DECS

Project plan

Project Plan: Versioning-Based File System with Snapshots, Rollback, and Diff Visualization

---

Overview

To build a versioning-based file system with features like snapshots, rollback, and diff visualization, using the **FUSE (Filesystem in Userspace)** API is the most practical approach. FUSE allows you to develop a file system in user space without modifying kernel code, offers flexibility, and is well-documented with community support.

Project Breakdown

The project is divided into modular parts, each with detailed steps and sub-steps. This structure facilitates parallel development and clarity.

Part 1: Environment Setup

Step 1.1: Install FUSE and Dependencies

- **Sub-step 1.1.1**: Install FUSE library
  - **For Linux**:
    - Run `sudo apt-get install libfuse-dev` (Debian/Ubuntu)
    - Or `sudo yum install fuse-devel` (CentOS/Fedora)
  - **For macOS**:
    - Install osxfuse
- **Sub-step 1.1.2**: Verify Installation

    - Compile and run a sample FUSE file system (e.g., hello.c from FUSE examples)

Step 1.2: Set Up Development Environment

- **Sub-step 1.2.1**: Choose Programming Language
    - **Option 1**: **C/C++** (for performance and direct FUSE API access)
    - **Option 2**: **Python** (easier development, use fusepy)
- **Sub-step 1.2.2**: Set Up IDE/Editor
    - Install preferred IDE (e.g., Visual Studio Code, CLion)
    - Configure build tools and debugger

Part 2: File System Design

Step 2.1: Define File System Structure

- **Sub-step 2.1.1**: File and Directory Representation
    - Use a hierarchical structure mirroring standard file systems
- **Sub-step 2.1.2**: Versioning Mechanism
    - Each file has associated versions stored separately
    - Use a hidden directory (e.g., `.versions/`) to store version data

Step 2.2: Design Data Structures

- **Sub-step 2.2.1**: File Metadata Structure
    - **Struct/FileMetadata**:
        - `filename` (string)
        - `attributes` (permissions, size, timestamps)
        - `version_list` (list of version identifiers)
- **Sub-step 2.2.2**: Version Information Structure
    - **Struct/VersionInfo**:

    - `version_id` (incremental number or UUID)
    - `timestamp`
    - `data_pointer` (reference to versioned data)

Step 2.3: Decide Storage Mechanism

- **Sub-step 2.3.1**: Metadata Storage
    - Store metadata in JSON or binary files within `.metadata/` directory
- **Sub-step 2.3.2**: Version Data Storage
    - Store complete copies or deltas in `.versions/` directory
    - Implement delta encoding if storage optimization is needed

---

Part 3: Implement Basic File System Operations

Step 3.1: Implement FUSE Callbacks

- **Sub-step 3.1.1**: `getattr` Implementation
    - Return file or directory attributes
- **Sub-step 3.1.2**: `readdir` Implementation
    - List contents of a directory
- **Sub-step 3.1.3**: `open` Implementation
    - Handle file opening logic
- **Sub-step 3.1.4**: `read` Implementation
    - Read data from files
- **Sub-step 3.1.5**: `write` Implementation
    - Write data to files
- **Sub-step 3.1.6**: `create` and `unlink` Implementation
    - Handle file creation and deletion
- **Sub-step 3.1.7**: `mkdir` and `rmdir` Implementation

- Handle directory creation and deletion

Step 3.2: Testing Basic Operations

- **Sub-step 3.2.1**: Mount File System
  - Use FUSE to mount the file system at a designated mount point
- **Sub-step 3.2.2**: Test Operations
  - Use shell commands (`ls`, `cat`, `echo`, `rm`) to test functionality

---

Part 4: Implement Versioning Support

Step 4.1: Enhance Write Operations

- **Sub-step 4.1.1**: Modify `write` Callback
  - Before writing, save the current version to `.versions/`
- **Sub-step 4.1.2**: Update Version Metadata
  - Increment version number
  - Update `version_list` in file metadata

Step 4.2: Adjust Read Operations

- **Sub-step 4.2.1**: Default to Latest Version
  - Ensure `read` fetches data from the latest version
- **Sub-step 4.2.2**: Access Specific Versions
  - Implement method to read specific versions (e.g., extended attributes or special file naming)

Step 4.3: Manage Version Metadata

- **Sub-step 4.3.1**: Store Version Details
  - Maintain a log of versions with timestamps and identifiers
- **Sub-step 4.3.2**: Implement Cleanup Policies
  - Optional: Implement version pruning to limit storage usage

Part 5: Implement Snapshot Functionality

Step 5.1: Create Snapshot Command

- **Sub-step 5.1.1**: Define Snapshot Trigger
  - Use a special file (e.g., `.snapshot`) where writing triggers a snapshot
- **Sub-step 5.1.2**: Implement Snapshot Identifier
  - Assign unique IDs or timestamps to each snapshot

Step 5.2: Store Snapshot Data

- **Sub-step 5.2.1**: Record File Versions
  - Map current file versions to the snapshot ID
- **Sub-step 5.2.2**: Save Snapshot Metadata
  - Store in a file within `.snapshots/` directory

Step 5.3: List Snapshots

- **Sub-step 5.3.1**: Implement Snapshot Directory
  - Expose `.snapshots/` as a readable directory
- **Sub-step 5.3.2**: Provide Snapshot Details
  - Include snapshot IDs and timestamps in listings

Part 6: Implement Rollback Functionality

Step 6.1: Create Rollback Command

- **Sub-step 6.1.1**: Define Rollback Mechanism
  - Use a special file (e.g., `.rollback`) where writing a snapshot ID triggers rollback
- **Sub-step 6.1.2**: Validate Snapshot ID
  - Ensure the provided snapshot ID exists

Step 6.2: Restore Files from Snapshot

- **Sub-step 6.2.1**: Iterate Over Files
  - For each file, revert to the version in the snapshot
- **Sub-step 6.2.2**: Handle Created/Deleted Files
  - Delete files not present in the snapshot
  - Restore deleted files present in the snapshot
- **Sub-step 6.2.3**: Update Metadata
  - Reflect changes in file metadata and version logs

Part 7: Implement Diff Visualization

Step 7.1: Develop Diff Command

- **Sub-step 7.1.1**: Define Diff Trigger
  - Use a special file (e.g., `.diff`) where reading outputs differences
- **Sub-step 7.1.2**: Accept Parameters
  - Allow specifying snapshot IDs or file paths for comparison

Step 7.2: Calculate Differences

- **Sub-step 7.2.1**: File-Level Diff
  - Use `difflib` (Python) or `diff` utilities to compare file contents
- **Sub-step 7.2.2**: Directory-Level Diff
  - Compare file lists and versions between snapshots
- **Sub-step 7.2.3**: Storage Statistics
  - Calculate space used by versions and snapshots

Step 7.3: Format and Present Output

- **Sub-step 7.3.1**: Choose Output Format
  - Textual diffs for files

    - Summaries for directories
- **Sub-step 7.3.2**: Display Output
    - Write diff results to standard output or a designated file

---

Part 8: Testing and Debugging

Step 8.1: Write Test Cases

- **Sub-step 8.1.1**: Unit Tests for Each Module
    - Test file operations, versioning, snapshots, rollback, and diffs
- **Sub-step 8.1.2**: Integration Tests
    - Simulate user workflows and edge cases

Step 8.2: Implement Logging

- **Sub-step 8.2.1**: Add Log Statements
    - Log operations, errors, and important state changes
- **Sub-step 8.2.2**: Configure Log Levels
    - Allow adjusting verbosity for debugging

Step 8.3: Optimize Performance

- **Sub-step 8.3.1**: Profile the File System
    - Identify bottlenecks using profiling tools
- **Sub-step 8.3.2**: Optimize Critical Paths
    - Improve read/write efficiency
    - Optimize metadata access

Part 9: Documentation

Step 9.1: User Documentation

- **Sub-step 9.1.1**: Create README File
  - Overview of the file system and its features
- **Sub-step 9.1.2**: Usage Guide
  - Instructions on mounting, commands for snapshots, rollback, and diff
- **Sub-step 9.1.3**: Examples
  - Provide sample commands and expected outcomes

Step 9.2: Developer Documentation

- **Sub-step 9.2.1**: Code Comments
  - Explain complex logic and important functions
- **Sub-step 9.2.2**: Architecture Overview
  - Describe modules, data structures, and their interactions
- **Sub-step 9.2.3**: Contribution Guidelines
  - Outline coding standards and procedures for future development

---

Modularization for Prompting

The project plan is structured to allow each part to be independently developed or used as a prompt for code generation. Each module corresponds to specific functionalities:

- **Module 1**: Basic File System Operations (Part 3)
- **Module 2**: Versioning Mechanism (Part 4)
- **Module 3**: Snapshot Functionality (Part 5)
- **Module 4**: Rollback Mechanism (Part 6)
- **Module 5**: Diff Visualization (Part 7)
- **Module 6**: Testing Suite (Part 8)
- **Module 7**: Documentation (Part 9)

Implementation Tips

- **Data Storage**: Use a consistent method for storing metadata and version data to simplify read/write operations.
- **Concurrency**: Consider thread safety if the file system will be accessed concurrently.
- **Error Handling**: Ensure all possible errors are handled gracefully, providing informative messages.

Conclusion

By following this detailed plan, you can methodically develop a versioning-based file system with snapshots, rollback, and diff visualization using the FUSE API. The modular approach allows for scalability and ease of maintenance, and the clear breakdown facilitates both individual and collaborative work.

Note: Each step and sub-step is designed to be self-contained, providing sufficient detail to implement the functionality without referencing external documents. This structure ensures that if passed as a prompt to a language model or used by a developer, the project can be fully realized.