Design Document: memFS- A Fast, In-Memory File System

Ajay Choudhury (24CS60R85)
CS69201: Computing Lab- 1
Department of Computer Science and Engineering(CSE)
Indian Institute of Technology Kharagpur, West Bengal, India

1 System Overview

memFS is an in-memory file system designed for high-speed data access and storage utilizing RAM. The system provides a command-line interface for file operations and supports multi-threaded execution for enhanced performance.

1.1 Key Features

- In-memory storage of files up to 2KB in size.
- Thread-safe operations using mutex locks.
- Command-line interface for user interaction.
- Support for both single and batch file operations.
- Real-time file metadata tracking

2 Design

2.1 Code Structure

- The file structure, memFS class and declaration of functions are in `memfs.h`.
- The class functions are defined in `memfs.cpp`.
- Later in `main.cpp` uses user input to decide which operation to perform.
- To compile the code, on terminal run `make` and then `./memfs` or `./benchmark` as required.

2.2 Core Components

- **File structure:**

```
struct File {
string name; // Name of the file
string content; // File content
size_t size; // Current size in bytes
string creation_time; // Creation timestamp
string last_modified; // Last modification timestamp
};
```

- **memFS Class:**

```
class MemFS {
private:
unordered_map<string, File> files; // File storage
mutex fs_mutex; // Thread synchronization
public:
// Core operations
void createFiles();
void writeFile();
void deleteFiles();
string readFile();
vector<vector<string>> listFiles();
};
```

2.3 Data Structures

- **Primary storage:**
  - Uses 'unordered_map' for O(1) file lookup.
  - Key: filename (string).
  - Value: File structure containing metadata like size, creation date and last modified date and content.
- **Thread safety:**
  - Mutex-based synchronization.
  - Lock granularity at the filesystem level.
  - Prevents race conditions during concurrent operations.

3 Component Design

3.1 Command Processor

- Parses user input into commands and arguments.
- Supports both single and batch operations.
- Validates input parameters and file constraints.
- Handles error conditions gracefully

3.2 File Operations

- **Create operation**
  - Validates filename uniqueness.
  - Initializes file metadata.
  - Supports batch creation with '-n' flag, e.g. `create -n 2 todo1.txt todo2.txt`
  - Example for single file creation: `create todo.txt`
  - Thread-safe implementation.
- **Write operation**
  - Validates file existence and size limits (2KB max).
  - Updates content and metadata.
  - Supports batch writing with '-n' flag, e.g. `write -n 2 todo1.txt ‘‘Hello kitty" todo2.txt ‘‘Previous message was a doll"`
  - Example for single file writing: `write todo.txt ‘‘Hello world"`
  - Thread-safe implementation.
- **Delete operation**
  - Validates file existence.
  - Removes the file from storage.
  - Supports batch deletion with '-n‘ flag, e.g. `delete -n 2 todo1.txt todo2.txt`
  - Example of deleting a single file is: `delete todo.txt`
  - Thread-safe implementation.
- **Read operation**
  - Returns file content if exists.
  - Throws exception for non-existent files.
  - Single file operation, e.g., `read todo.txt`
- **List operation**
  - Supports basic and detailed listing.
  - Shows file metadata with '-l‘ flag, e.g., `ls -l`
  - Simple `ls` command shows the available filenames.
  - Thread-safe implementation.

3.3 Multi-threading Implementation

- Thread pool for batch operations.
- Mutex-based synchronization.
- Atomic operations for thread safety.
- Load distribution across available threads.

4 Performance Considerations

4.1 Time Complexity

- File Creation: O(1).
- File Deletion: O(1).
- File Reading: O(1).
- File Writing: O(1).
- File Listing: O(n), where n is number of files.

4.2 Space Complexity

- Per File: O(content_size + metadata_size).
- Total: O(n * (avg_content_size + metadata_size))
- Maximum file size: 2KB
- Limited only by available RAM

4.3 Optimization Techniques

- **Memory Management:**
  - Direct memory access for fast operations.
  - No disk I/O overhead.
  - Efficient memory allocation.
- **Concurrency:**
  - Fine-grained locking for reduced contention.
  - Batch operation optimization.
  - Thread-safe data structures.

5 Error Handling

5.1 Error cases

- File already exists during creation.
- File doesn't exist during read/write/delete.
- Content size exceeds 2KB limit.
- Invalid command syntax.
- Insufficient memory.

5.2 Error Response

- Clear error messages.
- Partial success handling in batch operations.
- Exception handling for critical errors.
- Graceful degradation under load.