Simplified File System Based on pseudoFAT

Author: Milan Vlachovský

Contents

1. Introduction

The goal of this project was to create a simplified file system that operates on the principles of the well-known FAT file system. It is only a simulation of a file system created within a university course for study purposes.

A lower-level programming language that allows direct manipulation of program memory was required. Go was chosen thanks to its efficient memory management, support for concurrency, and relatively low level of abstraction compared to other modern languages.

2. Program Description

The program implements a simplified file system based on the principles of FAT (File Allocation Table). The goal is to enable the management of files and directories within a virtual disk stored as a binary file. The program provides basic operations such as creating, deleting, and moving files, working with directories, and loading or saving data.

- Supported commands:
  - `format <size>` – format the file system on a disk of the given size (erases all data)
  - `mkdir a1` – create a directory
  - `rmdir a1` – delete an empty directory
  - `cd a1` – change the current directory
  - `pwd` – show the current path
  - `incp s1 s2` – import a file from the host disk into the file system
  - `outcp s1 s2` – export a file from the file system to the host disk
  - `cp s1 s2` – copy a file
  - `mv s1 s2` – move or rename a file
  - `rm s1` – delete a file
  - `ls a1` – list directory contents
  - `cat s1` – display file contents
  - `info s1/a1` – information about a file (which clusters it occupies)
  - `load s1` – execute commands from a file
  - `check` – check file system consistency
  - `bug s1` – intentionally corrupt a file (to simulate an error)
  - `exit` – exit the program

> In all cases, `s1`, `s2`, and `a1` represent paths to files or directories in the virtual file system.

The program creates a virtual disk in the form of a binary file, to which it writes data and from which it reads. When an invalid command is entered (unknown command, invalid arguments, ...), the program prints an error message and keeps running. For practical reasons, the program limits the disk size to 4 GiB.

3. System Architecture

Main Structures

The file system is structured into several key parts:

- Main file system metadata:
    - Structure (`struct`) `FileSystem`
    - Contains information defining the layout of the file system on disk.
    - The structure is 31 bytes long and contains the following information:
        - disk size,
        - number of FAT tables,
        - starting addresses of the FAT tables,
        - starting address of the data region,
        - cluster size,
        - identifier of the file system's author.

```
// FileSystem is a struct representing the pseudo FAT file system. It
is 31 bytes long.
//
// WARNING: The variables are ordered in a way that they are aligned
in memory with the
// smallest possible padding. This is important for the byte handling
in the loader.go.
type FileSystem struct {
  // DiskSize is the size of the disk in bytes
  DiskSize uint32
  // FatCount is the number of records in the FAT
  FatCount uint32
  // Fat01StartAddr is the start address of the first FAT
  Fat01StartAddr uint32
  // Fat02StartAddr is the start address of the second FAT
  Fat02StartAddr uint32
  // DataStartAddr is the start address of the data region
  DataStartAddr uint32
  // ClusterSize is the size of a cluster in bytes
  ClusterSize uint16
  // Signature is the ID of the author of the file system
  Signature [consts.StudentNumLen]byte
}
```

> All variable sizes were chosen to save as much disk space as possible given the maximum file system size. The cluster size was chosen based on the average file size in a typical user environment.

- FAT tables:
  - A two-dimensional `int32` array (FAT typically uses 2 tables) with a length corresponding to the number of clusters.
  - Used to map allocated and free clusters on the virtual disk to the data region.
  - Special values denote unused blocks (`FatFree int32 = -1`), end of file (`FatFileEnd int32 = -2`), and bad blocks (`FatBadCluster int32 = -3`).
- Data region:
  - A byte array of length equal to the allocatable space on the disk (number of clusters × cluster size).
  - Contains metadata of files and directories themselves, including their contents (for files).
  - Each file or directory keeps a self-reference in its starting cluster (as a `DirectoryEntry` structure).
  - Then:
    - if it's a file, the following clusters contain the file's data;
    - if it's a directory, the following clusters contain references to files and directories contained within it.
  - Data in the data region is accessed via the FAT tables:
    - The offset of each record's beginning in the array is computed as `(clusterIndex * ClusterSize)`.

Secondary Structures

Other important structures:

- `DirectoryEntry` record for a file or directory:
  - A 20-byte structure containing metadata about the file or directory.
  - The structure contains the following information:
    - name of the file or directory,
    - flag indicating whether it is a file,
    - file size,
    - starting cluster of the file,
    - starting cluster of the parent directory.

```
// DirectoryEntry is a struct representing an item in a directory. It
is 20 bytes long.
type DirectoryEntry struct {
  // Name is the name of the file or directory
  Name [consts.MaxFileNameLength]byte
  // IsFile is a flag indicating if the item is a file
  IsFile bool
  // Size is the size of the file in bytes
  Size uint32
  // StartCluster is the start cluster of the file
  StartCluster uint32
  // ParentCluster is the start cluster of the parent directory
```

```
  ParentCluster uint32
}
```

> Each `DirectoryEntry` record occupies 1 cluster. This choice follows the FAT standard. It means that if the file system consists mostly of directories or very small files, disk space would be used inefficiently. However, this limitation is sufficient for the purposes of this work.

> The root directory is represented as a record named `/` with starting cluster 0. Its parent cluster points to itself. The program enforces constraints so that the root directory cannot be deleted or otherwise modified.

How the File System Works

The file system allocates files and directories within fixed-size clusters that are linked via the FAT table. Each file or directory is stored in one or more clusters, and the FAT maintains the sequence.

1. Creating a file/directory

Creating a file proceeds in several steps:

- Finding free space in the parent directory.
  - The system processes the given path and tries to obtain the parent directory for the target record (if the path is valid).
  - It loads the directory structure of the parent and checks whether the target name already exists among the children.
  - If it does not exist, a new record is added to the children (including updating the FAT table).
- Allocating the first cluster for the file/directory itself.
  - The system scans the FAT table to find the first free cluster.
  - This cluster is stored as the starting cluster (`StartCluster`) in the `DirectoryEntry`.
- Saving file data.
  - If the file occupies more than one cluster, each subsequent cluster is linked in the FAT table to the previous one.
  - The last cluster in the chain contains the `FatFileEnd` marker.

2. Reading a file/directory

Reading a file/directory proceeds as follows:

- Locating the file/directory.
  - The system attempts to find the record based on the provided path.
  - Reading the contents through the FAT table:
    - The starting cluster (`StartCluster`) is used as the entry point.
    - From the first cluster, the `DirectoryEntry` representing the file/directory is loaded.
    - Any subsequent clusters represent the directory's children or the file's data.
    - The system traverses the FAT table and reads data from all allocated clusters until it encounters `FatFileEnd`.

3. Deleting a file/directory

When deleting a file, the following steps are required:

- Locating the file/directory:
  - The system parses the path and finds the corresponding `DirectoryEntry` record in the parent directory.
  - If it is a directory, it must be verified that it is empty (contains no files or subdirectories).
- Removing the record from the parent directory:
  - The `DirectoryEntry` corresponding to the file or directory is removed from the parent directory (including updating the FAT table—shortening the cluster chain by one cluster).
  - The record's area in the data section is zeroed out.
- Releasing clusters:
  - Each cluster used by the file or directory is marked as `FatFree` in the FAT table.
  - The record's area in the data section that belongs to the file/directory according to the FAT table is zeroed out.

> Deleting directories creates free holes in the data region and FAT tables. Inserting new, larger records later causes fragmentation of the file system. Fragmentation is expected and common in FAT systems. In a real file system, it would be necessary to implement a defragmentation mechanism to ensure that files are stored as efficiently as possible.

4. Moving a file/directory

When moving a file or directory, the following steps occur:

- Finding the source record:
  - The system finds the corresponding `DirectoryEntry` of the file or directory.
- Checking the existence of the target location:
  - The system ensures the target path exists and that it does not already contain a file/directory with the same name.
- Creating a new record in the target directory:
  - A new `DirectoryEntry` is added to the target directory, referencing the same data as the original record.
- Removing the old record:
  - The original `DirectoryEntry` is removed from the source directory.
  - The FAT table is updated:
    - Remove a cluster from the parent directory.
    - Add a cluster to the target directory.
    - The cluster chains that contain the file/directory data remain unchanged.

5. Copying a file/directory

Uses the principles of creating a new record. Only the contents of the file/directory are copied to the new location, and the starting clusters are changed (according to the free space found).

4. Setup Guide

Requirements

To build and run the project, you need:

- Go 1.22 or newer
- make
    - > On Windows you can use, for example, make from Chocolatey, or other alternatives.

Building the project

To build the whole project, *Makefile* and *Makefile.win* were created with instructions for Unix and Windows OS. To build on Unix-like OS, run from the project root:

```
make
```

and on Windows, run:

```
make -f Makefile.win
```

The script builds an executable into the *bin/* folder. The executable is named *myfs*, or on Windows *myfs.exe*.

Running the program

Run the program with:

```
./bin/myfs <path to virtual disk>
```

> Files copied into the virtual disk and vice versa are always copied relative to the folder where the program is launched.

Running manually via Go

As an alternative to building, you can run the project directly with Go. It must be run from the *src/* folder. Start it with:

```
go run main.go <path to virtual disk>
```

5. Project Structure

The project is divided as follows:

- *Root folder* — Contains build files, the *src/* folder with the project's source code, the *bin/* folder for build outputs, and the *docs/* folder with documentation.

Root folder

- *Makefile* — Build file for Unix OS.
- *Makefile.win* — Build file for Windows OS.
- *docs/* — Folder containing documentation.
    - *docs/doc.md* and *docs/doc.pdf* — This document in Markdown and PDF format.
    - *docs/client_ref.html* — Link to client-side documentation.
    - *docs/server_ref.html* — Link to server-side documentation.
- *src/* — Folder containing the project source code.
    - *src/arg_parser/arg_parser.go* — Module for processing command-line arguments.
    - *src/cmd/* — Folder containing modules for processing commands.
        - *src/cmd/command.go* — Definition of the command structure.
        - *src/cmd/command_parser.go* — Module for parsing commands.
        - *src/cmd/command_validator.go* — Module for validating commands.
        - *src/cmd/command_executor.go* — Module for executing commands.
    - *src/consts/* — Folder containing constants used in the project.
        - *src/consts/cmds.go* — List of supported file system commands.
        - *src/consts/exit_codes.go* — Program exit codes.
        - *src/consts/fat_flags.go* — Constants defining special values in the FAT table (e.g., `FatFileEnd`, `FatFree`).
        - *src/consts/formats.go* — Definitions of all units and supported characters/symbols.
        - *src/consts/limits.go* — Defined file system limits (e.g., maximum filename length, ...).
        - *src/consts/msg.go* — Text messages used for outputs.
    - *src/custom_errors/errors.go* — Defines custom error types and error message constants used in the project.
    - *src/logging/logging.go* — Logging module, allows writing messages of different levels (INFO, WARNING, ERROR, ...).
    - *src/pseudo_fat/structures.go* — Definitions of the core data structures of the file system, including `FileSystem` and `DirectoryEntry`.
    - *src/utils/* — Folder with helper utilities for working with the file system.

        - *src/utils/data_transform.go* — Tools for converting data between formats, e.g., serialization and deserialization of binary structures.
        - *src/utils/loader.go* — Module for loading the file system into a binary file.
        - *src/utils/path.go* — Tools for processing file and directory paths.
        - *src/utils/pretty_print.go* — Module for formatting output and clearly displaying information for certain structures.
        - *src/utils/pseudo_fat_fs_operations.go* — Implementation of algorithms for working with the file system.
            - **This is where all supported commands are implemented, such as creating, deleting, copying, moving files and directories, working with directories, loading and saving data, ...**

    - *src/main.go* — The project's main file, which initializes the file system, reads commands, and ensures their execution.

    - *src/test.cmds*, *src/testh.cmds*, *src/testr.cmds* — Files containing test command sets to verify the correct behavior of the file system implementation (via the `load` command).

6. Implementation Details

The implementation description primarily covers the most important parts of the code needed to understand how the application works. The project is structured to enable efficient command processing and manipulation of the file system.

Main components and their roles

The project is divided into several key modules that closely cooperate:

- *main.go* — the application's main entry point; initializes the file system, starts goroutines for command processing, and manages program termination.
- *cmd/* — contains logic for parsing, validating, and executing commands.
- *pseudo_fat/* — definitions of the pseudo-FAT file system, including core data structures.
- *utils/* — helper functions for working with paths, data conversions, and **file system management**.
- *logging/* — handles system event logging.

Command execution flow

1. Application initialization (`main.go`)

    - The program starts by processing command-line arguments.
        - `arg_parser.GetFilenameFromArgs()`
    - Checks whether the file representing the file system exists; if necessary, creates a new one.
        - `getFileFromPath()`.
    - Then it attempts to load the file system into memory (or an uninitialized file system if the file contained no data).
        - `utils.GetFileSystem()`.
    - Goroutines are created for reading commands (`acceptCmds()`) and executing them (`interpretCmds()`).

- The program waits in (`handleProgramTermination()`) until terminated by the user (`exit` command or sending a `SIGINT` signal).

2. Reading commands (`acceptCmds()`)

   - The `acceptCmds()` goroutine continuously reads user input (`bufio.Scanner`).
   - Each input is parsed (`cmd.ParseCommand()`).
     - If a command is valid (`cmd.ValidateCommand()`), it is sent to the `cmdBufferChan`.

3. Executing a command (`interpretCmds()`)

   - The `interpretCmds()` goroutine reads commands from `cmdBufferChan`.
   - Each command is passed for execution (`cmd.ExecuteCommand()`).
     - `cmd.ExecuteCommand()` processes the command, performs simpler operations (changing the current directory, listing a directory, ...) or calls the corresponding function from *utils/pseudo_fat_fs_operations.go* to perform the operation on the file system.
     - On error, an appropriate error message is printed (*custom_errors*).
     - If the command succeeded, any changes are written to the file system (`utils.WriteFileSystem`).

The implementation of the actual file system operations is located in *utils/pseudo_fat_fs_operations.go*. Most functions are aptly named after the command they implement (e.g., `Mkdir()`, `Rmdir()`, `CopyInsideFS()`, ...).

> In production, functions in this file would need to be decomposed further, but to help the reader quickly navigate the operations required in the file system, the functions were left in a less modular form (so the reader does not have to jump between files/lines to different functions).

Bug when copying file data to the data region

During review by the assigner, a bug was found when writing a file to the file system. There was an error while copying data into memory—specifically when using Go's `copy()` to transfer data from the source array (`fileDataRef`) to the data region (`dataRef`).

The issue was that the copy range did not have a limiting end value set. This led Go to copy more data than intended, overwriting clusters that did not belong to the file. This corrupted the file system.

Code before the fix:

```
// write the file data to the filesystem
prevIndex := freeClusterIndex
for i, clusterIndex := range freeClusterIndicesData {
  addToFat(fatsRef, prevIndex, clusterIndex)
  byteOffset = int(clusterIndex) * int(pFs.ClusterSize)
  copy(dataRef[byteOffset:], fileDataRef[i*int(pFs.ClusterSize):])
  prevIndex = clusterIndex
}
```

Code after the fix:

```
// write the file data to the filesystem
bytesRemaining := len(fileDataRef)
prevIndex := freeClusterIndex
for i, clusterIndex := range freeClusterIndicesData {
  addToFat(fatsRef, prevIndex, clusterIndex)
  byteOffset = int(clusterIndex) * int(pFs.ClusterSize)
  fileDataRefStartOffset := i * int(pFs.ClusterSize)
  fileDataRefEndOffset :=
    min((i+1)*int(pFs.ClusterSize),
        fileDataRefStartOffset+bytesRemaining
       )
  currentSourceBytes :=
    fileDataRef[fileDataRefStartOffset:fileDataRefEndOffset]
  copiedBytesCount := copy(dataRef[byteOffset:], currentSourceBytes)
  bytesRemaining -= copiedBytesCount
  prevIndex = clusterIndex
}
```

This bug is analogous to the well-known issue in C when using `strcpy()` instead of the safer `strncpy()`, which can lead to buffer overflows.

7. Testing

Project testing was performed manually through user-level testing. No automated tests were implemented. Testing focused on verifying correct behavior of basic file system operations such as creating, deleting, copying, and moving files and directories. Correct path handling, input validation, and error detection were also tested.

8. Conclusion

This project implements a simplified file system based on the FAT (File Allocation Table) principles, using Go for efficient memory management and manipulation of binary data. The goal was to create a functional and understandable simulation of a file system that allows the user to perform basic operations on a virtual disk, such as creating, deleting, moving, and copying files and directories.

Several key challenges had to be addressed in the design and implementation, including:

- Management of clusters and FAT tables, including proper linking of cluster chains for files and directories.
- Efficient work with binary data, including serialization and deserialization of file system structures.
- Ensuring file system integrity and error detection, for example when manipulating files and their data blocks.
- Validation and processing of paths, including support for both relative and absolute paths.

The project is designed modularly, enabling easy extension and customization. The code structure is clearly divided among modules for command handling, the file system, paths, and helper functions, ensuring good readability and maintainability.

Summary of achieved goals:

- Basic file system operations: Implemented common operations such as `mkdir`, `rm`, `mv`, `cp`, `ls`, `pwd`, `info`, and others.
- Work with a virtual disk: Supports formatting the file system and managing files and directories.
- Handling common errors: Input validation, file system consistency checks (`check`), error detection and simulation (`bug`).
- Testing and debugging: Errors were discovered and fixed, for example the memory overwrite issue when copying files.
- Clear and maintainable architecture: Modular source code layout, well-defined interfaces between components.

The project is ready for further extensions such as implementing defragmentation, adding new file types like symbolic links, and more.

The project fulfilled its main purpose—implementing a functional file system and enabling its effective management. Although it is only a simulation, its design and implementation required solving many real-world problems common in the field of file and data management. This project provided valuable experience with the design and implementation of a low-level file system.