Umit Altintas - 171044005 - CSE 312 - Homework 3

File System Design Report for FAT12 Structure

1. Directory Table and Directory Entries

The directory table in the FAT12 file system is implemented using the `dir_entry_t` struct. Each entry represents a file or a directory in the file system.

Here is the structure of `dir_entry_t` :

```
typedef struct {
    unsigned char filename[8]; // filename (8 bytes)
    unsigned char ext[3]; // extension (3 bytes)
    unsigned char attr; // attributes (1 byte)
    unsigned short time; // last modification time (2 bytes)
    unsigned short date; // last modification date (2 bytes)
    uint_12 starting_cluster; // starting cluster of the file/dir
    unsigned int file_size; // size of the file (4 bytes)
} dir_entry_t __attribute__((packed));
```

Each directory entry contains the following information:

- filename (8 bytes),
- extension (3 bytes),
- file attributes (1 byte),

- time and date of last modification (2 bytes each),
- starting cluster of the file/directory (1.5 bytes), and
- size of the file (4 bytes).

This structure is packed to prevent the compiler from adding any padding for alignment.

2. Free Blocks Management

Free blocks in the file system are tracked using the File Allocation Table (FAT), specifically, `uint_12 *fat` . In this design, a 12-bit unsigned integer (`uint_12`) represents each block in the FAT. If a block's corresponding entry in the FAT is 0, that block is considered free. If the FAT entry is non-zero, it points to the next block of the file.

The `allocate_cluster` function scans the FAT for free blocks. Upon finding a free block, it assigns it and updates the FAT.

3. Superblock Design

The superblock of the file system contains crucial information about the system, including block size, root directory position, and block positions. This information is kept in the `boot_sector_t` struct. Here is the structure of `boot_sector_t` :

```
typedef struct {
    unsigned short block_size; // block size (2 bytes)
    unsigned short block_count; // number of blocks (2 bytes)
    unsigned short root_dir_entries_count; // number of entries in
} boot_sector_t __attribute__((packed));
```

The `block_size` field represents the size of each block in bytes. The `block_count` field represents the total number of blocks in the file system. The `root_dir_entries_count` field represents the total number of entries

in the root directory. The structure is packed to prevent any compiler-induced padding for alignment.

Finally, the `fat12_t` struct represents the whole file system, which includes the `FILE *file` (a file pointer representing the physical storage of the file system), `boot_sector_t boot_sector` (the superblock), `uint_12 *fat` (the file allocation table), and `dir_entry_t *root_dir` (the root directory table).

4. Command Implementation and Execution

The FAT12 file system design provides a variety of commands to manipulate the file system, directories, and files. Here are the details on how these commands are implemented and used.

4.1. Directory Operations

- `dir` **command**

This command lists the contents of a directory specified by a given path. The function `list_dir(fat12_t *fs, char *path)` is used to implement this command. If the path is "\" the root directory is listed.

- `mkdir` **command**

This command creates a new directory in a specified location. It uses the `fetch_dir_entry(fat12_t *fs, char *dir_name)` function. The function checks if a directory with the given name already exists, and if not, it creates a new one.

- `rmdir` **command**

This command removes an existing directory from a specified location. It uses the `delete_dir(fat12_t *fs, char *path)` function. The function checks if the directory exists and then deletes it.

4.2. File Operations

`write` command

This command creates a new file and writes data to it. The function `write_file(fat12_t *fs, char *path, void *data, size_t size)` is used. The function checks if the file already exists. If it does, it overwrites it. If it doesn't, it creates a new one and then writes the provided data to it.

`read` command

This command reads data from a specified file. The function `read_file(fat12_t *fs, char *path, char *dest_path)` is used. It reads the content from the file at the given path and then writes this data to the specified destination path.

`del` command

This command deletes a file from a given path. The function `delete_file(fat12_t *fs, char *path)` is used to implement this command. It checks if the file exists and if it does, deletes it.

4.3. File System Operations

`dumpe2fs` command

This command provides information about the file system. The function `dumpe2fs(fat12_t *fs)` is called. It lists the block count, free blocks, number of files and directories, and block size, along with the occupied blocks and the file names for each of them.

5. Code Overview

5.1 Directory Listing: `list_dir`

The function `list_dir(fat12_t *fs, char *path)` takes in a file system pointer and a directory path. It first fetches the directory entry for the

given path using `fetch_dir_entry_without_creation()`. If the directory exists, it reads it using `read_directory()`, and then prints out the names of all files and directories in the current directory.

5.2 Directory Deletion: `delete_dir`

The function `delete_dir(fat12_t *fs, char *path)` deletes a directory at a given path in the file system. After fetching the directory entry, it checks if it is a directory and if it is empty. If both conditions are met, it deletes the directory by setting the first character of its name to '\0' in the directory entry, and updates the File Allocation Table (FAT) to free up the clusters occupied by the directory.

5.3 File Deletion: `delete_file`

The function `delete_file(fat12_t *fs, char *path)` works similarly to `delete_dir()`, but is designed to delete files. It fetches the directory entry of the file and checks if it is indeed a file. If it is, it deletes the file and updates the FAT accordingly.

5.4 File Reading: `read_file`

The function `read_file(fat12_t *fs, char *path, char *output_path)` reads data from a file in the file system and writes it to an output file in the host system. It checks if the directory entry of the given path is a file, reads its data, and writes it to the output file.

5.5 Directory Entry Fetching: `fetch_dir_entry`

The function `fetch_dir_entry(fat12_t *fs, char *path)` retrieves a directory entry for a given path. If the directory does not exist, it creates a new one using the `create_dir()` function. The `parse_path()` function is used to split the path into tokens which are then used to traverse the directory tree.

These functions show the fundamental operations of reading, writing, and deleting directories and files in the file system. Understanding these will aid in understanding the larger file system design and its functionalities.

6 TESTING

The file system was tested using the `test.sh` script. The script creates a file system, creates directories and files,

```
> /bin/bash /Users/codefirst/Documents/Gtu/os-homeworks/hw-3/test
Starting the compilation process...
Compiling fileSystemOperation...
g++ -std=c++17 -Wno-ignored-attributes -o fileSystemOperation bui
Compiling makeFileSystem...
make: `makeFileSystem' is up to date.
Creating a source file with sample text...
Creating a file system...
File system created successfully.
Test 1: List directory contents (dir command)...
Test 2: Create directory (mkdir command)...
Directory created successfully.
Directory created successfully.
fname
Test 3: Remove directory (rmdir command)...
Test 5: Write data to a file (write command)...
Entry added successfully.
Test 6: Read data from a file (read command)...
Contents of the target file:
ASDDFASDF

Test 7: Delete a file (del command)...
Test 4: Dump filesystem info (dumpe2fs command)...
Block count: 2880

Block size: 2048
Root directory entries count: 14
Root directory:
0: /
1: ysa
2:
3:
4:
5:
```

```
6:
7:
8:
9:
10:
11:
12:
13:
Files:
0: /
1: ysa
Cleaning up - Removing the compiled program...
All tasks completed successfully!
```