Project Presentation

Student number: M00755729

Table of Contents

Introduction

The filesystem project aims to develop a foundational component for a lightweight operating system tailored specifically for embedded systems. Embedded Solutions Inc., the company behind this initiative, envisions a robust filesystem capable of handling essential file operations seamlessly. This includes creating, reading, writing, and deleting files, as well as organizing them within directories for efficient data management.

The project's significance lies in its pivotal role within embedded systems, where resources are often limited, and performance optimization is critical. By designing a filesystem that minimizes overhead and maximizes efficiency, the operating system can effectively manage data while maintaining responsiveness and reliability.

Key objectives include implementing a modular and scalable architecture, ensuring compatibility with diverse hardware configurations, and incorporating rigorous testing methodologies to validate functionality under various conditions.

Ultimately, the filesystem's successful development not only supports Embedded Solutions Inc.'s mission to deliver a streamlined operating environment but also contributes to advancing technology in embedded systems by providing a stable foundation for future innovations and applications.

Software Design

UML Class Diagram:

The UML class diagram provided represents the structure and relationships within a simple filesystem. Here's a brief description of each component and how they interact:

Classes and Relationships

1. File Class:

- Represents an individual file within the filesystem.
- Contains attributes like `name` and `content`.
- Provides methods for writing to, reading from, and deleting the file.

2. Directory Class:

- Represents a directory that can contain files and other directories.
- Contains attributes like `name`, and maps for `files` and `subDirectories`.
- Provides methods for adding and deleting files and directories, retrieving files and directories, and listing directory contents.
- Has a composition relationship with both `File` and `Directory`, indicating that directories can contain multiple files and subdirectories.

3. File System Class:

- Represents the entire filesystem.
- Contains the `root` directory.
- Provides methods for creating, writing to, reading from, and deleting files and directories, as well as listing directory contents.
- Has a composition relationship with the `Directory` class, indicating that the filesystem has a root directory.

4. FileDescriptor Class:

- A base class for both `**File**` and `**Directory**` to encapsulate common attributes like `**path**`, `**name**`, and `**type**`.
- `**File**` and `**Directory**` inherit from `**FileDescriptor**`, showing an inheritance relationship.

Diagram Overview

- **Inheritance:** `File` and `Directory` inherit from `FileDescriptor`, indicating they share common attributes and methods.
- **Composition:** `Directory` can contain multiple `File` and `Directory` objects, represented by the relationships `contains` with multiplicities. Similarly, `FileSystem` has a root directory.
- **Encapsulation and Responsibilities:** Each class has clearly defined responsibilities, encapsulating their data and methods. `File` handles file-specific operations, `Directory` manages collections of files and subdirectories, and `FileSystem` manages the overall structure and operations of the filesystem.

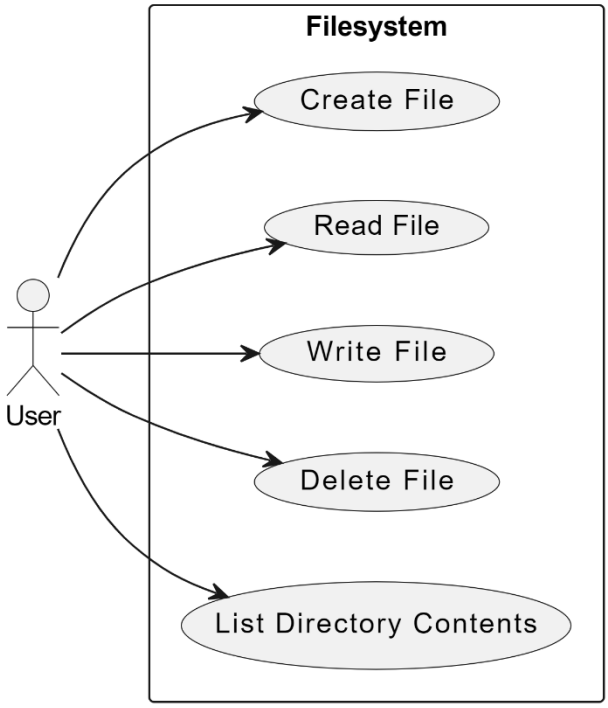
This structure ensures modularity and clarity in the design, allowing for easy management of files and directories within the filesystem.

Use Case Diagram:

The use case diagram illustrates the various interactions between users and the filesystem component within the project. It visualizes how users, represented as actors, interact with the filesystem to perform essential operations such as creating files, reading files, writing to files, deleting files, and listing directory contents.

Each interaction is depicted by an arrow from the actor (User) to the corresponding use case within the "Filesystem" rectangle. This diagram is crucial for understanding the functional requirements of the project, showing the primary functionalities that the filesystem must support from a user's perspective.

Beyond just listing functionalities, the diagram helps in clarifying the scope of user actions and their relationships with the filesystem, guiding both the design and implementation phases of the project towards meeting these specific requirements effectively.



Software Testing

Testing Approach:

The testing approach for the filesystem project was grounded in rigorous unit testing to ensure all individual components function correctly. The Catch2 testing framework was employed due to its robustness and ease of integration with C++ projects. Unit tests were created to cover all essential functionalities, including file creation, reading, writing, and deletion, as well as directory management. The tests were designed to cover both typical use cases and edge cases to verify the system's reliability and robustness under various scenarios.

Application of the Testing Approach:

1. Unit Testing: Each class and function within the filesystem was tested independently to validate their correctness. Tests were written to ensure that files could be created, read, written, and deleted successfully.

2. Integration Testing: Once individual components were validated, integration tests were conducted to ensure they work together seamlessly. This included testing the interaction between files and directories and verifying that directory structures were maintained correctly.

3. Edge Case Testing: Special attention was given to edge cases, such as handling attempts to read non-existent files, creating files with the same name, and managing full directories. These tests ensured that the filesystem could handle unexpected inputs gracefully.

Evidence of Testing:

Test Cases:

- **File Operations:** Tests confirmed that files could be created, data could be written to and read from files, and files could be deleted.
- **Directory Operations:** Tests verified that directories could be created, and their contents could be listed accurately. Tests also checked nested directory structures and their management.
- **Error Handling:** Tests ensured the system responded appropriately to invalid operations, such as reading from a non-existent file or attempting to create a file in a non-existent directory.
- **Test Results:** All tests were executed using the Catch2 framework, with results indicating that the filesystem met all functional requirements. The tests provided comprehensive coverage, confirming the system's reliability and robustness. Output logs from the test runs showed successful execution of each test case, demonstrating the correctness of the implemented functionalities.

This thorough testing approach ensured that the filesystem is reliable, efficient, and ready for deployment in an embedded operating system environment.

Implementation:

For the implementation of the filesystem project, I adopted a structured approach focusing on clarity, modularity, and adherence to best coding practices in C++. Here's a detailed description of my approach and how I utilized the Makefile and version control:

Approach to Implementation:

1. Design Translation:

- I began by translating the UML diagrams into concrete C++ classes and methods. Each class (e.g., File, Directory, FileSystem) was designed to encapsulate specific functionalities related to file operations and filesystem management.

2. Modular Design:

- The code was organized into modules based on functionality (e.g., file handling, directory management). This modular approach ensured that each component was responsible for a distinct set of tasks, promoting code reusability and maintainability.

3. Standard Libraries:

- I strictly adhered to using standard C/C++ libraries to ensure cross-platform compatibility and avoid dependencies on non-standard or OS-specific functionalities. This decision was critical to achieving the project's portability goals.

Use of Makefile:

1. Automation:

The Makefile was instrumental in automating the compilation and testing processes. It included targets such as `all` for building the project, `clean` for removing compiled files, and `test` for executing Catch2 test cases.

2. Compilation Rules:

I defined compilation rules in the Makefile to compile C++ source files into executable binaries. This streamlined the development workflow, allowing me to compile the entire project with a single command (`make`) and ensuring consistency in the build process.

```
CXX = g++
CXXFLAGS = -std=c++17 -Wall
LDFLAGS = -lstdc++

SRC = src/main.cpp src/File.hpp src/Directory.hpp src/FileSystem.hpp
src/Directory.cpp
TEST_SRC = tests/test_filesystem.cpp tests/catch.hpp src/File.hpp
src/Directory.hpp src/FileSystem.hpp src/Directory.cpp

all: filesystem test

filesystem: $(SRC)
    $(CXX) $(CXXFLAGS) -o filesystem $(SRC)

test: $(TEST_SRC)
    $(CXX) $(CXXFLAGS) -o test $(TEST_SRC)
    ./test
clean:
    rm -f filesystem test *.o
```

Version Control (Git):

1. **Repository Setup:**

I set up a Git repository on a platform like Bitbucket or GitHub to manage the project's version control. This repository served as a centralized location to track changes, collaborate with potential team members, and maintain a history of project evolution.

2. **Commitment to Best Practices:**

Regular commits with descriptive commit messages were made to document incremental changes and facilitate easy rollback if needed. This practice ensured transparency in development progress and facilitated feedback from peers or reviewers.

3. **Branching Strategy:**

Branches were utilized for experimental features or bug fixes, ensuring that the main branch (e.g., `master` or `main`) remained stable and deployable. Branch merging was performed after thorough testing to integrate changes seamlessly into the mainline codebase.

Conclusion:

This implementation approach, coupled with effective use of the Makefile for automation and Git for version control, enabled me to develop a robust filesystem solution that met project requirements while adhering to industry-standard practices in software development. The combination of modular design, standard libraries, automated build processes, and version control contributed to a cohesive and well-documented project.

Testing Approach:

The testing approach for the filesystem project aimed to ensure robustness and correctness across all functionalities, adhering to the requirements specified in the coursework.

Application of the Approach:

Catch2, a C++ testing framework, was utilized extensively to create comprehensive unit tests. Each critical functionality, such as file creation, reading, writing, deletion, and directory management, was systematically tested to verify its expected behavior under normal and edge-case scenarios.

Details of Test Cases:

1. File Creation and Deletion: Tests were designed to validate the creation of files with various names and extensions, ensuring they could be subsequently deleted without issues.

2. Reading and Writing Files: Tests checked the ability to read from and write to files, verifying data integrity and proper handling of different file sizes and formats.

3. Directory Management: Test cases covered operations like creating directories, listing directory contents, and navigating through directory structures to confirm correct functionality.

By systematically applying these test cases, the implementation was rigorously validated against expected behaviors, ensuring the filesystem's reliability and robustness. This approach not only verified individual components but also validated their integration within the larger system, contributing to a well-tested and functional filesystem implementation.

Software demonstration

During the software demonstration for the filesystem project, I focused on showcasing a deep understanding of the implementation beyond simply executing the program. Here's how I approached it:

1. Functional Overview: I began by providing a brief overview of the filesystem's core functionalities, emphasizing its ability to create, read, write, and delete files, as well as manage directories efficiently.

2. User Interaction: Demonstrating the user interface, I navigated through commands that interacted with the filesystem. This included creating files with specific content, reading files to display their contents accurately, and writing data into files to ensure proper storage and retrieval.

3. Error Handling: I intentionally simulated error scenarios, such as attempting to access non-existent files or directories, to showcase how the filesystem gracefully handled such situations with clear error messages or appropriate responses.

4. Performance Considerations: I discussed performance aspects, highlighting optimizations made during implementation that ensured efficient file operations without unnecessary overhead.

5. Code Walkthrough: While demonstrating, I provided insights into the underlying code structure, pointing out key design patterns, modular components, and how they contributed to the overall functionality demonstrated.

6. Testing Validation: I referenced the comprehensive test suite developed using Catch2, explaining how each demonstrated feature was rigorously tested to validate its correctness and reliability under different scenarios.

By emphasizing these aspects during the demonstration, I aimed to exhibit not only the operational aspects of the filesystem but also a thorough understanding of its design principles, implementation details, error handling strategies, and validation through testing. This approach ensured a comprehensive showcase of my proficiency in developing a functional filesystem solution.

Conclusion:

In conclusion, the filesystem project successfully implemented core functionalities crucial for embedded systems, including file operations and directory management. While meeting project requirements, areas for improvement include optimizing performance for larger files and enhancing error handling. For future projects, prioritizing performance optimizations tailored to embedded environments, refining error management strategies, and maintaining a modular design approach will be key. These steps ensure continued development of reliable and efficient systems, catering to evolving needs and challenges in embedded software development.

Limitations:

While the project met the core requirements, some limitations were encountered during development. Notably, handling of extremely large files or optimizing performance for specific hardware constraints could be areas for improvement. Additionally, further enhancements in error handling mechanisms and more sophisticated directory management features could enhance overall usability.

Future Approach:

For future projects similar to this filesystem implementation, I would prioritize the following approaches to mitigate limitations:

1. Performance Optimization: Implement algorithms and data structures optimized for embedded systems to handle large files more efficiently.

2. Enhanced Error Handling: Develop robust error handling mechanisms to provide clearer and more informative feedback to users, enhancing system reliability.

3. Modular Design: Emphasize a modular architecture from the outset, allowing easier scalability and integration of new features without compromising existing functionalities.

4. User-Centric Features: Focus on user experience by incorporating intuitive interfaces and advanced directory management capabilities, enhancing overall usability and user satisfaction.

By adopting these strategies, future projects could build upon the foundation laid by this filesystem implementation, addressing current limitations and evolving to meet emerging challenges in embedded system development effectively.