Project 4 report.

Manan Patel, Huzaif Mansuri
Mp1885, htm23

int get_avail_ino()
This function searches for an unused inode in the inode bitmap, marking it as used when found. This bitmap is loaded during system initialization and isn't immediately updated on disk; updates occur when the system is shut down or destroyed. If an available inode is found, the function returns its index. Otherwise, it returns an error if all inodes are in use, indicating the file system's inode capacity is full.

int get_avail_blkno()
This function locates and marks an available data block. It checks the data block bitmap, loaded at initialization, for an unused block (bit set to 0). When it finds one, it sets the corresponding bit to indicate usage. This bitmap update is saved to disk during system shutdown or destruction. The function returns the index of the allocated block or an error if the system reaches its maximum data block capacity.

int readi(uint16_t ino, struct inode *inode), int writei(uint16_t ino, struct inode *inode)

The readi and writei functions are essential for inode operations. readi reads an inode from the disk: it calculates the block number and offset for the inode, reads the block into a buffer, and copies the inode data into the provided inode structure. writei performs the reverse operation: it locates the inode's block number and offset, reads the block, updates the inode data in the buffer, and writes it back to the disk. Both functions are implemented to maintain the integrity and consistency of inodes, which are crucial for file system metadata.

int dir_find(uint16_t ino, const char *fname, size_t name_len, struct dirent *dirent)

This function is designed to locate a specific file or directory within a given directory. It starts by retrieving the inode of the target directory using its inode number. The function then iterates over the directory's data blocks, both direct and indirect, examining each directory entry. These entries contain names and inode numbers of files or subdirectories. The function compares each entry's name with the desired filename, ensuring an exact match in both name and length. If a match is found, the entry's details are copied to the given dirent structure, and the function updates the directory inode's access time to reflect this activity. In cases where the directory's size necessitates the use of indirect pointers, the function also traverses these pointers to find the entry. If no matching entry is found in any of the blocks, the function returns an error. This functionality helps navigate the directory structure in our RUFS, enabling the system to retrieve specific file(s) or directory information efficiently.

int dir_add(struct inode dir_inode, uint16_t f_ino, const char *fname, size_t name_len)

This function is designed to add a new directory entry to a specified directory. It begins by checking if the directory entry, identified by its name (fname), already exists in the parent directory, represented by its inode (dir_inode). This is done to prevent duplicate entries. If the entry already exists, the function terminates with an error. If the entry does not exist, dir_add proceeds to create a new directory entry, assigning it the provided file inode number (f_ino), name (fname), and name length (name_len). The function then identifies the appropriate data block in the parent directory's data blocks to store this new entry. This involves calculating the block number and offset to add the new directory entry based on the current size of the parent directory and the size of a directory entry. This calculation helps to add the new directory entry to the end of all the directory entries currently in the parent directory.

If the current directory's data blocks are insufficient to accommodate the new entry, the function allocates additional blocks. This allocation differentiates between direct and indirect blocks. For direct blocks, it directly assigns a new block number. For indirect blocks, it involves an additional layer of indirection, requiring the function to read and modify the indirect block pointers. Once the correct block is identified or allocated, the new directory entry is written to it. The function then updates the directory inode's metadata, including its size, access time, and modification time to reflect the addition of the new entry. These updates are crucial for maintaining the consistency and integrity of the file system's metadata. Finally, the updated directory inode is written back to disk, concluding the process of adding a new directory entry in RUFS. This function manages the file and directory structure within the file system, ensuring that new entries are correctly and efficiently added to the directories.

int dir_remove(struct inode dir_inode, const char *fname, size_t name_len)

This function is a critical component for managing directory contents, specifically for removing directory entries. The process begins with the function scanning through the directory inode (dir_inode), reading its data blocks to find a directory entry that matches the specified file name (fname). This search covers both direct and indirect blocks of the directory, ensuring a thorough check across the directory's entire span.
Upon locating the entry, dir_remove proceeds to eliminate it. If the target entry is the only one in its block, the function deallocates that block to optimize space usage, adjusting the inode's size and the bitmap to reflect this change. However, if there are multiple entries in the block, it shifts the last entry in the directory to the position of the removed entry, maintaining a contiguous sequence of entries. This shifting is a critical step to avoid fragmentation within the directory structure.

In cases where the directory utilizes indirect blocks, the function also takes care of updating or deallocating these blocks as necessary. This includes managing indirect block pointers and ensuring that any blocks that become empty due to the entry's removal are properly handled.

Finally, the function updates the directory inode's metadata, specifically the access time, to reflect the changes made. It then writes these updates back to the disk, ensuring that the directory's state on the disk is synchronized with its state in memory.
Throughout the operation, dir_remove incorporates comprehensive error-checking codes in cases where the specified entry is not found, or other issues arise.

int get_node_by_path(const char *path, uint16_t ino, struct inode *inode)

This function is key for translating a file path into its corresponding inode. It starts by checking if the given path is the root directory ('/'), returning the root inode. For non-root paths, it iteratively processes each segment of the path. Starting from the beginning of the path, the function separates each directory or file name (denoted by /) and uses dir_find to locate the corresponding directory entry within the current directory. If a segment is not found, the function exits with a failure status, signaling that the path is invalid. Otherwise, it continues to the next segment, updating the current inode number (‘ino’) to that of the found directory or file. This process repeats until the function reaches the end of the path. Once the final segment is resolved, get_node_by_path retrieves the inode of the target file or directory and updates the passed inode structure. This function fundamentally helps navigate the file system structure in RUFS, enabling the system to access files and directories based on their paths.

int rufs_mkfs()

This function helps to initialize the file system. It begins by creating the disk file and then opens it for setup. The function allocates memory for the superblock and a data block, configuring the super_block with essential details like the inode and data block bitmap locations, maximum numbers of inodes and data blocks, and their starting points. This superblock is then written to the disk's first block.

Subsequently, the inode bitmap is initialized, setting all inodes as free and written to the disk. The data block bitmap undergoes a similar initialization process, marking all data blocks as available. The function also prepares the root directory inode, assigning key attributes like inode number, size, type, permissions, and timestamps. It writes this root inode to the disk. The process successfully helps to layout the fundamental structure of the RUFS file system on the disk, enabling further operations within the file system.

static void *rufs_init(struct fuse_conn_info *conn), static void rufs_destroy(void *userdata)

These rufs_init and rufs_destroy functions are important to the file system, handling its initialization and cleanup. rufs_init starts by checking if the disk file exists. If not found, it calls rufs_mkfs to create the file system. If the disk file exists, it initializes in-memory data structures, including allocating memory for the superblock and bitmaps and then reads them from the disk. This includes reading the superblock, inode bitmap, and data block bitmap, ensuring that the file

system has all the necessary information in memory to operate. It also allocates memory for buffer data blocks, which are needed to operate the file system functions.

On the other hand, rufs_destroy is responsible for finalizing the file system state when it is unmounted. It writes the superblock, inode bitmap, and data block bitmap back to the disk, ensuring that all in-memory changes are persisted. Afterward, it frees the allocated memory for these structures and closes the disk file using dev_close. This function ensures that the file system is cleanly closed, with all data structures correctly saved and memory resources released.

static int rufs_getattr(const char *path, struct stat *stbuf)

This function is a key component for retrieving file or directory attributes. Upon invocation, it attempts to resolve the inode associated with the provided path using get_node_by_path. If the inode is successfully retrieved, the function populates the struct stat buffer (stbuf) with attributes from the inode. These attributes include the file mode (st_mode), which determines the file type and permissions, the number of hard links (st_nlink), the size of the file (st_size), the user and group IDs (st_uid and st_gid), and timestamps for last access (st_atime) and modification (st_mtime). It sets the mode for directories to indicate a directory and defaults the number of hard links to two. Contrarily, for regular files, it sets the mode to indicate a regular file. The function returns 0 on successful completion or an error code if the path resolution fails, ensuring that file system operations have the necessary information about files and directories.

static int rufs_opendir(), static int rufs_readdir()

The rufs_opendir and rufs_readdir functions are for directory operations. rufs_opendir begins by attempting to retrieve the inode associated with the provided path. If the inode is successfully found, the directory is considered open, and the function returns success. However, if the inode is not found, indicating that the path does not exist or is not accessible, the function returns a failure status (i.e. return -1). rufs_readdir takes this a step further by reading the contents of a directory. After successfully retrieving the directory's inode using the provided path, it iterates through the directory's data blocks, both direct and indirect. For each block, the function reads the directory entries (represented by struct dirent) and uses a filler function to transfer these entries into a buffer. This process includes copying each entry's name into the buffer. If the filler function cannot add more entries due to memory constraints, the function returns an error. Otherwise, it continues until all entries in the directory are read. This function helps implement the crucial functionality of listing directory contents, enabling users to navigate and interact with files and directories in the file system.

int dir_base_split(const char *path, char *dir_name, char *base_name)

This function is designed to split a given file path into its directory and base names. It first determines the length of the base name by counting characters from the end of the path until the first slash ('/') is encountered. This base name is then extracted and null-terminated. For the

directory name, the function checks if the path is more than just the root directory ('/'). If so, it copies the portion of the path that precedes the base name, null-terminating it appropriately. If the path is just the root directory, it sets the directory name to '/.' This function helps distinguish between a file or directory's name and its parent directory path, which is crucial for the implementation of various file system operations.

static int rufs_mkdir(const char *path, mode_t mode)

This function is responsible for creating a new directory. It begins by splitting the given path into the parent directory path and the target directory name using dir_base_split. The function then retrieves the inode of the parent directory. If the parent directory doesn't exist or the target directory already exists within the parent, it exits with an error. Upon successful validation, rufs_mkdir obtains an available inode number using get_avail_ino. It then calls dir_add to add a directory entry for the new parent directory. After that, the function initializes a new inode for the target directory, setting attributes such as inode number, size, type (including the specified mode), and timestamps. It also initializes direct and indirect pointers to -1, indicating no allocated data blocks yet. Finally, the new inode is written to disk using writei, and the function frees the allocated memory for the path names before returning success. This function helps implement the mkdir feature of the file system, allowing users to create new directories and subdirectories to organize files into new directories.

static int rufs_rmdir(const char *path)

This function handles the removal of directories. It starts by splitting the provided path into the parent directory and the target directory name. The function then retrieves the inode of the target directory. If the target directory is not found or is not empty, it exits with an appropriate error. After validating the target directory's existence and emptiness, the function clears the corresponding inode bitmap entry, marking the inode as free. It then fetches the inode of the parent directory. The next step involves removing the target directory's entry from its parent directory using dir_remove. Upon successful removal, the function frees the memory allocated for the path names and returns success. If any step fails, it exits with a failure status. This function helps implement the rmdir feature of the file system, allowing users to remove directories that are no longer needed.

static int rufs_create(const char *path, mode_t mode, struct fuse_file_info *fi)

This function is designed to create new files within the file system. It starts by separating the provided file path into the parent directory and the base name of the target file. The function then retrieves the parent directory's inode; if this directory is not found, it exits with an error. Next, it checks if a file with the target name already exists within the parent directory using dir_find. If the file exists, it exits, indicating a failure due to the file's existence.
If the file does not exist, rufs_create allocates a new inode for the file using get_avail_ino. It then adds a directory entry for the new file in the parent directory using dir_add. Following this, it

initializes the new file's inode with appropriate attributes, including the inode number, size, type (set as a regular file with the specified mode), user and group IDs, and timestamps. Finally, the new inode is written to disk with writei, and the function frees the memory allocated for path processing before returning success. This function is essential for creating new files in the RUFS file system, ensuring that files are properly registered and initialized.

static int rufs_open(), static int rufs_read()

The rufs_open and rufs_read functions are crucial for file operations. rufs_open simply checks if the file specified by the path exists by retrieving its inode using get_node_by_path. If the file does not exist, it returns an error; otherwise, it signals a successful opening.

rufs_read, build on top of it. It first retrieves the inode for the file specified in the path. The function then reads data from the file's data blocks into the provided buffer, starting from the specified offset and continuing up to the size requested. It handles both direct and indirect pointers to data blocks, ensuring that it reads from the correct blocks as per the file's inode structure. The function updates the inode's access and modification times after reading and writes these changes back to disk. Finally, rufs_read returns the total number of bytes read into the buffer. This process helps implement reading file contents, facilitating file access, and data retrieval in the file system.

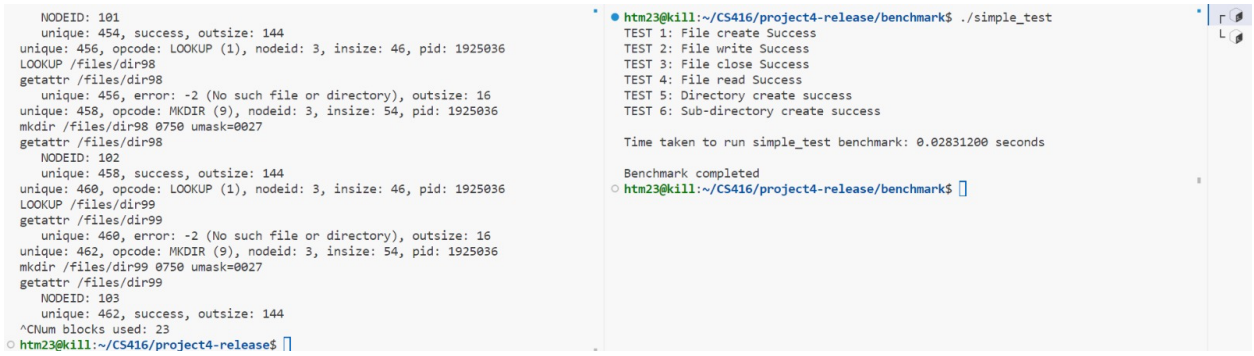
static int rufs_write( )

This function is tasked with writing data to a specified file. It begins by retrieving the inode of the file using its path. If the inode is not found, an error is returned. The function then writes data from the provided buffer into the file's data blocks, starting from a specified offset and continuing up to the given size. This involves handling both direct and indirect pointers within the inode. For direct pointers, the function reads the corresponding block into a buffer, writes the data from the user-provided buffer into this block buffer, and then writes the updated block back to the disk. If a direct pointer is not yet allocated, a new block number is obtained and assigned. For indirect pointers, the function first allocates indirect pointers if they are not already set. It then proceeds similarly to the direct pointers case, reading and updating data blocks. After writing all the data, the function updates the inode's access and modification times and its size to reflect the changes. It then writes these inode updates back to disk. In the end, the function returns the size of the data written, ensuring that the file content is correctly updated on the disk. Debug messages are printed at key stages if debugging is enabled, assisting in tracking the function's execution. This writing process is crucial for file modification and content updates in the file system.

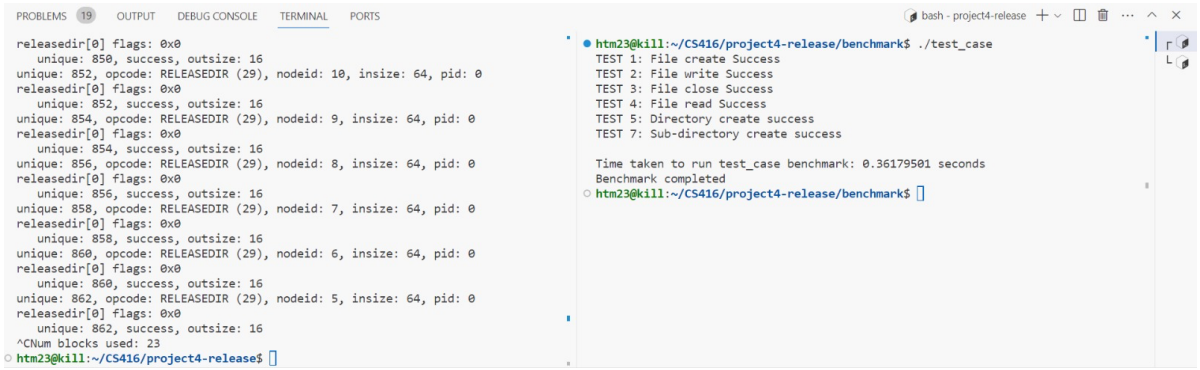
static int rufs_unlink(const char *path)

This function manages the removal of files from the file system. It begins by separating the provided file path into the parent directory path and the target file name. The function then fetches the inode of the target file; if not found, it exits with an error. Once the file's inode is

obtained, the function clears the data block bitmap entries for all direct and indirect pointers in the file's inode, marking these blocks as free.
After freeing the data blocks associated with the file, the function clears the inode bitmap entry for the file's inode, effectively freeing the inode. After that, it retrieves the inode of the parent directory and uses dir_remove to remove the directory entry corresponding to the target file in the parent directory. The function concludes by freeing the memory allocated for the path processing and returning success. If any step fails, it exits with a failure status. This function helps implement the rm feature of the file system, allowing users to remove files that are no longer needed.k



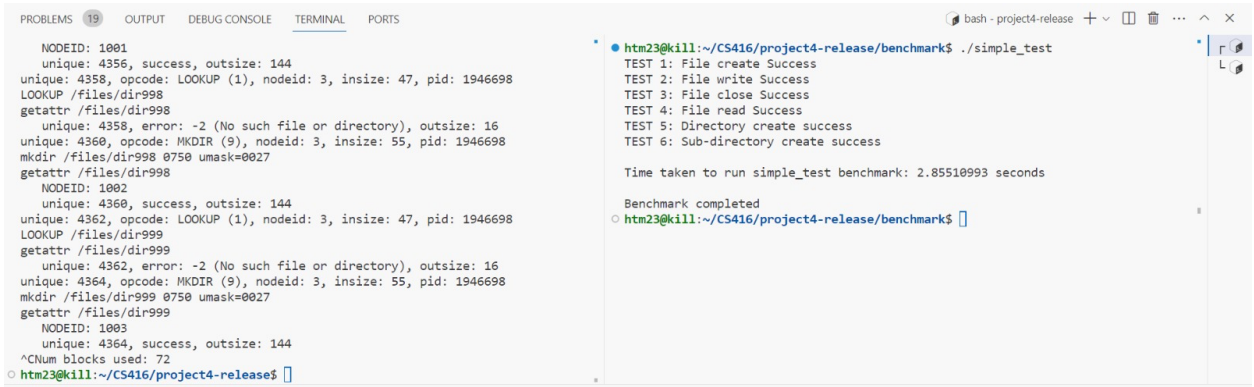
Note - Printing Total Blocks used excluding the superblock Block, data_bitmap Block, and inode_bitmap Block. The
Total time to run the simple test case is 0.0283 seconds.
The total number of blocks used are 23



Total time to run the test case is 0.361 seconds.
The total number of blocks used are 23

NFILES = 1000, ITERS = 160, testcase

Total time to run the test case is 5.58 seconds.
The total number of blocks used are 72



NUM_FILES = 1000, ITERS = 160, for simple_test

Total time to run the test case is 2.85seconds.
The total number of blocks used are 72

Possible issues in your code (if any).
No

. Collaboration and References: State clearly all people and external resources (including on the Internet) that you consulted. What was the nature of your collaboration or usage of these resources?

Understood concepts from lecture slides and textbook. Consulted TA's, which was really helpful