Assignment 4: File Systems

Due Dec 3 by 10pm **Points** 12

(Do not leave this assignment until the last few days, or you will NOT be able to complete it! Work together with your partner, do not split up the tasks since lots of common code may be reusable if you design your code together!)

Introduction

In this assignment, you will explore the implementation of the **ext2** **(https://wiki.osdev.org/Ext2)** file system, and will write tools to modify ext2-format virtual disks. To do this work, you will need to be comfortable working with binary data and will need to learn about the ext2 filesystem. The wikipedia page provides a good overview, but you will find the **ext2 wiki** **(https://wiki.osdev.org/Ext2)** and Dave Poirer's **Second Extended File System** **(http://www.nongnu.org/ext2-doc/index.html)** documentation invaluable when looking for details of the format.

You can work in pairs for this assignment. MarkUs will only create the appropriate a4 directory in your repository when you log into MarkUs and either invite a partner, or declare that you will work alone. As usual, please log into MarkUs well before the deadline to make sure you can access your repository. (Do **not** create an a4 the directory in your git repo, otherwise MarkUs won't know about it and we won't be able to see your work.)

This assignment contains some bonus features. Implementing a bonus will compensate for any possible marks lost in another section of the assignment, but cannot increase your mark more than 100%. For implementing any additional functionality which is not specified in the handout, or if you are unsure whether to handle some inode parameters in specific cases (after having read the documentation!), please ask on the discussion board.

Requirements

Your task is to write a set of programs (in C) that operate on an ext2 formatted virtual disk. The executables must be named **exactly as listed below**, and must take the specified arguments.

- ext2_mkdir <image file name> <path>

  This program takes two command line arguments. The first is the name of an ext2 formatted virtual disk. The second is an absolute path on your ext2 formatted disk. The program should work like mkdir, creating the final directory on the specified path on the disk. If any component on the path to the location where the final directory is to be created does not exist or if the spec fied directory already exists, then your program should return the appropriate error (ENOENT or EEXIST).

  Note:

- Please read the specifications to make sure you're implementing everything correctly (e.g., directory entries should be aligned to 4B, entry names are not null-terminated, etc.).
- When you allocate a new inode or data block, you *must use the next one available* from the corresponding bitmap (excluding reserved inodes, of course). Failure to do so will result in deductions, so please be careful about this requirement.
- Be careful to consider trailing slashes in paths. These will show up during testing so it's your responsibility to make your code as robust as possible by capturing corner cases.

- ext2_cp <image file name> <path to source file> <path to dest>

  The ext2_cp program takes three command line arguments. The first is the name of an ext2 formatted virtual disk. The second is the path to a file on your native operating system, and the third is an absolute path on your ext2 formatted disk. The program should work like cp, copying the file on your native file system onto the specified location on the disk. If the source file does not exist or the target is an invalid path, then your program should return the appropriate error (ENOENT). If the target is a file with the same name that already exists, you should not overwrite it (as cp would), just return EEXIST instead.

  Note:

  - Please read the specifications of ext2 carefully, some things you will not need to worry about (like permissions, gid, uid, etc.), while setting other information in the inodes may be important (e.g., i_dtime).
  - When you allocate a new inode or data block, you *must use the next one available* from the corresponding bitmap (excluding reserved inodes, of course). Failure to do so will result in deductions, so please be careful about this requirement.
  - Be careful to consider trailing slashes in paths. These will show up during testing so it's your responsibility to make your code as robust as possible by capturing corner cases.

- ext2_ln <image file name> <source path> <dest path>

  This program takes three command line arguments. The first is the name of an ext2 formatted virtual disk. The other two are absolute paths on your ext2 formatted disk. The program should work like ln, creating a link from the first specified file to the second specified path. This program should handle any exceptional circumstances, for example: if the source file does not exist (ENOENT), if the link name already exists (EEXIST), if a hardlink refers to a directory (EISDIR), etc. then your program should return the appropriate error code. Additionally, this command may take a "-s" flag, after the disk image argument. When this flag is used, your program must create a symlink instead (other arguments remain the same).

  Note:

  - For symbolic links, you will see that the specs mention that if the path is short enough, it can be stored in the inode in the space that would otherwise be occupied by block pointers - these are called fast symlinks. Do *not* implement fast symlinks, just store the path in a data block regardless of length, to keep things uniform in your implementation, and to facilitate testing. If in doubt about correct operation of links, use the ext2 specs and ask on the discussion board.

- ext2_rm <image file name> <path to link>

  This program takes two command line arguments. The first is the name of an ext2 formatted virtual disk, and the second is an absolute path to a file or link (not a directory) on that disk. The program should work like rm, removing the specified file from the disk. If the file does not exist or if it is a directory, then your program should return the appropriate error. Once again, please read the specifications of ext2 carefully, to figure out what needs to actually happen when a file or link is removed (e.g., no need to zero out data blocks, must set i_dtime in the inode, removing a directory entry need not shift the directory entries after the one being deleted, etc.).

  BONUS: Implement an additional "-r" flag (after the disk image argument), which allows removing directories as well. In this case, you will have to recursively remove all the contents of the directory specified in the last argument. If "-r" is used with a regular file or link, then it should be ignored (the ext2_rm operation should be carried out as if the flag had not been entered). If you decide to do the bonus, make sure first that your ext2_rm works, then create a new copy of it and rename it to ext2_rm_bonus.c, and implement the additional functionality in this separate source file.

- ext2_restore <image file name> <path to file>

  This program takes two command line arguments. The first is the name of an ext2 formatted virtual disk, and the second is an absolute path to a file or link (not a directory!) on that disk. The program should be the exact opposite of rm, restoring the specified file that has been previous removed. If the file does not exist (it may have been overwritten), or if it is a directory, then your program should return the appropriate error.

  - **Hint:** The file to be restored will not appear in the directory entries of the parent directory, unless you search the "gaps" left when files get removed. The directory entry structure is the key to finding out these gaps and searching for the removed file.
  - **Note:** If the directory entry for the file has not been overwritten, you will still need to make sure that the inode has not been reused, and that none of its data blocks have been reallocated. You may assume that the bitmaps are reliable indicators of such fact. If the file cannot be fully restored, your program should terminate with ENOENT, indicating that the operation was unsuccessful.
  - **Note(2):** For testing, you should focus primarily on restoring files that you've removed using your ext2_rm implementation, since ext2_restore should undo the exact changes made by ext2_rm. While there are some removed entries already present in some of the image files provided, the respective files have been removed on a non-ext2 file system, which is not doing the removal the same way that ext2 would. In ext2, when you do "rm", the inode's i_blocks do not get zeroed, and you can do full recovery, as stated in the assignment (which deals solely with ext2 images, hence why you only have to worry about this type of (simpler) recovery). In other FSs things work differently. In ext3, when you

rm a file, the data block indexes from its inode do get zeroed, so recovery is not as trivial. For example, there are some removed files in deletedfile.img, which have their blocks zero-ed out (due to how these images were created). There are also some unrecoverable entries in images like twolevel.img, largefile.img, etc. In such cases, your code should still work, but simply recover a file as an empty file (with no data blocks), or discard the entry if it is unrecoverable. However, for the most part, try to recover files that you've ext2_rm-ed yourself, to make sure that you can restore data blocks as well. We will not be testing recovery of files removed with a non-ext2 tool.

- **Note(3):** We will not try to recover files that had hardlinks at the time of removal. This is because when trying to restore a file, if its inode is already in use, there are two options: the file we're trying to restore previously had other hardlinks (and hence its inode never really got invalidated), _or_ its inode has been re-allocated to a completely new file. Since there is no way to tell between these 2 possibilities, recovery in this case should not be attempted.
- **BONUS:** Implement an additional "-r" flag (after the disk image argument), which allows restoring directories as well. In this case, you will have to recursively restore all the contents of the directory specified in the last argument. If "-r" is used with a regular file or link, then it should be ignored (the restore operation should be carried out as if the flag had not been entered). If you decide to do the bonus, make sure first that your ext2_restore works, then create a new copy of it and rename it to ext2_restore_bonus.c, and implement the additional functionality in this separate source file.

- ext2_checker <image file name>

  This program takes only one command line argument: the name of an ext2 formatted virtual disk. The program should implement a lightweight file system checker, which detects a small subset of possible file system inconsistencies and takes appropriate actions to fix them (as well as counts the number of fixes), as follows:

  a. the superblock and block group counters for free blocks and free inodes must match the number of free inodes and data blocks as indicated in the respective bitmaps. If an inconsistency is detected, the checker will trust the bitmaps and update the counters. Once such an inconsistency is fixed, your program should output the following message: "Fixed: X's Y counter was off by Z compared to the bitmap", where X stands for either "superblock" or "block group", Y is either "free blocks" or "free inodes", and Z is the difference (in absolute value). The Z values should be added to the total number of fixes.
  b. for each file, directory, or symlink, you must check if its inode's i_mode matches the directory entry file_type. If it does not, then you shall trust the inode's i_mode and fix the file_type to match. Once such an inconsistency is repaired, your program should output the following message: "Fixed: Entry type vs inode mismatch: inode [I]", where I is the inode number for the respective file system object. Each inconsistency counts towards to total number of fixes.
  c. for each file, directory or symlink, you must check that its inode is marked as allocated in the inode bitmap. If it isn't, then the inode bitmap must be updated to indicate that the inode is in use. You should also update the corresponding counters in the block group and superblock (they should be consistent with the bitmap at this point). Once such an inconsistency is repaired, your program

should output the following message: "Fixed: inode [I] not marked as in-use", where I is the inode number. Each inconsistency counts towards to total number of fixes.

d. for each file, directory, or symlink, you must check that its inode's i_dtime is set to 0. If it isn't, you must reset (to 0), to indicate that the file should not be marked for removal. Once such an inconsistency is repaired, your program should output the following message: "Fixed: valid inode marked for deletion: [I]", where I is the inode number. Each inconsistency counts towards to total number of fixes.

e. for each file, directory, or symlink, you must check that all its data blocks are allocated in the data bitmap. If any of its blocks is not allocated, you must fix this by updating the data bitmap. You should also update the corresponding counters in the block group and superblock, (they should be consistent with the bitmap at this point). Once such an inconsistency is fixed, your program should output the following message: "Fixed: D in-use data blocks not marked in data bitmap for inode: [I]", where D is the number of data blocks fixed, and I is the inode number. Each inconsistency counts towards to total number of fixes.

Your program must count all the fixed inconsistencies, and produce one last message: either "N file system inconsistencies repaired!", where N is the number of fixes made, or "No file system inconsistencies detected!".
You may limit your consistency checks to only regular files, directories and symlinks.

Hint: You might want to fix the counters based on the bitmaps, as a one-time step before attempting to fix any other type of inconsistency. Even if initially trusting the bitmaps may not be the way to go (since they could be corrupted), the counters should get readjusted in the later steps anyway, whenever the bitmaps get updated. The Z values from point a) should be added to the tally of fixes, but do not include any further superblock or block group counter adjustments from points c) and e) (since technically these may be just correcting the adjustments made in point a)).

All of these programs should be minimalist. Don't implement what isn't specified: only provide the required functionality. For example, don't implement wildcards. Also, can't delete directories? Too bad! Unless you want the bonus! :)
However, it is your responsibility to make your code as robust as possible by capturing corner cases for the functionality that you need to implement. For example, be careful to consider trailing slashes in paths (e.g., ext2_cp, ext2_mkdir, etc.). Such things will show up in our tests, so you need to test your code properly!

Some simplifying assumptions that you can make about the implementation:

- You may assume that a path will not include symlinks in the middle. However, keep in mind that a path may be invalid so you need to check every entry along the path.
- You may assume that for recovery purposes, you will not be asked to recover files with hardlinks. When attempting to restore a file, if its inode is marked as in-use in the inode bitmap, then you can safely assume that the file is not recoverable.
- If a file that gets removed happens to be the only entry in the last directory block of a directory, then you do not need to "reclaim" this directory block.

- When creating a file or directory, you should always create an entry at the end of the last block of the parent directory, or in a new block if there is no space. You should not attempt to create a new entry in a "gap" resulted from a previously removed entry.

Additionally, please be careful in following the specifications. When in doubt about how to handle a specific case, please ask on the discussion board. For example, a tricky corner case arises if a directory has more than one block and you are removing a file or directory which happens to be the first entry in a block other than the first block. In this case, you do not have "." and ".." to change the rec_len for. In order to avoid losing track of any potential subsequent entries, you should take the approach described in the specs. You should be able to infer from the specs that in this special case, your must zero the inode and adjust the rec_len to the next entry (or the end of the block). In this case, this file will become unrecoverable and that's ok.

You will find it **very** useful for these programs to share code. You will want a function that performs a path walk, for example. You will also want a function that opens a specific directory entry and writes to it.

Tools

We are giving you several tools that can be used on the teach.cs machines. They can be found in /u/csc369h/fall/pub/a4

To help you visualize your file system, we are giving you an already built tool, called ext2_ls. This program takes two command line arguments.
- The first is the name of an ext2 formatted virtual disk.
- The second is an absolute path on the ext2 formatted disk.
The program works like ls -1a (that's number one "1", not lowercase letter "L"): it prints one line for every directory entry (including "." and "..") from the directory specified by the absolute path. If the path does not exist, it prints "No such file or directory". If the path is a file or link, the tool simply prints the full path on a single line.

We are also giving you a tool called ext2_dump, which dumps all the raw information about the image contents. This is very similar to the readimage program that you have to implement for the tutorial exercises that give you practice with ext2 images. Once again, we encourage you to work on the tutorial exercises first, to gain experience with extracting various bits of information from an ext2 image.

We are also giving you a tool called ext2_corruptor, which corrupts file system images, introducing various inconsistencies like the ones that you have to fix. This tool has limited capabilities, and is solely to help you with basic testing. You are welcome to develop your own corruptor tool as well.

Finally, to help you in determine some basic correctness, we are giving you some sample test cases: running a set of commands and their expected outputs (image dumps). These self-tester test cases are found on the teaching labs under: /u/csc369h/fall/pub/a4/a4-self-test

Learning about the Filesystem

Sample virtual disk images can be found in /u/csc369h/fall/pub/a4/images:

- emptydisk: An empty virtual disk.
- onefile: A single text file has been added to emptydisk.
- deletedfile: The file from onefile has been removed.
- onedirectory: A single directory containing a text file has been added to emptydisk.
- hardlink: A hard link to the textfile in onedirectory was added.
- deleteddirectory: A recursive remove was used to remove the directory and file from onedirectory.
- twolevel: The root directory contains a directory called `level1` and a file called `afile`. `level1` contains a directory called `level2`, and `level2` contains a file called `bfile`.
- twolevel-corrupt: Same as `twolevel`, except that the image contains file system inconsistencies that you will have to repair with the checker.
- twolevel-norestore-afile: Same as `twolevel`, except that /afile has been removed, and its inode number has been reused. This image can be used for testing the case when a file cannot be restored.
- largefile: A file larger than 13KB (13440 bytes) is in the root directory. This file requires the single indirect block in the inode.

These disks were each created and formatted in the same way (on an ubuntu virtual machine):

```
% dd if=/dev/zero of=~/DISKNAME.img bs=1024 count=128
% mke2fs -N 32 DISKNAME.img
% sudo mount -o loop ~/DISKNAME.img /home/reid/mntpoint
% cd /home/reid/mntpoint
% ...... normal linux commands to add/remove files/directories/links .....
% cd ~
% umount /home/reid/mntpoint
```

Since we are creating images with mke2fs, the disks are formatted with the **ext2 file system (http://en.wikipedia.org/wiki/Ext2)**. You may wish to read about this system before doing some exploration. The **wikipedia page for ext2 (http://en.wikipedia.org/wiki/Ext2)** provides a good overview, but the **Ext2 wiki (http://wiki.osdev.org/Ext2)** and Dave Poirer's **Second Extended File System (http://www.nongnu.org/ext2-doc/index.html)** article provide more detail on how the system places data onto a disk. It's a good reference to keep on hand as you explore.

We are restricting ourselves to some simple parameters, so you can make the following assumptions when you write your code:

- A disk is 128 blocks where the block size is 1024 bytes.
- There is only one block group.
- There are 32 inodes.
- You do not have to worry about permissions or modified time fields in the inodes. You should set the type (in `i_mode`), `i_size`, `i_links_count`, `i_blocks`(disk sectors), and the `i_block` array.

We will **not** test your code on anything other than disk images that follow this specification, or on corrupted disk images.

Other tips:

- Inode and disk block numbering starts at 1 instead of 0.
- The root inode is inode number 2 (at index 1)
- The first 11 inodes are reserved.
- There is always a lost+found directory in the root directory.
- Disk sectors are 512 bytes. (This is relevant for the i_blocks field of the inode.)
- You should be able to handle directories that require more than one block.
- You should be able to handle a file that needs a single indirection
- Although you can construct your own structs from the information in the documentation above, you are welcome to use the **ext2.h** file from the exercises. I took out a bunch of components that we aren't using, but there are still quite a few fields that are irrelevant for our purposes.

However, you will probably also want to explore the disk images to get an intuitive sense of how they are structured. (The next three exercises will also help you explore the disk images and get started on the assignment.)

There are two good ways to interface with these images. The first way is to interact with it like a user by mounting the file system so that you can use standard commands (mkdir, cp, rm, ln) to interact with it. Details of how to do this are below. The second way is to interact with the disk as if it is a flat binary file. Use xxd to create hex dumps, diff to look for differences between the dumps, and your favorite text editor to view the diffs. For example (YMMV):

```
% diff <(xxd emptydisk.img) <(xxd onefile.img) > empty-onefile.diff
% vimdiff empty-onefile.diff
```

You should be able to use a combination of these techniques to understand how files are placed on disk and how they are removed. For example, you can create a new disk image, use mount to place files of various sizes on it, unmount it, and then use xxd and diff to see how the image differs from the other images you have.

Mounting a file system

If you have root access on a Linux machine (or Linux virtual machine), you can use mount to mount the disk into your file system and to peruse its contents. (Note: this requires sudo, so you will need to do this on a machine (or virtual machine) that you administer.

On the teaching labs, you can use a tool called FUSE that allows you to mount a file system at user-level (from your regular account). It may not work on an NFS mounted file system, so **this will only work on the teaching labs workstations (it will not work via ssh-ing remotely).**

Note: <UtorID> should be replaced with your own UtorID below.

```
# create a directory in /tmp and go there
mkdir -m 700 /tmp/<UtorID>-csc369h
cd /tmp/<UtorID>-csc369h
```

```
# to create your own disk image
dd if=/dev/zero of=DISKNAME.img bs=1024 count=128
/sbin/mke2fs -N 32 -F DISKNAME.img

# create a mount point and mount the image
# CWD is /tmp/<UtorID>-csc369h
mkdir mnt
fuseext2 -o rw+ DISKNAME.img mnt

# check to see if it is mounted
df -hl

# now you can use the mounted file system, for example
mkdir mnt/test

# unmount the image
fusermount -u mnt
```

You can use the same strategy to mount one of the images provided above.

Marking scheme

- ext2_cp: 12%
- ext2_mkdir: 12%
- ext2_ln: 12%
- ext2_rm: 14% (+5% bonus)
- ext2_restore: 20% (+5% bonus)
- ext2_checker: 20%
- Code style and organization: 10% - code design/organization (modularity, code readability, reasonable variable names, avoid code duplication, appropriate comments where necessary, proper indentation and spacing, etc.)
- **Negative deductions (please be careful about these!):**
  - Code does not compile -100% for *any* mistake, for example: missing source file necessary for building your code (including Makefile, provided ext2.h header, etc.), wrong Makefile target names, typos, any compilation error, etc.
  - No `plagiarism.txt` file: -100% (we will assume that your code is plagiarised, if this file is missing)
  - Warnings: -10%
  - Extra output to stdout: -20%
  - Code placed in subdirectories: -20% (only place your code directly under your A4 directory)
  - See more instructions at the end of this handout.

Submission

The assignment should be submitted to an A4 directory in your git repository. Don't forget to add and commit+push all of the code for the required programs. Please also provide a Makefile that will create your programs. Your Makefile should use -Wall, and produce no warnings. Please make sure that your Makefile includes the following **separate targets**:

- ext2_cp: compiles and produces the ext2_cp executable
- ext2_mkdir: compiles and produces the ext2_mkdir executable
- ext2_ln: compiles and produces the ext2_ln executable
- ext2_rm: compiles and produces the ext2_rm executable
- ext2_rm_bonus (optional) compiles and produces the ext2_rm_bonus executable (optional, for bonus)
- ext2_restore: compiles and produces the ext2_restore executable
- ext2_restore_bonus (optional): compiles and produces the ext2_restore_bonus executable (optional, for bonus)
- ext2_checker: compiles and produces the ext2_checker executable

Additionally, invoking make without arguments must compile all the targets.
Please consider separating the bonus parts as indicated above, in order to make it easier for us to determine if you did the bonus or not. The bonus source files and their counterparts may include large portions of the same code. This will make it easier for you as well, to make sure that at the very least the baseline implementation works, even if the bonus part may have problems or crash your baseline (non-bonus) implementation.

You may choose to submit an INFO.txt files that describes problems you've encountered, what isn't fully implemented (or doesn't work fully), any special design decisions you've taken, etc. Feel free to explain what is not implemented and describe what features you have completed. You may receive partial credit for functionality that is implemented but that does not complete one of the five required programs successfully.

Finally, whether you work individually or in pairs with a partner, you **must** submit a `plagiarism.txt` file, with the following statement:
"All members of this group reviewed all the code being submitted and have a good understanding of it. All members of this group declare that no code other than their own has been submitted. We both acknowledge that not understanding our own work will result in a zero on this assignment, and that if the code is detected to be plagiarised, severe academic penalties will be applied when the case is brought forward to the Dean of Arts and Science."

Very important notes (see also marking scheme and submission instructions above):
- Assignments **missing a Makefile** will receive a 0, as if the code did not compile!
- You must make sure that your **Makefile compiles all of your files, including possible helper files**, depending on your design, and that you have included all the necessary targets specified above.
- You must make sure that the **source files that are mandatory are named exactly as indicated** in the handout, and that the Makefile produces **executables with the same name excluding the .c extension** (for example: compiling ext2_cp.c should generate an executable named ext2_cp - do not submit the executables though).
- **Missing files due to submission mistakes** (forgot to add files, forgot to commit last version, etc.), will not be considered!
- It is your responsibility to ensure that **your code works exactly as you expect it to, on the teaching labs.**