Project 4 Report: RU File System (RUFS)

- Kelvin Ihezue (NetID: ki120)
- Bryan Shangguan (NetID: bys8)

Benchmark and Performance Analysis

Total Blocks Used

- Static Overhead (System Metadata):
    - Superblock: 1 Block (Block 0).
    - Inode Bitmap: 1 Block (Block 1).
    - Data Bitmap: 1 Block (Block 2).
    - Inode Region: Calculated as (MAX_INUM * sizeof(struct inode)) / BLOCK_SIZE.
        - With MAX_INUM = 1024 and sizeof(struct inode) ≈ 256 bytes, the inode region occupies roughly 64 Blocks.
    - Total Static Overhead: ~67 Blocks reserved immediately upon rufs_mkfs.
- Dynamic Usage (Benchmark Operations):
    - Root Directory: 1 Data Block (allocated during rufs_mkfs).
    - Test 1 (File Creation /file): Consumes 1 Inode (allocated from the reserved Inode Region).
    - Test 2 (File Write): The benchmark writes 16 iterations of 4096 bytes.
        - Total Data: 64KB.
        - Data Blocks Consumed: 16 Blocks.
    - Test 5 (Directory Creation /files): Consumes 1 Inode and 1 Data Block (for the new directory's entries).
    - Total Dynamic Data Blocks: 1 (Root) + 16 (File Data) + 1 (Dir Data) = 18 Blocks.
- Total Estimated Block Usage: 85 Blocks (67 Static + 18 Dynamic).

Implementation Strategy

Our implementation of RUFS follows a modular design centered around the FUSE API handlers in rufs.c and the block layer in block.c.

- Superblock & Initialization (rufs_mkfs, rufs_init):
    - We implemented rufs_mkfs to partition the "virtual disk" (flat file) into four distinct regions: Superblock, Bitmaps, Inode Table, and Data Blocks.
    - Critical validation occurs in rufs_init, where we check for the existence of DISKFILE. If missing, we format the disk; otherwise, we load the superblock into the global sb structure.
- Inode Operations (readi, writei):

  - We implemented a mapping function that converts a logical inode number (ino) into a physical disk block and offset.
  - readi reads the specific block containing the requested inode into a buffer and extracts the specific inode struct.
  - writei performs a read-modify-write operation to update a single inode without corrupting neighboring inodes in the same block.
- Bitmap Management:
  - We utilized the provided set_bitmap, unset_bitmap, and get_bitmap macros.
  - get_avail_ino and get_avail_blkno scan their respective blocks linearly to find the first zero bit, marking it as used immediately to prevent race conditions during allocation.
- Directory Operations (dir_find, dir_add):
  - Lookup: dir_find iterates through the direct pointers of a directory inode. It reads each data block and casts it to an array of struct dirent, checking valid flags and comparing filenames.
  - Insertion: dir_add scans for an invalid (empty) dirent slot. If a block is full, it dynamically allocates a new block via get_avail_blkno and updates the directory inode's size and direct_ptr.
- File Operations (rufs_read, rufs_write):
  - Read: Handles offsets by calculating the starting block index (offset / BLOCK_SIZE) and the intra-block offset (offset % BLOCK_SIZE). It copies data chunk-by-chunk into the user buffer.
  - Write: Similar to read, but handles dynamic allocation. If a write targets a block index that is currently 0 in the inode, we allocate a new block on the fly. We also handle file size updates in the inode metadata.

Difficulties and Issues Faced

Superblock Initialization Logic (Resolved)

- Issue: We initially encountered a critical bug where rufs_mkfs failed to create a valid root directory, resulting in "No such file or directory" errors during mounting.
- Cause: We were calling writei(0, &root) *before* initializing the global sb variable. The writei function relies on sb.i_start_blk to locate the inode region. Because sb was zeroed out (global default), writei wrote the root inode to Block 0, overwriting our Superblock and leaving the actual inode region empty.
- Solution: We reordered the code in rufs_mkfs to assign sb = newsb *before* calling writei. This ensured writei had the correct offsets.

Disk File Permissions (Resolved)

- Issue: We faced a disk_open failed: Permission denied error when restarting the program.
- Cause: The DISKFILE was created in a previous session (possibly with different user privileges or flags), and dev_open attempted to open it with O_RDWR which failed.
- Solution: We added a step to our workflow to explicitly rm DISKFILE before re-running the file system, ensuring rufs_init calls rufs_mkfs to create a fresh disk with the current user's ownership.

Unresolved Issues

- None.
  - We successfully resolved the critical logic bug in rufs_mkfs regarding the global superblock initialization.
  - We resolved the writei offset calculation errors.
  - The permission errors were resolved by cleaning the environment.
  - The benchmark cases (File Create, Small Write, Read, Directory Create) all pass successfully with the current codebase.
- Potential Future Improvements (If time permitted):
  - Indirect Pointers: Currently, the file system is limited to 16 * 4KB = 64KB per file because we only implemented direct pointers. Implementing indirect pointers would allow for much larger files.
  - Directory Entry Deletion: We did not implement rufs_rmdir or rufs_unlink. While not required for the assignment, this means the file system grows monotonically and cannot reclaim space.