



EE442 PROGRAMMING ASSIGNMENT 3

File System

Due: June, 12, 2022, 23:59

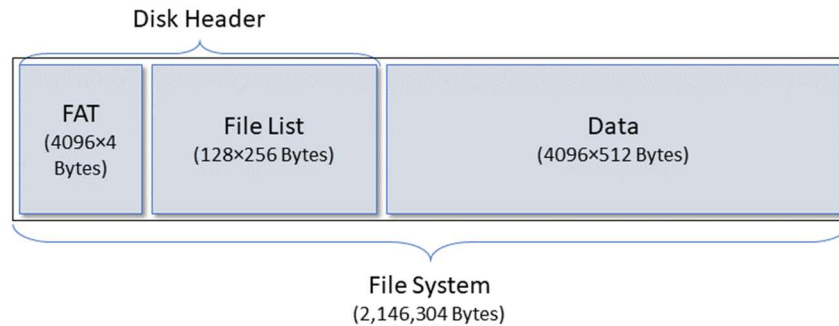
* For your questions use forum or send email to ksert@metu.edu.tr.

Submission

- Send your homework compressed in an archive file with name “eXXXXXXX_ee442_hw3.tar.gz”, where X’s are your **7-digit student ID number**. You will **not** get full credit if you fail to submit your work as required.
- Your work will be graded on its correctness, efficiency, clarity and readability as a whole.
- Comments will be graded. You should insert comments in your source code at appropriate places without including any unnecessary detail.
- Late submissions are welcome, but are penalized according to the following policy:
 - 1 day late submission : HW will be evaluated out of 70.
 - 2 days late submission : HW will be evaluated out of 40.
 - Later submissions : HW will NOT be evaluated.
- The homework must be written in **C** (**not** in C++ or any other language).
- You should **not** call any external programs in your code.
- **Check** what you upload. Do not send corrupted, wrong files or unnecessary files.
- The homework is to be prepared **individually**. Group work is **not** allowed. Your code will be **checked** for cheating.
- The design should be your original work. However, if you partially make use of a code from the Web, give proper **reference** to the related website in your comments. Uncited use is unacceptable.
- **METU honor code is essential**. Do **not** share your code. Any kind of involvement in cheating will result in a **zero** grade, for **both** providers and receivers.

Description:

In this homework, you are expected to implement a simple user mode file system (FS) that employs a file allocation table (FAT). The FS obtained could be used on a physical drive or on a disk image to be named as “disk”.



The FS consist of three parts:

1. File Allocation Table
2. File List
3. Disk Data

Block size is 512 bytes, meaning that the files are written in 512 byte blocks. FS has no directories.

1. File Allocation Table (FAT):

- FAT consists of 4096 entries. Each entry is 4 bytes long and bytes are laid out in little-endian order (meaning the most significant byte is at the highest address).
- The first entry must always be 0xFFFFFFFF.
- If an entry is 0x0, then the block is assumed to be empty.
- If an entry is 0xFFFFFFFF, then the block is assumed to be the last block of a file.
- If an entry has a value between 0x0 and 0x1000, the value points to the entry in the table where the next block of the file is located.

Example: Consider three files; “File A” occupies 4 blocks, “File B” occupies 2, and “File C” occupies 1. Then, the following FAT is possible (Table 1):

Entry	Value	Entry	Value	Entry	Value	Entry	Value
0000	0 x FF FF FF FF	0001	0 x 02 00 00 00	0002	0 x 05 00 00 00	0003	0 x 04 00 00 00
0004	0 x FF FF FF FF	0005	0 x 06 00 00 00	0006	0 x FF FF FF FF	0007	0 x 00 00 00 00
0008	0 x 00 00 00 00	0009	0 x FF FF FF FF	0010	0 x 00 00 00 00	0011	0 x 00 00 00 00
4092	0 x 00 00 00 00	4093	0 x 00 00 00 00	4094	0 x 00 00 00 00	4095	0 x 00 00 00 00

Table 1: Example File Allocation Table

In Table 1, "File A" occupies the blocks 1, 2, 5 and 6, while "File B" occupies the blocks 3 and 4, and “File C” occupies only block 9. Notice that the entries are laid out in little-endian style, so 0x05000000 simply means the number 5.

2. File List:

- The file list contains 128 items, each of which is 256 bytes long.
- A file item in the list has the following layout:
 - File name: Maximum of 247 characters + the ‘/0’ string delimiter (248 bytes)
 - First block: Location of the first block of the file in the FAT (4 bytes)
 - File size: Size of the file in bytes (4 bytes)

Example: Following the previous example, let “File A” be 2000 bytes long (4 blocks), “File B” be 900 bytes long (2 blocks), and “File C” be 100 bytes long (1 block). Then, the following file list (Table 2) is valid:

Item	File name	First block	File size (Bytes)
000	“File A”	1	2000
001	“File B”	3	900
002	“File C”	9	100
003	NULL	0	0
127	NULL	0	0

Table 2: Example File List

3. Data:

File contents are written in 512-byte chunks.

Core Features:

FS is expected to support the following features:

1. Formatting: Overwrites the disk header with an empty FAT and an empty file list. The user should be able to format a disk from the terminal by typing **`./myfs disk -format`**

2. Writing: Copies a file to the disk. The command **`./myfs disk -write source_file destination_file`** should obtain the `source_file` and write it to the disk with name `destination_file`.

3. Reading: Copies a file from the disk. The command **`./myfs disk -read source_file destination_file`** should get the `source_file` in the disk and copy it to the computer as `destination_file`.

4. Deleting: Deletes a file in the disk. i.e. Entries occupied by the file should be emptied (0x00000000). Command: **`./myfs disk -delete file`**

5. Listing: Prints all visible files (i.e. files whose names do not start with a ‘.’) and their respective sizes in the disk. Alphabetical order is not required. Command: **`./myfs disk -list`**

6. Printing File List: Prints File List to the “filelist.txt” file. The layout of the output will be same as File List in Table 2. All the entries of the file (including empty ones) should be printed. Command: **`./myfs disk -printfilelist`**

7. Printing FAT: Prints FAT to the “fat.txt” file. The layout of the output will be same as FAT in Table 1 (Use tab character to separate columns.). All the entries of the FAT (including empty ones) should be printed. Command: **`./myfs disk -printfat`**

8. Disk Defragmentation: Merges fragmented files into one contiguous space or block. This will allow your system to access files save new ones more efficiently. Command: **`./myfs disk -defragment`**.

*In order to test the correctness of the “Disk Defragmentation” you need to implement “Printing FAT” feature. Without implementing it, you will not get a grade from “Disk Defragmentation”.

Example: Consider the FAT for the first example. After defragmentation, the following FAT will be obtained.

Entry	Value	Entry	Value	Entry	Value	Entry	Value
0000	0 x FF FF FF FF	0001	0 x 02 00 00 00	0002	0 x 03 00 00 00	0003	0 x 04 00 00 00
0004	0 x FF FF FF FF	0005	0 x 06 00 00 00	0006	0 x FF FF FF FF	0007	0 x FF FF FF FF
0008	0 x 00 00 00 00	0009	0 x 00 00 00 00	0010	0 x 00 00 00 00	0011	0 x 00 00 00 00
4092	0 x 00 00 00 00	4093	0 x 00 00 00 00	4094	0 x 00 00 00 00	4095	0 x 00 00 00 00

Table 3: Example Defragmented File Allocation Table

Extra Features: In addition to core features, the following should also be implemented for a full functional FS (You are NOT required to do so) within the scope of the present homework.

- 1. Renaming:** Changes the name of the target file in the disk. Sample command: `./myfs disk -rename file_name new_name`
- 2. Duplicating:** Creates a copy of a file in the disk with a new name. Sample command: `./myfs disk -duplicate file_name new_name`
- 3. Hiding & Unhiding:** Hides or unhides the file in the disk. It should throw an error if the user tries hiding an already hidden file or unhiding an already visible file. Sample commands: `./myfs disk -hide file` and `./myfs disk -unhide file`

Example output:

```

Terminal File Edit View Search Terminal Help
ubuntu@ubuntu:~/Desktop/myfs$ ./myfs disk.image -write Tutorial/unix_introduction.pdf unix_intro.pdf
ubuntu@ubuntu:~/Desktop/myfs$ ./myfs disk.image -write Tutorial/unix_shells.pdf unix_shells.pdf
ubuntu@ubuntu:~/Desktop/myfs$ ./myfs disk.image -write Tutorial/shell_scripts.pdf unix_script.pdf
ubuntu@ubuntu:~/Desktop/myfs$ ./myfs disk.image -write shakespeare.txt shakespeare.txt
ubuntu@ubuntu:~/Desktop/myfs$ ./myfs disk.image -list
file name      file size
unix_intro.pdf 504179
unix_shells.pdf 360288
unix_script.pdf 277028
shakespeare.txt 6141
ubuntu@ubuntu:~/Desktop/myfs$ ./myfs disk.image -delete nonexistent.pdf
file not found
ubuntu@ubuntu:~/Desktop/myfs$ ./myfs disk.image -delete unix_shells.pdf
ubuntu@ubuntu:~/Desktop/myfs$ ./myfs disk.image -list
file name      file size
unix_intro.pdf 504179
unix_script.pdf 277028
shakespeare.txt 6141
ubuntu@ubuntu:~/Desktop/myfs$ ./myfs disk.image -read unix_script.pdf unix_script_output.pdf
ubuntu@ubuntu:~/Desktop/myfs$ ls -l
total 2420
-rw-rw-r-- 1 ubuntu ubuntu 2146304 May  8 01:37 disk.image
-rwxrwxr-x 1 ubuntu ubuntu  17968 May  7 12:55 myfs
-rw-rw-r-- 1 ubuntu ubuntu  10302 May  7 12:52 myfs.c
-rw-rw-r-- 1 ubuntu ubuntu    937 May  6 23:20 myfs.h
drwxrwxr-x 2 ubuntu ubuntu   4096 May  8 01:40 Others
-rw-rw-r-- 1 ubuntu ubuntu   6141 Mar 21 20:37 shakespeare.txt
drwxrwxr-x 2 ubuntu ubuntu   4096 May  8 01:33 Tutorial
-rw-rw-r-- 1 ubuntu ubuntu  277028 May  8 01:39 unix_script_output.pdf
ubuntu@ubuntu:~/Desktop/myfs$

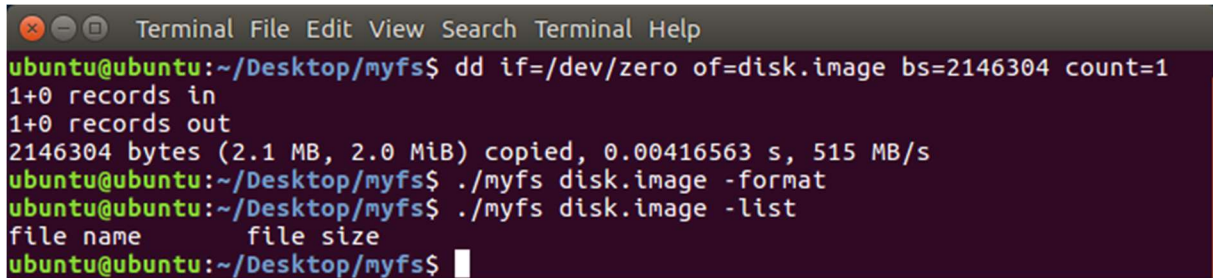
```

Other Specifications:

Your code should be written in C and compiled with GCC (GNU Compiler Collection). You can compile your code in the command line by typing **gcc myfs.c -o myfs -lm**

Hints:

- For **disk** in the terminal commands, you can use either a real drive or a disk image, since there is not any difference between the two in Unix systems.
 - If a disk image is used, the image should pre-exist, and should be formatted before used. To create a disk image, simply enter **dd if=/dev/zero of=disk.image bs=2146304 count=1** in the terminal. This will create a “disk.image” file in the current directory. Then, you can format the image, for example, with the command **./myfs disk.image -format**



```
Terminal File Edit View Search Terminal Help
ubuntu@ubuntu:~/Desktop/myfs$ dd if=/dev/zero of=disk.image bs=2146304 count=1
1+0 records in
1+0 records out
2146304 bytes (2.1 MB, 2.0 MiB) copied, 0.00416563 s, 515 MB/s
ubuntu@ubuntu:~/Desktop/myfs$ ./myfs disk.image -format
ubuntu@ubuntu:~/Desktop/myfs$ ./myfs disk.image -list
file name      file size
ubuntu@ubuntu:~/Desktop/myfs$
```

- While implementing the features for the driver, you may use the following function prototypes if you like.
 - void Format();
 - void Write(char *srcPath, char *destFileName);
 - void Read(char *srcFileName, char *destPath);
 - void Delete(char *filename);
 - void List();
 - void PrintFileList();
 - void PrintFAT();
 - void Defragment();
- While implementing the writing feature, you can search the file allocation table for empty blocks. For each block, use a buffer to transfer data from the local file to the disk. Do not forget to keep the first block information since you will need to store it in the file list.
 - The buffer can be declared as `char buffer[512];`
 - Before writing the file data to the disk, you can use the `fseek` function to adjust the offset of the file position.
- While implementing the reading feature, you can search the file list for the target file name and pick the first entry whose name matches. Then, you can copy the file block by block using a similar buffer.