Partner names: Vignesh Venkat, Divit Shetty
Partner netids: vvv11, dps190

Implementation of the code:

int get_avail_ino():

Firstly, we instantiate a buffer of size 4096 B (i.e. BLOCK_SIZE) to temporarily hold the inode bitmap. Then we read the inode bitmap from the disk block to get the number of inodes that are available. When an available inode is found, the bit in the bitmap is set to 1 by using set_bitmap and then the updated bitmap is written back to the disk, to reflect the changes. When our function successfully allocates an inode, it returns the inode number!

int get_avail_blkno():

Firstly, we instantiate a buffer of size BLOCK_SIZE to temporarily hold the data block bitmap. We then read the bitmap from the disk block, which allows us to know how many data blocks are available. We then read the bitmap into the buffer, and then our function iterates through all the bits to locate the first available data block. Once we find an available bit, we set it to 1 and then we write it back to the disk to ensure the changes are reflected. If a data block is successfully allocated, the function returns the block number.

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

Our function reads an inode from the disk and copies its contents to the provided inode structure. We check if the given inode number is within bounds by comparing it against a maximum number of inodes. If the inode number is invalid, the function logs and error message and returns -1. For all the valid inode numbers, the function calculates the block number containing the inode using this starting block of the inode table and determines the specific offset of the inode within the block. A buffer of BLOCK_SIZE is used to read the block containing the inode. If the read operation is successful, the function extracts the inode's data from the offset within the block and copies it into the provided inode structure using *memcpy*.

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

Our function writes the contents of a provided inode structure to the disk at the specified inode number. Our function first validates that the inode is within bounds by checking against the maximum allowable inodes. If the inode number is invalid, an error message is logged, and the function is -1. For all the valid inode numbers, the function calculates the block number where the inode is located using the starting block of the inode table and determines the specific offset of the inode within the block. It then uses a buffer of BLOCK_SZIE to read the block containing the inode using bio_read. Once the block is read, the function copies the inode data into the buffer and the appropriate offset using memcpy. Finally, the modified buffer is written back to the disk. If all ops were successful, the function returns 0.

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

Our function searches for a directory entry using a specific name in the directory represented by the inode number. We begin by using readi to load the inode of the specified directory into a local structure. Once the inode is loaded, the function iterates over the direct pointers of the inode, which point back to the data blocks of the directory. For each valid direct pointer, we read the corresponding data block into the buffer of BLOCK_SIZE using bio_read. The data block contains directory entries, and each directory entry is checked to see if it is valid and if the name matches the fname. If a match is found, the function copies the directory entry into the provided dirent and returns 0.

int dir_add(uint16_t ino, uint16_t f_ino, const char *fname, size_t name_len):
Our dir_add function adds a new directory entry to the specified directory inode. We begin by checking if a directory entry with the same fname already exists. If a duplicate is found using dir_find, the function returns -1. However, for non-duplicate entries, we prepare a new dirent structure with the provided inode number, validity flag, name, and name length. The function then iterates through the direct pointers in the directory's inode, attempting the find a free slot in the allocated blocks. For each valid block, we read the block into a buffer, examine the dirents, and locate a free slot. If a free slot is found, the new dirent is added, and the updated block is written back to the disk. Additionally, the inode of the directory is updated to reflect the change. If no free slots exist in the current blocks, the function allocates a new data block, and we update the directory inode to reference the new block. When we add successfully, the function returns 0.

int get_node_by_path(const char *path, uint16_t ino, struct inode *inode):
Our function is designed to navigate through a hierarchical file system and retrieve the inode corresponding to a given file or directory path. Starting from the root directory, we resolve the path by splitting it into components using strtok. If the path is the root, we directly load the inode using readi. For non-root paths, we iteratively traverse each component, checking the current directory's entries to find a match using dir_find. When a matching entry is found, its inode number is used to load the next level's inode using readi, effectively stepping through the directory hierarchy. Once the entire path is successfully traversed, we copy the resolved inode into the provided inode structure and return 0. This function provides a reliable way to resolve paths and locate inodes, ensuring accurate file or directory lookup.

int rufs_mkfs():
Our function is responsible for creating and initializing the file system on the specified disk file. It begins by calling dev_init to initialize the disk file and prepare it for use. Next, we set up the superblock by populating its fields, such as the magic number maximum number of inodes, maximum number of data blocks, and block indices for the inode bitmap, data block bitmap, and the starting blocks for inodes and data blocks. The superblock is then written to block 0 of the disks. Following the superblock setup, we initialize the inode bitmap and data block bitmap by allocating memory for them, ensuring they are empty, and writing them to their designated blocks. For the root directory, we initialize a new inode, setting its type to S_IFDIR, marking it as valid, and configuring its default properties, such

as size and link count. We then update the inode bitmap to mark the root inode as used, write the updated bitmap to disk, and store the root inode in the inode table. Finally, the function ensures all allocated resources are freed, and the newly created file system is ready for operation. This process lays the foundation for the file system, ensuring the disk is correctly formatted with all essential structures in place.

static void *rufs_init(struct fuse_conn_info *conn):
Our function initializes the file system when it is mounted. We begin by attempting to open the disk file specified by diskfile_path. If the disk file does not exist, it calls rufs_mkfs to create and format a new file system. If this process fails, our function returns NULL, indicating an initialization failure. If the disk file is successfully opened, the function proceeds to read the superblock from block 0 of the disk into a buffer. The superblock data is then copied into the global sb structure for in-memory use. We validate the superblock by checking its magic number to ensure the disk contains a valid file system. If the magic number is incorrect or the read operation fails, the function returns NULL. The function then reads the data block bitmap from the disk to ensure the file system's state is properly loaded into memory. If any errors occur during these operations, the function returns NULL. Otherwise, the initialization completes successfully, preparing the file system for use.

struct void rufs_destroy(void *userdata)
Our function is responsible for gracefully shutting down the file system by cleaning up resources and resetting on-disk structures. We begin by iterating through all inodes, reading each inode's block from disk, and checking if it is valid. For any valid inode, we verify if it has an indirect pointer in use. If so, we read the block referenced by the indirect pointer, clear its contents, and write the empty block back to disk. This ensures all indirect blocks are properly deallocated. Next, we handle the metadata bitmaps. We read the data block bitmap and inode bitmap from disk and reset their contents to zero, effectively marking all blocks and inodes as unused. These updates are then written back to disk. Finally, we close the disk file using dev_close, releasing any resources associated with it. This method ensures the file system is left in a clean state, ready for subsequent usage or reinitialization.

static int rufs_gettattr(const char *path, stuct stat *stbuf):
Our function retrieves the attributes of a file or directory specified by its path and populates a struct stat structure with these details. To begin, we use the get_node_by_path function to resolve the given path and locate the corresponding inode. If the path cannot be resolved or the inode cannot be found, the function returns -1 to indicate an error. Once the inode is successfully retrieved, we clear the stbuf structure and populate its fields using the information stored in the inode. This includes setting the inode number, file type and permissions, link count, file size, and block allocation details. We also set the user and group IDs, to match the current user, assign the block size used by the file system, and calculate the number of allocated blocks based on the file size. Lastly, the modification time is set to the current time.

static int rufs_opendir(const char *path, struct fuse_file_info *fi):
Our function is responsible for opening a directory at a specified path. It begins by calling get_node_by_path to resolve the path and locate the corresponding inode. Once the inode is retrieved, the function checks whether the inode represents a directory by verifying its type against the S_IFDIR flag. If both checks succeed, the directory is considered successfully opened, and the function returns 0. This ensures that only valid directories can be opened, maintaining the integrity of directory operations in the file system.

static int rufs_readdir(const char *path, void *buffer, fuse_fill_dir_t filler, off_t offset, struct fuse_file_info *fi):
Our function is designed to read the contents of a directory specified by its path and fill a buffer with the names of its entries. We begin by locating the inode corresponding to the given directory path. If the inode cannot be found or the path does not resolve to a directory, the function returns an error. Once the directory inode is verified, we use the filler function to add the special entries current directory and parent directory to the buffer. The function then iterates through the direct pointers of the directory's inode, which reference the data blocks containing directory entries. For each valid block, the data is read into a buffer using bio_read. The buffer is interpreted as an array of dirent structures, and we examine each entry to check if it is valid. If an entry is valid, its name is added to the buffer using the filler function. By the end of the iteration, the buffer contains the names of all valid entries in the directory.

static int rufs_mkdir(const char *path, mode_t mode):
Our function is responsible for creating a new directory at a specified path within the file system. It starts by separating the given path into the parent directory's path and the new directory's name using dirname and basename. We then locate the inode of the parent directory. If the parent directory does not exist or is not a valid directory, the function returns an error. Once the parent directory is verified, we allocate a new inode for the directory. If no inode is available, the function returns an error. Next, we use dir_add to add an entry for the new directory to the parent directory. If this operation is successful, we proceed to initialize the new directory’s inode with its attributes, such as its inode number, type, size, and link count. The link count is set to 2 to account for the special "." and ".." entries. All direct and indirect pointers are set to 0 as the directory is initially empty. Finally, the new directory inode is written to disk using writei. If all steps succeed, the function returns 0, indicating the directory was created successfully.

static int rufs_create(const char *path, mode_t mode, struct fuse_file_info *fi):
Our function handles the creation of a new regular file within the file system at the specified path. We begin by breaking down the given path into the parent directory's path and the new file's name using dirname and basename. The function then locates the inode of the parent directory. If the parent directory does not exist or is not a valid directory, the function returns an error. After validating the parent directory, we allocate a new inode for the file. We then add an entry for the new file to the parent directory using dir_add. Once

the directory entry is successfully added, we proceed to initialize the inode for the new file. This includes setting its inode number, marking it as valid, setting its size to 0, and assigning its type as a regular file along with the specified mode. The link count is set to 1, and all direct and indirect pointers are initialized to 0. Finally, the newly created file inode is written to disk using writei. If all steps are executed successfully, the function returns 0, indicating the file was created successfully.

static int rufs_open(const char *path, struct fuse_file_info *fi):
Our function is responsible for opening a file at the specified path within the file system. The function begins by locating the inode corresponding to the provided path. If the path cannot be resolved or the inode cannot be found, the function returns -1, indicating an error. Once the inode is successfully retrieved, the function verifies that the inode represents a regular file by checking its type against the S_IFREG flag. If the inode is not a regular file, the function also returns -1. If both checks pass, the file is successfully opened, and the function returns 0.

static int rufs_read(const char *path, char *buffer, size_t size, off_t offset, struct fuse_file_info *fi):
Our rufs_read function is responsible for reading data from a file at the specified path into a provided buffer. It begins by resolving the path to locate the corresponding inode using get_node_by_path. If the path cannot be resolved or the inode is not found, the function returns -1. Once the inode is retrieved, we ensure it represents a regular file by checking its type against the S_IFREG flag. If the inode is not a regular file, the function also returns -1. If the specified offset exceeds the file size, the function immediately returns 0, indicating that there is no data to read. Otherwise, the function calculates how much data can be read based on the file size, the specified size, and the offset. It then iterates over the direct data pointers of the file inode, reading each block into a temporary buffer using bio_read. The appropriate portion of each block is copied into the provided buffer, accounting for the offset and the number of bytes remaining to be read. This process continues until the requested data is fully read or the file's size is reached. Finally, the function returns the total number of bytes successfully read.

static int rufs_write(const char *path, const char *buffer, size_t size, off_t offset, struct fuse_file_info *fi):
Our function is responsible for writing data to a file at the specified path. It begins by resolving the path by locating the file’s inode. If the path cannot be resolved or the inode is not found, the function returns -1. We then validate that the inode represents a regular file by checking its type against the S_IFREG flag. If the inode is not a regular file, the function also returns -1. The function then iteratively processes the data to be written, using the specified offset to determine where the writing starts. For each direct pointer in the inode, it calculates how much of the current block is affected and either allocates a new block or uses an existing block. The relevant block is read into a temporary buffer, updated with the new data from the input buffer, and written back to disk using bio_write. This process continues until all the requested data is written or no more space is available. After writing

the data, the function updates the file size in the inode, ensuring it reflects the largest offset written to. Finally, it writes the updated inode back to disk using writei. If all operations are successful, the function returns the total number of bytes written.

We skipped these functions:
*static int rufs_rmdir(const char *path) {return 0;} static int rufs_releasedir(const char *path, struct fuse_file_info *fi) {return 0;}*
*static int rufs_unlink(const char *path) {return 0;}*
*static int rufs_truncate(const char *path, off_t size) {return 0;}*
*static int rufs_release(const char *path, struct fuse_file_info *fi) {return 0;}*
*static int rufs_flush(const char * path, struct fuse_file_info * fi) {return 0;}*
*static int rufs_utimens(const char *path, const struct timespec tv[2]) {return 0;}*

For testing:

We created a test_filler(), initialize_test_fs(), debug_bitmap(), and clear_bitmap() to help with our testing functions. All our testing functions are in test.c. To test, please copy the designated testing function into rufs.c and uncomment the appropriate testing function in the main function. Then run:
```
(1) make clear
(2) make CFLAGS="-D_FILE_OFFSET_BITS=64 -DTEST_MODE"
(3) ./rufs
```
And you should get the output you are looking for!