the chip design in the 80' noticed that computers are destined to serve a purpose beyond specific mathematical computations, namely consumer applications.
In practice this means that a processor may be running a program until an event, such as a pressed key, *interrupts* the execution of the program and invokes a handler for the event. The handler runs and upon completion the processor resumes the execution of the program.

4.2 Hardware

Most hardware needs to communicate with each other. It is used for *time synchronisation*, *data* and to send *commands*. Devices that are attached to the computer, such as keyboards and USB thumb drives as well as coprocessors and hardware located directly on the motherboard such as sound cards and the clock, need to be synced, programmed and set up data exchange. Most of this communication is done via *hardware interrupts*. All these attached devices are connected to the CPU with a wire called the *interrupt line*. When a device needs the CPU's attention it toggles the voltage of the interrupt line. This signals an *interrupt request* to the processor. The CPU then finishes executing the current instruction and notice the interrupt request and check which device requested the interrupt. Modern CPU architectures have multiple *interrupt lines* and each of them can be assigned to a handler. In action this would look like the following:
running cpu -¿ interrupt -¿ run handler -¿ resume execution
The handler then reprograms the device if needed or performs the data exchange. Hardware interrupts allow te processor to do its job and only attend hardware if necessary. This must be supported by the hardware (interrupt line).

4.2.1 Interrupts vs Polling

There are two paradigms used to await input, the aforementioned interrupts and *polling*, which is a continous check of hardware status. It is usually implemented in software and used in systems that do not support interrupts. This means that a running program must frequently check all the attached hardware. The rate of which hardware is checked is called the *polling rate*. A high polling rate means that the program must spend a lot of time polling and a low latency. This is useful in systems, such as routers, that do not perform computationally intensive calculations but need high responsiveness. Most of the time these devices *do nothing* until an event occurs. In the case of routers this is an arriving internet packet. The device only then stops polling, handle the event i.e. determine the packet destination and then reroute the packet accordingly. It then resumes polling once again. In the time slice between receiving the packet and rerouting it, the CPU was busy and was not able to check for another packet.

4.3 Software

4.4 Interrupt vector table

Interrupt handlers are essentially *functions* that are automatically executed as soon as an interrupt occurs. They differ from normal functions because they are not explicitly called. Processors must

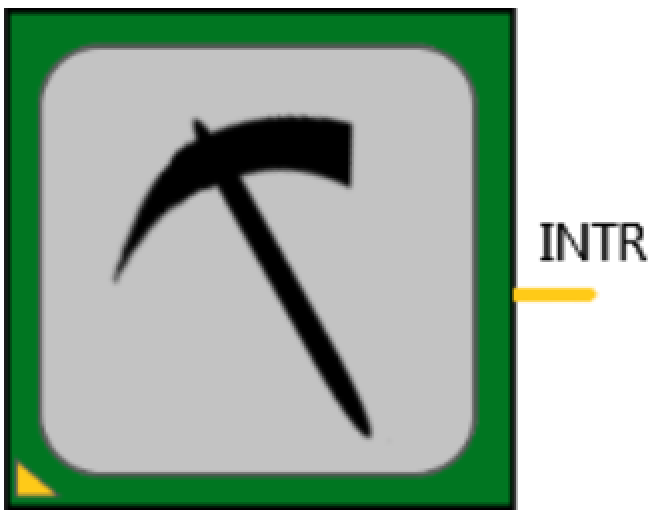


Figure 1: Processor running a regular program

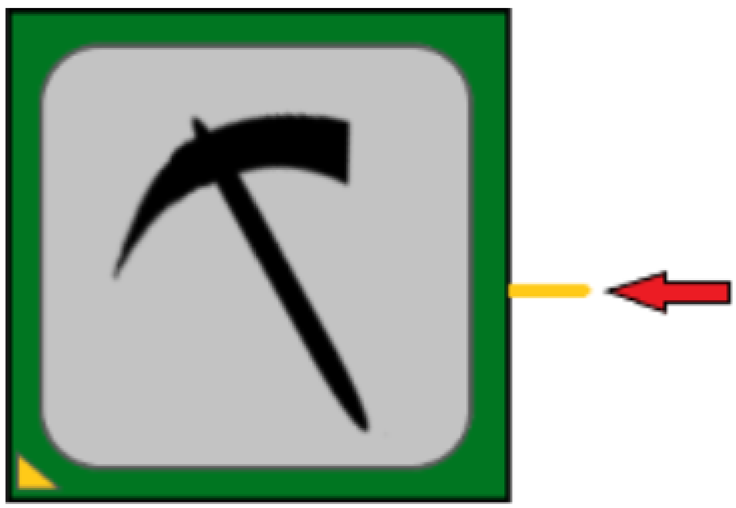


Figure 2: Interrupt line(INTR)

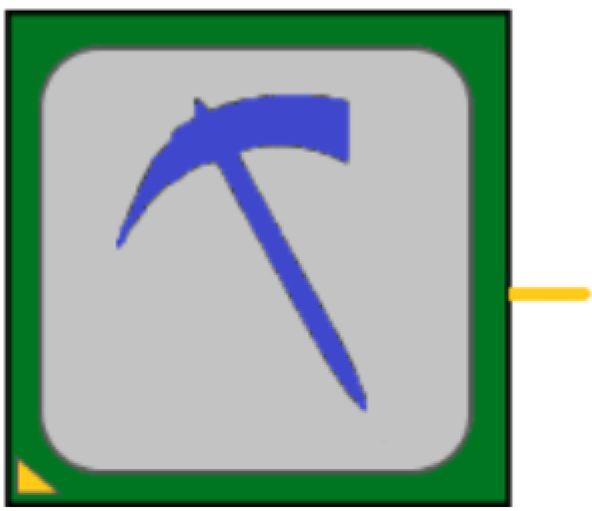


Figure 3:

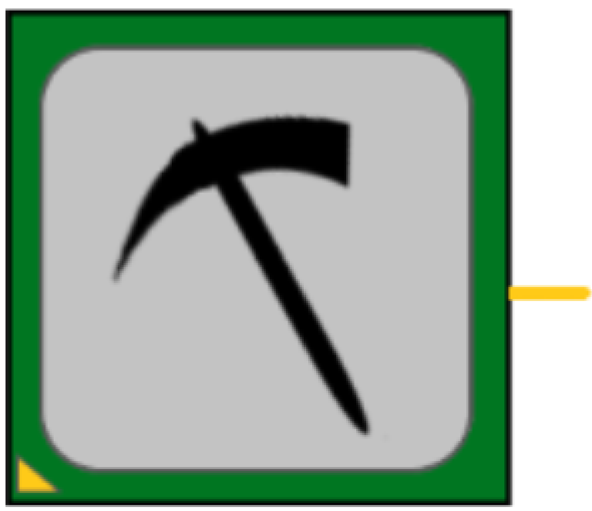


Figure 4:

know the *address of a function* in order to call them. Interrupt vector tables (IVTs) are a method to associate interrupts to interrupt handlers. It is essentially an *array of function pointers*. Interrupts are generally numbered and these numbers are used to look up the address of the interrupt handlers. For example, software interrupts are invoked with the 'int <NUMBER>' instruction. This number is the *interrupt number* and it is used by the CPU to determine the address of the handler. The interrupt number is used as an array index. The instruction 'int N' makes the CPU jump to the address pointed to by the IVT's Nth entry. Some architectures have the IVT at a fixed location. The earlier versions of x86 processors had the IVT at address 0x0. Every entry is four bytes long, meaning the address of the 'int N' handler is at adress '4 N'. As an exmple, the 'int 0x10' instruction makes the CPU jump to the address that is in the IVT's 0x10's address, i.e. the address 0x40. [8]

4.5 Interrupt requests lines

Interrupts are also invoked by peripheral devices and they often do not necessarily have a fixed interrupt number associated with themselves. In systems with multiple peripheral devices such as for example a keyboard and a network adapter it is more practical to have separate interrupt handlers instead of a single handler. Mapping each peripheral device to its own handler avoids the overhead of having to identify which device requested the interrupt. This mapping is done with a coprocessor often called the *Programmable Interrupt Controller*, PIC for short. An operating system's task is to *reprogram* the PIC by assigning every peripheral device to its own handler. Reprogramming the PIC also allows for *prioritized IRQs* i.e. determining which IRQ should be handled first when multiple IRQs occur simultaneously. Whenever a new hardware interrupt occurs, the reprogrammed PIC determines the correct handler, communicate the IRQ number to the processor which invokes said handler. In modern operating systems with dedicated drivers, the IRQ handler invokes the driver that takes care of the device reconfiguration or data transfer.

[8] Interrupts, from: Tutorialspoint, November 23, 2021
https://www.tutorialspoint.com/embedded_systems/es_interrupts.htm

File type	Signature
JPEG	ÿØÿÛ
MP3	ÿû
ZIP	PK\x03\x04
PDF	%PDF-

Table 1: A list of common file formats and their signatures
[9]

5 Files

One of the most convenient and widely used features provided by operating systems are files. Files in the traditional sense are persistent sequences of bytes associated with a *file type*. A traditional file is stored on a storage medium such as a hard disk and is *remembered* by the computer unless it is instructed to delete it. Traditional files can contain an arbitrary amount of data only limited by disk space and filesystem support.

5.1 File signatures

Magic numbers are more or less arbitrarily chosen numberic or textual constanst used across most of computer science but in the context of files they are seen as *file signatures*. Every file that has a special format such as JPEG, MP3, ZIP, PDF as well as ASCII-text. There are obviosly many more but these are very common. Most files of the aforementioned format have a specific file signature, a magic number located within the first few bytes of the file. It can be thought of as file extension stored in the file itself and not in the file name (metadata). The only difference is that extensions differ from the file signatures.

Whenever a user opens a file from Finder on MacOS or any similar file manager that does not contain a file extension it will open the file with the correct program nonetheless. One problem arises, namely ASCII has no directly associated file signature with it and script files as well as PDF files have an ASCII file signature. If a user was to write a file that *accidentaly* contained a file signature used for a non-ascii format such as `%PDF-` for PDFs, the file manager will try to open the file in a PDF file viewer. The file will be interpreted incorrectly and seem corrupted. This is why using file extensions is still a good idea, even in operating systems that support extensionless files[10].

5.2 Executables

Executables are files that can be *run*. They are sometimes called *programs* but executables refer strictly to the files, specificly files containing *instructions and data*. Whenever their name is typed into the command prompt, the executable is run and when finished, the user will be returned to the prompt. However, there are multiple types of files. A JPEG image is a excellent container for photographs and pictures but it is a terrible format to store instructions and data for a computer. It is a bad idea to execute a JPEG file because the instructions contained within the file are garbage at best or nefarious (such as malware) at worst. Operating systems have mechanisms to deter users from running non-executable files but the OS itself has to know wether a file is runnable or not. These files are identified in some operating systems by their *extension* or their *file signature*. An extension is a small appendix to the file name. It is of format `filename.extension`. The Windows NT family of operating systems relies heavily on extensions to differentiate between executables and regular files.

[10] File signatures, from: LSoft, retrieved November 23, 2021.

5.3 Pseudofiles

Because files are organized and structured in hierarchical filesystem, some operating systems use special files that are actually *interfaces* to *device drivers*. In practice this means that a special file can refer to a file that when `read` and `write` operations are performed on it, a driver intercepts the regular read and write commands and performs the operations not to a regular file but rather to a device. In unix-like OSes there a myriad of device files located in the `/dev` directory. Hard disks are named in `sd*` where `*` is an alphabetical character in order that the OS detected the device. `/dev/sda` is the first hard disk, `/dev/sdb` is the second and so forth. Accessing hard disks through `/dev/sd*` allows *raw* access to the device instead of the traditional file system interface where files are stored according to the filesystem structure. There are other special files in `/dev`such as `/dev/mem` that allows access to the whole system memory. Additionally there are input devices such as `/dev/input/mouse*` where `*` is the number in which the mouse was detected at startup. Mouse devices are a *character device* which means that operations are performed bytewise. Reading from `/dev/input/mouse0` allows a user or a program to view the bytes that are send from the mouse device. A display server (more in chapter Userinterfaces) reads from this device via the device file interface. The *"files"* located in `/dev` are not only device interfaces, there are also other special files called *pseude device files*. An example of those is `/dev/null`, which is essentially the garbage bin or trash. Writing to `/dev/null`, the data is simply discarded and can not be recovered be reading from it. Reading operations are not implemented for the Null-device. Another common device file in this directory is `/dev/urandom` or in some OSes `/dev/random`. Urandom is a pseudo device file that only supports the `read` operation and returns random data. It is frequently used to seed random number generators or to generate cryptographically secure keys. Writing to Urandom is also not permitted. The kernel regularly adds random data such as mouse input and alike to Urandom and performs cryptographical operations on the data to make it seem more random. Every `read` operation alters the content in Urandom and reading multiple times from it yields different data. This pseudo device interface makes it very practical to get random numbers. Windows also has device files and pseudo device files but they are not necessarily located in a directory, depending on the system configuration. The Windows 10 device file structure can be accessed with help of the *WinObj* tool which implements the \\?\`Device` folder`windev`. WinObj used the WindowsNT native API internally but allows access to device interfaces via file paths. For example, a raw hard disk partition can be accessed by writing to \\.\`Device`\`Harddisk*`\`partition#` with `*` being the number of the disk in which it was detected at startup and `#` referring to the partition number. Many special files in unix-like operating systems also exist in Windows, although named differently. Compatibility layers such as `zerodrv` implement drivers for unix-like equivalent special device files \\?\`Device/zero` and \\?\`Devices`\`null` on Windows[11].

5.3.1 Character devices

5.3.2 Block devices

[11] User1686: /dev/sda in Windows, retrieved November 23, 2021.

6 Filesystems

As mentioned before, one of an operating system's tasks is the management of resources. One of those resources is storage. There are three main reasons why storage (and many other hardware devices) are handled by the OS.

1. The OS provides a layer of abstraction for the running application programs.
2. Talking to the hardware reqires a certain privilege level (IO privileges).
3. Operating systems provide consistency across all files, users and processes.

Remembering file names is easier for humans than remembering the physical location of the file on the storage medium. Filesystems handle the numbers and metadata of the file and the mapping of file names to actual file contents. It controls and organizes how data is stored and retrieved from a storage device. It structures a storage device in files and directories and their metadata. Modern filesystems also provide mechanism for error detection and encryption.

6.1 Storage devices

Storage devices such as hard disks, floppy disks, USB flash drive and SSDs have no notion of files. These devices only know *sectors*. Sectors are the smallest container a storage device can work on and they can usually hold 512, 2048 or 4096 bytes of data. These devices are attached to the computer through I/O (input/output) ports, sometimes in the form of SATA connectors or USB slots, where they can receive commands from or send information to the CPU on the system. A command can be to `READ` sector number x to RAM address y or `WRITE` from RAM address y to sector x. This command is sent through specific I/O pins from the CPU to the storage device. The device then executes the command or returns an error if for example sector x is not a valid sector. Remembering which files are located on which sector numbers is tedious for both humans and computers. This is where filesystems come to help. [12]

6.2 The superblock

The heart of the file system is a datastructure that contains every single file and directory: the superblock. It holds the *metadata* (i.e. data about data) about every single file/directory, including its location on the storage device and additional information such as permissions, creation and modification date but not the contents of the file itself. Said metadata of a file/directory is grouped together in a structure called *inode*. Every inode is of equal size and is associated with an inode number. In essence, the superblock is an array of inodes, where the inode number is the array index. Having files be uniquely identifiable by an inode number instead of a filename allows the use of *hard* and *soft links*. In a nutshell, this means that a file can have multiple filenames. This can be useful in cases where some programs expect a certain path for a different program than the one present on the system. For example a *Makefile* expects the program *cc* (the C compiler) in `/usr/bin/cc`. there are many different C compilers and they have distinctive names. Let us say that the users C compiler is GCC and located at `/bin/gcc/`. A soft or hard link can be created to map the filename in path `/usr/bin/cc` to `/bin/gcc`. If files were only associated by their name and not their inode number, links would not be possible and the OS would have to copy the the entire executable GCC to the new location and store it under a different file name. Links allow the filesystem to effectively safe storage space and that is why modern filesystems use inode numbers to identify a file. In our case the whole space of the superblock is allocated upon disk formatting. This leaves the possibility of 'running out of space' without the storage device being even remotely full. This occures when all

[12] Storage devices, from: Tech21, November 23, 2021
https://www.tech21century.com/different-types-of-storage-devices/

the inodes in the superblock are used up. This happens only when a computer user creates too many files that take up close to no space.
It is important to point out that Windows' NTFS (new technology file system) groups file metadata differently. We are not going to cover NTFS because the official version is proprietary and therefore we have limited information on the internal workings. While most mechanisms in NTFS are similar to unix filesystem mechanisms, they often have a different name. The superblock is called the *Master File Table* and inode numbers are called *FileID*s. [13]

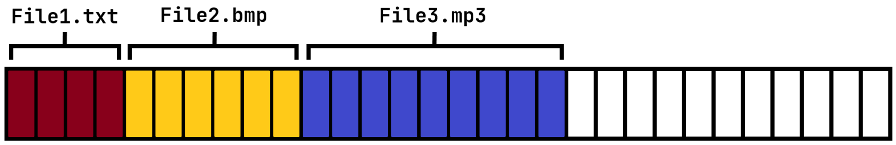
6.3 Sector allocation

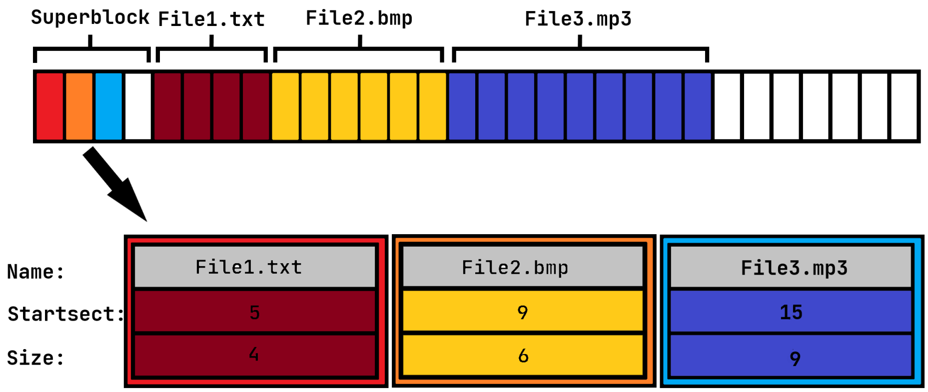
Another important data structure in filesystems is used by the *sector allocator* to keep track of the occupied and unoccupied sectors of the disk. It is important to keep track of which sectors are in use and which are unused. This data structure is usually a *linked list, bitmap* or some form of *tree.* The linked list approach is a common one. Every sector is an item in the list and contains the location of the next free sector *a.k.a* next item in the list. The filesystem must remember the location of the first free sector. But if a sector that is part of the linked list gets corrupted due to *data rot* (the decay of data on storage devices due to radiation or age), the whole rest of the linked list (i.e. free sector list) is lost. Redundancy of the data structure that keeps track of the free sectors is crucial in modern filesystems. Copying a linked list is easier said than done and this is when bitmaps come into play. Bitmaps are easy to copy and mantain and are more reliable than linked lists. They take up more space than linked lists because the filesystem keeps track of every single sector on the disk within the bitmap. The bitmap can be thought of as an array of N bits where N is the number of sectors (field of 512, 2048 or 4096 bytes). Every sector on the disk is enumerated and represented in the bitmap (a.k.a *bit array*) at bit number N. Said bit in the bitmap can be either 1 (free) or 0 (empty). Whenever a new file has to be written to disk, the filesystem checks whether there is enough space on the disk and figure out which sectors it can allocate to the file. Because storage devices are optimized to read sectors sequentally, the bitmap implementation of a sector allocator results in faster `read/write` speeds because it is easier to find sequental sectors represented in a bitmap than going through a linked list. The bitmap is also a lot faster, because a copy can be stored in RAM for faster editing. Lastly, there is the *tree* but because it is used in many different forms, namely *b*-tree or *b+*-tree and alike, it will not be explained in this booklet. Nonetheless, it also keeps track of used and unused sectors. [14]

6.4 Understanding primitive filesystems

To understand why these data structures are needed to make a good filesystem we will start off with a mediocre filesystem and add components to make it better while we are progressing through this chapter. This allows us to comprehend the reasoning behind the incorporation of the components that make up modern filesystems. Let us have a look at a simple filesystem that stores all the files sequentally on the storage device, starting from sector 0. This might seem like a good idea, but when we want to retrieve a file from such a filesystem (now *FS* for short) we would not know where it is located exactly. We would know what the first file's position is, namely the first sector of the medium, but we can't know the file's length and therefore nothing about the position of the following files. This is where a key component comes into play: the *superblock.* It is an array of inodes but in our primitive FS the inodes store the name of the file together with other metadata. The FS supports now various files and because it works so well we start to organize our files. Organizing files requires *directory* support for Directories, also known as *folders* on Windows, are filesystem objects that *"contain"* other files or. We decide to treat directories as regular files, except that we add a flag in the inode that marks it as a directory. In these directory entry files, i.e. the

[13] Superblocks, from: Linfo, November 23, 2021`http://linfo.org/superblock`
[14] Disk Sector, from: Stackoverflow, November 23, 2021
https://stackoverflow.com/questions/13799183/disk-sector-and-block-allocation-for-file





actual content of the directory, we can store the inode number of files and directories that are *"stored"* in it. Our hierarchical filesystem is built like a tree turned upside down, there is the root and every subdirectory is a branch (directory) or a leaf (regular file). Now that files are organized, there is need for a way to access them accordingly.
The next step is adding mechanisms for *path traversal* i.e. the ability to specify the position of a file that is nested within multiple directories. This is done by checking all the inodes in the directory until a file name matching an inodes file name field is found. If the path traverses multiple directories, the previous step of looking for inodes with a matching file name is done recursively.
Note: For readers used to Windows, the path separator / is a \ Note that storing the file name in the inode itself can be a bad idea because an inode is of fixed size and file name sizes vary greatly. One either has to preallocate a large inode size for a long filename or force short file names to safe storage space. This is very restrictive but there is a solution. Instead of storing the file name in the inode, it can be stored `"inside"` the parent directory, together with the inode number. This means that the contents of a directory entry file would look similar to this:

Figure 5: Layout of an example directory entry file *"containing"* `File.md` and `Image.png`

Every character in a box represents a byte on the disk. Notice the `\0` in the red field, it is our file name string terminator and tells us that the following byte in blue is to be interpreted as an inode number. After the inode number, the name of the next file starts and it is terminated with the next `\0`, which is again followed by the inode number of the file. If the byte after an inode number is a `\0` the FS knows that the directory doesn't have any more files.
Let us recap what we have looked at so far. We have the superblock that contains metadata about

files in per-file inodes. The inode stores the location of a file on the disk and contains a field that describes whether it is a regular file or a directory. The directory contains the file names and the inode numbers of the files and subdirectories that are inside the directory. A file path (such as `pictures/summer21/img012.png`) can be used to find the inode of the corresponding file and then the file location on the storage medium.
Our FS is very advanced now. Nonetheless, some things must be cleared up first. Because the name of a directory is stored within its *parent directory* (the directory that contains the subdirectory) there must be a directory that does not have a name, because it does not have a parent directory (unless there are infinitely many directories or a circular filesystem). This directory is the *root directory* and every other directory is either a direct or indirect subdirectory. The root directory is often shown as a plain `/`. Windows users can think of it as the `C:\` drive but that is actually not entirely the case. We will explain why shortly, but there are some things that are important to know first. If a path starts with the `/` such as `/home/Terry/tasks.txt` then the path is an *absolute path*. There is also a relative path and it depends on the *current working directory*. Let's say that a program is run from `/home/Terry/`. The current working directory is the aforementioned one. If the program wants to read the file `homework.txt` it can use the relative path to specify the file it wants to read. The full path of the file is actually "/home/Terry/homework.txt" but because the OS knows the current working directory (*cwd*) it can resolve the inode number of the file. Relative paths are very convenient but they still lack a feature in our FS. If we want to specify a path of the parent directory that does not traverse the *cwd*, we must give the absolute path of the parent directory and then the following subdirectories. But this is another issue that can be solved. Every directory contains at least 2 entries (with the exception of the root directory). These are `.` and `..`. The `.` is a directory entry file that points to itself. An example to understand it better is lets say directory `/home/Terry/pics` has the inode number 8 and its parent directory has the inode number 6. The directory entry file will contain the following:

```
'.' '\0' '\x08' '.' '.' '\0' '\x06' '\0'
```

The first entry in the directory `'.'` points to itself. This means that in a relative path a `'.'` points to the directory itself. From a viewpoint in `/home/`, `'.'` is equal to `/home/`. This is not very spectacular but if you look at the file directory above you can see that `'..'` points to a file with inode 6. This is the parent directory of `'.'`. This is where it gets interesting. A relative path can now contain a `'..'` and now we are able to give a relative path of every single file on the filesystem. `../../` is the path of the parent's parent directory. Unlike the `'..'`, the `'.'` does not have an effect on the path. `../pictures/pic.jpeg` is equal to `../pics/././tree.jpeg`. With our fully fledged FS there are also some speed improvements. If we would like to move a file from `pics/` to the parent directory the FS only has to delete the entry of the ¡file to be moved¿ from the `pics/` directory entry file and write the file name and inode number into the directory entry file of `'..'`. This is a lot faster than copying the contents of the ¡file to be moved¿ and writing them to a new file and deleting the old file. This FS also allows the renaming of open files, something that Windows still cannot do reliably. Windows simply queues the name change and commit it only after the file has been closed.

6.5 Journaling filesystems

Readers that have worked on old machines with primitive filesystems may have encountered one of the most annoying things when trying to get work done: A crash. Back in the days a crash could mean that the subsequent reboot of the system took hours to complete. This is because the system crashed while something was being written to disk and resulted in filesystem corruption because of an unfinished write operation to the storage medium. Usually this was just an issue when a write access to the superblock or other filesystem specific data structures was unexpectedly interrupted. In that case a time consuming disk recovery had to be run to find and fix the error. Journaling file systems came to fix the issue. A new area on the disk got assigned to the *journal file*. Whenever a write operation to the disk is queued the FS writes the write parameters to the journal. This includes the sectors that should be written to and the length of the data to be written to the storage medium. After the write operation has been completed successfully the FS clears the journal by overwriting it. Whenever the system boots up the OS, the filesystem checks the journal and if it finds an unfinished query listed in the journal it will just fix the query. There is no need to check the entirety of the FS and if the journal is empty, the OS can just continue starting up normally. The journal file is not really a file but rather just a few sectors at a fixed location totaling a fixed size for faster access times. Regular files might move around on the disk if they grow in size or some other cases. In summary, a modern filesystem consists of the superblock storing all filesystem objects metadata in inodes, a data structure that keeps track of (un)used sectors, a journal and finally a vast, organized area that holds the contents of regular files and all directory entries.

7 Processes

A process is defined as an instance of a computer program that is currently being executed. The operating system is ultimately in control of all the processes and Processes can be further categorized into what is called a *process life cycle*. The process life cycle is a categorization of the different states a process or program can be in. A process life cycle can vary from one operating system to another and the names are also not standardized. The aforementioned categorisation consists of 5 different stages:

- Start:
  The original state in which a process is first initiated in.
- Ready:
  A ready process in in wait for the processor to allocate them via the operating system. Processes will likely enter this state after the Start state. The other likely reason why a process enters this stage is because the OS scheduler(part of the operating system that manages CPU allocation to processes) will interrupt the process to assign the CPU to another process.
- Running: In this state the process has been assigned the CPU by the OS scheduler and executes its instructions.
- Waiting:
  This is the stage in which the process is waiting for a resource such as waiting for a file to become available or waiting for user input. ¡item Exit/Terminate:
  This is the last state a process enters when it has finished with its execution. The process is then removed from main memory. [15]

7.1 Threads

Threads are the most basic unit of CPU utilisation, meaning that this is the smallest amount that the workload of a process can be split. Threads consist of a stack, a set of registers and a program counter. Usually a process is tied to strictly one thread meaning that the CPU does not split the workload over multiple threads. However in modern programming often multiple threads are used by one process to perform different tasks independently at the same time. This is very useful in modern programming because this means that a task in a process will not block other task. For example a program like word can check for user input with one thread, load images on a second thread, check for grammar errors on a third thread and make backups on a fourth thread. Each thread has its own set of registers and a stack but all threads share the same files, code and data. Modern CPUs have multiple cores meaning that they have multiple processing units this allows multiple threads to do parallel processing because each thread is strictly related to one processing unit. This type of programming that uses threads to complete mutliple tasks in parallel is reffered to as multi-threading and this benefits four main categories:

- Sclalability:
  Multi-threading allows programmers to utilize multiple CPU cores for a single process as opposed to single thread processes which can only utilize a single CPU core.
- Responsiveness:
  When threads are still occupied with tasks in the process multi-threading allows for another thread to still check for input and thus allowes rapid responce from the user.

[15]Processes, from: Medium, November 23, 2021
https://medium.com/@imdadahad/a-quick-introduction-to-processes-in-computer-science-271f01c780da

- Economy:
  Multi-threading is much faster than single thread processes. Managing and creating threads allows for faster completion of tasks.

- Resource sharing:
  Threads share their recources amongst each other allowing for tasks to be completed in parallel in a single adress space. [16]

7.2 Program

A program is a file containing a set of instructions. These instructions can be written in many different programming languages such as: Python, C or C++. These are considered high lever programming languages as they have strong abstraction from the inner machinations of a computer. The CPU however cammpt read these instructions yet they first need to be translated into something that the computer can use. This is done via a compiler. The compiler translates when this program is loaded into memory it becomes a process and can be seperated into four parts:

- The Stack:
  The stack contains primarly local variables but also data such as function parameters and function return adresses. The data on the stack is often just used temporarily.

- The Heap:
  The heap contains memory that is dynamically allocated during the run time of the process.

- The Data:
  The data is the section containing the static and global variables.

- The Text:
  The text segment contains CPU instructions for the process, this is represented by the value of the program counter it also contains the contents of the processor's registers.

Further iformation on these four sections can be found within the memory chapter. [17]

7.3 Daemons and Init

Daemons are computer programs that run in the background of an operating system. Regular users (i.e. users without *Administrator rights* a.k.a *root privileges* on unix-derivatives) do not have direct access to daemon processes and have no control over them. The implementation for daemons will differ from operating system platforms. For Microsoft Windows NT systems, the programs that serve the same functions as daemons are called *Windows services*. In most cases they do not interact with user input and are started during the boot directly by the kernel. Since Windows 2000, the services can be manually started and stopped via the control panel[18]. In Unix-like systems, daemon processes usually end with the letter "d" for example the crond daemon is a job scheduler for background processes. In Unix systems there is also a special daemon from which all other daemons spawn the init daemon. The init daemon is the first process to be started at system boot and then places the sytem in single user mode or spawns a shell to read the system's startup files. [19]

[16] Threads, from: Uic November 23, 2021
https://www.cs.uic.edu/ jbell/CourseNotes/OperatingSystems/4_Threads.html

[17] Processes, from: Tutorialspoint, November 23, 2021
https://www.tutorialspoint.com/operating_system/os_processes.htm

[18] Services, from: Microsoft, November 23, 2021
https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2003/cc783643(v=ws.10)?redirectedfrom=MSDN

[19] Verma: Unix, (2006) P. 84
https://books.google.ch/books?id=JhS-TkWOtOYC&pg=PA84&redir_esc=y#v=onepage&q&f=false

7.4 Executables

Executables are files containing instructions and data for computers. When these files are *run*, they become procesess. Running a executable means that the contents of the file are loaded into memory. Executables are associated with a format that describes how exactly the contents must be loaded. The loading itself is done by the *program loader*. It is a crucial component in operating systems. The program loader maps the individual sections, primarily `.text` and `.data`, of an executable into memory according to the layout and position specified in the executable.

7.4.1 Binary executables

7.4.2 Scripts

There is also a different type of executables, namely *scripts*. Scripts are written in a human readable scripting language. They rely on software to interprete the instructions at run time in contrast of hardware (such as the CPU). The interpreting software is a program that contains instructions for the CPU. Windows uses the **.bat** or **.cmd** extension for scripts written for the interpreter `cmd.exe` and for the newer PowerShell scripts with extension **.ps1** it will use the interpreter called `powershell.exe`.

7.4.3 Executables on unix-like systems

Unix-like operating systems differ greatly from Windows NT ones. They rarely rely on extensions to identify executables but rather *file signatures*. Binary executables unix systems with the exception of MacOS contain the `\x7fELF` file signature and are in the ELF-format. A special type of file signature can be found on scripts. Even though they are made of plain ASCII characters, the author of the file creates the signature by him or herself. In scripts for Unix-like operating systems, the format is as follows:
`#!/path/to/the/script/interpreter -parameters\n` followed by instructions in the scripting language that can be interpreted by the interpreter specified in the path. The *shebang* (**#!**) is the script signature and tells the kernel that the program is not in a binary format such as Mach-O (MacOS Darwin) or ELF (Linux et al.), the unix-like counterparts to Windows **.exe** (PE). When the executable is invoked, the kernel will first invoke the interpreter, which is a binary executable and pass the name of the script to the interpreter.

8 User interfaces

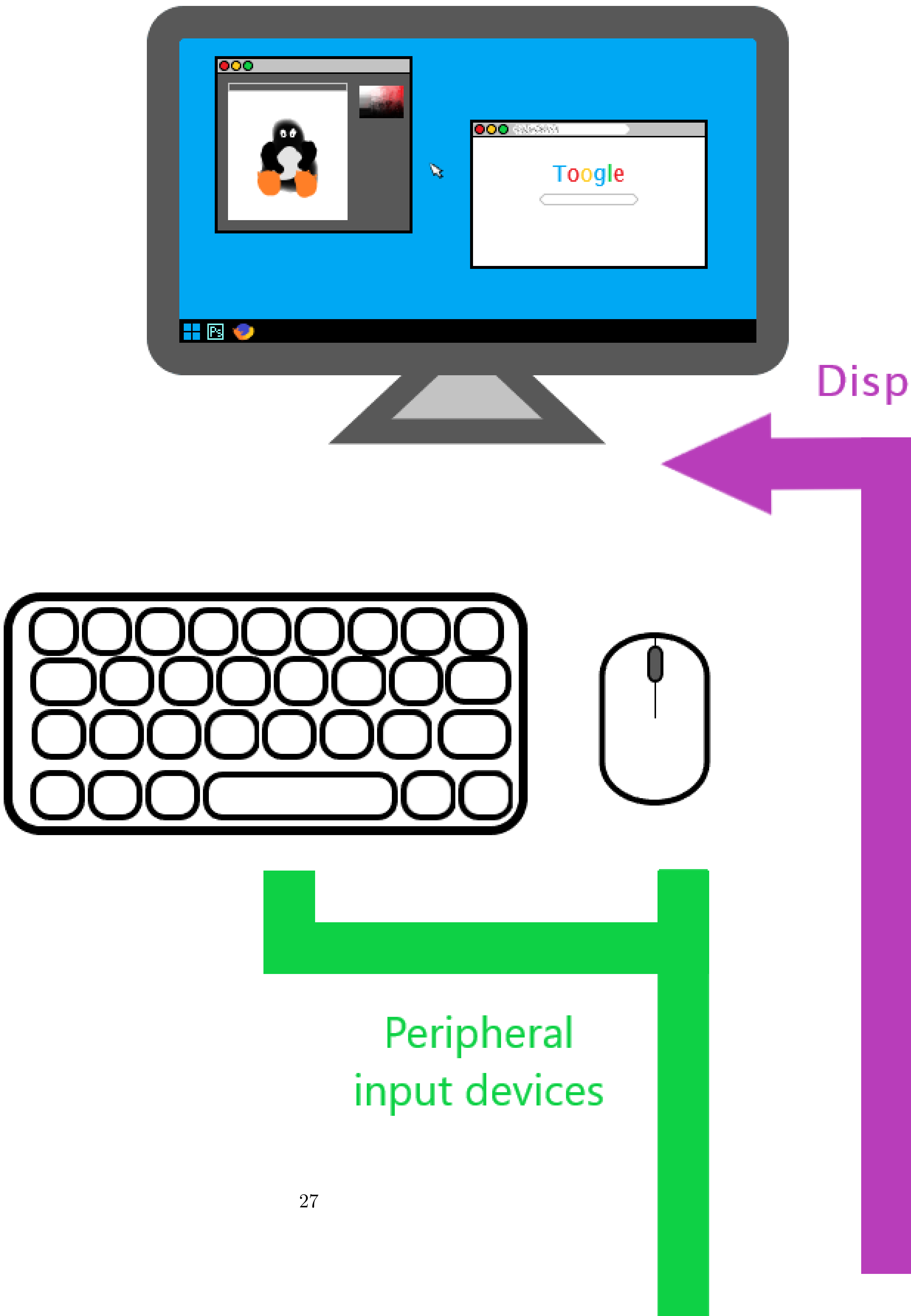
Most definitely, every reader has interacted with a computer before. But most likely there was no direct interaction with the kernel. Not only is it tedious to interact with the kernel but also extremely time consuming. This is where user interfaces (UIs) come to help.

8.1 The shell

The shell is the outermost layer of an OS. Users interact with the shell to start, pause and quit programs. Most readers will be familiar with a *graphical user interface*, GUI for short. GUIs come with a desktop and a taskbar where programs can be started by clicking on the icon. A GUI provides graphical windows that can be moved around with a mouse or a trackpad. Fortunately for users GUIs hide away a lot of the complexity of the OS. Users might think that their browser such as Chrome or Firefox is the only program that is running when they click their browsers icon in the taskbar. But actually many other programs are already running before users even get to see their desktops taskbar, let alone their login screen. Traditionally computers were accessed using a *command line interface* (CLI) that only provide monospaced character output, sometimes even only in monochrome, meaning one color on black background. CLIs don't use a lot of system resources. This is due to the fundamental difference in architecture between GUIs and CLIs.

8.2 Windowing systems and GUIs

GUIs are made of programs that facilitate windows, icons, menus and pointing with a cursor. One of its components is a program called the *display server*. It is responsible for the communication between all the programs that have a graphical output. This communication occurs through a *display server protocol* and the programs communicating with the display server are its clients. In windowing systems every program has its own *window buffer*. It is a dedicated area in memory that the graphical program can render its own graphical output to. Whenever the program has finished rendering its own window it will message the display server over the aforementioned display server protocol. The display server has access to all the window buffers and creates a single frame for a computer screen out of all the windows in a routine called *compositing*. In some windowing systems there is a separate process that composits the windows, namely a *window manager*. The window manager, as a separate process or not, draws the windows and their borders and may add effects such as transparency or gaussian blur. It will factor in the Z-order which is the ordering of windows from back to front and draw the windows in to a frame buffer accordingly. As a last step it will also draw the cursor at the cursor position.


Toogle
Disp
Peripheral
input devices

8.2.1 Rendering

Every program is responsible for rendering its own window buffer. This includes but is not limited to:

- font rendering
- drawing of images
- widgets such as text input fields and checkboxes

Together they form the magnificent windows and desktop background pictures we know and love. But rendering is a (relatively) time consuming and complex process. However, most windowing systems will also include font rendering tools that are optimized for speed and accuracy. There is no point in every program having its own font rendering tool as there is no point in reinventing the wheel. There are some other tools for faster rendering of shapes and images provided by either the display server itself or an extension thereof.
GUIs are heavily reliant on mouse input to drag, resize and reordering windows but keyboard input is just as important for a enjoyable user experience. There may be multiple windows running simultaneously and all of them are waiting for user input. Input from external devices is handled by the kernel. The display manager more or less exclusively acquires the right to the system input and decides whom to send key strokes or pointer events such as scrolling or rightclicking. Additionaly, the display server will check if the input received from the kernel requires a window focus change. Focus indicates the selected window which will get user input. If the display server notices that the user has clicked on a window outside of the current window with focus it will transfer the focus to the new window and change its Z-order to bring it to the front. The display manager will also check wether a click has been made on one of the programs window bar buttons such as the close, minimize or maximize buttons or if the window has been resized by clicking and draging the border. It will then tell the program that it has been resized and the program will redraw its buffer according to the new dimensions of its window and then communicate the changes to the display server.

8.3 CLI

Command line interfaces are a text-only interface that put emphasis on speed, practicality and efficency. The are purely controlled by the keyboard and are operated by entering commands into to the command prompt. The commands are processed by a program called a *command interpreter*. The command line interpreter is a program that awaits textual commands to invoke a program. A program can be started by typing its name into the shell. The shell evaluates the input of the user. Valid input can be one of the following:

- A shell built-in command
- Full or relative path of an executable program or a program name with its path in the **PATH** *environment variable.*
- Interactive scripting keywords

8.3.1 Shell built-ins

Many functionalities that are available to a user are *programs*. Some of those programs do such primitive tasks and encapsulating them in a separate program (i.e. not part of the command line interpreter) would be too much of an overhead. These small functionalities are called *commands* instead of program because they are provided by the command line interpreter itself. Examples of such commands are *cd/chdir*, which is used to change the *current working directory* or *help*, which is used to display helpful information about the command line interpreter. [20]

8.3.2 Environment variables

Typing out full path names of executables is very annoying when invoking commands, especially if there are multiple directories where executables are located. Operating system designers came up with a clever solution to circumvent the inconvenience. Every process is assigned *environment variables.* They are made of **name -¿ value** pairs. Environment variables are passed from a parent process to its child. Every process can change its own environment variables and pass them on to its child processes if it chooses so. Environment variables are extremely useful in context of CLIs. On of those environment variables is '´PATH´'. It is a string with the following format: "path0:path1:path2:path3:..." where path$N$ is an absolute path to a directory and the colon ":" in unix-like systems or semicolon ";" in Windows-family operating systems signifies a separator. When the user enters the name of an executable it will check wether it is a full path. If that is not the case the shell will check its environment variables for "PATH" and look at its value. It will check every directory listed in "PATH" sequentally and probe wether an executable with the specified name exists in the directory. The command line interpreter will let the user know if it fails to find a executable with a matching name. Environment variables are widely used nowadays but are mostly hidden from users autilizing a GUI. The GUI will hide away most of the complexity of the OS, one of them being environment variables.

8.3.3 Scripting

Users sometimes have more demanding requests of programs they wish to invoke. Users wishing to invoke a program one hundred times would need to type in the desired programs name a hundred times. Invoking a program over a hundred times is rather unusual in a GUI environment but in a CLI it may have its appliances. There are **shell builtins** to get around this issue. These are tools provided by the shell (not individual programs) that allow for primitive but nonetheless powerful scripting. Such functionality can be used by typing in keywords such as 'while', 'if', 'else', 'do' and alike. As mentioned, the command line interpreter is a program and the implementation may vary. Some interpreters have a slightly different syntax. [21]
CLIs appeared in the times of teletype machines that could only display monochrome color output. All modern monitors are capable of displaying high resolution multi-colored images and todays graphics accelerated hardware has become cheaper and gotten fast, which resulted in GUIs prevailing over CLIs. However, CLIs are very convenient for system administrators and automation of routine computer tasks such as backups. Most graphical OSes come bundled with a *terminal emulator*, a graphical program running a command line interpreter (such as CMD). This provides flexibility for people who use GUIs for their web browsing et al. and system administrative tasks or running text-only programs.

8.4 Remote Access

As computers incorporated more and more internet capabilities, engineers added functionalities for *remote access.* This meant that a user connected to a *local area network* (LAN) with his computer was able to interact with another computer located on the LAN and interact with the remote machine as it was a local device.

[20]Shell, from: Wikimili, November 23, 2021
https://wikimili.com/en/Shell_(computing)
[21]Forsythe: Scripting, November 23, 2021
https://wikimili.com/en/Shell_(computing)

8.4.1 Secure shell

The first type of remote access was a login program listening on the network. It was over a protocol called telnet and a user was able to connect to another computer running a *telnet server* using a *telnet client*. However, it was quickly overtaken by a new protocol due to its unencrypted nature: **SSH**, the *Secure SHell*. SSH works similar to telnet in a sense of accessing a remote computer. A user can log in to a different computer running a textitSSH server by using a *SSH client*. Unlike telnet, SSH encrypts all internet traffic, which is crucial when sending passwords over a network. SSH has become a standard today and is widely used to manage server computers on both local and wide-area networks. [22]

8.4.2 Remote Desktop

Many programs (such as web browsers) exist only in *graphical mode* and do not support a text-only interface. However, remote access can also be served in a graphical format. The protocol differs greatly from SSH because it must additionaly support mouse input and screen output. On the host machine, the display server redirects the framebuffer output to the client. The client consistently sends keystrokes and mouse events to the display server. Example of such protocols is the *Remote Desktop Protocol (RDP*.

[22]Secure shell from: CMU, November 23, 2021
https://computing.cs.cmu.edu/security/security-ssh

9 Sources

Many webpages we used to gather information to learn about operating systems didn't provide us with just a few information points that can be cited. We feel like these websites deserve honorable mentions. They range from very hardware specific info sites to wikis.
SORT BY ALPHABET
1. Ralph Brown's Interrupt List, hosted at http://www.ctyme.com/intr/alhpa.htm, retrieved on November 23, 20212. OSDev Wiki 3. Félix Cloutier6 refrence, https://www.felixcloutier.com/x86/, retrieved on November 23, 20214. 5. 6.

References

[1] Ahmad, Imdad: A quick introduction to processes in Computer Science, https://medium.com/@imdadahad/a-quick-introduction-to-processes-in-computer-science-271f01c780da, Downloaded on: November 23, 2021

[2] Verma, Archana: Unix and Shell Programming, New Delhi: Firewall Media (2006), ISBN: 81-7008-959-X

Without authors:

[3] Booting in Operating system, from: Javapoint, https://www.javatpoint.com/booting-in-operating-system, Downloaden on: November 23, 2021

[4] Bellevue Linux Users Group: Kernel Definition, http://www.linfo.org/kernel.html, Donloaded on: November 23, 2021

[5] Storage devices, from: Tech21, https://www.tech21century.com/different-types-of-storage-devices/, Downloaded on: November 23, 2021

[6] Superblocks, from: Linfo, http://linfo.org/superblock, Downloaded on: November 23, 2021

[7] Disk Sector, from: Stackoverflow, https://stackoverflow.com/questions/13799183/disk-sector-and-block-allocation-for-file, Downloaded on: November 23, 2021

[8] Interrupts, from: Tutorialspoint, https://www.tutorialspoint.com/embedded_systems/es_interrupts.htm, Downloaded on: November 23, 2021

[9] Processes, from: Medium, https://medium.com/@imdadahad/a-quick-introduction-to-processes-in-computer-science-271f01c780da, Downloaded on: November 23, 2021

[10] Threads, from: Uic, https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/4_Threads.html, Downloaded on: November 23, 2021

[11] Processes, from: Tutorialspoint, https://www.tutorialspoint.com/operating_system/os_processes.htm, Downloaded on: November 23, 2021

[12] Services, from: Microsoft, https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2003/cc783643(v=ws.10)?redirectedfrom=MSDN, Downloaded on: November 23, 2021

[13] Forsythe: Scripting, https://wikimili.com/en/Shell_(computing), Downloaded on: November 23, 2021

[14] Secure shell from: CMU, https://computing.cs.cmu.edu/security/security-ssh, Downloaded on: November 23, 2021

[15] CPU, from: Learncomputerscienceonline, https://www.learncomputerscienceonline.com/what-are-cpu-registers/, Downloaded on: November 23, 2021

[16] Memory, from: Javapoint, https://www.javatpoint.com/volatile-memory, Downloaded on: November 23, 2021

[17] Jumps, from: Infoscinstitute, https://resources.infosecinstitute.com/topic/conditionals-and-jump-instructions/, Downloaded on: November 23, 2021

[18] Negative Numbers, from: Binaryhakka, https://binaryhakka.blogspot.com/2020/03/assembly-language-negative-numbers.html, Downloaded on: November 23, 2021

https://en.wikipedia.org/w/index.php?title=List_of_file_signatures&oldid=1056162959
https://superuser.com/questions/1204522/dev-sda-equivalent-in-windows
https://www.tech21century.com/different-types-of-storage-devices/
https://www.lsoft.net/posts/file-signatures/
https://uefi.org/sites/default/files/resources/UEFI_Spec_2_9_2021_03_18.pdf
All graphs and images, unless stated explicitly otherwise, were created by us.