Implementation of inode file system

Khan Ahmed Umair - 20172060

Md Zeeshan Ashraf - 20172100

Sharan Kumar Gangarapu - 20172014

Sai Charan Nivarthi - 20172086

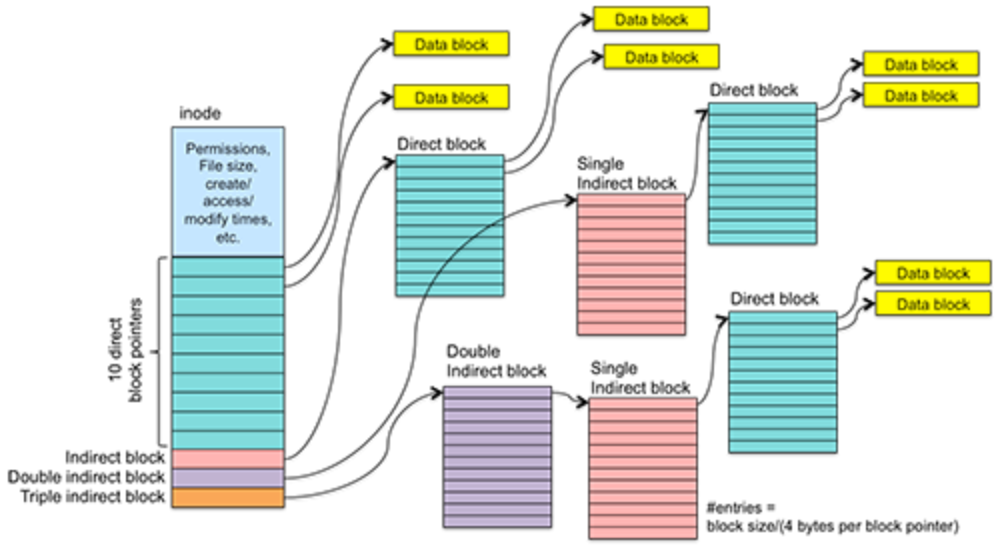
File System

In computing, a **filesystem** is used to control how data is stored and retrieved. Without a file system, information placed in a storage medium would be one large body of data with no way to tell where one piece of information stops and the next begins. By separating the data into pieces and giving each piece a name, the information is easily isolated and identified. Taking its name from the way paper-based information systems are named, each group of data is called a "file". The structure and logic rules used to manage the groups of information and their names is called a "file system".

inode

An inode is a data structure on a filesystem on Linux and other Unix-like operating systems that stores all the information about a file except its name and its actual data. A data structure is a way of storing data so that it can be used efficiently

inode structure



Data Structures used

The ***Super Block*** is typically the first block of the disk, and it stores information about the location of the other data structures. For example, you can store in the super block the whereabouts of the directory, the start of the inode blocks and start of the data blocks.

The ***Directory Structure*** holds the names of the files. It stores the mapping from file names to corresponding inodes.

The ***Inode Structure*** holds the information of attributes of a file like inode number, file type , file size , data block pointers.

Features

1.Create Disk

A disk will be created of appropriate size (Number of Data blocks times block size) using **create_disk()** method.

1. Input: use global variable **char filename[20]** (already defined) to create a file and allocate one data block to this file.
2. check if filename already exists in directory (using map dir_map)
3. check for free inode(if present i.e **sb.next_freeinode != -1**)=**in_free** & set it 0 (0 means now this inode no is not free)
4. set pointer of sb.next_freeinode to point next free inode if no free inode available then set **sb.next_freeinode=-1**
5. check for free data block (if present i.e != -1)
6. Assign a 4kb data block & set all the structure variables
7. set **inode_arr[in_free].filesize** as 0
8. set value of map **dir_map[string(filename)]** = **in_free**;
9. set directory structure also i.e **strcpy(dir_structure[in_free].file_name**, **filename**) & **dir_structure[in_free].inode_num** = **in_free**
10. set current nextfreeinode and nextfreedb indices and choose **nextfreeinode** and **nextfreedb** indices

2.Disk mounting

Mounting of a disk technically means initializing all the variables in super block structure of inode file system. This is done using **mounting()** function.

1. retrieve super block from virtual disk and store into global struct **super_block sb**
2. retrieve DS block from virtual disk and store into global **struct dir_entry dir_structure[NO_OF_INODES]**
3. retrieve Inode block from virtual disk and store into global struct inode **inode_arr[NO_OF_INODES];**
4. Store all file names into map
5. populate **freeinodevec** and **freedbvec**
6. Populate Free Filedescriptor vector

3.Create a file

File name will be provided by user and it will be created by allocating default size using **create_file(char* filename)** function.

1. input: use global variable **char filename[20]** (already defined) to create a file and allocate one DB to this file
2. check if filename already exists in directory (using map dir_map)
3. check for free inode(if present i.e **sb.next_freeinode != -1**)=**in_free** & set it 0 (0 means now this inode no is nt free)
4. set pointer of sb.next_freeinode to point **nextfree** inode if no free inode available then **set sb.next_freeinode=-1**
5. check for free db(if present i.e != -1)
6. assign a 4kb db & set
7. set **inode_arr[in_free].filesize=0;**
8. set value of **map dir_map[string(filename)]** = **in_free;**
9. set directory structure also i.e strcpy(dir_structure[in_free].file_name, filename) & dir_structure[in_free].inode_num = in_free
10. set current nextfreeinode and nextfreedb indices and choose **nextfreeinode**

4.Open file

A file-descriptor is assigned to a file and it is returned using **open_file(char* name)** function. A map with file descriptor as key and a pair of Inode and current cursor position as value because fd must be unique . Each time on opening a file openfile_count++ is done and we get a free file descriptor from free File Descriptors Vector and make it as key and assign it to Inode and on closing openfile_count- - is done and remove the entry from map and push file descriptor in **freeFileDescriptorsVec** limit the **count_of_openFile** to max number of fd's. Can allocate the file descriptor until freeFileDescriptorsVec becomes empty allocate the file descriptor and set cursor position to 0 return the allocated file descriptor to the called function.

5.Delete a file

Deletes a file based on the filename given.

1. if file is open show the prompt to user whether they want to continue delete (Y/N)
2. if filename doesn't exists then throw error
3. get inode no=in_num from **map dir_map**, go to that inode and track all DB and set **sb.datablock_freelist[]** = **0**;//0 means DB is free
4. Set **inode_arr[in_num].pointer[0-12]** = **-1** and set **sb.inode_freelist[in_num]=0;** 0 means inode is free
5. Remove entry of this inode from map dir_map and directory st dir_structure
6. Iterate through inode pointers [0-9][11][12]
7. Now free the inode;
8. Resetting inode structure with default values.
9. writes 'size' no of bytes from buf into file pointed by fd from **cur_pos** mentioned in fd

6.List

This is simulation of 'ls' command of linux. Here all the files present in the disk are listed. File need not be open to find itself in List of files.

7.Unmount the disk

The disk that has been created is unmounted.This function unmounts your file system from a virtual disk with name disk_name. As part of this operation, you need to write back all meta-information so that the disk persistently reflects all changes that were

made to the file system (such as new files that are created, data that is written, ...). You should also close the disk. The function returns 0 on success, and -1 when the disk disk_name could not be closed or when data could not be written to the disk (this should not happen).

It is important to observe that your file system must provide persistent storage. That is, assume that you have created a file system on a virtual disk and mounted it. Then, you create a few files and write some data to them. Finally, you unmount the file system. At this point, all data must be written onto the virtual disk. Another program that mounts the file system at a later point in time must see the previously created files and the data that was written. This means that whenever unmount_fs is called, all meta-information and file data (that you could temporarily have only in memory; depending on your implementation) must be written out to disk.

REFERENCES

1. ***http://www.cs.ucsb.edu/~chris/teaching/cs170/projects/proj5.html***