File System Assignment

Introduction

The last project is to build a simplified version of the original Unix file system, called PseudoFS. There will be four components to the system:

1. Shell: A simple menu driven application that allows the user to carry out various operations on the PseudoFS. These tasks include printing diagnostic information about the file system, formatting a new file system, mounting and unmounting a file system, creating and deleting files, and copying data in and out of the pseudoFS from your computer's hard drive. To achieve this the shell commands or menu options will translate these commands into the file system API calls. The shell in this case essentially accesses the pseudo file system and hard drive as though it was in real mode and acts as the kernel.

   Shell commands
   - create: Create a new file to simulate a hard drive
   - format: Format a file that has already been created with the Pseudo File System
   - mount: Mount an already formatted pseudoFS drive. For simplicity, only mount one disk at a time
   - unmount: Unmount a mounted pseudoFS drive. Ensure any cached data is updated
   - debug: Print information about the superblock, inodes, and blocks on the mounted file system
   - delete <name>: Delete a file with the given name. Don't allow deleting the / directory
   - cat <name>: Print the contents of the file name to the screen
   - ls: Print list of files and inode data for files stored in the pseudoFS file system
   - copyin <file>: Copy a file from your computer's hard drive into the pseudoFS file system
   - copyout <file>: Copy a file from the pseudoFS file system to your computer's hard drive
   - help: Display a help menu explaining how to use the shell
   - exit: Seriously? I need to explain this? ☺
2. File System: The file system component takes the commands from the user via the shell and performs them on the PseudoFS disk image. The file system is responsible for handling the on-disk data structures and all record keeping needed to allow for persistent storage of data. The first block will be the superblock. The next 10% of the blocks will be used by the disk inodes. The remaining 90% of the blocks will be used to store data. To store the data, the file system will need to work with the disk emulator through its API calls. The pseudoFS will format the file system, read, and write data in 1024 byte blocks.

   API:
   - void diagnostics(Disk disk) // used by the shell debug option
     - Scan a disk and print a report on the file system, including:
       - Filesystem
         - Magic number is valid/invalid
         - # of reads during this mount
         - # of writes during this mount

        - Inodes
            - Total # of inodes on filesystem
            - # or % of used and free inodes
            - # of writes during
        - Blocks
            - Total # of blocks on filesystem
            - # or % of used and free blocks
    - bool create(File file)
        - Create a file to act as a disk emulator. Write out the correct number of blank lines to simulate the disk size and save the file
    - bool format(File file)
        - Write a superblock and inode blocks to the file, and empty all data blocks. Count the line numbers of the file to calculate the correct size of the filesystem. This destroys any data that may have existed on the filesystem prior.
        - Return true if successful, false otherwise
    - bool mount(File file)
        - Open a file and associate it with the instance of the file system in a Disk structure
        - Scan the entire disk and build the list of free/used inodes and free/used blocks for the superblock
    - bool unmount()
        - Write all unsaved data and close the disk file so it is no longer in use
    - inode readInode(int inodeNumber)
    - bool writeInode(int inodeNumber, inode updatedNode)
    - inode getFreeInode()
    - Block readBlock(int blockID)
    - bool writeBlock(int blockID, Block block)
    - Block getFreeBlock()

3. Disk Emulator: The lowest component emulates a disk by dividing a normal file (called a disk image) into 1024 byte blocks and only allows the File System (3rd component) to read and write in terms of complete blocks.
   The emulator will persistently store data to the disk image using the Rust standard IO library. The disk image will be a plain text file, where each line of the file will serve as one block and therefore must store exactly 1024 bytes of data. The disk size will be determined when creating the disk image and defines the number of lines in the file.
   API:
    - bool open(File file)
        - Open the disk parameter. If successful and the superblock is valid, the file system is mounted and the superblock instance is available. The superblock instance stored in memory after reading from block 0 should have a list of all free inodes and blocks from the disk (see design)
    - bool close(Disk disk)
        - Close the disk. All data write operations must be completed. If successful, the file system is unmounted.

- Block read(Disk disk, int blockID)
    - Read a 1KB block of data from the file and return an instance of Block with the file data from the block
- bool write(Disk disk, int blockID, Block block)
    - Write the block parameter to the given block id on the given disk. Return true if successful and false if not

Pseudo File System Design

To implement the file system component, first consider how formatting the disk will lay out the file system structure. Each disk block will store 1KB of data. The first block in the file system is the superblock which describes the layout of the rest of the file system. A certain number of blocks after the superblock are used to store the inode table (10% of the total disk blocks are typically used to store inodes), and the remaining blocks are used as direct or indirect data blocks for data storage. One block of the inode table will contain several inodes. To have 10% as inodes, there will be multiple blocks reserved for the inode table (block 1 to whatever 10% of the disk is).

The Super Block

- Magic number: A magic number which serves as a file system signature. pseudoFS' magic number will always be 0x70736575646F4653. This is placed in the superblock when the file system is formatted, and the number is validated when the file system is mounted. If the number is invalid, the file system cannot be mounted.
- Number of total blocks on the system (equals the number of lines in the file)
- List or bitmap of all free blocks on the system (created by scanning the file when mounted. Updated when files are created/deleted/modified)
- Number of total inodes on the system (calculated as 10% of total disk space. Validate when mounting)
- List or bitmap of all free inodes on the system (created by scanning the file when mounted. Updated when files are created/deleted)

The list of all free blocks on the system and all free inodes will be generated dynamically when the file system is mounted, and will not be saved to disk. This is a departure from the classic Unix File System approach but should be simpler to operate with. You will need to scan the file for the inode blocks and the data blocks to achieve this when you mount the file.

The inode

- number: The inode number. Also the index of the inode in the inode table.
- type: Could be an enum or an int. Types are 0/free, 1/file, 2/directory, 3/symlink
- startBlock: The first block (line number) of the file contents, representing the head of the linked list
- size: The total size of the data in the file in KB (equals the number of data blocks written)
- cTime: The date/time the file was created
- Fields the original Unix File System also had but aren't required for this would include userid, groupid for owners, permissions (rwxr-xr--, etc.), lock, access time, modified time

  - Userid, groupid, permissions are self-explanatory from the Linux class and would require a user struct and a group struct
  - Lock is a flag indicating a process has the file open. Since we only have a shell in this app operating in real mode we don't need this but you could store the process ID of the process, or -1 if the file is not locked (process id 0 is typically the kernel in Linux/Unix).

Inodes are stored on data blocks, but those blocks occupy a reserved portion of the hard drive. You'll want to serialize them in some fashion and write out an array of them to the reserved inode table on the file system (the first 10% of the file system right after the super block). When the file system is mounted you'll want to deserialize them back to struct instances. If they change, be sure to update them in the file system (serialize and re-write to the disk) not just in memory. Inode 1 will always represent the / directory file. The super block doesn't have an inode associated with it, and only your shell can work with the superblock by reading/writing block 0 directly.

The data block

- nextNode: This will be the line number of the next block of data in the file. The last block should have an invalid number such as -1 to indicate it is the last block in the file. The last block will most likely not have a complete data payload. I would suggest making this a fixed size string representing the next block (line) number with leading zeros if necessary, and using a substring of a set number of characters (0 to the length of this field). If you try to use a delimiter such as a : or , to separate these two fields on the disk, it's highly likely that the data in the block may contain this field as well and cause problems. Another approach might be to serialize the block as a single line of JSON.
- data: This will be the string representing this chunk, or block of the file contents. This will involve dividing up the file into block size chunks and saving them when writing, then reading them back and reassembling them into a contiguous string representing the complete file contents when reading the file back off the disk.

Thoughts: You could serialize these two structures to JSON and write the JSON out as a single line, then read it back and deserialize it into your data structure instances. Another approach which would take less disk space would be to format the fields as fixed-length strings (add leading 0's to int's, for instance with string format) and have a CSV style line format and simply use string split to get the fields. More data would fit in the 1KB file line this way.

You can choose from different approaches. One approach is to serialize a block to JSON and have two fields. First field is the next node number, and the second field would be the block data. This will take more space on the block for JSON formatting so less data can fit in the data field, but will be easier to serialize/deserialize.
An example might look like { "nextNode": "123", "data": "asdfa09oirkln,.ml…"}

Another approach might use a fixed-length leading field for the next node number, and the rest of the line would serve as the block data.
An example might look like 0000123asdfa09oirkln,.ml…
In this example the first seven (or however many characters are needed) lines are the next block number and the remainder of the line is the block file data.

If you want to use a different approach you are welcome to, just remember the linked list approach requires that the next node address and the data all fit on one block.

The pseudoFile

This will represent the raw data of a file on the pseudoFS before writing the file to various blocks or after reading the file back from the blocks of storage. This will be used when doing the copyin, copyout, and cat operations of the shell. The content of the file will be essentially one large string with all of the file's data in it.

The directory

Directories are really special operating system files, and act like a database table in many ways. The records consist of an inode number and a name, and this is what the OS uses to map contents into a directory. For the pseudoFS, you only need one directory, the / which will be given inode number 1. All other files will be stored in the / directory. The format should basically be the inode number, a colon, and the inode name for each record. Records can be separated by a comma, such that the contents of the file will look like this:

2:Picture.jpg, 3:Song.mp3

You can read the contents of this file into a string and split it on the comma to get each inode/file name record, and then split each record on the colon. You can then read these into a map using the name as the key and the inode number as the value to provide a friendly way of referencing files by name instead of by inode number. You are welcome to serialize/deserialize this data in a different fashion (JSON, etc.).

The FileSystem

The file system will represent the structured data stored on a disk. One line in the disk represents one 1KB block of data. The first block is used by the superblock always. The next 10% of the file system is used to store inodes. The remainder is used to store data. A file system is associated with one disk and will take up the entire disk (only one partition per disk).

The Disk

- File file
- int blocks
- long reads
- long writes
- bool mounted

A disk will be how the file system interacts with the underlying file. A disk instance is created when the file is mounted and will open the file for read/write. The file will stay opened through the disk structure as long as the file system is mounted. The fields in the disk structure listed here can be added to the diagnostics output to show the number of block reads/writes (otherwise they aren't needed).

I would recommend using the seek operation of the file handle to move the line pointer around on the file while the disk is mounted I order to get directly to the line/block you want to work with.

Bonus

As if this wasn't challenging enough, if you get all of this done and want to go a step further, bonus will be offered for the following additional tasks:

- Add aTime and mTime to the inode and enable their use. Add their output to ls
- User/group ownership and permissions. Add an su <username> command to the shell and enforce user/group permissions on files. Also add a chown and chmod command to the shell and on cat enforce the permissions and update the ls command to show ownership and permissions
- Add access control lists to the standard permissions, allowing multiple users/groups their own rwx permissions
- Add in-line encryption so that the shell prompts for the password when started and if valid decrypts each file or block as it is read off the disk and encrypts each file or block as it is written
- Instead of using a linked list, redesign the file system to use direct and indirect blocks
- Consider what it would take to implement RAID 0 (striping), RAID 1 (mirroring) and RAID 5 (parity). Implement them
- Research data deduplication on a hard drive. If you implement this, I'll give you a degree immediately no other questions asked! (I can't really do that, but you would deserve it!)