University of West Bohemia

Faculty of Applied Sciences

KIV/ZOS Fundamentals of Operational Systems

Semester Project

Filesystem on i-nodes

Student: Malakhov Leonid
Student ID: A22B0387P
Date: December 28, 2025

1 Introduction

The goal of this semester project is to design and implement a simplified virtual file system based on the concept of i-nodes. The project focuses on the internal structure and behavior of a file system rather than on interaction with a real operating system kernel.

The resulting application simulates a standalone virtual file system stored inside a single host file. Users interact with the system through a custom command-line shell that provides basic file system operations such as file creation, deletion, copying, directory management, and data inspection.

The project emphasizes correctness, clear separation of responsibilities, and consistency with the provided specification.

2 Problem Analysis and Requirements

The assignment requires the implementation of a simplified file system using i-nodes as the primary abstraction for files and directories. The system must support both absolute and relative paths and provide a fixed set of commands with a strictly defined output format.

Core functional requirements include:

- Creation and deletion of files and directories
- File copying, moving, and concatenation
- Import and export of files between the host system and the virtual file system
- Navigation within the directory hierarchy
- Display of file system metadata and statistics

The file system must persist its state inside a single container file and allow repeated mounting and unmounting during program execution.

3 Used Technologies and Tools

The project is implemented in the C programming language. This choice allows direct control over memory management and data structures, which is essential for simulating low-level file system behavior.

The development was performed using:

- Standard C library for file I/O and memory handling
- Modular source code organization across multiple layers
- Command-line interface for user interaction

The virtual file system is stored in a regular host file, enabling easy testing and portability across systems.

4 Project Architecture

The application is structured as a layered system, where each layer has a clearly defined responsibility. This approach improves readability, maintainability, and separation of concerns.

4.1 Layered Architecture Overview

The system is divided into the following logical layers:

- Shell layer
- Logic layer
- Metadata layer
- Disk layer

The shell layer handles user input and output formatting. The logic layer implements file system operations and enforces semantic rules. The metadata layer manages i-nodes, directories, and allocation structures. The disk layer performs low-level block access within the file system container file.

4.2 Example of Layer Interaction

To demonstrate how individual layers of the system interact, this section describes the execution flow of the `cp` command, which copies a file inside the virtual file system.

The `cp` command is a representative example because it involves path resolution, inode lookup, data reading and writing, as well as error propagation across multiple layers.

4.2.1 Shell Layer

The execution begins in the shell layer, which is responsible for parsing user input and dispatching commands.

After the user enters a command in the interactive shell, the input is parsed and validated. For the `cp` command, the shell invokes the corresponding logic-layer function:

```
fs_copy(source_path, destination_path);
```

At this stage, the shell layer does not perform any filesystem logic. Its responsibility is limited to argument validation and printing the final result returned by lower layers.

4.2.2 Logic Layer

The logic layer implements the semantic behavior of the command.

First, both source and destination paths are normalized into absolute paths. This ensures consistent internal processing regardless of whether the user provides relative or absolute paths.

Next, the logic layer performs the following steps:

- Resolve the source path and locate the corresponding i-node
- Verify that the source exists and represents a regular file
- Resolve the destination parent directory
- Create a new i-node for the destination file

After validation, the logic layer coordinates the data transfer by reading file contents from the source inode and writing them into the newly created destination inode.

The logic layer does not directly manipulate disk structures. Instead, it delegates low-level operations to the metadata and disk layers.

4.2.3 Metadata Layer

The metadata layer manages all inode-related structures and directory contents.

During the execution of the `cp` command, this layer is responsible for:

- Mapping paths to inode identifiers
- Allocating a new inode for the destination file
- Updating directory entries
- Providing inode metadata to higher layers

By centralizing inode management, the metadata layer ensures filesystem consistency and prevents direct manipulation of internal structures from higher layers.

4.2.4 Disk Layer

The disk layer represents the lowest level of the system and operates directly on the filesystem container file.

For the `cp` command, the disk layer:

- Reads data blocks referenced by the source inode
- Allocates free data blocks for the destination file
- Writes file data into newly allocated blocks

All block-level operations are isolated within this layer, allowing higher layers to work with abstract file and inode representations instead of raw disk offsets.

4.2.5 Returning Results to the Shell

Once the copy operation is complete, control flows back upward through the layers.

Each layer returns a status code indicating success or a specific error condition. The shell layer interprets this code and prints a predefined message such as:

```
OK
FILE NOT FOUND
PATH NOT FOUND
```

This design ensures a clear separation between internal logic and user-facing output while maintaining a predictable command interface.

4.3 Filesystem Data Model

The virtual file system is based on a simplified i-node data model.

Each filesystem object, whether a regular file or a directory, is represented by a single i-node. An i-node stores metadata such as object type, file size, and references to data blocks.

Directories are implemented as special files containing directory entries that map filenames to i-node identifiers. This design enables hierarchical organization and efficient path resolution.

Regular files use a combination of direct and indirect block references. Direct blocks provide fast access for small files, while an indirect block allows the file to grow beyond the direct addressing limit.

4.4 Filesystem Lifecycle

The filesystem follows a well-defined lifecycle that governs its availability and consistency.

At program startup, the filesystem container is mounted and validated. If the container file is missing or contains invalid metadata, the filesystem enters a restricted state.

In this restricted state, all filesystem operations are disabled except for formatting. The `format` command initializes a new filesystem structure and transitions the system into an operational state.

During normal operation, the filesystem remains mounted and processes user commands. On program termination, the filesystem is cleanly unmounted, and all modified metadata and data blocks are flushed to persistent storage.

This explicit lifecycle management prevents inconsistent states and ensures data durability.

4.5 Filesystem Formatting and On-Disk Layout

Formatting the filesystem initializes a fixed internal layout inside the container file. During this process, the file is divided into several logically distinct regions, each serving a specific purpose.

The following structures are created and initialized during formatting:

- Superblock
- Inode bitmap
- Data block bitmap
- Inode table
- Data block area

5 User Interaction and Program Workflow

The application is executed from the command line and requires a single mandatory argument: the name of the file that serves as a container for the virtual file system.

```
./vfs <filesystem_file>
```

This file represents the persistent storage of the virtual file system, including all metadata, inodes, and data blocks.

5.1 Filesystem Initialization and Formatting

After startup, the application attempts to mount the specified filesystem container. If the file does not exist or if its internal metadata does not match the expected format (identified by a predefined magic number), the filesystem is considered invalid.

In this state, the filesystem refuses to process any commands except for formatting. The user is explicitly instructed to initialize the filesystem using the `format` command.

```
format <sizeMB>
```

Formatting creates a new filesystem structure inside the container file, initializes all metadata, and prepares the system for further interaction. Only after successful formatting does the filesystem become fully operational.

This mechanism prevents accidental operations on invalid or corrupted filesystem data and enforces a well-defined initialization workflow.

5.2 Interactive Shell

Once the filesystem is successfully mounted, the application enters an interactive shell mode.

The shell operates as a read–evaluate–print loop (REPL), where the user enters commands line by line and immediately receives feedback. The shell maintains a current working directory, allowing the use of both absolute and relative paths.

All commands follow a fixed syntax and produce standardized output messages as required by the assignment specification.

5.3 Supported Commands

After successful initialization, the user can interact with the filesystem using the following commands:

- `ls [path]` Lists the contents of a directory.
- `cd <path>` Changes the current working directory.
- `pwd` Displays the current working directory.
- `mkdir <path>` Creates a new directory.
- `rmdir <path>` Removes an empty directory.
- `rm <path>` Removes a regular file.
- `cp <source> <destination>` Copies a file within the virtual filesystem.
- `mv <source> <destination>` Moves or renames a file or directory.
- `cat <path>` Displays the contents of a file.
- `xcp <file1> <file2> <destination>` Creates a new file as a concatenation of two existing files.
- `add <file1> <file2>` Appends the contents of one file to another.
- `incp <host_file> <vfs_path>` Imports a file from the host filesystem into the virtual filesystem.
- `outcp <vfs_file> <host_path>` Exports a file from the virtual filesystem to the host filesystem.
- `info <path>` Displays detailed inode information, including referenced data blocks.
- `statfs` Displays global filesystem statistics such as block and inode usage.
- `load <script_file>` Executes commands from a text file sequentially.

- `format <sizeMB>` Reinitializes the filesystem with the specified size.
- `exit` Unmounts the filesystem and terminates the application.

Each command validates its input parameters and reports errors using predefined messages, ensuring consistent and predictable user interaction.

5.4 Error Handling and Feedback

The shell provides immediate feedback for all user actions. Invalid commands, missing parameters, incorrect paths, or unsupported operations are detected and reported without affecting filesystem consistency.

By strictly separating command parsing, filesystem logic, and data storage, the system ensures that user errors do not propagate into internal structures.

6 Testing

Testing was performed manually by executing individual commands and command sequences through the interactive shell. Special attention was paid to boundary cases such as:

- Operations on non-existing paths
- File size limits
- Directory emptiness constraints
- Insufficient free space or i-nodes

The consistency of the file system was verified by repeatedly mounting, unmounting, and inspecting metadata using diagnostic commands.

7 Limitations and Known Issues

The file system is intentionally simplified and does not support features such as permissions, symbolic links, or concurrent access.

All data is loaded and written synchronously, which limits performance but improves reliability and predictability.

Fragmentation is not explicitly handled, and block allocation follows a straightforward strategy.

These functions were not declared as a part of course project.

Also, it is necessary for the user to properly exit the program by command exit. Otherwise, data will not be synchronized and will not be correctly loaded after the next startup.

8 Possible Extensions

Future improvements could include:

- Support for file permissions and ownership
- Improved block allocation strategies
- Defragmentation mechanisms
- Extended diagnostic and debugging tools

These extensions would move the system closer to real-world file system behavior.

9 Conclusion

The project successfully implements a simplified virtual file system based on i-nodes, fulfilling all core requirements of the assignment.

During development, a strong emphasis was placed on clean architecture, clear responsibility separation, and robust error handling.

The project provided practical insight into internal file system structures and low-level data management, reinforcing theoretical knowledge with hands-on implementation.