Micro File System

CIS 657 - Principles of Operating System

Lab M003 - Group 1

ABSTRACT

A file system is a component which stands between process and raw device and is responsible for efficiently organizing data on device. The existing file systems are complicated to understand and modify according to one's specific requirements especially for naive programmers. In this project, we develop a basic file system which can be extended with minimal efforts to serve such specific purposes - Micro-FS (MFS). MFS is a simple file system written in C for Linux Kernel 4.9.6.

Please visit the following link for the complete code: *https://github.com/Bhushan-Jagtap-2013/MicroFS*

1. INTRODUCTION

A file system is a set of abstract data types that are implemented for the storage, hierarchical organization, manipulation, navigation, access and retrieval of data. It is a Linux kernel module used to access files and directories. It provides access to this data for applications and system programs through consistent, standard interfaces exported by the VFS, and enables access to data that may be stored persistently on local media or on remote network servers/devices, or even transient data stored temporarily in RAM or special devices. Historically, Unix has provided four basic file system-related abstractions: files, directory entries, inodes and mount points.

A file system is a hierarchical storage of data adhering to a specific structure. File systems contain files, directories and associated control information. Typical operations performed on file systems are creation, deletion and mounting. In Unix, file systems are mounted at a specific mount point in a global hierarchy known as a *namespace*. This enables all mounted file systems to appear as entries in a single tree. Files are organized in directories. A *directory* is analogous to a folder and usually contains related files. Directories can also contain other directories, called subdirectories. In this fashion, directories may be nested to form paths. Each component of a path is called a *directory entry*. Unix systems separate the concept of a file from any associated information about it, such as access permissions, size, owner, creation time, and so on. This information is sometimes called *file metadata* (that is, data about the file's data) and is stored in a separate data structure from the file, called the *inode.*

All this information is tied together with the filesystem's own control information, which is stored in the *superblock*. The superblock is a data structure containing information about the file system as a whole. Sometimes the collective data is referred to as the *filesystem metadata*. File system metadata includes information about both the individual files and the file system as a whole [1-3].

Traditionally, Unix file systems implement these notions as part of their physical on disk layout. For example, file information is stored as an inode in a separate block on the disk; directories are files; control information is stored centrally in a superblock, and so on. The Unix file concepts are *physically mapped* on the storage medium. The Linux Virtual File System (VFS) is designed to work with filesystems that understand and implement such concepts.

In this work, we tried to implement a basic file system which can be extended with minimal efforts to serve such specific purposes named Micro-FS (MFS). MFS is a simple file system written in C for Linux Kernel 4.9.6. The aim of this project was to develop an extensible file system which will provide only basic functionalities. We hope that our file system would serve as a platform for building more feature-rich file systems providing , say, encryption, compression, etc.

In this report, we talk about the motivation for this project followed by related work from this area. Then we briefly describe the steps involved in the implementation and elaborate upon the implementation approach. Finally, we present some limitations, future scope and the conclusion of our project.

2. MOTIVATION

Our inspiration for this project was to learn how an actual file system is written for an operating system. We wanted to discover and get acquainted with the the challenges involved in writing a part of an OS and how those could be overcome. Moreover, we aim to release MFS for those who are relatively beginners in the field of systems so that they can build their own advanced file systems based on it.

The idea is to implement a bare-minimum file system recognized by Linux. It is primarily based upon the section 5 from our course - Storage Management. Specifically, it deals with File Systems Interface & Implementation.

3. RELATED WORK

Currently, there exist different file systems like ext2, ext3, ext4, etc., which are very mature and provide advanced features in addition to the primitive operations like create and delete file, but at the same time are difficult to alter for one's particular needs.

As with any other Linux file system, we have implemented our file system using the existing "Virtual File System". Moreover, we wrote MFS in C, making use of its advanced features like structures and pointers as and when needed in the project.

4. MFS DESIGN

MFS uses the abstraction, VFS, to interact with the connected devices. Client applications access the connected devices uniformly through MFS. Fortunately, VFS makes it easy to add support for new file system types to the kernel simply by fulfilling its contract.

4.1 MFS OBJECTS

Just like VFS, a family of data structures represents the common file model in MFS. The four primary object types of the MFS are as follows:

1. **The *superblock* object**, which represents a specific mounted filesystem. This object usually corresponds to the *filesystem superblock* or the *filesystem control block*, which is stored in a special sector on disk.

2. **The *inode* object**, which represents a specific file. The inode object represents all the information needed by the kernel to manipulate a file or directory. The structure *inode_operations* describes the filesystem's implemented functions that the VFS can invoke on an inode. We implemented inode in order to handle file creation and deletion operations. A *file* structure instance represents an open instance of an inode (for a pathname, with open flags). We can create files, write data and read data and perform many other common file operations. The inode object is represented by the structure *inode* and inode operations are defined in *linux/vfs.h*.

3. **The *dentry* object,** which represents a directory entry. It is a specific single component in a path. Dentry objects are represented by the structure *dentry* and is defined in *linux/dcache.h*.

4. **The *file* object,** which represents an open file as associated with a process. The file object is used to represent a file opened by a process. The file object is represented by the structure *file* and is defined in *linux/vfs.h*.

4.1 MFS OPERATIONS

Some of the implemented operations that are specified by these objects are as follow:

- **mfs_iget:** Returns the inode from cache, if present, else reads it from the device and then returns the same.
- **mfs_alloc_inode:** Called to get in-memory inode from get_locked.
- **mfs_destroy_inode:** Deallocates the given inode.
- **mfs_write_inode:** Writes the given inode to disk. The wait parameter specifies whether the operation should be synchronous.
- **mfs_drop_inode:** Called by the VFS when the last reference to an inode is dropped. Normal Unix filesystems do not define this function, in which case the VFS simply deletes the inode.
- **mfs_fill_super:** Fills the super block information from device.

4.2 DISK LAYOUT

MFS requires the devices to be block devices providing buffered access. In other words, MFS accesses the data on devices in blocks, each of size 1024 bytes or 1k. The whole disk can be then viewed as a number of blocks of fixed size 1k each. Following is the layout that MFS uses to mount the data structures and files onto the disk:

Block No.	**Bytes on disk**	**Usage**
0	0 - 1024	Super block
1	1024 - 2048	inode usage map
2	2048 - 3072	Data block usage map
3 - 66	3072 - 68608	inode structure array (ilist)
67 onward	68608 onward	Data blocks

Table 1: MFS layout

4.3 FEATURES

- Supports up to 1024 blocks of size 1024 bytes each
- Supports directory name of up to 14 characters
- Supports maximum 64 directory entries
- Allows creation and removal of up to 1024 files
- Maintains time of creation, deletion and modification of files
- Supports access permissions (Read - Write - Execute - owner)

5. IMPLEMENTATION APPROACH

Figure 1 below shows how a Linux application accesses the device through a file system.

VFS is a layer which provides uniform access to devices irrespective of the type of file system on it. It achieves this by exposing predefined functions which the application can use as per its requirement.

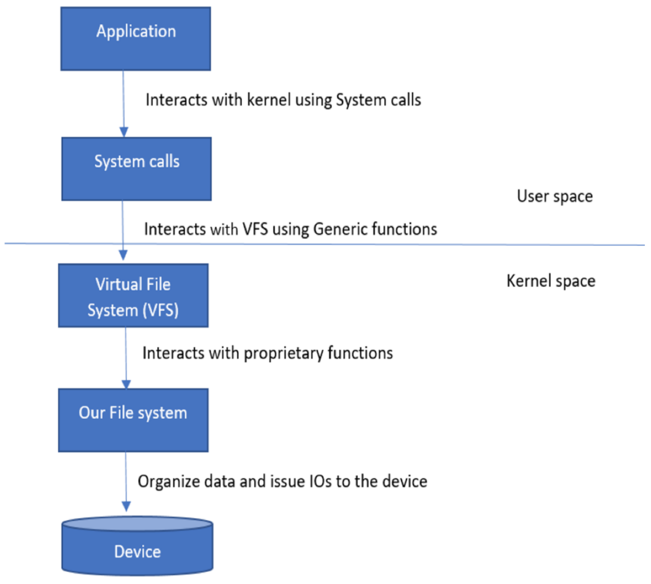

Application
Interacts with kernel using System calls
System calls
User space
Interacts with VFS using Generic functions
Virtual File System (VFS)
Kernel space
Interacts with proprietary functions
Our File system
Organize data and issue IOs to the device
Device

Figure 1: High-level view of the involved components

We therefore provide implementation for these interfaces from the file system point of view. Following is a high-level classification of the interfaces we implemented:

1. Super block operations

2. Inode operations
3. Directory operations

Apart from these interfaces, MFS is also responsible for maintaining the metadata (important information regarding the organization of data present on device). Following is a high-level list of the structures we implemented for organizing the data:

1. Super block structure - information regarding the overall file system
2. Inode - information regarding files
3. Inode list - information regarding free or used file inodes
4. Free block list - information regarding blocks which are free or available
5. Directory entry - information regarding mapping between files and inode

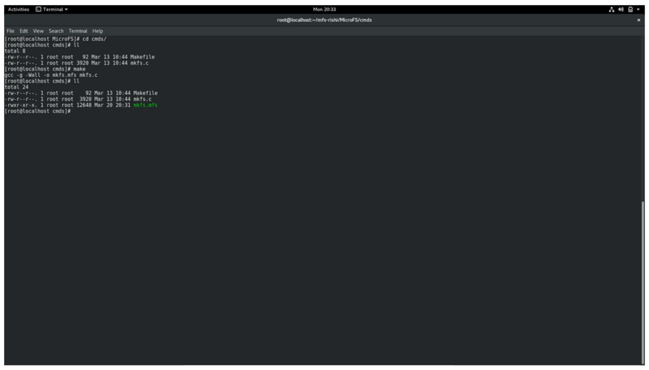
We began by deciding the structures of inode, ilist, directories, super block and free block list. Then we performed several steps as part of implementation consisting that of:

- Utility to create MFS on disk (mkfs.mfs)
- Debugging utility to print data structures
- Function to read superblock
- Module to register/unregister a file system with Linux
- Functions to create and delete files

6. DEMONSTRATION

Step 1: 'make' the mkfs

1. On a terminal console, go to the 'cmds' directory in the source directory of the MFS project (here, MicroFS/cmds)
2. Run 'make' to generate the 'mkfs.mfs' file

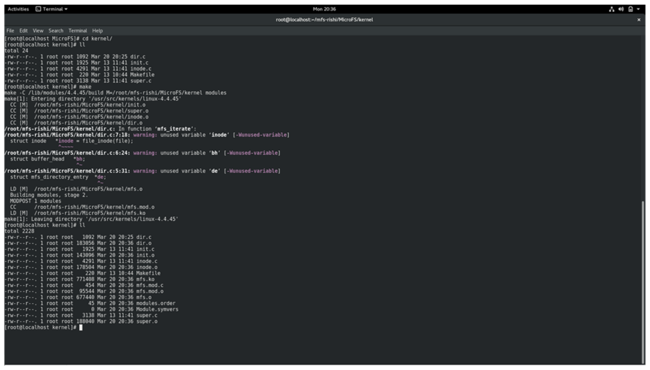


```
[root@localhost MicroFS]# cd cmds/
[root@localhost cmds]# ll
total 8
-rw-r--r--. 1 root root   92 Mar 13 10:44 Makefile
-rw-r--r--. 1 root root 3920 Mar 13 10:44 mkfs.c
[root@localhost cmds]# make
gcc -g -Wall -o mkfs.mfs mkfs.c
[root@localhost cmds]# ll
total 24
-rw-r--r--. 1 root root    92 Mar 13 10:44 Makefile
-rw-r--r--. 1 root root  3920 Mar 13 10:44 mkfs.c
-rwxr-xr-x. 1 root root 12648 Mar 20 20:31 mkfs.mfs
[root@localhost cmds]#
```

Figure 2: Making mkfs

Step 2: 'make' the filesystem kernel module 'mfs.ko'

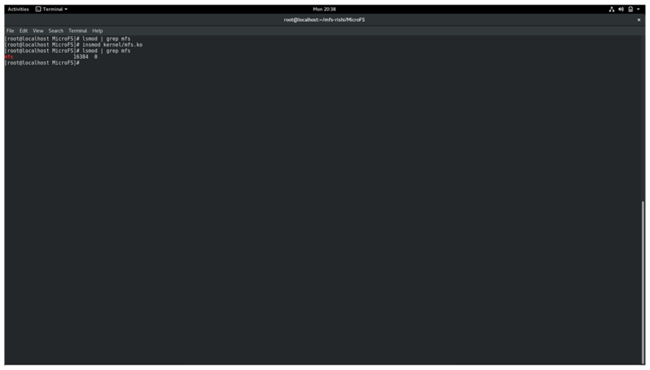
1. Navigate to the 'kernel' directory in the source directory (here, MicroFS/kernel)
2. Run 'make' to generate the 'mfs.ko' file and some other required binary (.o) files



```
[root@localhost MicroFS]# cd kernel/
[root@localhost kernel]# ll
total 24
-rw-r--r--. 1 root root 1092 Mar 20 20:25 dir.c
-rw-r--r--. 1 root root 1925 Mar 13 11:41 init.c
-rw-r--r--. 1 root root 4291 Mar 13 11:41 inode.c
-rw-r--r--. 1 root root  220 Mar 13 10:44 Makefile
-rw-r--r--. 1 root root 3138 Mar 13 11:41 super.c
[root@localhost kernel]# make
make -C /lib/modules/4.4.45/build M=/root/mfs-rishi/MicroFS/kernel modules
make[1]: Entering directory '/usr/src/kernels/linux-4.4.45'
  CC [M]  /root/mfs-rishi/MicroFS/kernel/init.o
  CC [M]  /root/mfs-rishi/MicroFS/kernel/super.o
  CC [M]  /root/mfs-rishi/MicroFS/kernel/inode.o
  CC [M]  /root/mfs-rishi/MicroFS/kernel/dir.o
/root/mfs-rishi/MicroFS/kernel/dir.c: In function 'mfs_iterate':
/root/mfs-rishi/MicroFS/kernel/dir.c:7:18: warning: unused variable 'inode' [-Wunused-variable]
  struct inode   *inode = file_inode(file);
                  ^~~~~
/root/mfs-rishi/MicroFS/kernel/dir.c:6:24: warning: unused variable 'bh' [-Wunused-variable]
  struct buffer_head   *bh;
                        ^~
/root/mfs-rishi/MicroFS/kernel/dir.c:5:31: warning: unused variable 'de' [-Wunused-variable]
  struct mfs_directory_entry  *de;
                               ^~
  LD [M]  /root/mfs-rishi/MicroFS/kernel/mfs.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /root/mfs-rishi/MicroFS/kernel/mfs.mod.o
  LD [M]  /root/mfs-rishi/MicroFS/kernel/mfs.ko
make[1]: Leaving directory '/usr/src/kernels/linux-4.4.45'
[root@localhost kernel]# ll
total 2228
-rw-r--r--. 1 root root   1092 Mar 20 20:25 dir.c
-rw-r--r--. 1 root root 183056 Mar 20 20:36 dir.o
-rw-r--r--. 1 root root   1925 Mar 13 11:41 init.c
-rw-r--r--. 1 root root 143096 Mar 20 20:36 init.o
-rw-r--r--. 1 root root   4291 Mar 13 11:41 inode.c
-rw-r--r--. 1 root root 178504 Mar 20 20:36 inode.o
-rw-r--r--. 1 root root    220 Mar 13 10:44 Makefile
-rw-r--r--. 1 root root 771408 Mar 20 20:36 mfs.ko
-rw-r--r--. 1 root root    454 Mar 20 20:36 mfs.mod.c
-rw-r--r--. 1 root root  95544 Mar 20 20:36 mfs.mod.o
-rw-r--r--. 1 root root 677440 Mar 20 20:36 mfs.o
-rw-r--r--. 1 root root     45 Mar 20 20:36 modules.order
-rw-r--r--. 1 root root      0 Mar 20 20:36 Module.symvers
-rw-r--r--. 1 root root   3138 Mar 13 11:41 super.c
-rw-r--r--. 1 root root 188040 Mar 20 20:36 super.o
[root@localhost kernel]# 
```

Step 3: Insert the module in the linux kernel

1. From the current directory (here, MicroFS), run 'insmod <path_to_mfs.ko>' (here, MicroFS/kernel/mfs.ko)

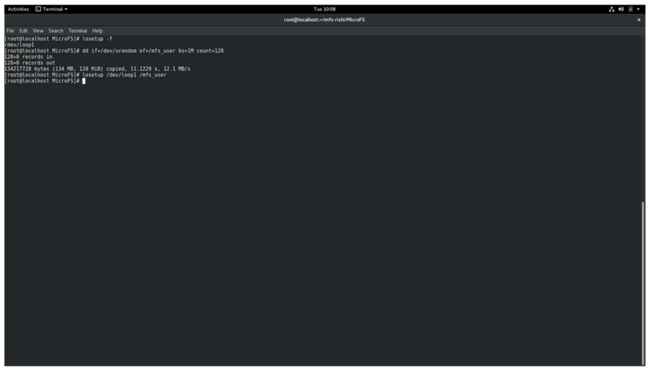


```
[root@localhost MicroFS]# lsmod | grep mfs
[root@localhost MicroFS]# insmod kernel/mfs.ko
[root@localhost MicroFS]# lsmod | grep mfs
mfs                    16384  0
[root@localhost MicroFS]#
```

Figure 3: Making the kernel module

Step 4: Create a loop device

1. Run 'losetup -f' to find the first available loop device (here, /dev/loop1)
2. Create a block file '/mfs_user' (for mounting the MFS later onto) and write 128 blocks (1M bytes at a time) to it using the pseudo-random number generator /dev/urandom with the following command: *dd if=/dev/urandom of=/mfs_user bs=1M count=128*
3. Setup the device '/dev/loop1' to use the newly created '/mfs_user' block file with the following command: *losetup /dev/loop1 /mfs_user*

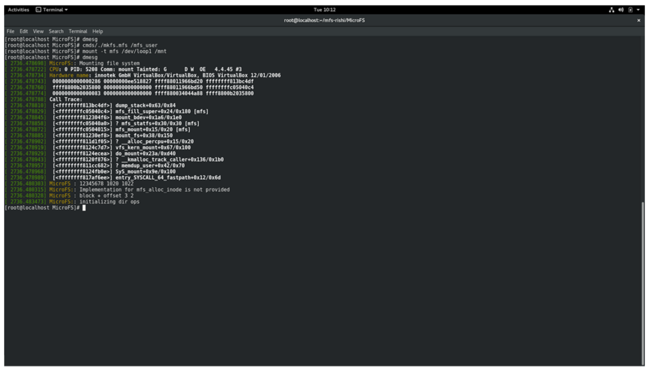


```
[root@localhost MicroFS]# losetup -f
/dev/loop1
[root@localhost MicroFS]# dd if=/dev/urandom of=/mfs_user bs=1M count=128
128+0 records in
128+0 records out
134217728 bytes (134 MB, 128 MiB) copied, 11.1229 s, 12.1 MB/s
[root@localhost MicroFS]# losetup /dev/loop1 /mfs_user
[root@localhost MicroFS]# 
```

Figure 4: Creating a loop device

Step 5: Mounting MFS onto the device

1. From the 'cmds' directory visited earlier, run './mkfs.mfs /mfs_user' to make the filesystem on the device
2. Now run 'mount -t mfs /dev/loop1 /mnt' to mount the device '/dev/loop1' onto the directory location '/mnt'

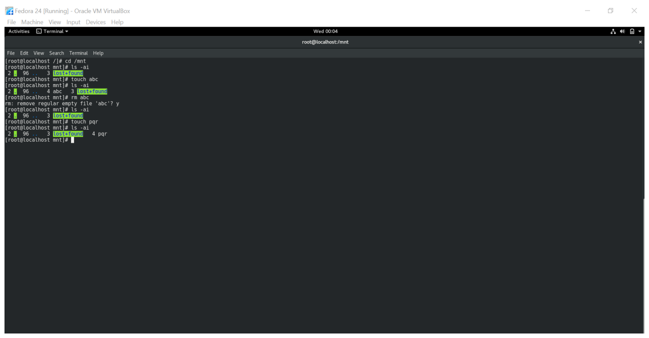


```
[root@localhost MicroFS]# dmesg
[root@localhost MicroFS]# cmds/./mkfs.mfs /mfs_user
[root@localhost MicroFS]# mount -t mfs /dev/loop1 /mnt
[root@localhost MicroFS]# dmesg
[ 2736.478698] MicroFS:: Mounting file system
[ 2736.478722] CPU: 0 PID: 5208 Comm: mount Tainted: G      D W  OE   4.4.45 #3
[ 2736.478734] Hardware name: innotek GmbH VirtualBox/VirtualBox, BIOS VirtualBox 12/01/2006
[ 2736.478743]  0000000000000286 00000000ee518827 ffff88011966bd20 ffffffff813bc4df
[ 2736.478760]  ffff8800b2035800 0000000000000000 ffff88011966bd50 ffffffffc05040c4
[ 2736.478774]  0000000000000083 0000000000000000 ffff880034044a88 ffff8800b2035800
[ 2736.478788] Call Trace:
[ 2736.478810]  [<ffffffff813bc4df>] dump_stack+0x63/0x84
[ 2736.478829]  [<ffffffffc05040c4>] mfs_fill_super+0x24/0x180 [mfs]
[ 2736.478845]  [<ffffffff812304f6>] mount_bdev+0x1a6/0x1e0
[ 2736.478858]  [<ffffffffc05040a0>] ? mfs_statfs+0x30/0x30 [mfs]
[ 2736.478872]  [<ffffffffc0504015>] mfs_mount+0x15/0x20 [mfs]
[ 2736.478885]  [<ffffffff81230ef8>] mount_fs+0x38/0x150
[ 2736.478902]  [<ffffffff811d1f05>] ? __alloc_percpu+0x15/0x20
[ 2736.478919]  [<ffffffff8124c7d7>] vfs_kern_mount+0x67/0x100
[ 2736.478929]  [<ffffffff8124ecea>] do_mount+0x23a/0xd40
[ 2736.478943]  [<ffffffff8120f876>] ? __kmalloc_track_caller+0x136/0x1b0
[ 2736.478957]  [<ffffffff811cc682>] ? memdup_user+0x42/0x70
[ 2736.478968]  [<ffffffff8124fb0e>] SyS_mount+0x9e/0x100
[ 2736.478989]  [<ffffffff817af6ee>] entry_SYSCALL_64_fastpath+0x12/0x6d
[ 2736.480303] MicroFS : 12345678 1020 1022
[ 2736.480315] MicroFS:: Implementation for mfs_alloc_inode is not provided
[ 2736.480328] MicroFS : block + offset 3 2
[ 2736.483473] MicroFS:: initializing dir ops
[root@localhost MicroFS]#
```

Figure 5: Mounting the file system

Step 6: Creation and Deletion of files

1. On a terminal console, go to the '/mnt' directory. List all the files along with inode numbers using the command *"ls -ai"*.
2. Fire command *"touch <file name>"* (here, *touch abc*) to create new file. Fire *"ls -ai"* again to see the new file with its inode number, in this case it is *'4 abc'*.
3. Now, fire command *"rm <file name>"* (here, *rm abc*) to delete the existing file. After running *"ls -ai"*, file *abc* will not be listed anymore.
4. Add another file and fire *"ls -ai"*, you will notice that the new file is created with the freed inode number 4.



```
[root@localhost /]# cd /mnt
[root@localhost mnt]# ls -ai
 2 .   96 ..    3 lost+found
[root@localhost mnt]# touch abc
[root@localhost mnt]# ls -ai
 2 .   96 ..    4 abc   3 lost+found
[root@localhost mnt]# rm abc
rm: remove regular empty file 'abc'? y
[root@localhost mnt]# ls -ai
 2 .   96 ..    3 lost+found
[root@localhost mnt]# touch pqr
[root@localhost mnt]# ls -ai
 2 .   96 ..    3 lost+found   4 pqr
[root@localhost mnt]#
```

Figure 6: Create and Delete files

7. PERFORMANCE ANALYSIS

We compared the create and delete operations of MFS its counterparts from the popular Ext4 file system. In particular, we wrote and ran a script in bash to create a bunch of 60 files and measured the time it required using the *time* command. We did this once for both the file systems multiple times. Figure 6 shows the results of our tests.

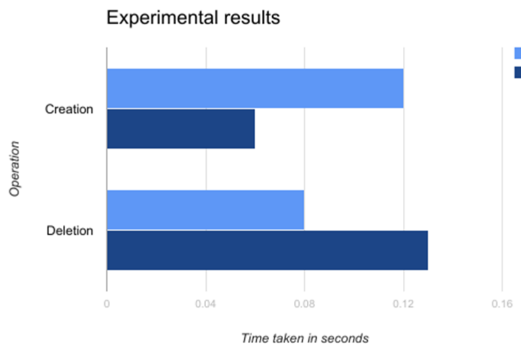


Figure 6: Performance comparison

From the above results, it can be observed that MFS performs approximately twice as better in file creation as compared to Ext4. On the other hand, the file deletion gives better results on Ext4, though it is less worse than twice in case of MFS. We hypothesize that despite the simple organization of MFS, the added lookup for the supplied *inode* during deletion adds the overhead.

8. LIMITATIONS

Being a bare-minimum functional file system gives MFS both its perks as well as some major limitations which are explained below:

1. MFS does not provide/use any kind of paging/caching mechanism, which gives it its simplicity but restricts the efficiency.
2. The maximum number of data blocks MFS can operate upon is limited by its design itself and cannot be modified without modifying the code.
3. Currently, the code is written so as to write the *dentry* objects in a single block instead of multiple. Though adding the relevant code was no big hassle, it was a potential invitation to manual errors.

9. FUTURE SCOPE

While the limitations in themselves pose a significant scope for future work on this project, we particularly wish to experiment with and revamp the organization of MFS to improve its efficiency without losing simplicity.

10. CONCLUSION

We learnt how to build a file system for an operating system by attempting to develop a basic one that can be recognized by Linux and supports simple file operations - create and delete files. For the implementation, we only needed the Linux source code, VirtualBox and reference material while virtual (loop) devices sufficed for testing the file system. We believe to have answered to ourselves, the following questions:

1. What is a file system?
2. What is VFS?
3. What is a kernel module?
4. What are the data structures?

Further, we reached the goals that we defined earlier - Understanding the existing file systems, registering our file system and making it visible to Linux and writing functions to create and delete files.

We completed our file system by understanding and working with the VFS layer, interfaces, module programming and debugging mechanism of Linux. Moreover, we began by dividing the study material among our group of four and learning collaboratively followed by creating the mkfs and the debugging utilities. Finally, we wrote the functions for creating and deleting files.

11. REFERENCES

[1] Rodriguez, R.; M. Koehler; R. Hyde (June 1986). "The Generic File System". *Proceedings of the USENIX Summer Technical Conference*. Atlanta, Georgia: USENIX Association. pp. 260–269..

[2] Pate SD. UNIX filesystems: evolution, design, and implementation. John Wiley & Sons; 2003 Feb 17.

[3] Love R. Linux kernel development. Pearson Education; 2010 Jun 22.