CORE:
Custom OS-level Resource Emulator

Richmonde Paulo G. Gatchalian, Lauren Frances M. Salazar, Ralph Christian B. Tejada

Department of Electronics and Computer Engineering
Gokongwei College of Engineering
De La Salle University Manila
2401 Taft Avenue, Malate, Manila, Philippines 1004
richmonde_gatchalian@dlsu.edu.ph, lauren_salazar@dlsu.edu.ph, ralph_tejada@dlsu.edu.ph

Abstract*—File systems are fundamental to modern computing, governing how data is stored, retrieved, and managed. This paper presents the design and implementation of a custom file system built using C and shell scripting, operating on a virtual disk. The system simulates core file operations such as creation, deletion, viewing, editing, and listing of files while using contiguous allocation for storage management. Implemented with a block-based virtual disk image, the system demonstrates how real-world file systems like FAT and ext4 handle file allocation and space utilization. It features a command-line interface and a basic file entry structure, offering a simple yet functional environment for understanding underlying OS concepts. By working directly with binary data and simulating block-level access, this project offers valuable insights into virtual file storage and allocation mechanisms. The simplicity of the design makes it an educational tool for students studying operating systems or systems programming.

Keywords — File System, Virtual Disk, File Allocation, Contiguous Allocation, C Programming, Operating Systems

I. INTRODUCTION

File systems are fundamental components of modern operating systems, enabling structured storage, efficient access, and reliable management of data on both physical and virtual devices. As digital systems evolve, the need for lightweight, secure, and transparent file storage frameworks becomes more apparent, particularly in environments where abstraction and scalability often obscure the core mechanisms of data handling. This paper presents CORE: Custom OS-level Resource Emulator, a simplified file system developed in C and shell scripting that emulates key OS-level storage behaviors on a 1MB virtual disk image. CORE implements basic file operations such as creation, deletion, listing, and editing, using a contiguous allocation scheme to simulate low-level storage interactions. Contemporary work in embedded and cloud systems reflects similar goals. MIFS, a flash file system for constrained devices, introduces B+ tree indexing over continuous memory to improve performance and reduce memory use [1]. BlockPres applies blockchain-integrated IPFS storage for medical data, demonstrating secure, decentralized file control [2]. In large-scale systems, Chevron's data platform shows the importance of scalable and structured data delivery for decision-making across industrial infrastructures [3]. Research on vPIM explores performance optimization in virtualized memory environments, offering insights into low-overhead system design [4]. Meanwhile, SigmaOS unifies microservice and serverless workloads through a minimal, efficient OS structure, emphasizing the growing demand for flexible, OS-level abstractions [5]. CORE aims to bridge these ideas with an educational and modular design that exposes the inner workings of file allocation, metadata tracking, and block-based storage through direct interaction, offering a transparent learning tool for operating system fundamentals.

II. BACKGROUND OF THE STUDY

File systems are essential components of operating systems, responsible for managing how data is stored, accessed, and organized within storage media. They provide the structure for handling both file content and metadata, ensuring that users and applications can interact with stored information reliably. Widely used systems such as FAT, NTFS, ext4, and HFS each follow different design philosophies, particularly in how they allocate space and track file states. For example, FAT uses simple linked allocation, while NTFS incorporates a Master File Table (MFT) and journaling to enhance recoverability and metadata tracking [6]. Understanding these internal mechanisms is critical not only for systems programming but also for fields such as digital forensics, where the ability to analyze deleted, modified, or fragmented files depends on knowledge of file system architecture [7]

.

Recent research has demonstrated how forensic analysis tools reconstruct event timelines by parsing NTFS metadata. These tools extract values such as event identifiers and user session information to establish behavioral traces in a system.

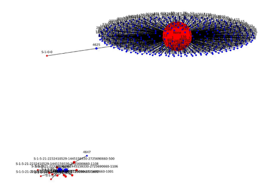


Fig. 1. Graph showing relationships between NTFS event_id and user_sid attributes, illustrating how metadata links user actions to system events [8]

Additionally, timestamp metadata, such as Modified, Accessed, Changed, and Birth (MACB) values are crucial for identifying patterns in file activity and anomaly detection.

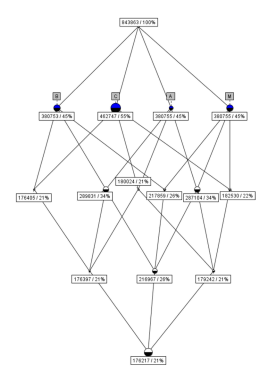


Fig. 1. Concept lattice for MACB timestamps in NTFS data, illustrating frequency and combinations of file system metadata events [8].

Complementary studies also show the value of simplified and configurable operating system environments for education and experimentation. A user-friendly kernel that includes essential features such as memory handling, syscalls, and file systems can help students better understand the role and behavior of core OS components [9]. CORE: Custom OS-level Resource Emulator builds on these ideas, offering a hands-on platform that simulates file allocation, metadata management, and virtual storage using a block-based approach to help users grasp the core principles of file system design.

III. STATEMENT OF THE PROBLEM

Modern advancements in storage hardware have significantly outpaced the evolution of traditional software stacks, creating inefficiencies that limit system performance and transparency. Although solid-state drives (SSDs) and protocols like NVMe and SAS offer high-speed data access, legacy storage architectures continue to introduce delays and obscure file system behavior [10]. This disconnect makes it difficult for learners and developers to fully understand how storage systems function at a low level.

At the same time, growing concerns over data reliability and loss, particularly in distributed and cloud environments, highlight the risks of relying on abstracted storage layers. Systems that implement encryption and multi-file redundancy demonstrate the importance of having visibility into file allocation and storage processes [11]. CORE: Custom OS-level Resource Emulator was developed in response to these challenges. It provides a simplified, transparent platform that allows users to interact directly with file operations and block-level allocation through a virtual disk.

IV. SIGNIFICANCE OF THE STUDY

Understanding file systems at the block level is a foundational skill in systems programming, cybersecurity, and computer engineering. Despite their importance, modern operating systems often obscure the inner workings of file allocation, metadata tracking, and low-level storage interaction. As a result, students and entry-level developers are frequently introduced to file systems only through high-level APIs, with limited exposure to how data is actually structured and managed. CORE: Custom OS-level Resource Emulator bridges this gap by offering a lightweight, emulator-based environment where users can directly interact with virtual storage, simulating key file operations such as creation, deletion, and content editing using contiguous block allocation.

Projects like forkSim, which visualizes process-level concurrency in operating systems education, illustrate the growing value of simulation-based tools for teaching technical concepts [12]. Similarly, CORE provides a structured, interactive platform for learning file system fundamentals. Its role is comparable to DeimOS, an educational operating system developed for robotics platforms, which demonstrates the importance of making OS-level tools accessible for students and researchers [13]. In parallel, the rising interest in decentralized file storage architectures such as IPFS and Web3 underscores the continued need for foundational knowledge in traditional storage systems [14]. CORE supports this learning by simulating the behaviors and constraints that define real-world file systems, encouraging a deeper understanding of how operating systems manage data behind the scenes.

V. OBJECTIVES

To develop an intuitive and interactive emulator, CORE, which simulates a basic operating system's file allocation process, providing an educational tool for students to visualize and understand file system behavior and resource management. The following are the key objectives:

1. Design and implement a virtual disk image within the CORE emulator to simulate file storage and allocation using contiguous block allocation.
2. Develop user-friendly interfaces that allow users to interact with the emulator by creating, viewing, deleting, and editing files, offering real-time feedback on file system operations.
3. Simulate core file system behaviors, such as allocation, deletion, and reallocation, to help students visualize the internal workings of file systems and resource management.
4. Provide clear documentation and guides within the emulator to assist students in understanding the impact of different file operations on storage, file fragmentation, and space utilization.
5. Test the emulator with students and assess its effectiveness in improving their comprehension of OS-level storage management, resource allocation, and file system structures.

VI. DESCRIPTION OF THE SYSTEM

CORE: Custom OS-level Resource Emulator is a simplified file system emulator designed to simulate how operating systems manage storage at the block level. Developed in C and supported by a shell script interface, CORE operates on a 1MB virtual disk (virtual_disk.img) and allows users to perform basic file operations such as creation, deletion, viewing, editing, and listing. The system uses a FileEntry structure to manage file metadata and a block_map to track allocation across 4096 fixed-size blocks. All interactions with the virtual disk are handled through standard C file I/O functions, providing a clear and accessible view into how real file systems perform low-level data manipulation.

CORE uses contiguous allocation, where each file is stored in sequential blocks. This method is intentionally chosen for its simplicity and ease of visualization, allowing learners to better grasp concepts like block tracking, fragmentation, and allocation strategy. The architecture includes a shell-based interface (file_system.sh) for initializing the virtual disk and compiling the system, while the main logic resides in file_system.c. By stripping down modern file system complexity into a modular, transparent emulator, CORE serves as a hands-on educational tool that bridges the gap between theoretical OS concepts and real-world file system behavior.

VII. METHODOLOGY

Fig. 3. System Flowchart

The development of CORE followed a structured, modular approach to emulate a working file system within a constrained virtual environment. The emulator was built using C programming for direct file manipulation and shell scripting to handle user interaction, compilation, and disk setup. A virtual disk image (virtual_disk.img) was created to simulate a block-based storage system, with each block set to 256 bytes. The file system tracks allocation using a block map and manages metadata through a file entry structure stored in memory.

Users interact with the system through a simple terminal menu presented via a shell script. Upon selection, the appropriate C function is invoked to execute file operations such as creation, deletion, viewing, editing, or listing. For file creation, the system checks for duplicate filenames, calculates the required number of blocks, searches for free contiguous space, writes the content into the virtual disk, and updates the metadata. Similar structured logic applies for the other operations, with file data being accessed or modified directly through standard I/O functions like fseek() and fwrite(). All changes are immediately reflected in the virtual disk to simulate real-time persistence.

VIII. REVIEW OF RELATED LITERATURE

The design and behavior of file systems have long been central topics in systems programming, yet their internal mechanisms often remain opaque to learners due to the abstraction provided by modern operating systems. This gap between theoretical understanding and practical application has driven researchers to develop solutions that improve transparency, performance, and accessibility. Several works have focused on enhancing storage performance through smarter allocation strategies and memory-aware designs. Yu et al. introduced VMAlloc, a wear-leveling-aware multi-grained allocator for persistent memory file systems. Their system significantly reduced uneven write distribution, which is a major cause of early memory degradation in PM systems [15]. In a similar direction, Liu and Wang proposed CFFS, a file system built around contiguous file allocation and fine-grained metadata to improve mmap performance and reduce fragmentation in high-speed memory environments [16]. These contributions highlight the value of optimizing memory allocation and reuse in block-based storage, a concept CORE seeks to simulate within an educational environment.

Beyond allocation techniques, researchers have explored fault-tolerant and power-safe systems for embedded and resource-constrained platforms. Munegowda et al. examined transaction-safe cluster allocation in the TexFAT system, designed for embedded Windows CE systems. Their study emphasizes the importance of file system behavior under restricted power conditions and limited resources [17]. Devulapalli et al. proposed integrating object-based storage devices into parallel file systems to improve efficiency and decentralize metadata management, which can reduce performance bottlenecks in distributed systems [18]. Peng et al. focused on addressing write amplification in SSDs by introducing PULSE, a log-structured enhancement for B-epsilon-tree storage models that reduces wear and improves throughput [19]. These studies collectively show that understanding allocation strategies and their real-world implications is crucial for system designers, and CORE can provide a foundation for such knowledge by simulating similar behaviors through basic contiguous allocation logic.

Recent advancements have also shown growing interest in decentralized storage systems. Blockchain-based frameworks, in particular, offer improved security, transparency, and control over data. Mishra et al. proposed a prototype for secure medical record management using blockchain and IPFS, providing a decentralized platform with smart contract support for access control and data integrity [20]. Khan et al. developed BlockPres, a system that uses IPFS to securely store patient prescriptions while ensuring access control for multiple healthcare providers [2]. Yang et al. contributed a ring signature scheme that allows for both anonymity and traceability, linking medical data across different providers while maintaining security and auditability [21]. Anand et al. combined blockchain and transfer learning to create TraVel, a framework for filtering and securing IoT data streams before storage on a private Ethereum network [3]. To improve scalability in such systems, Mythili and Gopalakrishnan designed an adaptive sharded blockchain protocol for healthcare data, significantly increasing throughput while maintaining low latency [22]. Although these systems are far more complex than CORE, they emphasize the long-term importance of understanding file and data storage structures, even at a fundamental level.

A few researchers have addressed the need for educational or simplified platforms. Poinot et al. presented a simulation framework for data management in distributed systems that uses a data-oriented model to optimize workflow and reduce redundant storage [#Poinot]. On a more hybridized approach, Nguyen and Leis revisited the longstanding debate on storing files versus BLOBs in databases. They proposed a system that uses a lightweight FUSE layer to expose database-stored BLOBs as files, balancing transactional integrity with usability [23]. These systems, although aimed at enterprise or research environments, align with the goals of CORE by emphasizing visibility, direct access, and architectural clarity. CORE positions itself uniquely as a teaching tool by offering a simplified, hands-on experience with file creation, deletion, viewing, and metadata tracking. By making these processes transparent and accessible, the system provides learners with a direct understanding of how operating systems manage data at the most fundamental level.

IX. THEORETICAL CONSIDERATIONS

The design of CORE draws from established theories in memory allocation, system optimization, and resource modeling. At its core, CORE implements a contiguous block allocation strategy, which reflects principles outlined by Duong and Hur in their study of translation lookaside buffer (TLB) prefetching. Their work demonstrated how exploiting memory contiguity at the block level improves I/O efficiency and reduces latency caused by page-table walks, a concept that CORE simplifies to illustrate allocation behavior in a virtual environment [24]. In modeling resource interactions, CORE also aligns with the framework proposed by Samy et al., who used deep reinforcement learning to optimize task offloading and resource consumption in mobile edge computing. While CORE does not employ machine learning algorithms, it simulates static, deterministic resource allocation that mirrors foundational ideas in system-level decision-making under constrained conditions [25]. Additionally, the simulation's structure draws inspiration from the differential game-based optimization model discussed by Lv, where interactions between institutions and technologies were analyzed for optimal outcomes. CORE reflects a similar spirit by allowing users to interactively explore how changes in file size, placement, and deletion affect virtual disk space, allocation efficiency, and resource reuse [26]. These theoretical models collectively reinforce CORE's role as a minimalistic but pedagogically grounded emulator designed to demystify low-level file system operations through hands-on experimentation.

X. DATA AND RESULTS

The group has developed the file system simulator in a virtual box using the Fedora Linux environment. The implementation of CORE successfully demonstrates key file system operations on a simplified virtual disk environment. The emulator's behavior aligns with expected outcomes based on the underlying principles of file system design using contiguous allocation.

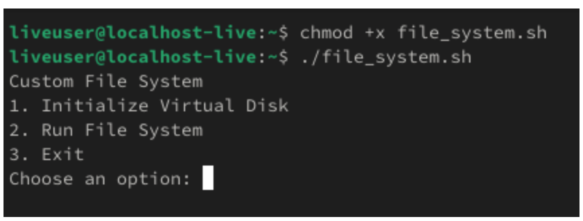


Fig. 4. Setting up and Shell Script Menu

First, chmod +x is used to make the shell script executable. The script is then run using ./, which tells the system to execute the file from the current directory. Once executed, the script displays a menu that allows the user to either initialize a new virtual disk or launch the main file system. If a virtual disk already exists, the script will notify the user that it is ready for use. By default, the virtual disk is allocated 1 MB of storage for the emulator's file operations, though this size can be adjusted as needed.

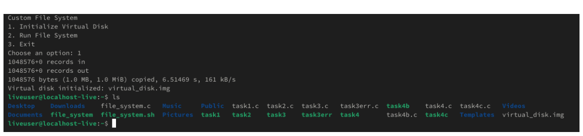


Fig. 5. Initializing virtual disk and checking

Once the virtual disk is initialized, the user is notified and provided with the disk's details. As shown in Figure 5 above, the ls command is used to verify that the virtual disk has been successfully created in the current directory.

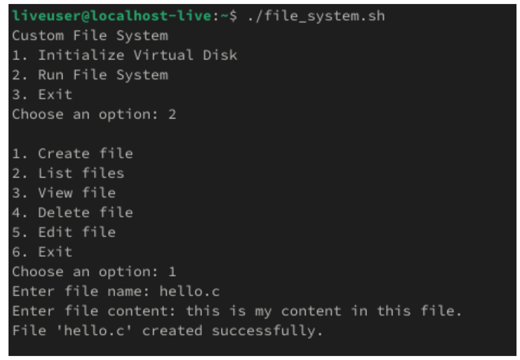


Fig. 6. Create File

Users can create a file by selecting the option, they will be asked for the file name then write the content for the file. The user will be notified once the file was successfully created or not.

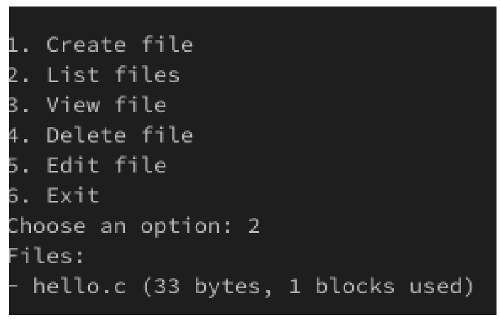


Fig. 7 List File

The List Files option in the menu allows users to view all existing files in the virtual disk. This includes checking if a specific file has been created. The listing also displays each file's size based on its content, along with its corresponding block size.

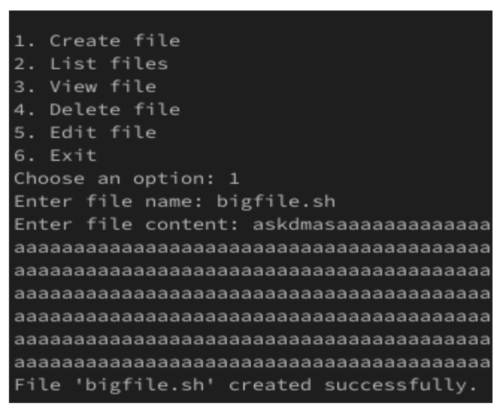
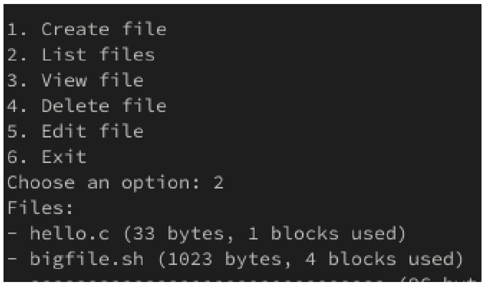


Fig. 8. Creating a File with larger content

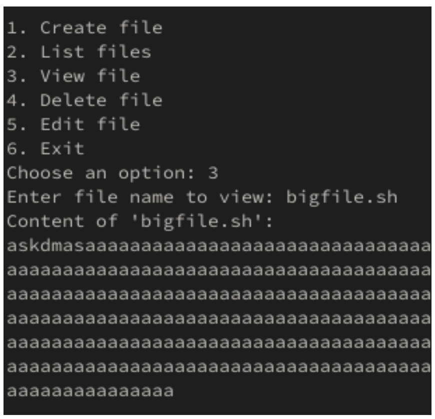
As shown in figure 8, the user tried to create a file with more content compared to figure 7.



```
1. Create file
2. List files
3. View file
4. Delete file
5. Edit file
6. Exit
Choose an option: 2
Files:
- hello.c (33 bytes, 1 blocks used)
- bigfile.sh (1023 bytes, 4 blocks used)
```

Fig.9. Comparison of Files

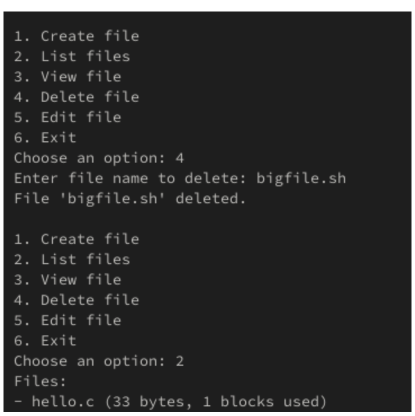
Since the content of "bigfile.sh" is larger than "hello.c," the details in the list view are updated accordingly. It shows that "bigfile.sh" contains 1 KB of data and uses 4 blocks of storage.



```
1. Create file
2. List files
3. View file
4. Delete file
5. Edit file
6. Exit
Choose an option: 3
Enter file name to view: bigfile.sh
Content of 'bigfile.sh':
askdmasaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaa
```

Fig. 10 View File

The View File feature allows users to view the contents of a file without the ability to edit its current content. This feature is particularly useful for accessing and tracking files, especially when multiple files are created, providing users with a read-only preview for easy reference.

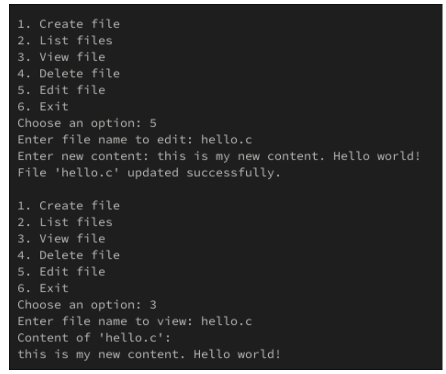


```
1. Create file
2. List files
3. View file
4. Delete file
5. Edit file
6. Exit
Choose an option: 4
Enter file name to delete: bigfile.sh
File 'bigfile.sh' deleted.

1. Create file
2. List files
3. View file
4. Delete file
5. Edit file
6. Exit
Choose an option: 2
Files:
- hello.c (33 bytes, 1 blocks used)
```

Fig. 11. Delete File

The Delete File feature allows users to remove a file by entering its name. After attempting to delete the file, the user will be notified whether the deletion was successful. To verify if the file has been deleted, the user can select option 2 to view the list of files. If the file was successfully deleted, it will no longer appear in the directory.



```
1. Create file
2. List files
3. View file
4. Delete file
5. Edit file
6. Exit
Choose an option: 5
Enter file name to edit: hello.c
Enter new content: this is my new content. Hello world!
File 'hello.c' updated successfully.

1. Create file
2. List files
3. View file
4. Delete file
5. Edit file
6. Exit
Choose an option: 3
Enter file name to view: hello.c
Content of 'hello.c':
this is my new content. Hello world!
```

Fig. 12. Edit File

The Edit File feature allows users to modify the content of an existing file if there are any mistakes. This is more efficient than creating a new file to replace the incorrect content, as it helps conserve space on the virtual disk.

A. Code Used:

file_system.sh:

```
#!/bin/bash

VIRTUAL_DISK="virtual_disk.img"
DISK_SIZE=1048576  # 1MB virtual disk

# Initialize Virtual Disk
init_disk() {
   if [ ! -f $VIRTUAL_DISK ]; then
      dd if=/dev/zero of=$VIRTUAL_DISK bs=1
count=$DISK_SIZE
      echo "Virtual disk initialized: $VIRTUAL_DISK"
   else
      echo "Virtual disk already exists."
   fi
}

# Run File System Operations
run_fs() {
   gcc file_system.c -o file_system
   ./file_system
}

# Main Menu
echo "Custom File System"
echo "1. Initialize Virtual Disk"
echo "2. Run File System"
echo "3. Exit"
read -p "Choose an option: " option

case $option in
   1) init_disk ;;
   2) run_fs ;;
   3) exit ;;
   *) echo "Invalid option";;
esac
```

file_system.c:

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>
#include <stdbool.h>

#define BLOCK_SIZE 256
#define MAX_BLOCKS 4096
#define MAX_FILES 128

// Allocation type
#define ALLOC_CONTIGUOUS 0

// File entry structure
typedef struct {
        char name[32];
        int start_block;
        int size_bytes; // actual content length
        int blocks_used;
        int allocation_type;
        bool used;
} FileEntry;

// File system structure
typedef struct {
        FileEntry files[MAX_FILES];
        bool block_map[MAX_BLOCKS];
} FileSystem;

FileSystem fs;
FILE *virtual_disk;

void init_file_system() {
        memset(&fs, 0, sizeof(fs));
        virtual_disk = fopen("virtual_disk.img", "r+b");
        if (!virtual_disk) {
        virtual_disk = fopen("virtual_disk.img", "w+b");
        fseek(virtual_disk, BLOCK_SIZE *
MAX_BLOCKS - 1, SEEK_SET);
        fputc('\0', virtual_disk);
        fflush(virtual_disk);
        }
}

void shutdown_file_system() {
        fclose(virtual_disk);
}

int find_free_blocks(int blocks_needed) {
        int count = 0;
        for (int i = 0; i <= MAX_BLOCKS -
blocks_needed; i++) {
        bool found = true;
        for (int j = 0; j < blocks_needed; j++) {
        if (fs.block_map[i + j]) {
        found = false;
        break;
        }
        }
        if (found) return i;
        }
        return -1;
}

void create_file(const char *name, const char *content) {
        for (int i = 0; i < MAX_FILES; i++) {
        if (fs.files[i].used && strcmp(fs.files[i].name,
name) == 0) {
        printf("File with this name already exists.\n");
        return;
        }
        }
```

```
        int content_length = strlen(content);
        int blocks_needed = (content_length +
BLOCK_SIZE - 1) / BLOCK_SIZE;

        int start_block =
find_free_blocks(blocks_needed);
        if (start_block == -1) {
        printf("Not enough contiguous space
available.\n");
        return;
        }

        for (int i = start_block; i < start_block +
blocks_needed; i++) {
        fs.block_map[i] = true;
        }

        for (int i = 0; i < MAX_FILES; i++) {
        if (!fs.files[i].used) {
        strcpy(fs.files[i].name, name);
        fs.files[i].start_block = start_block;
        fs.files[i].size_bytes = content_length;
        fs.files[i].blocks_used = blocks_needed;
        fs.files[i].allocation_type =
ALLOC_CONTIGUOUS;
        fs.files[i].used = true;
        break;
        }
        }

        fseek(virtual_disk, start_block * BLOCK_SIZE,
SEEK_SET);
        fwrite(content, 1, content_length, virtual_disk);
        fflush(virtual_disk);

        printf("File '%s' created successfully.\n", name);
}

void list_files() {
        printf("Files:\n");
        for (int i = 0; i < MAX_FILES; i++) {
        if (fs.files[i].used) {
        printf("- %s (%d bytes, %d blocks used)\n",
        fs.files[i].name,
        fs.files[i].size_bytes,
        fs.files[i].blocks_used);
        }
        }
}

void view_file(const char *name) {
        for (int i = 0; i < MAX_FILES; i++) {
        if (fs.files[i].used && strcmp(fs.files[i].name,
name) == 0) {
        char *buffer = malloc(fs.files[i].size_bytes + 1);
        fseek(virtual_disk, fs.files[i].start_block *
BLOCK_SIZE, SEEK_SET);
        fread(buffer, 1, fs.files[i].size_bytes,
virtual_disk);
        buffer[fs.files[i].size_bytes] = '\0';
        printf("Content of '%s':\n%s\n", name, buffer);
        free(buffer);
        return;
        }
        }
        printf("File not found.\n");
}
```

```
void delete_file(const char *name) {
        for (int i = 0; i < MAX_FILES; i++) {
        if (fs.files[i].used && strcmp(fs.files[i].name,
name) == 0) {
        for (int j = fs.files[i].start_block;
        j < fs.files[i].start_block +
fs.files[i].blocks_used; j++) {
        fs.block_map[j] = false;
        }
        fs.files[i].used = false;
        printf("File '%s' deleted.\n", name);
        return;
        }
        }
        printf("File not found.\n");
}

void edit_file(const char *name, const char *new_content) {
        for (int i = 0; i < MAX_FILES; i++) {
                if (fs.files[i].used &&
strcmp(fs.files[i].name, name) == 0) {
                        int new_length =
strlen(new_content);
                        int new_blocks =
(new_length + BLOCK_SIZE - 1) / BLOCK_SIZE;

                        // Free old blocks
                        for (int j =
fs.files[i].start_block;
                          j <
fs.files[i].start_block + fs.files[i].blocks_used; j++) {

fs.block_map[j] = false;
                        }

                        // Find new space
                        int new_start_block =
find_free_blocks(new_blocks);
                        if (new_start_block == -1)
{
                                printf("Not
enough contiguous space available for updated content.\n");

                                // Reallocate
the old blocks back if new space can't be found
                                for (int j =
fs.files[i].start_block;
                                  j <
fs.files[i].start_block + fs.files[i].blocks_used; j++) {

fs.block_map[j] = true;
                                }

                                return;
                        }

                        for (int j =
new_start_block; j < new_start_block + new_blocks; j++) {

fs.block_map[j] = true;
                        }

                        fs.files[i].start_block =
new_start_block;
                        fs.files[i].size_bytes =
new_length;
                        fs.files[i].blocks_used =
new_blocks;
                                fseek(virtual_disk,
new_start_block * BLOCK_SIZE, SEEK_SET);
                                fwrite(new_content, 1,
new_length, virtual_disk);
                                fflush(virtual_disk);

                                printf("File '%s' updated
successfully.\n", name);
                                return;
                        }
        }
        printf("File not found.\n");
}



int main() {
        init_file_system();

        int choice;
        char name[32];
        char content[1024];

        while (1) {
        printf("\n1. Create file\n2. List files\n3. View
file\n4. Delete file\n5. Edit file\n6. Exit\nChoose an option:
");
        scanf("%d", &choice);
        getchar();

        switch (choice) {
        case 1:
        printf("Enter file name: ");
        fgets(name, sizeof(name), stdin);
        name[strcspn(name, "\n")] = '\0';
        printf("Enter file content: ");
        fgets(content, sizeof(content), stdin);
        content[strcspn(content, "\n")] = '\0';
        create_file(name, content);
        break;
        case 2:
        list_files();
        break;
        case 3:
        printf("Enter file name to view: ");
        fgets(name, sizeof(name), stdin);
        name[strcspn(name, "\n")] = '\0';
        view_file(name);
        break;
        case 4:
        printf("Enter file name to delete: ");
        fgets(name, sizeof(name), stdin);
        name[strcspn(name, "\n")] = '\0';
        delete_file(name);
        break;
        case 5:
        printf("Enter file name to edit: ");
        fgets(name, sizeof(name), stdin);
        name[strcspn(name, "\n")] = '\0';
        printf("Enter new content: ");
        fgets(content, sizeof(content), stdin);
        content[strcspn(content, "\n")] = '\0';
        edit_file(name, content);
        break;
        case 6:
        shutdown_file_system();
        return 0;
        default:
```

```
            printf("Invalid option.\n");
            }
            }
}
```

X1. ANALYSIS OF THE DATA

The CORE file system emulator, which was implemented and tested in a Fedora Linux environment, successfully demonstrates file system operations through contiguous allocation. The system allows users to initialize a virtual disk, open files, read and write content, and control file deletions. The virtual disk is established in the directory during initialization, and the ls command confirms its characteristics. It has variable storage capacities that can be modified as required. Users can define files by name and content, and the system monitors file size and utilization of blocks, as observed with large files such as "bigfile.sh," which occupies more than one block for storage. The List Files function enables users to see all the files with their size and block utilization, while the View File option offers a read-only view of file content. The Delete File feature ensures that files can be removed, with confirmation of successful deletion provided to the user, and the Edit File feature allows users to modify file content without creating new files, conserving disk space. The emulator showcases effective file management and provides a simple interface for tracking and managing files in a virtual disk system.

XIII. CONCLUSION

CORE: Custom OS-level Resource Emulator provides a crucial educational tool that bridges theoretical knowledge and practical experience in operating system design. By simulating file allocation processes using contiguous block allocation, CORE allows students to visualize the internal workings of file systems. This hands-on approach supports the growing demand for interactive learning platforms that enhance understanding of complex systems, as demonstrated by Ferreira et al. with their web-based oscilloscope interface [27]. CORE helps demystify OS-level operations, offering a practical, user-friendly environment. Furthermore, inspired by Alzakari et al.'s work on using deep learning for security in IoT environments [28], there is potential for CORE to address real-world issues such as security in file systems. Future developments could incorporate features like ransomware detection, offering students both system management and security insights. CORE's ability to simulate and interact with file systems, while also addressing potential security vulnerabilities, significantly enhances its educational value. Overall, CORE not only fills a gap in systems programming education by making abstract concepts tangible but also opens opportunities for advancements in OS education. Expanding CORE's functionality could allow students to explore a wider range of OS operations and security challenges, preparing them for both academic and industry problems in computer engineering.

XII. RECOMMENDATION

To further enhance the CORE: Custom OS-level Resource Emulator, the integration of blended learning strategies, as outlined by Amarathunga et al. [29], could significantly optimize the learning process. By combining asynchronous online modules, interactive learning environments, and real-time simulations, students would have the opportunity to practice system-level simulations remotely, supporting self-regulated learning and fostering collaborative problem-solving. Additionally, the application of simulation-based learning, as demonstrated by Pakosch et al. in their "StudyTalk" project, could offer students the opportunity to manage and design virtual file systems in a sandboxed environment. This would allow learners to engage in realistic system configurations and error-handling tasks, mimicking actual industry scenarios [30]. Incorporating techniques that address motivation, self-efficacy, and collaborative learning, as suggested by Amarathunga et al., could also enhance the effectiveness of CORE, enabling students to improve their practical skills while engaging with complex systems programming challenges [29]. By embedding project-based, real-time system management simulations, CORE can move beyond theoretical OS design, offering a robust platform for experiential learning, thus preparing students for future roles in systems programming, data management, and security.

REFERENCES

[1] Jiang, J., Yang, M., Qiao, L., Wang, T., & Chen, X. (2025). MIFS: A low overhead and efficient mixture file index management method in flash file system. *Journal of Systems Architecture*, *162*. Scopus. https://doi.org/10.1016/j.sysarc.2025.103387

[2] Khan, A., Litchfield, A., Alabdulatif, A., & Khan, F. (2025). BlockPres IPFS: Performance evaluation of blockchain based secure patients prescription record storage using IPFS for smart prescription management system. Cluster Computing, 28(4). Scopus. https://doi.org/10.1007/s10586-024-05054-6

[3] Anand, P., Singh, Y., & Singh, H. (2025). Secure IoT data dissemination with blockchain and transfer learning techniques. *Scientific Reports*, *15*(1). Scopus. https://doi.org/10.1038/s41598-024-84837-8

[4] Kim, H., Heo, G., & Doh, I. (2025). Role-based federated learning exploiting IPFS for privacy enhancement in IoT environment. *Computer Networks*, *263*. Scopus. https://doi.org/10.1016/j.comnet.2025.111200

[5] Szekely, A., Belay, A., Morris, R., & Kaashoek, M. F. (2024). *Unifying serverless and microservice workloads with SigmaOS*. 385–402. Scopus. https://doi.org/10.1145/3694715.3695947

[6] Gandhi, H. S., & Deshmukh, A. D. (2024). Understanding File System Forensics: FAT, NTFS, HFS, and EXT. In Digital Forensics in the Age of Ai (pp. 177–194). Scopus. https://doi.org/10.4018/979-8-3373-0857-9.ch007

[7] Toolan, F. (2025). File system forensics (p. 464). Scopus.

[8] Marková, E., Sokol, P., Krišáková, S. P., & Kováčová, K. (2024). Dataset of Windows operating system forensics artefacts. Data in Brief, 55. Scopus. https://doi.org/10.1016/j.dib.2024.110693

[9[Deak, T., Deaconescu, R., Kroning, M., & Monti, A. (2024). High-Configurability for Operating Systems. Proceedings - RoEduNet IEEE International Conference. Scopus. https://doi.org/10.1109/RoEduNet64292.2024.10722499

[10] Hao, D., Gao, C., & Shu, J. (2025). Design of User Space Storage Architecture for SCSI Subsystem. Jisuanji Yanjiu Yu Fazhan/Computer Research and Development, 62(3), 633–647. Scopus. https://doi.org/10.7544/issn1000-1239.202440632

[11] Batra, R., Mandal, S. K., Singh, D., Parmar, Y., Bhatt, M., & Kharbas, V. K. (2025). Novel method for preventing loss of information in multi-cloud storage assistance. International Journal of System Assurance Engineering and Management. Scopus. https://doi.org/10.1007/s13198-024-02681-5

[12] Farina, D. A. R., Campiolo, R., Rufino, J., & Pereira, M. J. V. (2024). Automatic and Dynamic Visualization of Process-Based Concurrent Programs. 120. Scopus. https://doi.org/10.4230/OASIcs.SLATE.2024.6

[13] Kegeleirs, M., Garzón Ramos, D., & Birattari, M. (2025). DeimOS: A ROS-Ready operating system for the e-puck. Journal of Open Research Software, 13(1). Scopus. https://doi.org/10.5334/jors.437

[14] Mahajan, J., & Prachi, A. (2024). Decentralized File Storage: Leveraging Blockchain, Polygon, Web3, and IPFS. 2024 Parul International Conference on Engineering and Technology, PICET 2024. Scopus. https://doi.org/10.1109/PICET60765.2024.10716189

[15] Yu, Z., Zhang, R., Yang, C., Nie, S., & Liu, D. (2023). An efficient wear-leveling-aware multi-grained allocator for persistent memory file systems. Frontiers of Information Technology and Electronic Engineering, 24(5), 688–702. Scopus. https://doi.org/10.1631/FITEE.2200468

[16] Liu, J.-K., & Wang, S.-D. (2022). CFFS: A Persistent Memory File System for Contiguous File Allocation With Fine-Grained Metadata. IEEE Access, 10, 91678–91698. Scopus. https://doi.org/10.1109/ACCESS.2022.3202532

[17] Munegowda, K., Raju, G. T., Raju, V. M., & Manjunath, T. N. (2015). Characterization of transaction-safe cluster allocation strategies of texfat file system for embedded storage devices. Smart Innovation, Systems and Technologies, 31, 503–515. Scopus. https://doi.org/10.1007/978-81-322-2205-7_47

[18] Devulapalli, A., Dalessandro, D., Wyckoff, P., Ali, N., & Sadayappan, P. (2007). Integrating parallel file systems with object-based storage devices. Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, SC'07. Scopus. https://doi.org/10.1145/1362622.1362659

[19] Peng, H.-D., Chen, Y.-S., Chen, T.-Y., & Chang, Y.-H. (2025). PULSE: Progressive Utilization of Log-Structured Techniques to Ease SSD Write Amplification in B-epsilon-tree. 245–250. Scopus. https://doi.org/10.1145/3658617.3697561

[20] Mishra, D. P., Rajeev, B., Mallick, S. R., Lenka, R. K., & Salkuti, S. R. (2025). Efficient blockchain based solution for secure medical record management. International Journal of Informatics and Communication Technology, 14(1), 59–67. Scopus. https://doi.org/10.11591/ijict.v14i1.pp59-67

[21] Yang, Y., Liu, M., Wang, L., Pu, Y., Zheng, R., Wu, Q., & Guo, J. (2025). A ring signature scheme with linkability and traceability for blockchain-based medical data sharing system. Peer-to-Peer Networking and Applications, 18(3). Scopus. https://doi.org/10.1007/s12083-025-01913-0

[22] Mythili, J., & Gopalakrishnan, R. (2025). Improving data transmission through optimizing blockchain sharding in cloud IoT based healthcare applications. Egyptian Informatics Journal, 30. Scopus. https://doi.org/10.1016/j.eij.2025.100661

[23] Nguyen, L.-D., & Leis, V. (2024). Why Files if You Have a DBMS? 3878–3892. Scopus. https://doi.org/10.1109/ICDE60146.2024.00297

[24] Duong, T. D., & Hur, J. Y. (2025). Contiguity aware TLB prefetching for embedded I/O devices. IEICE Electronics Express, 22(3). Scopus. https://doi.org/10.1587/elex.21.20240664

[25] Samy, A., Elgendy, I. A., Yu, H., Zhang, W., & Zhang, H. (2022). Secure Task Offloading in Blockchain-Enabled Mobile Edge Computing With Deep Reinforcement Learning. IEEE Transactions on Network and Service Management, 19(4), 4872–4887. Scopus. https://doi.org/10.1109/TNSM.2022.3190493

[26] Lv, F. (2025). Research on optimization strategies of university ideological and political parenting models under the empowerment of digital intelligence. Scientific Reports, 15(1). Scopus. https://doi.org/10.1038/s41598-025-92985-8

[27] Ferreira, J., Rocha, A., Alves, M., & Oliveira, P. C. D. (2025). On a Web-Based Oscilloscope Interface App for E-Learning: Software Architecture, Practical Applications, and User Experience. Sci, 7(1). Scopus. https://doi.org/10.3390/sci7010019

[28] Alzakari, S. A., Aljebreen, M., Ahmad, N., Alhashmi, A. A., Alahmari, S., Alrusaini, O., Al-Sharafi, A. M., & Almukadi, W. S. (2025). An intelligent ransomware based cyberthreat detection model using multi head attention-based recurrent neural networks with optimization algorithm in IoT environment. Scientific Reports, 15(1). Scopus. https://doi.org/10.1038/s41598-025-92711-4

[29] Amarathunga, B. (2025). Blended learning trends and future directions: A systematic literature review and bibliometric analysis. International Journal of Information and Learning Technology, 42(2), 147–164. Scopus. https://doi.org/10.1108/IJILT-04-2024-0072

[30] Pakosch, A., Allerkamp, D., & Koschel, A. (2025). Simulating Real IT Project Life for Students. 2, 1569–1570. Scopus. https://doi.org/10.1145/3641555.3705249

[30]