Design Document for memFS (CL Project)

Gana Jayant Sigadam
Roll No: 24CS60R12

November 17, 2024

1 Overview

`memFS` is a Command Line Interface (CLI) tool designed to create files in-memory and perform high-speed operations on them. Once the program terminates, all the files created in memory are lost. This makes `memFS` particularly useful for testing scenarios where temporary file operations are required without impacting the actual disk.

Key Features

- Fast in-memory file operations: Files are created, written, and deleted directly in memory, enabling faster operations compared to disk I/O.
- Temporary storage: All in-memory files are lost once the program ends, ensuring no residual files on disk.
- Concurrency: Supports multi-threaded operations to handle multiple files simultaneously, enhancing performance for bulk operations.

Implementation Details

- Language and Concepts: The tool is implemented in **C++** and leverages **multi-threading** to achieve high performance.
- Command Parsing: To handle user input, a custom parser has been implemented. This parser validates commands and arguments, ensuring correct execution. The code for the parser can be found in the `CommandInterpreter.hpp` file.
- Concurrency for Operations:
  - Multi-threading is used to optimize **create**, **write**, and **delete** operations.
  - Users can specify multiple files for operations, and threads handle these operations in parallel, ensuring efficiency.

Flow Diagram

To better understand the process flow of the `memFS` tool, refer to the flow diagram below.

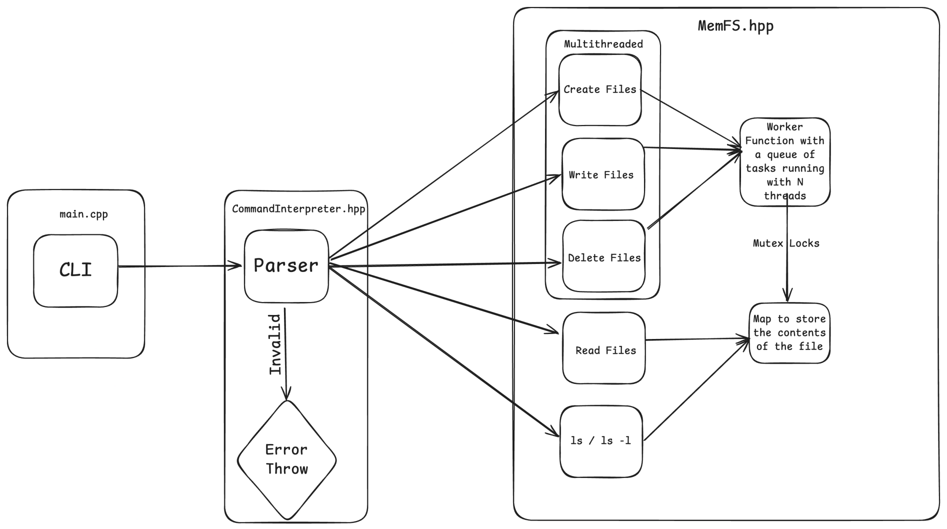

Figure 1: Flow Diagram of `memFS`

2 Context

Why we need this tool?

- Since RAM is significantly faster than traditional storage, operations such as reading, writing, creating, or deleting files are much quicker in MemFS.
- Traditional filesystems incur latency due to physical storage access, disk scheduling, and caching. MemFS bypasses these layers, reducing overhead.
- MemFS is useful for testing purposes where we need to create files and perform operations on them without actually creating them on disk.

3 Goals & Non-Goals

3.1 Goals

- Create files in-memory.
- Write data to files in-memory.
- Read data from files in-memory.
- Delete files in-memory.
- Faster operations on files.

3.2 Non-Goals

- This tool is not intended to store files permanently.
- This tool is not intended to store large files.

4 Proposed Solution

4.1 Parsing Commands

First, I have implemented a command parser that validates user input and extracts the command and arguments. The parser ensures that the command is valid and the arguments are correctly formatted using regex and matching techniques. The parser is implemented in the `CommandInterpreter.hpp` file.

4.2 MemFS Logic

4.2.1 Data Structuring for Files

To store files in memory, I have used a `std::map` data structure where the key is the file name and the value is the `File` object. The `File` object contains the content, size, create time, and last modified time of the file which will be used for operations.
This acts as a critical section for the threads to access the file system. The critical section is protected by a mutex to ensure thread safety.
Worker threads are spawned to handle file operations concurrently. The number of worker threads can be passed in the constructor of the `MemFS` class.
This worker threads constantly check are there any operations in the queue. If there are any operations, they will pick the operation and execute it.

4.2.2 Worker Queue

The worker queue is a `std::queue` data structure that stores the operations to be performed on files. The worker threads continuously check the queue for operations. If an operation is present, the worker thread picks the operation and processes it and after processing it let's the main thread know that the operation is completed using `std::condition_variable`.

4.2.3 Create Operation

This operation takes multiple file names as input and creates files in memory. The files are stored in the `std::map` data structure. Each file is added to the worker queue for processing. Corresponding errors are displayed if the file already exists.

4.2.4 Write Operation

This operation takes multiple file names and data as input and writes the data to the files in memory. The files are updated in the `std::map` data structure. Each file is added to the worker queue for processing. Corresponding errors are displayed if the file does not exist.

4.2.5 Read Operation

This operation is single-threaded and takes one filename as input. It reads the content of the file from memory and displays it. Corresponding errors are displayed if the file does not exist.

4.3 Delete Operation

This operation takes multiple file names as input and deletes the files from memory. The files are removed from the `std::map` data structure. Each file is added to the worker queue for processing. Corresponding errors are displayed if the file does not exist.

4.4 Ls Operation

This operation is single-threaded and displays the list of files present in memory. There is special flag `-l` which will display the detailed information of the files.

5 Benchmarking

For the benchmarking of the tool please refer to the Benchmarking document.

6 Usage of the tool

Please refer to the README file for the usage of the tool.

7 Open Questions

The futures scope of the project is to implement the following features:

- Implement a lock-free data structure for file operations essentially a atomic hash map.
- Implementing actual file system with directory structure & additional commands like `cd`, `cp`.