B+ Tree Enhanced FAT32 Filesystem

Aprameya V. Madhwaraj Ishaq Mustafa Khan Roy Khoo

Abstract

This work provides an in-depth exploration of integrating a B+ Tree structure with the FAT32 filesystem to enhance file management efficiency and concurrency control. The proposed approach leverages the hierarchical indexing capabilities of the B+ Tree for faster file access and improved organization, addressing the inherent limitations of traditional FAT32 linear searches. Additionally, the Distributed Mutual Exclusion (DME) Token algorithm is employed to ensure efficient and conflict-free access to shared resources, enabling multi-threaded and distributed file operations. This system showcases significant improvements in storage performance, latency reduction, and scalability, making it a promising solution for modern file management challenges.

1 Problem Definition

1.1 Problem Addressed

The traditional FAT32 file system uses a linked-list structure for directory and file management. Although this approach is simple and space-efficient, it suffers from slow file search and inefficient management of large directories. This project addresses these limitations by integrating a B+Tree structure into FAT32, enhancing the performance of file lookups, insertions, and deletions.

1.2 Importance of Problem

Efficient file management is vital for modern systems due to the increasing file sizes and complexities of the directory. Optimizing FAT32 using B+Tree seeks to address some of these concerns through:

1. Reducing file search time significantly
2. Enhance scalability for larger file systems
3. Provide a more structured and manageable approach to file indexing
4. Exceed the limitations of 32GB partition size and 16TB hard drive from FAT32

1.3 Relevance of FAT32

Despite being an older filesystem, FAT32 continues to be widely used in various applications and contexts due to its unique strengths:

1. **Universal Compatibility**: FAT32 is supported by almost all operating systems, including Windows, macOS, Linux, and even embedded systems like cameras and gaming consoles amongst other microcontroller uses. This makes it a suitable choice for portable storage devices like USB drives and SD cards

2. **Simplicity**: Its simple structure ensures minimal resource usage, making it suitable for devices with limited computational power. Subsequently, due to its simplicity, a smaller device can avoid overheads incur by more complex filesystems features (e.g.: NTFS with journaling, security permissions, etc)

3. **Cross-Platform Usage**: Unlike NTFS, which is optimized for Windows and has limited write support on macOS and Linux, FAT32 provides seamless interoperability.

By enhancing FAT32 with modern data structures like B+Tree, our goal is to bridge the gap between its compatibility and performance limitations, ensuring its continued relevance in an evolving technological landscape.

1.4 System Model and Assumptions

1. The filesystem operates in a single node or in a distributed environment.

2. Files are stored as flat text files for simplicity in simulation.

3. Concurrency control is handled using locks (e.g. pthread locks).

1.5 Goals

1. Replace the linear search in FAT32 with a logarithmic search using B+Tree.

2. Maintain compatibility with traditional FAT32 operations.

3. Support concurrent read/write access with Distributed Mutual Exclusion (DME).

2 Conceptual Design

2.1 System Composition

The system integrates multiple layers and components to enhance the traditional FAT32 filesystem with modern indexing and distributed access controls. It includes:

1. **B+ Tree Index Layer**:
   - Acts as a hierarchical structure for efficient search, insertion, and deletion operations.
   - Replaces the traditional linked-list directory structure in FAT32 to achieve logarithmic time complexity for directory operations.
2. **DME (Distributed Mutual Exclusion) Access Control**:
   - Implements token-based mutual exclusion for synchronized access to shared resources across distributed nodes.
   - Ensures safe and conflict-free operations in multi-process environments.
3. **Storage Layer**:
   - Manages space allocation using bitmap-based directory management, optimizing storage efficiency and reducing fragmentation.
   - Supports up to 64K directory entries, with a uniform page-caching mechanism to minimize I/O overhead.
4. **FAT32 Management Layer**:
   - Handles file creation, updates, deletions, and storage allocation using the FAT32 framework.
   - Provides metadata storage and cluster management for efficient data retrieval.

2.2 Component Functionalities

1. **B+ Tree Index Layer**:
   - Supports key operations like search, insertion, update, and deletion with dynamic rebalancing to maintain optimal performance.
   - Uses a dense B+ Tree variant for faster disk I/O and scalability, suitable for large datasets and frequent directory modifications.
2. **DME (Distributed Mutual Exclusion) Access Control**:

- Coordinates distributed operations using a centralized token system, ensuring only one process accesses a critical resource at a time.
- Uses lightweight algorithms to manage token queues and reduce overhead in multi-node systems.

3. **Storage Layer**:
   - Employs multi-level bitmap management for efficient space allocation and tracking of occupied/free sectors.
   - Implements a partner allocation algorithm to improve space utilization and access patterns.
4. **FAT32 Management Layer**:
   - Provides core FAT32 functionalities such as cluster allocation, directory entry management, and file metadata handling.
   - Enhances traditional FAT32 with mechanisms for managing large directories and supporting extended file operations.

2.3 Major Principles

- **Efficiency:**
  The B+ Tree integration ensures logarithmic time complexity for directory operations, minimizing delays during file access and modifications.
- **Scalability:**
  The system supports large datasets, distributed access, and multiple concurrent processes without compromising performance.
- **Compatibility:**
  Combines the simplicity and universal support of FAT32 with advanced indexing and synchronization features, ensuring ease of adoption.
- **Reliability:**
  Includes redundant FAT copies for error handling, synchronized access control via DME, and dynamic tree balancing to maintain operational integrity.
- **Optimization:**
  Reduces I/O costs through bitmap-based management and uniform page caching while maintaining minimal overhead for metadata storage.

3 Implementation Description

The code for this project was completely written by our group, drawing on concepts and inspirations from existing research and repositories. While the conference paper by Zhao et al. [1] proposed the idea of enhancing the FAT file system with advanced design elements, it did not include any implementation. Additionally, the repositories [2, 3] served as inspiration for writing our FAT32 management script, particularly for understanding basic FAT file system operations.

We expanded on these foundational ideas by adding unique features, including Distributed Mutual Exclusion (DME) for synchronized access in distributed environments and bitmap-based directory management to optimize storage allocation and retrieval.

3.1 Development Platform and Tools

The system was implemented and tested on a robust development environment to ensure compatibility and performance under realistic conditions. The platform details include:

- **Virtualization Platform:** Oracle VM VirtualBox.
- **Operating System:** Ubuntu 20.04, providing a reliable Linux-based environment.
- **Hardware Configuration:**
  - Memory: 4 GB RAM.
  - Processor: 2 CPU cores, 4.0 GHz.
  - Storage: 10 GB Virtual Disk.
- **Programming Language:** C, for its performance and low-level control over memory and file operations.
- **Libraries and Frameworks:**
  - POSIX threads for multithreading and synchronization.
  - Standard C libraries for memory management, I/O, and string operations.
- **Version Control:** Git for tracking changes and collaborative development.

3.2 Major Data Structures

The implementation relied on the following core data structures:

1. **B+ Tree:**
   - `BPTreeNode`: Represents nodes of the B+ Tree. Each node contains keys, pointers to children or leaf values, and metadata like the number of keys and whether it is a leaf.

- `BPTree`: Encapsulates the entire B+ Tree, including the root node and associated bitmap for tracking node space usage.

2. **FAT32:**

   - `FAT32_FileSystem`: Represents the FAT32 filesystem, including the FAT table, data region, and metadata like the number of sectors and clusters.
   - `FAT32_Entry`: Represents a single file or directory entry in the FAT32 system, storing metadata such as filename, size, and starting cluster.

3. **Distributed System:**

   - `DistributedNode`: Represents a node in the distributed system. Includes metadata like node ID, access token status, and a priority queue for task management.
   - `PriorityQueue`: Implements a heap-based priority queue for managing tasks based on timestamps.
   - `Task`: Encapsulates a distributed operation (e.g., insert, search, delete) with attributes like operation type, process ID, and timestamp.

4. **Bitmap:**

   - Used in both the B+ Tree and FAT32 layers for efficient space management. Each bit in the bitmap indicates whether a corresponding block or entry is occupied.

3.3 Major Methods

1. **B+ Tree Operations:**

   - `insertNonFull`: Dynamically splits nodes and balances the tree when a node exceeds its capacity. Implements right-biased splitting for uniform distribution.
   - `search`: Traverses from the root to the appropriate leaf node using keys for guidance. Operates in logarithmic time.
   - `delete`: Handles underflows using redistribution or merging to maintain tree balance.
   - `update`: Searches for a key and modifies its associated value. Reorganizes the tree if the key structure changes.

2. **FAT32 Operations:**

   - `allocate_clusters`: Allocates clusters for a new file based on the size. Links clusters using the FAT table.
   - `free_clusters`: Deallocates and clears linked clusters for deleted files.
   - `fat32_write`: Writes data to allocated clusters, handling overflow across clusters.

- `fat32_read`: Reads file data from the clusters sequentially.
- `fat32_delete`: Deletes a file entry and deallocates associated clusters.

3. **Distributed Mutual Exclusion (DME):**
   - `requestToken`: Ensures synchronized access to shared resources by waiting for the token.
   - `releaseToken`: Passes the token to the next node after completing a critical operation.
   - `enqueue`: Adds tasks to a priority queue with timestamp-based ordering.
   - `dequeue`: Retrieves the highest-priority task for execution.
4. **Performance Testing:**
   - **Sequential and Random Access:** Measured the performance of file access patterns to compare sequential and random operations.
   - **Bulk Insertions:** Evaluated the system's efficiency in handling a large number of files simultaneously.
   - **Large File Handling:** Tested the system's ability to manage large files and allocate contiguous storage efficiently.

4 User Guide

4.1 Installation

To set up the system and prepare it for execution, follow these steps:

1. Ensure that the required dependencies are installed:
   - GCC or a compatible C compiler.
   - POSIX-compliant operating system (e.g., Ubuntu 20.04).
   - `make` build automation tool.
2. Clone the project repository or obtain the source files: Project Repository
3. Navigate to the project directory and build the program using:

   ```
   make
   ```

4. Run the program using:

   ```
   make run
   ```

5. To clean up generated files after execution:

```
make clean
```

4.2 Usage

Once the program is running, users can execute different operations by entering the following commands:

- `'b'`: Execute bulk tests on file samples of sizes {100, 200, 500, 1000, 2000}. Randomly generated files will be tested for insertion, access, and large file handling in the B+Tree file system.
- `'c'`: Perform basic CRUD (Create, Read, Update, Delete) operations on the filesystem to validate functionality.
- `'d'`: Simulate distributed mutual exclusion using three nodes (threads) accessing the B+Tree system.
- `'s'`: Perform sequential and random access tests. (*Note: Currently bugged and commented out.*)
- `'quit'`: Exit the program.

Example Execution Workflow:

```
$ make
$ make run
Enter commands (type 'exit' to quit)
Commands: b
Commands: c
Commands: d
Commands: quit
```

5 Self Evaluation

5.1 Design Goals Accomplished

1. **Performance Improvements with B+ Tree Integration:**
    - The system showed improvements in file management performance by replacing traditional linear FAT32 search with a logarithmic B+Tree-based structure. As observed in Figure 1
2. **Adaptability to Varying Parameters:**
    - The experiments confirmed that the system adapts well to different configurations, including varying numbers of files and B+Tree parameters like maximum leaf size
    - **Results:** Larger leaf sizes reduced overhead in operations such as bulk insert and large file handling while maintaining stable random access performance as shown in Figures 3a, 3b, 3c, 3d and 4b
3. **Concurrency Management with DME:**
    - The token-based Distributed Mutual Exclusion (DME) algorithm successfully enabled synchronized access to the shared B+Tree and FAT32 filesystem across multiple nodes.
4. **Scalability:**
    - The system scaled efficiently across datasets ranging from 100 to 2000 files, maintaining predictable performance trends and demonstrating suitability for large-scale file systems. As is shown in Figures 2a, 2b and 4a

We successfully implemented the core mechanism of a token-based Distributed Mutual Exclusion (DME) algorithm to manage concurrent access to a B+Tree-enhanced FAT32 file system. Our primary goal of integrating B+Tree indexing to improve file system performance was achieved, allowing efficient bulk inserts, random access, and large file handling. However, due to time constraints, we were unable to compare our DME implementation with other algorithms or extensively test its scalability under various conditions. While there is room for improvement, particularly in optimizing token passing and handling node failures, we delivered the key functionalities we set out to achieve, laying a strong foundation for future enhancements.

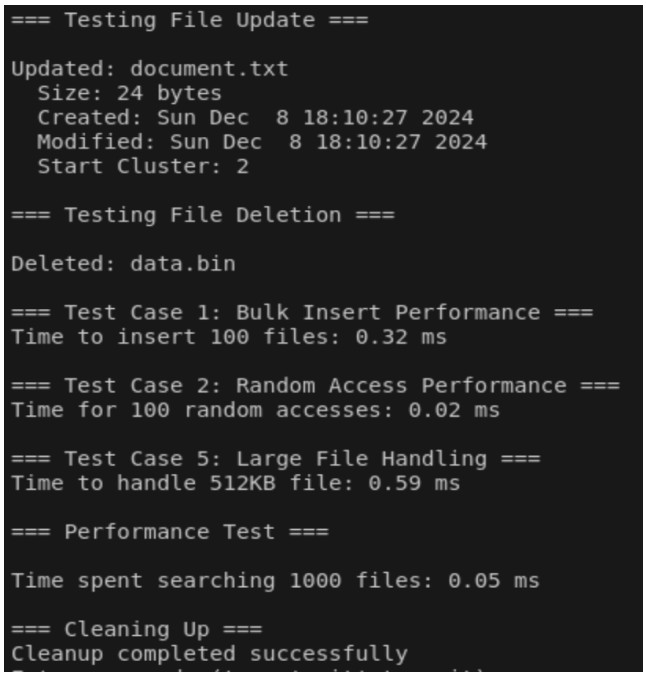
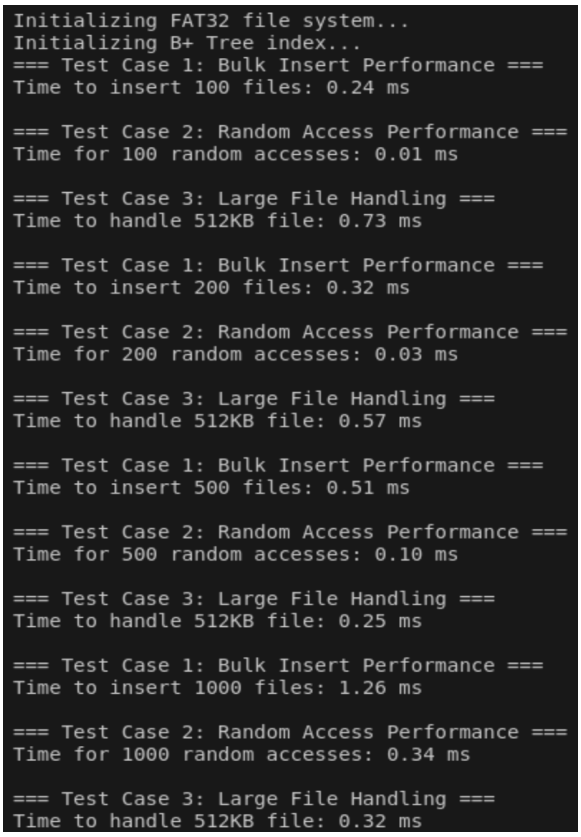
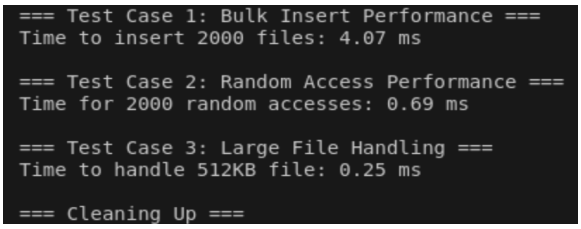


Figure 1: CRUD Test on B+Tree File System

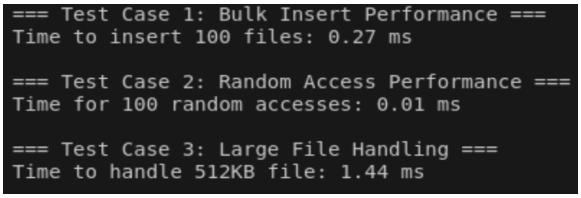


(a) Console output: Bulk Insert, Random Access, Large File Insert.

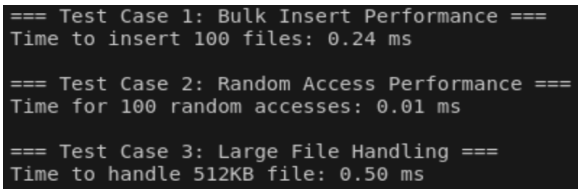


(b) Console output for further performance metrics.

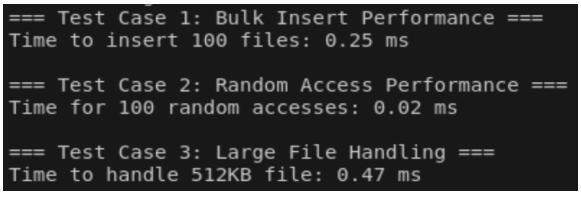
Figure 2: Console results for performance of different tests with varying file counts.



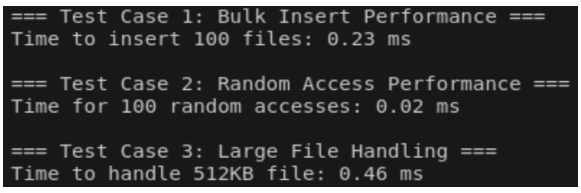
(a) Result for max leaf size = 4.



(b) Result for max leaf size = 10.

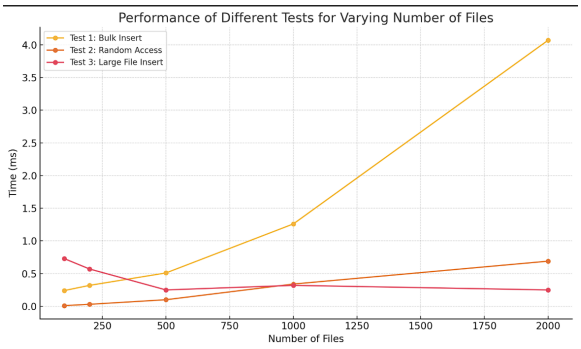


(c) Result for max leaf size = 25.

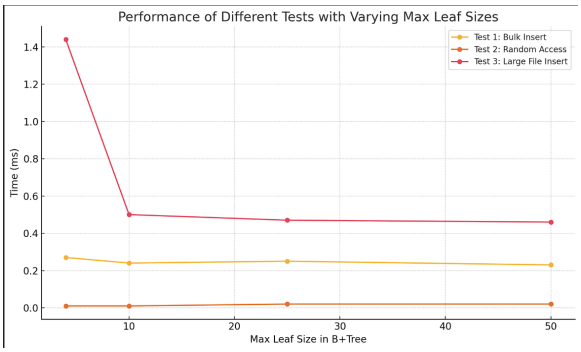


(d) Result for max leaf size = 50.

Figure 3: Performance results for varying maximum leaf sizes.



(a) Performance with varying file counts.



(b) Performance with varying maximum leaf sizes.

Figure 4: Comparison of performance across different experiments.

5.2 Limitations

While the project successfully integrates a B+ Tree structure with a FAT32 file system and introduces features like Distributed Mutual Exclusion (DME) and bitmap-based directory management, the following limitations were observed:

- **File Size Limitation:** The system inherits the FAT32 constraint of a maximum file size of 4 GB, which limits its applicability for larger datasets.
- **Partition Size Limitation:** The FAT32 file system restricts partitions to a maximum size of 2 TB, making it unsuitable for modern large-scale storage systems.
- **Concurrency Overhead:** While DME ensures safe access in distributed environments, it introduces latency and overhead, particularly under high contention.
- **Security Limitations:** The system does not include advanced permission management or encryption, relying on the basic security features of FAT32.

5.3 Future Work

The following improvements and extensions are identified as potential areas for future development:

- **Sparse Tree Implementation:** A sparse tree structure can be implemented with modifications to the key-handling mechanisms in the B+ Tree, allowing for better optimization of storage in large datasets, like sorting based on date or file size.
- **Extending File and Partition Limits:** Transitioning to an extended FAT32 (exFAT) structure could overcome the size constraints and allow for support of larger files and partitions.
- **Enhanced Security Features:** Adding user-level permission management and encryption capabilities would improve security for sensitive data.

- **Caching and Pre-fetching:** Implementing caching strategies for frequently accessed files could reduce access times and improve performance in high-demand environments.
- **Branch Based Intention Locking:** Due to overhead and inefficiency of leaf node locking we have opted to use a generalized lock for the whole tree. Improvements can be made to extends locking to be 2 process based lock where an 'intention' lock can be requested leading from the root to the intended node and then a regular read/write lock, which can allow for more concurrency and less overhead than a leaf based approach.

References

1. H. Zhao, X. Li, L. Chang, and X. Zang, "Fat File System Design and Research," *2015 International Conference on Network and Information Systems for Computers*, Wuhan, China, 2015, pp. 568-571, doi: 10.1109/ICNISC.2015.114.

2. ThinFAT32 Repository on GitHub.

3. FAT32 Implementation by StrawberryHacker on GitHub.

4. ReiserFS on Wikipedia.