Ext Implementation Documentation

Asher Griess

2025-05-07

Abstract: Documentation regarding a customized ext implementation in python.

Keywords: Ext, Extent, Python

Contents

I How to use

1.1 Initialize Disk

After cloning the repository, you will need to use an initialization script to create the `disk.img` file. This can be run using the command `"python dbrowse.py 'diskName'` which create the disk image file.

1.2 Browse Disk

After initializing the disk, you may now use the disk browsing script, which allows you to run commands that with write and read the disk. Simply run the command `python dbrowse.py` which will bring you into the terminal application to browse the disk.

II Commands

2.1 General Features

Absolute and relative paths are supported for all commands. This means you can specify a path for a command to take to get to the desired entry. This can be from the current working directory, or it can be from the root node.

2.2 Definitions

- Entry: A item in the file system that can be either a file, directory, or other.
- Directory: A item in the file system that stores reference to other entries.
- File: A item in the file system that stores some type of data. In terms of file systems files usually mean file or directory but in this case it won't so we can be more specific. Typically, the term for this is known as a "plain file" or "regular file".

2.3 Commands

dir | ls

Print a list of all files in a given directory.

pwd

Print the absolute path to the current directory.

help

Print a list of all commands with descriptions.

cd PATH_TO_DIRECTORY

Change the current directory to the given path.

stat PATH_TO_ENTRY

Print the name of the entry, inode location, type of entry, number of links, size of entry, location of directs, and location of the indirect.

read PATH_TO_FILE

Print the contents of a given file to the command line.

write PATH_TO_FILE TEXT_TO_WRITE

Create a file at a given path containing a given text.

touch PATH_TO_FILE

Create a file at a given path.

mkdir PATH_TO_DIRECTORY

Create a directory at a given path.

rmdir PATH_TO_DIRECTORY

Remove a directory and all of its entries. If there exists more than one hard link to any directory present in the path, its' sub entries will not be added to the deletion queue.

delete PATH_TO_FILE

Remove file from directory. If there exists less than two hard links to the file, both data and inode are deleted.

copy PATH_TO_FILE PATH_TO_FILE_COPY

Create a copy of a given file using a new inode and data blocks completely independent of the original.

link PATH_TO_ENTRY PATH_TO_ENTRY_LINK

Create a new hard link to an inode and increment the inode link count. Data blocks and inode are shared with the original.

III Disk Structure

3.1 General Structure

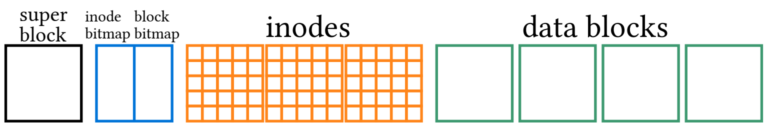

Figure 3.1: structure of disk by block.

This particular implementation uses a block size of 512 bytes. The disk consists of 2114 blocks, in which is present 1 superblock, 1 bitmap block, 64 inode blocks, and 2048 data blocks.

The architecture of this file system is loosely based on ext2, with the inclusion of some additional features which improve its overall performance. The theoretical maximum capacity is constrained by the size of the location value, which is currently 2 bytes. If it were increased to 4 bytes, the file system would be capable of storing approximately 4 terabytes of data using 1024 byte blocks.

3.2 Superblock

First block in the disk, contains disk metadata (currently only the name).

3.3 Inode Bitmap & Block Bitmap

In this implementation, the inode and block bitmaps are split across a single block. The first half is the inode bitmap, while the second half is the data bitmap. These bitmaps are used to determine if a block or inode is currently in use. If a particular block or inode is in use, the corresponding bit will be set to 1, and while not in use it will be set to 0.

3.4 Inode Structure

Name	Bytes Offset	Size Bytes	Description
Type	0	2	Type of inode file(2222) or directory(1111).
Links	2	2	Amount of current links in the file system.
Size	4	4	Current size in bytes of the file.
Direct 1	8	2	First direct to data block.
Direct 2	10	2	Second direct to data block.
Direct 3	12	2	Third direct to data block.
Indirect	14	2	Indirect to file that stores extents to other data blocks.

Table 3.1: byte structure of inode.

3.5 Indirect Extents Structure

Name	Bytes Offset	Size Bytes	Description
Start Block	0	2	Start block of the extent.
Length	2	2	Amount of data blocks in extent.

Table 3.2: byte structure of an entry in an indirect data block.

The implementation of extents in this file system is a significant improvement upon the ext2 file system. Through the use of extents, storing the data blocks of large files is dramatically more efficient. This structure also making reading large amounts of contiguous blocks much faster as large chunks can be read all at once instead of piece by piece. It also helps reduce the amount of fragmentation in the disk, as this structure encourages the use of large contiguous blocks of data.

Some internal fragmentation is theoretically possible when it comes to using extents, however this implementation does not allocate extra blocks. Another potential downside is added complexity to certain systems, such as the addition of future features such as backups. I choose this feature because it provides many benefits with very few downsides, making it a valuable and viable improvement. If implemented with careful consideration, even the potential downsides of extents, such as internal fragmentation, can be mitigated or even negated.

3.6 Directory Entry Structure

Name	Bytes Offset	Size Bytes	Description
Inode Location	0	2	Location of the inode.
Entry Name	2	30	Name of the entry in the directory.
Entry Hash	32	32	SHA256 hash of filename.

Table 3.3: byte structure of directory data block.

Due to time constraints, directories only use a single direct and thus and only store eight entries. This could easily be changed, however I have opted to not add this feature for now.

The use of a hash table to store directory entries is a significant improvement upon the ext2 file system. By using a hash table, searching for a file by its name now becomes a O(1) operation instead of a O(n) operation. This results in a significant improvement in speed when checking if a file already exists or retrieving a file's entry. Operations such as copying large amounts of small files and making a series of searches will be much faster. I choose this data structure because of its dramatic effect on the speed of file searches in the file system. While it may increase the size of entries in directories, it is likely users will have most space taken up by large files and not extreme amounts of small entries. Even in the event that they do have many small entries, they would still benefit from fast search times for a large amount of entries.

IV Report

4.1 Overview

I decided to do this project in python because its simplicity would allow me to focus on adding features and data structures to the file system. I originally planned to add hash trees as my data structure, however I ended up choosing hash tables due to time constraints. Furthermore, I also ended up adding extents since I felt that hash tables were not unique enough on their own. I've written about these two features in their disk structure sections and how they affect the performance and integrity of the file system. In this section, I will focus more on implementation specifics and their challenges.

4.2 Adding Relative & Absolute Paths

One of the first features that I added to the file system is support for relative and absolute paths. This means that when using commands, we are able to specify entries that are not in the current working directory. So if I wanted to read a file in a directory that I was not in, I could run the command `read folder/text.txt`. This command would find the directory named `folder` in my current working directory and then in that directory find and read the file `text.txt`. I also added support for absolute paths, which means that I can start from the root directory and not the current working directory. For example, I could run the command `read /folder/test.txt`. This would start at the root directory, find the directory named `folder`, and then find the file named `test.txt`. It also makes it easy to get back to the root directory by running the command `cd /`.

There were some challenges when implementing this feature, as there are many things that the user is able to input that need to be handled. One thing that was a challenge was making sure that I was correctly tracking the directory from the root node so that when the cd command is used, the current working directory is, in fact, current. I needed to use the combination of the working directory and entered path to figure this out. Entering things such as `..` and `.` added some challenge, as they had different behaviors that I had to account for. I also needed to make sure that I would stop at the root node when using `..` and would not try to remove a previous directory. I also had to make sure to remove the last entry from the path given to return, as some commands only work with files or directories and often need their data. Overall, this complexity and many cases made this slightly challenging to get working.

4.3 Adding Recursive Folder Deletion

This was the second-most difficult feature that I had to add to the file system. I wanted the `rmdir` command to delete all sub-entries in the path. This is useful for deleting large amounts of files that are all contained in a folder. This however ended up being more difficult because of its recursive nature and the difficulty in dealing with another feature I added called hard links, made using the `link` command. With hard links, all entries have a number of links stored in their inode which represents the amount of times that entry is seen in directories.

This function was difficult to create as there were many cases that could exist and the fact that sub-entries are continuously added to the deletion queue. An entry is deleted if it has less than 2 links, otherwise its link count is decremented by 1. If this entry is a directory, then all entries contained in it should also be checked if they need to be deleted. I also had to make sure that this function would not somehow get back to the root and delete the whole file system, as this could be potentially dangerous.

4.4 Adding Hash Table

This feature was actually fairly simple to add to the file system, but I had to create a new python file for initializing the disk, since this feature changed the structure of the disk given in class. I decided to use SHA-256 to create hashes from the name of the entries in their directories. This means that I created a hash table for a directory when I navigate to it, using the hash as the key and the entry as the value. I changed some functions to support this instead of the previous way of searching the array. Overall, however, this allowed me to simplify my code while increasing search efficiency.

4.5 Adding Extents

This feature was slightly more complicated to implement than the last. I had to modify many functions to properly deal with this feature. I had to change how files were read, written, and decide how to find contiguous areas of disk space. For simplicity, I went from left to right grabbing all blocks until enough space was found and turning contiguous sections into extents. Furthermore, I made some changes to the original `Disk.py` file that I was given in class. Additionally, I added a new function that allowed me to write multiple blocks at once. This means that with extents, I was able to massively speed up write times to the disk.