Yash Patel | yp315
Kev Sharma | kks107

P4 report

```
^CTotal data blocks used: 190
```

```
yp315@ls:~/CS416/cs416_os/p4/code/benchmark$ time
 ./simple_test
TEST 1: File create Success
TEST 2: File write Success
TEST 3: File close Success
TEST 4: File read Success
TEST 5: Directory create success
TEST 6: Sub-directory create success
Benchmark completed

real    0m0.024s
user    0m0.002s
sys     0m0.000s
```

Total data block used = 190
It is because we are allocating in data bitmap the inode data blocks that will be used so when we call getavailblockno we get index of user data block to work with

So for simple test it will be 3 for superblock and two bitmaps + 64 (for inodes not all are taken) + 1 for root then 6 (blocks for /files because we have 100 sub directories and 1 block can store 19 dirents so we need 6 blocks for 100 subdirectories + . and ..) we also have another 100 is used because each subdirectory has dirent . and .. entries so they each use 1 block so it will be 3 + 64 + 1 (root) + 6 (/files) + 100 (each subdir with . and ..)

While setting them 1 on data bitmap is not needed it makes it easier for us to get_avail_block_no as it will give the index that we need to use without any other calculations and also keeps inodes together.

Code explanation

Get_avail_ino(): we use loop and call get_bitmap() until it returns zero then we set that index to 1 and return that index

get_avail_blkno(): same logic as above

readi(): This functions read the inode from disk by calculating the blk index and offset inside that block and return that inode to the pointer given

writei(): This function takes the inode given and get the block index to write and offset to write at and write to the disk.

dir_find(): given an inode of a directory we just check if there exist a dirent entry that is same name and return based on if it was successful or not

dir_add(): tries to find if the entry exists if it does return it exists else it checks where we need to put the new entry. Either we create a new block of dirents and add at 0th position or add in an existing block if we add successfully we return success else of there are no block available we return ran of of space error

get_node_by_path(): strtok to tokenize the string by "/" we start from root then we through each directory trying to find the next subdirectory to reach the path. If we can't find it, return -1 else success and put the node in the pointer given.

rufs_mkfs(): makes the superblock with the right information as we assign 1 block for superblock , 1 for inode bitmap and 1 for data bitmap. It also gets all the memory required to keep those 3. Our mkfs also allocates the bitmap for the inodes as those (64) blocks will be used for inode and the rest can be for user data like dirents and actual file data. Then we create the root directory, assign it an inode and set its data and add dirent . and .. to root and write to the disk to retain current information.

rufs_init(): We try to open a disk file if we can't open it. We call mkfs else we get the memory for metadata and get the data from disk and write to that memory.

rufs_destroy(): frees all memory that is allocated for metadata information.

rufs_getattr(): gets the vstat of the inode that we get from the path and copies that into the buffer.

rufs_opendir(): if we can get inode by path then we return success else -1

rufs_readdir(): We call node by path to get inode and read each dirent can caller filler to add to the buffer.

rufs_mkdir(): We get the basename and parent path then we create a new inode with new number and dir_add that inode to the parent. Then we set data for the inode and add . and .. dirent to the new directory that we created

rufs_create(): We get the basename and parent path then we create a new inode with a new number and dir_add to the parent. Then we set the data of the inode as a file and write to the disk

rufs_open(): we call get_node_by_path if the inode exist we return success else -1

rufs_read(): we calculate what block we start from offset/Block size and we get offset in that block from where we start then we read up to requested bytes or file size bytes and fill the buffer with the data that we read from the file.

rufs_write(): same as before with block and offset calculation then we loop until we are able to write the size of the write request or max direct pointer size and keep writing to necessary blocks as needed and write them to disk.

Optional functions were not implemented due to time constraints and we did not have any problems.