Write-Once File System

Names: Vennela Chava(vc494), Sai Laxman Jagarlamudi(sj1018)

Objective

To build a File System that supports unmount(), mount(), open(), read(), write(), and close() calls using a 4MB file in the native Operating System.

Design

We reduced metadata(inode information, superblock structure) space as much as possible in order to increase the actual available space for data. To reduce metadata space as much as possible, we employed a minimal structure for inode and superblock. We also addressed improper use of unmount(), mount(), open(), read(), write(), and close() calls and returned their respective diagnostic error messages.

Superblock and Inode Structure, Data blocks

Superblock, which includes the inode index, inode size, data index, and File descriptor table, is an essential component of the File system. The Superblock's information is crucial to the file system's ability to function effectively. Since the superblock is the first block in our disk, which makes it is easy to access our file system using a single pointer.

Inode includes the name of the file, size of the file, first data block of the file, number of blocks in a file, if the file is used or not flag, and file descriptors. It is used to describe files, directories, and corresponding meta information.

The blocks of the disk used to contain the actual content of files are known as data blocks. A file may span numerous blocks depending on its size since each virtual disk block has a size of 1024 bytes. Even if a block has spare space, a file that is smaller than the size of the block will still occupy it, which could lead to internal fragmentation on the virtual disk. Data blocks are an essential part of the file system, as they provide the actual storage for the files on the disk.

mount()

Mount() is in charge of initializing and preparing the file system for use. When this function is invoked, It will attempt to read a complete "disk file". If it is a success, it will return 0, and any other value is thrown as an error. It sets errno and returns the error code it received in the event of a file access problem. Format checks are taken care of in this mount() function. If the mount discovers that the "disk" is empty, it creates a preliminary structure(superblock initialization) required for accessing the "disk". It reports the issue and releases built-in structures if it discovers that the "disk" format is flawed. The possible diagnostic error messages scenarios

- **Case1:** Checks if the file name is NULL or not (invalid Filename)
- **Case2**: Invalid Superblock location

unmount()

Unmount() is in charge of writing the file system's modifications back to the virtual disk. Additionally, any data structures that were utilized during file system activities are cleaned up and appropriately closed. The most recent file system changes are updated onto the virtual drive using this process. When this function is invoked, the whole memory-to-disk information is transferred in blocks of 1KB back to a disk file. It writes a complete "disk file" while on call. If it is successful, it returns a 0, and any other value is an error. It sets errno and returns the error code it received in the event of a file access problem. After the changes have been written to the virtual disk, the wo_umount function releases the memory allotted for the file system's logical components by calling the free function on the memory addresses where the superblock, Inode information. This ensures that the file system is set up for the following round of operations by resetting the file descriptor table and marking all of the entries as empty.

open()

open() is in charge of opening files and assigning them file descriptors. This enables other file system functions to access and work with the file. In order to confirm that the file is present and that file descriptors can still be assigned, the program first runs some error checking. The function will return an error if one or both of these requirements are not satisfied. A boolean flag is then set to indicate that the file descriptor is now in use, allowing access to and manipulation of the file using the

file descriptor. This file descriptor is then associated with the file. If the maximum number of file descriptors is not reached, it also permits opening the same file more than once, each time with a different independent file descriptor. Open() functions checks If you aren't in file creation mode when you open, try to find the file. Set errno and return the appropriate error code if it doesn't exist. In the other case, look into permissions. Set the err-no variable and return the appropriate error code if they are incompatible with the request.

read()

read() is a file system's function for reading data from files. The file descriptor, a buffer in which the read data will be saved, and the number of bytes to read are its three required arguments. In order to find the correct root directory and the file itself, the function first locates the filename associated with the file descriptor in the file descriptor table. The function must decide how much data to read after locating the file. The function will only read the data that is available, up to the size of the file, if the requested number of bytes to read exceeds the size of the file. If not, it will read the number of bytes that the user or application specifies. The function then calculates how many blocks should be read from the file by dividing the quantity of data to be read by the size of a file system block (approximately 512 bytes). In order to guarantee that all of the requested data is returned, the function rounds up to the nearest whole number if the result is not an integer. The function chooses the first block to read from after determining the total number of blocks to be read. This is accomplished by dividing the size of a block in the file system by the current offset of the file (as provided by the file descriptor). This will indicate how many blocks must be skipped before the reading itself can start. The function then decides where to start within the starting block. To do this, divide the remaining value of the file's current offset by the size of a block in the file system. This will indicate where the reading will start and where in the starting block it will be. A "bounce buffer," which is a transient buffer that stores one block of data at a time, is used to do this. The function replicates the desired data from the bounce buffer into the user-supplied buffer by reading one block of data at a time into the bounce buffer. Up until all the desired data has been read, this procedure is repeated.

write()

A file system function that publishes data to a file. The file descriptor, a buffer containing the data to write, and the number of bytes to write are the three arguments it requires. In order to find the correct root directory and the file itself, the function first locates the filename associated with the file descriptor in the file descriptor table. Verify the validity of the provided file descriptor upon invocation

(has an entry in your current table of open file descriptors). If not, set errno and return with an error. If the operation is valid, write bytes from the specified location to the specified file, or vice versa. If required, based on the amount to be written to the file, it might require allocating disk blocks.

close()

This function is in charge of freeing the associated file descriptor when a file is closed. When software has finished accessing a file and is no longer required to manipulate it, this is done.
It initially finds the file descriptor that corresponds to the file that is being closed by iterating over the list of file descriptors and searching for the one that is linked to the file. The function changes the boolean flag that indicates whether the file descriptor is in use to false once it has located the file descriptor. The file descriptor is now accessible to other files, as indicated by this. Once the file has been successfully closed, and the file descriptor has been released, the function returns. This ensures that the file is no longer accessible using that file descriptor and frees up the space taken up by the file descriptor for usage by other files.

Utility Functions

For implementing unmount(), mount(), open(), read(), write(), and close() calls, we defined the following helper functions:

- **createdisk():** It takes the disk name and checks if the disk name is null or not and if it can be opened or not. If the disk can be opened, then we write our blocks into it, and we close the disk.
- **create_filesystem():** This will create a disk if not created and open the disk. Here, Superblock is initialized here.
- **opendisk():** Returns an error when the disk name is invalid, or the disk is already opened. It also checks and verifies if the user has permission to open the disk and returns an error if he doesn't have the permissions
- **read_block():** Checks if the disk is open or not and checks if the file is open or not. Returns an error if block index is out of bounds or when lseek is failed or read is failed else reads files and returns successfully.

- **closedisk:** Closes the opened disk and returns an error if not.
- **get_file:** Given the file name returns the index of the file(inode) else returns an error.
- **get_fd():** returns an available file descriptor or else returns an error
- **get_next_block**(): returns the next available block if no block is available or if it is out of bounds it returns an error

Performance

All the required functionalities are satisfied, and the file system is being created optimally