# Report on MemFS Benchmarking Results

## Overview

The MemFS class simulates a memory-based file system that allows basic operations like file creation, writing, reading, and deletion. This benchmark aims to measure the system's performance, focusing on latency, CPU usage, and memory consumption as the number of files and threads increases.

The benchmarking code is executed across varying thread counts (1, 2, 4, 8, and 16) and file counts (100, 1000, and 10000) to observe how concurrent operations affect system performance and resource utilization.

## Code Explanation

The main components of the code are:

1. **FileMetadata struct**: Stores metadata for each file, including content, size, creation date, and last modification date.
2. **MemFS class**: Implements file operations:
    - createFile: Creates a file with initial metadata.
    - writeFile: Writes content to an existing file.
    - readFile: Reads content from an existing file.
    - deleteFile: Deletes a file from memory.
3. These operations are synchronized with mutex locks to ensure thread safety during concurrent access.
4. **MemFSBenchmark class**: Manages the benchmarking process by performing file operations in batches to control resource usage.
    - Files are created, written to, read from, and deleted in separate threads to simulate concurrent file access.
    - Batch processing and multithreading allow us to observe how system performance scales with increasing file and thread counts.

# Benchmarking Methodology

The benchmarking code measures:

- **Average Latency**: Total time divided by the number of operations, representing the average time per file operation.
- **CPU Usage**: Time spent by the CPU to execute the benchmark.
- **Memory Usage**: Maximum memory consumed during the benchmark run.

The benchmarks are executed for combinations of file counts (100, 1000, 10000) and thread counts (1, 2, 4, 8, 16).

# Observations and Analysis

## Latency

- **Low Latency with Higher Threads**: As thread count increases, latency per operation decreases slightly, especially noticeable at higher file counts. For example, with 10000 files, latency reduces from 0.1772 ms (1 thread) to 0.1009 ms (16 threads). This is due to the parallel execution of file operations that improves efficiency, but with diminishing returns at higher thread counts.
- **Batch Processing**: Processing files in batches (1000 files) maintains latency at acceptable levels by preventing excessive memory usage and CPU strain, especially as the number of files grows.

## CPU Usage

- **Increased CPU Usage with More Threads**: CPU usage increases with thread count, reflecting the overhead of multithreaded operations. For example, with 16 threads and 10000 files, CPU usage is 4.46444 seconds, significantly higher than the single-threaded case (0.760306 seconds).
- **Thread Management Overhead**: As thread count rises, managing multiple threads and synchronization (mutex locks) adds CPU overhead, which affects overall CPU usage. Beyond 8 threads, the marginal benefit of adding more threads decreases.

## Memory Usage

- **Linear Growth with File Count**: Memory usage increases roughly linearly with the number of files, as each file occupies space in memory. For instance, with 100 files, memory usage is around 4MB, while with 10000 files, it rises to approximately 11MB.

- **Slight Increase with More Threads**: Memory usage is slightly higher with additional threads due to the stack memory allocated for each thread. However, the impact is relatively modest compared to the file count's effect on memory.

## Summary of Results

| No of Threads | No of Files | Avg Latency(ms) | CPU Usage(s) | Memory Usage(KB) |
|---|---|---|---|---|
| 1 | 100 | 0.19 | 0.02181 | 4399104 |
| 1 | 1000 | 0.18 | 0.089754 | 8146944 |
| 1 | 10000 | 0.1772 | 0.760306 | 10604544 |
| 2 | 100 | 0.13 | 0.768668 | 10604544 |
| 2 | 1000 | 0.151 | 0.845295 | 10604544 |
| 2 | 10000 | 0.1445 | 1.59943 | 10948608 |
| 4 | 100 | 0.14 | 1.60825 | 10948608 |
| 4 | 1000 | 0.135 | 1.69525 | 10948608 |
| 4 | 10000 | 0.1356 | 2.58098 | 11145216 |
| 8 | 100 | 0.15 | 2.59009 | 11145216 |
| 8 | 1000 | 0.12 | 2.67514 | 11145216 |
| 8 | 10000 | 0.111 | 3.53041 | 11145216 |
| 16 | 100 | 0.17 | 3.54009 | 11145216 |
| 16 | 1000 | 0.151 | 3.65564 | 12251136 |
| 16 | 10000 | 0.1009 | 4.46444 | 12251136 |

These results demonstrate the impact of thread count and file count on performance metrics. As expected, parallelism reduces latency but with diminishing returns at higher thread counts. CPU and memory usage increase with both the number of threads and the number of files processed.

# Conclusions and Recommendations

- **Optimal Thread Count**: 4-8 threads provide a good balance between performance and resource usage for large file counts. Beyond 8 threads, benefits in latency are minimal while CPU and memory overhead increase.
- **Batch Processing for Large Filesets**: Handling files in batches of 1000 helps to maintain system stability and reduce excessive memory consumption.
- **Scalability**: MemFS is reasonably scalable for in-memory file operations with up to 10,000 files, showing low average latency. However, further increases in file count may require optimization or hardware with higher memory capacity.