simple file management system

Made by: Bouchefra Yassine

Section: B

Group: 2

Marticule: 232331388118

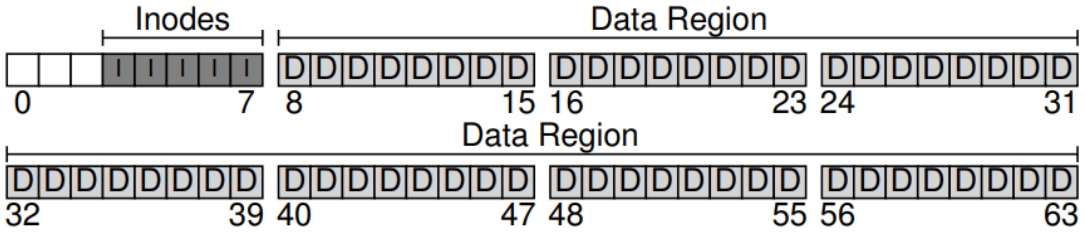
The idea

the implementation of a simple file system called vsfs (Very Simple File System), which serves as an introduction to the basic concepts of file systems.

It discusses the on-disk structures, access methods, and policies that are common in various file systems,The text emphasizes the importance of understanding both the data structures (like inodes and bitmaps) and access methods (system calls like open, read, write) to develop a mental model of how file systems operate.

It covers the organization of data on disk, the role of inodes for metadata storage, directory organization, free space management, and the performance implications of file system operations.

What are the Structures



Client

- int ID: A unique identifier for the client.
- char name[MAX_NAME_LENGTH]: The name of the client (up to MAX_NAME_LENGTH characters).
- int age: The age of the client.
- float balance: The account balance of the client.
- int LD: Logical deletion flag (1 for logically deleted, 0 for active).

Block

- `entries[FB]`: An array of `Client` structures. Each block can hold FB number of `Client` records (with FB likely being a constant defining the block's capacity in terms of records).
- `nbr_records`: The number of records currently stored in the block. This is important because a block might not be fully populated with records.
- `next`: A pointer (or index) to the next block in the chain. If this block is the last in the chain,

Meta

- char name[MAX_NAME_LENGTH]: The name of the file.
- int start_block: The index of the first block of the file.
- int file_size_in_blocks: The total number of blocks occupied by the file.
- int file_size_in_records: The total number of records in the file.
- int global_organisation_mode: Organization mode of the file:
  - 0 for contiguous organization.
  - 1 for chained organization.
- int internal_organisation_mode: Organization mode of records within blocks:
  - 0 for unsorted records.
  - 1 for sorted records.

Position

This struct represents a specific position within the file system, such as a metadata entry or a record.

- int block_index: The index of the block containing the desired entry.
- int index: The index within the block (e.g., for a Client record).

Global Constants and Variables

1. **`records_per_block`**: Calculates how many `Client` records can fit into one `Block`.
2. **`meta_per_block`**: Calculates how many `Meta` entries can fit into one `Block`.

3. **nbr_meta_blocks**: Determines the number of blocks needed to store all metadata entries.
4. **first_data_region_block**: Finds the index of the first block used for data (after metadata blocks).
5. **nbr_free_blocks**: Tracks how many free blocks are available in the file system.
6. **nbr_free_metas**: Tracks how many free metadata blocks are available.

Error Handling Variables

1. **NO_FREE_BLOCKS_ERROR_MSG**: Error message for no free blocks available.
2. **is_storage_full**: Flag indicating if the storage is full.
3. **FILE_TOO_BIG_ERROR_MSG**: Error message for files that are too large.
4. **is_file_too_big**: Flag indicating if a file is too big to store.
5. **CONTIGUOUS_SPACE_ERROR_MSG**: Error message for insufficient contiguous space for a file.
6. **contiguous_space_issue**: Flag indicating a problem with contiguous space.
7. **NO_FREE_META_ERROR_MSG**: Error message for no free metadata blocks available.
8. **is_metadata_full**: Flag indicating if there are no free metadata blocks.
9. **FILE_NOT_FOUND_ERROR_MSG**: Error message for a file that doesn't exist.
10. **file_not_found_flag**: Flag indicating that a file was not found.
11. **FILE_NAME_ALREADY_EXIST_ERROR_MSG**: Error message for when a file name already exists.
12. **is_file_name_duplicate**: Flag indicating a file name conflict.
13. **RECORD_NOT_EXIST_ERROR_MSG**: Error message for a record that does not exist.
14. **record_not_found_flag**: Flag indicating that a record was not found.

Functions

`initFileSystem(FILE* MS)`

- **Purpose**: Initializes the entire file system.
    - Sets up the allocation table.
    - Initializes the metadata.
    - Initializes all the blocks (set to empty).
    - Writes empty blocks to the file system file `MS`.

`printFileSystem(FILE* MS)`

- **Purpose**: Prints out the current status of the file system.
    - Displays the allocation table.
    - Displays the metadata for all files, including details like file name, starting block, size in blocks, and organization modes.

`initInode(Meta* inode)`

- **Purpose**: Initializes an inode (metadata) to its default state.
    - Sets the inode's name to an empty string and all size and organizational modes to -1 or 0, indicating it’s not yet assigned.

`printBlocks(FILE* MS)`

- **Purpose**: Prints the details of all blocks in the file system.
    - For each block, it prints its index, number of records, and the pointer to the next block.

`EmptyDisk(FILE* MS)`

- **Purpose**: Clears and reinitializes the file system.
    - Opens the file `MS.bin` for writing and initializes the file system, erasing any existing data.

`validation(int file_size_in_blocks)`

- **Purpose**: Validates whether the file system has enough resources for a file.
    - Checks if there are free blocks available for storage.
    - Checks if the file's size exceeds the available free blocks.
    - Checks if there are free metadata blocks available.

`displayErrors()`

- **Purpose**: Displays all error messages based on the current flags.
    - Constructs a string with all the error messages that have been flagged (e.g., storage full, file too big, no free metadata, etc.) and prints them.

`resetFlags()`

- **Purpose**: Resets all error flags to their default (false) state.
    - Clears all the flags related to errors or issues, allowing the system to be ready for new operations or validations.

`readAllocationTable(FILE* MS, int allocation_table_buffer[])`

- **Purpose**: Reads the allocation table from the file system.
    - It rewinds the file to the beginning, reads the allocation table (which keeps track of used and free blocks), and then moves the file pointer to the beginning of the data blocks.

`writeAllocationTable(FILE* MS, int allocation_table_buffer[])`

- **Purpose**: Writes the current allocation table to the file system.

- It rewinds the file to the beginning and writes the updated allocation table to the file, then moves the file pointer to the first data block.

`initAllocationTable(FILE* MS, int allocation_table_buffer[])`

- **Purpose**: Initializes the allocation table by setting all blocks as free (value of 0).
  - The function initializes the allocation table to zero and then writes this initialized table back to the file system.

`findFreeAdjacentBlock(FILE* MS, int file_size_in_blocks)`

- **Purpose**: Finds contiguous free blocks to store a file.
  - It reads the allocation table and searches for adjacent free blocks that can accommodate the file's required size. If contiguous blocks are found, they are marked as allocated, and the updated table is written back to the file. If no contiguous blocks are found, it sets the flag for a contiguous space issue.

`printAllocationTable(FILE* MS, int allocation_table_buffer[])`

- **Purpose**: Prints the current allocation table to the console.
  - Displays the status (allocated or free) of each block in the allocation table by printing each entry in the `allocation_table_buffer`.

`initMetadata(FILE* MS, Meta metabuffer[])`

- **Purpose**: Initializes the metadata for the file system.
  - Creates and writes empty `Meta` entries (inodes) to the metadata blocks in the file system. Each block in the metadata region is filled with empty inodes using the `initInode` function.

`promptInode(FILE* MS, Meta *inode)`

- **Purpose**: Prompts the user to input metadata for a file.
  - Asks the user to input the file's name, number of records, global organization mode (contiguous or chained), and internal organization mode (sorted or not sorted). It also calculates the file size in blocks based on the number of records.

`fillStartBlock(FILE* MS, Meta* inode)`

- **Purpose**: Allocates a starting block for a file.
    - Based on the file's global organization mode (contiguous or chained), it finds and allocates a starting block for the file. If there's not enough space or free metadata, it sets the start block to `-1`.

`displayInode(FILE* MS, char* filename)`

- **Purpose**: Displays metadata of a specified file.
    - Finds the metadata for a file using its name, and if found, displays the file's information (name, number of records, number of blocks, organization modes, and start block). If the file is not found, it sets an error flag.

`searchMetadata(FILE* MS, char name[])`

- **Purpose**: Searches for the metadata of a file by its name.
    - Searches through the metadata blocks to find the file with the specified name. If found, it returns the position (block index and entry index) of the metadata. If not found, it sets an error flag and returns an invalid position.

`generateClient(int id)`

- **Purpose**: Generates a random client record.
    - Creates a `Client` struct with a random ID, name, age (between 20 and 60), balance (between 1000 and 5000), and sets the LD value to 0.

`fillFileDataAndMeta(FILE* MS, Meta inode)`

- **Purpose**: Fills both metadata and data for a file.
    - First, it updates the file's metadata by writing the provided `inode` into the metadata region. Then, depending on the global organization mode (contiguous or chained), it fills the corresponding blocks with client data. In contiguous mode, blocks are written sequentially, while in chained mode, each block points to the next available block.

`displayFileData(FILE* MS, char* filename)`

- **Purpose**: Displays the data of a specified file.

- Searches for the metadata of the file using its name, retrieves the starting block, and then reads the data blocks sequentially (or chained if applicable). It displays the records (client details) in the blocks.
-

`createFile(FILE* MS)`

- **Purpose**: Creates a file in the file system.
  - Prompts the user for file metadata, allocates the file's starting block, and writes the data and metadata to the file system. If the file creation fails (due to space or metadata issues), it displays relevant errors. If successful, it displays the file's metadata and data.

`search(FILE* MS, char* filename, int id, Position* p)`

- **Purpose**: Searches for a specific record in the file (by its ID) within the file specified by `filename`.
- **How it works**:
  - It finds the metadata for the given file, reads it, and checks if the file is sorted or unsorted.
  - If unsorted, it performs a linear search over blocks and their records.
  - If sorted, it performs a binary search over the blocks and inside the blocks (depending on the global organization mode—contiguous or chained).
  - Returns `true` if the record is found and updates the `Position` structure with the block index and record index. If not found, returns `false`.

`searchRecord(FILE* MS, char* filename, int id, Position* p)`

- **Purpose**: Searches for a specific record by ID in a file.
- **How it works**:
  - Similar to the `search()` function but focused on finding an individual record. If the file is in sorted mode (whether contiguous or chained), it performs binary search; if unsorted, it performs a linear search.
  - Returns `true` if the record is found and updates the position with the block and index, otherwise returns `false`.

`insertRecord(FILE* MS, char* filename)`

- **Purpose**: Inserts a new record (client) into the file.
- **How it works**:
    - It first searches for the appropriate position in the file and handles record insertion in either contiguous or chained mode.
    - If in contiguous mode, it looks for available space in the last block or allocates a new block.
    - If in sorted mode, it handles insertion while maintaining order within the blocks.
    - It handles different types of allocation strategies (whether adjacent blocks are free or not).
    - The function updates both the file data and the metadata upon successful insertion of the new record.

`deleteFile(FILE* MS, char* filename)`

- **Purpose**: Deletes a file by removing its metadata and all associated blocks.
- **How it works**:
    - It first finds the metadata for the file, then clears the metadata by writing default values to the block where the metadata was stored.
    - Then, it reads and deallocates the blocks that store the file's data, freeing up space in the allocation table.
    - Finally, it updates the allocation table to reflect the freed blocks.

`renameFile(FILE* MS, char* old_filename, char* new_filename)`

- **Purpose**: Renames a file in the file system.
- **How it works**:
    - The function first searches for the metadata of the file using `old_filename`.
    - If the file is found, it checks that the new file name does not exceed the maximum length.
    - It then reads the metadata, updates the `name` field of the `Meta` structure with the new filename, and writes the updated metadata back to the disk.
    - If the file is not found, it sets the `file_not_found_flag` to `true` and exits.
    - If the new filename exceeds the allowed length, it prints an error message.

`binarySearch(Client clients[], int left, int right, int val)`

- **Purpose**: Performs a binary search on a sorted array of `Client` structs.
- **How it works**:
  - The function searches for a specific `ID` (`val`) in the `clients` array.
  - It calculates the midpoint index and compares the value at that index to the target value.
  - If the value is found, it returns the index of the client.
  - If the value is not found, it returns `-1`, indicating the absence of the client.

`binarySearchInsertion(Client clients[], int left, int right, int val)`

- **Purpose**: A variant of the binary search that finds the insertion point of a value in a sorted array.
- **How it works**:
  - This function performs a binary search to find the appropriate position for inserting a new client ID (`val`) in a sorted array of `Client` structs.
  - It returns the index where the value should be inserted to maintain the sorted order.
  - This is particularly useful for inserting new records into a sorted list while keeping it ordered.

`displayMenu()`

- **Purpose**: Likely intended to display a user interface menu.
- **How it works**:
  - This function seems to be the beginning of a user interface but is incomplete in the code you provided. Typically, this would display options to the user and wait for their input to navigate through the program (e.g., opening, renaming, or deleting files).