Unit 10 – The FAT32 File System

Unit 10.1 – Introduction

First the meaning of the name. **FAT** means that cluster allocation is done with a **file allocation table**. **32** refers to the fact that addresses are 32 bits long. Sometimes FAT32 is referred to as **FAT32 LBA**. **LBA** means that **logical block addresses** are used instead of cylinder/head/sector addresses. Another designation is **VFAT**. This was an extension of FAT to include long file names. The modern FAT32 file system stores long file names.

FAT file systems were used from the early days of MSDOS. As drives got larger (the early ones were floppy disks) the size of the addresses grew from 12 bits, to 16 bits and finally 32 bits. The cylinder/head/sector addresses put a limit on addressing so that when drives exceeded 2 GB in capacity that scheme had to be abandoned. The modern FAT32 addressing extended this beyond the 2 GB limit, so it is still in use today.

Microsoft stopped using FAT file systems for operating systems with the advent of Windows-NT in the mid to late 1990s. Early versions of Windows assumed that there was only one user. The FAT file system didn't have any way to store file ownership or file access permissions, so it was unsuitable for an operating system that could handle multiple users.

The one arena where FAT32 still survives is in USB drives. They need to go from a camera, to a PC, to a phone, etc. Support for FAT32 is ubiquitous, so it is still the standard on USB drives.

The allocation unit of FAT32 is the **cluster**. I've seen cluster sizes from 8 sectors to 64 sectors on modern USB drives. This is not due to a limitation of the addressing, but is a compromise to keep the size of the FAT small. The larger the FAT, the less room to store files. On the other hand, the larger the cluster size, the more wasted space in storing small files. In the extreme case of 64 sector clusters a file of a single byte would require 32 kB of space. All the space after that first byte is slack.

Unit 10.2 – Common Features of File Systems

Before we get into the details of the FAT32 file system let's look at some features that all file systems have.

All file systems have...

1. A group of blocks, usually 4 kB (4096 bytes) in size (but other sizes are possible). These blocks are used to store new directories and files as needed.

2. A scheme to keep track of which block are used and which are unused.

3. An initial sector that describes where to find the root directory and how big the various tables that keep track of things are.

4. A directory structure that starts with the root directory. The root directory can store files and subdirectories within it.

5. A way to record which blocks belong to a particular file or directory.

Most file systems also have a way to indicate file ownership and access permissions. The FAT32 file system does not have this, which is one of the reasons that it has been replaced as the OS file system in modern Window's system.

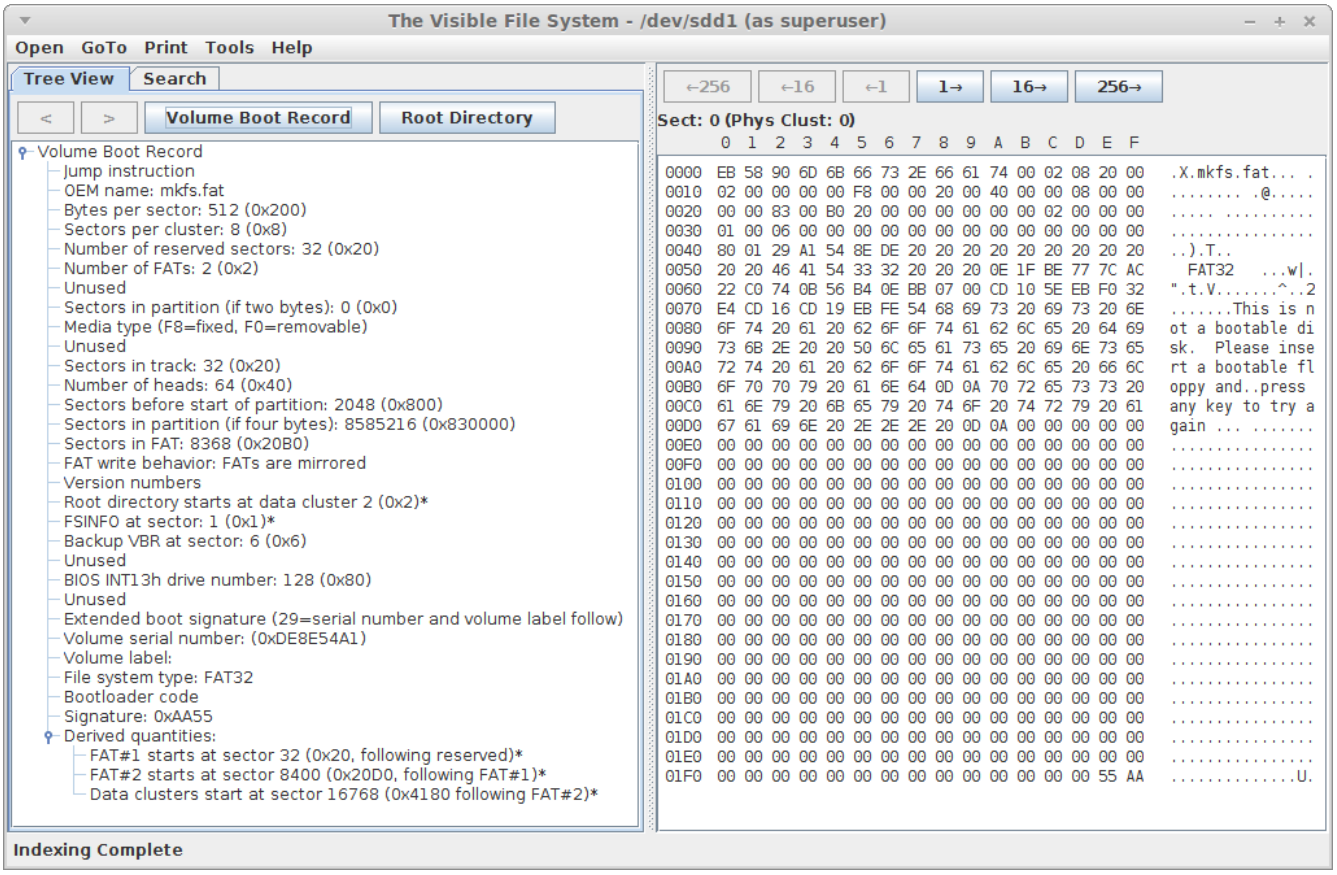
You will see that file systems are very different in how they manage all of these things.

Unit 10.3 – The FAT32 Volume Boot Record (VBR)

The first sector of a FAT32 partition is the **volume boot record** (VBR). It contains information about the partition such as the number of bytes per sector, the number of sectors per cluster, the number of reserved sectors, how many FATs there are, the size of each FAT, etc.

Reading the VBR the OS knows where the data clusters start and can find the **root directory** (always at data cluster #2).

Here's a screenshot of the VisibleFS program showing the VBR of a FAT32 file system.



Notice the number of sectors per cluster, etc. It also tells you that the root directory is at data cluster 2.

In FAT32 LBA we must distinguish between "**physical clusters**" and "**data clusters**". Physical clusters start at the beginning of the partition and data clusters start immediately following the second FAT, and the first one is numbered 2. Sometimes the data clusters are not aligned with physical clusters (off by a few sectors).

All references to clusters in directories and in the FAT are references to data clusters. These must be mapped to physical clusters in order to go to the correct sector.

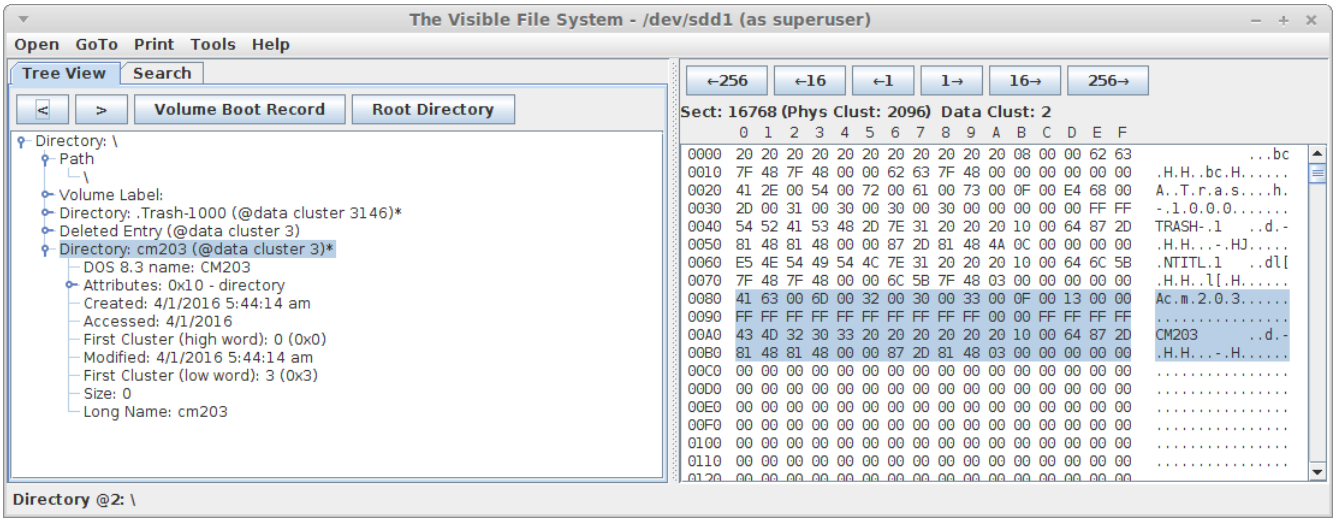
By the way, there is usually an **FSINFO** record in the second sector of the partition. It gives the number of free data clusters and the number of the next free data cluster. This is what the OS uses to figure out where to start the next file. My guess is that this will prevent fragmentation, using all the empty space on a new drive all the way to the end before going back and filling in the holes.

Unit 10.4 – The Root Directory

The location of the root directory is given in the VBR, but it is always at data cluster 2. Directories can span many data clusters, in which case their data clusters are allocated using the FAT, similar to files

(see next section). In the example below the root directory only occupies one cluster.

Here is a look at the root directory in the VisibleFS program.



Of the 4 highlighted lines the last two are the actual directory entry. The first 2 are the long file name (I'll get into that later).

Starting at 0xA0 we see the first 8 bytes of the DOS 8.3 filename: “CM203 ” (note the 3 spaces). At 0xA8 the 3 bytes of the file extension are spaces (it is a directory so they should be). At 0xAB there is a 0x10 flag which tells us this is a directory. From 0xAE to 0xB1 are bytes 0x48812D87 (little-endian) which is the created timestamp(see below). The next two bytes are the accessed date (0x4881), followed by the high word of the starting data cluster (0000), then the modified timestamp from 0xB6 to 0xB9 (0x48812D87) and then the low word of the starting data cluster (0x0003). The next four bytes are the size allocated to this entity (0x00000000) which is zero since it's a directory.

DOS timestamps can be analyzed as follows. The first two bytes are the date. The number 0x4881 written in binary is 0100100010000001. Next separate the bits into groupings of 7, 4 and 5 like this: 0100100_0100_00001. The first group is the number of years since 1980. In this case it is 36. The year is thus 1980+36 = 2016. The second group gives the month, which is 4 or April. The last group gives the day, which is 1. So the date is April 1, 2016.

The next two bytes are the time. The number 0x2D87 in binary is 0010110110000111. Separating the bits in groups of 5, 6 and 5 we write 00101_101100_00111. The first group is the hour, which in this case is 5. The second group is the minute, which in this case is 44. The last group gives the number of double seconds, which is 7. Multiply by 2 to get 14 seconds. The time is thus 5:42:14. AM and PM are included in the hour, which can range from 0 to 23. There is no way to account for daylight savings time in this scheme. Also notice that all second values are even numbers.

Unfortunately the cluster number is separated by the modified timestamp and the program reading this directory must piece together two different numbers. Thus the starting cluster of this directory is

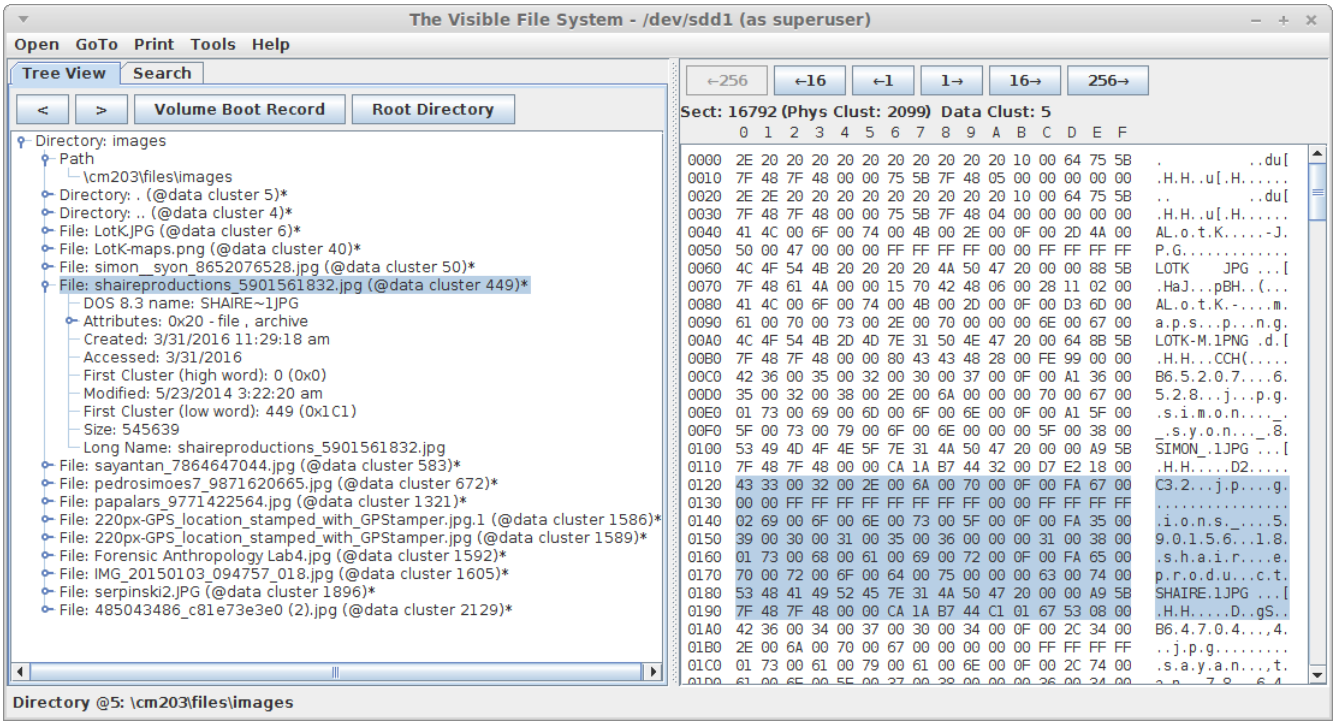
0x00000003. (I speculate that originally the access date was a 4 byte timestamp like the others, but as addresses got larger they took two bytes from the access timestamp and used it to make the cluster addresses bigger.)

The **long filename** is a bit kludgey (it was definitely added as an afterthought). To get the long filename you work backwards from the actual directory entry looking for flag bits of 0x0F. The first byte gives you a sequence number (in this case one record above is marked with a 41. The 4 tells you that this is the end of the long filename records. The 1 tells you it is the first record of the long filename. We'll see directory entries with longer file names a little later.

After the first byte we see the long file name in UTF-16 little-endian. You need to skip over 3 bytes when you hit the flags (offset 0xB to 0xD), and two more bytes where the low word of the start cluster would be stored (offset 0x1A to 0x1B), then resume at the second byte of the next record. Unused bytes after the file name are filled with 0xFF.

The last entry is a deleted entry. **Deleted entries** are marked with a 0xE5 in the first byte. The rest of the data, including most of the filename and all the timestamps are still intact (and sometimes of forensic value).

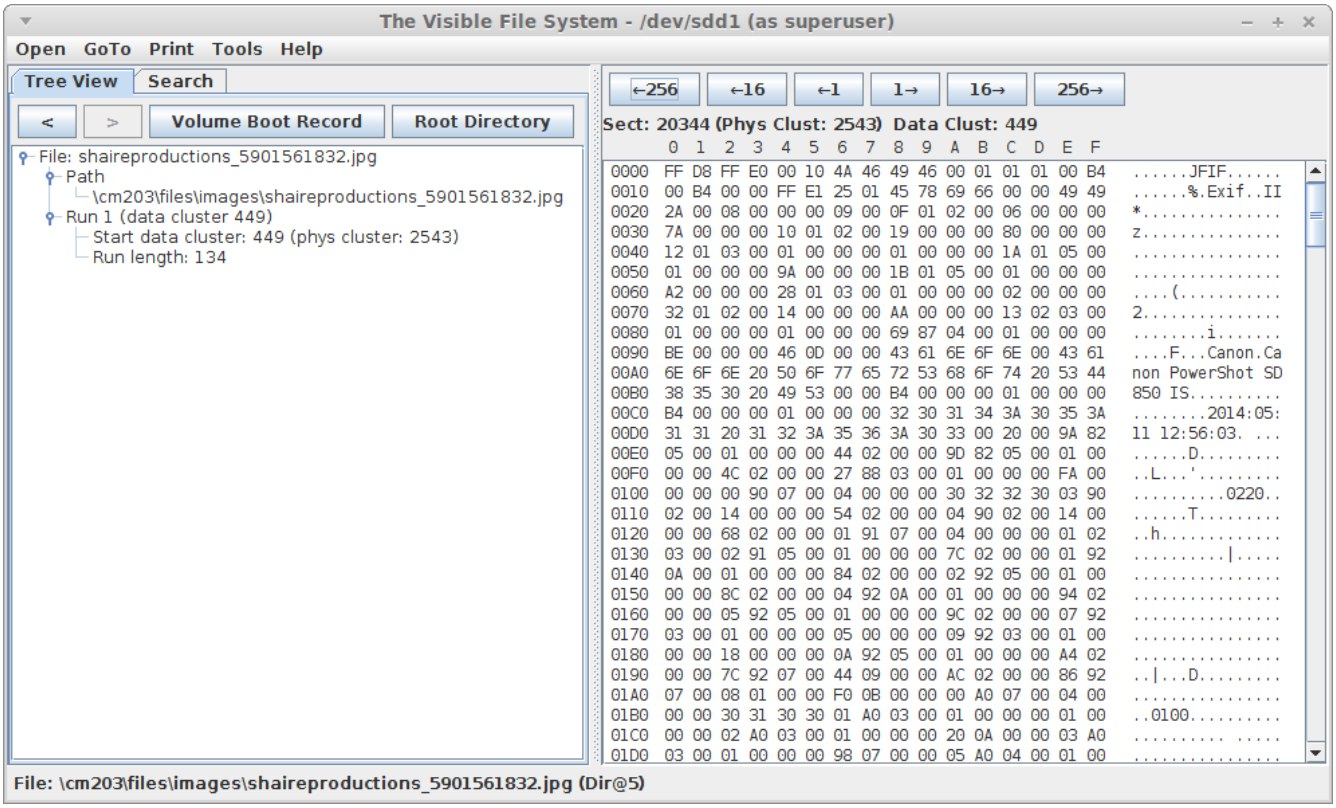
Below is a screenshot of the directory \cm203\files\images.



The long file name has three segments. They are numbered 01, 02 and 43 going upwards from the standard DOS directory entry. The 4 is used to signal the end of the sequence. Skipping over the reserved bytes (see above) we see the filename “shaireproductions_5901561832.jpg”. Notice the size

of the file is 545639 bytes (requires 1072 sectors or 134 data clusters).

Double-clicking on that file we go to a page that shows the data cluster runs and the actual file contents as seen in the screenshot below:



The file starts on data cluster 449 and has a run length of 134 data clusters. There is only one run, so the file is stored contiguously. The last data cluster is 449 + 134 -1 = 582 (try to find it in the FAT!).

Unit 10.5 – The File Allocation Table (FAT)

We use the FAT to find the allocated directory or file blocks. On modern drives there are two FATs. Their locations are given in the VBR. The second is a backup copy of the first. I suspect that the two FATs are used to provide robustness in case the power goes out during a disk write. The first FAT is updated first and then the second. If the power goes out while updating the first FAT then the second FAT is copied over the first one. Any of the interrupted data is lost. If the power goes out during the update of the second FAT, then the first FAT is copied over the second one. In this case the interrupted data is not lost.

FAT entries are 4 bytes long (32 bits, thus the 32 in FAT32). Each FAT entry represents a data cluster on the partition. Thus there must be as many FAT entries as there are data clusters on the partition. Thus the size of the FAT in bytes is 4*number of data clusters. Divide by 512 and round up to get the

size of the FAT in sectors.

Each FAT entry points to the next FAT entry for the directory or file. We call this the FAT chain. When you reach the end of the directory or file you will find 0x0FFFFFFF or 0x0FFFFFF8 (converted to little-endian). These values mark the end of the chain and the last cluster of the file or directory.

Entries 0 and 1 are not used. The root directory is always at entry 2.

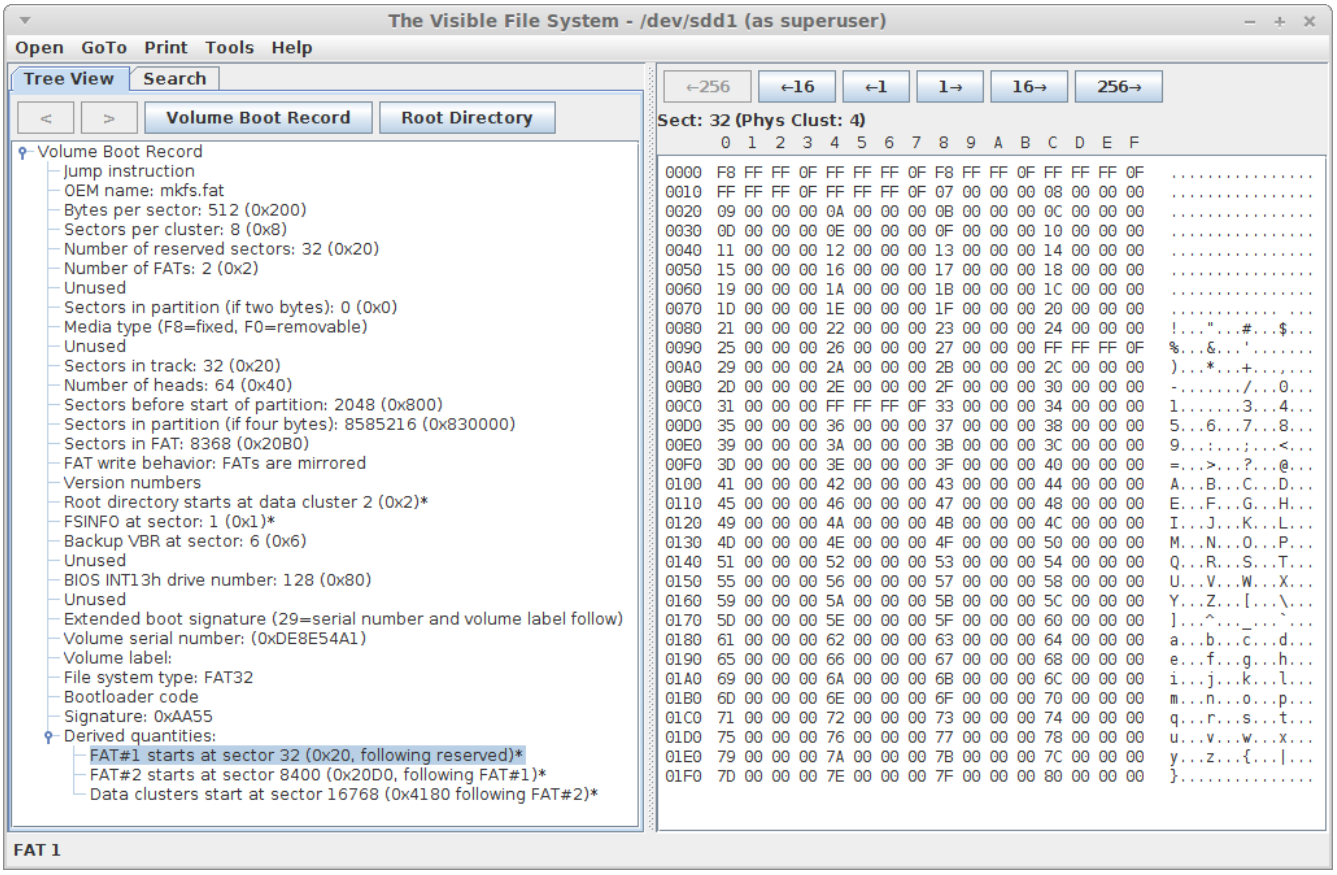
Unallocated data clusters are marked with all zeros.

Let's consider a simple example first. For simplicity I used FF to indicate the end of the chain. Suppose a file is stored in data clusters 3, 4, 5, A, B, C, 11, 12, and 1A in that order. The directory entry would point to data cluster 3. The entry at location 3 tells you to go to location 4 for the next cluster. The entry at 4 tells you to go to 5. The entry at 5 tells you to go to A. The entry at A points to B, B points to C and C points to 11. 11 points to 12, and 12 points to 1A. The entry at 1A indicates the end of the chain, which is the last block in the file.

Here's a sketch of what this FAT would look like

0 FF	1 FF	2 FF	3 4
4 5	5 A	6 0	7 0
8 0	9 0	A B	B C
C 11	D 0	E 0	F 0
10 0	11 12	12 1A	13 0
14 0	15 0	16 0	17 0
18 0	19 0	1A FF	1B 0

Below is the first sector of the actual FAT from the same file system as the directory in the previous section. There are a total of 8368 sectors in the FAT (see the VBR). This means there are a maximum of 8368*128 = 1071104 data clusters (the FAT doesn't have to fill the last sector). When the partition is first created the OS will mark the number of free clusters in the FSINFO block and that number gets decremented for each cluster that is used (and incremented for each one that is freed up).



You can see that data clusters 2, 3, 4 and 5 have one data cluster each (they are small directories). Data cluster 6 continues for a total of 34 contiguous data clusters (see the sequence 7, 8, 9, A, B, etc. from 0x6=6 to 0x27=39).

Here's a formula to help you figure out which sector contains the next FAT entry.

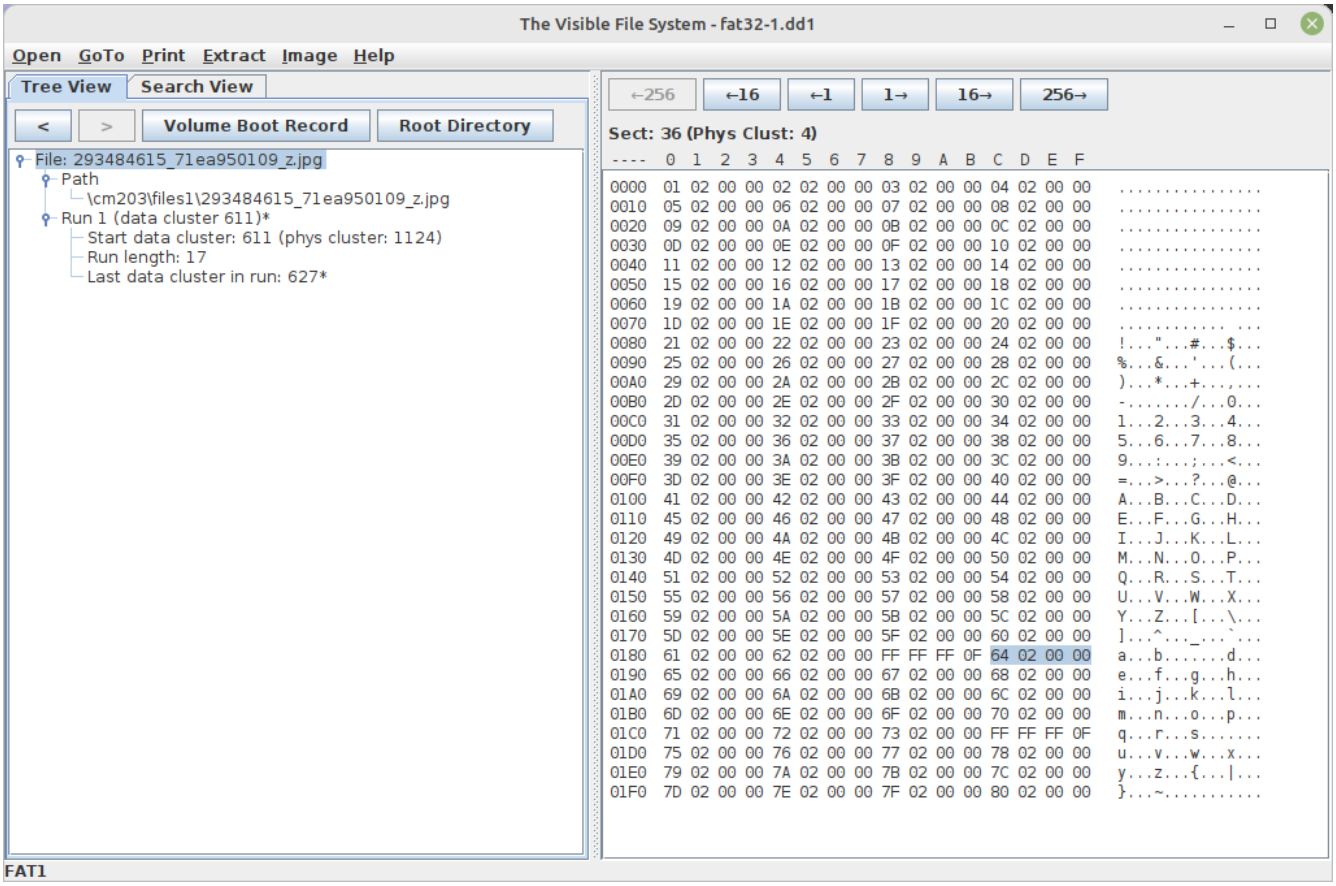
n = clust/128 (throw away fraction) (since there are 128 entries per sector)
sect = n + 32 (since first sector is #32)
offset = (clust – n*128) * 4 (to determine where entry is relative to start of sector)

For example, let's say we want to find data cluster 51804 in the FAT. Divide the data cluster number by 128 to get n=404. Add 32 to get the sector because the FAT starts at sector 32. You get sector 436. Go there. The offset from the start of that sector is determined as follows. Subtract 404*128=51712 from 51804 and you'll get 92. Multiply by 4 to get 368. Convert to hex to get 0x170. That's the offset to the FAT entry.

Unfortunately when a file is deleted the OS will go though the FAT and set all the entries to zeros. This is necessary to mark them as unallocated data clusters, but it erases the FAT entry chain that tells us where the file blocks are. However, the deleted directory entry will still point to the first data cluster,

and, with any luck, the file clusters may be contiguous.

The VisibleFS program allows you to go to any particular entry in the FAT. Just click on “GoTo” in the menu and choose “Go to FAT Entry…” and type in the data cluster that you want to see. The screenshot below shows the entry for data cluster 611 (highlighted on the right) which is the start of a JPG image file (see the directory entry on the left). Note the FFFFFF0F just before it showing the end of the FAT chain for the previous file.



Unit 10.6 – The Big Picture

And that's it. You have seen everything there is to see in FAT32. But it probably helps to put the pieces together with some examples.

The OS starts by reading the VBR and finding out where the root directory is. It reads through the root directory looking for the subdirectory or file the user is looking for.

If it is looking for a subdirectory, it starts at its first data cluster and looks through it for the next subdirectory or file. If it doesn't find it in the first data cluster, it follows the FAT to the next one. Eventually it will find what it's looking for or reach the end of the FAT chain and issue an error.

When it finally gets to a file, it will follow the FAT chain to find out which data clusters to load to get that file in RAM. And that's it.

In the simple file system shown here all files were stored contiguously. This is because they were added to an empty drive, in order and never modified. Changes made over time often result in fragmented files because they exceed the space originally allocated to them. If a file never changes after it is first saved, on a drive with lots of empty space, it will probably be contiguous. Given that modern drives are so big, there's a good chance that files will be contiguous.