Assignment 4

Due: 13th March at 23:59

Goals

The primary goal for this project is to use the FUSE file system framework to implement a simple 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 on extending 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 file system---that we will call the **All-in-One Folder File System** (**AOFS)**---largely from scratch, and integrating it into the kernel using the FUSE user-space file system driver. We'll specify the basics for the AOFS 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 `asgn4` folder of the repository. Files (documentation, test programs, etc.) *must* go under the `asgn4` directory, make sure you add files to the git repository using `git add`. FreeBSD 11 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. **(Note that you should use fuse 2.9.5 instead of the most recent one as the most recent one might not work properly in FreeBSD 11)** 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. Because you're running in FUSE, **your entire disk is contained in a file in the underlying file system (this will be referred to as the FS_FILE in the rest of this document)**, and all your operations (except for formatting the file system) are written as routines to be called by the FUSE library.

Functionality 0: The basic structure of the File system

The AOFS starts as an empty filesystem. AOFS does not have directories (except the root directory), it only has files. The filesystem will be divided to 4KB blocks. Initially, the FS_FILE will have the following structure:

(1) A superblock. This superblock will have a magic number 0xfa19283e followed by a bitmap of free blocks. In the bitmap, each bit corresponds to a 4KB block in the filesystem. A bit is 0 if the block is free and 1 if it is occupied.
(2) After, you will have your files. Initially, the space after the superblock is empty.

Functionality 1: Adding files

When a request to add a file to the filesystem is received, it is inserted to the FS_FILE (by finding the first free block using the bitmap). For example, after a request to create a new file (F1) is received, your FS_FILE will become:

(1) A superblock.
(2) A 4KB block for the new file F1. This will be divided to two parts, the first part is meta-information about the file (for example, the time it was created, the file size, etc). You should decide what meta-information you should maintain, so that you can implement the filesystem system calls. The second part is the actual contents of the file F1. So when the user write to the file F1, it should be reflected in the second part.

Now, if another file is created, it is appended after the last appended file. So, if a user requests creating a file F2, the following will be the FS_FILE structure:

(1) A superblock.
(2) A 4KB block for file F1.
(3) A 4KB block for file F2.

Functionality 2: Writing and reading to files

Your filesystem should enable read and writing to files in FS_FILE. This include the ability to find these files (for example by the user running the command `ls`). This also includes maintaining and reporting meta-information of files, such as when they are created and others.

Functionality 3: Writing to files beyond 4KB

The other functionality that you have to implement is extending the file size beyond one block. You will implement this by appending a new block and connecting it to the block(s) of the file. For example, assume that the user wrote more than 4KB in file F1, you append another block to make your FS_FILE look like the following:

(1) A superblock.
(2) A 4KB block for file F1. (this will be the first part of file F1)
(3) A 4KB block for File F2.
(4) A 4KB block for file F1. (this will be the second part of file F1)

Functionality 4: Removing files

Your AOFS should support removing files. If a user removes file F2, for example, the FS_FILE structure will become:

(1) A superblock.
(2) A 4KB block for file F1. (this will be the first part of file F1)
(3) A free 4KB block.
(4) A 4KB block for file F1. (this will be the second part of file F1)

(Un)supported operations

You need to at least support all the following basic operations: file creation/deletion, file reads, writes and seeks. 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`, 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.

Multi-threading

To keep things simple, you may use a single-threaded implementation of your 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. The documentation should include a comparison of the two cases. 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 `asgn4` directory. Write, in the README file, how to run your filesystem.

As usual, you'll commit your files using `git`. The following is about the banchmark and benchmark report.

Benchmark and benchmark report

You should write a benchmark code to test the performance of your file system compared to the underlying operating system. This can be a simple benchmark to test the time needed to do various operations such as (1) creating 100 files, (2) writing to files, (3) reading from files.

Both the benchmark code AND the benchmark report must be part of the deliverables. Also, the graders should be able to run your benchmark.

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 support 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.
- 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.

More about installing FUSE

Do the following steps to install and run a simple hello filesystem with FUSE:

$ cd /usr/ports/devel/pkgconf

$ sudo make install

Next, install the FUSE Library:

$ sudo pkg install fusefs-libs

This will install the FUSE header files and libraries on your system.

Header files are installed at: /usr/local/include/

Library files are installed at: /usr/local/lib/

If at this point you run:

$ pkgconf fuse --cflags --libs

You should see "-I/usr/local/include/fuse -D_FILE_OFFSET_BITS=64 -L/usr/local/lib -lfuse -pthread" as the output.

Please note that the version of the FUSE library this will install is 2.9.5. This means if you use FUSE examples which are written with FUSE 3.X.X, then those examples will fail to compile still. To acquire examples from this version of the library, download this package and extract it: libfuse-2.9.5

Download FUSE repo using the following directions:

$ fetch --no-verify-peer https://github.com/libfuse/libfuse/releases/download/fuse_2_9_5/fuse-2.9.5.tar.gz

Unzip as follows

$ gunzip fuse-2.9.5.tar.gz

$ tar -xvf fuse-2.9.5.tar

Within the folder, you will find another folder for 'examples'. There is a file called 'hello.c' and this is the only file you need from this entire library folder, as the actual library was installed through the 'pkg install' command earlier.

$ cp ./fuse-2.9.5/examples/hello.c .

Now try to compile this simple hello.c example.

Firstly, copy the hello.c to your home directory. You may delete the folder now if you wish to.

Then, run these commands:

$ cd ~

$ sudo kldload fuse

$ cc hello.c -o hello `pkgconf fuse --cflags --libs`

Note that those are ticks (character above the TAB key) in the 'cc' command, not single quotes.

Then, you run the new filesystem on a new folder newHelloFS

$ mkdir newHelloFS

$ sudo ./hello newHelloFS

You need to run the './hello/ program as sudo because you are technically mounting a file system and that requires root permissions

Now, if you go inside newHelloFS folder, it is implemented using the functions in hello.c. You will notice you cannot do a lot of things, only read the file hello. You can use this hello.c and extend it to implement AOFS.