Rutgers University CS416 Fall 2023 Operating Systems Project 4 Report
December 13th 2023
Nivesh Nayee: nn395
Rohith Malangi: rgm111

Implementation

Global variables:
Here we will describe some of the global variables we have used

num_of_inode_blocks: This stores the number of data blocks that we need to allocate in our disk to store inodes
block & **first_block:** These are buffers that we use throughout the code to store data blocks temporarily between reading from and writing the data blocks to the DISKFILE.

Function Descriptions

int get_avail_ino()
We read the inode bitmap block from disk. Then loop through each bit until we find one that is 0 (unallocated inode). We then update that bit to be a 1, write the updated bitmap back to the bitmap block on disk, and return the available inode number we found. If an inode wasn't found, we return -1.

int get_avail_blkno()
Similarly to get_avail_ino:
We read in the data block bitmap from disk. Then loop through the bitmap until we find a bit equal to 0. We update the bitmap to set the found bit to 1 and then write the bitmap back to the data block bitmap block on disk. We return as "starting data region + the available block" because data block bitmap is just for the data block region, not including the metadata (superblock, inode & data block bitmap, and inode region). Thus, after we get the available data block from 0-MAXDNUM, we have to return the data block by adding with the starting region of the data block, which will then give us the exact block number of the disk that is free, If a data block wasn't found, we return -1.

int readi(uint16_t ino, struct inode *inode)
We calculate the data block number that the inode will be in. we add 3 to the inode number passed in divided by the number of inodes that fit in a block because we know the 1st 3 blocks in our disk will be reserved for superblock and the 2 bitmaps. We then read in the block from disk that contains the requested inode and get the byte offset into the block. We copy the inode from the offset into the pointer passed in.

int writei(uint16_t ino, struct inode *inode)
Same exact logic as readi except last step we write the updated inode block to our disk.

int dir_find(uint16_t ino, const char *fname, size_t name_len, struct dirent *dirent)
With the inode number passed in, we get the corresponding inode using our readi function. We calculate the number of dirents that can fit in one block.
We then loop through all the direct ptrs until the dirent is found:

- If the ptr is valid, we read the data block number it holds in from disk, and then check through all the dirents in that block to see if their names match the one passed in.

If the dirent wasn't found in the direct ptrs, we go through each indirect ptr until the dirent is found:

- If the indirect ptr is valid, we get it's data block number and read in that block from disk. That block is a block of direct ptrs. So then we loop through all the direct ptrs and use the same logic to check if the name passed in matches.

int dir_add(struct inode dir_inode, uint16_t f_ino, const char *fname, size_t name_len)
We first call dir_find to check if the name of the requested dirent already exists in the current directory. If it does, we return -1.
For Direct Ptr: We first find the **last_half_used_blk** which is done by the **total dir entries / dir entries 1 blk can have** and then **ceil** that so we get the rounded one ex. 2.3 = 3 half used blk. Then, we also get the **total_full_blk** by doing the same exact thing just exclude the ceil part. Next, we find the total entries stored in the half block (**offset**) by **total dir entries - (total_full_blk * blk_dir_entries)**(dir entries can have in 1 blk). Now, we differentiate between handling direct and indirect pointers based on the **total_full_blk** of directory entries. We have two checks, if a new block needs to be allocated or if an existing half-used block can be utilized. We differentiate this we checking if the **total_full_blk** and **last_half_used_blk** is the same or not, same means new block needs to be allocated and the **last_half_used_blk** is greater than that block is half used. Lastly, with the help of **offset**, we directly go there and add the dirent entry, update the inode time vstat, and write the block and inode to the disk.

For Indirect Ptr: Here, we first deduct the **total dir entries** by 16(direct ptr) then we do the same calculation for **last_half_used_blk, total_full_blk & offset.** We also find the appropriate index of indirect ptr and the index for inner entries. We use the same logic to find if a new block needs to be allocated or if we can utilize the half used block. For using half used block, we directly go to the appropriate indirect ptr and inner entry indexes (direct ptrs that the indirect ptrs point to) and then add the dirent entry there. Next, we update the inode and the block to the disk with the vstat time. For new block allocation, there are two case we needed to handle which are: 1. indirect ptr is allocated but need to allocate new data block to the inner entries, 2. if need to allocate both data blk for inner entries and dir_entries data blk. According to the cas, we get the free data block and assign that block/blocks then add the dirent entries with same updating the block/blocks and inode to the disk with time.

Within this whole process, if anything goes wrong then it directly returns -1(falied) or at the end returns 0(succes).

int get_node_by_path(const char *path, uint16_t ino, struct inode *inode)
We have implemented this function recursively.
Our base case is if the path is NULL, the string terminator character, or just a single '/'.
If one of these base cases is true, we call the readi to write the inode of 'ino' into the passed in inode parameter.

We use string manipulation to split the path by the '/' character and. We chop off the dirent in the path string and then pass the remaining path into a recursive function call by modifying the inode number, which will be dir_entry.ino (ino = inode number).

static void *rufs_init(struct fuse_conn_info *conn)
We open the diskfile. We initalize our superblock and bitmaps and write them to disk.

static void rufs_destroy(void *userdata)
We close the diskfile.
You can also see commented out the code we used to calculate the number of data blocks used while running the benchmarks.

static int rufs_getattr(const char *path, struct stat *stbuf)
We first check if the path is valid using the get_node_by_path function. If the path is valid, we copy in the fields from the found inode to the stat buffer. If the inode is for a directory we set the appropriate directory stats.

static int rufs_opendir(const char *path, struct fuse_file_info *fi)
Calls get_node_by_path to see if path is valid.

static int rufs_readdir(const char *path, void *buffer, fuse_fill_dir_t filler, off_t offset, struct fuse_file_info *fi)
We first check if the path is valid, using get_node_by_path function. If the path is valid:
We use similar logic as before to loop through all the direct and indirect ptrs and get their dirents. If the dirents are valid we use the filler to put them in the buffer.

static int rufs_mkdir(const char *path, mode_t mode)
We use string manipulation to split the base name and the parent directory. Then call get_node_by_path to get inode of the parent directory. We then call get_avail_ino to get an inode for the new dirent. We then call dir_add to add the new dirent for the new directory to the parent directory. We then create a new inode for the new directory. We then fill in all the fields and use writei to write the new inode to our disk.

static int rufs_create(const char *path, mode_t mode, struct fuse_file_info *fi)
Similar to rufs_mkdir we use use string manipulation to split the base name and parent directory path. Then get the inode of the parent directory. Then create a new inode for the new file with help of get_avail_ino. Then use dir_add to add a new dirent to the parent directory. Fill in the new inode fields and write it to disk.

static int rufs_open(const char *path, struct fuse_file_info *fi)
Calls get_node_by_path to check if the path is valid.

static int rufs_read(const char *path, char *buffer, size_t size, off_t offset, struct fuse_file_info *fi)
We first check if the path is valid. If it is:
We calculate the starting block to write with the help of the offset by **offset/BLKSIZE** which will give the exact blk to write with the 0-based indexing. Next, we also need to find out how many bytes to skip according to the offset given, which can be done as **offset%BLKSIZE.** Now, we simply iterate till the size > 0, We also handled the direct ptrs and indirect ptrs here by checking if the **blk_to_write,** if it is greater than 16 means indirect and if not then direct ptrs. For both case, we check if the size remaining is greater than the **BLOCK_SIZE - bytes_to_skip**, then we write as many bytes it can have in one block or else we simply just write the remaining size left. First we read the block from the disk and then copy it from the block to buffer. We add 1 for "**((char*)block + bytes_to_skip) +1**" because **bytes_to_skip** calculates the number of bytes to bypass, and the additional +1 ensures that copying begins immediately after the skipped bytes. Lastly, we increment the buffer with **bytes_to_write**, and the **blk_to_write;** and decrement the size by the **bytes_to_write.**

For indirect Ptrs, We do the same calculation with the addition of finding the indirect ptr index and inner entries index.

After all this, we simply update the time for that inode and then return the size that we read.

static int rufs_write(const char *path, const char *buffer, size_t size, off_t offset, struct fuse_file_info *fi)
Here, we do that same exact logic as read with the addition of writing the block to disk and if needed, initialize the data block, then update and write to disk. Another thing that is different is the memcpy where we swap the buffer and the block. Here we write to the block from the buffer given in the parameter. The rest is the same as rufs_read.

int dir_remove(struct inode dir_inode, const char *fname, size_t name_len)
We check through all the valid ptrs if the filename passed in match to one of its dirents.
If the one of the dirents matches the file name, we set that dirents valid bit to 0, zero out its name and length. After this, we get the last block and the offset within the last block which is stored in the list of inodeMap, this last block and offset for the very last dirent. We create the temporary dirent to store that dirent. Then we add null dirent entry to the very last dirent entry. Next, we check if the block is the same as the last block and if it is then we read that block again for updated block which we just added. Next we also check if the offset is not same then we add the temporary dirent to that removed space and if offset is same then we dont do this part. Which means that user wants to delete the last dirent. Then we need to update the last block and offset for the last dirent. Two case, if the offset is 0 that means we need to go to the last block and get the last dirent by decrementing the last block number and the setting the offset as (BLOCK_SIZE) - sizeof(struct dirent). Another case is within the case, where we just decrement by size of the dirent.

Then we add to block and update the time as well. Here the idea is basically that we are swapping the last dirent to the inbetween dirent to remove so that there will be contiguous dirent entries.
We also search through the indirect ptrs using same logic as before.

static int rufs_rmdir(const char *path)
We check if the directory to be removed has files in it. If it does, we return an error. Otherwise, we will clear the bit for the directory to be deleted in the inode bitmap. Then we get the inode of the parent directory and remove the dirent corresponding to the just deleted directory.

static int rufs_unlink(const char *path)
We calculates how many blocks of data we need to clear based on the size of the file passed in. We go through however many direct ptrs and indirect ptrs we need to free the blocks that are pointed from them. We unset the bits in the data block bitmap that correspond to the blocks that were just freed and invalidate the direct ptrs that were previously pointing to them. We then unset the bit in the inode bitmap for the inode of the file that's being deleted. We then remove the dirent in the parent directory that corresponds to the file that was just deleted.

int rufs_mkfs():

- This function will be called the first time the DISKFILE is not found. Firstly, it will initialize the DISKFILE and then assign superblock(0th block), inode & data block bitmap(1 & 2 blocks), and create root directory inode, which will be written in the starting block of inode region as 0th inode number.

- SuperBlock: This has fields such as magic num which helps to know which file system to use. Max INUM and DNUM specify the limit for the inodes and data blocks. It also specifies the block number where the bitmaps lie and the starting region of the inode and data blocks.
- Bitmap: initially, we set everything as 0 as free space. The length for bitmaps will be max dnum or inum / 8 as we read the entries as bits. Lastly, we simply bio_write it to the specifies block number to the disk.

Benchmark Results

<u>Number of blocks used:</u>
#of data blocks used for simple_test.c with N_FILES = 1000, BLOCKSIZE = 4096, and ITERS = 16 was 71 data blocks.

#of data blocks used for test_case.c with N_FILES = 100, BLOCKSIZE = 4096, and ITERS = 16 was 23 data blocks.

<u>Time taken:</u>
The time it took to run simple_test.c with N_FILES = 1000, BLOCKSIZE = 4096, and ITERS = 16 was on average 1.871 seconds on cd.cs.rutgers.edu

Time for test_case.c to run with N_FILES = 100, BLOCKSIZE = 4096, and ITERS = 16 was on average 0.312 seconds on cd.cs.rutgers.edu