RV College of Engineering®

(Autonomous Institution Affiliated to VTU, Belagavi)

CREATING YOUR OWN FILE SYSTEM-using file system API, superblocks, inode and data block management.

Experiential Learning Report

Submitted by

RUCHITHA M-1RV22CS165

SANIKA KAMATH -1RV22CS176

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Submitted to

Dr. Jyothi Shetty

Department of Computer Science & Engineering

R V College of Engineering

Contents

INTRODUCTION

A file system is a crucial component of any operating system, responsible for managing how data is stored, organized, and accessed on storage devices such as hard drives, solid-state drives (SSDs), and flash drives. It provides an abstraction layer between the physical storage media and the higher-level software applications.

Key components of a file system include:

1.File System API: An Application Programming Interface (API) defines the interface through which applications interact with the file system. It provides a set of functions or methods for creating, reading, writing, and deleting files, as well as for managing directories and metadata associated with files.

2.Superblock: The superblock is a critical data structure at the beginning of a file system that contains metadata about the file system itself. This metadata includes details such as the total size of the file system, the number of blocks or sectors it occupies, the size of each block, the number of inodes, and pointers to other important data structures within the file system.

3.Inode: Short for "index node," an inode is a data structure that represents a file or directory in the file system. Each inode stores metadata about a specific file or directory, including its permissions, ownership, timestamps (creation, modification, access), size, and pointers to the data blocks that store the actual content of the file or directory.

4.Data Block Management: Data blocks are the fundamental units of storage within a file system. They are used to store the actual contents of files and directories. Data block management involves allocating and deallocating data blocks efficiently to store file data while minimizing fragmentation and optimizing performance.

The file system API serves as the interface through which applications interact with the file system, allowing them to perform operations such as creating, reading, writing, and deleting files. The superblock provides essential metadata about the file system, while inodes represent individual files and directories, storing their metadata and pointers to data blocks. Data block management ensures efficient allocation and utilization of storage space within the file system.

Implementing a file system involves designing and implementing algorithms and data structures to manage these components effectively, ensuring reliable storage and efficient access to data. Different file systems, such as FAT, NTFS, ext4, and ZFS, employ various techniques and optimizations tailored to specific use cases and requirements.

The list of commands we are executing in our file system is:

1. Listing files under a given directory
2. Changing directories
3. Copying files to directories
4. Removing files
5. Removing directories
6. Creating files
7. Creating directories
8. Renaming files
9. Renaming directories
10. Reading files
11. Printing the current directory path
12. Writing contents to the files
13. Finding a specific file or directory.

OBJECTIVES:

1. **Design and Implement File System API**: Develop a set of functions or methods that provide an interface for applications to interact with the file system. These should include operations for creating, reading, writing, and deleting files, as well as for managing directories and metadata.

2. **Create Superblock Structure:** Define the structure and content of the superblock, which holds critical metadata about the file system, such as its size, block size, number of inodes, and pointers to other important data structures. Implement functions to initialize, read, and update the superblock.

3. **Implement Inode Management:** Design data structures and algorithms to manage inodes effectively. Each inode should store metadata about a file or directory, including permissions, ownership, timestamps, and pointers to data blocks. Implement functions to create, read, update, and delete inodes.

4. **File System Operations:** Implement file system operations such as file creation, deletion, reading, writing, renaming, and moving. Ensure that these operations are performed correctly and efficiently while maintaining data integrity.

5. **Testing and Validation:** Develop comprehensive test cases to validate the correctness and robustness of the file system implementation. Test the file system under different scenarios and workloads to ensure that it behaves as expected and can handle various edge cases and failure conditions.

PROBLEM STATEMENT

Our advanced file system management solution is engineered to revolutionize how users interact with their files and directories, delivering a seamless and intuitive experience across diverse operations.

Through an ergonomic design, users effortlessly navigate directories and swiftly locate files, aided by dynamic directory path printing for contextual awareness. This system boasts a comprehensive suite of functionalities, empowering users with efficient file listing, directory navigation, and robust file and directory creation capabilities. Users can perform precise file or directory searches, enhancing productivity by swiftly locating desired items.

Furthermore, advanced functionalities such as file copying, moving, and deletion are seamlessly integrated, ensuring smooth file management workflows.

Content reading and writing operations are optimized for speed and reliability, facilitating seamless data manipulation. Our solution prioritizes user experience, leveraging intuitive interfaces and intelligent algorithms to streamline file management tasks.

By combining efficiency, flexibility, and usability, our file system management solution sets a new standard for productivity, enabling users to effortlessly organize, access, and manipulate their files with unparalleled efficiency and ease.

SYSTEM ARCHITECTURE

Hardware Components:

1. **Processor:** The CPU executes instructions, including those related to filesystem operations.
2. **Memory:** RAM is used for storing data structures, program code, and buffers during filesystem operations.
3. **Storage Devices:** Hard disk drives (HDDs) or solid-state drives (SSDs) store file data and metadata, including inodes and directory structures.
4. **Input/Output Devices:** These devices facilitate user interactions with the filesystem, such as keyboards, mice, and displays.
5. Software Components:
6. **Operating System:** Manages hardware resources and provides file system services such as file I/O, process management, and memory allocation.
7. **Filesystem Implementation:** The provided code represents the filesystem implementation, including data structures (superblocks, directories, inodes) and operations (file creation, deletion, copying, etc.).
8. **Applications:** User-level programs interact with the filesystem through system calls or library functions provided by the operating system.
9. **Drivers:** Device drivers facilitate communication between the operating system and hardware components such as storage devices and input/output devices.

Structure:

1. **Arrangement and Organization:**
   The filesystem structure consists of a superblock containing metadata about the filesystem and an array of directories. Each directory contains metadata about files within it and an array of pointers to inode structures representing individual files. Inodes store metadata about files, including file names, content, size, and type (regular file or directory). The arrangement allows for hierarchical organization of files and directories, with directories containing references to files and subdirectories.
2. **Data Flow:** Data flows between components during filesystem operations, such as file creation, deletion, reading, and writing. For example, during file creation, data flows from user input to the filesystem interface, which updates directory structures and allocates memory for inodes to store file metadata and content.

Functionality:

System Objectives:

- The file system provides functionality for creating, deleting, reading, writing, copying, moving, and renaming files and directories.
- It also supports operations for listing files in directories, changing directories, finding files, and printing the current directory path.

Component Interaction:

- The functionality is achieved through interactions between components such as the operating system, filesystem implementation, and user-level applications.
- For example, user commands trigger filesystem operations, which are executed by the filesystem implementation using system calls provided by the operating system.

Interfaces:

Hardware Interfaces:

- Electrical connections and protocols facilitate communication between hardware components, such as the CPU, memory, storage devices, and input/output devices.

Software Interfaces:

- APIs (Application Programming Interfaces) and system calls provide interfaces between user-level applications and the filesystem implementation.
- Communication protocols and data structures define interfaces between file system components, allowing them to interact with each other.

Performance:

- **Speed:** Performance considerations include the speed of filesystem operations such as file access, creation, deletion, and data transfer.
- **Scalability**: The filesystem should scale efficiently with increasing numbers of files, directories, and users.
- **Reliability**: Measures such as data integrity, fault tolerance, and error handling ensure reliable operation even in adverse conditions.
- **Availability**: The filesystem should be available for use when needed, with minimal downtime or service interruptions.
- **Optimization**: Performance optimizations may include caching, buffering, concurrency control, and efficient data structures and algorithms.

Constraints:

- **Budget**: Development and deployment costs may influence architectural decisions, such as choice of hardware components and software technologies.
- **Time**: Time constraints may impact the development schedule and the extent of features implemented in the filesystem.
- **Resources**: Availability of resources such as memory, storage capacity, and processing power may impose limitations on system design and functionality.
- **Compatibility Requirements**: Compatibility with existing hardware, software, and standards may dictate design choices and interface specifications.
- **Technological Limitations**: Constraints imposed by technological limitations, such as hardware capabilities or software dependencies, may influence system architecture and implementation decisions

METHODOLOGY

The mini file system is built upon a well-structured design that encompasses various components to ensure efficient file storage and management. One of the foundational elements is the initialization of the file system superblock, which serves as a central repository for crucial metadata such as block size, number of blocks, and inode information. This superblock initialization is essential for providing the necessary context for subsequent file system operations.

Inode management is another critical aspect of the mini file system, responsible for efficiently storing and managing file metadata. The inode table plays a central role in this process, facilitating inode creation, deletion, and updates as files are created, modified, or removed. By efficiently managing inodes, the file system can effectively track file attributes and facilitate quick access to file data.

Data block allocation is essential for optimizing storage utilization within the file system. Through dynamic allocation and deallocation of data blocks as needed, the mini file system ensures that storage space is efficiently utilized, minimizing wastage and maximizing available storage capacity.

The file system supports a wide range of operations, including file and directory manipulation, which are made possible through various internal functions dedicated to disk I/O, block allocation, and general file system operations. These functions work together seamlessly to provide users with a smooth and intuitive experience when interacting with files and directories.

Testing the mini file system involved a comprehensive approach to ensure its functionality, reliability, and performance. Initially, unit tests were conducted to verify the correctness of individual functions and their interactions within the filesystem implementation. These tests covered scenarios such as file creation, deletion, reading, writing, directory manipulation, and error handling.Various test cases were designed to assess the filesystem's behavior under normal conditions as well as edge cases.

In summary, the minis file system implementation is characterized by its comprehensive design, efficient data management, thorough testing methodologies, and seamless integration with standard tools and APIs. By leveraging these elements, the minis file system offers users a robust and reliable platform for file storage and management, ensuring optimal performance and scalability across diverse environments.

SYSTEM CALLS USED

Certainly! Let's delve deeper into the system calls used in the provided mini filesystem implementation and their roles in facilitating filesystem operations:

1. **Memory Allocation System Calls**:
   - `malloc()`: This system call is part of the C standard library and is used to dynamically allocate memory during runtime. In the mini filesystem, `malloc()` is utilized to allocate memory for `struct Inode` instances when creating files. Each file in the filesystem is represented by an inode, and dynamic memory allocation allows for flexibility in managing file metadata and content.
   - `free()`: Similarly, the `free()` system call is used to deallocate memory that was previously allocated using `malloc()`. When files are deleted or directories are removed, the associated memory for inodes and other data structures needs to be released to prevent memory leaks. `free()` ensures that memory is returned to the system's memory pool for reuse.

2. **Input/Output System Calls**:
   - `printf()`, `scanf()`, and `puts()`: These system calls are fundamental for interacting with users through the command-line interface. `printf()` is used to display formatted output, providing information about filesystem operations, directory contents, and file contents to the user. `scanf()` enables the program to accept formatted input from the user, allowing them to enter commands and input data. `puts()` is similar to `printf()` but simpler, used for outputting strings to the standard output. Together, these system calls enable user interaction and feedback, making the filesystem accessible and usable.

3. **Filesystem Management System Calls**:
   - `open()`, `read()`, `write()`, and `close()`: Although not directly used in the provided code, these system calls are fundamental for interacting with the filesystem of the underlying operating system. In a full-fledged filesystem implementation, these calls would be utilized for reading from and writing to files stored on disk. For instance, `open()` would be used to open files, `read()` for reading data from files, `write()` for writing data to files, and `close()` for closing files after operations are completed. While the provided code simulates filesystem operations using data structures and algorithms within the program's memory, these system calls would be essential for actual filesystem interactions in a real-world scenario.

4. **Sleep System Call**:
   - `sleep()`: This system call is used to suspend the execution of the program for a specified number of seconds. In the provided code, `sleep()` is utilized within the main loop of the program to introduce a delay, pausing the execution and waiting for user input. This periodic waiting allows the program to remain responsive while waiting for user commands, enhancing user experience by preventing excessive CPU utilization and unnecessary resource consumption.

These system calls collectively form the backbone of the mini filesystem implementation, enabling memory management, user interaction, and simulated filesystem operations within the program. While the provided code focuses on in-memory filesystem operations, understanding these system calls provides insights into how file system interactions would be handled in a real-world scenario.

OUTPUT & RESULT

Implementing the provided mini filesystem involves translating the conceptual design outlined in the code into functional C code. The implementation follows a structured approach, beginning with defining necessary data structures and functions to represent filesystem components and operations. The `struct Superblock`, `struct Directory`, and `struct Inode` data structures encapsulate metadata about the filesystem, directories, and files, respectively. Key filesystem operations such as file creation, deletion, reading, writing, directory manipulation, and file copying are implemented as functions within the code. These functions interact with the filesystem data structures, manage memory allocation and deallocation, and ensure proper error handling. Throughout the implementation process, considerations are made to address performance, scalability, and reliability requirements. For example, dynamic memory allocation is used to manage memory resources efficiently, and efficient data structures and algorithms are employed to optimize file system operations. Additionally, the implementation incorporates error checking and validation mechanisms to handle edge cases and ensure robustness. Testing is integral to the implementation process, with unit tests, integration tests, and stress tests performed to validate functionality, reliability, and performance. Iterative development and refinement are carried out based on testing results and feedback, leading to a finalized implementation of the mini filesystem.

Result:

*****Command List***
1. ls [Dir]... list files under directories.
2. cd [Dir] change to [Dir].
3. cp [file]...[Dir] copy files to [Dir].
4. rm [file]... remove files.
5. rmdir [Dir]... remove directories.
6. touch [file]... create files.
7. mkdir [Dir]... create directories.
8. mv [file] [filename] rename the file.
9. mvdir [Dir] [directoryname] rename the directory.
10. cat [file]... read files.
11. pwd print the current directory path.
12. echo [content] > [file] write content to the file.
13. find [Dir] [file/Dir] find specific file or directory.

0. Exit

Enter your option: 7
Enter directory name: test_dir
Directory 'test_dir' created.

Enter your option: 6
Enter directory name: test_dir
Enter file name: test_file.txt
File 'test_file.txt' created in directory 'test_dir'.

Enter your option: 1
Enter directory name: test_dir
Files in directory 'test_dir':
- test_file.txt

Enter your option: 10
Enter directory name: test_dir
Enter file name: test_file.txt
Content of file 'test_file.txt' in directory 'test_dir':
(empty)

Enter your option: 12
Enter directory name: test_dir
Enter file name: test_file.txt
Enter content: This is a test file.
Content written to file 'test_file.txt' in directory 'test_dir'.

Enter your option: 10
Enter directory name: test_dir
Enter file name: test_file.txt
Content of file 'test_file.txt' in directory 'test_dir':
This is a test file.

Enter your option: 0
Exiting...

CONCLUSION AND FUTURE WORK

In conclusion, the file system project has achieved significant milestones in implementing crucial components like the file system API, superblock, inode management, and efficient data block management. These components collectively facilitate seamless interaction with files and directories, offering functionalities such as listing, changing, copying, removing, creating, renaming, reading, writing, printing the current directory path, and locating specific files or directories.

Throughout the development process, careful attention was paid to ensuring data integrity, efficient storage allocation, and optimal performance. By adhering to best practices in algorithm design and data structure implementation, the file system demonstrates robustness and reliability in managing diverse storage devices.

Looking ahead, there are several avenues for further enhancement and refinement. Strengthening security measures to control access and safeguard data, optimizing performance to handle larger workloads with minimal latency, and implementing fault tolerance mechanisms to mitigate risks of data loss or corruption are critical priorities.

Additionally, exploring advanced features such as snapshots, encryption, and support for distributed storage environments can expand the capabilities and versatility of the file system, catering to evolving user needs and technological advancements.

By addressing these areas for improvement and remaining responsive to emerging trends and user feedback, the file system project is poised to evolve into a resilient, feature-rich solution capable of meeting the demands of modern computing environments while delivering a seamless and intuitive user experience.

LITERATURE REVIEW

[1]File System - A Component of Operating System

Brijender Kahanwal, Tejinder Pal Singh2,
Ruchira Bhargava, Girish Pal Singh

Department of Computer Science and Information Technology, Maharaja Ganga Singh University, Bikaner, Rajasthan

The file system provides the mechanism for online storage and access to file contents, including data and programs. This paper covers the high-level details of file systems, as well as related topics such as the disk cache, the file system interface to the kernel, and the user-level APIs that use the features of the file system. It will give you a thorough understanding of how a file system works in general. The main component of the operating system is the file system. It is used to create, manipulate, store, and retrieve data. At the highest level, a file system is a way to manage information on a secondary storage medium. There are so many layers under and above the file system. All the layers are to be fully described here. This paper will give the explanatory knowledge of the file system designers and the researchers in the area. The complete path from the user process to secondary storage device is to be mentioned. File system is the area where the researchers are doing lot of job and there is always a need to do more work.

[2]Incremental File System Development

Erez zadok, Rakesh iyer, Nikolai joukov, Gopalan sivathanu, and Charles p. wright

Developing file systems from scratch is difficult and error prone. Layered, or stackable, file systems are a powerful technique to incrementally extend the functionality of existing file systems on commodity OSes at runtime. In this paper, we analyze the evolution of layering from historical models to what is found in four different present day commodity OSes: Solaris, FreeBSD, Linux, and Microsoft Windows. We classify layered file systems into five types based on their functionality and identify the requirements that each class imposes on the OS. We then present five major design issues that we encountered during our experience of developing over twenty layered file systems on four OSes. We discuss how we have addressed each of these issues on current OSes, and present insights into useful OS and VFS features that would provide future developers more versatile solutions for incremental file system development.

[3]A File System to Trace Them All
San Francisco, CA, 129–143 ANDERSON, D., C HA SE , J., AND VADHAT, A. 2000

File system traces have been used for years to analyze user behavior and system software behavior, leading to advances in file system and storage technologies. Existing traces, however, are difficult to use because they were captured for a specific use and cannot be changed, they often miss vital information for others to use, they become stale as time goes by, and they cannot be easily distributed due to user privacy concerns. Other forms of traces (block level, NFS level, or system-call level) all contain one or more deficiencies, limiting their usefulness to a wider range of studies.Tracefs is a groundbreaking innovation, presenting a versatile and portable solution for capturing file system traces with unprecedented ease. Its non-intrusive design allows seamless integration with any file system without requiring modifications. Offering a spectrum of trace granularity options, from users and processes to file operations and more, Tracefs empowers users to tailor their tracing experience to specific needs. Furthermore, its adaptability extends to data transformation capabilities, enabling the conversion of trace data into various formats such as aggregate counters, compressed, checksummed, encrypted, or anonymized streams. With flexible buffering and destination routing features, Tracefs ensures efficient data handling, whether directing information to sockets, disks, or other destinations. The modular and extensible architecture of Tracefs unlocks diverse applications, from debugging file systems to facilitating user activity data analysis for Intrusion Detection Systems. Evaluation of the prototype on Linux confirms Tracefs as a high-performance solution with negligible overhead, promising a new era of file system tracing and analysis.

REFERENCES

[1.] E. Zadok, R. Iyer, N. Joukov, G. Sivathanu, and C. P. Wright (2006), "On Incremental File System Development", ACM Transaction on Storage, Vol. 2, No. 2, pp. 1-33.
[2.] M. A. Halcrow (2004), "Demands, Solutions, and Improvements for Linux Filesystem Security", in the proceedings of the Linux symposium, Ottawa, Canada, pp. 269-286.
[3.] E. Zadok et. al (1999), "Extending File Systems Using Stackable Templates", in proceedings of the annual USENIX Technical Conference, USENIX Association, Monterey, CA, pp. 57-70.
[4.] Brian Carrier (2005), "File System Forensic Analysis", Addison Wesley Professional.
[5.] Steve D. Pate (2003), "UNIX Filesystems: Evolution, Design, and Implementation", Wiley Publishing, Inc.,
[6.] Dominic Giampaolo (1999), "Practical File System Design with the Be System", MORGAN KAUFMANN PUBLISHERS, INC. San Francisco, California, ISBN
[7.] Silberschatz, P. B. Galvin, and G. Gagne (2002), "Operating System Concepts", 6th Ed. John Wiley & Sons, Inc.,