CSC369 A3 Writeup

999506850 | g3chowju | Julian Chow

ext2_rm:

In my implementation, for most of the data structures used to read the data in the image file, I used the included structs using the following include: `#include <linux/ext2_fs.h>`

From there, I went ahead and read all all my relevant data from the file to the blocks defined in the include above. My traversal algorithm is based upon a while loop and with the exit condition that I come across a file. One of the pillar stones of my program is that I used a helper function that splits the path into a string array so we can compare using strcmp when we are analyzing the data in the datablocks

Not included in the header file above, I have created my own struct to read a directory / file defined in functions.c, since data entries in datablocks always have $8 + N$ bytes of data (they are variable length because of the string names), I store the first 8 bytes in a struct and within that struct I can obtain data on how long exactly the string we're looking at is; I then can simply allocate a character array with that information and read the string into the character array to compare. **As far as I've tested, this program handles all cases.**

Traversal Algorithm

- My traversal algorithm begins after I read the root inode (assumed to be inode #2) into a inode struct
    - As the while loop repeats, we will repeatedly read different inodes into the same inode struct (this works because of pointers and inodes are always the same sze)
- The first while loop checks whether or not the current inode we are looking at is describing a file or a directory
    - $0x4000$ describes a directory and $0x8000$ describes a file
    - As long as we are looking at a directory and not a file, we keep following where the inode points to and finds the next part of the path
    - When we arrive at a data block for a directory, we automatically skip the first 2 entries that are guaranteed to be . and .. since we are not interested in it
        - This is where I use a struct that I defined myself to read the first 8 bytes of data and handle the variable length string separately using that data
    - I first check if the string length is the same as the length of the current index of the string array, then if it passes, I check if the strings are identical; if they are it means we have found the file we want
        - If the first check (string length) fails, then we move on to the next data entry
        - Here, it is tricky to go to the next data entry since the string is at the end of the data entry and they are variable length, however I noticed that data entries always start and the next set of 4 bytes so I can use the condition: `while(data->name_length % 4 != 0)` to increment to get to the next entry
        - If I come across a entry where I find that the name length is 0, it means I am looking at garbage and there is no more relevant data in that block

Exit Condition:

- If the while loop condition is not satisfied it will mean we have found a file of matching name; which leads me to a series of cleanup operations where I modify and immediately write to the file the relevant data structures, these include:
    - The inode that points to the file
        - To remove a file from the system, we remove the pointer to the data block containing the contents of the file we are interested in removing
    - The inode bitmap
        - Since we are essentially clearing and allocating a inode from the file system, we need to modify the inode bitmap to keep it consistent
    - The superblock
        - The superblock contains information on how many inodes are free, since we unallocated a inode (we have one more available / free inode now) we have to increment the counter that keeps track of free inodes

Others:

I was unable to complete the other implementation programs mostly due to time, but I have included a `skeleton.c` that I used as a foundation to craft my remove program.

Right now the `skeleton.c` program is set up to output the status of all 16 inodes in the program and output the inode number of the first free inode, to interpret this output

- If a inode # corresponds to a string of only zeroes, that inode is unallocated

Here are my plans for the other programs:

Make directory:

- I would approach this program mostly the same as my remove program with the following changes
    - Instead of looking for a file for my while loop, I would be looking for a match with the string in my last index of my string path array (splitted)
        - Once I traversed to that directory, I would use the helper function `find_free_inode` and seek to that inode.
            - Also need to find a data block in a similar manner
        - Next I would make a new inode and fill in relevant data
        - Next, modify the `inode → i_blocks` array to point to the next free data block
        - In the free data block
            - Write 24 bytes to account for the . and ..
                - First one should have a reference to it's inode
                - Second should have a reference to the root (inode #2)
            - Fill in other relevant data and write them to the file

Copy

- Mostly the same traversal algorithm as Make directory would implement
    - Once I arrived at the directory in the same manner as `mkdir`
        - I would go to the next free data entry (I would do this by checking if the next data entry has length 0)
            - Find a free inode and make this data entry reference it (with my `struct`)
            - Fill in the size of the file (I know this value using simple file mechanisms like `fgetc` on a target file)
            - Fill in the `name_length`
            - Write `name_length` bytes of the filename to the data entry
        - Next I have to visit the new inode I just made the data entry reference
            - Now I need a new datablock
            - Make the inode point to that datablock and go to that block
                - Fill in the contents of the targetfile here
    - Now all that's left is the general cleanup business
        - Update superblock, inode_bitmap, datablock_bitmap etc.
            - The bitmaps will be updated using bitwise operators then writing them back into the file