ECS 150: Project #3 - File system

Prof. Joël Porquet-Lupine

UC Davis, Winter Quarter 2021

Changelog

> *The specifications for this project are subject to change at anytime for additional clarification. Make sure to always refer to the **latest** version.*

- v1: First publication

General information

- Due before **11:59 PM, Friday, March 5th, 2021**.
- You will be working with a partner for this project.
- The reference work environment is the CSIF.

Objectives of the project

The objectives of this programming project are:

- Implementing an entire FAT-based filesystem software stack: from mounting and unmounting a formatted partition, to reading and writing files, and including creating and removing files.
- Understanding how a formatted partition can be emulated in software and using a simple binary file, without low-level access to an actual storage device.
- Learning how to test your code, by writing your own testers and maximizing the test coverage.
- Writing high-quality C code by following established industry standards.

Program description

Introduction

The goal of this project is to implement the support of a very simple file system, **ECS150-FS**. This file system is based on a FAT (File Allocation Table) and supports up to 128 files in a single root directory.

The file system is implemented on top of a virtual disk. This virtual disk is actually a simple binary file that is stored on the "real" file system provided by your computer.

Exactly like real hard drives which are split into sectors, the virtual disk is logically split into blocks. The first software layer involved in the file system implementation is the *block API* and is provided to you. This block API is used to open or close a virtual disk, and read or write entire blocks from it.

Above the block layer, the *FS layer* is in charge of the actual file system management. Through the FS layer, you can mount a virtual disk, list the files that are part of the disk, add or delete new files, read from files or write to files, etc.

Constraints

Your code must be written in C, be compiled with GCC and only use the standard functions provided by the GNU C Library (aka `libc`). *All* the functions provided by the `libc` can be used, but your program cannot be linked to any other external libraries.

Your source code should adopt a sane and consistent coding style and be properly commented when necessary. One good option is to follow the relevant parts of the Linux kernel coding style.

Assessment

This assignment will only be auto-graded.

The ECS150-FS specifications

For this project, the specifications for a very simple file system have been defined: it's the **ECS150-FS** file system.

The layout of ECS150-FS on a disk is composed of four consecutive logical parts:

- The *Superblock* is the very first block of the disk and contains information about the file system (number of blocks, size of the FAT, etc.)
- The *File Allocation Table* is located on one or more blocks, and keeps track of both the free data blocks and the mapping between files and the data blocks holding their content.
- The *Root directory* is in the following block and contains an entry for each file of the file system, defining its name, size and the location of the first data block for this file.
- Finally, all the remaining blocks are *Data blocks* and are used by the content of files.

The size of virtual disk blocks is **4096 bytes**.

> Unless specified otherwise, all values in this specifications are *unsigned* and the order of multi-bytes values is *little-endian*.

Superblock

The superblock is the first block of the file system. Its internal format is:

Offset	Length (bytes)	Description
0x00	8	Signature (must be equal to "ECS150FS")
0x08	2	Total amount of blocks of virtual disk
0x0A	2	Root directory block index

Offset	Length (bytes)	Description
0x0C	2	Data block start index
0x0E	2	Amount of data blocks
0x10	1	Number of blocks for FAT
0x11	4079	Unused/Padding

If one creates a file system with 8192 data blocks, the size of the FAT will be 8192 x 2 = 16384 bytes long, thus spanning 16384 / 4096 = 4 blocks. The root directory block index will therefore be 5, because before it there are the superblock (block index #0) and the FAT (starting at block index #1 and spanning 4 blocks). The data block start index will be 6, because it's located right after the root directory block. The total amount of blocks for such a file system would then be 1 + 4 + 1 + 8192 = 8198.

FAT

The FAT is a flat array, possibly spanning several blocks, which entries are composed of 16-bit unsigned words. There are as many entries as *data blocks* in the disk.

The first entry of the FAT (entry #0) is always invalid and contains the special `FAT_EOC` (*End-of-Chain*) value which is `0xFFFF`. Entries marked as 0 correspond to free data blocks. Entries containing a positive value are part of a chainmap and represent a link to the next block in the chainmap.

Note that although the numbering in the FAT starts at 0, entry contents must be added to the data block start index in order to find the real block number on disk.

The following table shows an example of a FAT containing two files:

- The first file is of length 18,000 bytes (thus spanning 5 data blocks) and is contained in consecutive data blocks (DB#2, DB#3, DB#4, DB#5 and DB#6).
- The second file is of length 5,000 bytes (this spanning 2 data blocks) and its content is fragmented in two non-consecutive data blocks (DB#1 and DB#8).

Each entry in the FAT is 16-bit wide.

FAT index:	0	1	2	3	4	5	6	7	8	9	10	…
Content:	0xFFFF	8	3	4	5	6	0xFFFF	0	0xFFFF	0	0	…

Root directory

The root directory is an array of 128 entries stored in the block following the FAT. Each entry is 32-byte wide and describes a file, according to the following format:

Offset	Length (bytes)	Description
0x00	16	Filename (including NULL character)
0x10	4	Size of the file (in bytes)
0x14	2	Index of the first data block
0x16	10	Unused/Padding

An empty entry is defined by the first character of the entry's filename being equal to the NULL character.

The entry for an empty file, which doesn't have any data blocks, would have its size be 0, and the index of the first data block be `FAT_EOC`.

Continuing the previous example, let's assume that the first file is named "test1" and the second file "test2". Let's also assume that there is an empty file named "test3". The content of the root directory would be:

Filename (16 bytes)	Size (4 bytes)	Index (2 bytes)	Padding (10 bytes)
test1\0	18000	2	xxx
test2\0	5000	1	xxx

Filename (16 bytes)	**Size (4 bytes)**	**Index (2 bytes)**	**Padding (10 bytes)**
test3\0	0	`FAT_EOC`	xxx
\0	xxx	xxx	xxx
…	…	…	…

Formatting program

A FS formatter is provided to you in `/home/cs150jp/public/p3/apps/fs_make.x`. The purpose of this program is to create a new virtual disk and initialize an empty file system on it.

It accepts two arguments: the name of the virtual disk image and the number of data blocks to create:

```
$ ./fs_make.x disk.fs 4096
Created virtual disk 'disk.fs' with '4096' data blocks
```

Note that this formatter can not create a file-system of more than 8192 data blocks (which already makes a 33MB virtual disk).

Reference program and testing

A reference program is provided to you in `/home/cs150jp/public/p3/apps/fs_ref.x`. This program accepts multiple commands (one per run) and allows you to:

- Get some information about a virtual disk: `fs_ref.x info <diskname>`
- List all the files contained in a virtual disk: `fs_ref.x ls <diskname>`
- Etc.
- In order to have the list of commands: `fs_ref.x`

The code of this executable is actually provided to you in `test_fs.c` and you will have to implement the complete API that this program uses. The creation of new virtual disks is the only tasks that you don't have to program yourself as it is provided by `fs_make.x`.

Otherwise your implementation should generate the same output as the reference program, and the manipulations that are performed on the virtual disk should be understood by both the reference program and your implementation. For example, after creating a virtual disk, the output of the command `info` from the reference program should match exactly the output from your implementation:

```
$ ./fs_make.x disk.fs 8192
Creating virtual disk 'disk.fs' with '8192' data blocks
$ ./fs_ref.x info disk.fs > ref_output
$ ./test_fs.x info disk.fs > my_output
$ diff ref_output my_output
$
```

Scripting

The provided tester (`test_fs.c`) can be scripted, which can be particularly useful when testing very specific reading or writing scenarios. Refer to the documentation in `apps/scripts/README.md` to discover how this scripting facility works.

Suggested work phases

Phase 0: Skeleton code

The skeleton code that you are expected to complete is available in `/home/cs150jp/public/p3`. This code already defines most of the prototypes for the functions you must implement, as explained in the following sections.

```
$ cd /home/cs150jp/public/p3
$ tree
.
├── apps
│   ├── fs_make.x
│   ├── fs_ref.x
│   ├── Makefile
│   ├── scripts
│   │   ├── script.example
│   │   └── README.md
│   └── test_fs.c
└── libfs
    ├── disk.c
    ├── disk.h
    ├── fs.c*
    ├── fs.h
    └── Makefile*
```

The code is organized in two parts. In the subdirectory `apps`, there is one test application that you can use (and enhance to your needs).

In the subdirectory `libfs`, there are the files composing the file-system library that you must complete. The files to complete are marked with a star (you should have **no** reason to touch any of the headers which are not marked with a star, even if you think you do…).

Copy the skeleton code to your account.

Phase 1: Mounting/unmounting

In this first phase, you must implement the functions `fs_mount()` and `fs_umount()`.

`fs_mount()` makes the file system contained in the specified virtual disk "ready to be used". You need to open the virtual disk, using the block API, and load the meta-information that is necessary to handle the file system operations described in the following phases. `fs_umount()` makes sure that the virtual disk is properly closed and that all the internal data structures of the FS layer are properly cleaned.

For this phase, you should probably start by defining the data structures corresponding to the blocks containing the meta-information about the file system (superblock, FAT and root directory).

In order to correctly describe these data structures, you will probably need to use the integer types defined in `stdint.h`, such as `int8_t`, `uint8_t`, `uint16_t`, etc. Also, when describing your data structures and in order to avoid the compiler to interfere with their layout, it's always good practice to attach the attribute `packed` to these data structures.

Don't forget that your function `fs_mount()` should perform some error checking in order to verify that the file system has the expected format. For example, the signature of the file system should correspond to the one defined by the specifications, the total amount of block should correspond to what `block_disk_count()` returns, etc.

Once you're able to mount a file system, you can implement the function `fs_info()` which prints some information about the mounted file system and make sure that the output corresponds exactly to the reference program.

It is important to observe that the file system must provide persistent storage. Let's assume that you have created a file system on a virtual disk and mounted it. Then, you create a few files and write some data to them. Finally, you unmount the file system. At this point, all data must be written onto the virtual disk. Another application that mounts the file system at a later point in time must see the previously created files and the data that was written. This means that whenever `fs_umount()` is called, all meta-information and file data must have been written out to disk.

Phase 2: File creation/deletion

In this second phase, you must implement `fs_create()` and `fs_delete()` which add or remove files from the file system.

In order to add a file, you need to find an empty entry in the root directory and fill it out with the proper information. At first, you only need to specify the name of the file and reset the other information since there is no content at this point. The size should be set to 0 and the first index on the data blocks should be set to `FAT_EOC`.

Removing a file is the opposite procedure: the file's entry must be emptied and all the data blocks containing the file's contents must be freed in the FAT.

Once you're able to add and remove files, you can implement the function `fs_ls()` which prints the listing of all the files in the file system. Make sure that the output corresponds exactly to the reference program.

Phase 3: File descriptor operations

In order for applications to manipulate files, the FS API offers functions which are very similar to the Linux file system operations. `fs_open()` opens a file and returns a *file descriptor* which can then be used for subsequent operations on this file (reading, writing, changing the file offset, etc). `fs_close()` closes a file descriptor.

Your library must support a maximum of 32 file descriptors that can be open simultaneously. The same file (i.e. file with the same name) can be opened multiple times, in which case `fs_open()` must return multiple independent file descriptors.

A file descriptor is associated to a file and also contains a *file offset*. The file offset indicates the current reading/writing position in the file. It is implicitly incremented whenever a read or write is performed, or can be explicitly set by `fs_lseek()`.

Finally, the function `fs_stat()` must return the size of the file corresponding to the specified file descriptor. To append to a file, one can, for example, call `fs_lseek(fd, fs_stat(fd));`.

Phase 4: File reading/writing

In the last phase, you must implement `fs_read()` and `fs_write()` which respectively read from and write to a file.

It is advised to start with `fs_read()` which is slightly easier to program. You can use the reference program to write a file in a disk image, which you can then read using your implementation.

For these functions, you will probably need a few helper functions. For example, you will need a function that returns the index of the data block corresponding to the file's offset. For writing, in the case the file has to be extended in size, you will need a function that allocates a new data block and link it at the end of the file's data block chain. Note that the allocation of new blocks should follow the *first-fit* strategy (first block available from the beginning of the FAT).

When reading or writing a certain number of bytes from/to a file, you will also need to deal properly with possible "mismatches" between the file's current offset, the amount of bytes to read/write, the size of blocks, etc.

For example, let's assume a reading operation for which the file's offset is not aligned to the beginning of the block or the amount of bytes to read doesn't span the whole block. You will probably need to read the entire block into a *bounce* buffer first, and then copy only the right amount of bytes from the bounce buffer into the user-supplied buffer.

The same scenario for a writing operation would slightly trickier. You will probably need to first read the block from disk, then modify only the part starting from the file's offset with the user-supplied buffer, before you can finally write the dirty block back to the disk.

These special cases happen mostly for small reading/writing operations, or at the beginning or end of a big operation. In big operations (spanning multiple blocks), offsets and sizes are perfectly aligned for all the middle blocks and the procedure is then quite simple, as blocks can be read or written entirely.

Submission

Content

Your submission should contain:

- The source code of your library in `libfs/`.
- The source code of your tester(s) in `apps/`.
- `AUTHORS.csv`: student ID and email of each partner, one entry per line formatted in CSV (fields are separated with commas). For example:

```
$ cat AUTHORS.csv
00010001,jdupont@ucdavis.edu
00010002,mdurand@ucdavis.edu
$
```

- `libfs/Makefile`: a Makefile that compiles your source code without any errors or warnings (on the CSIF computers), and builds a static library named `libfs.a`.

  The compiler should be run with the options `-Wall -Wextra -Werror`.

  There should also be a `clean` rule that removes generated files and puts the directory back in its original state.

The Makefile should use all the advanced mechanisms presented in class (variables, pattern rules, automatic dependency tracking, etc.)

Your submission should be empty of any clutter files (such as executable files, core dumps, backup files, `.DS_Store` files, and so on).

Gradescope

Gradescope will be opened for submission on Wednesday, March 3rd at 0:00. At that time, you will be able to submit your project as a Git repository.

There should be only one final submission per group, submitted by one of the two partners.

The other partner should be added to the submission as "group member".

Academic integrity

Novelty

You are expected to write this project **from scratch**.

Therefore, you cannot use any existing source code available on the Internet, or even reuse your own code if you took this class before.

Authorship

You are also expected to write this project **yourself**.

Asking anyone someone else to write your code (e.g., a friend, or a "tutor" on a website such as Chegg.com) is not acceptable and will result in severe sanctions.

Sources

You must specify in your report any sources that you have viewed to help you complete this assignment. All of the submissions will be compared with MOSS to determine if students have excessively collaborated, or have used the work of past students.

Violation

Any failure to respect the class rules, both as explained above and in the syllabus, or the UC Davis Code of Conduct will automatically result in the matter being transferred to Student Judicial Affairs.
