

Report of Project 2025

Group members: Qihong Chen, Yulin Fang

1. Design Choices and Implementation Summary

Our OuicheFS file system adopts a hybrid storage mechanism to handle both small and large files efficiently. Files smaller than or equal to 128 bytes are stored as slices packed into blocks, while larger files are stored using a traditional block indexed format. This approach minimizes space waste for small files while maintaining compatibility with conventional file storage methods.

We designed a slice bitmap structure to track per block slice usage, and encoded slice metadata directly into the `index_block` field. A linked list of partially filled sliced blocks (`s_free_sliced_blocks`) enables efficient reuse of space. For larger files, our implementation dynamically falls back to block-based storage using an index block, and transitions are handled automatically.

In addition, we extended OuicheFS with kernel level instrumentation through `sysfs`, enabling users to inspect real time statistics about the file system. Debugging support was enhanced through the implementation of a custom `ioctl` command that prints the contents of each slice in a sliced block. We carefully reimplemented the read and write logic to comply with VFS semantics while operating outside the page cache.

2. Implemented Functionalities

2.1 Custom Read/Write Logic: We fully reimplemented the read and write paths (`ouichefs_read`, `ouichefs_write`) to bypass the Linux page cache for small files. Instead, we rely on direct block access using `sb_bread`, and synchronize changes with `mark_buffer_dirty` and `sync_dirty_buffer`.

2.2 Slice-Based Storage for Small Files: Files smaller than or equal to 128 bytes are stored as a single slice within a 4KB sliced block. Each sliced block has 32 slices (0 reserved for metadata, 1–31 for file data). Slice availability is tracked via a 32-bit bitmap in the block metadata.

2.3 Multi-Slice Allocation for Medium Files: Files between 129B and 4KB are stored using multiple consecutive slices. We search partially used blocks to locate free slices, and if none are available, allocate a new sliced block. This ensures that moderate-sized files avoid the overhead of full block allocation.

2.4 Slice Metadata Encoding in Inodes: The location of a sliced file is encoded directly into the 32-bit `index_block` field of the inode using the format: $\text{index_block} = (\text{slice_no} \ll 27) \mid \text{block_no}$. This allows the filesystem to identify both the block and the offset within the block using a single field.

2.5 Dynamic Migration to Block Storage: When a file originally stored in slices grows beyond 128 bytes, it is automatically migrated to block-based storage using the `convert_slice_to_block()` function. The original slices are copied into a newly allocated data block, the slice is released, and the inode's index is updated.

2.6 Slice Reclamation on File Deletion: When a small file is deleted, we use `release_slice()` to update the slice bitmap, potentially freeing the entire sliced block if no other slices are in use. Fully freed blocks are returned to the general free block pool.

2.7 Sysfs Interface for Monitoring Internal State: The superblock metadata is exposed to user space via a kobject under `/sys/fs/ouichefs/<device>/`. We implemented read only attributes for metrics such as `free_blocks`, `used_blocks`, `sliced_blocks`, `total_free_slices`, `efficiency`, `total_data_size`, and `small_files`. This provided invaluable insights during debugging and testing.

2.8 ioctl Interface for Slice Visualization: We implemented a custom `ioctl` command that prints the content of all 32 slices in a given sliced block to the kernel log. This was especially useful for verifying that slice writes and reads were working correctly.

2.9 Index Block Allocation and Indirection Support for Large Files: For files that exceed the slice threshold, we allocate a dedicated index block (struct `ouichefs_file_index_block`) which points to up to 1024 data blocks. This mechanism allows us to support files up to 4MB, as specified in `OUCHEFS_MAX_FILESIZE`.

2.10 Block and Inode Allocation Using Bitmaps: Free inodes and blocks are tracked using bitmap based allocators defined in `bitmap.h`. The functions `get_free_block`, `put_block`, `get_free_inode`, and `put_inode` are used consistently across file creation and deletion paths.

3. Partial or Missing Features

3.1 Bonus Task 1.11: Our current implementation does not include a defragmentation algorithm for slice-based storage. As files are added and deleted over time, external fragmentation may increase due to partially filled sliced blocks. Implementing a background defragmentation routine (triggered automatically or via `ioctl`) could improve storage efficiency by consolidating active slices into fewer blocks.

4. Conclusion and Experimental Insights

Through this project, we transformed a baseline educational file system into a space-efficient hybrid storage system with kernel instrumentation and block-level debugging support. The most intellectually rewarding aspect was designing and implementing the

slice allocation model, which is compact, efficient, and directly integrated into inode metadata.

The final outcome is a stable and functional file system that supports read/write operations across a range of file sizes, optimizes storage for small files, and provides tools for introspection and debugging. The experience solidified our understanding of filesystem internals