CS342 Operating Systems – Spring 2022
Project #4 – Exploring a FAT32 File System Image

Assigned: April 18, 2022.
Due date: May 13, 2022, 23:59. Document version: 1.2

*This project can be done in groups of two students. You can do it individually as well. You will use the C programming language. You will develop your project in Linux. As always, the project is **for learning** and **developing skills**.*

Objectives: To learn and practice with file systems, FAT32 file system, formatting, mounting, block-storage device, loop device, sectors, blocks, clusters, directories, directory entries, root directory, file attributes, file operations, directory operations, pathnames, disk space allocation, linked allocation, file allocation table, free space management, boot sector, volume control information, superblock, doing experiments, collecting and interpreting experimental data.

Part A (80 pts):

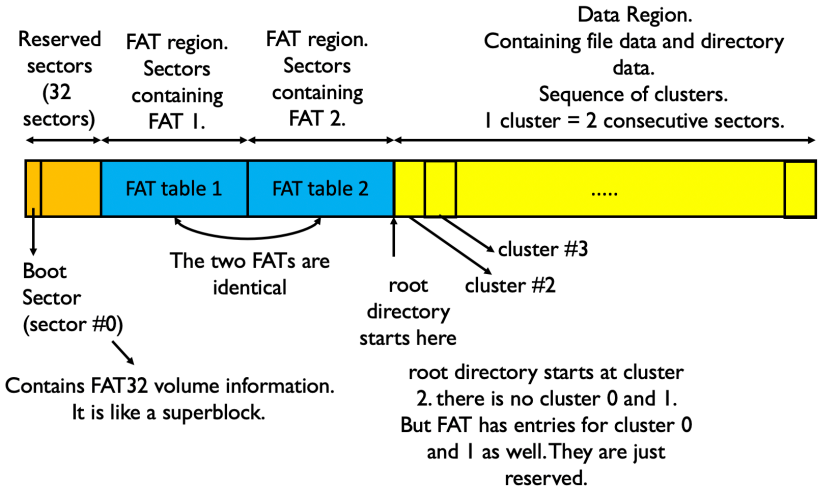
In this project you will write a program that will explore a **FAT32 disk image** (a FAT32 **volume**) and print out some information extracted from it. The disk image will be a regular Linux file. The Linux file will act as a disk **formatted** with FAT32 file system. Your program will open the Linux file with the usual `open()` system call and will access it directly, by utilizing the `read()` system call. Your program will read and interpret the disk image in **raw** mode without mounting the file system in it.

This project is about **FAT32** file system, not VFAT or exFAT. FAT32 file system is still popular and frequently used. It is a file system that is compatible with a huge variety of devices and operating systems. There is a lot of information available in Internet about FAT32 [1,2,3,4]. The reference [1] is the formal specification of FAT32 file system, including all the details about its on-disk data structures. You can consult that for some details like, for example, which bits of the date field in a directory entry are storing the day, month, and year information.

Understanding FAT32

FAT32 specification is using the terms **sectors** and **clusters**. A FAT32 file system is considering a disk as a sequence of logical sectors (**blocks**), the first sector being sector 0. The next sector is sector 1, and so on. Hence, each sector is addressed with a number (also called logical block address - **LBA**). The sector size is usually 512 bytes. The FAT32 file system accesses the disk in sectors. But, it allocates disk space to files in multiples of contiguous sectors, called clusters. A **cluster** is a sequence of ***N* contiguous sectors** on the disk. *N* is a power of 2 and is fixed for a particular FAT32 file system installed on a disk (specified at file system creation time). Hence, all clusters on a disk have

the same size, and a cluster is the **unit** of disk space allocation for files and directories. In this project, the cluster size will be **2 sectors** (i.e., 1024 bytes) (*N*=2). FAT32 allocates clusters to a file. Each cluster has a cluster number. Below, we see the typical **layout** of a FAT32 file system on a disk.



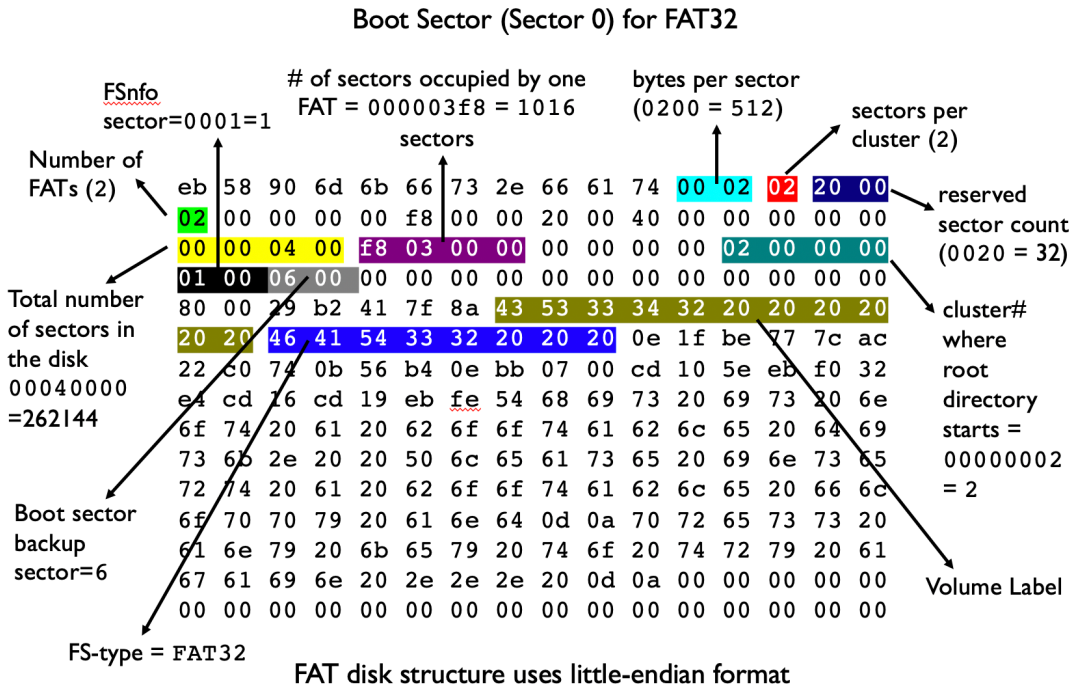
The first 32 sectors (0...31) are reserved. **Sector 0** has boot and volume information. It is called the **boot sector**. But since we will not use a bootable disk, there will be no boot code in this sector. This sector will have some important file system information.

After the **32 reserved sectors**, immediately comes two **FAT tables** (file allocation tables). FAT table 1 starts at sector #32. FAT table 2 starts immediately after FAT table 1. We have two FAT tables for reliability reasons (if one is corrupted, we can recover from the other one). Both FATs have the identical information (kept consistent). The size of a FAT table is a multiple of sector size. After two FAT tables, immediately comes the **Data Region**. The data region is considered as a sequence of **clusters**. The data region starts with cluster #2. Then, we have cluster 3, cluster 4, and so on. We do not have clusters 0 and 1 (these clusters are undefined). But we have the respective entries in the FAT table (entry 0 and 1). A file is allocated zero or more clusters (an empty file has zero clusters allocated). The numbers of the clusters allocated to a file are included in the FAT table as a **chain**. A FAT table has one **entry** per cluster. The root directory starts at cluster 2. It can occupy one or more clusters. The clusters allocated to a file or directory do **not** have to be **contiguous**.

The following figure shows a sample content (in hexadecimal) for **the sector #0**, boot sector, that is relevant and applicable for this project. To give such examples in this assignment, we prepared a FAT32 disk image file called `disk1` (a FAT32 volume containing some files and directories) that you can download from [6]. It is a regular Linux file that will act as a formatted disk.

The figure is showing a portion of the boot sector of `disk1`. The sector starts with offset 0. At that byte we have the value `0xeb`. At offset 11, we have a two-bytes value, which is indicating the **sector size** in bytes. We see it in

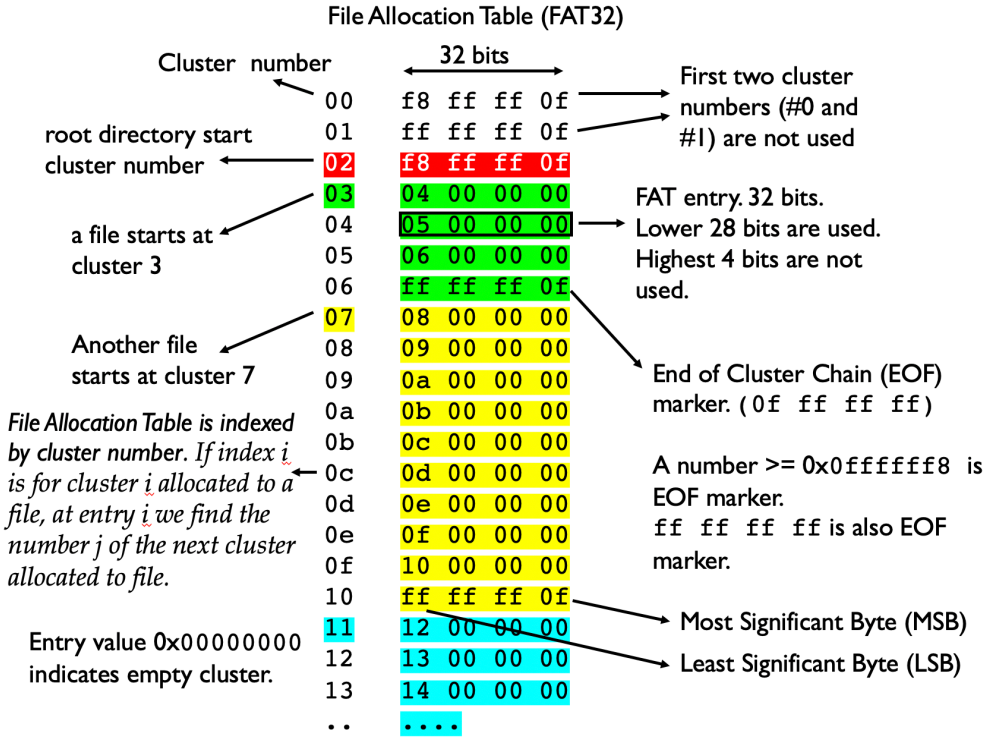
little-endian form in the picture. In FAT32, the multi-byte integer values (2 byte or 4 bytes) are stored in the disk in little-endian form. That means at *lowest* address (offset), we have the *least significant byte* of an integer. In big-endian format, we have the most significant byte at the lowest address (offset). The **x86-64** architecture is little-endian. In the figure we see the two-byte content starting at offset 11 as `00 02`. `00` is the least significant byte value and `02` is the most significant byte value. Therefore, we should read it as `0x0200`. It is 512 in decimal. Hence, the sector size is 512 bytes.



At offset 13, we have the **number of sectors per cluster** information (shown in red). It is `0x02`. That means we have 2 sectors per cluster. Reserved sector count is `0x0020`, that means 32 sectors (at offset 14). At offset 16, we see the byte (green) giving the **number FAT tables** in the volume. It is 2. Then at offset 32, we have a 4-byte integer value (shown in yellow color), which gives the **total number of sectors** in the disk. In the example, the byte sequence is `00 00 04 00` (little-endian), and the respective integer value is `0x00040000`, which makes 262144 in decimal. That means there are 262144 sectors in the disk. FAT32 file system considers the disk as a sequence of sectors (blocks), numbered as 0, 1, 2, ...3, 262143, in this example. At offset 44, we have the **cluster number where the root directory** starts. It is a 4-byte value. In this example, it is `0x00000002`, that means 2 in decimal. These are the most important information that you should know about a FAT32 volume that can be obtained from the first sector of a disk.

A **FAT table** contains information about the allocated and free clusters. The FAT table has one entry per cluster. The size of each entry is **4 bytes** (32 bits). A FAT table occupies a number of disk sectors (`sectors_per_FAT`). Then, the number of clusters that the file system can manage can be found as follows: `cluster_count` = (`sectors_per_FAT` * 512 / 4). Each **FAT entry** stores the number of the **next cluster** allocated to a file. A file starts at some

cluster. The number of that first cluster is stored in the directory entry for the file. That cluster number is used as an index into the FAT table to find the information about the next cluster allocated to the file. If at that index, we see a valid cluster number, it is the next cluster allocated to the file. Then we use that cluster number as an index into the FAT table again. There we will see the number of the next cluster allocated to the file, and so on. That means FAT32 is using the **linked allocation scheme,** but the pointers (cluster numbers) are not stored in data blocks. Instead, they are stored in the FAT table. We have a **chain** of cluster numbers for a non-empty file. A FAT table entry that contains all 0s (**0x00000000**) indicates that the corresponding cluster is **free (unused)**. A FAT table entry that is **equal or bigger than 0x0FFFFFF8** indicates the end of a cluster chain (end of file) (**EOC** or **EOF**), for a file or directory. The value **0x0FFFFFF7** is **bad cluster** mark. A file of size zero has the value 0 as the first cluster number in its directory entry. Below we see a portion of the FAT table of disk1.



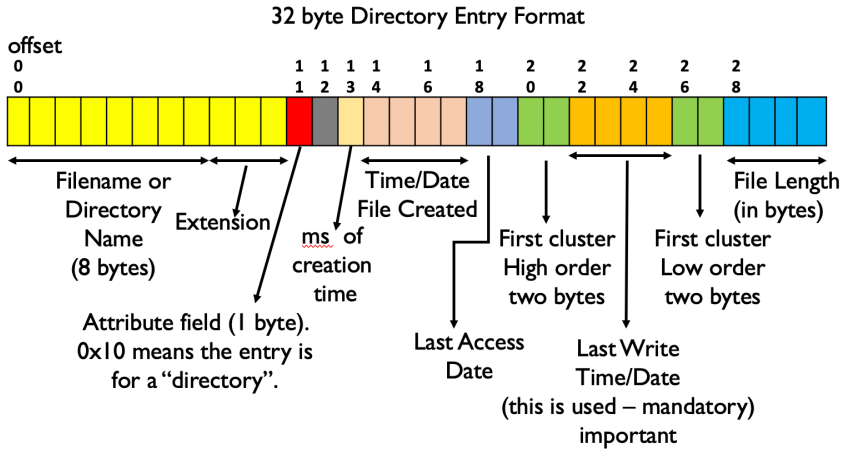
As we can see, the first 2 entries (entry 0 and 1) of the table have special values. The corresponding clusters do not exist on the disk. **Data region** starts with cluster 2. And at that cluster we have the root directory. **Root directory** can occupy one or more clusters (depending on the number of entries it has). In this example, it occupies just one cluster. Hence at index 2 of the FAT, the entry contains the special value `0x0ffffff8`, which indicates the end of file (the end of cluster chain) representing the root directory. We have a file (green) starting at cluster 3. At entry 3 of the FAT table, we have the number of the next cluster, which is 4, allocated to the green file. At entry 4, we have the address of the next cluster allocated, and so on. Hence, clusters 3, 4, 5, and 6 are allocated to the green file. At index 6 of the FAT table, we see an **end of file marker (EOF)**. Yellow file starts at cluster 7 and occupies the clusters 7, 8, 9, 10, 11, 12, 13, 14, 15, and 16 (in this order). It occupies 10

clusters. The first cluster number, in this case 7, is included in the directory entry for the file. The allocated clusters to a file do not have to be contiguous, contrary to this example.

Even though a FAT32 table entry is 32 bits long, only the **first 28 bits** are used to indicate a cluster number. The highest order 4 bits are not used. Therefore we can have at most 2^{28} clusters in a FAT32 disk. The maximum size of a disk that FAT32 can manage depends also on the clustersize (number of bytes in a cluster). If cluster size is 1024 (2 sectors), then the disk size can be at most 2^{28} x 2^{10} = 2^{38} = 256 GB. If clustersize is 32 KB, then the disk size could be 2^{28} x 2^{15} = 2^{43} bytes = 8 TB, but there is another factor that affects the maximum disk size. It is the sector count field in the boot sector, which is giving the number of sectors in the disk. That field is 32 bits long. Therefore, the **maximum disk size** that can be supported by FAT32 is 2^{32} x 512 = 2^{41} = **2 TB**, assuming sector size is 512 bytes.

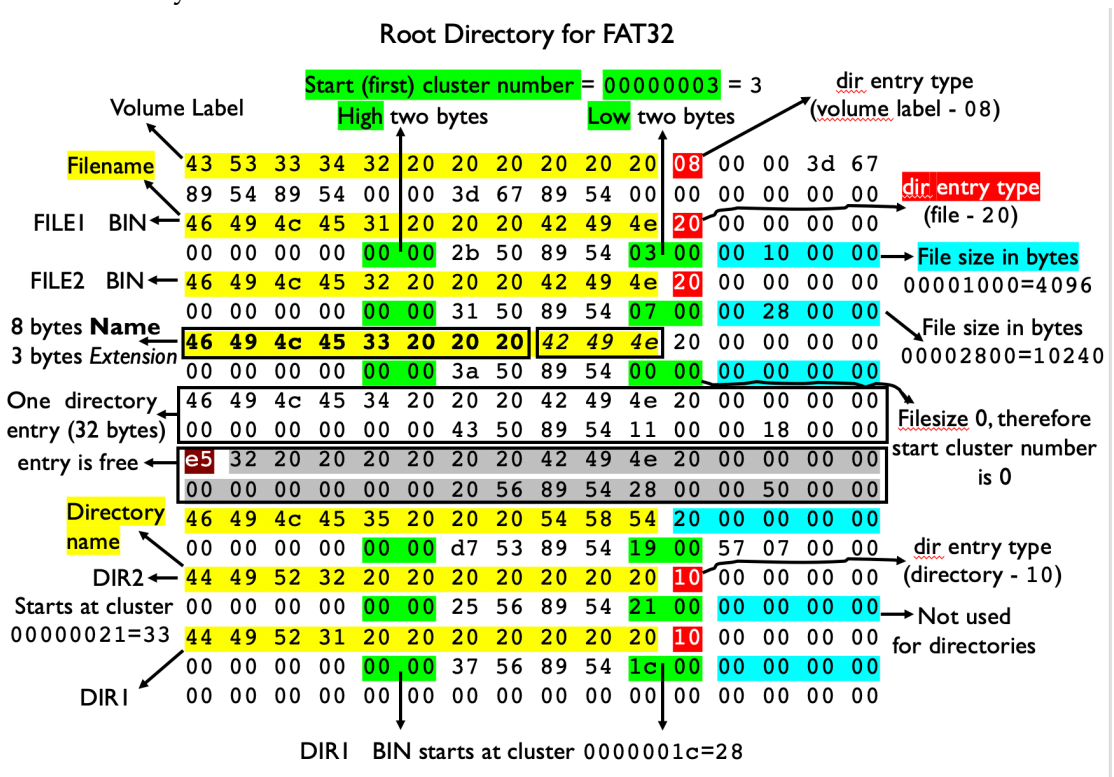
The **maximum file size**, however, can not be that big. It is limited with the filesize field (in bytes) in the directory entry for a file. The filesize field is 32 bits long. Therefore, the maximum file size can be **4 GB**.

A **directory** (for example, the root directory) is also like a file. It can occupy one or more clusters. The content of those clusters is a sequence of directory entries. A **directory entry** contains important information about a file, like the name of the file, the size of the file, the first cluster number, etc. Figure below shows the structure of a FAT32 directory entry. FAT32 uses fixed size directory entries. The size of each directory entry is 32 bytes.



The figure below shows a sample content of a directory, in this case the root directory of disk1. It is a sequence of **32-bytes** entries (directory records). The first entry is storing the **volume label**. In this example it is "`CS342`". It represents the root directory itself. Then, the next 32 bytes (next entry) is for a file with name `FILE1.BIN`. In a directory entry, the first 11 bytes (offset 0...10) store the name (8 bytes) and the extension (3 bytes) of a file (or subdirectory). Then at offset 11 we see a 1-byte **attributes** information. The bits of that byte indicate the type of the file. If the byte value is **`0x10`**, it means the entry is for a *directory* (i.e., sub-directory). If the byte value is `0x08`, then

the entry is for volume ID, i.e., for root directory (not a regular file entry). For a *regular file*, the least significant 3 bits of the attribute byte indicates the type of the file (like hidden, system, etc.). For a regular file all these 3 bits may be zero, or some of them may be 1. The least significant 4 bits of the attributes field should not be all 1s in this project. All 1s for these bits indicate the use of **long** filenames. We will not use long file names in this project (for simplicity). If the attribute value is `0x20`, that means the file is to be archived. A backup program will clear the bit after taking the backup of the file. The 2-bytes (**High** bytes) at offset 20 and the 2-bytes (**Low** bytes) at offset 26 indicate together the **first cluster number** of a file or directory (i.e., at which cluster the file data starts). For example, for the file `FILE1.BIN`, the High two bytes are `00 00` and the Low two bytes are `03 00`. When we combine them, it makes the number `0x00000003`, which is 3 in decimal. That means `FILE1.BIN` starts at cluster 3. The 4 bytes at offset 28 indicate the **size of the file** in bytes. For example, for file `FILE1.BIN`, the size is `0x00001000`, that means 4096 bytes. A directory entry that starts with byte value **0xe5** is empty, i.e., is available (was used earlier and then deleted). It can be used for a new file that is created. An entry that starts with byte value **0x00** is also an unused entry.



Creating and Using a FAT32 Disk Image

In this project, the FAT32 file system will not sit on a separate physical media, but it will sit on a regular Linux file. Therefore, we need to create a **regular Linux binary file** that will act as our disk, i.e., a virtual disk. We can do this with Linux **`dd`** command as follows. The `dd` command will create a binary Linux file and initialize it to all zeros. We specify the block size as 512 bytes.

```
dd if=/dev/zero of=disk1 bs=512 count=256K
```

In this example, we create a non-empty file of size 256K sectors. A FAT32 volume should have at least 65536 clusters. Since the cluster size is 2 sectors in this project, a file of size 256K sectors will include more than 65536 clusters. The file that will act as our disk in this example is named as disk1.

We can use the **`losetup`** command to associate a Linux **loop** device file (for example, `/dev/loop0`) with a Linux file. In this way, the Linux file (in this example the file disk1) will look like a **block storage** device [5]. To find the first available loop device in a Linux system, we type:

```
losetup -f
```

It will give the first available (unused) loop device, for example, `/dev/loop30`. Then we can associate that loop device, which is **block-oriented**, with our disk file as follows.

```
sudo losetup /dev/loop30 disk1
```

(To delete the association between a loop device file and a Linux file, we can use the `–d` option of `losetup` command as "`loosetup –d /dev/loop30`".)

After setting up the association with a loop device, our Linux file `disk1` will be treated as a *block-oriented* storage device. It can be formatted with a file system and then be mounted with the usual `mount` command.

We will format our disk with FAT32 file system. For that we can use the **`mkfs.fat`** command of Linux. We should execute the command with the options given in the example below. In our tests, we will use the same set of options. Therefore, if you are formatting a virtual disk (a Linux file generated as shown above) in your own test cases, you need to format it with same options shown below. You can get more information about `mkfs.fat` using its manual page (`man mkfs.fat`).

```
mkfs.fat –a –v –S 512 –s 2 –F 32 –n CS342 disk1
```

It can print out an output as below.

```
mkfs.fat 4.1 (2017-01-24)
disk1 has 64 heads and 32 sectors per track,
hidden sectors 0x0000;
logical sector size is 512,
using 0xf8 media descriptor, with 262144 sectors;
drive number 0x80;
filesystem has 2 32-bit FATs and 2 sectors per cluster.
FAT size is 1016 sectors, and provides 130040 clusters.
There are 32 reserved sectors.
Volume ID is 8a7f41b2, volume label CS342
```

In the command above, the `-F` option says that the file system will be **FAT32**, the `-s 2` option says that the cluster size will be 2 sectors, that means 1KB, `-S 512` option says that a logical sector is 512 bytes long, and the `-n` option gives a name for the volume. After execution of this command, we now have a FAT32 volume, i.e., a FAT32 file system installed to our disk. In other words, the file disk1 is a FAT32 disk image.

Now, we can **mount** this volume and use it as we are using any mounted file system. To mount the volume, we first need to create a *mount point* (i.e., an empty subdirectory) in our Linux home directory (or in any directory that you want). Assume the name of that empty directory is `dirfat`. We type the following to create the directory `dirfat`.

```
mkdir dirfat
```

We can now mount disk1 (using the associated loop device file) by executing the **`mount`** command with the following options. You should use the same set of options properly adjusted to your environment and username.

```
sudo mount -t msdos /dev/loop30 dirfat -o uid=korpe -o gid=korpe
```

Now the file system is mounted. We can change into its **root** directory. Its root directory is mounted at directory `dirfat`. Therefore, we can change to directory `dirfat`. When we are in the directory `dirfat`, that means we are in the root directory "`/`" of the mounted file system. There we can create and use files with any tools that we like: `touch` command, `dd` command, `cp` command, an editor (like `emacs`, `vi`, `pico`, etc.), or something else. We can also create subdirectories. In this example, we already created some files and directories in the mounted file system of `disk1`. For example, if we list the content of the root `(/)` directory (mounted on `dirfat`) we get the following output.

```
$ ls -al
drwxr-xr-x 4 korpe korpe  1024 Jan  1  1970 .
drwxr-xr-x 6 korpe korpe  4096 Apr 16 21:11 ..
drwxr-xr-x 3 korpe korpe  1024 Apr 10 16:17 dir1
drwxr-xr-x 2 korpe korpe  1024 Apr  9 13:49 dir2
-rwxr-xr-x 1 korpe korpe  4096 Apr  9 13:01 file1.bin
-rwxr-xr-x 1 korpe korpe 10240 Apr  9 13:01 file2.bin
-rwxr-xr-x 1 korpe korpe     0 Apr  9 13:01 file3.bin
-rwxr-xr-x 1 korpe korpe  6144 Apr  9 13:02 file4.bin
-rwxr-xr-x 1 korpe korpe  1917 Apr 16 21:11 file5.txt
```

After we create some additional files, or after doing some modifications to the file system, like creating a subdirectory, creating or deleting a file, or editing a file, we can execute the **`sync`** command, so that the cached modifications are reflected to the disk. For that we simply type `sync` at the command line. Then, we can unmount the file system. To **unmount**, we type the following command in the parent directory of the mount point.

```
sudo umount disk1
```

The Program Specification

Your program will **read** such a **FAT32 disk image** (like the disk1 in this example) and will extract and print out some information that are specified below. Your program will access the disk file in sectors or in clusters. The name of the program that you will write will be `fat.c` and the respective executable file will be called as `fat`. You can create and use a header file `fat.h`, if you wish. Your program will read and analyze the disk file directly (it will not mount the disk file internally). That means your program will access the disk image file in **raw mode** without mounting.

In your program, you can assume that the directory information (sequence of directory entries) for a directory (i.e., a subdirectory) will **fit** into a single cluster. You can also assume that only **short filenames** will be used. A short filename has at most 8 characters for the filename part and 3 characters for the extension part. "`project.pdf`", for example, is a short filename. We will not use long filenames.

We next describe the **operations** that your program will perform on a FAT32 disk image (all operations are *read-only*). An invocation of your program with a particular option and parameters will perform a specific operation and will print out the related information. Your program will support the options and operations listed and described below. Each operation listed below is performed by a separate invocation of your program with related parameters. Each invocation must have the `DISKIMAGE` parameter, specifying the FAT32 disk image file to use.

1. **`fat DISKIMAGE -v`**: print some summary information about the specified FAT32 volume `DISKIMAGE`. Most of the information is obtained from the boot sector (sector #0). An example output is given below (use the same format and parameters). Command: `./fat disk1 -v`

```
File system type: FAT32
Volume label: CS342
Number of sectors in disk: 262144
Sector size in bytes: 512
Number of reserved sectors: 32
Number of sectors per FAT table: 1016
Number of FAT tables: 2
Number of sectors per cluster: 2
Number of clusters = 130048
Data region starts at sector: 2064
Root directory starts at sector: 2064
Root directory starts at cluster: 2
Disk size in bytes: 134217728 bytes
Disk size in Megabytes: 128 MB
Number of used clusters: 142
Number of free clusters: 129906
```

2. **`fat DISKIMAGE -s SECTORNUM`**: print the content (byte sequence) of the specified sector to screen in hex form. An example output is shown below. Use the same format. Each line of output will have a sequence of 16 bytes of the sector printed out in hex form. First, in a line, the offset of the first byte in the sequence is printed in hex form (using 8 hex digits). Then, the sequence of 16 bytes are printed out in hex form (between 2 hex digits we have a SPACE character). Then the same sequence of 16 bytes are also printed out with *printable* characters if possible. For that you will use the `isprint()` function. If for a character `isprint()` returns true, then you will print the character using %c format specifier in `printf()`. If the character (byte) is not printable, i.e., `isprint()` returns false, then you print the character '.' (dot) instead of the byte, again using the %c format specifier in `printf`(). Command: `./fat disk1 -s 0`

```
00000000: eb 58 90 6d 6b 66 73 2e 66 61 74 00 02 02 20 00     .X.mkfs.fat... .
00000010: 02 00 00 00 00 f8 00 00 20 00 40 00 00 00 00 00     ........ .@.....
00000020: 00 00 04 00 f8 03 00 00 00 00 00 00 02 00 00 00     ................
00000030: 01 00 06 00 00 00 00 00 00 00 00 00 00 00 00 00     ................
00000040: 80 00 29 b2 41 7f 8a 43 53 33 34 32 20 20 20 20     ..).A..CS342
00000050: 20 20 46 41 54 33 32 20 20 20 0e 1f be 77 7c ac       FAT32   ...w|.
00000060: 22 c0 74 0b 56 b4 0e bb 07 00 cd 10 5e eb f0 32     ".t.V.......^..2
00000070: e4 cd 16 cd 19 eb fe 54 68 69 73 20 69 73 20 6e     .......This is n
00000080: 6f 74 20 61 20 62 6f 6f 74 61 62 6c 65 20 64 69     ot a bootable di
00000090: 73 6b 2e 20 20 50 6c 65 61 73 65 20 69 6e 73 65     sk.  Please inse
000000a0: 72 74 20 61 20 62 6f 6f 74 61 62 6c 65 20 66 6c     rt a bootable fl
000000b0: 6f 70 70 79 20 61 6e 64 0d 0a 70 72 65 73 73 20     oppy and..press
000000c0: 61 6e 79 20 6b 65 79 20 74 6f 20 74 72 79 20 61     any key to try a
000000d0: 67 61 69 6e 20 2e 2e 2e 20 0d 0a 00 00 00 00 00     gain ... .......
000000e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00     ................
000000f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00     ................
00000100: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00     ................
00000110: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00     ................
00000120: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00     ................
00000130: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00     ................
00000140: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00     ................
00000150: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00     ................
00000160: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00     ................
00000170: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00     ................
00000180: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00     ................
00000190: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00     ................
000001a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00     ................
000001b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00     ................
000001c0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00     ................
000001d0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00     ................
000001e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00     ................
000001f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 55 aa     ..............U.
```

3. `fat` **`DISKIMAGE -c CLUSTERNUM`**: print the content (byte sequence) of the specified cluster to the screen in hex form. Note that cluster 0 and cluster 1 do not exist. An example output (partially) is shown below (for cluster #2). Use the same format. Command: `./fat disk1 -c 2`

```
00000000: 43 53 33 34 32 20 20 20 20 20 20 08 00 00 3d 67     CS342      ...=g
00000010: 89 54 89 54 00 00 3d 67 89 54 00 00 00 00 00 00     .T.T..=g.T......
00000020: 46 49 4c 45 31 20 20 20 42 49 4e 20 00 00 00 00     FILE1   BIN ....
00000030: 00 00 00 00 00 00 2b 50 89 54 03 00 00 10 00 00     ......+P.T......
00000040: 46 49 4c 45 32 20 20 20 42 49 4e 20 00 00 00 00     FILE2   BIN ....
00000050: 00 00 00 00 00 00 31 50 89 54 07 00 00 28 00 00     ......1P.T...(..
00000060: 46 49 4c 45 33 20 20 20 42 49 4e 20 00 00 00 00     FILE3   BIN ....
00000070: 00 00 00 00 00 00 3a 50 89 54 00 00 00 00 00 00     ......:P.T......
00000080: 46 49 4c 45 34 20 20 20 42 49 4e 20 00 00 00 00     FILE4   BIN ....
00000090: 00 00 00 00 00 00 43 50 89 54 11 00 00 18 00 00     ......CP.T......
000000a0: e5 32 20 20 20 20 20 20 42 49 4e 20 00 00 00 00     .2      BIN ....
000000b0: 00 00 00 00 00 00 20 56 89 54 28 00 00 50 00 00     ...... V.T(..P..
000000c0: 46 49 4c 45 35 20 20 20 54 58 54 20 00 00 00 00     FILE5   TXT ....
000000d0: 00 00 00 00 00 00 73 91 90 54 99 00 7d 07 00 00     ......s..T..}...
```

```
000000e0: 44 49 52 32 20 20 20 20 20 20 20 10 00 00 00 00     DIR2       .....
000000f0: 00 00 00 00 00 00 25 56 89 54 21 00 00 00 00 00     ......%V.T!.....
00000100: 44 49 52 31 20 20 20 20 20 20 20 10 00 00 00 00     DIR1       .....
00000110: 00 00 00 00 00 00 23 6a 8a 54 1c 00 00 00 00 00     ......#j.T......
00000120: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00     ................
00000130: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00     ................
00000140: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00     ................
00000150: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00     ................
```

4. **`fat DISKIMAGE -t`**: print all directories and their files and subdirectories starting from the root directory, recursively, in a depth-first search order. That means you will traverse the whole directory tree and print the names of the files and directories encountered. An example output is shown below. Use the same format. Each line gives the full pathname of a file or directory. Whether the pathname is for a file or directory is indicated with (f) or (d) in the beginning of the line. Command: `./fat disk1 -t`

```
(f)  /FILE1.BIN
(f)  /FILE2.BIN
(f)  /FILE3.BIN
(f)  /FILE4.BIN
(f)  /FILE5.TXT
(d)  /DIR2
(d)  /DIR2/.
(d)  /DIR2/..
(f)  /DIR2/F2.BIN
(f)  /DIR2/F1.TXT
(d)  /DIR1
(d)  /DIR1/.
(d)  /DIR1/..
(f)  /DIR1/PROGRAM.C
(f)  /DIR1/AFILE1.BIN
(d)  /DIR1/DD1
(d)  /DIR1/DD1/.
(d)  /DIR1/DD1/..
(f)  /DIR1/DD1/PFILE.BIN
(f)  /DIR1/DD1/PF1.TXT
(f)  /DIR1/DD1/PFILE2.BIN
```

5. **`fat DISKIMAGE -a PATH`**: print the content of the ascii text file indicated with `PATH` to the screen as it is. You can assume the specified file contains printable ascii text characters. Where each line will end will be dictated by the newline characters that may exist in the specified file. Hence you will not worry about the line length. Just print the characters (by using the %c format specifier) as they appear in the file. An example output is below, which is the content of the file `/DIR2/F1.TXT`. We will invoke this option only for ascii text files. Command: `./fat disk1 -a /DIR2/F1.TXT`

```
NAME   losetup - set up and control loop devices
SYNOPSIS
       Get info:
           losetup [loopdev]
            losetup -l [-a]
            losetup -j file [-o offset]
       Detach a loop device:
            losetup -d loopdev...
       Detach all associated loop devices:
```

```
            losetup -D
       Set up a loop device:
            losetup [-o offset] [--sizelimit size] [--sector-size size]
                    [-Pr] [--show] -f|loopdev file
       Resize a loop device:
            losetup -c loopdev
DESCRIPTION
       losetup is used to associate loop devices with regular files or block
devices, to detach loop devices, and to query the
       status of a loop device.  If only the loopdev argument is given, the status
of the corresponding loop device is  shown.
       If no option is given, all loop devices are shown.
       Note  that  the old output format (i.e., losetup -a) with comma-delimited
strings is deprecated in favour of the --list
       output format.
       It's possible to create more independent loop devices for the same backing
file.  This  setup  may  be  dangerous,  can
       cause data loss, corruption and overwrites.  Use --nooverlap with --find
during setup to avoid this problem.
```

6. **`fat DISKIMAGE -b PATH`**: print the content (byte sequence) of the file indicated with `PATH` to the screen in hex form in the following format. The file can be a binary file or an ascii file, does not matter. An example output is shown below. Use the same format. It is the output for the file `/DIR2/F1.TXT` (a portion of it). Command: `./fat disk1 -b /DIR2/F1.TXT`

```
00000000: 4e 41 4d 45 20 20 20 6c 6f 73 65 74 75 70 20 2d     NAME   losetup -
00000010: 20 73 65 74 20 75 70 20 61 6e 64 20 63 6f 6e 74      set up and cont
00000020: 72 6f 6c 20 6c 6f 6f 70 20 64 65 76 69 63 65 73     rol loop devices
00000030: 0a 53 59 4e 4f 50 53 49 53 0a 20 20 20 20 20 20     .SYNOPSIS.
00000040: 20 47 65 74 20 69 6e 66 6f 3a 0a 20 20 20 20 20      Get info:.
00000050: 20 20 20 20 20 20 6c 6f 73 65 74 75 70 20 5b 6c           losetup [l
00000060: 6f 6f 70 64 65 76 5d 0a 20 20 20 20 20 20 20 20     oopdev].
00000070: 20 20 20 20 6c 6f 73 65 74 75 70 20 2d 6c 20 5b         losetup -l [
00000080: 2d 61 5d 0a 20 20 20 20 20 20 20 20 20 20 20 20     -a].
00000090: 6c 6f 73 65 74 75 70 20 2d 6a 20 66 69 6c 65 20     losetup -j file
000000a0: 5b 2d 6f 20 6f 66 66 73 65 74 5d 0a 20 20 20 20     [-o offset].
000000b0: 20 20 20 44 65 74 61 63 68 20 61 20 6c 6f 6f 70        Detach a loop
000000c0: 20 64 65 76 69 63 65 3a 0a 20 20 20 20 20 20 20      device:.
000000d0: 20 20 20 20 20 6c 6f 73 65 74 75 70 20 2d 64 20          losetup -d
000000e0: 6c 6f 6f 70 64 65 76 2e 2e 2e 0a 20 20 20 20 20     loopdev....
000000f0: 20 20 44 65 74 61 63 68 20 61 6c 6c 20 61 73 73       Detach all ass
00000100: 6f 63 69 61 74 65 64 20 6c 6f 6f 70 20 64 65 76     ociated loop dev
00000110: 69 63 65 73 3a 0a 20 20 20 20 20 20 20 20 20 20     ices:.
00000120: 20 20 6c 6f 73 65 74 75 70 20 2d 44 0a 20 20 20       losetup -D.
00000130: 20 20 20 20 53 65 74 20 75 70 20 61 20 6c 6f 6f         Set up a loo
00000140: 70 20 64 65 76 69 63 65 3a 0a 20 20 20 20 20 20     p device:.
00000150: 20 20 20 20 20 20 6c 6f 73 65 74 75 70 20 5b 2d           losetup [-
00000160: 6f 20 6f 66 66 73 65 74 5d 20 5b 2d 2d 73 69 7a     o offset] [--siz
00000170: 65 6c 69 6d 69 74 20 73 69 7a 65 5d 20 5b 2d 2d     elimit size] [--
00000180: 73 65 63 74 6f 72 2d 73 69 7a 65 20 73 69 7a 65     sector-size size
00000190: 5d 0a 20 20 20 20 20 20 20 20 20 20 20 20 20 20     ].
000001a0: 20 20 20 20 20 20 5b 2d 50 72 5d 20 5b 2d 2d 73           [-Pr] [--s
000001b0: 68 6f 77 5d 20 2d 66 7c 6c 6f 6f 70 64 65 76 20     how] -f|loopdev
000001c0: 66 69 6c 65 0a 20 20 20 20 20 20 20 52 65 73 69     file.       Resi
000001d0: 7a 65 20 61 20 6c 6f 6f 70 20 64 65 76 69 63 65     ze a loop device
000001e0: 3a 0a 20 20 20 20 20 20 20 20 20 20 20 20 6c 6f     :.            lo
000001f0: 73 65 74 75 70 20 2d 63 20 6c 6f 6f 70 64 65 76     setup -c loopdev
00000200: 0a 44 45 53 43 52 49 50 54 49 4f 4e 0a 20 20 20     .DESCRIPTION.
00000210: 20 20 20 20 6c 6f 73 65 74 75 70 20 69 73 20 75         losetup is u
00000220: 73 65 64 20 74 6f 20 61 73 73 6f 63 69 61 74 65     sed to associate
00000230: 20 6c 6f 6f 70 20 64 65 76 69 63 65 73 20 77 69      loop devices wi
00000240: 74 68 20 72 65 67 75 6c 61 72 20 66 69 6c 65 73     th regular files
00000250: 20 6f 72 20 62 6c 6f 63 6b 20 64 65 76 69 63 65      or block device
00000260: 73 2c 20 74 6f 20 64 65 74 61 63 68 20 6c 6f 6f     s, to detach loo
00000270: 70 20 64 65 76 69 63 65 73 2c 20 61 6e 64 20 74     p devices, and t
```

7. **`fat DISKIMAGE -l PATH`**: print the names of the files and subdirectories in the directory indicated with `PATH`. The pathname for the

root directory is "/". In each line of output, print the name of a file or directory (not pathname), its first cluster number, its size (in bytes), and its date-time information. You will get the date and time information from offset 22 (4 bytes) of a directory entry (or from the `date`, `time` field of the `msdos_dir_entry` structure explained later). An example output is shown below. Use the same format. Command: `./fat disk1 -l /DIR1`

```
(d)  name=.           fcn=28      size(bytes)=0         date=09-04-2022:10:38
(d)  name=..          fcn=0       size(bytes)=0         date=09-04-2022:10:38
(f)  name=PROGRAM.C   fcn=32      size(bytes)=102       date=09-04-2022:10:40
(f)  name=AFILE1.BIN  fcn=60      size(bytes)=51200     date=09-04-2022:10:48
(d)  name=DD1         fcn=110     size(bytes)=0         date=10-04-2022:13:18
```

8. **`fat DISKIMAGE -n PATH`**: print the numbers of the clusters storing the content of the file or directory indicated with `PATH`. If the size of the file is 0, nothing will be printed. An example output is shown below. Command: `./fat disk1 -n /FILE2.BIN`

```
cindex=0          clusternum=7
cindex=1          clusternum=8
cindex=2          clusternum=9
cindex=3          clusternum=10
cindex=4          clusternum=11
cindex=5          clusternum=12
cindex=6          clusternum=13
cindex=7          clusternum=14
cindex=8          clusternum=15
cindex=9          clusternum=16
```

9. **`fat DISKIMAGE -d PATH`**: print the content of the directory entry of the file or directory indicated with `PATH`. Some information from the directory entry will be printed out. An example output is shown below. Use the same format and parameters. Command: `./fat disk1 -d /FILE2.BIN`

```
name         = FILE2.BIN
type         = FILE
firstcluster = 7
clustercount = 10
size(bytes)  = 10240
date         = 09-04-2022
time         = 10:00
```

10. **`fat DISKIMAGE -f COUNT`**: print the content of the FAT table. The first `COUNT` entries will be printed out. Each line will include a cluster number (FAT index in decimal form) and the respective FAT entry content in decimal form. Entry index will start from 0 (that means cluster numbers will start from 0). If `COUNT` is `-1`, information about all entries will be

printed out. An example output is shown below (for the first 15 entries). Use the same format. Command: `./fat disk1 -f 15`

```
0000000: EOF
0000001: EOF
0000002: EOF
0000003: 4
0000004: 5
0000005: 6
0000006: EOF
0000007: 8
0000008: 9
0000009: 10
0000010: 11
0000011: 12
0000012: 13
0000013: 14
0000014: 15
```

11. **`fat DISKIMAGE -r PATH OFFSET COUNT`**: read `COUNT` bytes from the file indicated with `PATH` starting at `OFFSET` (byte offset into the file) and print the bytes read to the screen as shown in the example below, which shows the output for 200 bytes from the file starting at offset 14 of the file (the first byte of a file has offset 0). Use the same format. If `OFFSET + COUNT` is greater that the filesize, just read till the end of the file and print the respective content. Command: `./fat disk1 -r /FILE5.TXT 14 200`

```
0000000e: 4c 69 6e 75 78 20 66 69 6c 65 73 79 73 74 65 6d     Linux filesystem
0000001e: 20 74 79 70 65 73 3a 20 65 78 74 2c 20 65 78 74      types: ext, ext
0000002e: 32 2c 20 65 78 74 33 2c 20 65 78 74 34 2c 20 68     2, ext3, ext4, h
0000003e: 70 66 73 2c 20 69 73 6f 39 36 36 30 2c 20 4a 46     pfs, iso9660, JF
0000004e: 53 2c 20 6d 69 6e 69 78 2c 20 6d 73 64 6f 73 2c     S, minix, msdos,
0000005e: 20 6e 63 70 66 73 20 6e 66 73 2c 20 6e 74 66 73      ncpfs nfs, ntfs
0000006e: 2c 20 70 72 6f 63 2c 20 52 65 69 73 65 72 66 73     , proc, Reiserfs
0000007e: 2c 20 73 6d 62 2c 0a 20 20 20 20 20 20 20 73 79     , smb,.       sy
0000008e: 73 76 2c 20 75 6d 73 64 6f 73 2c 20 76 66 61 74     sv, umsdos, vfat
0000009e: 2c 20 58 46 53 2c 20 78 69 61 66 73 2c 0a 0a 44     , XFS, xiafs,..D
000000ae: 45 53 43 52 49 50 54 49 4f 4e 0a 20 20 20 20 20     ESCRIPTION.
000000be: 20 20 57 68 65 6e 2c 20 61 73 20 69 73 20 63 75       When, as is cu
000000ce: 73 74 6f 6d 61 72 79 2c                             stomary
```

12. **`fat DISKIMAGE -m COUNT`**: print a map of the volume. For each cluster, you need to print if that cluster is used or not, if used to which directory or file it belongs (full path will be printed out). The map of the first `COUNT` clusters will be printed out. If `COUNT` is `-1`, then information about all clusters will be printed out. Cluster numbers (FATtable indices) will be printed in decimal. An example output is shown below (for 50 clusters) . Use the same format. Command: `./fat disk1 -m 50`

```
0000000: --EOF--
0000001: --EOF--
0000002: /
0000003: /FILE1.BIN
0000004: /FILE1.BIN
0000005: /FILE1.BIN
0000006: /FILE1.BIN
```

```
0000007: /FILE2.BIN
0000008: /FILE2.BIN
0000009: /FILE2.BIN
0000010: /FILE2.BIN
0000011: /FILE2.BIN
0000012: /FILE2.BIN
0000013: /FILE2.BIN
0000014: /FILE2.BIN
0000015: /FILE2.BIN
0000016: /FILE2.BIN
0000017: /FILE4.BIN
0000018: /FILE4.BIN
0000019: /FILE4.BIN
0000020: /FILE4.BIN
0000021: /FILE4.BIN
0000022: /FILE4.BIN
0000023: --FREE--
0000024: --FREE--
0000025: /FILE5.TXT
0000026: /FILE5.TXT
0000027: --FREE--
0000028: /DIR1
0000029: --FREE--
0000030: --FREE--
0000031: --FREE--
0000032: /DIR1/PROGRAM.C
0000033: /DIR2
0000034: --FREE--
0000035: --FREE--
0000036: --FREE--
0000037: --FREE--
0000038: /DIR2/F1.TXT
0000039: /DIR2/F1.TXT
0000040: /DIR2/F2.BIN
0000041: /DIR2/F2.BIN
0000042: /DIR2/F2.BIN
0000043: /DIR2/F2.BIN
0000044: /DIR2/F2.BIN
0000045: /DIR2/F2.BIN
0000046: /DIR2/F2.BIN
0000047: /DIR2/F2.BIN
0000048: /DIR2/F2.BIN
0000049: /DIR2/F2.BIN
```

13. **`fat -h`**: print a help page showing all 12 options (operations) listed above and their respective parameters, if any.

Sample invocations of the program can be as follows:

- `./fat disk1 -v`
- `./fat disk1 -s 32`
- `./fat disk1 -c 2`
- `./fat disk1 -t`
- `./fat disk1 -r /DIR2/F1.TXT 100 64`
- `./fat disk1 -b /DIR2/F1.TXT`
- `./fat disk1 -a /DIR2/F1.TXT`
- `./fat disk1 -n /DIR1/AFILE1.BIN`
- `./fat disk1 -m 100`
- `./fat disk1 -f 50`
- `./fat disk1 -d /DIR1/AFILE1.BIN`
- `./fat disk1 -l /`
- `./fat disk1 -l /DIR2`
- `./fat disk1 -h`

As said earlier, you are provided an example FAT32 file system image, called disk1, in the webpage of the course [6]. You can start in your program with reading and parsing this file. As a cross-check, you can see the content of a binary file in hex form using the `xxd` Linux command.

```
xxd disk1 | more
```

Developing the Program

In Linux system, the header file `/usr/include/linux/msdos_fs.h` includes the definition of some of the **FAT32 structures**. For example, it includes the definition of the boot sector structure called `struct fat_boot_sector`. It also includes the definition of the directory entry structure called `struct msdos_dir_entry`, as shown below.

```
struct msdos_dir_entry {
    __u8     name[MSDOS_NAME];   /* name and extension */
    __u8     attr;       /* attribute bits */
    __u8     lcase;      /* Case for base and extension */
    __u8     ctime_cs;   /* Creation time, centiseconds (0-199) */
    __le16   ctime;      /* Creation time */
    __le16   cdate;      /* Creation date */
    __le16   adate;      /* Last access date */
    __le16   starthi;    /* High 16 bits of cluster in FAT32 */
    __le16   time,date,start;  /* time, date and first cluster */
    __le32   size;       /* file size (in bytes) */
};
```

The `fat_boot_sector` and `msdos_dir_entry` structure definitions can very useful for you to understand the related on-disk structures and also to parse the related on-disk byte sequences. For example, if `unsigned char sector[512]` is defined as a buffer (array) in your program and is currently holding the content of the sector 0 of the disk, then you can do the following **type casting** to easily access various information from the boot sector (byte array) using the fields of the `fat_boot_sector` structure; assuming that the machine *is little-endian*.

```
struct fat_boot_sector * bp;
unsigned char num_fats, sectors_per_cluster;
unsigned int num_sectors, sectors_per_fat;
unsigned int root_start_cluster;

bp = (struct fat_boot_sector *) sector; // type casting

// the following example shows how we can
// access the related data from the sector
sectors_per_cluster = bp->sec_per_clus;
num_sectors = bp->total_sect;
num_fats = bp->fats;
sectors_per_fat = bp->fat32.length;
root_start_cluster = bp->fat32.root_cluster;
```

Similarly, assume that you retrieved a cluster from the disk that is storing directory information for a directory (a sequence of directory entries) in the buffer (byte array) `cluster`. Then you can access the **directory entry** fields again easily by using type casting as below.

```
unsigned char cluster[1024];  // 1024 is clustersize.
struct msdos_dir_entry  *dep;

dep = (struct msdos_dir_entry *) cluster;
// to access related dir entry data in the cluster
// we can utilize the fields of dir entry structure
dep->name
dep->attr
dep->date;
dep->time;
dep->size;
```

You access the next directory entry in the cluster by simply incrementing the `dep` pointer.

```
dep++;
```

For **date** and **time** information for a file, you need to use the `time`, `date` fields of the `msdos_dir_entry` structure shown above (not `ctime`, `cdate`). To use `fat_boot_sector` and `msdos_dir_entry` structures, you need to include the `linux/msdos_fs.h` header file in your program. If you wish, you can define your own macros and structures instead of or in addition to the ones available in `msdos_fs.h` header file.

If you wish, you can also use `memcpy()` or `bcopy()` to copy `n` bytes from an `offset` (index) of the byte-array holding a disk cluster (`unsigned char cluster[1024]`) to a variable, as shown in the example below, assuming that the machine is *little-endian*.

```
unsigned int size;
bcopy ((void*) &size, (void*),
          (void *) (cluster + 28), sizeof(int));
```

This will copy 4 bytes (integer size) from the byte array `cluster` into an integer called `size`.

The FAT32 disk image needs to be accessed sector by sector. And the data region of the FAT32 volume needs to be accessed in clusters. Below is a code example about how to **read** and **write a sector** from the disk. Assume `data_start_sector` is the number of the sector where the data region starts. Since the boot sector gives us the FAT table length (in sectors), we can find `data_start_sector` as below:

```
data_start_sector = 32 + 2 * fat_length_in_sectors;
```

```
#define SECTORSIZE  512
#define CLUSTERSIZE 1024

int readsector (int fd, unsigned char *buf,
                unsigned int snum)
{
     off_t offset;
     int n;
     offset = snum * SECTORSIZE;
     lseek (fd, offset, SEEK_SET);
     n = read (fd, buf, SECTORSIZE);
     if (n == SECTORSIZE)
          return (0);
     else
          return(-1);
}


int writesector (int fd, unsigned char *buf,
                 unsigned int snum)
{
     off_t offset;
     int n;
     offset = snum * SECTORSIZE;
     lseek (fd, offset, SEEK_SET);
     n = write (fd, buf, SECTORSIZE);
     fsync (fd);  // write without delayed-writing
     if (n == SECTORSIZE)
          return (0); // success
     else
          return(-1);
}

int readcluster (int fd, unsigned char *buf,
                 unsigned int cnum)
{
     off_t offset;
     int n;
     unsigned int snum; // sector number
     snum = data_start_sector +
            (clusternum - 2)  * sectors_per_cluster;
     offset = snum * SECTORSIZE;
     lseek (fd, offset, SEEK_SET);
     n = read (fd, buf, CLUSTERSIZE);
     if (n == CLUSTERSIZE)
          return (0);   // success
     else
          return (-1);
}

int main ()
```

```
{
     unsigned char sector[SECTORSIZE];
     unsigned char cluster[CLUSTERSIZE];
     fd = open ("disk1",  O_SYNC | O_RDONLY);  // disk fd
     readsector (fd, sector, 0); // read sector #0
     readcluster (fd, cluster, 2); // read cluster #2
     // always check the return values
}
```

In this project, your program will only *read* a FAT32 disk image file. Therefore, only `readsector()` and `readcluster()` functions are to be used. You will not *write* anything to the disk image in your program. All operations are *read-only* operations, not requiring any write to disk to be performed.

Try to use **unsigned** integers and characters for various numbers stored in the boot sector and other FAT32 structures.

Part B – Experiments (20 pts):

Please do some timing experiments. Measure the time required to perform various operations with various parameter values. You can use the `time` command at the command line to measure the time elapsed to run your program with some particular parameters. Put your results into a `report.pdf` document. Try to interpret them and try to draw conclusions.

Submission:

Put all your files into a directory named with your Student ID (if done as a group, the ID of one of the students will be used). In a `README.txt` file, write your name, ID, etc. (if done as a group, all names and IDs will be included). Do not include executable file in your tar package. Include a `Makefile` so that we can just type make and compile your program. Then `tar` and `gzip` the directory. For example, a student with ID 21404312 will create a directory named "`21404312`" and will put the files there. Then he will tar the directory (package the directory) as follows:

```
tar cvf 21404312.tar 21404312
```

Then he will gzip the tar file as follows:

```
gzip 21404312.tar
```

In this way he will obtain a file called `21404312.tar.gz`. Then he will upload this file into Moodle.

Tips and Clarifications:

- Start early, work incrementally.
- In our tests, the **minimum** disk size will be 80 MB, and the **maximum** disk size will be 512 MB. 1 MB = 2^{20} bytes.

- The depth of the directory tree can be anything normally, but in our test cases it will never be more than 10.
- You can load the FAT table into memory if you wish. For that you need to use `malloc()`. Check the return value of the `malloc()` if it could allocate the space needed to hold the FAT table. If it can not allocate the needed memory, then you need to access the FAT table from disk in sectors.
- Since clustersize is 1024 bytes and directory entry size is 32 bytes, we can have at most 32 entries in a directory (we are assuming a directory will fit into a single cluster in this project). So, you can assume that in our tests we will never have more than 25 files or subdirectories in a directory.
- 1) if you want to access the disk1 in raw mode in your program fat.c (i.e., open it with open() system call in your program and start reading it with read() system - or readsector(), readcluster() functions given in the assignment), it is ready to use. it is formatted with FAT32, therefore has a file system created, and it has some files and directories.
  2) if you want to mount it and see what it is inside first, as any other mounted file system, you need to do the following (as described in the assignment):
  first execute
  `losetup -f`
  to find an available loop device in your Linux system.
  say it is `/dev/loop5`
  Then you need to associate the loop device file with disk1 file by typing the following at the command line:
  `sudo losetup /dev/loop5 disk1`
  Then, you need to mount it with the following command:
  `sudo mount -t msdos /dev/loop5 x -o uid=yourloginname -o gid=yourloginname`
  assuming your mount point is x.
  Then you can change into x and see all the files. you can create new files and directories if you wish.
  Then you can unmount.
  Later you can mount again whenever you wish.
- In invocation of your program, if filename, directory name, or pathname is entered as a parameter, then in your program you can convert the lowercase letters in the name to uppercase first, so that you can easily compare against the names stored in the filesystem. Names are stored in the filesystem so that all letters are uppercase.

References:

[1.] FAT32 File System Specification.
https://msdn.microsoft.com/en-us/library/gg463080.aspx

[2]. Understanding FAT32 File Systems.
http://www.pjrc.com/tech/8051/ide/fat32.html.

[3]. Design of FAT Filesystems.
https://en.wikipedia.org/wiki/Design_of_the_FAT_file_system

[4]. FAT File Systems.
http://wiki.osdev.org/FAT

[5]. Anatomy of the Linux File system.
http://www.ibm.com/developerworks/linux/library/l-linux-filesystem/

[6]. An example FAT32 disk image (a Linux file), called `disk1`, is provided at the following link. You can download and use it as an initial FAT32 volume to work with (size 128 MB). URL:
https://www.cs.bilkent.edu.tr/~korpe/courses/cs342spring2022/disk1