LLFS Report

1. Introduction

The **Little Log File System (LLFS)** project aims to provide a simplified yet functional implementation of a file system to facilitate understanding of core concepts such as disk management, metadata handling, and crash recovery. File systems are fundamental to modern computing, responsible for organizing, storing, and retrieving data efficiently while ensuring robustness against failures. LLFS is designed to simulate these operations within a controlled environment, making it a valuable educational tool.

The primary goal of LLFS is to implement a persistent file system that supports basic operations such as creating, writing, reading, and deleting files. The system ensures that its core data structures, such as the free block vector, inode table, and directory structure, are stored on disk and remain consistent even in the event of a system crash. Additionally, LLFS integrates a crash recovery mechanism to validate and restore these structures, ensuring reliability.

This report describes the design, implementation, and experimentation of LLFS, focusing on its modular architecture and robust data structures. While the current implementation achieves basic functionality, it also highlights challenges such as limited file size support and the absence of advanced features like journaling and subdirectories. These limitations serve as a basis for future enhancements and experimentation.

2. Design and Architecture

System Overview

LLFS is built on a modular architecture that separates key responsibilities into distinct components. Each component interacts with a simulated disk, represented by a file on the host system, to perform specific operations. The primary components of LLFS include:

1. **DiskManager**: Handles low-level disk I/O, reading and writing data blocks.
2. **FreeBlockManager**: Manages the allocation and deallocation of disk blocks using a bitmap.
3. **InodeManager**: Maintains inodes, which store metadata for files and directories.
4. **DirectoryManager**: Provides a mapping between file names and their corresponding inodes.
5. **CrashRecovery**: Ensures consistency by validating and repairing on-disk data structures after a crash.

These components collectively manage the lifecycle of files and directories, ensuring persistence, consistency, and efficient disk usage.

Core Data Structures

Superblock

- **Location**: Block 0
- **Purpose**: Stores metadata about the file system, including:
    - Magic number (“LLFS”) to identify the file system.

    - Total number of blocks.
    - Number of inodes.
- **Persistence**: Updated only during formatting or major structural changes.

Free Block Vector

- **Location**: Block 1
- **Purpose**: A bitmap where each bit represents the allocation status of a block:
    - 1 indicates the block is free.
    - 0 indicates the block is allocated.
- **Persistence**: Updated on disk whenever blocks are allocated or freed.

Inode Table

- **Location**: Blocks 2 onward.
- **Purpose**: Stores inodes, which contain metadata for files and directories:
    - File type (file or directory).
    - File size.
    - Direct block pointers (up to 10).
- **Persistence**: Updated whenever a file or directory is created, modified, or deleted.

Directory Entries

- **Purpose**: Maps file names to inode IDs within a directory.
- **Persistence**: Stored in the blocks pointed to by directory inodes.

Key Components

1. DiskManager

- Provides low-level operations to read and write data blocks.
- Abstracts the details of the simulated disk, allowing higher-level components to interact seamlessly.

2. FreeBlockManager

- Tracks free and allocated blocks using the free block vector.
- Provides methods to allocate the next available block and mark blocks as free.
- Ensures consistency by persisting updates to the free block vector on disk.

3. InodeManager

- Manages the allocation, deallocation, and metadata updates of inodes.
- Provides functionality to retrieve and modify inode data.
- Ensures that inode updates are reflected both in memory and on disk.

4. DirectoryManager

- Handles directory operations, such as adding or removing entries.
- Manages the mapping between file names and inode IDs.

- Provides methods to list files within a directory.

5. CrashRecovery

- Validates key data structures (superblock, free block vector, inode table) during startup.
- Detects and repairs inconsistencies, such as orphaned blocks or invalid inodes.
- Ensures the system can recover from unexpected crashes without losing data integrity.

Flow of Operations

1. **File Creation**:
    - Allocate an inode for the file using InodeManager.
    - Allocate blocks for file data using FreeBlockManager.
    - Update the directory entry in DirectoryManager.
2. **File Writing**:
    - Split data into block-sized chunks.
    - Allocate blocks for the data and write each chunk using DiskManager.
    - Update the inode with block pointers and file size.
3. **File Reading**:
    - Retrieve the inode for the file.
    - Read blocks referenced by the inode.
    - Assemble the data into a contiguous buffer.
4. **File Deletion**:
    - Remove the directory entry.
    - Mark the inode as free.
    - Mark the associated blocks as free in the free block vector.
5. **Crash Recovery**:
    - Validate the superblock to ensure file system identification.
    - Rebuild the free block vector by scanning block usage.
    - Validate inodes and ensure all allocated blocks are accounted for.

3. Experiments and Results

Tests for Each Class

DiskManager

1. Test reading and writing individual blocks.
2. Validate disk formatting initializes all blocks to zero.
3. Check disk persistence by reading back written data after simulated restarts.

FreeBlockManager

1. Test allocation of free blocks.

2. Validate freeing blocks updates the free block vector correctly.
3. Simulate out-of-space scenarios and verify appropriate error handling.

InodeManager

1. Test allocation of inodes.
2. Validate inode updates and metadata consistency.
3. Ensure inodes are marked free after deletion.

DirectoryManager

1. Add entries to the directory and verify retrieval.
2. Remove directory entries and check for proper updates.
3. Test listing directory contents.

CrashRecovery

1. Simulate crashes during file writes and validate recovery.
2. Test recovery of orphaned blocks.
3. Validate consistency of free block vector and inode table after recovery.

LLFS Core

1. Functional tests for file creation, reading, writing, and deletion.
2. Stress tests for creating and managing large numbers of files.
3. Performance benchmarks for read and write operations.

LLFS Benchmark

1. Functional tests for file creation, reading, writing, and deletion.
2. Measure the time taken for writing and reading large files (e.g., 1 MB) over multiple iterations.

```
(base) tianyu@Tianyus-MBP Little-Log-File-System % ./build/LLFS_Benchmark
Formatting disk...
Superblock written with:
  Magic number: LLFS
  Total blocks: 4096
  Number of inodes: 512
Free block vector initialized.
Root directory initialized with inode 0.
Running functional test...
Added entry: testfile.txt to path: /
Functional test passed!
Added entry: largefile.txt to path: /
Running write benchmark...
Write benchmark for 10 iterations completed in 0.00160563 seconds.
Running read benchmark...
Read benchmark for 10 iterations completed in 0.00160483 seconds.
```

Average write time per iteration:

Average Write Time = Total Time / Iterations =189 microseconds

Throughput (assuming the file size is maxFileSize, which is 5120 bytes):

Throughput = Total Data Written / Total Time

= 5120 bytes × 10 / 0.00188779 s

≈ 27.1 MB/s

Average read time per iteration:

Average Read Time = Total TimeIterations = 235 microseconds

Throughput (assuming the file size is maxFileSize, which is 5120 bytes):

Throughput = Total Data Read / Total Time

= 5120 bytes × 10 / 0.00234767 s

≈ 21.8 MB/s

Hexdump -C

```
00000000  4c 4c 46 53 00 10 00 00  00 02 00 00 00 00 00 00  |LLFS............|
00000010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000200  1f e7 7f ff ff ff ff ff  ff ff ff ff ff ff ff ff  |................|
00000210  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |................|
*
00000400  00 00 00 00 02 00 00 00  00 00 00 00 00 00 00 00  |................|
00000410  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000a00  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41  |AAAAAAAAAAAAAAAA|
*
00000c00  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00001400  48 65 6c 6c 6f 2c 20 4c  4c 46 53 21 00 00 00 00  |Hello, LLFS!....|
00001410  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00001600  41 41 41 41 41 41 41 41  41 41 41 41 41 41 41 41  |AAAAAAAAAAAAAAAA|
*
```

00200000

Decoded Information:

1. **Block 0 - superblock**
    1. 4c 4c 46 53: ASCII for LLFS (magic number), correctly identifies the file system.
    2. 00 10 00 00: Total blocks = 0x00100000 (hex) = 4096 blocks (matches your LLFS setup).
    3. 00 02 00 00: Total inodes = 0x00000200 (hex) = 512 inodes.
    4. The rest of the block is filled with zeros, indicating unused metadata space.

2. **Block 1 - free block vector**
    1. Each bit in the free block vector represents whether a block is free (1) or allocated (0)
    2. 1f (hex) = 00011111 (binary):
    3. 000 indicates Blocks 0, 1, and 2 are allocated (reserved for superblock, free block vector, and inode)
    4. 11111: blocks 3 - 7 are free
    5. e7 (hex) = 11100111
    6. Blocks 10 → Allocated (0).
    7. Blocks 8, 9, 11–15 → Free (1).
    8. 7f (hex) = 01111111
    9. Block 22 → Allocated (0).
    10. Blocks 16–21, 23 → Free (1).
    11. The remaining bits (ff ff ...) indicate that all other blocks are free.

3. **Block 2 - first inode**
    1. The inode structure likely includes:
    2. 00 00 00 00: File type (e.g., directory, file, etc.), possibly unused.
    3. 02 00 00 00: File size = 0x00000002 (hex) = 2 bytes.
    4. Remaining Zeros: Pointers to direct/indirect blocks, currently unused.

4. **File data block**
    1. 41 (hex) = ASCII A.
    2. This block is filled with the character A and likely corresponds to the file largefile.txt written during the benchmark.
    3. This is the data for a file created during the write benchmark (largefile.txt).
    4. Since the file fits within direct blocks, its content occupies consecutive blocks

5. The file spans multiple blocks.

5. Another file

1. 48 65 6c 6c 6f 2c 20 4c 4c 46 53 21:
2. ASCII for Hello, LLFS!.
3. Interpretation:
    1. This block contains the data for testfile.txt created during the functional test.
    2. The content matches what was written (Hello, LLFS!).

4. Limitations and Future Work

Limitations

- **File Size**: Limited to 10 blocks per file (due to direct block pointers).
- **Directory Structure**: Only supports a flat directory (root only).
- **Journaling**: No journaling mechanism for crash resilience.
- **Disk Size**: Fixed during initialization.

Future Work

- Support for larger files using indirect block pointers.
- Hierarchical directory structures.
- Journaling for atomic updates and better crash recovery.
- Dynamic disk resizing.
- Performance optimization through caching and deferred updates.

5. Reference

Arpaci-Dusseau, R. H., & Arpaci-Dusseau, A. C. (2023). *Operating Systems: Three Easy Pieces* (Version 1.10). Arpaci-Dusseau Books. Retrieved from https://pages.cs.wisc.edu/~remzi/OSTEP/

Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). *Operating System Concepts* (10th ed.). Wiley. Chapter 14: File-System Implementation. Retrieved from https://codex.cs.yale.edu/avi/os-book/OS10-global/bib-dir/14.pdf

GeeksforGeeks. (2023, July 20). *File System Implementation in Operating System*. Retrieved from https://www.geeksforgeeks.org/file-system-implementation-in-operating-system/

6. Github Repository

https://github.com/Tianyu-Fang/Little-Log-File-System