Design Documentation

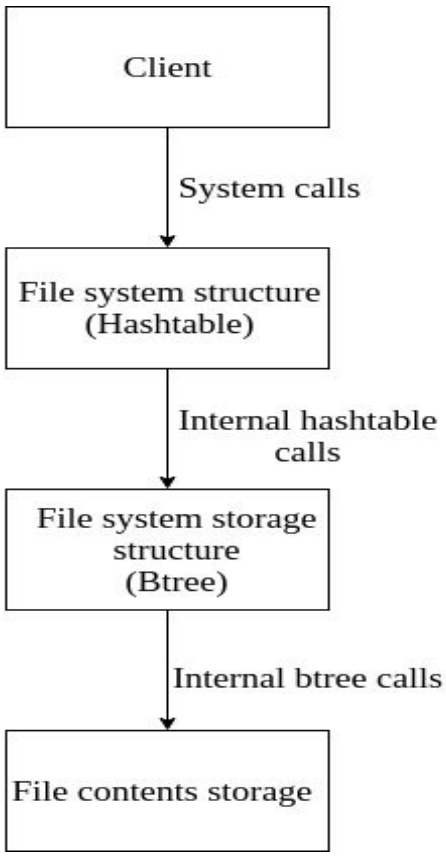
Problem statement:

To build a simple file system.

Overview:

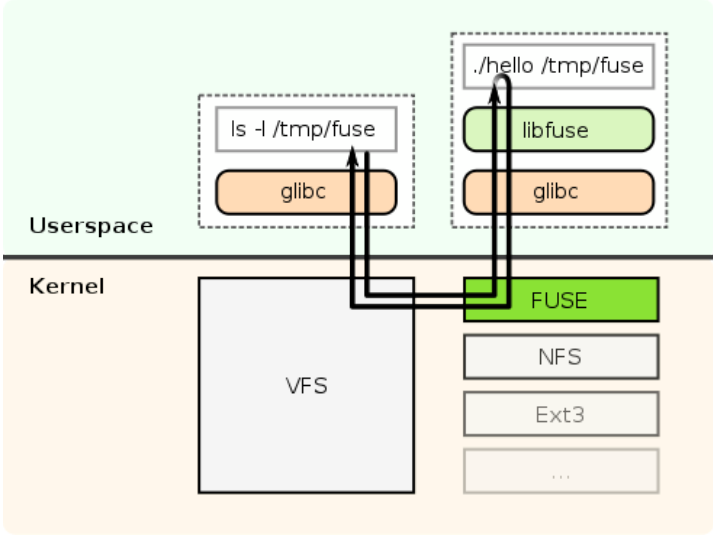
The objective of the project was to have a practical knowledge about the implementation of a Linux like file system, thereby familiarising ourselves to concepts like file systems, file IO, device interactions and system calls.

The major components of the file system were to implement system calls in order to interface with the file system data structures(implemented using hashtable), which internally call functions to access the file system storage structures(implemented using Btree). The block diagram for our file system is given below,



Phase I : System call implementation

This phase involved implementation of system calls for various file operations as issued by Linux. For the system call components, the software facility called FUSE can be utilised. **Filesystem in Userspace** (**FUSE**) is a software interface for Unix-like computer operating systems that lets non-privileged users create their own file systems without editing kernel code. This is achieved by running file system code in user space while the FUSE module provides only a "bridge" to the actual kernel interfaces.



FUSE interface - How it works?

By phase I, the following tasks were completed,

- Implementation of mkfs to make a file system in the emulator along with File I/O operations and directory functions.
- Implement the file abstractions (block management, block maps, directories, etc.) using the in-memory emulator
- Integrate the file system implementation with FUSE

Thus, an in-memory file system was emulated using FUSE and the system calls implemented were,

- Open
- Read
- Write
- Create
- GetAttribute
- Mkdir
- Cd
- Ls
- Close
- Rmdir
- Rm

Phase II : File Systems Abstraction

One of the main points and features of a filesystem is **abstraction**. With a filesystem, we can organize our data into files, directories, and other constructs, and manipulate them in various ways. To open a file, we need only its path; it's not necessary to figure out the exact location on disk and instruct the hard drive controller to move the read head to that position.

Implemented block operations functions like insert-block, delete-block, display-block where block is implemented using a Btree.
The file system data structures were implemented using hashtable with suitable implementations for the corresponding operations. The design of the file system is as follows,

Design of the file system and data structures chosen :

- The files and directories in our FS are represented by inode structures which store the various properties of the file like inode number, mode, file type and size, access, modification and change status time.
- The file system structure is implemented as a hashtable. Here, the path to file mapping is done using the key-value pair of the hashtable. The key represents the path and the value represents the inode structure which holds the attributes of the file. The reason for choosing this is to make file searching more efficient

(constant time in this case), when compared to the linear search for an array or a linked list implementation.

Hashtable : This is an array like structure which stores data as key-value pairs. The key is hashed (using a hash function) which returns a value which is the index in the hashtable at which the value will be inserted. Because of the hash function, the insert, delete and search are constant time operations.

- The storage structure of our FS is a Btree in which, the key represents the inode number and the value/node points to the actual file contents. This kind of an implementation is extremely efficient, as there is no hole creation on delete since Btree uses splitting and merging techniques in order to provide optimal space utilisation(external fragmentation of memory is completely eliminated). Along with optimization in memory Btree also allows you to perform search in O(log n) time which is much better than the linear search which would occur in case of contiguous memory allocation or the linked list implementation of data blocks. Thus providing maximum optimization both in terms of speed as well as memory utilisation. This was inspired by the popular use of B+Trees in distributed file system, nowadays.

Phase III : Secondary storage

This phase implemented persistence in the file system such that the file system gets preserved across multiple machine boots. Persistence was achieved by making the hashtable and the Btree persistent, ie, every insert, delete must be updated on a metadata file such that on every mount, the metadata is checked and the data structures are populated with the respective values.

The Btree is a superblock for our FS and the file content is stored in the nodes of the Btree.

Overall flow of code :

- The client side makes system call

  Eg : cat src_file - system call
- This call reaches the interface of the hashtable which makes internal hashtable calls in order to read from or write into the hashtable.

  Eg : ht_display_inode_file_contents(source_path) - hashtable call

  Here, source_path is the key which we hash and the result obtained is used to index the hashtable to extract the corresponding inode.
- The hashtable functions call Btree functions to access the actual data blocks.

  The inode number obtained from the inode struct in the hashtable is used as a key in the Btree to reach the data blocks represented by the nodes of the Btree.

  Eg : display_content(tree_ptr,ht_node.inode.inode_num) - Btree call

Here, the Btree is searched in O(log n) time for the required data block using the inode_num parameter.

Unique Features:

Use of custom made shell instead of FUSE:

- To make our file system independent of any API, we built our own dummy shell which is portable with our file system.
- The shell both mounts and opens the file system when it is booted, thus making it a personalized interface.
- The shell basically consists of a parser which captures and analyses the command line arguments.
- Once the commands are analysed, the respective system calls are called to invoked to execute the operations on the file system.

Additional optimizations

In addition to the Shell, we also provide a much more efficient way to read and write into files by making use of buffers (in the form of text files) to provide optimized file IO. When a file is opened the contents of the file are loaded into the buffer (the text file) and all the file operations are performed on the same. The modified data is written back on the disk only at the time of close. Thus, the file on the hard disk is not modified for every operation and is instead updated after all the operations have been completed. This allows a single disk write at the end instead of having a disk write for every operation

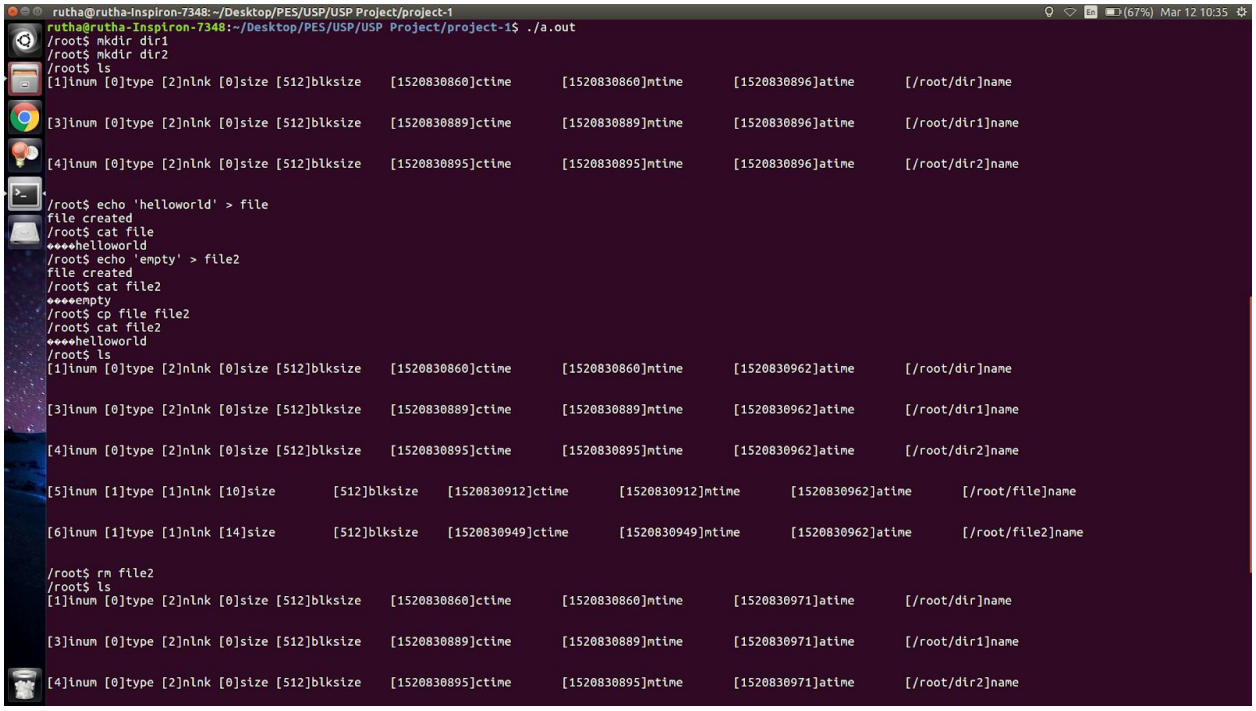
thus improving the performance as disk writes are expensive and using read-write buffers provides optimization to the implementation.

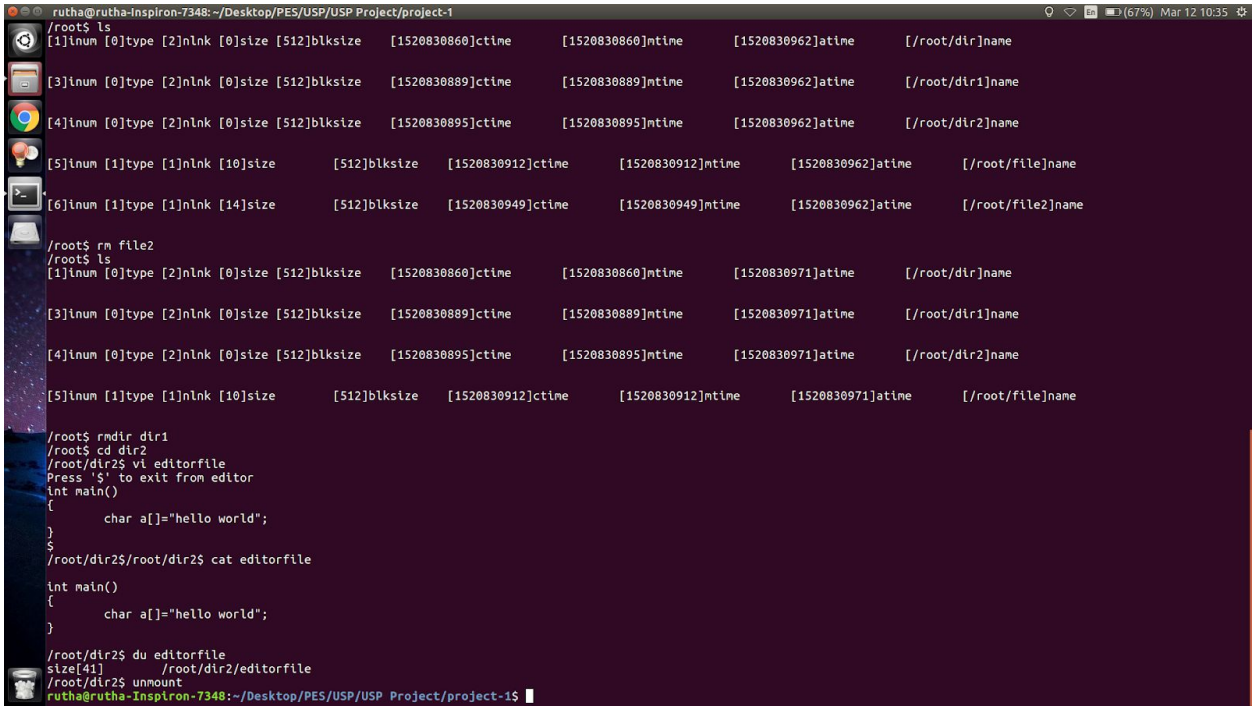
Unix Commands Implemented:

Along with the standard Unix commands given below, we implemented our own version of Vi editor in our shell. The editor opens a new file if non-existent, and returns an error if the user accesses an already present file. The editor can be exited on entering a “$”. The editor works on buffered input storage and thus can be edited till the exit, as no data is written into the file.

- echo ‘string’ > filename (creates a file and writes string into it)
- echo ‘string’ >> filename (appends string to existing file, returns error if file non-existent)
- cp sourcefile targetfile
- rm filename
- ls file/dir_name (lists all files)
- du -s filename (displays file size)
- cat filename (displays the contents of the file)
- cd target_dir (goes to target directory if exists)
- mkdir target_dir (creates the directory if non-existent, returns error if exist)
- rmdir dir_name

Unix Commands Example:





ReadMe.txt instructions (Visual representation) :