INTRODUCTION TO OPERATING SYSTEMS LAB

COURSE CODE : UE16CS305

PROJECT REPORT

ON

"FILE SYSTEM IMPLEMENTATION WITH FUSE"

Team Member	Name	SRN
1.	Abhishek Narayanan	01FB16ECS016
2.	Abhishek Prasad	01FB16ECS017
3.	Abijna Rao	01FB16ECS019
4.	Adapa Shivani	01FB16ECS022

PROJECT TITLE :

Implementing our own fully functional file system including the internal data structures and system calls similar to a Simple File System in UNIX Operating Systems using FUSE (File System in User Space)

GOALS :

The purpose of this project was to gain practical knowledge of Linux internals by creating a basic file system and understanding the internal data structure representations along with system call implementations.

ABSTRACT :

This mini project is aimed at creating our own implementation of the data structures and system calls at par with the existing file system for the UNIX Environment. The data structures involved such as the super block, inode and data bitmaps, data blocks, inodes and inode table, etc. were all implemented from scratch in this project .

After the creation of a file system abstraction, using our own helper library, for creating the backend structures and performing operations with them, we implement from scratch, few system calls of UNIX OS, commonly used in practice for necessary file system operations.

This is achieved as the FUSE framework (File Systems in User Space) provides a way to intercept system calls from shell and transfer the control flow to our own program handler rather than to the kernel itself.

The whole project has been decomposed into III phases for a step by step procedural approach for implementing a fully functional file system from scratch.
The III phases in brief are summarized below and elaborated later.
From an engineering perspective, the project decomposes into three tasks:

system call implementation-- implementing the system calls that can be issued by Linux on files.

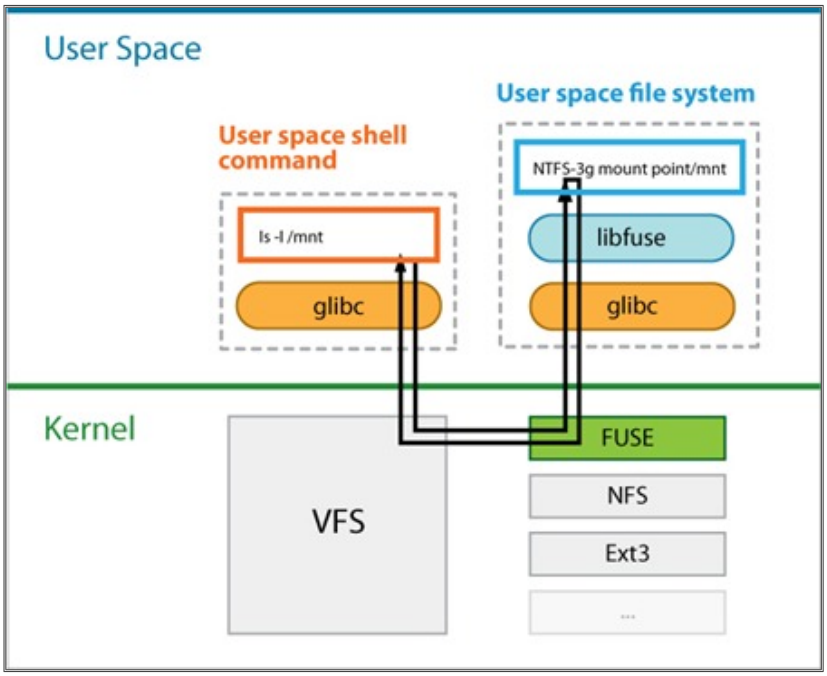
implementing the file abstraction-- building the internal data structures and procedures necessary to implement files.

implementing secondary storage management-- building the parts of the file system that persist in secondary storage

Implicitly, there is a fourth task which is to integrate these three tasks into a single, working file system.

OVERVIEW OF FUSE (File System in User Space) :

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. The resulting file system resides in the userspace and exists as a layer of abstraction over a more concrete and robust filesystem. FUSE is free software originally released under the terms of the GNU General Public License and the GNU Lesser General Public License.



To implement a new file system, a handler program linked to the supplied libfuse library needs to be written. The main purpose of this program is to specify how the file system is to respond to read, write, stat or other requests. The program is also used to mount the new file system. At the time the file system is mounted, the handler is registered with the kernel. If a user now issues read/write/stat requests for this newly mounted file system, the kernel forwards these IO-requests to the handler and then sends the handler's response back to the user, as shown in the figure above. FUSE is particularly useful for writing virtual file systems. Unlike traditional file systems that essentially work with data on mass storage, virtual file systems don't actually store data themselves. They act as a view or translation of an existing file system or storage device.

In principle, any resource available to a FUSE implementation can be exported as a file system.

IMPLEMENTATION DETAILS :

The project is broadly divided into three phases:

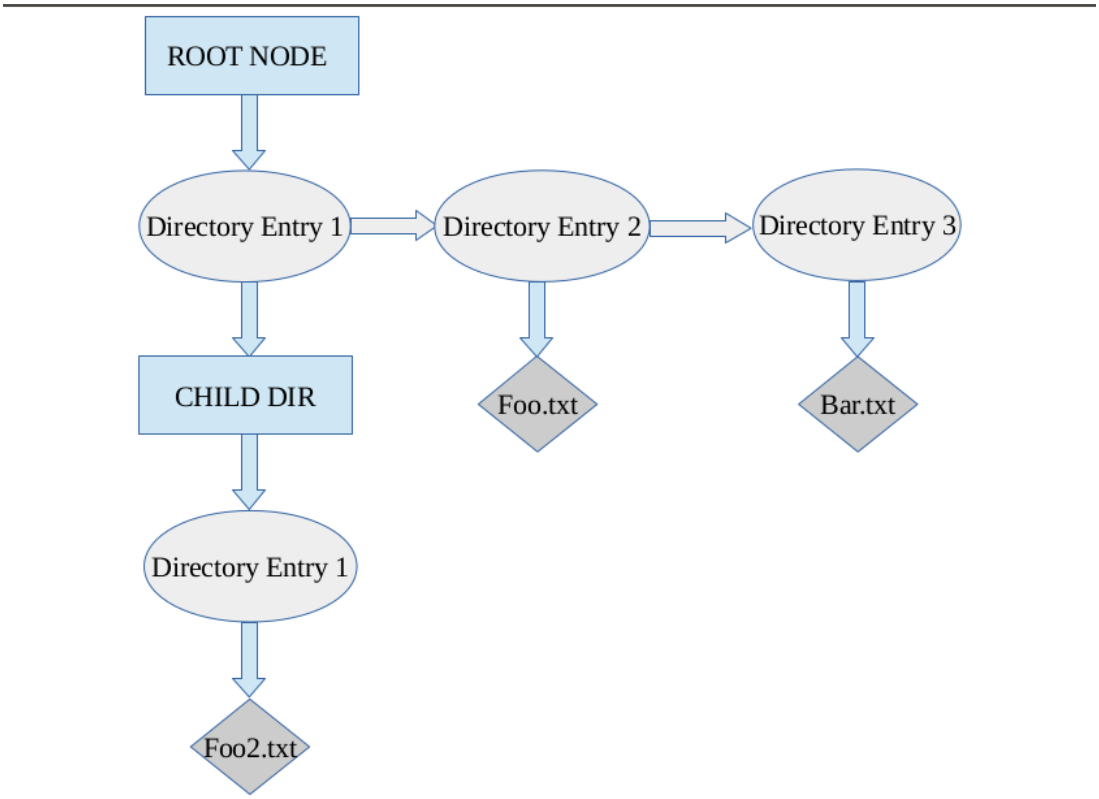
PHASE 1 :

GOAL :

In this Phase, We set up fuse and experiment with a pass through file system and implement simple file operations via system calls using FUSE , without developing file a system structure and allocating blocks in memory such that when a user invokes a system call, the program flow is redirected to the daemon process of the handler which we implemented, rather than the actual Linux file system implementations.
We do not yet implement the actual internal file system abstractions, however we simulate the basic functionalities of the system calls prescribed using our own data structure abstractions, the details of which have been elaborated below further.
We do not ensure persistence in disk at this level and perform all operations in memory.

FILE SYSTEM ABSTRACTION FOR PHASE I :

The File System Abstract Structure which we used to implement all these system calls has been represented pictorially below : (Rectangles represent Directories, Ellipses represent Directory Entries, Diamonds represent Regular Files)



The structure of the file system we created is portrayed in detail in the figure above.

- All system calls have been implemented assuming the above data structure abstraction.
- Some key points about the above simple file system :
- The File System was implemented as a hierarchical Linked List as shown in the diagram above.
- We have assumed that each file entry can occupy a maximum of one whole data block.
- The Block size has been assured to be 4096 byes or 4kB.
- The file system is represented as a structure with the root node on top of the hierarchy.
- The directories contain directory entries consisting of the following :

  ➔ **File name**
  ➔ **Inode for the corresponding file**

- Pointer to the next directory entry within the same parent directory
- The directory entries are linked together as a singly linked list, so as to be traversed easily and contain either further directory entries or file nodes.
- Each file is represented by a node structure analogous to the inode of real time file systems, where a node consists of the following attributes :

  ➔ **Block Number for the corresponding data block of the file**
  ➔ **File Descriptor to keep track of number of open files**
  ➔ **A stat structure containing the meta data about the file**

- Since all nodes and structures are dynamically allocated and appended to the hierarchy structure, we have not placed any bar on the number of files to be stored,
- However we assume that each file is allocated only one single data block, which may not support larger files in this phase.
- Every allocation/deallocation task for each operation is performed in the main memory and not on the secondary storage, and therefore the entire data in the file system is erased on unmounting the file system.

SYSTEM CALLS IMPLEMENTED :

The FUSE operations Structure specified by us is as follows :

```
static struct fuse_operations memfs_oper =
 {
  .getattr      =   osproject_fs_getattr,
  .readlink    =    osproject_fs_readlink,
  .readdir     =    osproject_fs_readdir,
  .mknod      =    osproject_fs_mknod,
  .mkdir       =    osproject_fs_mkdir,
  .unlink      =    osproject_fs_unlink,
  .rmdir       =    osproject_fs_rmdir,
  .link        =    osproject_fs_link,
  .truncate    =    osproject_fs_truncate,
  .utimens    =    osproject_fs_utimens,
  .open        =    osproject_fs_open,
  .read        =    osproject_fs_read,
  .write       =     osproject_fs_write,
  .release    =     osproject_fs_release,
  .rename     =     osproject_fs_rename
};
```

The Fuse functions which were implemented to simulate actual system calls are as follows :

Sl No.	SYSTEM CALLS	FUNCTIONALITY
	OPERATIONS INVOLVING DIRECTORIES	
1.	mkdir( )	Creates a new directory with name specified from the given path by adding the “path” to the directory table with mode specified by the passes parameter, the default mode being 777.
2.	readdir( )	Reads the entries in a directory and fills the buffer provided by the caller.
3.	rmdir( )	It removes the given directory, provided it is empty.
	OPERATIONS INVOLVING REGULAR FILES	
4.	mknod( )	Creates a special file with the given name and of type specified in the mode parameter passed.
5.	open( )	Opens a file in the given path.
7.	rename( )	renames a file to the parameter specified and internally calls rmdir(for a directory) and unlink( ) (for a regular file).
8.	read( )	reads from a file and stores the bytes read into a buffer.
9.	write( )	writes “size” bytes of data starting from the given “offset”, specified from the buffer onto the file.
10.	getattr( )	Initializes the default attributes of a file or a directory.
11.	link( )	Create a hard link between two files specified by its arguments.
12.	readlink( )	If path is a symbolic link, fill bufferwith its target, up to size.
13.	unlink( )	Removes (deletes) the given file, symbolic link, hard link, or special node. Note that if unlink only deletes the data when the *last* hard link is removed.
14.	truncate( )	Truncate or extend the given file so that it is precisely size bytes

PHASE II :
GOAL :

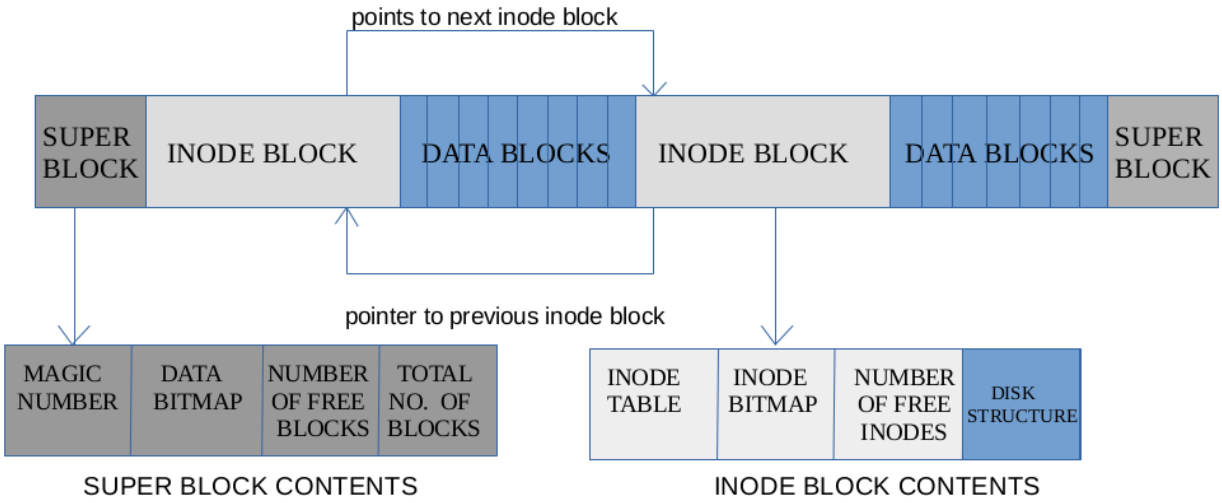
In this phase, We come up with a simple file system structure and implement the file system abstractions like the inode blocks, data blocks, directory structure and also to begin porting our previously implemented file system from memory to secondary storage .
In order to achieve this task, we need to create a disk emulator for persisting every file operations we perform. To do this, we create a large binary file of 8.2 MB by zero filling of the file. This single large UNIX file is used to emulate the disk or secondary storage since it is not feasible to use an actual raw disk for this purpose. Hence we implement our own disk emulation library to simulate disk operations. The library functions implemented are capable of dividing the file created into blocks of size 4kB each and perform open, read and write operations with blocks into this virtual disk. This will be the underlying support for the whole project and act as a disk interface for the user performed file operations to run on top of this layer.

DISK EMULATION :

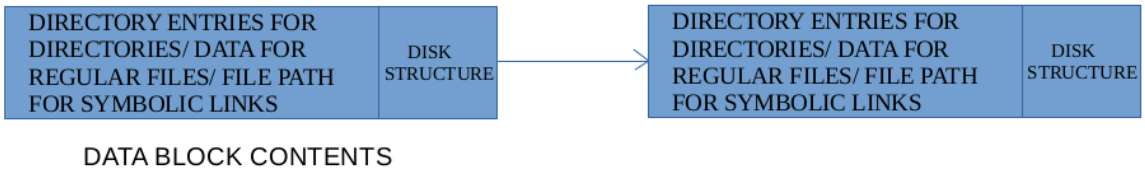
We implement the following disk abstractions :

int openDisk( ) :
This helper utility opens the binry file we created to store all information and changes pertaining to the file system or creates a new binary file and zero fills it to make it at par with the assigned size (8.2 MB).
If the file was already exitent, this method reads both of the super blocks at the beginning and end of file and checks if the super block has been corrupted. Further operations proceed if both the super blocks show no variation, else the entire file system is crashed.
int readBlock( )
Reads a disk block specified by a given block number from a specified offset into a buffer. Returns the number of bytes read and also raises an error if an attempt is made to read an invalid byte (i.e., offset or size specified is too large than expected).
int writeBlock( )
Writes a disk block specified by the given block number pointed to by the disk. A black may be overwritten without any errors, and write may also be performed starting from a specified offset. However an error is raised if the offset and number of bytes to be written happen to be invalid.
int superblockread()
Reads the super block at the beginning and returns the number of free available data blocks.
char getblocktype( )
Function to know what kind of block it is, i.e. Inode,directory,data.
int get_next_block( )
Function to get next or previous block numbers with respect to a specified block number, based on a flag which represents which block number is to be returned.
int reqblock( )
Function to request a block of given type and allocate the block if sufficient memory is available in the virtual disk.
int relblock( )
Function to release a block assigned and adjust the next or previous blocks links suitably to ensure that an existing linked list is not disturbed because of the deletion.

FILE SYSTEM ABSTRACTION FOR PHASE II:



A particular file or directory may occupy a single data block or multiple data blocks in case of larger blocks. Data Blocks belonging to a particular file are linked together as a singly linked list.



The Inode block consists of the following :

```
struct inode_block
 {
        //inode block structure
        struct node inodes[25]; //inodes
        char bitmap[25]; //bitmap
        int free_inode_no; //number of freeinodes
  }
```

Each File node contains the following :

```
struct node
 {
        struct stat   vstat;
        int data; //block no. of data block
        unsigned int  fd_count;// File descriptor
 }
```

Each directory entry consists of the following :

```
struct direntry
{
        char  name[256]; //assume PATH_MAX is 256 (for file name)
        ino_t   node_num; //inode number
        int next; //stores 0 or 1 indicating if there is a file or directory in the same level
}
```

FILE SYSTEM ABSTRACTION DESCRIPTION :

- Block size assumed is 4kB or 4096 bytes.
- Every data block pertaining to a particular file Block is linked to each other as a linked list.
- The File system hierarchy follows a similar hierarchy as Phase 1 overall, to which the above shown abstractions have been added for simulating a virtual file system on lines of the Linux File System.
- Initial number of inodes in the inode table has been assumed to 25. The entire inode block contains the above shown entries. However in case more than 25 files are required, we may request the creation of additional block for inode entries and allocate more inodes dynamically as per requirement.
- The Inode blocks are also linked to each other as a doubly linked list so that they can be conveniently traversed for allocation/deallocation of inodes.
- Total file system size has been initially assumed to be approximately 8.2 MB, which may be expanded by tweaking the maximum size parameter before the creation of the storage disk abstraction.
- We have assumed two super blocks in the file system abstraction to ensure protection from dangers associated with corrupted super blocks as the super block contains some key information pertaining to the file system and this is where the reading of the file system begins during the time of mounting of the file system.
- Hence we can check for corruption in our file system or super block by comparing the values in the super blocks at both the ends and crash the file system if the parameters appear to vary in both of them.
- Each directory may have a maximum of 14 directory entries, which may be tuned as per requirement.
- Initially, each file or directory is allocated only one data block. If needed, additional blocks may be allocated for supporting larger files and these blocks are all linked as a linear linked list data structure.
- Every data block contains a structure written at the extreme end of the block. This disk structure contains the type of the block

(whether it represents a data block of a regular file, directory or is a special inode block). It also contains the block number of the next block (-1 if none) belonging to the same file.

- In case of inode blocks, the structure also stores the previous block number to make it a doubly linked list.
- Each super block consists of the following :
  - ➔ A Magic number or key code to represent the file system type
  - ➔ A data bit map of 2046 bytes
  - ➔ Number of free blocks and Total number of blocks

- The inode numbers assigned follow the following formula to compute the inode number using block number and the inode number (between 1-25) as present in the inode tables as follows :

Actual Inode Number = virtual Inode number(1 to25)*10000+block no of inode block.

SYSTEM CALLS IMPLEMENTED IN PHASE II :

The Fuse functions which were implemented to simulate actual system calls are as follows :
In contrast to the previous Phase I implementation, we employ our own file system abstractions and disk emulator to implement the various system calls.

Sl No.	SYSTEM CALLS	FUNCTIONALITY
OPERATIONS INVOLVING DIRECTORIES		
1.	mkdir( )	Creates a new directory with name specified from the given path by adding the “path” to the directory table with mode specified by the passes parameter, the default mode being 777.
2.	readdir( )	Reads the entries in a directory and fills the buffer provided by the caller.
3.	rmdir( )	It removes the given directory, provided it is empty.
OPERATIONS INVOLVING REGULAR FILES		
4.	mknod( )	Creates a special file with the given name and of type specified in the mode parameter passed.
5.	open( )	Opens a file in the given path.
7.	rename( )	renames a file to the parameter specified and internally calls rmdir(for a directory) and unlink( ) (for a regular file).
8.	read( )	reads from a file and stores the bytes read into a buffer.
9.	write( )	writes “size” bytes of data starting from the given “offset”, specified from the buffer onto the file.
10.	getattr( )	Initializes the default attributes of a file or a directory.
11.	link( )	Create a hard link between two files specified by its arguments.
12.	readlink( )	If path is a symbolic link, fill bufferwith its target, up to size.
13.	unlink( )	Removes (deletes) the given file, symbolic link, hard link, or special node. Note that if unlink only deletes the data when the *last* hard link is removed.
14.	truncate( )	Truncate or extend the given file so that it is precisely size bytes long.

PHASE III :

GOAL :

In this phase, We port the file system to secondary memory and ensure persistence of the file system and preserve its state across reboots.

ACHIEVING PERSISTENCE :

For achieving persistence , the entire file system allocated in memory was written to a file. During initialization of the file system, if there wasn’t any previously existing file system to restore, then a new file system is created with a new root node, otherwise the previously created binary file which serves as our disk emulator and stores the file system structure is read into a memory and further operations are performed using the disk library implemented in phase II.

In order to achieve this task, we use the previously created a disk emulator for persisting every file operations we perform. To do this, we create a large binary file of 8.2 MB by zero filling of the file. This single large UNIX file is used to emulate the disk or secondary storage since it is not feasible to use an actual raw disk for this purpose.

Hence we implement our own disk emulation library to simulate disk operations. The library functions implemented are capable of dividing the file created into blocks of size 4kB each and perform open, read and write operations with blocks into this virtual disk. This will be the underlying support for the whole project and act as a disk interface for the user performed file operations to run on top of this layer.

The details of the above mentioned disk emulator has already been elaborated in phase II and we have only provided an overview here.

Whenever any change is made to the filesystem such as: Creation of a file, Writing into a file, Appending to a file, creation of a directory, deletion of a directory and deletion of a file the update made in memory is immediately written into the disk.

In case of other operations such as reading a file etc, no change is made and hence the disk isn’t updated, to ensure efficiency and reduce effective execution time by avoiding writing of redundant changes or operations performed with the fie system.

TESTING OF OUR FILE SYSTEM :

The set of test cases for ensuring persitence in all forms and testing the functionalities implemented are as follws :

S. No	Desctiption	Commands	Expected Output	Remarks	
1	Create Directories			Use mkdir and cd commands	
2	Create File	echo "Hello World1" > TestFile1	TestFile1 created	In Root Directory	
3	Check opened file	ls -l Testfile1; cat TestFile1	"Hello World1" Text output		
4	Append to an existing file	echo "Hello World2" >> TestFile1	Text Appended		
5	Copy file	cp TestFile1 TestFile2	File copied		
6	Check copied file	ls -l TestFile2; cat TestFile2	Same as TestFile1		
7	Copy one directory below	cp TestFile1 TestDir1/TestFile3	Copied		
8	Check copied file	ls -l TestDir1/TestFile3; cat TestDir1/TestFile3	Same as TestFile1		
9	Create TestFile4	cd TestDir2; echo "Test4" > TestFile4	file created		
10	Check created file	ls -l testFile4; cat TestFile4	"Test4" text output		
11	Create duplicate file	use an editor to create TestFile4	Error Message		
12	Copy File	cp TestFile4 ../TestDir1/TestDir3/TestFile5	File copied		
13	Check copied file	ls -l ../TestDir1/TestDir3/TestFile5; cat ../TestDir1/TestDir3TestFile5	"Test4" text output		
14	Remove a file	rm TestFile4	File removed		
15	Try to remove a nonempty directory	cd ..; rmdir TestDir1	Error Message		
15	Remove an empty directory	rmdir TestDir2	Directory removed		
16	Unmount the file system		FS unmounted		
17	Remount the file system		FS remounted		
18	Check for files	ls -lR	No files	Files not saved	

Phase I

S. No	Desctiption	Commands	Expected Output	Remarks	
1	Create Directories			Use mkdir and cd commands	
2	Create File	echo "Hello World1" > TestFile1	TestFile1 created	In Root Directory	
3	Check opened file	ls -l Testfile1; cat TestFile1	"Hello World1" Text output		
4	Append to an existing file	echo "Hello World2" >> TestFile1	Text Appended		
5	Copy file	cp TestFile1 TestFile2	File copied		
6	Check copied file	ls -l TestFile2; cat TestFile2	Same as TestFile1		
7	Copy one directory below	cp TestFile1 TestDir1/TestFile3	Copied		
8	Check copied file	ls -l TestDir1/TestFile3; cat TestDir1/TestFile3	Same as TestFile1		
9	Create TestFile4	cd TestDir2; echo "Test4" > TestFile4	file created		
10	Check created file	ls -l testFile4; cat TestFile4	"Test4" text output		
11	Create duplicate file	use an editor to create TestFile4	Error Message		
12	Copy File	cp TestFile4 ../TestDir1/TestDir3/TestFile5	File copied		
13	Check copied file	ls -l ../TestDir1/TestDir3/TestFile5; cat ../TestDir1/TestDir3TestFile5	"Test4" text output		
14	Remove a file	rm TestFile4	File removed		
15	Try to remove a nonempty directory	cd ..; rmdir TestDir1	Error Message		
15	Remove an empty directory	rmdir TestDir2	Directory removed		
16	Create a file spanning 4 blocks	Write 1 to 512 using C/Python to TestFile2	File created		
17	Check block usage	du -s TestFile2	4 blocks should be used		
18	Reduce file size	Remove lines 101 onwards and save the file			
19	Check block size	du -s TestFile2	Should be still 4 blocks		
20	Add additional blocks	Add 101 to 1024 to TestFile2			
21	Check block size	du -s TestFile2	Block size should still be 4		
22	Unmount the file system		FS unmounted		
23	Remount the file system		FS remounted		
24	Check for files	ls -lR; cat "files";	**Files Intact (other than removed files/directories); verify contents of files**	Files in Secondary Storage Intact	

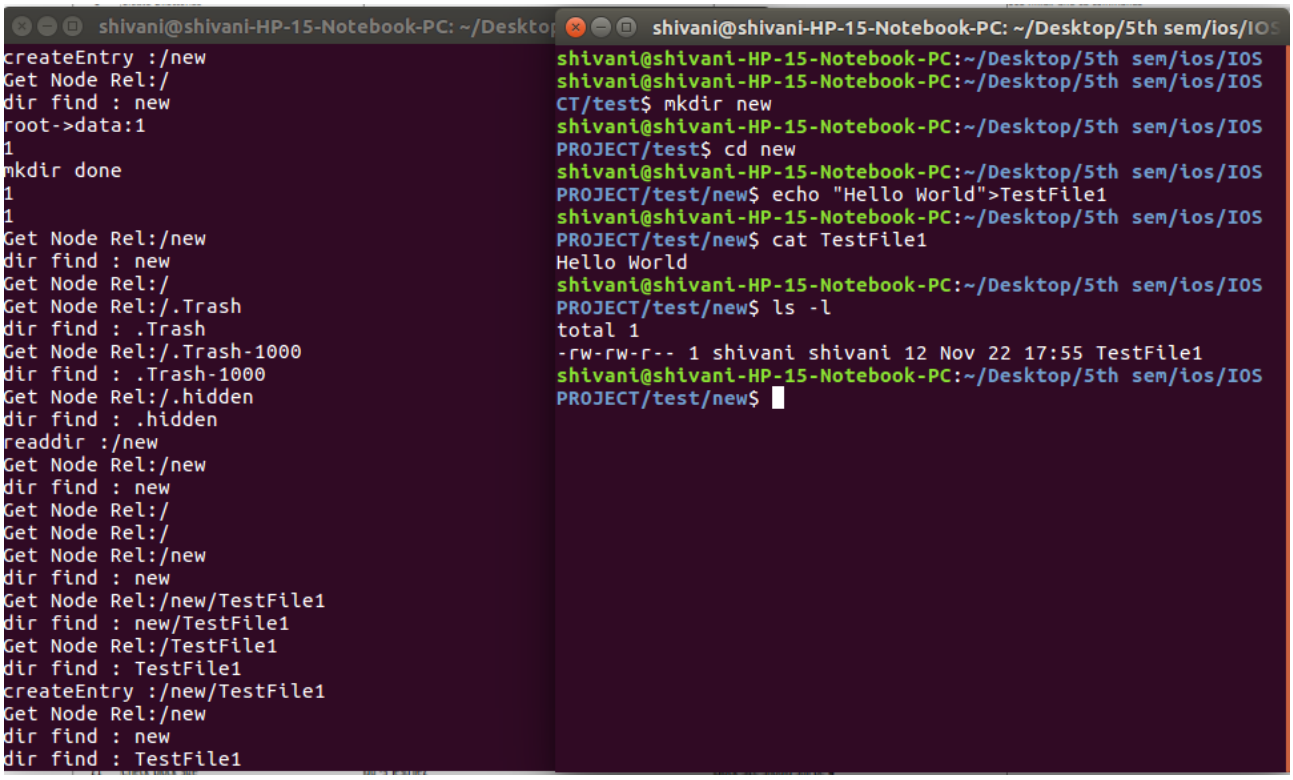
Phase 2

S. No	Desctiption	Commands	Expected Output	Remarks	
1	Create Directories			Use mkdir and cd commands	
2	Create File	echo "Hello World1" > TestFile1	TestFile1 created	In Root Directory	
3	Check opened file	ls -l Testfile1; cat TestFile1	"Hello World1" Text output		
4	Append to an existing file	echo "Hello World2" >> TestFile1	Text Appended		
5	Copy file	cp TestFile1 TestFile2	File copied		
6	Check copied file	ls -l TestFile2; cat TestFile2	Same as TestFile1		
7	Copy one directory below	cp TestFile1 TestDir1/TestFile3	Copied		
8	Check copied file	ls -l TestDir1/TestFile3; cat TestDir1/TestFile3	Same as TestFile1		
9	Create TestFile4	cd TestDir2; echo "Test4" > TestFile4	file created		
10	Check created file	ls -l testFile4; cat TestFile4	"Test4" text output		
11	Create duplicate file	use an editor to create TestFile4	Error Message		
12	Copy File	cp TestFile4 ../TestDir1/TestDir3/TestFile5	File copied		
13	Check copied file	ls -l ../TestDir1/TestDir3/TestFile5; cat ../TestDir1/TestDir3TestFile5	"Test4" text output		
14	Remove a file	rm TestFile4	File removed		
15	Try to remove a nonempty directory	cd ..; rmdir TestDir1	Error Message		
15	Remove an empty directory	rmdir TestDir2	Directory removed		
16	Create a file spanning 4 blocks	Write 1 to 512 using C/Python to TestFile2	File created	**Keep Blocksize 512 bytes**	
17	Check block usage	du -s TestFile2	4 blocks should be used		
18	Reduce file size	Remove lines 101 onwards and save the file			
19	Check block size	du -s TestFile2	Should be still 4 blocks		
20	Add additional blocks	Add 101 to 1024 to TestFile2			
21	Check block size	du -s TestFile2	Block size should still be 4		
22	Check for Persistence	While editing a file, reboot the machine; ls -lR; cat "files"	**Files Intact (other than removed files/directories and non saved portions of files); check contents of files**	Files in Secondary Storage Intact	

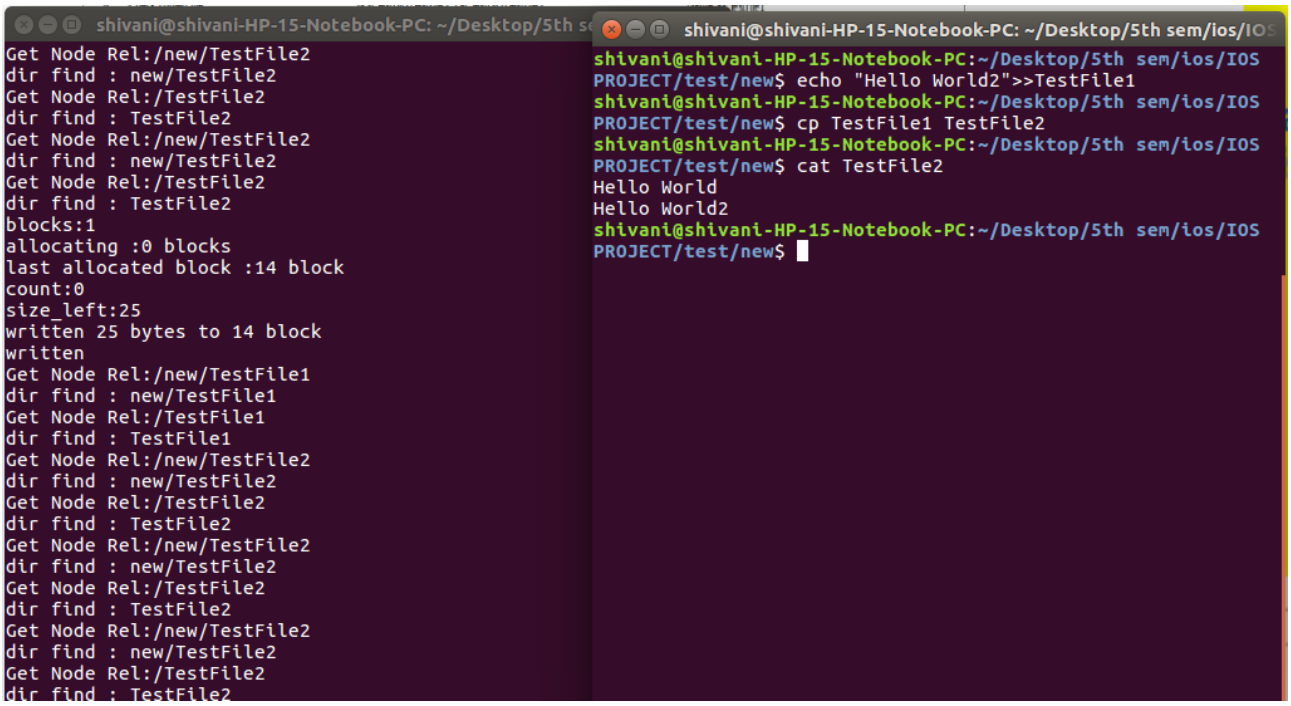
Phase 3 (Persistence)

RESULTS :

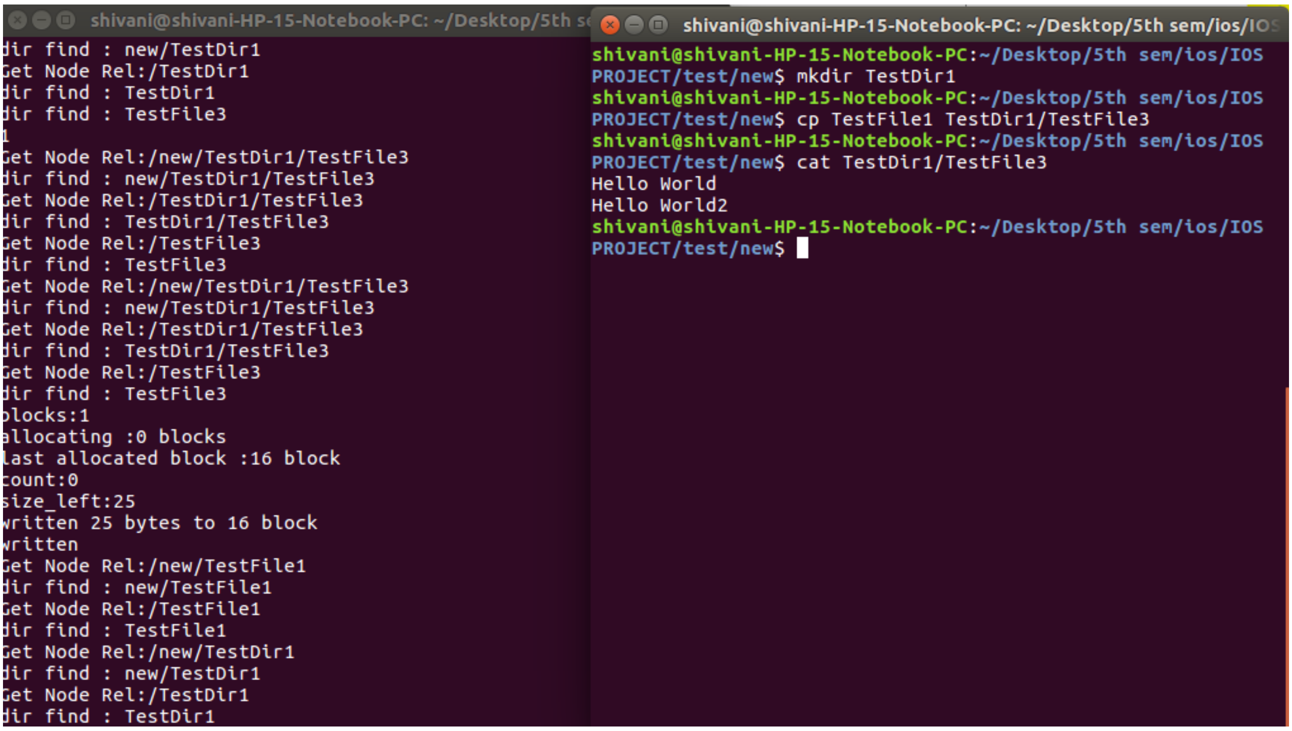
EXECUTION OF SYSTEM CALLS SPECIFIED BY ABOVE TEST CASES :



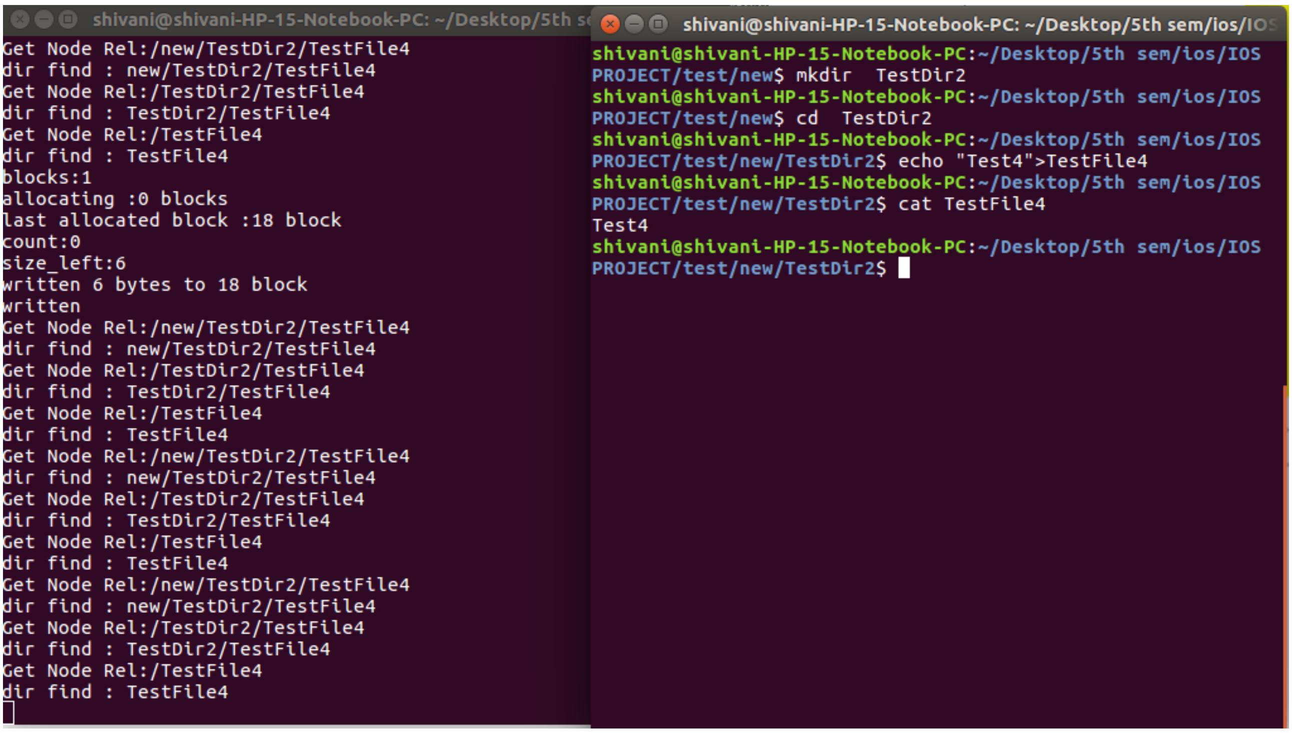
Creating directories, performing cd and creation of files



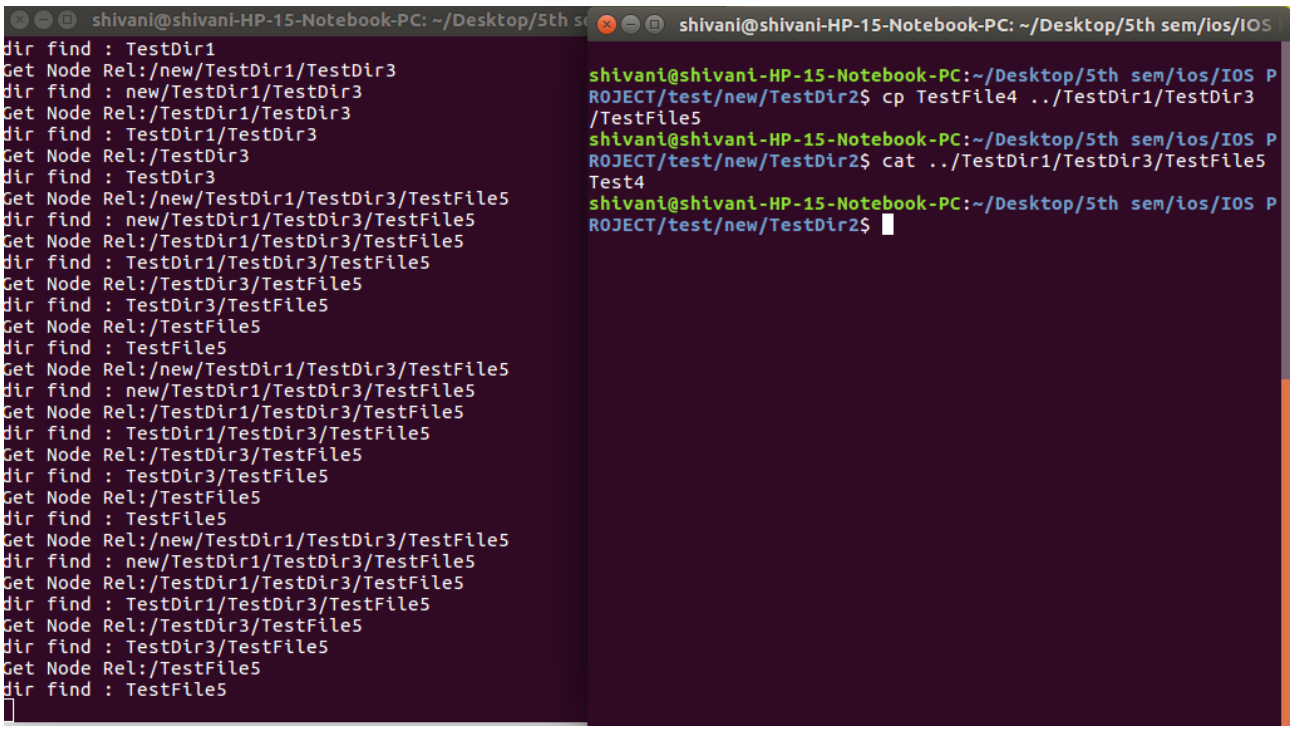
Appending and reading a file



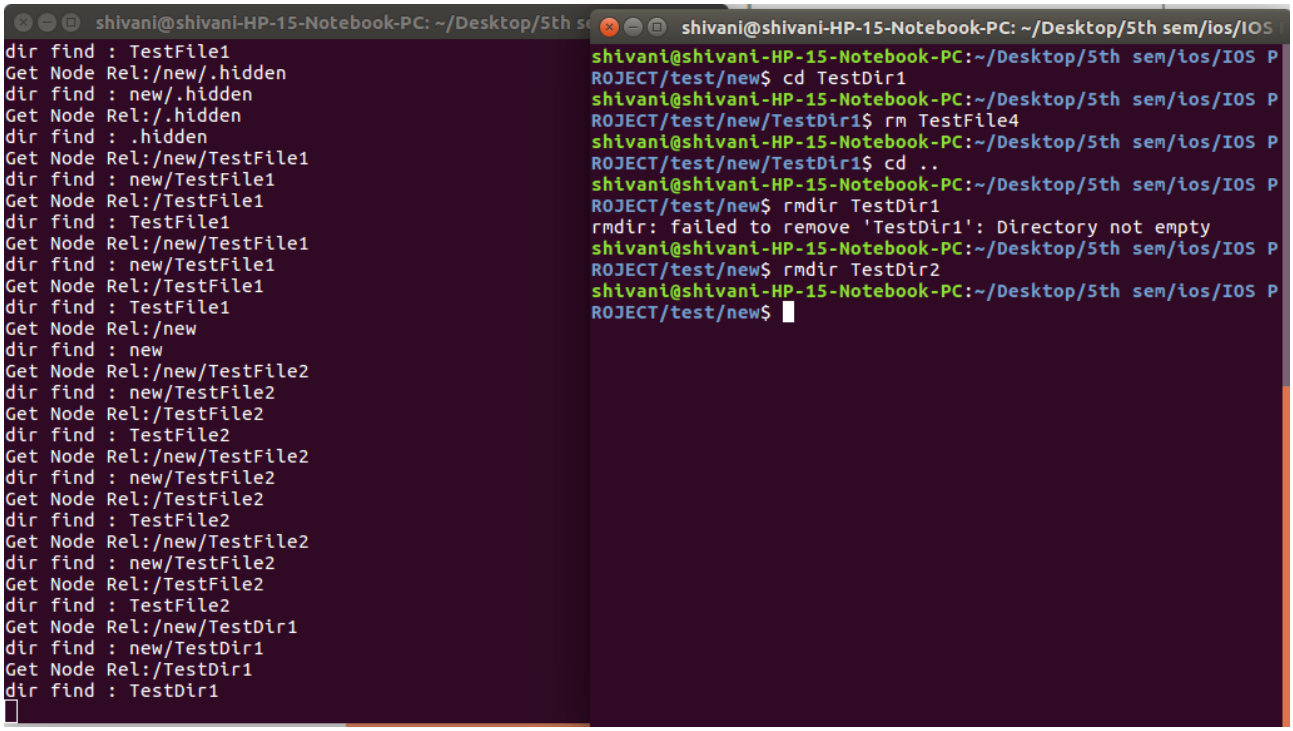
Copying a file into another file



Creating new files and directories



Copying newly created file into a new file inside a nested directory



Removing an empty directory