Customized Virtual File System

Dr. Gauri Ghule[1], Dr. Pallavi Deshpande[2], Shahabaz Sayyad[3], Amol Thorat[4], Dheeraj Lokhande[5]

[1,2]Asst Professor, Department of Electronics and Telecommunication Vishwakarma Institute of Information and technology, India
[3,4,5]Student, Department of Electronics and Telecommunication, Vishwakarma Institute of Information and Technology, India

Abstract
The superblock, inode structure, file table, and user file descriptor table (UFDT) are among the fundamental elements of the CVFS architecture. These components are all vital to the management of file metadata, open files, and user interactions. The CVFS implements a number of file operations, such as creation, opening, reading, writing, and deletion, to enable smooth file administration.
We evaluate the functionality and performance of the constructed CVFS by a set of methodical experiments and evaluations, with particular attention to file manipulation, error management, and resource utilisation. We also talk about possible improvements and optimisations for upcoming versions of the CVFS, with the goal of enhancing scalability, dependability, and user experience in general.

Keywords: Error Handling, File System, Metadata, Virtual File System, File Operations

1. Introduction

The project's goal is to use the C programming language to create and implement a virtual file system (CVFS). Users can conduct file operations in a controlled environment by using a virtual file system, which acts as an abstraction layer between the user and the real file system. The project's main goal is to develop a reliable and effective CVFS that can handle file creation, deletion, reading, writing, and other tasks that are frequently performed by conventional file systems.To effectively handle file operations, disc space allocation, and file metadata, the implementation makes use of a variety of data structures and algorithms.
The superblock, inode structure, file table, and user file descriptor table (UFDT) are important parts of the CVFS, each with a distinct role in the construction and functionality of the file system.The CVFS initialises its data structures and allots the resources required to manage files efficiently upon programme beginning. Through a command-line interface, users can interact with the CVFS by providing commands to carry out file operations like creating, opening, reading, writing, and deleting files. To handle exceptions and guarantee the integrity of file operations, the CVFS makes use of error handling techniques.To give customers complete file management skills, the project also contains features like file listing, file statistics display, file truncation, and error reporting.Furthermore, a help system is put in place to aid users in comprehending the syntax and use of accessible commands, improving the CVFS's usability and user experience.

2. Objectives

- This project simulates all data structures needed by the operating system to manage file system activity.
- Understanding the underlying concepts that govern file system structure and function is a critical goal.
- Implementation of essential file system components, including file tables and superblocks, is planned.
- Our primary focus is on developing procedures for file generation, opening, reading, writing, and searching.
- The project involves creating a command-line interface to manage files and interact with CVFS.
- Establish robust error-handling processes to ensure the file system's stability and resilience.
- Users will receive clear feedback, including error messages and command instructions.

3. Methodology

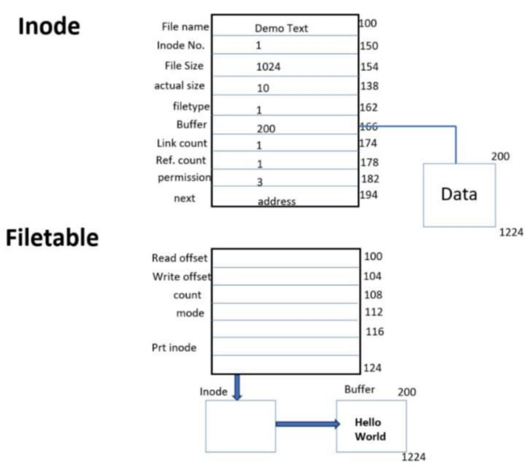
The implementation of the Virtual File System (CVFS) is structured to ensure a systematic development process. It begins with a thorough study of project specifications to determine the critical elements required for effective file management. During the design process, essential features such as superblocks, file tables, and inodes are rigorously defined, as are the data structures that represent them. A well-defined set of functions provides functionality for file operations and system management.

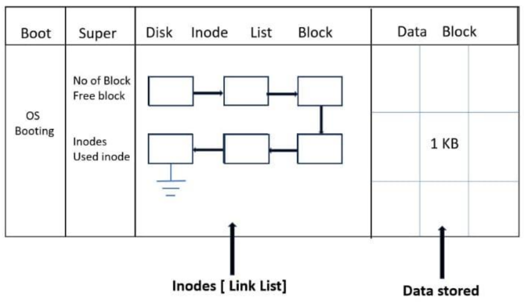
Furthermore, a command-line interface (CLI) is developed to allow users to communicate with the CVFS, and robust error-handling methods are included to resolve any potential concerns. Testing processes are stringent, including unit tests for individual functions, integration tests to check component interactions, and extensive scenario-based testing to assure robustness and dependability.

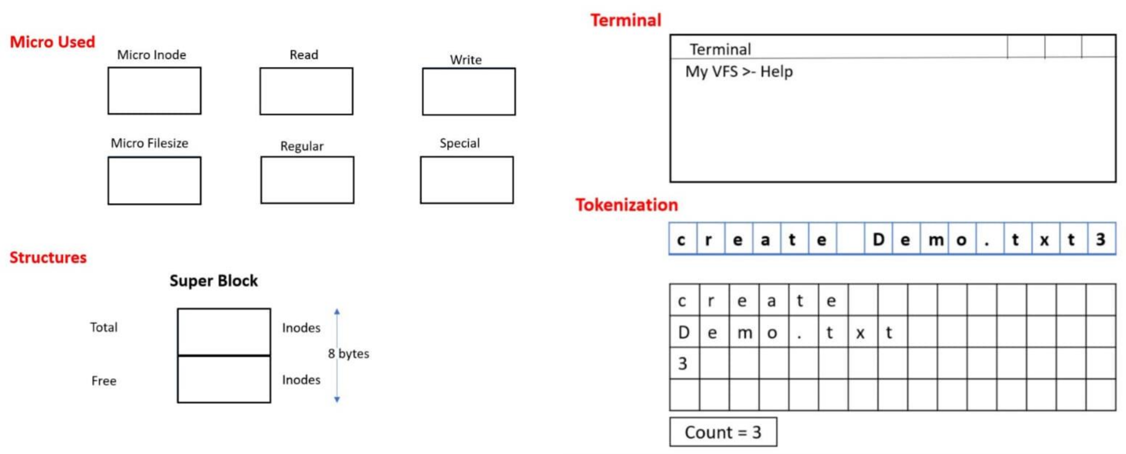
Documentation is critical for capturing design rationale, code implementation details, and usage instructions, all of which help with system knowledge and maintenance. After testing, the code is thoroughly reviewed and optimised to ensure functionality and maintainability.

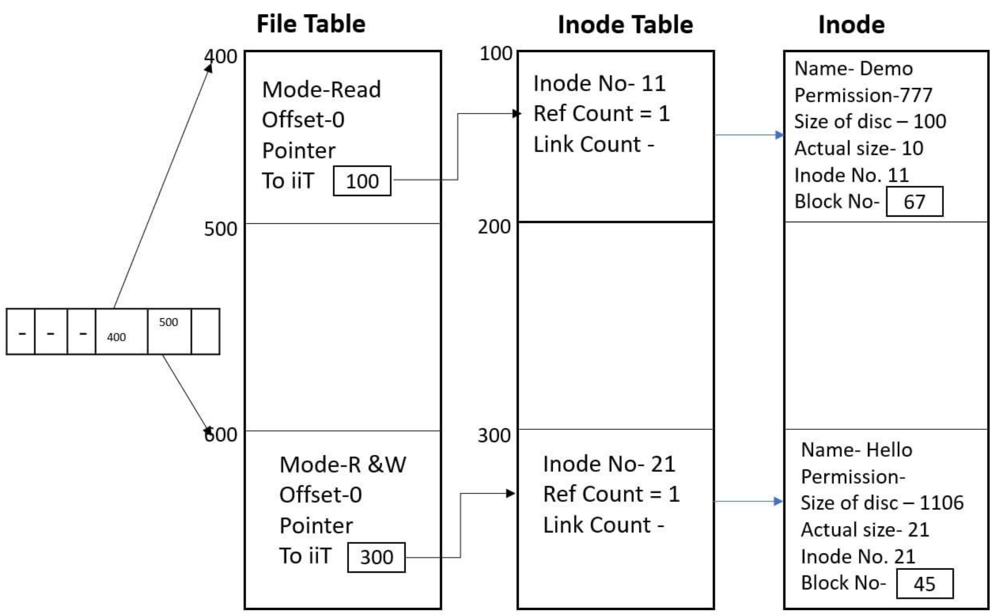
During the deployment phase, both the code and documentation are packaged for distribution, and users are provided with extensive support to ensure that the CVFS is deployed and used efficiently. This rigorous approach ensures the construction of a robust and efficient CVFS that aligns with project objectives.

4. System Implementation









4.1. Structures:

- The superblock (struct superblock) contains information on the overall number of inodes and free inodes in the file system.
- Inodes (struct inodes) represent files and directories in the file system. It maintains metadata about the file, including the filename, inode number, file size, real file size, file type, buffer (for file content), link count, reference count, and permissions.
- The file table (struct filetable) stores information about open files, including read and write offsets, mode (read, write, read/write), and a pointer to the inode.
- The User File Descriptor Table (UFDT, struct ufdt) manages open files through an array of file tables.

4.2. Functions:

- CreateDILB() initialises the Disc Inode List Blocks (DILB) by generating a linked list of inodes.
- InitialiseSuperBlock(): Starts the superblock with the total number of inodes and sets the number of free inodes to its maximum.
- CreateFile(char *name, int permission): Creates a new file with the given name and permissions. It creates an inode for the file, updates the file table, and assigns RAM to the file buffer.

- rm_File(char *name): Removes a file from the file system by reducing its link count. If the link count hits 0, the inode is classified as free.
- ReadFile(int fd, char *arr, int isize): Reads data from an open file into the specified buffer.
- WriteFile(int fd, char *arr, int isize): Transfers data from the supplied buffer to an open file.
- OpenFile(char *name, int mode) opens an existing file with the supplied name and mode (read, write, or read/write). It creates a file table entry and increases the inode reference count.
- CloseFileByName(char *name): Closes an open file by reducing its reference count and removing its read and write offsets.
- CloseAllFile(): Closes all open files by looping through the UFDT and clearing their read and write offsets.
- LseekFile(int fd, int size, int from): Changes the read or write offset of an open file to the supplied size and from location (start, current, end).
- ls_file(): Lists all files in the file system, including their metadata.
- Fstat_file(int fd): Displays statistical information about an open file based on its file descriptor.
- stat_file(char *name): Displays statistics information about a file based on its name.
- truncate_File(char *name): Truncates a file by clearing its buffer, read offset, write offset, and actual file size.

4.3. Main Function:

- The main function offers a command-line interface for users to interact with their file system.
- Command Parsing: Parses user commands and calls relevant routines.
- The main function handles problems and shows notifications for invalid parameters or unsupported requests.

4.4. Command Line Interface (CLI):

- User Commands: Using commands like create, open, read, write, close, and rm, users can conduct a variety of file operations such as creating, opening, reading, writing, and deleting files.
- Help and Exit: Users can access help information with the help command and exit the file system by using the exit command.

4.5. Error Handling:

- Error Messages: The implementation generates descriptive error messages for various circumstances such as invalid parameters, permission refused, file not found, end of file reached, memory allocation failure, and so on.
- Functions report error codes to reflect the type of error encountered during execution.

5. Software & Hardware Requirements

Software Requirements:

- Operating System: The project works with any operating system that supports C programming, such as Linux, Windows, and macOS.
- A C compiler is required to compile and run the source code. Popular options include GC (GNU Compiler Collection) for Linux and MinGW for Windows.
- Text Editor or IDE: Any text editor, such as Vim, Emacs, or an Integrated Development Environment (IDE), like Visual Studio Code, Eclipse, or Code:Blocks can be used to modify the source code.

Hardware Requirements:

- Processor: Any modern processor that can run the given operating system and compiler efficiently.
- Memory (RAM): Enough memory to compile and run the programme efficiently. This is a simple file system project, thus memory requirements are small.
- Storage: Enough capacity to store the source code, the resulting executable, and any necessary documentation or test files. The storage needs for this project are minimal.

6. Future Scope

The future scope of your virtual file system project is broad, with several opportunities for extension and refinement. To begin, consider improving security measures by including advanced encryption techniques and access control systems to protect sensitive data. Consider cloud integration to enable seamless backup, synchronisation, and access from multiple devices and locations. Scalability and performance should also be prioritised, with an emphasis on providing caching techniques and performance advancements that can efficiently handle higher file volumes and concurrent accesses. Cross-platform compatibility is critical for increasing user accessibility, which necessitates the creation of platform-specific drivers or adherence to cross-platform file system standards. Advanced file system operations like as versioning, snapshotting, and deduplication can help to improve data management capabilities.

Furthermore, exploring distributed file system architectures for fault tolerance and scalability, as well as incorporating machine learning techniques for predictive caching and intelligent data management, are fascinating topics of investigation. Improvements in user interface design and compliance with data protection standards should be explored, as well as the creation of a thriving community around the project to encourage cooperation and sustainability. By following these steps, your virtual file system project can continue to evolve and meet the changing needs of users and technology improvements.

7. Conclusion

- In conclusion, while the C-based Virtual File System (CVFS) provides basic file management capabilities and error handling, there are major opportunities for future improvements to increase its utility and durability.
- Integrating directory administration with hierarchical file organisation would increase overall efficiency, but combining access control techniques such as file permissions and access control lists (ACLs) would improve security and user management. Furthermore, using concurrency control mechanisms like file locking would protect data integrity in multi-threaded or multi-process contexts.
- Advanced features such as symbolic connections, mounting capabilities, and extended attributes have the potential to broaden the CVFS's capabilities even more. Furthermore, enhancing error handling, optimising speed, and boosting security measures would help to create a more dependable and efficient system.
- Expanding support for other file types and network file systems will increase the CVFS's flexibility, while adding graphical options to the user interface would improve experience and accessibility.
- By addressing these areas for improvement, the CVFS has the potential to evolve into a more comprehensive and mature system that can better fulfil the different needs of its users and environments.

8. Acknowledgement

I express heartfelt gratitude to Professor Dr. Gauri Ghule and Dr. Pallavi Deshpande for his invaluable guidance and thorough review, which significantly contributed to the refinement of our customized virtual file system. Additionally, I appreciate the assistance and encouragement from my peers, whose collaboration was essential to the project's success.

9. References

1. Tanenbaum, Andrew S., and Albert S. Woodhull. Operating systems: design and implementation. Vol. 68. Englewood Cliffs: Prentice Hall, 1997
2. Love, Robert. Linux kernel development. Pearson Education, 2010.
3. Wang, Jun, Rui Min, Yingwu Zhu, and Yiming Hu. "UCFS-a novel User-space, high performance, Customized File System for Web proxy servers." IEEE transactions on computers 51, no. 9 (2002): 1056-1073.
4. McKusick, M. K., Bostic, K., Karels, M. J., & Quarterman, J. S. The Design and Implementation of the 4.4BSD Operating System. Addison-Wesley.
5. Robbins, A., & Robbins, H. UNIX Systems Programming: Communication, Concurrency, and Threads. Prentice Hall.
6. Love, R. Linux Kernel Development. Addison-Wesley.
7. Kerrisk, M. The Linux Programming Interface: A Linux and UNIX System Programming Handbook. No Starch Press.
8. Goodheart, B., & Cox, J. The Magic Garden Explained: The Internals of UNIX System V Release 4. Prentice Hall.
9. Marshall, K., McKusick, M. K., & Keith Bostic, M. Design and Implementation of the FreeBSD Operating System. Addison-Wesley.