Simplified FAT12 File System Design and Implementation

Tugay Talha İçen
210104004084
Gebze Technical University
Department of Computer Engineering
CSE 312 - Operating Systems

June, 2024

Abstract

This report details the design and implementation of a simplified FAT12 file system. The file system supports basic file operations, including creation, reading, writing, and deletion, as well as directory management and permission handling. The design mimics the FAT12 structure and includes features such as dynamic file name lengths, owner permissions, and password protection for files.

1 Introduction

The purpose of this project is to design and implement a simplified FAT12-like file system in C/C++. The file system supports file operations and includes attributes such as file name, size, owner permissions, modification and creation dates, and password protection. This report outlines the design considerations and implementation details of the file system.

2 File System Design

2.1 Directory Table and Directory Entries

The directory table is a critical structure in the file system that holds directory entries. Each directory entry represents a file or directory and contains the following attributes:

- **File Name:** The file name is stored as a dynamically allocated string to support arbitrary lengths. This allows the file system to handle file names of any length efficiently.
- **File Size:** The size of the file in bytes, which is an integer value indicating the total number of bytes the file occupies.
- **Owner Permissions:** Permissions for the file owner, including Read (R) and Write (W) permissions. These are implemented as boolean flags.
- **Last Modification Date and Time:** A timestamp indicating when the file was last modified. This is stored as a string in the format "YYYY-MM-DD HH:MM:SS".
- **File Creation Date and Time:** A timestamp indicating when the file was created, stored in the same format as the modification date.
- **Password:** If the file is password-protected, the password is stored as a hashed string to ensure security.

```
struct DirectoryEntry {
    char* fileName;             // Dynamically allocated file name
    int fileSize;               // Size of the file in bytes
    bool readPermission;        // Owner read permission
    bool writePermission;       // Owner write permission
    char lastModified[20];      // Last modification timestamp
    char creationDate[20];      // File creation timestamp
    char* passwordHash;         // Hashed password for protection
};
```

2.2 Free Blocks Management

Free blocks are managed using a bitmap, where each bit represents the status of a block in the file system:

- **0:** The block is free and available for allocation.
- **1:** The block is allocated and in use.

The bitmap is an efficient way to keep track of free and used blocks, allowing quick allocation and deallocation of space.

```
#define TOTAL_BLOCKS 8192  // Total number of blocks (for a 4 MB file
bool blockBitmap[TOTAL_BLOCKS];

// Initialize all blocks to free
for (int i = 0; i < TOTAL_BLOCKS; ++i) {
    blockBitmap[i] = false;
}
```

2.3 Handling Arbitrary Length File Names

File names in our file system are not restricted to a fixed length. Instead, they are stored as pointers to dynamically allocated memory, allowing file names of arbitrary length. This approach avoids the limitations and inefficiencies associated with fixed-length strings.

```
// Function to create a new directory entry with an arbitrary length f
DirectoryEntry* createDirectoryEntry(const char* name, int size) {
    DirectoryEntry* entry = new DirectoryEntry;
    entry->fileName = new char[strlen(name) + 1];
    strcpy(entry->fileName, name);
    entry->fileSize = size;
    // Initialize other fields...
    return entry;
}

// Function to free the memory allocated for a directory entry
void freeDirectoryEntry(DirectoryEntry* entry) {
    delete[] entry->fileName;
    delete entry;
}
```

2.4 Permissions Handling

Permissions in the file system are managed using two boolean flags for each directory entry. These flags control the read and write access for the file owner:

- **Read (R):** If this flag is set, the file can be read by the owner.
- **Write (W):** If this flag is set, the file can be written by the owner.

Operations on files check these flags before performing the requested action to ensure proper access control.

```
// Function to check if the file can be read
bool canRead(const DirectoryEntry* entry) {
    return entry->readPermission;
}

// Function to check if the file can be written
bool canWrite(const DirectoryEntry* entry) {
    return entry->writePermission;
}
```

2.5 Password Protection

Password protection is an optional feature for files. When a password is set for a file, it is stored as a hashed value. This ensures that the actual password is never stored in plain text, enhancing security. Operations that require access to a protected file must provide the password, which is then hashed and compared to the stored hash.

```
#include <openssl/sha.h>

// Function to hash a password
char* hashPassword(const char* password) {
    unsigned char hash[SHA256_DIGEST_LENGTH];
    SHA256((unsigned char*)password, strlen(password), hash);
    char* hashString = new char[SHA256_DIGEST_LENGTH*2 + 1];
    for (int i = 0; i < SHA256_DIGEST_LENGTH; ++i) {
        sprintf(&hashString[i*2], "%02x", hash[i]);
    }
    return hashString;
}

// Function to add a password to a directory entry
void addPassword(DirectoryEntry* entry, const char* password) {
    entry->passwordHash = hashPassword(password);
}

// Function to verify the password
bool verifyPassword(const DirectoryEntry* entry, const char* password)
    char* hash = hashPassword(password);
```

```
    bool result = (strcmp(entry->passwordHash, hash) == 0);
    delete[] hash;
    return result;
}
```

3 Implementation

3.1 Creating an Empty File System

The `makeFileSystem` function creates an empty file system:

```
void makeFileSystem(int blockSizeKB, const char* fileName) {
    int fileSize = 4 * 1024 * 1024; // 4 MB in bytes
    FILE *file = fopen(fileName, "wb");
    ftruncate(fileno(file), fileSize);
    fclose(file);
}
```

This function initializes the file system structure, including the superblock, data blocks, directory structure, and free blocks bitmap.

3.2 File System Operations

The `fileSystemOper` function performs various file system operations:

- **dir:** Lists the contents of a directory.
- **mkdir:** Creates a new directory.
- **rmdir:** Removes a directory.
- **dumpe2fs:** Provides information about the file system.
- **write:** Creates and writes data to a file.
- **read:** Reads data from a file.
- **del:** Deletes a file.
- **chmod:** Changes file permissions.
- **addpw:** Adds a password to a file.

3.3 Function Implementations

3.3.1 Listing Directory Contents

The `dir` function lists the contents of a directory:

```
void listDirectory(const char* path)
```

3.3.2 Creating a Directory

The `mkdir` function creates a new directory:

```
void makeDirectory(const char* path)
```

3.3.3 Removing a Directory

The `rmdir` function removes a directory:

```
void removeDirectory(const char* path)
```

3.3.4 Dumping File System Information

The `dumpe2fs` function provides information about the file system:

```
void dumpFileSystemInfo()
```

3.3.5 Writing to a File

The `write` function creates and writes data to a file:

```
void writeFile(const char* path, const char* linuxFile)
```

3.3.6 Reading from a File

The `read` function reads data from a file:

```
void readFile(const char* path, const char* linuxFile)
```

3.3.7 Deleting a File

The `del` function deletes a file:

```
void deleteFile(const char* path)
```

3.3.8 Changing File Permissions

The `chmod` function changes file permissions:

```
void changePermissions(const char* path, const char* permissions)
```

3.3.9 Adding a Password to a File

The `addpw` function adds a password to a file:

```
void addPassword(const char* path, const char* password)
```

4 Conclusion

This report has outlined the design and implementation of a simplified FAT12 file system. The file system supports essential file operations, directory management, and permission handling. The design choices made ensure that the file system is both functional and secure, providing a basis for further enhancements and optimizations.