Team NorthFace

Team members: Brian Nguyen, Abel Seyoum, Jonathan Sum, Ysidro Alfaro

GitHub Link:
https://github.com/CSC415-2023-Spring/csc415-filesystem-Eskinaz/tree/main

File System Description:
The file system consists of various components working together to manage file storage, access, and organization. At its core, the Entry Struct represents a single file or directory and contains information such as its name, author, type, access permission, and size in bytes. The extent table within Entry Struct allows efficient storage and retrieval of data throughout continuous block ranges.

To manage free and occupied space, the file system uses a bitmap. Users can interact with the file system through an interface known as the Fsshell. Functions like openVolume(), volumeRead(), and volumeWrite() enable interaction with the file system and provide flexible data access with varying sizes.

The file system maintains a current working directory and allows entry access by parsing file paths. You are able to create and delete entries with functions for creating files (mkfile) or directories (mkdir). To move entries, the file system includes fs_mv()

In addition, functions like open(), seek(), read(), and write() manages file access, positioning, and data manipulation. You are able to delete files and directories by using functions such as fs_delete() and fs_rmdir(). Lastly, the closeVolume() ensures that the system shuts down and any resources are cleaned up when a volume is closed. These features allow the file system to handle various tasks related to file management.

Issues We Had:

Issue: Implementing ParsePath

At first, the idea of parsePath sounded simple as you just took the pathname, and looked through the directories until you found the correct entry with the matching type. Our first implementation did go through the directory entries but treated the entries as an array. The problem is that you could not access the correct LBA once it left the root directory. The resolution was to introduce the use of getEntryBlock(), which passed in our for-loop iterator as the offset from the starting LBA block.

The second issue regarding parsePath was dealing with the absolute/relative calls it received. The way we dealt with this issue was by determining if the path's name started with a '/', which would set our entryLBA from root(6). If we didn't see this character at the start of the path, then we would take the parentLBA which was set from our volume control blocks' current working LBA

Issue: Setting up Root Directory

While setting up our root directory, the first problem that came up was that our root directory wasn't appearing in the correct block when we ran our hex dump. In addition to that, our magic number wasn't appearing, so we knew immediately that our volume control block was not written back to the disk correctly. (I don't remember the resolution someone fill in plz)

Issue: growEntry not corrupting volume

When creating growEntry(), some of the difficulty was in that if one command failed, we needed to roll back all the blocks allocated in the current call. The reason for doing this is that we did not want to abandon the grow because that would leave the block marked as used despite not being assigned to the entry. However, attempting to roll back the current changes also had a chance to fail. So the solution was to calculate all the blocks we would need for allocating the entry and its extent tables before we even start overwriting. Then, if we failed, it was much easier to undo the new extents written because we would just erase starting from the end.

Issue: Implementing setcwd

In the beginning, when we were working on fs_setcwd(), we tried to parse the input path and verify if there was a valid directory. However, we encountered challenges while addressing both absolute and relative paths, as well as navigating the path components like ".." and ".". To overcome these difficulties, we revised our approach and made serval adjustments to the code.

For improving the handling of absolute and relative paths, we decided to concatenate the current working directory (cwd) with the user's input in case of relative paths, while processing absolute paths independently. We then broke the combined path into individual components by tokenizing it, allowing us to efficiently manage the ".." and "." path components by collapsing the path, essentially eliminating any unnecessary or redundant parts.

Once we reconstructed the collapsed path, we transformed it into an LBA and confirmed the directory at that location. If the directory was not found or if the path pointed to a file instead, we display an error message and return a failure code.

We update the global working directory variable with the new entry and release any allocated memory. Throughout this process, we had to make multiple modifications to our initial code to ensure it could handle all possible edge cases.

Detail of how your driver program works:

Entry Struct:

- Represents a single file or directory in the file system
- Reasons behind variables
- Name
    - Hold the file's name
- Author
    - Stores the file's author
- Type
    - Used to determine if the entry is a directory or a file
- Permission
    - Represents the file's permissions (read, write, execute)
- blockCount
    - Stores the number of blocks the file occupies in the file system
- Size
    - Indicates the file's size in bytes
- Data
    - An extent table is an array of extents (continuous block ranges).
    - Each extent contains a starting block address and the number of contiguous blocks
    - This table allows efficient storage and retrieval of file data in the file system
    - The maximum number of extents supported depends on the 'TABLE_SIZE' constant
- Created
    - Is a timestamp that stores when the entry was created
- Modified
    - Is a timestamp that stores when the entry was modified
- Accessed
    - Is a timestamp that stores when the entry was accessed
- How it holds allocated blocks / How extent table works
    - Max # of extents supported
    - getEntryBlock()'s LBA conversion

Free Space management:

- Bitmap
    - Manages free and occupied blocks in the file system.
    - Each bit represented a block in the file system, with 1 indicating the block is occupied, and 0 indicating it's free
- How blockAlloc() chooses what to allocate
    - It iterates through the freeMap bitmap to find a free block (with a bit value of 0) and marks it as occupied (setting the bit value to 1)
    - It'll return an array of extents, which are structures containing information about the allocated blocks (LBA) and the number of blocks
- How we did bit-shifting for freeMap
    - Mark blocks are used or free in the freeMap bitmap that uses bit-shifting operations.

Fsshell:

- How our program interacts with it

openVolume():

- Sets up the system for interacting with the volume file and returns 0 if successful
- The function checks the parameters to ensure that they are valid and that no other volume is currently open
- The function sets up the volume control block (VCB) by allocating memory and reading its content from the volume file
- It checks if the volume is formatted using the magic number. If it is, it proceeds to get the existing freeMap from the volume
- If the volume is not formatted, it initializes a new volume by calling newVolume() with its given volume size and block size parameters
- The current working directory is initialized, setting the root directory as the initial current working directory and setting the path to "/"
- Once the initialization is completed, the function returns 0 to indicate success

volumeRead():

- Purpose: allow smaller than blockSize access. Reading data from storage with flexible sizing
- Reads data from storage starting at a specific LBA block
- Takes 3 inputs: a buffer to store data, a number of bytes to read, and the starting block (LBA)
- Checks if the request is valid
- Reads data in two steps: directly reads whole blocks, then reads partial blocks

- Returns the number of bytes read

volumeWrite():

- Purpose: allow smaller than blockSize access. Reading data from storage with flexible sizing
- Writes data to storage starting at a specific LBA block
- Takes 3 inputs: a buffer containing data, the number of bytes to write, and the starting block (LBA)
- Writes data in two steps: directly writes whole blocks, then writes partial blocks
- Returns the number of bytes written

Entry Access:

parsePath:

- Is to analyze a given file path and locate the corresponding block address (LBA) in the filesystem
- It starts by checking whether the path begins from the root directory or the current working directory
- The function then breaks the path into smaller tokens, separated by "/"
- When a matching entry is found, it updates the parent directory and continues processing the next token in the path
- If the end of the directory is reached without finding a matching entry, the function will return -1, indicating it failed
- If the successful parades the entire path, it'll return the LBA of the final entry in the path

Current working directory:

- We store the current working directory as a global variable
- Setcwd
    - Change the current working directory in the file system to the desired location
    - Takes 1 input: a pathname representing the desired new working directory
    - Combines the current working directory with the user-provided pathname
    - Tokenizes the combined path, splitting it into its components(directories)
    - Collapses the path, removing redundant parts (".." and ".")
        - ".." means to go up one level, so it eliminates itself and the previous directory
        - "." means the current directory, so it can be removed
    - Rebuilds the collapse path, combining the remaining parts into a single string

        - Checks if the provided pathname corresponds to an existing directory
        - Updates the global working directory variable
        - Free allocated memory and returns 0 (success)

Shell Commands

- LS
    - Talk about the commands needed for it to run
    - fs_dir(), fs_opendir(), displayFiles(): fs_readdir(), fs_stat(), fs_closeddir(), fs_isfile, fs_getcwd()

Entry Creation:

- fs_mkfile
    - Checks if the pathname is free
    - Splits the pathname into the path to the parent directory and the name of the new file
    - Gets the LBA of the parent directory using parsePath()
    - If the parent directory doesn't exist, return -1
    - Ensure the parent is a directory
    - Gets the first free entry in the parent directory
    - Creates a new file at the obtained entry LBA with the specified name
    - Free the memory allocated for the path and name
- fs_mkdir
    - Checks if the pathname is free
    - Splits the pathname into the path to the parent directory and the name of the new directory
    - Gets the LBA of the parent directory using parsePath()
    - If the parent directory doesn't exist, create it and get its LBA
    - Ensure the parent is a directory
    - Gets the first free entry in the parent directory
    - Creates a new directory at the obtained entry LBA with the specified name
    - Free the memory allocated for the path and name
- Difference between the two
    - Both fs_mkfile() and fs_mkdir() follow similar steps, but mkfile() creates a new file while fs_mkdir() creates a new directory

- fs_mv()
    - Used to move entries from one location to another given paths
    - Takes 2 parameters of a source path(srcPath) and a destination path(destPath)

- If the srcPath type is equivalent to a file
  - volumeRead() entry from srcPath
  - Memcpy from the srcEntry to the destEntry
  - If the destination file does not exist
    - Create a new file and memcpy the same way
- If the srcPath type is equivalent to a directory
  - We take the same methods as a file, but we create a pointer to an entry and populate it that way

Milestone 3

- **b_open():**
  - Opens a file in the volume control block and assigns access based on certain flags
  - Takes a filename and flags
    - Flags can be read through bit comparisons
  - Before starting, it checks if the filename is valid through parsePath() as well as if there are any free control blocks
    - If there are no free control blocks, return -1
  - O_RDONLY
    - Sets access of the file in the fcb to O_RDONLY
  - O_WRONLY
    - Sets access of the file in the fcb to O_WRONLY
  - O_RDWR
    - Sets access of the file in the fcb to O_RDWR
  - For the remaining flags, check to see if the file is WRONLY or RDWR
    - O_CREATE
      - If the file doesn't exist from parsePath(), fs_mkfile
      - volumeRead() entry from newly created file LBA
      - Change time parameters
      - volumeWrite()entry back to disk
    - O_TRUNC
      - Checks if the path is valid
        - If not, return an error of -1
      - Reads in the entry with VolumeRead()
      - memset() the entry to 0 as well as set the size of the entry to 0
      - Change time parameters
      - volumeWrite() entry back to disk
    - O_APPEND
      - Checks if the path is valid
        - If not, return an error of -1
      - Reads entry in from volumeRead()

        - Sets the vcb filePos to the entry's size - 1
        - Change time parameters
        - volumeWrite() back to disk

- **b_seek():**
    - It is used to change the position of the file pointer within a file
    - It takes three arguments: a file descriptor (fd), an offset, and a reference point(whence) for the changes
    - Before starting, it checks if the given file descriptor is valid and retrieves the file size
    - Depending on the value of the reference point(whence), it'll adjust the file pointer.
        - SEEK_SET: moves the file pointer to a specified offset from the beginning of the file
        - SEEK_CUR: moves the file pointer by the specified offset from its current position
        - SEEK_END: moves the file pointer to the specified offset from the end of the file
    - When it is in read-only mode, the function seeks past the end of the file

- **b_read():**
    - Reads data from a file into a buffer provided by the user
    - Takes three arguments: a file descriptor(fd), a buffer to store the data, and the number of bytes to read (count)
    - It tracks the number of bytes read, the remaining bytes to read, the current block position, and the offset within the block
    -
- **b_write():**
    - Initializes the system if it has not started yet
    - Checks if the file descriptor is within the valid range, and returns -1; if not
    - Keeps track of the number of bytes written and the remaining bytes to be written
    - Determines the current position in the file and calculates the starting LBA
    - If the starting LBA is -1, the end of the file has been reached and the function will return 0
    - If there are remaining bytes to be written
        - Calculates the number of bytes to copy from the block buffer
        - If the number of bytes to copy is greater than the remaining bytes, adjust the bytes to copy accordingly
        - Copies the data from the user-provided buffer to a system buffer and writes it to the volume at the specified LBA

        - Updates the number of bytes written, the remaining bytes, and the file position and the file position accordingly
    - Returns the total number of bytes written
    - Talk about how it decides to grow()?
- **b_close():**
    - Ensures the that the specified file is properly closed
    - Checks if the given file descriptor is within the valid range
    - Retrieving the file control block that is associated with the file descriptor
    - Calculating allocated and required blocks
        - Determines the current number of blocks allocated to the file
        - The number of blocks needed is based on the file size
    - Releasing the extra blocks
        - If there are extra blocks allocated to the file
        - Then using the shrink
    - Updating the file control block
        - Releasing extra blocks, the file control block is updated to reflect the new block count
    - Resetting the file descriptor
        - The file descriptor in the file control block array is reset once the file is closed
    - Return success if everything works
    - Talk about how it decides to shrink()?

Entry Deletion:

- Fs_delete
    - Locates the specified file or directory using its path
    - If the file or directory was not found, an error message will be displayed and return -1
    - Reads the entry associated with the file or directory from the disk
    - Frees all blocks assigned to the file
    - If a tertiary table exists, it frees the associated primary and secondary tables
    - Sets the entry name to an empty string and the entry type to FREE_ENTRY
    - Writes to the modified entry to the disk
- Fs_rmdir
    - Uses the provided path to find the location(LBA) of the specified directory
    - If the directory was not found it’ll return -1
    - Reads the entry at the LBA
    - Ensures that the entry time is a directory; if not, it’ll return -1
    - Checks if the directory is empty; if not, return -1
    - Frees the directory is empty and any allocated memory associated with it

    - Return 0 if it is successful in removing the directory

Close Volume:

- Is responsible for shutting down the system and cleaning up resources
- It checks if there is an open volume by checking vcb
- If no volume is open, it'll inform the user with a message
- If the function is successful it'll close the volume and clean up resources, and return a success code

Screenshots showing each of the commands listed in the readme: