Fats Domino File System

Prof. Darrell Long
(largely stolen from Prof. Miller)

Computer Engineering Department
Jack Baskin School of Engineering
University of California, Santa Cruz

Due: 17 March at 1700h

There are NO late submissions on this assignment

Goals

The primary goal for this project is to use the FUSE file system framework to implement a simple FAT-based file system in user space. Unlike the previous two assignments, your code for this assignment runs in user space, though it's tied closely to the OS kernel, since it can be called via *up-calls* from the kernel. thus, it'll give you experience extended an operating system using user code. It'll also give you an idea of how you can quickly implement new file system functionality.

Basics

In this assignment, you'll be implementing a FAT-based file system, largely from scratch, and integrating it into the kernel using the FUSE user-space file system driver. We'll specify the basics for the FAT-based file system; your team should make any other design decisions amongst yourselves, and document both the decisions and the rationales in your design document.

Because your code will run at user level, it should be easier to build and debug, though you *can* still crash the system if you make a mistake.

Before you start

Before you start, please choose a team captain and set up his/her repository so it can be shared. Make sure that, as you go along, everyone pulls the most recent changes made by others, especially if you're modifying related parts of the repository.

Your changes should all be in the `assgn4` branch of the repository. You can switch to this branch using `git checkout assgn4`. Non-kernel files (documentation, test programs, etc.) *must* go under the `assgn4` directory, which is already created for you—make sure you add files to the git repository using `git add`! FreeBSD 10 includes a basic FUSE driver in the kernel, so the only files you'll need to add to your `git` repo are those you write to implement the project. The user-space files (and examples) you'll need are available on Github. Please only `git add` the files you need to compile your project; we suspect that files in `test`, `doc`, and `example` won't be necessary.

If you have any questions about workflow, please discuss them in section or office hours the first week so that they can be worked out.

Details

You're writing a file system to be accessed via FUSE. The file system uses FAT (file allocation table) structures, but is *not* the standard MS-DOS FAT file system. Because you're running in FUSE, your entire disk is contained in a file in the underlying file system, and all your operations (except for formatting the file system) are written as routines to be called by the FUSE library.

Disk structure

Your disk is divided into 4KB blocks, each of which should be read or written in its entirety. You can read or write a sector by using `lseek()` to seek to the appropriate offset (`4096 * block_number`), followed by a `read()` or `write()` of 4096 bytes. The blocks are laid out as follows:

Block	Content
0	Superblock
1–*k*	File allocation table
k+1 and higher	Data blocks

Assume that your file system has a total of *N* blocks. The first one is reserved for the superblock. Since each block pointer in the FAT is 32 bits (4 bytes), each FAT block can hold 1024 block pointers, so you need `N/1024` FAT blocks (but obviously at least one). The first *k*+1 FAT pointers (0–*k*) are unused, since they point to disk blocks that can't be included in files. The remainder of the blocks (*k*+1 and higher) are available for use in files and directories.

Superblock

Word index	Content
0	Magic number: 0xfadedbee
1	*N*: total number of blocks in the file system
2	*k*: number of blocks in the file allocation table
3	Block size (4096 for your file system)
4	Starting block of the root directory

These are all stored in 32-bit words. None of these values should change over time, but rewrite the superblock if they do. Also, we recommend starting your root directory in block *k*+1, but this isn't a requirement.

The file allocation table is an array of (signed) 4-byte block numbers. A 0 in the FAT means that a block is free, and a −2 means that the block is the last one in the file. Otherwise, the entry in the file allocation is the block number of the *next* block in the file. As noted above, the first *k*+1 entries of the FAT are invalid, and should be set to −1 so they're never allocated.

You can also use −1 to indicate other FAT entries that should never be allocated. Please see your lecture notes or the book for further details on the FAT.

Directories

Directory entries in this file system are 64 bytes long, and contain the following information:

Word index	Content
0–5	File name (null-terminated string, up to 24 bytes long including the null)
6–7	Creation time (64-bit integer)
8–9	Modification time (64-bit integer)
10–11	Access time (64-bit integer)
12	File length in bytes
13	Start block
14	Flags
15	Unused

There's no user ID in the file system; you're not implementing protection information. If you must provide it, return a reasonable default such as owned by `root`, all have `rwx` permissions. And, yes, this leaves 4 bytes of free space in the directory entry in case we (or you) forgot something. If the directory entry is free, the starting block field should be set to *zero*. This allows you to ignore entries past the end of a directory but still in a block that's part of the directory file, which is particularly useful for the root directory. It also allows you to have holes in a directory so you don't have to compact it when a file is deleted. As with the superblock, the fields in the directory entry *must* appear in the above order.

When you create a directory, make sure you create a (`..`) entry that points to the directory's parent directory. This entry can't be deleted, though the directory itself may be removed if it's the only entry in the directory.

The only flag you must maintain is the DIRECTORY flag (bit 0 of the `flags` field in the directory entry), a single bit that indicates whether the directory entry is for a regular file or a directory. A 0 means it's a regular file; a 1 means it's a directory.

Supported operations

You need to support all the basic operations: file creation, directory traversal, file reads, writes and seeks, *etc*. When the operation asks for information you're not maintaining (*e.g.* permissions), return reasonable and consistent values. If asked for an operation beyond the basic ones that you support, your file system should return an error.

You don't need to support hard or soft links, so you may return an error code if the FUSE driver asks for a hard or soft link. As a result, `unlink` always removes a file if it exists.

Your FUSE file system should support the operation of such programs as `cat`, `mkdir`, and `ls`. You shouldn't modify these programs; they must work unmodified on your file system.

Your file system *should* track creation, modification, and access times for individual files. This means that you need to update the time stamps for each file when the file is created, modified, or accessed. Note that this requires you to keep track of open files and keep a copy of their directory blocks in a cache to quickly update them if needed. You may also want to keep the superblock and entire FAT in a cache, again to speed access to them.

Multi-threading

To keep things simple, you may use a single-threaded implementation of your file system. If you choose to implement a multi-threaded version, you can get 10 extra credit points, but only if the project achieves an 85 score without the extra credit. Note that multi-threading means you must be careful about synchronization between multiple threads that could be trying to modify the same files and other structures in the file system.

Deliverables

The deliverables for this project are, as usual, the design document and other documentation you've written, source code to implement the file system, and any source code you might have written for testing purposes. Don't submit raw data files, such as a sample disk image. Since you're not modifying the kernel for this assignment, we expect all of your files to go in the `assgn4` directory.

As usual, you'll commit your files using `git`; please read the `git` shortcuts page for details, including instructions on how to share a repository among your team.

Testing your project

You're implementing a file system, so tests should exercise file system functions. There's no need to write specific programs to test your file system; you can use a script (executable or a set of commands that a human types) to see if the file system works. Either way, document what you did and how you tested the file system.

Hints

- ***START EARLY!*** Meet with your group ASAP to discuss your plan and write up your design document. design, and check it over with the course staff.
- Experiment! You're running in an emulated system—you *can't* crash the whole computer (and if you can, let us know).
- Look at the examples provided with the FUSE download, and see how they work. Understand how your file system must work *before* implementing anything. Knowing which calls you need to support and how you're going to supporting them is probably the most difficult part of the assignment.
- Get your design done *before* you write a single line of code. It's especially critical here, since you need to make sure that your disk layout includes everything it needs to.
- Write a user-space program, not running through FUSE, to initialize a brand-new file system. It should take two arguments, the name of the file to contain the disk image, and the number of blocks in the file system, and then create the disk image. Do this first, before writing any FUSE code.
- Since the file system is user-space code, build a test harness outside of FUSE to test individual routines. Write a program that simulates the FUSE harness's calls to your code, so you don't need to rebuild FUSE for testing. This also avoids the problem that bugs in your user-space FUSE code *can* crash the kernel, typically by causing hangs.

- Include a plan for which routines you're going to get working first. You need to have directory-based routines working before user-initiated read and write, so get them working first.

This project requires significantly more coding than Assignments 2 and 3, but there's no long kernel recompile, so it should be faster. There are plenty of examples online of how to code FUSE file systems; be sure you cite any examples you use. Keep in mind, too, that the code you write must be your own. If you copy code from the Internet and cite it, that part of the code will receive no credit. If you copy code from the Internet and *don't* cite, we will file academic honesty charges against your group.