Middle East Technical University | Department of Computer Engineering

CENG 334

Introduction to Operating Systems

Spring 2021-2022

Homework 3 - Fat32 Operations

Due date: 28 06 2022, Tuesday, 23:59

1 Introduction

The objective of the final homework is to understand and implement some operations on the FAT32 filesystem by creating a shell like application that read the relevant information from a FAT32 formatted image file, output information and/or modify contents.
Keywords: *FAT32, filesystems*

2 FAT32

2.1 Filesystem Details

FAT is a file system primarily developed for floppy disks and later became the default for Microsoft Windows computers before giving way to NTFS filesystem. The filesystem uses an index table to identify chains of clusters of a file, called File Allocation Table (FAT). The table consists of linked list of entries for each cluster.

The *root directory* contains the number of the first cluster of each file in that directory. It is possible to traverse each file by looking up the cluster number of each successive part of the disk file as a cluster chain until the end of the file is reached. The sub-directories are implemented as special files containing *directory entries* of their respective files. The table entries are fixed in size and this determines the version of the FAT. FAT32 uses 32 bit(4 bytes) size table entries.

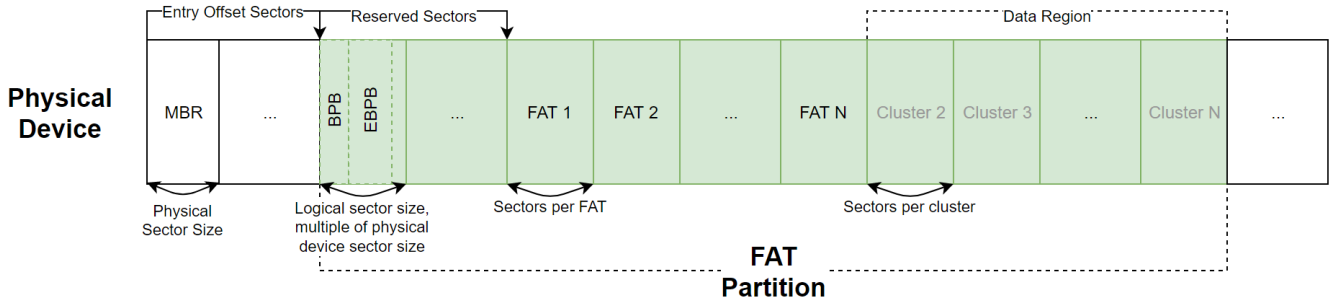

Figure 1: FAT32 Layout

FAT layout is shown in Figure 1 consists of four main parts. First there are reserved sectors for general information about the filesystem and its parameters. Following that part, there are the File Allocation Tables. More than one of them can exist for redundancy sake and the default number is two. Third is the root director region which only exists in FAT12/16 versions so it is not relevant in our case. Finally there are the data regions which for FAT32 contain the root directory. The regions, their sizes and contents can be summarized by the table below:

Table 1: Main Regions and Their Information

Region	**Size in sectors**	**Contents**
Reserved sectors	(# of reserved sectors)	Boot Sector
		FS Information Sector (FAT32 only)
		More reserved sectors (optional)
FAT Region	(# of FATs) * (sectors per FAT)	File Allocation Table #1
		File Allocation Table #2 ... (optional)
Root Directory Region	(# of root entries * 32) / (bytes per sector)	Root Directory (FAT12/16 only)
Data Region	(# of clusters) * (sectors per cluster)	Data Region (for files and directories)

The first part of the reserved sector is the boot sector which contains crucial file system information. It is called BIOS Parameter Block (BPB) and it has the same structure across all FAT versions. It contains information as bytes per sector and logical sectors per cluster. Moreover, it is possible use the values of the first three bytes to perform an integrity check on the FAT system if needed. There are also some FAT12/16 parameters which will not be needed for FAT32. The parameters can be seen below:

BIOS Parameter Block

```
struct BPB_struct {

    uint8_t BS_JumpBoot[3];         // Jump Instruction. Size: 3 bytes
    uint8_t BS_OEMName[8];          // The system that formatted the disk.
    uint16_t BytesPerSector;        // Bytes per logical sector
    uint8_t SectorsPerCluster;      // Logical sectors per cluster
    uint16_t ReservedSectorCount;   // # of reserved logical sectors
    uint8_t NumFATs;                // Number of file allocation tables.
    uint16_t RootEntryCount;        // It is 0 for FAT32.
    uint16_t TotalSectors16;        // Total logical sectors.
    uint8_t Media;                  // Media descriptor
    uint16_t FATSize16;             // It is 0 for FAT32.
    uint16_t SectorsPerTrack;       // Not relevant
    uint16_t NumberOfHeads;         // Not relevant
    uint32_t HiddenSectors;         // Not relevant
    uint32_t TotalSectors32;        // Total logical sectors
    BPB32_struct extended;          // Extended parameters for FAT32
};
```

This structure is followed by version specific information called Extended BIOS Parameter Block (EPBP) follows. It contaisn information about drive description, version and the cluster of the root directory. It also contains the boot signature to determine FAT version. Finally, the size of the FAT table is given, to calculate necessary information for accessing the data region. The structure can be seen below:

Extended BIOS Parameter Block

```
struct BPBFAT32_struct {
    uint32_t FATSize;               // Logical sectors per FAT.
    uint16_t ExtFlags;              // Drive description/Mirroring flags.
    uint16_t FSVersion;             // Version.
    uint32_t RootCluster;           // Cluster number of root directory start.
    uint16_t FSInfo;                // Sector number of FS Information.
    uint16_t BkBootSec;             // First sector of a copy FAT32 boot sector.
    uint8_t Reserved[12];           // Reserved bytes.
    uint8_t BS_DriveNumber;         // Physical drive number.
    uint8_t BS_Reserved1;           // Reserved.
    uint8_t BS_BootSig;             // Boot signature. 0x29 for FAT32.
    uint32_t BS_VolumeID;           // Volume ID.
    uint8_t BS_VolumeLabel[11];     // Volume Name.
    uint8_t BS_FileSystemType[8];   // File system type.
};
```

2.1.1 File Allocation Table

A volume's data area is split into identically sized clusters. Each file may occupy one or more clusters depending on its size. Thus, a file is represented by a chain of clusters (a singly linked list). The FAT32 file system uses 32 bits per FAT entry, with each entry spanning four bytes in little-endian byte order. The four top bits of each entry are reserved for other purposes; they are cleared during formatting and should not be changed otherwise. They must be masked off before interpreting the entry as 28-bit cluster address. The File Allocation Table (FAT) is a contiguous number of sectors immediately following the area of reserved sectors. It represents a list of entries that map to each cluster on the volume. Each entry records one of five things:

- the cluster number of the next cluster in a chain
- a special end of cluster-chain (EOC) entry that indicates the end of a chain
- a special entry to mark a bad cluster
- a zero to note that the cluster is unused

An example FAT can be seen below:

Table 2: Example File Allocation Table

Offset	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+A	+B	+C	+D	+E	+F
+0000	F0	FF	FF	0F	FF	FF	FF	0F	FF	FF	FF	0F	04	00	00	00
+0010	05	00	00	00	06	00	00	00	07	00	00	00	08	00	00	00
+0020	FF	FF	FF	0F	0A	00	00	00	14	00	00	00	0C	00	00	00
+0030	0D	00	00	00	0E	00	00	00	0F	00	00	00	10	00	00	00
+0040	11	00	00	00	FF	FF	FF	0F	00	00	00	00	FF	FF	FF	0F
+0050	15	00	00	00	16	00	00	00	19	00	00	00	F7	FF	FF	0F
+0060	F7	FF	FF	0F	1A	00	00	00	FF	FF	FF	0F	00	00	00	00
+0070	00	00	00	00	F7	FF	FF	0F	00	00	00	00	00	00	00	00

- First chain (1 cluster) for the root directory, pointed to by an entry in the FAT32 BPB (here: #2)
- Second chain (6 clusters) for a non-fragmented file (here: #3, #4, #5, #6, #7, #8)
- Third chain (7 clusters) for a fragmented, possibly grown file (here: #9, #A, #14, #15, #16, #19, #1A)
- Fourth chain (7 clusters) for a non-fragmented, possibly truncated file (here: #B, #C, #D, #E, #F, #10, #11)
- Empty clusters (here: #12, #1B, #1C, #1E, #1F)
- Fifth chain (1 cluster) for a sub-directory (here: #13)
- Bad clusters (3 clusters) (here: #17, #18, #1D)

2.1.2 Directory Table

A directory table is a special type of file. Each file or sub-directory stored within the table is represented by a 32-byte entry. Each entry records the name, extension, attributes (archive, directory, hidden, read-only, system and volume), the address of the first cluster of the file/directory's data, the size of the file/directory, and the date and also the time of last modification. The first byte of the filename can be marked to indicate four cases:

- **0x00**: Entry is available and no subsequent entry is in use.
- **0x05**: Initial character is actually `0xE5`.
- **0x2E**: Dot' entry; either "." or "..".
- **0xE5**: Entry has been previously erased and/or is available.

Directory entry can be structured with the following format:

FAT Directory Entry in 8.3 format

```
struct FatFile83 {
    uint8_t filename[8];          // Filename for short filenames.
    uint8_t extension[3];         // Remaining part
    uint8_t attributes;           // Attributes
    uint8_t reserved;             // Reserved to mark extended attributes
    uint8_t creationTimeMs;       // Creation time down to ms precision
    uint16_t creationTime;        // Creation time with H:M:S format
    uint16_t creationDate;        // Creation date with Y:M:D format
    uint16_t lastAccessTime;      // Last access time
    uint16_t eaIndex;             // Used to store first two bytes of the first clust
    uint16_t modifiedTime;        // Modification time with H:M:S format
    uint16_t modifiedDate;        // Modification date with Y:M:D format
    uint16_t firstCluster;        // Last two bytes of the first cluster
    uint32_t fileSize;            // Filesize in bytes
};
```

If the filename does not fit within the 11 byte space in the previous structure, additional long filename structure can be added before the directory entry. It is called VFAT long file names (LFN). LFNs can be added as much as needed until the maximum filename size 256 is reached. The structure contain 13 more character spaces in UTF-16 format (which is backward ASCII compatible). There are also three different parameters that differ from 8.3 filename format to indicate whether it is a LFN or not. LFNs causes the operating systems to ignore the name written in the 8.3 structure. In the modern systems, there is always at least one LFNs preceding 8.3 filename format and it solely stores the name of the file. This means that name of the file or folder in a directory is stored using only LFNs. Finally it contains a checksum to calculate whether the structure is correct. The structure is given below:

VFAT Long File Name Structure

```
struct FatFileLFN {
    uint8_t sequence_number;
    uint16_t name1[5];      // 5 Chars of name (UTF-16 format)
    uint8_t attributes;     // Always 0x0f
    uint8_t reserved;       // Always 0x00
    uint8_t checksum;       // Checksum of DOS Filename.
    uint16_t name2[6];      // 6 More chars of name (UTF-16 format)
    uint16_t firstCluster;  // Always 0x0000
    uint16_t name3[2];      // 2 More chars of name (UTF-16 format)
};
```

The following links contain more details about FAT32 and should be your go-to references when writing your code:

- Design of the FAT file system article on Wikipedia:
  https://en.wikipedia.org/wiki/Design_of_the_FAT_file_system
- OSDev wiki article on FAT: https://wiki.osdev.org/FAT

2.2 Image Creation

To create an FAT32 filesystem image, first create a zero file via dd. Here's an example run creating a 100MB file (102400 1024-byte blocks).

```
$ dd if=/dev/zero of=example.img bs=1024 count=102400
```

Then, format the file into a filesystem image via mkfs.vfat. The following example creates an FAT32 filesystem with 2 sectors per cluster and a sector size of 512 bytes. Sector size will always be 512 for this homework.

```
$ mkfs.vfat -F 32 -S 512 -s 2 example.img
```

You can check filesystem details with the `fsck.vfat` command:

```
$ fsck.vfat -vn example.img
```

Now that you have a filesystem image, you can mount the image onto a directory. The FUSE based fusefat command allows you to do this in userspace without any superuser privileges (use this on the ineks[1]). The below example mounts our example filesystem in read-write mode:

```
$ mkdir fs-root
$ fusefat -o rw+ -o umask=770 example.img fs-root
```

Now, fs-root will be the root of your filesystem image[2]. You can cd to it, create files and directories, do whatever you want. To unmount once you are done, use fusermount:

```
$ fusermount -u fs-root
```

Make sure to unmount the filesystem before running programs on the image. On systems where you have superuser privileges, you can use the more standard mount and unmount:

```
$ sudo mount -o rw example.img fs-root
$ sudo umount fs-root
```

You can check the consistency of your filesystem after modifications with the same `fsck.vfat` command. The below example forces the check in verbose mode and refuses fixes (-n flag). This will help you find bugs in your implementation later on.

```
$ fsck.vfat -vn example.img
```

[1]Note that fusermount does not currently work on the ineks without setting group read-execute permissions for the mount directory (chmod g+rx) and group execute permissions (g+x) for other directories on the path to the mount directory (including your home directory). This is not ideal and unfortunately there is no fix. My suggestion is to during the period where your home directory is readable quickly make changes and unmount the image file. After that you can fix the permission to the previous settings

[2]umask is critical when mounting on ineks. Otherwise you will not be able to unmount the image file and would need to restart the pc. On your own pc, it is not required.

3 Implementation

You are going to implement a shell like application that contains six file system operations. It should take a single image file as an argument and execute the operations on that image file. During execution, it should print a prompt containing current directory in the following format:

`[current_working_directory]>[space]`

Current working directory is given as an absolute path from the root separated with `/` character. Example prompt can be written as:

```
/home/videos>
```

After every operation, your program should print the prompt again, even if there is no output from the operation. At the beginning of the execution, the current directory is root. Full example from start is given below:

```
./hw3 example.img
/>
```

The program should continue until receiving the `quit` command. The operations your program must implement is given below:

- Change Directory
- List Directory Contents
- Make Directory
- Create File
- Move File/Directory
- Show File Contents

3.1 Change Directory

Your program should change its working directory with the change directory command. The command is same as the linux shell command `cd`. It takes a single argument, which is the target directory. It can be a relative or absolute path. Examples:

```
/> cd /home/pictures/cat
/home/pictures/cat> cd ..
/home/pictures> cd ../videos
/home/videos> cd /img/source
/img/source>
```

If the folder does not exists, simply keep the previous directory without any change. Example:

```
/home/pictures> cd wrongfolder
/home/pictures>
```

3.2 List Directory Contents

Your program should print the directory contents to the standard output upon receiving this command. The command to list directory contents is the same as the linux shell command `ls`. It can be called with up to two arguments. If it is called without any arguments, it should print the files in the current working directory separated with a space. Example:

```
/home/pictures/cat> ls
cat1.png cat2.jpg fluffy.png
/home/pictures/cat>
```

It can also take a directory as an argument in relative or absolute path. In this case it should print the files of that folder.

```
/home/pictures/cat> ls /home/videos
graduation.mp4 cats.mkv dogvideos
/home/pictures/cat>
```

Finally it can take `-l` as argument to print more information to the output. This argument can be called with or without a directory, however, it will always be given before the directory path. The print format should be the same as the linux command. However, since the FAT32 filesystem does not support file permissions, file ownership, and links, the beginning is fixed. Format for every file and folder in the directory is given below. File:

```
-rwx------ 1 root root <file_size_in_bytes> <last_modified_date_and_time> <filename>
```

Folder:

```
drwx------ 1 root root 0 <last_modified_date_and_time> <filename>
```

If it is called without a folder argument, it should print the current working directory in detail. Example:

```
/> ls -l
-rwx------ 1 root root 2464 May 25 10:12 f1
-rwx------ 1 root root 10240 May 23 17:21 f2
-rwx------ 1 root root 0 May 21 15:36 f3
drwx------ 1 root root 0 May 21 15:37 t5
drwx------ 1 root root 0 May 21 15:37 t6
drwx------ 1 root root 0 May 21 15:37 t7
```

Similarly, it can also be called with an additional folder argument. Example:

```
/> ls -l /home/pictures
drwx------ 1 root root 0 June 17 03:01 cat
drwx------ 1 root root 0 March 03 23:30 dog
drwx------ 1 root root 0 April 21 12:23 etc
```

If the directory given as an argument does not exists, there should be no output.
Note: Do not forget that every file or folder has at least one LFN to represent its filename.

3.3 Make Directory

Your program should create a directory with the make directory command. The command is same as the linux shell command `mkdir`. It takes a single argument, which is the target directory to be created. It can be a relative, absolute path or just the folder name. In the last case, the folder should be created in the current working directory. Examples:

```
/home/pictures> ls
cat dog etc
/home/pictures> mkdir birthdays
/home/pictures> ls
cat dog etc birthdays
/home/pictures> mkdir /home/pictures/flags
/home/pictures> ls
cat dog etc birthdays flags
/home/pictures> mkdir ../videos/movies/catvideos
/home/pictures> ls ../videos/movies
graduation.mp4 cats.mkv dogvideos catvideos
```

If the folder already exists or the path is wrong, there should be no changes and no output to indicate failure.
Note: Please create at least one LFN to represent the filename of a file or directory before 8.3 structure. Even if the filename is shorter than 11 characters. You do need to set the filename to a unique value in the 8.3 format to prevent duplicate filename error on fsck. Recommended format:

```
~<index>
```

where index is the order file has appeared. Make sure to write down the checksum correctly.

3.4 Create File

Your program should create a file with the create file command. The command is same as the linux shell command `touch`. It takes a single argument, which is the target filename to be created. It can be a relative, absolute path or just the filename. In the last case, the file should be created in the current working directory. Examples:

```
/home/videos> ls
graduation.mp4 cats.mkv dogvideos
/home/videos> touch footage.mkv
/home/videos> ls
graduation.mp4 cats.mkv dogvideos footage.mkv
/home/videos> touch  /home/pictures/cute.png
/home/videos> ls /home/pictures
cat dog etc cute.png
/home/videos> touch ../pictures/cat/cat3.png
/home/videos> ls ../pictures/cat
cat1.png cat2.jpg fluffy.png cat3.png
```

If the file already exists or the path is wrong, there should be no changes and no output to indicate failure.
Note: Please create at least one LFN to represent the filename of a file or directory before 8.3 structure. Even if the filename is shorter than 11 characters. You do need to set the filename to a unique value in the 8.3 format to prevent duplicate filename error on fsck. Recommended format:

```
~<index>
```

where index is the order file has appeared. Make sure to write down the checksum correctly.

3.5 Move File/Directory

Your program should move a directory with the move command. The command is same as the linux shell command `mv`. It takes two arguments. First argument is the source file/folder and the second argument is the destination directory. Both can be a relative, absolute paths. First argument can also be just the file/folder name. In that case, the file/folder should be moved from the current working directory. Unlike the actual linux command, this cannot be used to rename files/folder. The second argument just shows the destination directory. Examples:

```
/home/videos/dogvideos> ls
terrier.mp4 lucky.mkv
//home/videos/dogvideos> mv ../cats.mkv .
/home/videos/dogvideos> ls
terrier.mp4 lucky.mkv cats.mkv
/home/videos/dogvideos> mv cats.mkv /home/videos
/home/videos/dogvideos> cd ..
/home/videos> ls /home/pictures
cat dog etc
/home/videos> mv /home/pictures/cat ../pictures/etc/
/home/videos> ls ../pictures/etc
waterslide.png vacation.jpeg cat
```

If the file/folder already exists in the destination or any of the paths are wrong, there should be no changes and no output to indicate failure.

3.6 Show File Contents

Your program should print the contents of a file with the show file contents command. The command is same as the linux shell command `cat`. It takes a single argument, which is the target filename to be shown. It can be a relative, absolute path or just the filename. In the last case, the file should be in the current working directory. Examples:

```
/home/books> ls
magna_carta.txt fantasy scifi
/home/books> cat magna_carta.txt
JOHN, by the grace of God King of England, Lord of Ireland, Duke of Normandy and Aquitaine, an
...
...
...
... seventeenth year of our reign (i.e. 1215: the new regnal year began on 28 May).
/home/books> cat scifi/dune1.txt
<contents of dune>
```

If the file does not exists or the path is wrong, there should be no changes and no output to indicate failure.

4 Specifications

- Your program must terminate when the user gives the *quit* command. Otherwise your homework will not be able to be graded.
- When creating a file with `touch` or a folder with `mkdir` command, creation and modification dates and times should be set to the current date and time. Move command is not considered modification so there is no need to update the time.
- Please create at least one LFN to represent the filename of a file or directory before 8.3 structure. Even if the filename is shorter than 11 characters. You do need to set the filename to a unique value in the 8.3 format to prevent duplicate filename error on fsck. Recommended format:

  ```
  ~<index>
  ```

  where index is the order file has appeared. Make sure to write down the checksum correctly. However, this error will **NOT** take any points from your grade so it is not critical.
- Evaluation will be done using black box technique. So, your programs must not print any unnecessary characters.
- The file and folder names will not contain any spaces.
- Folders given as arguments do not contain the '`/`' at the end.
- The filesystem should not be corrupted after executing commands.
- During grading, commands will be both tested individually and together. Both of them will consists of equal parts of the grade for that command.
- Tentative grade percentages for commands are:
  - **cd**: %10
  - **ls**: %25
  - **mkdir**: %10
  - **touch**: %10
  - **mv**: %15
  - **cat**: %30

5 Regulations

- **Programming Language:** Your program should be coded in C or C++. Your submission will be compiled with `gcc` or `g++` on department lab machines. Make sure that your code compiles successfully.
- **Late Submission:** Late submission is allowed but with a penalty of $5 * day * day$.
- **Cheating:** Everything you submit must be your own work. Any work used from third party sources will be considered as cheating and disciplinary action will be taken under or "zero tolerance" policy.
- **Newsgroup:** You must follow the ODTUClass for discussions and possible updates on a daily basis.
- **Grading:** This homework will be graded out of 100. It will make up 10% of your total grade.

6 Submission

Submission will be done via ODTUClass. Create a tar.gz file named `hw3.tar.gz` that contains all your source code files along with a makefile. The tar file should not contain any directories! The make should create an executable called hw3. Your code should be able to be executed using the following command sequence.

```
$ tar -xf hw3.tar.gz
$ make
$ ./hw3 <image_file>
```