

CSC369 Assignment 3 - File Systems

June 2, 2020

1 Introduction

In this assignment, you will explore the implementation of a particular file system, `ext2`, 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`.

This assignment contains some bonus features. While you cannot get more than 100% on this assignment, implementing a bonus will compensate for any possible marks lost in another section of the assignment.

2 Requirements

Your task is to write five programs (in C) that operate on an `ext2` formatted virtual disk. The executables must be named `ext2_ls`, `ext2_cp`, `ext2_mkdir`, `ext2_ln`, and `ext2_rm` and must take the specified arguments.

- **`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 should work like `ls -l` (that's number one "1", not lowercase letter "L"), printing each directory entry on a separate line. If the flag `"-a"` is specified (after the disk image argument), your program should also print the `.` and `..` entries. In other words, it will print one line for every directory entry in the directory specified by the absolute path. If the path does not exist, print "No such file or directory", and return an `ENOENT`. Directories passed as the second argument may end in a `"/"` - in such cases the contents of the last directory in the path (before the `"/"`) should be printed (as `ls` would do). Additionally, the path (the last argument) may be a file or link. In this case, your program should simply print the file/link name (if it exists) on a single line, and refrain from printing the `.` and `..`

- **ext2_cp:** This 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 specified file or target location does not exist, then your program should return the appropriate error (`ENOENT`). 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*).
- **ext2_mkdir:** 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 specified directory already exists, then your program should return the appropriate error (`ENOENT` or `EEXIST`). Again, 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.).
- **ext2_ln:** 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. If the source file does not exist (`ENOENT`), if the link name already exists (`EEXIST`), or if either location refers to a directory (`EISDIR`), then your program should return the appropriate error. Note that this version of `ln` only works with files. 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). If in doubt about correct operation of links, use the ext2 specs and ask on the discussion board.
- **ext2_rm:** 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(5% extra): 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.

All of these programs should be minimalist. Don't implement what isn't specified: only provide the required functionality and specified errors. (For example, don't implement wildcards. Also, can't delete directories? Too bad! Unless you want the bonus!)

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.

3 Learning about System

Here are several sample virtual disk 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.
- 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):

```
1 % dd if=/dev/zero of=~ /DISKNAME.img bs=1024 count=128
2 % mke2fs -N 32 DISKNAME.img
3 % sudo mount -o loop ~/DISKNAME.img /home/reid/mntpoint
4 % cd /home/reid/mntpoint
5 % ..... normal linux commands to add/remove files/directories/links .....
6 % cd ~
7 % umount /home/reid/mntpoint
```

Listing 1: lesson_01/Prompter.java

Since we are creating images with `mke2fs`, the disks are formatted with the ext2 file system. You may wish to read about this system before doing some exploration. The wikipedia page for ext2 provides a good overview, but the Ext2 wiki and Dave Poirer's Second Extended File System 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 that I used for the test code. 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):

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.

3.1 Mounting about File System

4 Submission