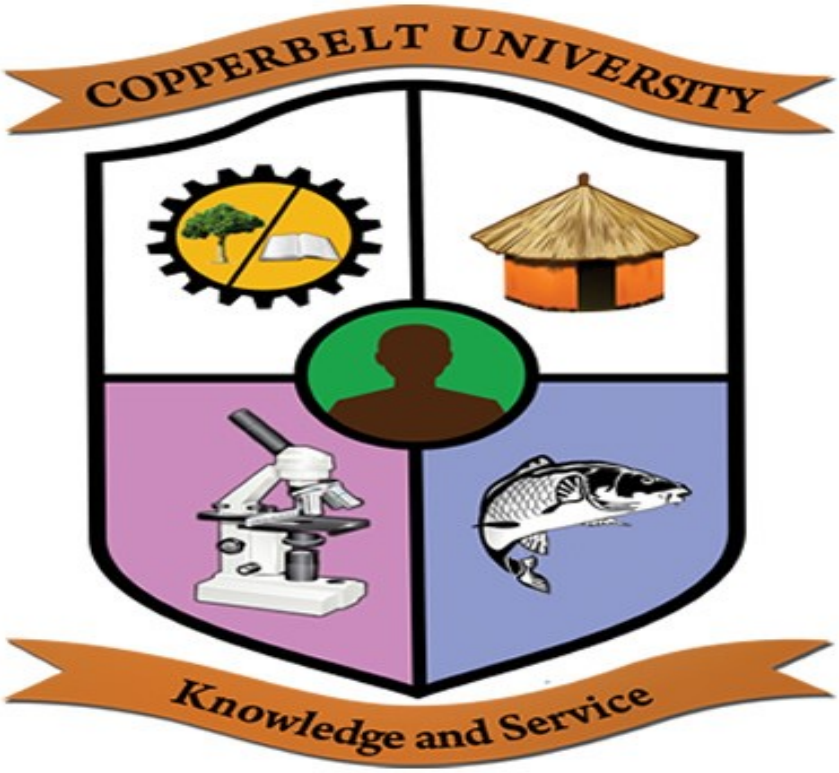


THE COPPERBELT UNIVERSITY

SCHOOL OF INFORMATION AND COMUNICATION TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

INTRODUCTION TO OPERATING SYSTEMS (CS225)

NAME: EDGAR MUMBA MWILA.

SIN: 22105649.

&

NAME: MWAMBA STEPHEN

SIN:22104039

ASSIGNMENT: ASSIGNMENT 1.

DUE DATE: 31st JULY 2024.

LECTURER: Dr Derrick Ntalasha

IMPLEMENTATION OF A FILE SYSTEM IN USER SPACE.

Abstract

A simulated filesystem project made using C programming language and the gtk library to demonstrate file system logic in the user space. Made as for the purpose of submittion as assignment 1 for introduction to operating systems (CS225) on 31st July 2024.

Table of Contents

Introduction

Background of the Project

Overview of File Systems in Operating Systems

File systems are crucial components of operating systems that manage how data is stored, retrieved, and organized on storage devices. They provide a structured way to store and access files and directories, ensuring data integrity and efficient use of storage space.

Importance of File System Implementation

Implementing a file system helps understand the complexities involved in data management, including storage allocation, directory management, and file operations. It provides insights into how modern operating systems handle data and can be a valuable educational tool.

Brief Explanation of Simulated File Systems

Simulated file systems allow for the testing and development of file system concepts without affecting the actual storage. They are often used for educational purposes and research to demonstrate how file systems work in a controlled environment.

Objectives

- To implement a simulated file system in user space using C.
- To develop a graphical user interface for interacting with the file system.
- To incorporate basic security features like password protection.

Literature Review

1. Overview of File System Concepts

File systems are fundamental components of operating systems, responsible for organizing and managing data on storage devices. They provide an abstraction layer between the physical storage and the user, allowing for efficient data retrieval and manipulation.

Key concepts in file systems include:

a) File and Directory Structure:

- Files are the basic units of data storage.
- Directories (or folders) provide a hierarchical organization of files.
- Metadata associated with files and directories (e.g., creation time, size, permissions) is crucial for management.

b) Allocation Methods:

- Contiguous Allocation: Files occupy consecutive blocks on disk. While simple and efficient for sequential access, it can lead to external fragmentation.
- Linked Allocation: Each block points to the next block of the file. This method eliminates external fragmentation but can be inefficient for random access.
- Indexed Allocation: Uses an index block to store pointers to file blocks. This method supports efficient random access and mitigates fragmentation issues.

c) Free Space Management:

- Bit vectors, linked lists, and grouping are common techniques for tracking free disk space.

d) File System Consistency:

- Mechanisms like journaling or log-structured file systems ensure data integrity in case of system failures.

2. Discussion of Existing File System Implementations

Modern operating systems employ sophisticated file systems that balance performance, reliability, and features. Some notable examples include:

a) NTFS (New Technology File System):

- Used in Windows operating systems.
- Features: Journaling, encryption, large file and volume support, and fine-grained access control.
- Uses a Master File Table (MFT) to store metadata and small files.

b) ext4 (Fourth Extended File System):

- Widely used in Linux distributions.
- Improvements over its predecessor (ext3) include larger file system and file size support, improved performance, and enhanced reliability.
- Uses extent-based file storage for improved performance and reduced fragmentation.

c) HFS+ and APFS (Apple File System):

- Used in macOS and iOS devices.
- APFS, introduced in 2017, offers features like snapshots, space sharing, and optimization for solid-state drives.

d) ZFS (Zettabyte File System):

- Originally developed by Sun Microsystems, now used in various Unix-like systems.
- Offers advanced features like pooled storage, copy-on-write, and built-in RAID-like capabilities.

3. Review of Simulated File Systems

Simulated file systems play a crucial role in education and research, offering a controlled environment to understand and experiment with file system concepts:

a) Educational Benefits:

- Allow students to interact with file system internals without the risk of damaging actual system data.
- Provide a platform to implement and test various allocation strategies and data structures.

b) Research Applications:

- Enable testing of new file system concepts and optimizations in a controlled environment.
- Facilitate performance analysis and comparison of different file system designs.

c) Examples of Simulated File Systems:

- MINIX File System Simulator: A teaching tool that simulates the MINIX file system, allowing students to explore its internals.
- FUSE (Filesystem in Userspace): While not strictly a simulator, FUSE allows for the creation of custom file systems in user space, which is valuable for both education and prototyping.

d) Limitations and Considerations:

- Simulated systems may not fully capture the complexities of real-world scenarios, such as concurrent access and hardware-specific optimizations.
- The level of abstraction in simulated systems needs to be carefully balanced to provide meaningful learning experiences while remaining manageable.

In conclusion, file systems are complex and critical components of modern computing environments. The study of file system concepts, existing implementations, and simulated systems provides valuable insights into data management, storage efficiency, and system reliability. Simulated file systems, in particular, offer a unique opportunity to explore these concepts in a controlled and educational setting, bridging the gap between theoretical understanding and practical implementation.

Methodology

Tools and Techniques

- **Programming Language:** C
- **GUI Framework:** GTK
- **Development Environment:** Visual studio code, running on a linux OS(ZORIN OS distro)
- **Version Control:** git and github.

System Design

File System Architecture

System Architecture

Three-Tier Architecture: The system follows a three-tier architecture consisting of: a) Presentation Layer (GUI): Implemented in gui.c b) Business Logic Layer: Implemented in simulatedFileSystem.c c) Data Layer: Simulated disk space in simulatedFileSystem.c. The link between these two main files is through the simulatedFileSystem.h file.

Components

1. **Graphical User Interface (GUI)**
    - Developed using GTK.
    - Provides an interactive interface for users to perform file system operations.
    - Includes buttons and dialogs for creating, deleting, moving, and renaming files and directories.
2. **Core File System Implementation (`simulatedFileSystem.c`)**
    - Contains the main logic for file system operations.
    - Manages file and directory structures, file allocation, and security features.
    - Implements functions for creating, deleting, moving, reading, and writing files and directories.
3. **Header File (`simulatedFileSystem.h`)**
    - Declares the data structures and function prototypes used in the core file system implementation.
    - Provides an interface for the GUI to interact with the core file system.

Interaction

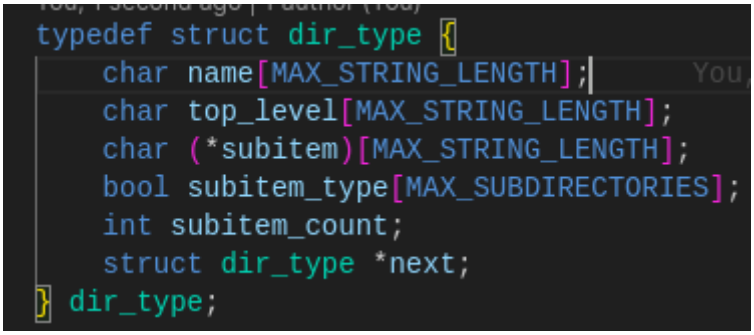
- The GUI interacts with the core file system through the functions and data structures declared in the header file.
- User actions in the GUI trigger function calls to the core file system implementation, which then performs the requested operations and updates the GUI accordingly.

Detailed Description

1. **GTK GUI:**
    - Contains various widgets such as buttons and dialogs that allow users to interact with the file system.
    - For example, when a user clicks the "Create" button, the GUI triggers the `create_file_dialog` function, which eventually calls the corresponding function in the core file system via the header file.
2. **Header File (simulatedFileSystem.h):**
    - Acts as a bridge between the GUI and the core file system implementation.
    - Declares the necessary data structures (e.g., `dir_type`, `file_type`) and function prototypes (e.g., `create_directory`, `delete_file`) used in `simulatedFileSystem.c`.
3. **Core File System (simulatedFileSystem.c):**
    - Implements the main logic for managing files and directories.
    - Functions defined in this file are called by the GUI through the header file to perform operations like creating, deleting, moving, reading, and writing files and directories.
    - Also includes security features such as password protection to secure files and directories.

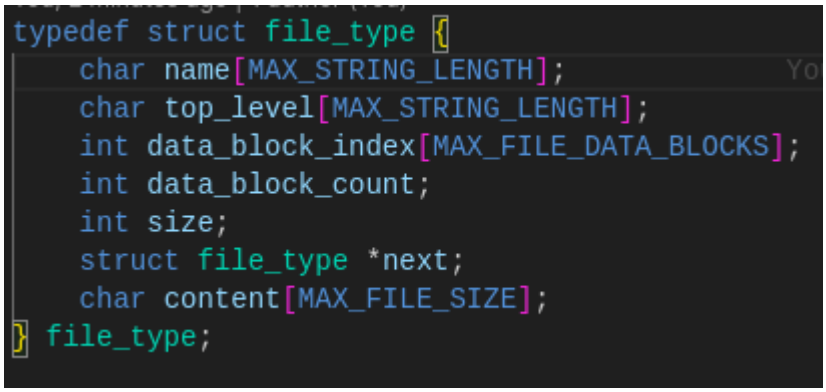
Key Data Structures

dir_type



```c
typedef struct dir_type {
    char name[MAX_STRING_LENGTH];
    char top_level[MAX_STRING_LENGTH];
    char (*subitem)[MAX_STRING_LENGTH];
    bool subitem_type[MAX_SUBDIRECTORIES];
    int subitem_count;
    struct dir_type *next;
} dir_type;
```

file_type

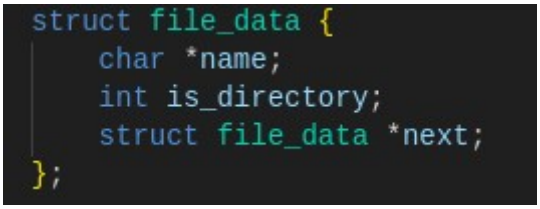


```c
typedef struct file_type {
    char name[MAX_STRING_LENGTH];
    char top_level[MAX_STRING_LENGTH];
    int data_block_index[MAX_FILE_DATA_BLOCKS];
    int data_block_count;
    int size;
    struct file_type *next;
    char content[MAX_FILE_SIZE];
} file_type;
```

securedItem

```c
typedef struct {
    char path[512];
    char password[256];
} SecuredItem;
```

file_data



```c
struct file_data {
    char *name;
    int is_directory;
    struct file_data *next;
};
```

Simulated Disk Space Implementation

Simulated disk space in this project is implemented using arrays and structures in C. The simulated disk space represents storage where files and directories are managed. Each file and directory is stored in a data structure, and operations like creation, deletion, and modification are performed on these structures.

File Allocation Mechanism

The file allocation mechanism in this project uses contiguous allocation. This means that each file is stored in consecutive blocks of memory. The implementation manages these blocks to ensure efficient storage and quick access.

Directory Structure Management

Directories and subdirectories are managed using a tree structure. Each directory can contain multiple subdirectories and files. The `dir_type` structure maintains a list of its subdirectories and files, along with a count of each.

File and Directory Operations

Creation

Creating a file or directory involves allocating memory for the new item and adding it to the parent directory's list of subdirectories or files.

Deletion

Deleting a file or directory involves freeing the allocated memory and removing the item from the parent directory's list.

Moving

Moving a file or directory involves changing its parent directory reference.

Reading and Writing

Reading from and writing to a file involves accessing and modifying its content.

Security Implementation

Password protection for securing files and directories is implemented using a `SecuredItem` structure, which stores a password along with the item as seen above. The functions relating to password security is seen below.

```
static void store_password(const char *path, const char *password) {
    if (secured_item_count < MAX_SECURED_ITEMS) {
        strncpy(secured_items[secured_item_count].path, path, sizeof(secured_items[secured_item_count].
        strncpy(secured_items[secured_item_count].password, password, sizeof(secured_items[secured_item
        secured_item_count++;      You, 2 hours ago • Uncommitted changes
    } else {
        log_event("Maximum number of secured items reached");
    }
}
```

This stores the password and path for the of the item being secured which can be retrieved by other functions to authenticate. It also checks to insure that the number of secure items doesn't exceed the allocation.

GUI Development and Integration

The GUI for this project is developed using GTK. It provides buttons for various file system operations like creating, deleting, moving, and renaming files and directories. All functions and code relating to the GUI implementation can be found in the gui.c file of the project directory.

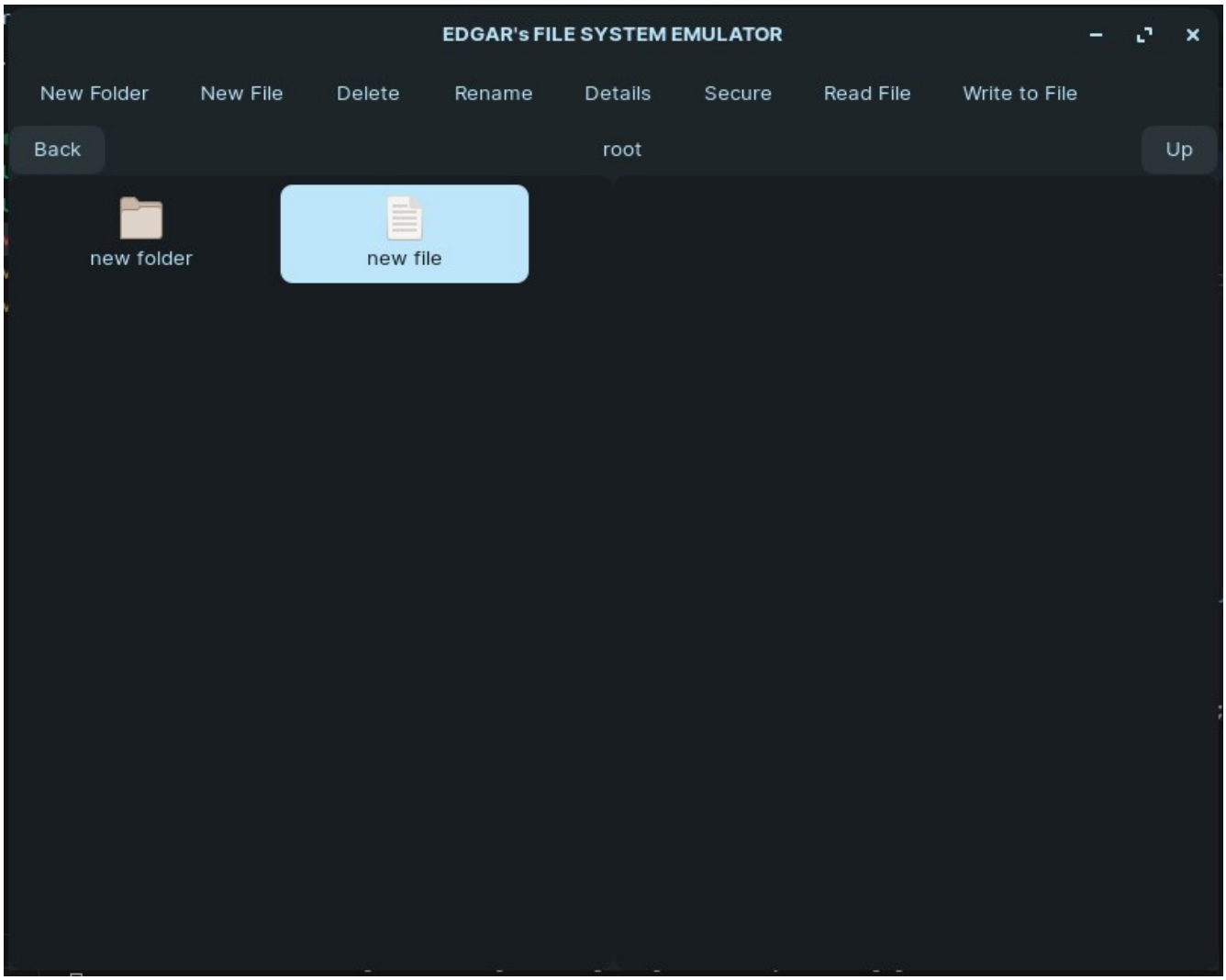
On running the application (using the ./fs in the terminal after the compilation) the main function calls the root function, which activates the file system, and the app function which invokes the activate function callback that initializes the icons, buttons and window into view.

```
int main(int argc, char **argv) {
    // Initialize the file system
    do_root("", "");

    GtkApplication *app;
    int status;

    app = gtk_application_new("org.gtk.example", G_APPLICATION_FLAGS_NONE);
    g_signal_connect(app, "activate", G_CALLBACK(activate), NULL);
    status = g_application_run(G_APPLICATION(app), argc, argv);
    g_object_unref(app);

    return status;
}
```

Results

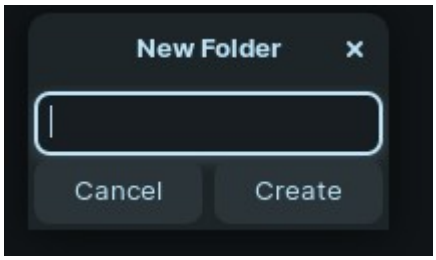
Presentation and Findings

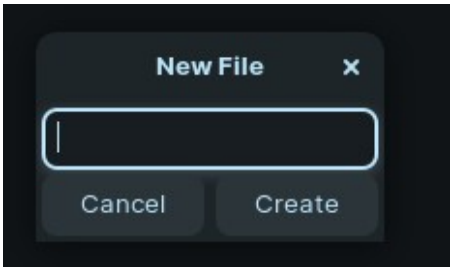
The generated UI looks as shown below.



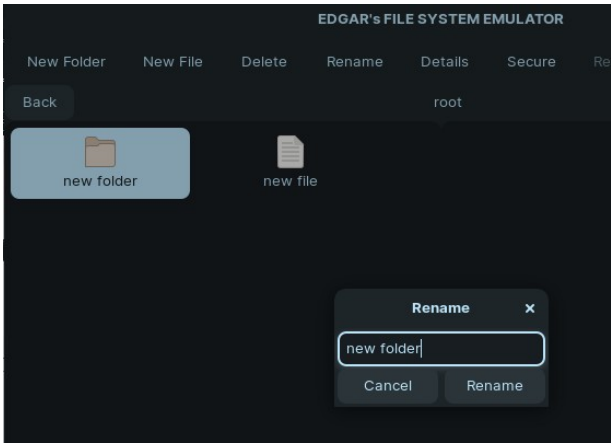
File System Operations

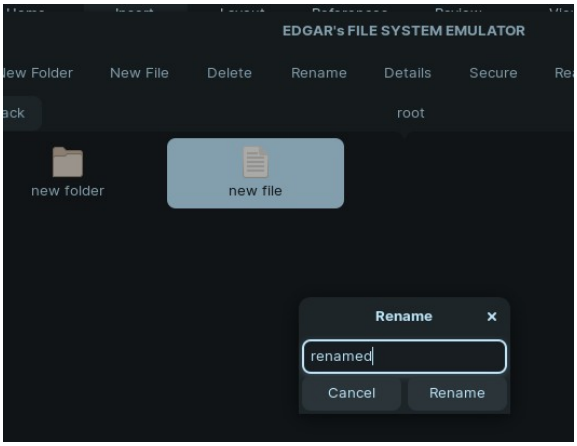
Creating Directories and Files



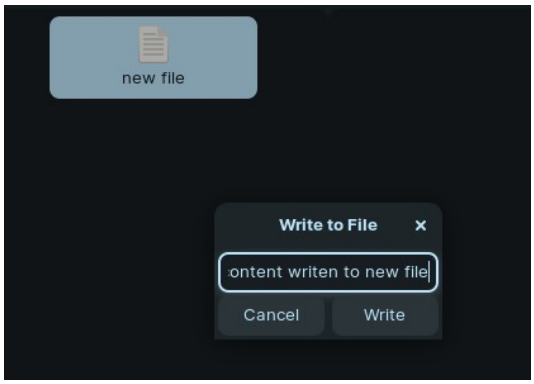


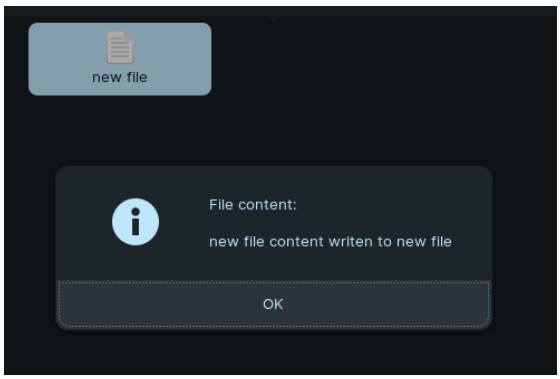
Renaming Items



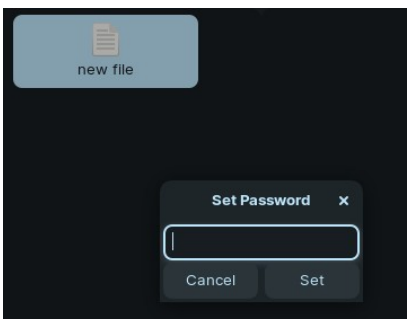


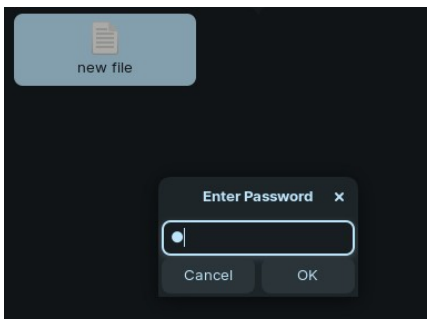
Reading from and Writing to Files

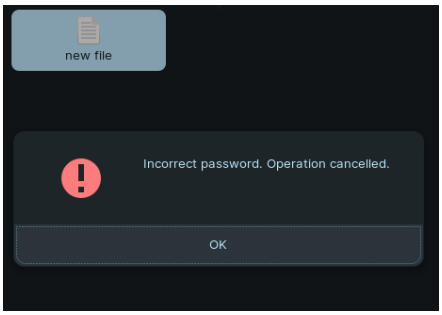




Securing Items with Passwords







Discussion

Interpretation of Results

The simulated file system implemented in this project achieves its primary objectives of creating a functional file system within user space and providing a graphical user interface for interaction. The following points summarize the key findings and interpretations:

- **Functionality**: The file system supports basic operations such as creating, deleting, moving, reading, and writing files and directories. These operations are efficiently managed through a contiguous allocation mechanism, ensuring quick access and modification.
- **Security**: The implementation includes a basic password protection feature, securing files and directories from unauthorized access.
- **User Interface**: The GTK-based GUI offers an intuitive interface for users, facilitating easy interaction with the file system. Users can perform operations through simple button clicks and dialogs.

Analysis of the File System's Functionality

The file system's functionality was evaluated based on its ability to handle core operations:

- **File and Directory Management**: The system allows for the creation, deletion, and movement of files and directories. It also supports renaming and reading/writing file contents.
- **Allocation Mechanism**: Contiguous allocation ensures that files are stored in consecutive memory blocks, making access times faster but potentially leading to fragmentation issues.
- **Security**: Password protection is a critical feature, albeit basic, that enhances the security of the file system.
- **GUI Integration**: The seamless integration of the file system with the GTK GUI allows users to perform file system operations visually, enhancing usability.

Comparison with Objectives

- **Implementation in User Space**: Successfully implemented a simulated file system in user space using C.
- **GUI Development**: Developed a graphical user interface using GTK, allowing users to interact with the file system.
- **Basic Security Features**: Incorporated password protection for files and directories, aligning with the security objectives.

Strengths of the Implementation

- **User-Friendly Interface**: The GUI makes it easy for users to manage files and directories without needing to interact directly with the code.
- **Basic Security**: The password protection feature adds a layer of security to the file system.
- **Efficient Operations**: The use of contiguous allocation provides efficient file access and management.

Limitations

- **Fragmentation**: Contiguous allocation can lead to fragmentation, making it difficult to allocate large files if there is no contiguous free space.
- **Security**: While password protection is implemented, it is basic and can be improved with more robust security measures.
- **Scalability**: The current implementation may not scale well with a very large number of files and directories due to limitations in memory management.

Challenges Faced

- **Memory Management**: Ensuring efficient use of memory and avoiding fragmentation was a significant challenge.
- **GUI Integration**: Integrating the core file system with the GUI required careful coordination to ensure seamless interaction.
- **Security Implementation**: Designing and implementing a basic security feature that is both functional and user-friendly was challenging.

Conclusion

Summary of the Project Achievements

- Successfully implemented a simulated file system in user space using C.
- Developed a user-friendly GUI using GTK for file system interaction.
- Incorporated basic password protection to enhance security.
- Achieved efficient file and directory management through contiguous allocation.

Reflection on Learning Outcomes

- Gained hands-on experience in implementing a file system in user space.
- Learned how to develop and integrate a GUI with a core system in C.
- Understood the importance and challenges of memory management and security in file systems.

Potential Applications

- **Educational Tool**: The simulated file system can be used as an educational tool to teach students about file system concepts and operations.
- **Research**: Can serve as a foundation for research in file system optimization and security.
- **Prototyping**: Useful for prototyping new file system features and testing them in a controlled environment.

Suggestions for Future Improvements

- **Enhanced Security**: Implement more robust security features such as encryption and multi-factor authentication.
- **Improved Allocation Mechanisms**: Explore other allocation methods like linked or indexed allocation to reduce fragmentation.
- **Scalability**: Optimize the system to handle a larger number of files and directories more efficiently.
- **User Experience**: Enhance the GUI with more features and better user experience.

References

- Socoyash. (n.d.). *Simulated file system source code*. GitHub repository. Retrieved July 30, 2024, from https://github.com/scocoyash/File-System-in-C/blob/master/simulatedFileSystem.c
- GTK Documentation. (n.d.). Retrieved from https://docs.gtk.org/
- Ntalasha, D. B. (n.d.). *Operating systems-CS225 unit 3 lecture notes*.
- Debugging tools from ChatGPT and Claude AI. (n.d.).

Appendix

The code can be found in Edgar Mwila's git repository at https://github.com/Edgar-mwila/file_system_emulator_with_c./tree/v1.2 .