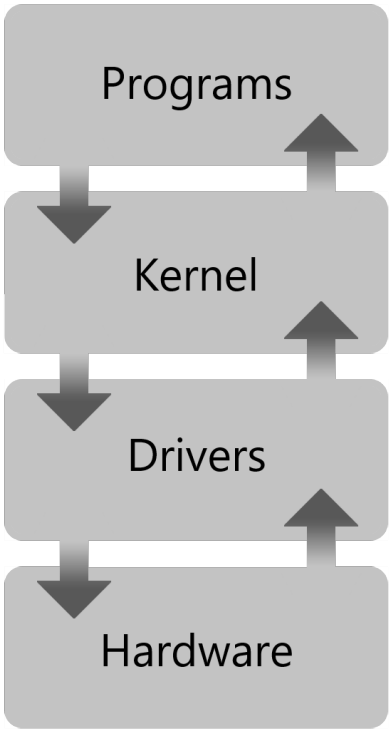
Operating Systems
Structure and Implementation

Daniel Huber, Nick Gilgen and Moray Yesilgüller

November 22, 2021

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



Maturaarbeit Kantonsschule Baden

Contents

1 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, GPU as well as storage devices. If system critical components such as RAM or the processor are not present, the PC speaker will let of a beep and turn off. After a successfull POST, the BIOS will gather some details about the hardware and write some basic information of the hardware into the Bios Data Area (BDA). Because the BIOS prepares the machine for an operating system (OS) it presents the hardware 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 time and sets up interrupt handlers (more about interrupts in chapter XXXXXXXX). Finally, the BIOS will search 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 as the identifying number for bootsectors. When the BIOS finds a valid bootloader sector, it will load the entire sector into RAM at the fixed location 0x7C00. Yet again, this value was chosen randomly by the OEM's back in the 1970s but to this day IBM-PC compatible systems will load the bootsector into that location. After a successfull load, the BIOS will transfer control to the bootloader by telling the CPU to fetch instructions from location 0x7C00.

1.1 Bootloader

The bootloader has 2 important tasks. The first one is gathering more information about the RAM of the system. It will determine wether there is enough RAM to load the OS and if this is true it will load the OS. Loading the OS means that it will access the storage device and read 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. It is interrupt number 0x13 in hex or decimal 19 and it is invoked with parameters that will load x many sector on y many platters into RAM address determined by a parameter. After a successful load the bootloader hands over the execution to the operating system, specifically the *kernel*.

1.2 Kernel

The kernel is the most important part of the OS. It controls 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 maaged 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 Kernels 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 kernels 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 no one is requesting any. The other components of an operating system are programs that create an environment that provide an user experience. Such programs are responsible for things such as 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.
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

1.3 UEFI

2 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.

2.1 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 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:

2.1.1 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.

2.1.2 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.

2.2 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 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.

2.3 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 .

2.4 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 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.

2.5 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**.

3 Filesystems and resource management

As mentioned before, one of an operating systems tasks is the managing of resources. One of those resources is storage. There are 3 main reasons why storage (and many other hardware devices) are hanlded 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.
We will look at them in greater detail further in this chapter. But first, we will have a look at what a filesystem actually is. 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 retrived 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.

3.1 Storage devices

Storage devices such as hard disks, floppy disks, USB flash drive and SSDs have no notion of files. The only thing these devices know are *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 IO ports, sometimes in the form of SATA connectors or USB slots, where they can receive commands from the CPU on the system. A command can be to 'READ' sector number ***x*** to RAM address ***y*** or 'WRITE' from RAM addres ***y*** to sector ***x***. This command is sent through specific IO 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 [**?**].

3.2 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 in 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 being 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 another program than the present one 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 different names. Let's say that the users C compiler is GCC and located at '/bin/gcc/'. A soft or hard link can be created to maps the filename in path '/usr/bin/cc' to '/bin/gcc'. If files were associated only 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 thats 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 the inodes in the superblock are used up. This happens only when a computer user creates many files that take up close to no space.
It is important to point out that Windows' NTFS groups file metadata differently. We are not going to cover NTFS because the official version is proprietary. While most mechanism in NTFS are

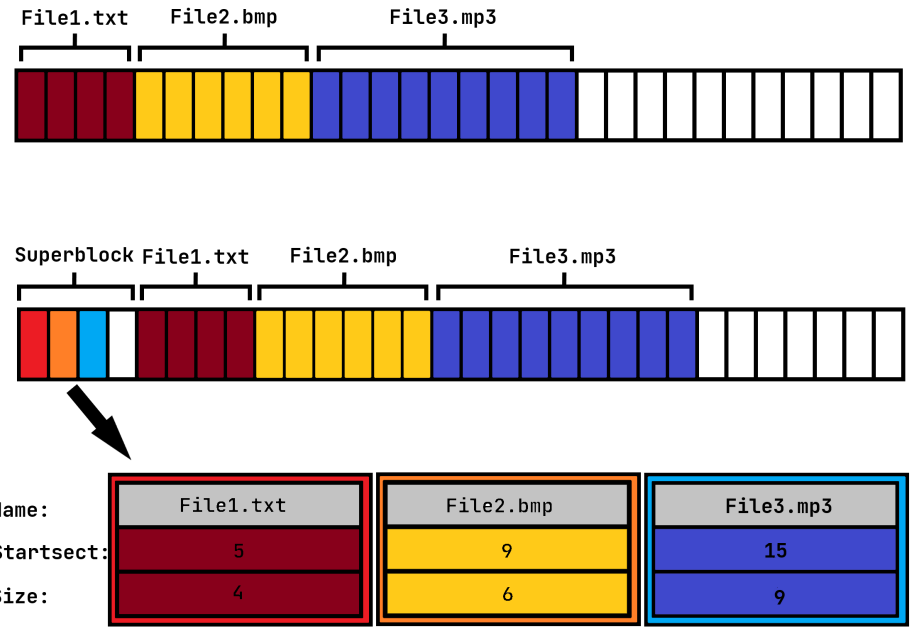
similar to unix filesystem mechanisms, they often have a different name. The superblock is called the *Master File Table* and inode numbers are called *FileID*s.

3.3 Sector allocation

Another importand data structure in filesystems is used by the *sector allocator* to keep track of the occupied and free 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 will only have to 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 list (free sectors) 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 thats 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. Tha bitmap can be thought of as an array of ***N*** bits where ***N*** is the number of sectors (512, 2048 or today 4096 bytes). Every sector on the disk is enumerated and it is represented in the bimap (a.k.a *bit array*) at bit number ***N***. Said bit in the bitmap will be either 1 (free) or 0 (empty). Whenever a new file has to be written to disk, the filesystem will check wether 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 can result 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 etc., it will not be explained in this booklet. Nonetheless, it also keeps track of used and unused sectors.

3.4 Primitive filesystems

To understand why these data structures ara needed to make a good filesystem we will start off with a mediocre filesystem and add components to make it better on the run. This allows us to comprehend the reasoning behind the incorporation of the components that make up modern filesystems. Let's 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 files position is, namely the first sector of the medium but we can't know the files length and therefore we won't now the position of the following files. This is where a key component comes into play: the *superblock*. It is an array of inodes and our primitive FS stores the name of the file in the inode itself together with other metadata.
Our 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 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 previos step of looking for inodes with a matching file name is done recursively.



4 IMAGE OF PATH TRAVERSAL

Note: For readers used to Windows, the path separator "/" is a "¨ After using the filesystem for a while, we notice that storing the file name in the inode itself is a dumb idea because an inode is of fixed size. We either preallocate a large inode size for a filename or we 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, we store it in the directory where the file is located together with the inode number. This means that the contents of a directory entry file would look some thing like this:

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 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 wether 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 'Pics/Party/Kodak/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 it 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 afeature 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 the itself! This means that in a relative path a "." will always point 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 "../pictures/./.pic.jpeg". The "." does have its usecase but it is largely to reasons unrelated to filesystems themselves. More about "." and its purpose can be read in another chapter. 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 will queue the name change and commit it only after the file has been closed.

4.1 Journaling filesystems

Readers that have worked on old machines with primitive filesystems maybe 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 rebbot 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 interrupted unexpectedly. 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 will write 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 successfuly the FS clears the journal by overwriting it. Whenever the system boots up the OS again the filesystem will first check 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 normaly. 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.

References

[1] https://www.tech21century.com/different-types-of-storage-devices/, retrieved November 22, 2021