UNIX SYSTEM PROGRAMMING LABORATORY (UE15CS355)

TEAM MEMBERS

Nitish J Makam	01FB15ECS197
Parashara R	01FB15ECS202
Pavan Yekbote	01FB15ECS205

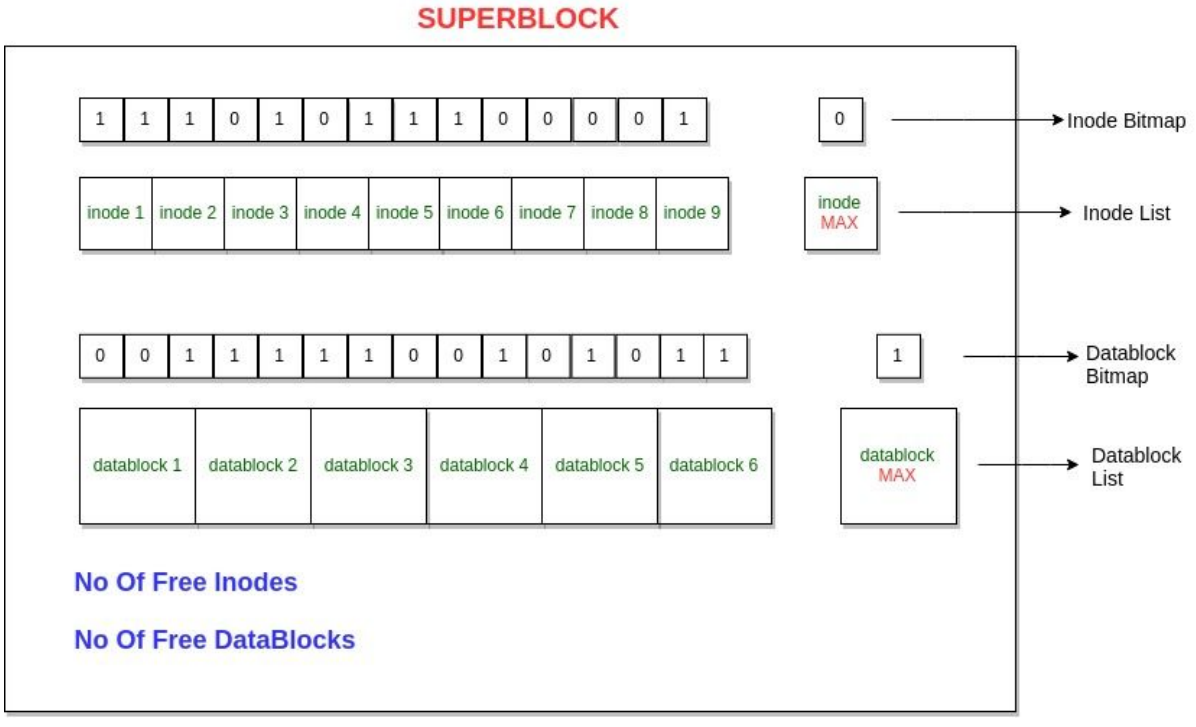
IMPORTANT STRUCTURES USED

We have three structures that are fundamental to our file system. They are -

1.SuperBlock :

The superblock is the heart of our file system. It contains of a bitmap for inodes that indicate whether a particular inode is free or not.
It also contains a pool of inodes. Whenever a new file or directory is created, the inodes are picked up from this pool.
Similarly, it contains a bitmap for datablocks that indicate whether a particular datablock is free or not.
It contains a pool of datablocks. Whenever content is written to a file, datablocks are picked up from this pool and written into them.
It also contains two integer variables which store the number of free datablocks and free inodes.

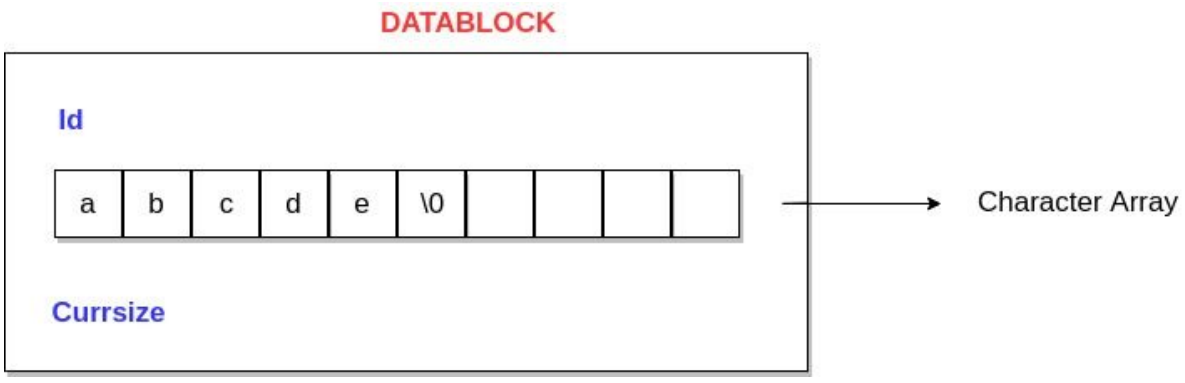
It's block diagram is as shown below.



2. DataBlock :

The second important structure in our file system is the DataBlock.
Each DataBlock structure contains an integer variable id - that uniquely identifies the data block.
It also has a character array that stores the contents of the datablock.
Another integer variable called currsize denotes the number of characters currently written into the datablock.

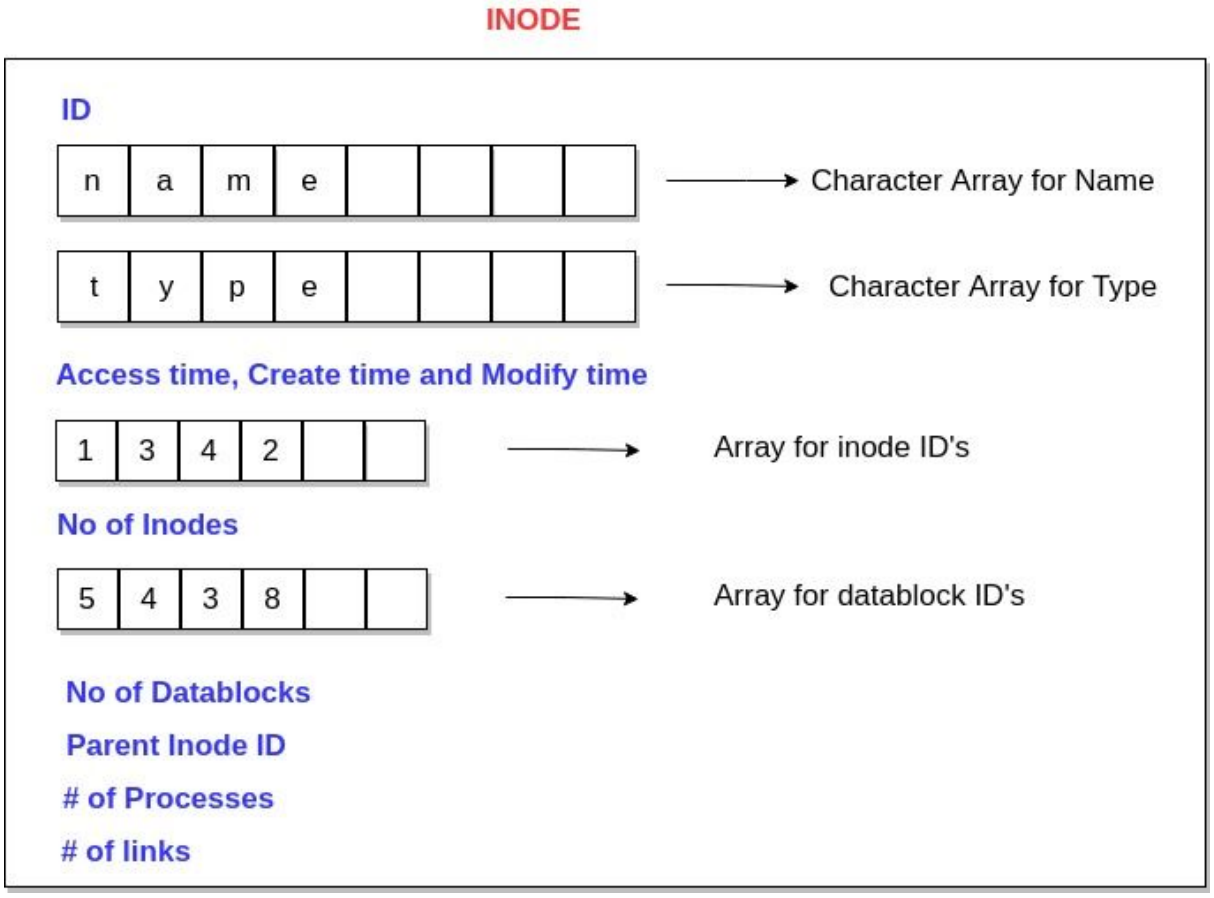
The block diagram of the datablock is as shown below.



3. Inode :

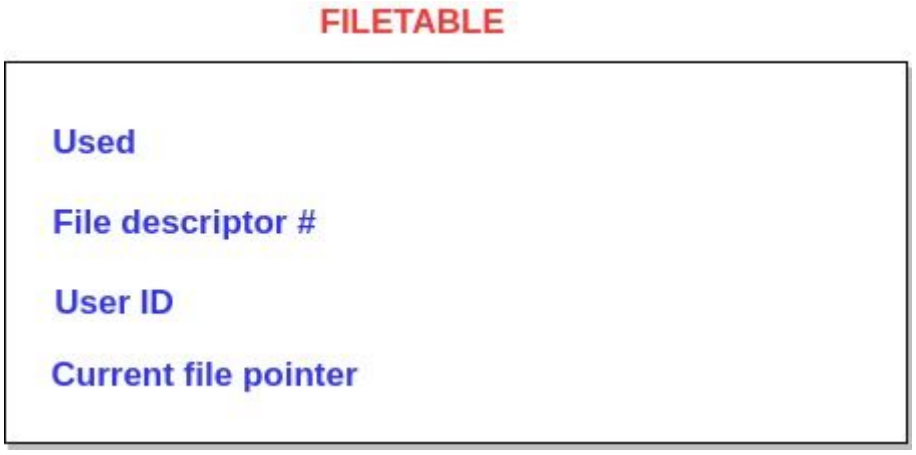
The third important structure in our file system is the inode.
Firstly, it contains an integer variable id - that uniquely identifies the inode in the pool of inodes.
It has two character arrays - one for the name and one for the type(file or directory).
It has three time variables - access, create and modification times.
It has an integer array that stores the IDs of the inodes of the files and directories within it if it is a directory itself. An integer variables denotes the number of files and directories within itself.
It also has an integer array that stores the IDs of the datablocks if it itself denotes a file. An integer variable indicates the number of datablocks in which the file's contents are stored.
Another integer variable stores the ID of the inode of it's parent directory.
There are two more integer variables that indicate how many processes have currently opened the file and the number of links to the file.

The basic layout of the inode structure is as shown below.



4. FileTable :

Another structure that is vital to our file system is the FileTable.
There are a predefined limited number of entries that the file table can hold.An integer variable indicates whether this entry is used or unused. The other fields include an integer variable which denotes the file descriptor number. It also records the ID of the user who has the file open. Another variable stores the current file pointer of the file. The basic layout of our filetable structure is as shown below.



OVERVIEW OF THE FUNCTIONS

There are several functions that help in performing the operations of the file system. Below are their prototypes and a brief description of what functionality each of them help achieve and an overview of their implementation -

→ `int createInode(char * name, char * type)`

This function takes a free inode from the inode pool in the superblock by calling the getInode function.It also sets its name and its type - either file or directory. It also initializes all the other attributes of the inode - such as the times, the number of links, its parent directory etc.

→ `int deleteInode(inode * entry)`

This function returns an inode to the pool of inodes in the superblock and deletes the entry for this inode in it's parent directory. It the inode refers to a file, it first unlinks all it's datablocks and then returns the inode to the pool. If it's a directory, the directory is deleted only if it is empty.

→ `char *readDataBlocks(inode *file)`

This function takes a pointer to an inode as argument and reads the contents of all the datablocks in the file by scanning the array of datablock IDs that the inode holds. It concatenates the contents of each datablock and returns a pointer to this string.

→ `datablock* writeDataBlock(char* content)`

This function takes a character string as an argument and takes a free datablock from the empty pool of datablocks by calling the getDataBlock function in the superblock and writes the string into the datablock. It also sets the value of the length of the string written into the datablock. It returns a pointer to the datablock to which it wrote the contents to.

→ `int unlinkDataBlock(inode *file)`

This function takes as argument a pointer to an inode that refers to a file, scans the array that stores the datablock IDs that contains the contents of the file pointed to by the inode, and releases them back to the free pool of datablocks in the superblock by marking them as unused in the bitmap for the datablocks in the superblock structure.

→ `int addEntry(inode *file,int who)`

This function takes as arguments a pointer to inode that refers to a file along with the current user ID and creates an entry within the filetable while setting the currentfilepointer to 0. It returns 1 on success and 0 on failure.

→ `int removeEntry(inode *file,int who)`

This function takes as arguments a pointer to inode that refers to a file along with the current user ID and removes the corresponding entry in the filetable. It returns 1 on success and 0 on failure.

→ `int flushfiletable(inode *file)`

This function takes as arguments a pointer to inode that refers to a file and removes all entries corresponding to the file. It returns 1 on success and 0 on failure.

→ `filetable *getEntry(int fd, int who)`

This function takes as arguments a file descriptor corresponding to a file along with the current user ID and fetches the entry which matches the filedescriptor and the user ID from the file table. Returns a pointer to the corresponding filetable entry on success and NULL on failure.

→ `int superblockInit()`

This function takes no arguments and creates the superblock structure and the pool of inodes and datablocks.It also initializes all the members of the superblock. It returns an integer 1 for success and 0 for failure.

→ `inode *getInode()`

This function scans the bitmap for the inode pool in the superblock and returns a pointer to the first inode that is unused. It takes no arguments. If it does not find an inode that is unused, it returns NULL.

→ `datablock *getDatablock()`

This function takes no arguments and is similar to the getInode function. It scans the bitmap for the datablock pool in the superblock and returns a pointer to the first datablock that is unused. If there is no unused datablock, it returns NULL.

→ `void printSupportedCommands()`

This function takes no arguments and returns nothing. It prints the commands that are supported by our file system and gives the syntax for each of the commands.

→ `char ** split(char * input,int noofwords)`

This function takes as input a character string and the number of words that are to be split and returned. It splits the string that it takes as an argument. The space character serves as the delimiter. It returns an array of pointers to the constituent strings.

→`inode * getInodeFromCurrDirectory(char *name,char *type)`

This function takes as arguments two character strings - one for the name and another for the type. It searches the entries in the current directory for inodes within itself with the same name and type as its arguments. It returns a pointer to this inode. If no inode is found with the matching values, it returns NULL.

→ `void showFileTableContents()`

This function takes no arguments and returns nothing. When this function is called it prints the contents of all the entries in the filetable. This filetable is system-wide open-file table.

→ `int getSizeOfFile(inode * file)`

As the name suggests, this function takes as argument a pointer to an inode that refers to a file, scans the array that contains the IDs of it's datablocks and returns the sum of the number of characters in each of it's datablocks.

→ `int mkfs(int mount)`

This function takes as arguments an integer variable - if it's value is 1, it restores all the files and directories - if it's value is 0, it creates a file system afresh. It returns an integer variable with values 1 for success and 0 for failure.

→ `int sfscreate(char * name)`

This function takes a character string name as input and creates a file with this name in the current directory by calling the createInode function with name and type as file. It also links this to the current directory. It returns an integer variable 1 on success and 0 on failure.

→ `int sfsmkdir(char * name)`

This function takes as argument a character string name and creates a directory by calling the createInode function with name and type as directory. It also links it to the current directory. It returns an integer variable 1 on success and 0 on failure.

→ `int sfschangedir(char * name)`

This function takes as argument a character string name which represents an existing directory and changes the current working directory to the one specified. It returns 1 on success and 0 on failure.

→ `int sfsreaddir()`

This function takes no arguments. It traverses the array that contains the IDs of the inodes of files and directories within this directory and prints the name and type of the contents of this directory. It returns an integer variable 1 on success and 0 on failure.

→ `int sfsrmdir(char *name)`

This function takes as arguments a character string name and deletes a directory entry with the same name within the current working directory. It releases the inode that referred to the directory with this name into the pool of inodes by calling the deleteInode function. It returns an integer variable 1 on success and 0 on failure.

→ `int sfsdelete(char *name)`

This function takes as arguments a character string name and deletes the entry within the current working directory for a file with the same name as the argument in received. It releases the inode that referred to the file with name 'name' by calling the deleteInode function. It returns an integer variable 1 on success and 0 on failure.

→ `int sfsclose(char * name,int who)`

This function takes as arguments a character string name along with the current user ID, it retrieves the corresponding inode for the filename specified and calls the removeEntry function which removes the corresponding entry for the file from the filetable. Returns 1 on success and 0 on failure.

→ `int sfsopen(char * name , int who)`

This function takes as arguments a character string name that refers to a name of the file in the current directory and the user ID and appends an entry into the system wide open file table with these values and sets the offset in this entry to 0. It returns an integer variable with value 1 on success and 0 on failure.

→ `int sfslseek(char *name,int who,int offset)`

This function takes as arguments a character string that denotes the filename, the user ID and an integer variable - offset. It scans the filetable for entry that matches the name and user ID and sets the current file pointer in this entry to 'offset'. It returns an integer variable 1 on success and 0 on failure.

→ `int sfswrite(char *filename,int who,char *content)`

This function takes as arguments a character string name that denotes the filename, the user ID and a character string called content and writes the content into the file specified by its filename. It returns 1 on success and 0 on failure.

→ `char * sfsread(char * name,int who,int nbytes)`

This function takes as arguments a character string that refers to a filename in the current directory, the user ID and an integer that denotes the number of bytes to be read from the file. If the number of databytes to be read from the file exceeds the actual size of the file, NULL is returned. Else a pointer to the character string with length nbytes is returned.

→ `int automount()`

This function acts as a wrapper function for superblockDiskread() and filetableDiskRead(). It is called if we wish to mount our previous filesystem. It also sets the disk argument to PersistentDisk.txt and returns 1 on success and 0 on failure.

→ `int dumpfs()`

This function acts as a wrapper function for diskWrite() and sets the disk argument to PersistentDisk.txt. It is called upon exiting from the shell. Returns 1 on success and 0 on failure.

→ `int diskWrite(char *disk)`

This function is called from dumpfs(). It writes the current superblock and filetable structures into a file specified by the argument disk using fwrite() in order to achieve persistent. It returns 1 on success and 0 on failure.

→ `int superblockDiskRead(char *disk)`

This function is called from automount(). It reads the superblock structure present in the PersistentDisk.txt file by using fread(). It returns 1 on success and 0 on failure.

→ `int filetableDiskRead(char *disk)`

This function is called from automount(). It reads the filetable structure present in the PersistentDisk.txt file by using fread(). It returns 1 on success and 0 on failure.

MAIN

The main function is in the file myshell.c. It runs a loop that behaves like a shell. Given commands, it calls the split function to split the command into constituent pieces and calls the required function with the appropriate arguments to perform the appropriate actions conveyed by the command.

INSIGHTS & CONCLUSION

- ➔ Implementing our own file system exposed us to the environment of working in teams and made us understand the importance of getting the design right and then implementing it.
- ➔ Implementing our own file system gave us a better insight on how inodes, datablocks and the superblock work together hence improving our understanding of a filesystem.
- ➔ It also refreshed our knowledge of C. We feel a lot more confident to code in C.
- ➔ We also learnt what goes into making a project and how to assign and divide work among ourselves. We also used git for this project.