Project #4
FAT32 File System Image Modifier

Assigned: April 22, 2024 Document version: 1.2
Due date: May 15, 2024

- You will develop the project in C/Linux. Submitted projects will be tested in Ubuntu 22.04 Linux 64-bit.
- You are not allowed to share this project and/or its solution.
- The project will be done in groups of two students. You can do it individually as well. Members of a group can be from different sections.
- *Objectives/keywords*: Learn and practice with file systems, FAT32, file allocation table, mounting, block allocation, linked allocation, directory entries, free space management, sectors, blocks and clusters, block-oriented device, boot sector, superblock, device file, file attributes, raw disk access, experimentation.

In this project, you will develop a program to access and modify a FAT32 disk image (FAT32 volume). The disk image will be stored as a regular Linux file, which will simulate a disk formatted with the FAT32 file system. Your program will open the Linux file using the standard `open()` system call and access it directly in raw mode, utilizing the `read()` and `write()` system calls, without mounting the FAT32 file system.

The program will be named **fatmod**. Through various options, it will interact with a file system image, enabling reading and writing of files. Detailed explanations of the specific supported operations will be provided later in the document.

It's important to note that this project focuses specifically on the FAT32 file system, excluding VFAT or exFAT. Extensive information about FAT32 can be found online, including resources such as [1; 2; 3; 4; 6]. The reference [1] serves as the formal specification of the FAT32 file system, encompassing all the details regarding its on-disk data structures.

1. FAT32 Information

The FAT32 specification employs the terms *sectors* and *clusters*. In a FAT32 file system, a disk is conceptualized as a sequence of logical sectors (blocks), with the first sector designated as sector 0, followed by sector 1, and so forth. Therefore, each sector is identified by a number, also known as a logical block address (LBA). Typically, the sector size is 512 bytes (that means a sector can contain 512 bytes of data). While the FAT32 file system accesses the disk in terms of sectors, it allocates disk space to files in multiples of contiguous sectors, referred to as clusters.

A cluster represents a sequence of N contiguous sectors on the disk, where N is a power of 2 and remains fixed for a particular FAT32 file system established on a disk (determined at file system creation). Consequently, all clusters on a disk share the same size, with a cluster serving as the fundamental unit of disk space allocation for files and directories. For this project, the cluster size will be set at 2 sectors, equivalent to 1024 bytes (N=2). Clusters are numbered sequentially.

The Figure 1 illustrates the typical **layout** of a FAT32 file system on a disk. The initial 32 sectors (sectors 0...31) are reserved within the FAT32 file system. Sector 0, known as the boot sector, contains boot and volume information. However, as we are not implementing a bootable disk, this sector will not contain any boot code. Instead, it will store essential file system information.

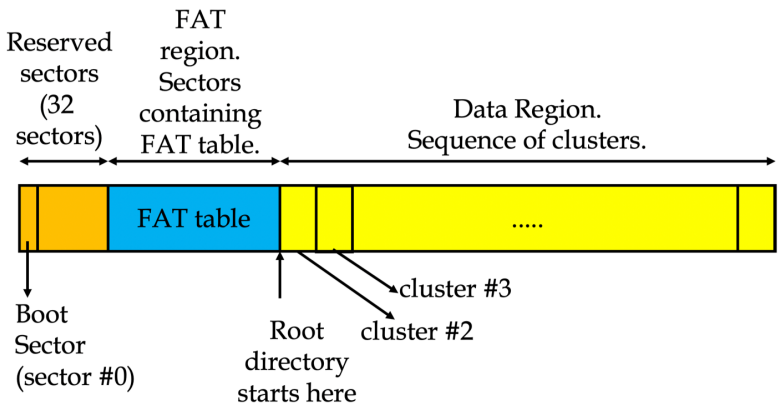

Figure 1: A typical layout of a FAT32 file system. Data region is a sequence of clusters. In this project cluster size is 2 sectors. Clusters are numbered sequentially. Numbering starts at 2. Hence the root directory starts at cluster 2. The FAT table, however, has entries for cluster 0 and cluster 1 as well.

Following the 32 reserved sectors, the FAT32 file system features the FAT table (file allocation table) directly. The FAT table commences at sector 32 and its size is a multiple of the sector size. Subsequently, the Data Region begins immediately after the FAT table. The Data Region is perceived as a sequence of clusters, commencing with cluster 2, followed by clusters 3, 4, and so forth. Notably, clusters 0 and 1 are undefined, though their respective entries exist in the FAT table (entry 0 and 1).

A file may have zero or more clusters allocated to it, with an empty file having zero clusters allocated. The clusters allocated to a file are tracked in the FAT table as a chain, where each entry in the FAT table corresponds to a cluster. The root directory begins at cluster 2 and can occupy one or more clusters. It's crucial to note that the clusters assigned to a file or directory are not obligated to be contiguous.

Figure 2 shows a sample content (in hexadecimal) for the sector 0, **boot sector**, that is relevant and applicable for this project. This is the sector 0 of the FAT32 disk image that is posted together with this assignment, which can be downloaded from [7].

Each line of output starts with the offset (in hexadecimal form) of the first byte displayed on that line. Each line displays the values of 16 bytes (in hexadecimal) from the sector. The sector starts with offset 0. At that byte we have the value 0xeb. At offset 11 (decimal), there is a two-byte value indicating the sector size in bytes, presented in **little-endian** form in the image. In FAT32, multi-byte integer values (2 bytes or 4 bytes) are stored on the disk in little-endian format. This means that at the lowest address (offset), we find the least significant byte of an integer. In contrast, in big-endian format, the most significant byte is located at the lowest address (offset). The x86-64 architecture follows the little-endian convention.

```
00000000: eb 58 90 6d 6b 66 73 2e 66 61 74 00 02 02 20 00
00000010: 01 00 00 00 00 f8 00 00 20 00 40 00 00 00 00 00
00000020: 00 00 04 00 fc 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 f2 71 51 40 43 53 33 34 32 20 20 20 20
00000050: 20 20 46 41 54 33 32 20 20 20 0e 1f be 77 7c ac
00000060: 22 c0 74 0b 56 b4 0e bb 07 00 cd 10 5e eb f0 32
00000070: e4 cd 16 cd 19 eb fe 54 68 69 73 20 69 73 20 6e
00000080: 6f 74 20 61 20 62 6f 6f 74 61 62 6c 65 20 64 69
00000090: 73 6b 2e 20 20 50 6c 65 61 73 65 20 69 6e 73 65
000000a0: 72 74 20 61 20 62 6f 6f 74 61 62 6c 65 20 66 6c
000000b0: 6f 70 70 79 20 61 6e 64 0d 0a 70 72 65 73 73 20
000000c0: 61 6e 79 20 6b 65 79 20 74 6f 20 74 72 79 20 61
000000d0: 67 61 69 6e 20 2e 2e 2e 20 0d 0a 00 00 00 00 00
```

Figure 2: The initial portion of sector #0. This is also called boot sector. It contains important information about the file system. You can consider it as superblock.

In the figure, the two-byte content starting at offset 11 appears as `0x00 0x02`. Here, `0x00` represents the least significant byte value, while `0x02` signifies the most significant byte value. Therefore, we interpret it as `0x0200`. In decimal, this translates to 512. Consequently, the sector size is 512 bytes.

At offset 13 (decimal), we encounter the information regarding the number of sectors per cluster, denoted as `0x02`. This indicates that there are 2 sectors per cluster. The reserved sector count is `0x0020`, corresponding to 32 sectors, as observed at offset 14. Moving to offset 16 (decimal), we find a byte specifying the number of FAT tables in the volume, which is indicated as 1.

Continuing, at offset 32 (which is `0x02` in hexadecimal), a 4-byte integer value represents the total number of sectors on the disk. In the given example, the byte sequence reads `0x00, 0x00, 0x04, 0x00` (in little-endian format), yielding the integer value `0x00040000`, equivalent to 262144 in decimal. This indicates that there are 262144 sectors on the disk. In the context of the FAT32 file system, the disk is perceived as a sequence of sectors (blocks), numbered consecutively from 0 to 262143 in this instance.

Proceeding to offset 44 (decimal), we encounter a 4-byte value indicating the cluster number where the root directory begins. In the provided example, this value is `0x00000002`, corresponding to 2 in decimal. Finally, at offset 35, we find a 4-byte integer value specifying the number of sectors occupied by the FAT table. In this example, it is 1020 in decimal, indicating the size of the FAT table in sectors. Thus, the FAT table spans 1020 sectors after the initial reserved 32 sectors. These represent the crucial pieces of information obtainable from the first sector of a FAT32 volume.

A **FAT table** contains information about the allocated and free clusters. It has one entry per cluster, with each entry being 4 bytes (32 bits) in size. As stated, a FAT table occupies a certain number of disk sectors (`sectors_per_FAT`). Therefore, the number of clusters that the file system can manage can be calculated 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 a certain cluster, and the number of that first cluster is stored in the

directory entry for the file. This cluster number serves as an index into the FAT table to retrieve information about the next allocated cluster for the file. If a valid cluster number is found at that index, it indicates the next cluster allocated to the file. This process repeats using the cluster number as an index into the FAT table to find the subsequent allocated clusters, forming a chain of cluster numbers for a non-empty file.

FAT32 utilizes a linked allocation scheme, with the pointers (cluster numbers) stored in the FAT table rather than in data blocks. A FAT table entry containing all 0s (`0x00000000`) indicates that the corresponding cluster is free (unused). An entry equal to or greater than `0x0FFFFFF8` indicates the end of a cluster chain (end of file) (EOC or EOF) for a file or directory. The value `0x0FFFFFF7` signifies a bad cluster mark. Additionally, a file of size zero has the value 0 as the first cluster number in its directory entry.

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 $2^{28} \times 2^{10} = 2^{38}$ bytes = 256 GB. If clustersize is 32 KB, then the disk size could be $2^{28} \times 2^{15} = 2^{43}$ bytes = 8 TB. However, 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 $2^{32} \times 5^{12} = 2^{41}$ bytes = 2 TB, assuming sector size is 512 bytes.

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

A **directory** (such as the root directory) is also like a file, capable of occupying one or more clusters. The content of those clusters forms a sequence of directory entries. Each directory entry contains vital information about a file, such as its name, size, first cluster number, etc. The structure of a FAT32 directory entry is illustrated in Figure 3. FAT32 employs fixed-size directory entries, with each entry being 32 bytes in size.

Figure 4 displays a sample content of the root directory in our example. A directory comprises a sequence of 32-byte entries, also known as directory records. In the root directory, the first directory entry stores the volume label, which in this example is `CS342`, representing the root directory itself.

The subsequent 32 bytes (next entry) are allocated for a regular file named `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. At offset 11 (decimal), there is a 1-byte attribute information. The bits of this byte indicate the type of the file. If the byte value is `0x10`, it indicates that the entry is for a directory (i.e., sub-directory). If the byte value is `0x08`, then the entry is for the volume ID, i.e., for the root directory (not a regular file entry).

For a regular file, the least significant 3 bits of the attribute byte indicate the type of the file (e.g., hidden, system, etc.). For a regular file, all these 3 bits may be zero, or some of them may be 1. However, the least significant 4 bits of the attribute field should not all be set to 1 in this project. Setting all 1s for these bits

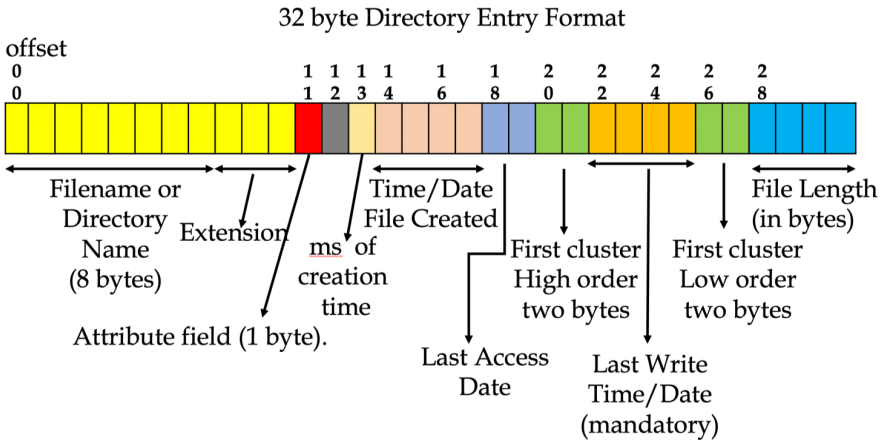

Figure 3: Directory entry structure. A directory is a sequence of such directory entries. Each directory entry keeps information about a file (or subdirectory). The information include filename (name and extension parts), first cluster number (the number of the cluster at which file data starts), and file length (in bytes).

indicates the use of long filenames, which will not be used in this project for the sake of simplicity. If the attribute value is `0x20`, it means the file is a regular file.

The 2 bytes (high bytes) at offset 20 (decimal) and the 2 bytes (low bytes) at offset 26 together indicate 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 `0x00, 0x00`, and the low two bytes are `0x03, 0x00`. When combined, they form the number `0x00000003`, which is 3 in decimal. This indicates that the file starts at cluster 3.

The 4 bytes at offset 28 indicate the size of the file in bytes. For example, for the file `FILE1.BIN`, the size is `0x00001400`, which means 5120 bytes.

A directory entry that starts with a byte value of `0xe5` is empty, indicating it is available (previously used and then deleted), and can be used for a new file that is created. An entry that starts with a byte value of `0x00` is also an unused entry.

2. Creating and Using a FAT32 Image

We can create a regular Linux binary file that will act as a disk, a virtual disk, in various ways. One simple method is using the `dd` command as follows. It 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 (2^{18}) 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 `disk1`.

We can use the `losetup` command to associate a Linux loop device file (for example, `/dev/loop22`) 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
```

```
00000000: 43 53 33 34 32 20 20 20 20 20 20 08 00 00 56 8c
00000010: 95 58 95 58 00 00 56 8c 95 58 00 00 00 00 00 00
00000020: 46 49 4c 45 31 20 20 20 42 49 4e 20 00 00 00 00
00000030: 00 00 00 00 00 00 ae 74 95 58 03 00 00 14 00 00
00000040: 46 49 4c 45 32 20 20 20 42 49 4e 20 00 00 00 00
00000050: 00 00 00 00 00 00 b2 74 95 58 00 00 00 00 00 00
00000060: 46 49 4c 45 33 20 20 20 42 49 4e 20 00 00 00 00
00000070: 00 00 00 00 00 00 b8 74 95 58 08 00 00 48 00 00
00000080: 46 49 4c 45 34 20 20 20 54 58 54 20 00 00 00 00
00000090: 00 00 00 00 00 00 ce 74 95 58 1a 00 79 1c 00 00
000000a0: 46 49 4c 45 35 20 20 20 54 58 54 20 00 00 00 00
000000b0: 00 00 00 00 00 00 e5 74 95 58 22 00 13 00 00 00
000000c0: e5 49 4c 45 35 20 20 20 42 49 4e 20 00 00 00 00
000000d0: 00 00 00 00 00 00 f9 74 95 58 23 00 00 90 01 00
000000e0: 46 49 4c 45 36 20 20 20 42 49 4e 20 00 00 00 00
000000f0: 00 00 00 00 00 00 f9 74 95 58 23 00 00 90 01 00
00000100: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Figure 4: An example root directory content.

It will give the first available (unused) loop device, for example, `/dev/loop22`. Then we can associate that loop device, which is block-oriented, with our disk file as follows:

`sudo losetup /dev/loop22 disk1.`

To delete the association between a loop device file and a Linux file, we can use the `-d` option of `losetup` command as:

`sudo loosetup -d /dev/loop22`

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 (i.e., put a file system on the 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 the same options shown below.

`mkfs.fat -v -S 512 -s 2 -F 32 -f 1 -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 1 32-bit FAT and 2 sectors per cluster.
FAT size is 1020 sectors, and provides 130546 clusters.
There are 32 reserved sectors.
Volume ID is 405171f2, volume label CS342
```

In the command above, the `-F 32` 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, `-f 1` option says that there will be only 1 FAT table created, 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.

You can get more information about `mkfs.fat` using its manual page (`man mkfs.fat`).

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 `top`. We type the following to create the directory `top`.

```
mkdir top
```

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/loop22 top -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 `top`. Therefore, we can change to directory `top`. When we are in the directory `top`, 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.

After we create some additional files, or after doing some modifications to the file system, like 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 top
```

3. Program Operations (Options)

Your program will access a FAT32 disk image (such as `disk1`) and will read/write some data and metadata from/to it, as specified below, without mounting it. The name of the program you will write will be `fatmod.c`, and the respective executable file will be called `fatmod`.

You can assume that there will be only one directory in the file system, which is the root directory. Additionally, you can assume that the root directory will be at most `N_ROOTDIR_CLUSTERS` long (these clusters do not have to be adjacent to each other). Furthermore, you can 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. For example, `project.pdf` is a short filename. Long filenames will not be used in this project.

Below are the operations that your program will perform on a FAT32 disk image. An invocation of your program with a particular option and parameters will execute a specific operation and print out the related information. Your program will support the options and operations listed and described below.

Each operation is performed by a separate invocation of your program with related parameters, and each invocation must include the `DISKIMAGE` parameter, specifying the FAT32 disk image file to use.

1. `fatmod DISKIMAGE -l`. When invoked with the `-l` option, the program will list the names of the files in the root directory along with their extensions and size information (in bytes). For instance:

   ```
   AFILE.BIN 1200
   BFILE.TXT 540
   CFILE.BIN 6500
   DFILE.BIN 125000
   ```

2. `fatmod DISKIMAGE -r -a FILENAME`. When executed with the `-r -a` option, the program will display the content of the file named `FILENAME` in ASCII form on the screen. It is assumed that the file is an ASCII text file. The `DISKIMAGE` argument is the name of the Linux file acting as a FAT32 formatted disk.

3. `fatmod DISKIMAGE -r -b FILENAME`. When executed with the `-r -b` option, the program will display the content of the file named `FILENAME` in binary form on the screen. The content can be either binary or text. Each byte will be printed in hexadecimal form. Below is an example output, with each line printing 16 bytes. The first hexadecimal number indicates the start offset of that line in the file.

   ```
   00000000: 4e 41 4d 45 20 20 20 6c 6f 73 65 74 75 70 20 2d
   00000010: 20 73 65 74 20 75 70 20 61 6e 64 20 63 6f 6e 74
   00000020: 72 6f 6c 20 6c 6f 6f 70 20 64 65 76 69 63 65 73
   00000030: 0a 53 59 4e 4f 50 53 49 53 0a 20 20 20 20 20 20
   00000040: 20 47 65 74 20 69 6e 66 6f 3a 0a 20 20 20 20 20
   00000050: 20 20 20 20 20 20 6c 6f 73 65 74 75 70 20 5b 6c
   00000060: 6f 6f 70 64 65 76 5d 0a 20 20 20 20 20 20 20 20
   00000070: 20 20 20 20 6c 6f 73 65 74 75 70 20 2d 6c 20 5b
   00000080: 2d 61 5d 0a 20 20 20 20 20 20 20 20 20 20 20 20
   ```

4. `fatmod DISKIMAGE -c FILENAME`. With the `-c` option, your program will create a file named `FILENAME` in the root directory. This file will have a corresponding directory entry, an initial size of 0, and no blocks allocated for it initially.

5. `fatmod DISKIMAGE -d FILENAME`. With the `-d` option, the program will delete (remove) a file and all its associated data. The blocks allocated to the file will be deallocated, and it will no longer have a directory entry.

6. `fatmod DISKIMAGE -w FILENAME OFFSET N DATA`. With the `-w` option, the program will write data into the file starting at offset `OFFSET`. The number of bytes to write is specified by `N`. The same data byte will be written for `N` consecutive bytes. The content of the byte is given by the `DATA` argument, which is an unsigned integer between 0 and 255. For example, if `DATA` is 65, it represents the letter 'A' in ASCII encoding.

   For instance, if we specify `DATA` as 48 and `N` as 10, then 10 consecutive bytes in hexadecimal will be: `0x30, 0x30, 0x30, 0x30, 0x30, 0x30 0x30, 0x30, 0x30, 0x30`.

It's important to note that with this operation, existing data in the file may be overwritten, and new data may be added. If new data is added, it may require allocating one or more new blocks to the file.

7. `fatmod -h`. When the `-h` option is specified, the program will display a help page containing a list of all available options (operations) along with their respective parameters, if any.

Note that the program will be invoked with one option at a time. That means a single invocation of the program will perform only one of the operations specified above. To do multiple operations, we need to invoke the program multiple times.

4. Developing the Program

In a 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 `struct fat_boot_sector` and `struct 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 `struct fat_boot_sector` structure; assuming that the machine is **little-endian**.

```
    unsigned char sector[512];
    ...
    struct fat_boot_sector * bp;
    unsigned char num_fats, sectors_per_cluster;
    unsigned int num_sectors, sectors_per_fat;
    unsigned int root_start_cluster;

    // ....read sector 0 into  array sector
    ...
    bp = (struct fat_boot_sector *) sector; // type casting
    // below we access the related data from the boot 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;

// read a cluster data from disk into array cluster

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 after incrementing the `dep` pointer as: `dep++`.

To use `struct fat_boot_sector` and `struct 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()` functions 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 *)(cluster + 28), sizeof(int));
```

This will copy 4 bytes (integer size) from the offset 28 of 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. 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 * sectors_per_FAT;
```

Below is a code example about how to read and write a sector from the disk file that is opened and its descriptor is `fd`. Assume the variable `buf` is a character (byte) array defined (or allocated space) outside of these functions and will store the content of one sector.

```
#define SECTORSIZE  512
#define CLUSTERSIZE 1024
...
int
```

```
readsector (int fd, unsigned char *buf, uint 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);  // error
        return (0);
}

int
writesector (int fd, unsigned char *buf, uint snum)
{
        off_t offset;
        int n;
        offset = snum * SECTORSIZE;
        lseek (fd, offset, SEEK_SET);
        n  = write (fd, buf, SECTORSIZE);
        fsync (fd);  // enforce output immediately to disk
        if (n == SECTORSIZE)
                return (0);
        else
                return (-1); // error
        return (0);
}
```

Using these, you can easily develop readcluster() and writecluster() functions.

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

5. Additional Information and Constraints

1. Start early, work incrementally.
2. 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.
3. 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.
4. Since clustersize is 1024 bytes and directory entry size is 32 bytes, we can have at most 32 entries in a directory.
5. In invocation of your program, if a FAT32 filename 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 FAT32 file system so that all letters are uppercase.

6. Tips, Clarifications, Additional Requirements

1. Tips, clarifications, additional requirements about the project will be added to the following *web-page*: `https://www.cs.bilkent.edu.tr/~korpe/courses/cs342spring2024/dokuwiki/doku.php?id=p4-s24`
2. The page above will also be linked from course's Moodle page (under the project document). Please check this project web-page regularly to follow the clarifications.
3. This page may include additional requirements or specifications. If there is a conflict between a requirement stated here and requirement stated in the Clarifications page, you need to follow the one in the Clarifications page. Clarification page requirements will supersede the requirements here (if any).
4. Further *useful information*, *tips*, and *explanations* on this page will help you in developing the project. It will be a dynamic, living page, that may be updated (based on your questions as well) until the project deadline.
5. An initial project skeleton (including a `Makefile`) is added to github at the following address. You can start with this if you wish. `https://github.com/korpeoglu/cs342spring2024-p4`

7. Experiments and Report

You will design and conduct some performance experiments. You will then interpret them. You will put your results, interpretations, and conclusions in a report.pdf file. You will include this file as part of your uploaded package. Experiments and the related report will be **20% of the project grade**.

8. Submission

Submission will be the same as in the previous projects.

Put all your files into a directory named with your Student ID. If the project is done as a group, the IDs of both students will be written, separated by a dash '-'. In a `README.txt` file, write your name, ID, etc. (if done as a group, all names and IDs will be included). The set of files in the directory will include `README.txt`, `Makefile`, and program source file(s). We should be able to compile your program(s) by just typing `make`. No binary files (executable files or .o files) will be included in your submission. Then **tar** and **gzip** the directory, and submit it to **Moodle**.

For example, a project group with student IDs `21404312` and `214052104` will create a directory named `21404312-214052104` and will put their files there. Then, they will tar the directory (package the directory) as follows:

```
tar cvf 21404312-214052104.tar 21404312-214052104
```

Then they will gzip (compress) the tar file as follows:

```
gzip 21404312-214052104.tar
```

In this way they will obtain a file called `21404312-214052104.tar.gz`. Then they will upload this file to **Moodle**. For a project done individually, just the ID of the student will be used as a file or directory name.

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] FAT Description. https://www.win.tue.nl/~aeb/linux/fs/fat/fat-1.html.

[7] 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/disk1.