Operating Systems : Assignment 5 - Filesystem

Milestones (see Canvas for due dates)

0. **Milestone 0: Design (Submitted to Canvas as PDF)**
1. **Milestone 1: Basic Building Blocks**
2. **Milestone 2: Directory Traversal, Inodes, and File Descriptors**
3. **Milestone 3: Data management**

1. Introduction:

The task for this project is to use the block storage library created in Assignment 4 as a logical storage device, on top of which you are now going to build the **Filesystem (FS)**. **FS** will closely model the original Unix Filesystem and the Unix Fast Filesystem.

Please review your class notes. Additionally, these references may be useful:

- https://www.cs.berkeley.edu/~brewer/cs262/FFS.pdf
- http://en.wikipedia.org/wiki/Extended_file_system
- http://www.cs.cornell.edu/courses/cs6410/2010fa/lectures/04-filesystems.pdf
- https://en.wikipedia.org/wiki/Ext2

2. Filesystem Design:

Files and filesystems are designed to operate using an abstraction of the underlying storage hardware. You will implement a filesystem that interfaces to a ***Block Store*** similar to Assignment 4. Recall the *Block Store* library specifications; specifically, the purpose of the library is to simulate a block storage device that provides access to storage blocks based on a block index.

The following are the sizing constraints of the *Block Store* (note these sizes differ from the block store assignment):

1. The total number of blocks is 2^16.
2. The size of blocks is fixed at 4096 bytes.
3. Some of these blocks are pre-allocated to the free block map (as before).

The following are sizing constraints with regard to the file system itself. In particular, these constraints should allow you to pass the write_full_fill() test.

1. Any overhead (meta-data) for your file system should be stored within 5 blocks.
2. If your overhead takes less than 5 blocks, say 4, then waste the additional block.

3. Milestones:

This project will be divided into four submissions:

0. **Milestone 0 (Design)**,
1. **Milestone 1**,
2. **Milestone 2**, and
3. **Milestone 3**

All milestones after milestone 1 will provide code for the previous milestone for you to use as you wish. As such, **no late submissions will be accepted**.

Each milestone also requires pseudocode for the next milestone. The pseudocode should be submitted as **comments** in the appropriate functions.

Milestone 0 – Design:

A failure to plan is a plan to fail. This will be a large project; understanding the basic concepts, planning your structures, functions, and what issues can occur is **vital** to any programming project.

You are to draft pictorial representations of (drawings on paper and scanned/photographed are sufficient):

1. Inodes and their block references
2. The filesystem layout, and how its layout relative to Back Store.

You are to put together your proposed layout (read: structure definitions) for:

1. FS objects
2. Inodes
3. Directory files

You are to write pseudocode for the following problems given the specified information and list potential errors that can occur:

1. File/Directory Creation given an absolute path to the file/directory and a flag indicating file or directory
2. File Writing given a **file descriptor,** a buffer, and a byte count
3. File/Directory Deletion given an absolute path to the file/directory

Once you have completed your draft designs, structure definitions, and pseudocode; collect them into a PDF document and submit them to Canvas by the M0 due date.

Milestone 1 - Basic Building Blocks:

You are to implement formatting, mounting, and unmounting (`fs_format`, `fs_mount`, and `fs_unmount`) of your FS filesystem. You will also prepare pseudocode for the following milestone.

Once you have completed your implementation, be sure to commit all changes and push to master branch on OSGit by the M1 due date.

Milestone 2 - Directory Traversal, Inodes, and File Descriptors:

You are to implement the creation of files and directories, the opening and closing of files, and the enumeration of directories (`fs_create`, `fs_open`, `fs_close`, and `fs_get_dir`). You will also prepare pseudocode for the following milestone.

Once you have completed your implementation, be sure to commit all changes and push to master branch on OSGit by the M2 due date.

Milestone 3 - Data management:

You are to implement the reading and writing of data to a file, the seeking of file descriptors, and the deletion of files as well as empty directories (`fs_read`, `fs_write`, `fs_seek`, and `fs_remove`). All students will also need to submit implementation for the moving files and directories as well as the creation of hardlinks (`fs_move` and `fs_link`). It is suggested that you start early on these features, as you will need to consider these operations while implementing Milestone 2.

Once you have completed your implementation, be sure to commit all changes and push to master branch on OSGit by the M3 due date.

4. Filesystem Specification, Implementation, and API:

Your implementation of the Filesystem must be a CMake system library project. A header with the expected functions and operation details will be located in your repository. Please refer to it for implementation details and bring any additional questions to the Canvas LMS.

Specifications:

Your FS implementation must be capable of:

- Format an FS file
- Mount and unmount an FS file
- Create directories and regular files
- Open, close, and seek files
- Read and write to files
- Remove files
- List directory contents
- Move files and directories
- Create hardlinks

The FS filesystem will incorporate file descriptors as an intermediary for operating on files, and they will be used much like they are in a POSIX-compliant OS.

Structuring the FS

Regarding inodes:

Each inode will be exactly 64 bytes. The general inode structure is as follows:

Metadata / ACL: bytes 0-47
Data Block Pointers: bytes 48-63

The inode table will be 16 KB in size, i.e., 4 data blocks worth of inodes. Inodes will use 16-bit addressing for block references. Your filesystem root directory must be located at the first inode.

Regarding block pointers:

Inodes contain 8 block pointers: 6 direct, 1 indirect, and 1 double indirect. Direct pointers, as the name suggests, are the id numbers of the data blocks that make up the file. Indirect pointers point to a data block that is entirely made of direct pointers. A double indirect pointer points to a block of data that is made entirely of indirect pointers. Indirect and double indirect pointers greatly increase the maximum size of a file while the direct pointers allow for quick and simple access to small files.

Regular File Block Structure:

The regular file blocks have no structure, they are pure data. Be sure to keep track of the file's size somewhere in your inode metadata!

Directory File Block Structure:

To simplify the filesystem, directory files will be limited to 31 entries. Each entry will consist of:

1. The file's name: 127 bytes
2. The file's inode number: 1 byte

This allows a directory to be contained on a single data block. The remaining space on the directory's data block should be allocated to Metadata / ACL as you see fit.

Note that creating an empty directory requires only an inode. However, as soon as a file (or subdirectory for that matter) is created within that directory, it will be necessary to allocate space for the directory block. This is important in order to assure that you pass all of the tests correctly.

File Descriptors:
The FS filesystem will be using file descriptors to manage open files. The filesystem will be limited to **256 file descriptors**. Each file descriptor will track a single read/write position for that descriptor. How you manage this is up to you. Files should be able to be opened multiple times, meaning your implementation should support multiple descriptors to the same file, but with different read/write positions. Seeking allows the user to move the read/write position to where they want. Be sure to read the header for additional notes or requirements on how a function affects a file descriptor.

Hardlinks:
Hard links allow for multiple references to the same file on a disk, typically in a way that is invisible to the user. In FS, this is achieved by decoupling the file's name from the file's contents. With FS, there is no differentiation between the original file and the hardlinked file (i.e. there is no "original"). A hardlinked file must have all links removed before the file will have its resources released. An inode cannot be referenced more than **255** times.

Testing:
Your implementation should be capable of doing all operations listed above cleanly and safely. Like all projects, a tester will be provided to test the frontend of your implementation. You will be expected to deliver a functional and correct solution for each milestone. Be sure to be comfortable with various debugging and file inspection tools, and they will prove vital to writing a correct implementation.

5. Rubric:

Milestone 0 (Design)	25 points
Milestone 1	25 points
Milestone 2	50 points
Milestone 3	50 points
Total	150 points

For all students, for all milestones:

Insufficient Parameter Validation: up to -25% of rubric score

Insufficient Error Checking: up to -25% of rubric score

Insufficient Comments: up to -25% of rubric score

Incorrect implementation: up to -50% of rubric score

Memory Leaks: up to -20% of rubric score

Submission compiles with warnings: -90% of rubric score

Submission does not compile, or refuses to build: -100% of rubric score