File Systems

Files

So what exactly is a file? It's an abstraction the operating system provides us so that we don't have to deal with the nitty-gritty details of converting our data into 0s and 1s, storing those 0s and 1s as magnetic or electronic impulses on a disk, and then remembering exactly where all of that is located on the disk. The operating deals with all of that for us. It allows us to bundle our information into an object we call a file, and it keeps track of where all the information is located on the disk. It also stores *metadata* about the file, which are things like the file's size, last modification date, etc.

In most operating systems, files are given a *type* that is usually, but not always, indicated by its *extension* (like .jpg, .pdf, .docx, etc.). The file's type is determined by how its bits are stored. For instance, a jpeg image starts off with some header data followed by RGB values of each pixel. A program that reads jpegs looks at the header to get the info it needs and then reads the image data. It assumes everything is in a very specific place. If you open that jpeg in another program, like a text editor, you would just see garbage. A simple ASCII-based text editor would assume that the bits are arranged in groups of 8, each one corresponding to a specific ASCII character. When we open a jpeg in such a text editor, it tries to read the bits of the file as if they were ASCII characters, and we end up getting a weird output since the jpeg is not designed that way.

File systems

The file system is what the OS uses to keep track of files. Each OS has its own system, and that system sometimes changes from version to version. On Linux some of the common file systems are ext4, ext5, and XFS. On Windows, there is FAT32 and NTFS. On Mac, there is HFS and APFS. The file system controls things like naming conventions, where files are stored on the disk, how the system recovers from a crash, etc.

How file systems keep track of which blocks are free on a disk

Space on the drive is broken up into equal-sized regions called *blocks*. Typically, blocks are a few kilobytes in size, though the exact value various from system to system.

One technique that is pretty efficient for keeping track of which blocks are free is called a *bitmap*. Each block of the disk is represented by a single bit with 0 indicating the block is free and 1 indicating it is occupied. For instance, suppose we have a very small drive that has 10 blocks, with the first two and last two blocks free and the others in use. Then the bitmap would be 0011111100.

If the block size is 4 KB and we have a 1 TB drive, then there would be 1,000,000,000,000 / 4000 = 250,000,000 blocks in total. Each block takes 1 bit to store, so dividing by 8 tells us that we would need around 30 megabytes to store the bitmap. There are certain compression techniques that can be used to reduce this amount. For instance, if a part of our bitmap consists of 500 1s followed by 1000 0s, we could use *run-length encoding* to store that as [500,1,1000,0], which takes considerably less space than listing out all those 0s and 1s.

How to store which blocks a file consists of

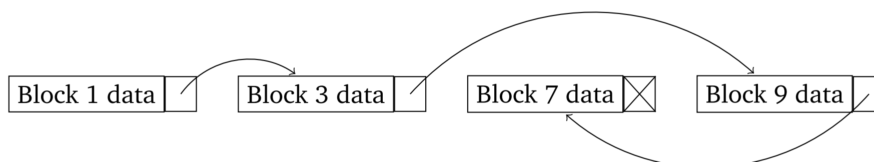
There are three common approaches that we will cover here.

Approach 1 Store the blocks contiguously. In other words, the file is all in the same place on the disk. This gives good performance, especially with HDDs, because we don't have to jump all over the disk with slow disk seeks. The slow part of HDDs is repositioning the read head and waiting for the platters to spin around. Once that is done, reading bytes sequentially is relatively fast.

But there is a serious drawback, namely that as files are continually created and deleted, the disk becomes fragmented, where there may still be a fair amount of space free on the disk, but it's all trapped in small unusable holes between files. In this case, we can defragment the disk, but that's a slow process, often taking hours. It's also unhealthy to use on SSDs. Each cell in an SSD has a limited number of writes before it wears out, and defragmenting adds excessive wear.

Because of this, storing files contiguously is not used except in very particular use cases. One of those is for CD-ROMs and DVD-ROMs, which are only written once, so fragmentation is not an issue.

Approach 2 Use a linked list. Here we'll assume that the file's blocks are stored all over the disk, not necessarily all next to each other. At the end of each block, we add a pointer that points to the block where the next part of the file is stored. For instance, let's suppose that a file's data is stored in blocks 1, 3, 9, and 7, in that order. The linked list approach for this file is pictured below.



This pure linked list approach is not used because if we only need to read one of the later blocks in the file, we have to walk through all the earlier blocks, which can be really slow on an HDD, with all the disk seeks that are involved. Instead, what we do is move the links into a table. For example, suppose the system has blocks 0 to 15, with File A being in blocks 2, 5, 14, 9, and 8, and File B being in blocks 6, 7, and 12. The table would look like below. In that table 0 indicates a free block, −1 indicates the end of a file, and everything else is a link. For instance, the entry for Block 2 is 5 indicating that the next block of the file after Block 2 is Block 5.

Block	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Link	0	0	5	0	0	14	7	12	−1	8	0	0	−1	0	9	0

The file system stores the starting block for each file and then uses the table to locate other blocks. Reading the table is much faster than walking the blocks on the disk. This table is called a *file allocation table*. It is the basis of the FAT file system, which has been an industry standard for years. It is supported by all major OSes and was the primary file system of DOS and Windows prior to XP. It was extended to a 32-bit version called FAT-32. Versions of FAT are still used on some USB drives, flash memory cards, and digital cameras. It's not practical for large drives as the table becomes too large to store in memory. For instance, a 4 TB drive with 4 KB blocks would require several gigs of RAM to store the table.

Approach 3 Use *Inodes*. An inode is a data structure that uses a list to store the blocks that make up a file. For instance, if a file contains blocks 1, 3, 9, and 7, then the inode would store the list [1, 3, 9, 7]. Often a fancier data structure like a B-tree is used instead of a list. Inodes also store the file's metadata. Most modern file systems use inodes or something like them.

Inodes require less space than a file allocation table. In main memory, we only need enough space for the currently open files. Note that if a file is large, the list of blocks can get very long, so what is often done is something analogous to multilevel page tables where the list is not just a list of blocks, but a list of lists of blocks. To store really huge files, a different system called an *extent* is used, which is basically a contiguous block of memory.

Each file is associated with a single inode. Those inodes are stored in a table. In some systems, the total number of inodes is a fixed (though very large) value, meaning you will never be able to have more than that number of files, even if you have a lot of space remaining on your drive.

Layout of a drive

Shown below is the typical layout of a drive.

					Active partition			
master boot record	partition table		boot block	super block	bitmap	inode table	data blocks	

The master boot record contains info that is used to boot up the computer. The hard drive may be partitioned into parts for different operating systems. The partition table indicates how the disk is partitioned and which one is the active partition that the system will boot into. Shown after this is the active partition. Further partitions aren't shown in this picture.

In each partition, the first block is the boot block which contains info for booting the OS. Next is the super block, which contains info about the file system, such as its type, how many blocks are in it, etc. Following that is the bitmap or whatever data structure is used to keep track of free blocks. The inode table comes next. After that are the blocks that hold actual data.

Important system matters

Note on deleting files A really important fact to remember is that when you delete a file, its contents are not immediately wiped from the drive. All that happens is that its blocks are now relisted as free, but their contents are left as is. Eventually, something new will be put in the those blocks, but until then, the old data is still there. If you ever need to dispose of a drive that contains sensitive data, there are special tools to run that overwrite the entire drive multiple times with garbage. And if it is really sensitive data, then the only safe option is to physically destroy the drive, often by dissolving it in acid.

Crash recovery Whenever a file is created or written to, a lot happens behind the scenes. In particular, here are three important things that need to be done.

1. The bitmap has to be consulted to find free blocks in which to write the data, and then the bitmap needs to be updated.
2. The inode needs to be updated with the new blocks that are part of the file and any changes to the metadata.
3. The actual data needs to be written to the data blocks of the drive.

To save time, rather than do this all at once, often a bunch of disk I/O operations are queued and then done all at once. Sometimes the actual write operations can happen as much as 30 seconds later. So if the system crashes in the meantime, you might lose data even if you saved it some time before the crash. And the order of these I/O operations might be determined by some other part of the OS or by the hard drive firmware, so there's no guarantee about the order that 1-3 above will occur in.

One of the important features of a file system is how it deals with inevitable crashes. Bad things can happen if a system crashes in the middle of a drive write. In particular, if some of steps 1-3 above complete, but not others, then we can lose data or the file system can become corrupted. Here are a few of the many possibilities that can go wrong:

1. If the file data is written to the drive, but the inode is not yet updated, then the data that is written to the drive will not be recorded in the file system and will be inaccessible. Users will see an old version of the file.
2. If the inode is written, but not the file data, then things are worse. The file system is now corrupted because the file system will think certain blocks do or don't belong to a file, and that won't match what is

actually on the drive. Users may see garbage in their file. If it's an executable file, then it may not run at all.

3. If the bitmap is written but not anything else, then we won't have an accurate representation of what is free and what isn't. If we were writing data requiring new blocks, the bitmap would have those blocks as being occupied, but if the data didn't get written before the crash, then those blocks will not have anything in them. They will be permanently marked as occupied even though they are really free.
4. If the data and inodes are written, but not the bitmap, then we could have a situation where the system thinks that certain blocks that are part of a file are actually free. Eventually something new might use those blocks, causing a corruption of the file.

For the last two problems, one solution is to use the inode table to remake the bitmap. This is the approach used by Microsoft's scandisk utility and Linux's fchk.

Journaling, log-structured file systems, and copy on write Journaling is a technique used by some file systems to help reduce problems associated with crashes. The basic idea is that you write down what you plan to do before you actually do it. So if a crash happens in the middle of a drive write, you can replay the information in the journal to make it right.

An issue here comes up if the crash happens while you're writing the journal entry itself. There are a few things that can be done to deal with this. One is to write the data first, then write the journal entry, and finally write the inode and bitmap information. This mostly eliminates the problem of file system corruption. If a crash happens while writing the journal entry, then we are in situation #1 from the previous section, where we lose the most recent updates to the file, but the inodes and bitmaps are still consistent. We also need to make sure to have a special marker to indicate the end of a journal entry so that the system realizes when a journal entry is incomplete due to a crash.

There are some file systems called log-structured file systems that structure the file system as one big journal entry along with a table indicating what the current entry is for each file. The tricky part of this is that it tends to fill up the drive quickly, and then you have to decide which old log entries to delete. This approach is used on some CD-RW and DVD-RWs as well as some flash memory where the number of rewrites is relatively small.

Some systems, including the newest Apple file system, use something called *copy on write*. The idea is to never overwrite old data. Instead always write a new copy. The system deletes copies that are several versions old.

Data integrity

There are a variety of different things that can go wrong when writing to a disk.

One thing is a *latent sector error*. Recall that a HDD is a mechanical device and sometimes mechanical problems can happen, like the read head bumping into the surface of the disk. If that happens, it ruins that section of the disk. Another cause of latent sector errors is cosmic rays. These are a type of electromagnetic radiation usually emanating from deep space, and they can flip bits on drives (and other places). According to Wikipedia, the error rate due to cosmic rays is something like 1 error per every 256 megabytes per month, and probably higher if the drive is at high altitude or in planes a lot. A related issue is *data rot*. Storage media have a limited lifetime before they start to break down. Most common options like USB memory sticks, DVDs, and floppy disks can last anywhere from 5 to maybe 30 years under ideal conditions before the data starts developing errors.

One solution to these problems is to use in-disk *error-correcting codes*. The idea is that when you write data, you write some extra information beyond in addition to the data. This is a little like the parity bits used in RAID, but a different scheme is used. In the case of an error, you can use the extra data to both determine that something has gone wrong and fix the error. But if the error is too big, then the error-correcting code won't help, and the disk read will return an error.

The second type of error is *block corruption*. This can happen from a bug in the disk's firmware that causes a block to be written to a wrong location. The corruption can also happen in the bus as the data goes to the disk.

The solution here is to use a checksum. This again is similar to the parity calculation used by RAID, though unlike an error-correcting code, checksums just detect errors; they can't be used to fix them. The checksum is stored in the block with the data or possibly somewhere else. When you read the data, you calculate the checksum and compare it to the expected checksum.

Types of checksums

There are several types of checksums. Here are a few.

1. Simple XOR — Group the data into bits and XOR them together. For instance, if our block is 0010000010100111, we could break it into groups of 4, namely 0010, 0000, 1010, and 0111, and XOR those groups together. This is just like adding them without worrying about carries. The result is the checksum (which is 1111 in this case). It is simple and fast and catches all single bit errors. But it misses some multi-bit errors, such as two flipped bits in the same column.

2. Fletcher's checksum — Here is the pseudocode for the algorithm:

   ```
   sum1 = sum2 = 0
   for each bit in block:
       sum1 += b
       sum2 += sum1
   check1 = 255 - (sum1 + sum2) mod 255
   check2 = 255 - (sum1 + check1) mod 255
   ```

   This still pretty fast, though not as fast as Simple XOR. It catches all single and double bit errors, as well as many "burst errors" where a whole group of bits in a row gets corrupted.

3. CRC — This is a nice checksum that is a bit slower than the others, but it catches more errors. The math behind it is more sophisticated than we can get into here. It is also widely used in networking.

Checksums add some storage overhead. A 4 kb block typically has an 8 byte checksum, which is about a 0.2% overhead. Computing the checksum also adds some time to each disk read. Sometimes these checks are held off until a time when the system is not busy. One weakness of checksums is they won't detect a situation where a write didn't take, like if the read head fails to properly magnetize a section of disk. In this case the checksum that was there from the previous data would still come out correct (for the old data).