Final Report: Kernel Level Persistent File System with Deduplication

Group Number: 1

Denil Vira, Nitin Tak, Pooja Routray, Sumeet Hemant Bhatia

CSC 568: Enterprise Storage Architecture

May 1, 2015

Abstract

This project builds a filesystem in the Linux kernel space that is deployed as a loadable kernel module. This filesystem is persistent, hierarchical and supports basic POSIX/Unix commands like readdir, create, mkdir, write, read, getattr, rmdir, unlink, lookup and statfs. The metadata is referred using the standard inode methodology. The file system provides deduplication using eager approach and fixed size blocks. The hash values for these fixed size blocks are computed using SHA-1 algorithm. This project provides an enterprise level filesystem as its outcome.

Introduction

The Linux kernel provides an abstraction in the form of a Virtual Filesystem (VFS) to allow multiple filesystems to coexist and interoperate. This enables applications to use standard Unix system calls to work with different filesystems as shown in Figure 1. VFS achieves this by multiplexing filesystem calls and invoking corresponding routine for each filesystem operation. VFS thus implements all the file and filesystem-related interfaces. Each deployed filesystem is required to provide certain interfaces and data structures that the VFS expects.

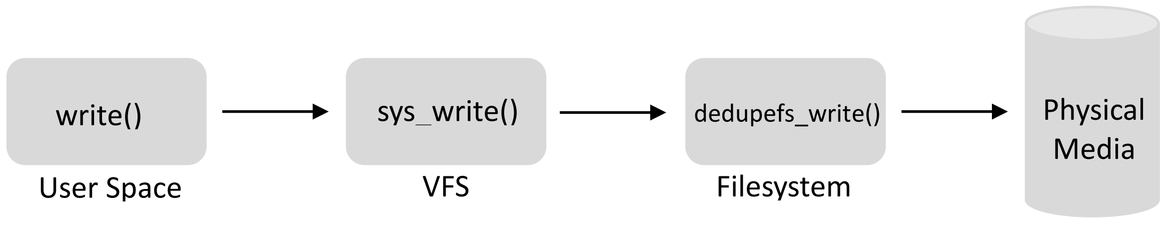


Figure 1: Filesystem Call

(Image Credits: Linux Kernel Development, Third edition, Robert Love)

Four basic abstractions supported by VFS are:

- *superblock* object: It provides information about a specific filesystem. This object corresponds to a filesystem superblock which resides on a special sector on the disk.
- *inode* object: It represents information required by the kernel to manipulate a file or directory
- *dentry* object: It represents a directory entry, which is a single component of a path
- *file* object: It represents an open file as associated with a process and is an in-memory representation of an open file

Each of these primary objects contain an operation object which describes the methods that the kernel invokes against them. The kernel uses two structures, *file_system_type and vfsmount*, to manage data related to filesytems. Each registered filesystem is represented by a *file_system_type* structure and *vfsmount* structure describes a specific filesystem instance or mount point. Two other structures, *fs_struct* and *file*, are used to describe the filesystem and files associated with each process.

An important distinction while working with filesystems is between block devices and character devices. Block devices (abbreviated as blkdevs) are addressable devices that are accessed randomly via a special file called block device node. Fixed chunks of data (called blocks) are read from block devices. Hard disks, flash memory, floppy disks, etc. are categorized as block devices. On the other hand character devices (abbreviated as cdevs) are non-addressable devices that are accessed through character device node file. Data in a character device is accessed as a sequential stream, one byte after another. Few examples include keyboard and serial ports [1].

This project implements a filesystem that provides specific instances of each of the structures that the VFS requires and provides users a fully functional filesystem. It is implemented on a block device which is specified using FS_REQUIRES_DEV flag of *file_system_type* structure. This implementation also addresses the issue of reducing disk footprint of the ever increasing data by implementing an intelligent compression technique known as deduplication.

Deduplication is a technique based on the concept of single-instance multi-reference storage as shown in Figure 2. It exploits data redundancy by eliminating the need for storing it repeatedly [2]. This significantly reduces the storage requirements and economizes storage space. It also leads to faster backup and better disaster recovery which eventually reduces the cost requirements to persist and maintain data. Deduplication implementation is computationally intensive and require more processing power. Thus there exists a tradeoff between optimizing disk storage and computational overhead.

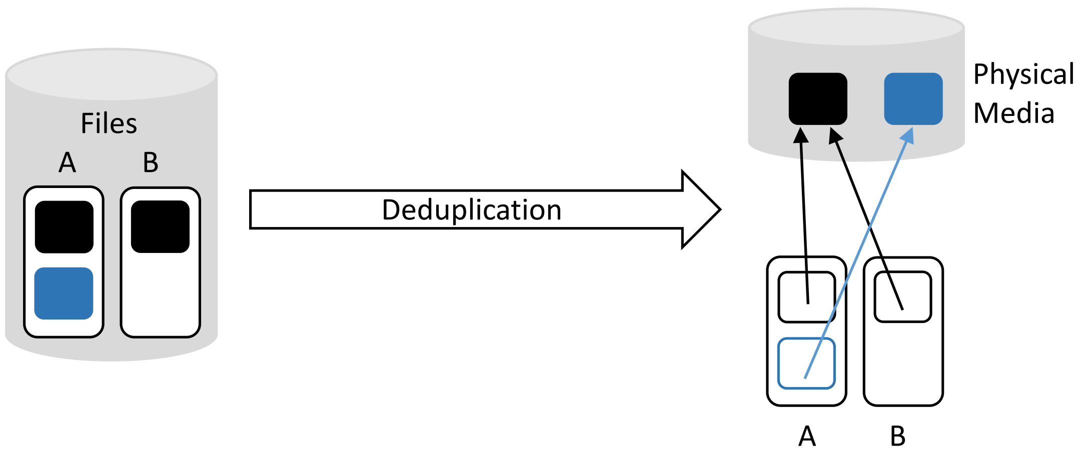


Figure 2: Deduplication

Deduplication can be deployed as either a source or a target based solution. When backups are pushed over the network to an appliance where deduplication takes place it is known as target based approach as the data is sent to some target for deduplication. If deduplication takes place at the origination of data itself, it is termed as source deduplication [3]. Source deduplication is generally provided at the file system level. This project provides source level deduplication to improve the efficacy of storage devices.

Another aspect of deduplication is chunking data using blocks of either fixed or variable sizes. Fixed length deduplication will break a file into subfiles of fixed length data segments. The primary drawback of this approach is when data in a file is shifted, for example, a user may rectify the spelling mistakes in a previously saved file. This rectification will cause all the subsequent blocks in the file to be rewritten and each subfile is likely to be considered different from those in the original file. Consequently deduplication effect is less significant. Variable length deduplication is an advanced approach which divides a file into subfiles based on their interior data patterns. This addresses the data shift problem of fixed size blocks [4].

Deduplication can either be performed when the data is being stored or it can be performed at periodic intervals. The approach in which the data is deduplicated right away before storing on to the disk is termed as eager deduplication. In contrast, lazy deduplication is a technique which involves deduplication of stored data at regular interval of time.

This project implements deduplication using fixed size blocks and eager approach. The data is divided in blocks of 4 KB size, which are referred to as chunks. Each chunk is processed using Secure Hash encryption Algorithm, SHA-1. SHA-1 can take input up to length 2^{64} to produce a 160-bit output termed as chunk ID. This chunk ID will uniquely identify the chunks used for deduplication [5].

Design

The kernel level file system is implemented using the inode methodology. The inodes structure describes the metadata of regular file and directories. Metadata consists of attributes like a unique inode number, creation time, last access time, inode type (file/directory), file size, number of sub-directories and files, and number of data blocks used by this inode. Inode also contains a list of data blocks associated with it. Depending on the file type, the blocks in this list either contain a list of all the files and/or sub-directories within that directory or data that composes the particular file. This meta-data information is stored at fixed location on the mounted device and is referred to as inode cache. The details like magic number, map of used and free blocks on the disk, number of inodes on disk, etc. describing the entire file system are stored in a superblock structure. A bit vector approach is used to maintain different mappings. This helps in achieving effective space utilization on the mounted device and optimal operational efficiency as all the operations were performed using bitwise operations.

A lookup data structure is maintained to map the path of a file to its corresponding inode. This is required to fetch the corresponding inode whenever any look up operations are involved. This data structure is also maintained as a list on disk. Each directory in the file system stores a list of the aforementioned structure in the data area associated with its inode. This abstraction separates the placement of data on the device from the metadata in case of a directory.

This file system is inserted as a pluggable kernel module using *insmod* command. When the file system is inserted in the kernel it invokes the *init* function wherein the file system is registered using *register_filesystem* call. The *register_filesystem* call takes a *file_system_type* object as its parameter which specifies the function that is to be invoked to mount the file system. The file system is then mounted on a block device using *mount_bdev* API call provided by the Linux kernel. The super block of this file system is filled up by the function that is invoked by a function pointer specified in *mount_bdev* routine [1]. The registration flow is summarized by Figure 3.

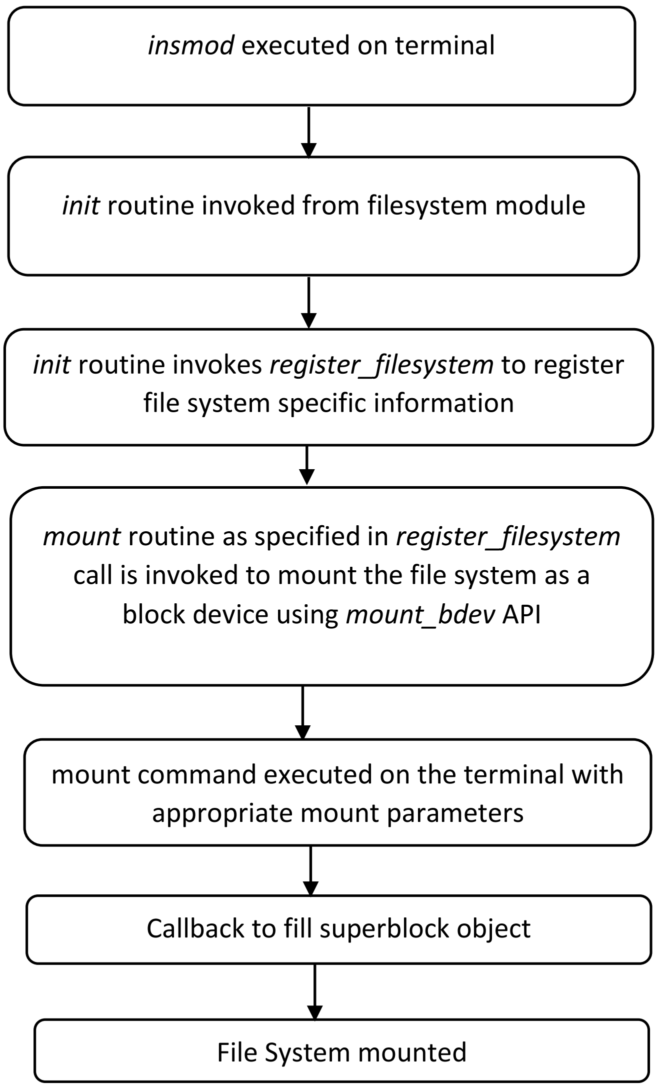

insmod executed on terminal
init routine invoked from filesystem module
init routine invokes *register_filesystem* to register file system specific information
mount routine as specified in *register_filesystem* call is invoked to mount the file system as a block device using *mount_bdev* API
mount command executed on the terminal with appropriate mount parameters
Callback to fill superblock object
File System mounted

Figure 3: Flowchart for registering a filesystem as a Linux kernel module

Deduplication is performed in an eager approach whenever the file is saved by the user. The file contents are divided into fixed size chunks of 4 KB. The 4 KB chunk size is used to leverage the benefit of clustering, thus, implicitly improving performance. The updates to a file are currently not supported by the implemented file system. However, the users can append to existing files. The append flow will invalidate the hash of the last block where the update is being performed and re-compute the hash of the new data and update the deduplication cache on the mounted device. The other alternative to perform deduplication is using a variable sized block approach. The variable sized approach is more effective in saving the disk space while dealing with arbitrary data patterns typically encountered in a data center. However, the main motive of this project is to highlight the fact that deduplication saves disk space for enterprise storage and hence the alternative of variable sized block was overlooked.

To implement deduplication, a separate data structure is created and maintained. This structure holds the chunk ID, the corresponding data block associated with this chunk ID and count of the number of references to it. Figure 4 depicts the implementation flow adopted for deduplication.

Reference count ensures that chunk id gets deleted only when there exists no inode referring to that chunk ID. This data structure is maintained as a part of file system metadata and is referred to as dedupe cache. This dedupe cache is stored in the blocks following the inode cache on the mounted device. To read this chunked data, each regular file inode maintains a file recipe in form of a list of data blocks numbers where each chunk is stored. Thus, the data block numbers act as the mapping between dedupe cache and the inode meta-data. Upon deletion of a file, the reference count of the data blocks associated with it is decremented. These data blocks are reclaimed only when the reference count reaches zero.

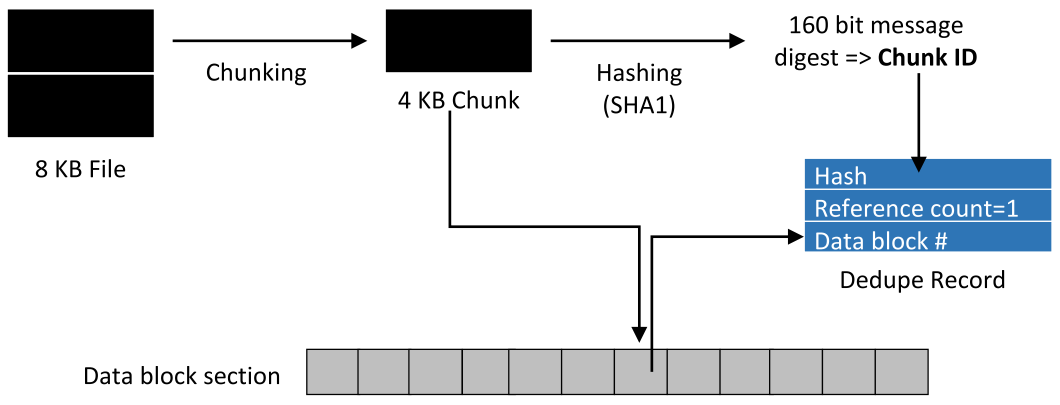


Figure 4: Implementation of deduplication

The chunk IDs are computed using a hash function. The hash function takes a chunk of 4 KB or less as its input and generate a unique fixed size message digest called as chunk ID. The design choices were between two widely used hash algorithms - Secure Hash Algorithm (SHA-1) and Message Digest 5 (MD5). SHA-1 produces 160 bit or 20 bytes fingerprint of any given input while MD5 produces a 128 bit fingerprint. SHA-1 is more secure than MD5 against brute force attacks. Also, it is practically impossible to produce same hash value for any two different inputs using SHA-1. Relevant research in this direction has proven that it takes approximately 2^{69} trials before a collision can occur in SHA-1[6] whereas it takes 2^{41} trials for a collision to occur in MD5 [7]. This is an important consideration for deduplication since a hash collision leads to a chunk ID referring to an incorrect chunk, thus, causing inconsistency while recreating the data. Linux kernel provides implementations of the SHA-1 and MD5 algorithms through its cryptographic library. The function definitions for SHA-1 were referred from <crypto.h> and < scatterlist.h> header file. This project employed SHA-1 as the hashing algorithm to obtain the chunk ID for fixed size chunks to leverage its dual advantage of robustness and better hash collision protection as opposed to MD5. The hash for the chunks was computed using a combination of *crypto_hash_init*, *crypto_hash_update*, *crypto_hash_final* functions.

Figure 5 illustrates the layout of metadata and data on the device.

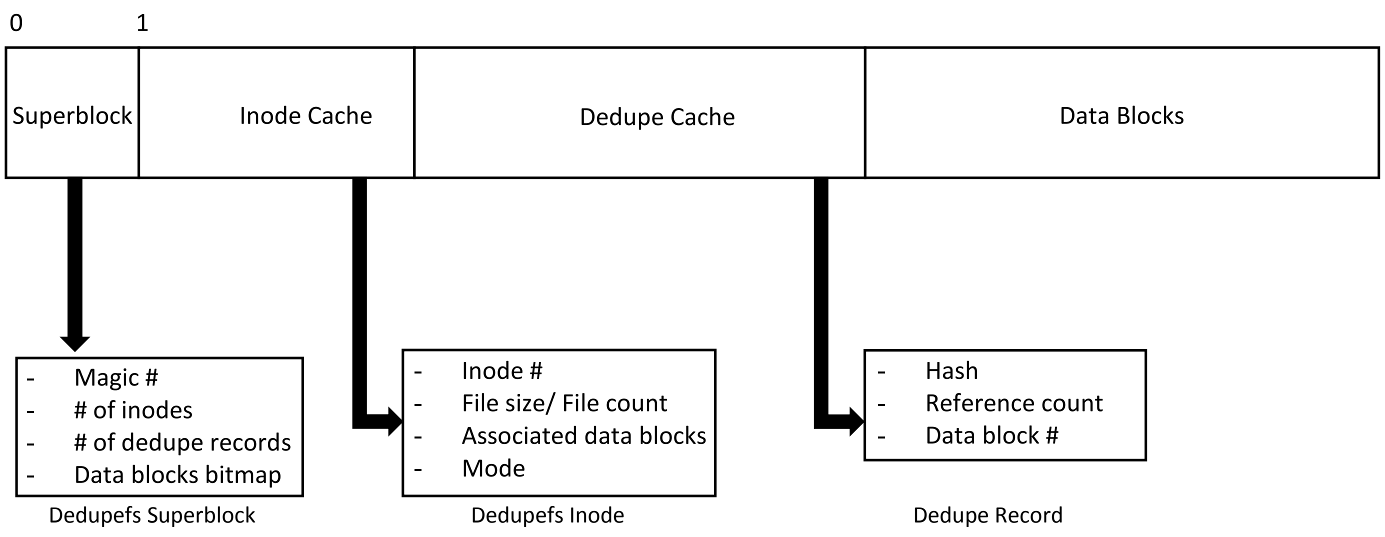


Figure 5: Layout on device

The metadata blocks are stored in fixed locations on the device. This design is required since file meta-data is to be read while mounting the file system. A fixed location on the device helps to read super-block and root inode contents when the file system is mounted. Since, the data and meta-data are written on the mounted device it implicitly handles the persistence feature.

The defects and kernel crashes in the module development had to be addressed. Debugging at the kernel level played a significant role in the analysis of these defects. Following debugging techniques were employed in the kernel space:

- *printk()* – This enabled printing error/warning message at the kernel log which were viewed using dmesg
- *BUG()* – This routine caused an oops, which resulted in a stack trace being printed
- *BUG_ON(conditional expression)* – This caused an oops when the Boolean condition specified by conditional expression was evaluated to true and caused the stack trace to be printed
- *dump_stack()* – This simply prints the current stack trace without causing a kernel oops

Other approaches like using certain configurable parameters as a conditional variable and executing certain code snippets only when those configurable parameters were set was used. These configurable parameters were passed while mounting the file system. This facilitated in easily narrowing down to the actual root cause of an issue without the need to reengineer the entire system [1].

Project Milestones

The development and testing efforts were sub-divided into the following tasks:

Serial No.	Description
1	Understanding of Linux kernel internals • VFS layer • Block IO Layer
2	Inserting, removing and debugging kernel modules • insmod • rmmod
3	Survey of existing Linux file systems • Log Structured File System • Write Anywhere File Layout
4	Analysis and implementation of basic data structures
5	Analysis and design of ADT required to form building blocks of kernel file system • Design of superblock and inode structure • Design of lookup data structure • Design of data structures for deduplication
6	Implementation of file system operations • Filesystem calls • Deduplication feature • Script automation for insertion, removal and testing of filesystem
7	Verification and validation • Testing the basic flows • Defect fixing

Table 1: Project Tasks

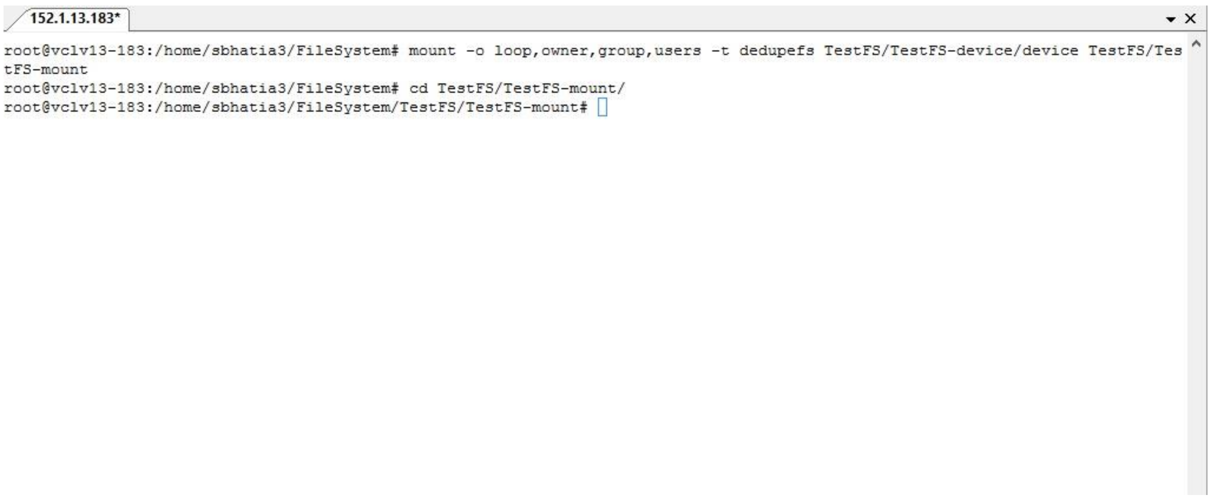
Verification and Validation

The filesystem is developed on Ubuntu 14.04, 64-bit base image, Linux kernel version 3.13, on a server reservation of NC State University's VCL. The development efforts were verified by performing basic file operations. Manual testing was done to validate all claims made in the project objectives. Following test cases were executed successfully after mounting the file system

- Creation of directories within the root directory
- Creation of sub-directories within the newly created directory
- Creating new files using cat, touch and echo commands
- Appending contents to an existing file
- Deleting files and directories
- Reading contents from file
- Creating replicas of file and verifying their size
- Creating replicas of file and verifying the disk utilization using df –h Linux command
- Un-mount, remove, re-insert and re-mount the kernel module and to verify presence of files and directories created in previous step to ratify persistence

Results

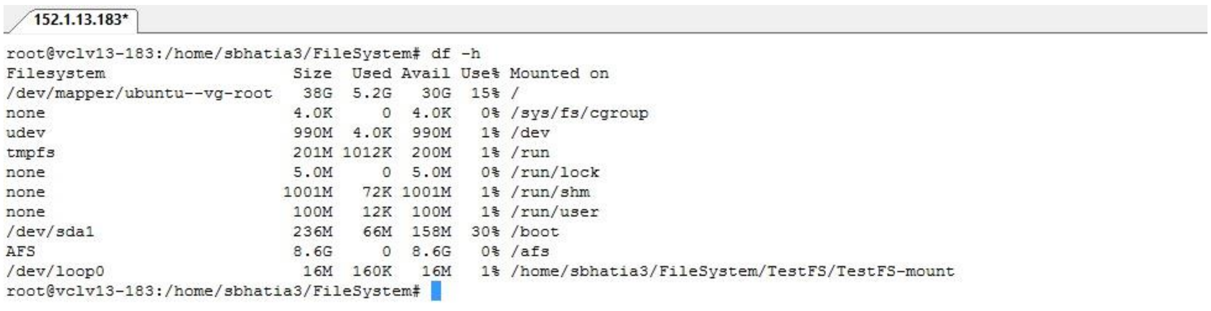
The outputs of the test cases described in the previous section are captured in the following screenshots. Figure 6 indicates the mount point of the filesystem. The mount point is attached to a loop device.



```
root@vclv13-183:/home/sbhatia3/FileSystem# mount -o loop,owner,group,users -t dedupefs TestFS/TestFS-device/device TestFS/Tes
tFS-mount
root@vclv13-183:/home/sbhatia3/FileSystem# cd TestFS/TestFS-mount/
root@vclv13-183:/home/sbhatia3/FileSystem/TestFS/TestFS-mount#
```

Figure 6: Mount point of the filesystem

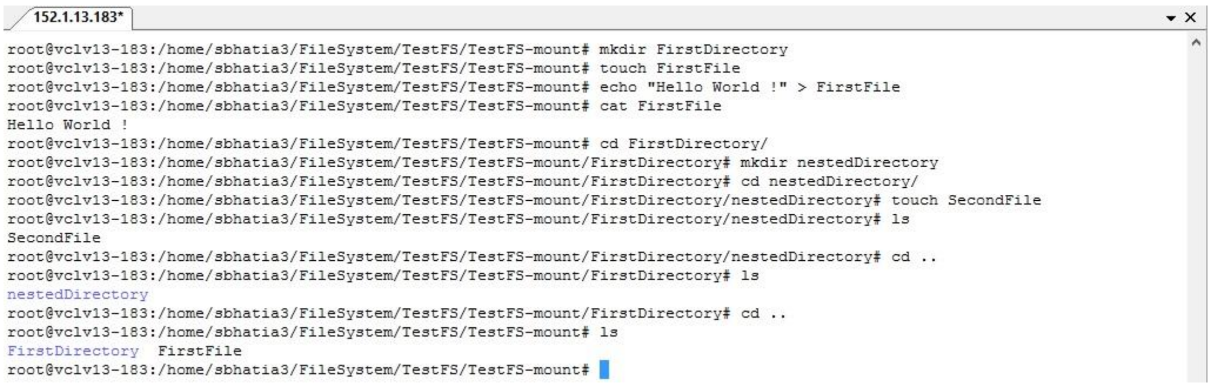
Figure 7 illustrates the disk utilization when the file system is mounted. One percent of the device space is used to store the filesystem metadata.



```
root@vclv13-183:/home/sbhatia3/FileSystem# df -h
Filesystem                    Size  Used Avail Use% Mounted on
/dev/mapper/ubuntu--vg-root    38G  5.2G   30G  15% /
none                          4.0K     0  4.0K   0% /sys/fs/cgroup
udev                          990M  4.0K  990M   1% /dev
tmpfs                         201M 1012K  200M   1% /run
none                          5.0M     0  5.0M   0% /run/lock
none                         1001M   72K 1001M   1% /run/shm
none                          100M   12K  100M   1% /run/user
/dev/sda1                     236M   66M  158M  30% /boot
AFS                           8.6G     0  8.6G   0% /afs
/dev/loop0                     16M  160K   16M   1% /home/sbhatia3/FileSystem/TestFS/TestFS-mount
root@vclv13-183:/home/sbhatia3/FileSystem#
```

Figure 7: Metadata - Disk Utilization

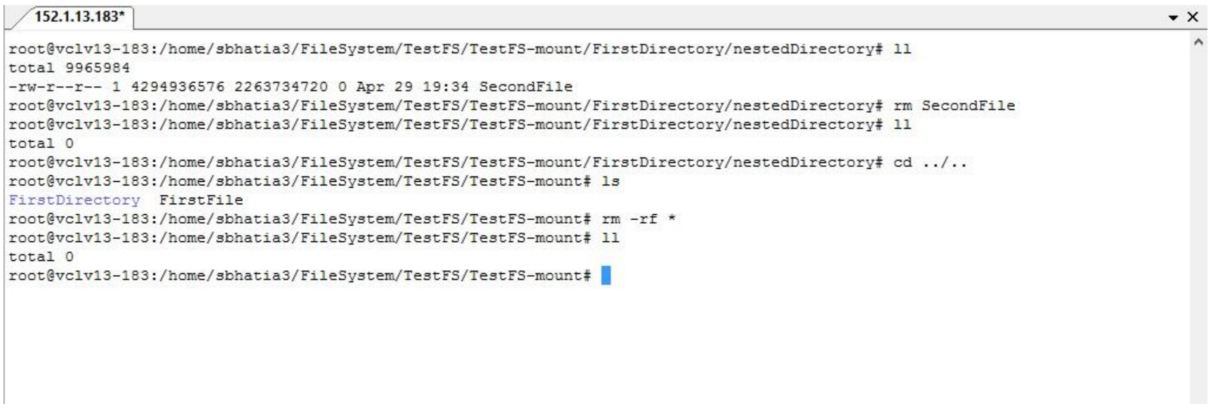
Figure 8 shows basic file/directory creation operation.



```
root@vclv13-183:/home/sbhatia3/FileSystem/TestFS/TestFS-mount# mkdir FirstDirectory
root@vclv13-183:/home/sbhatia3/FileSystem/TestFS/TestFS-mount# touch FirstFile
root@vclv13-183:/home/sbhatia3/FileSystem/TestFS/TestFS-mount# echo "Hello World !" > FirstFile
root@vclv13-183:/home/sbhatia3/FileSystem/TestFS/TestFS-mount# cat FirstFile
Hello World !
root@vclv13-183:/home/sbhatia3/FileSystem/TestFS/TestFS-mount# cd FirstDirectory/
root@vclv13-183:/home/sbhatia3/FileSystem/TestFS/TestFS-mount/FirstDirectory# mkdir nestedDirectory
root@vclv13-183:/home/sbhatia3/FileSystem/TestFS/TestFS-mount/FirstDirectory# cd nestedDirectory/
root@vclv13-183:/home/sbhatia3/FileSystem/TestFS/TestFS-mount/FirstDirectory/nestedDirectory# touch SecondFile
root@vclv13-183:/home/sbhatia3/FileSystem/TestFS/TestFS-mount/FirstDirectory/nestedDirectory# ls
SecondFile
root@vclv13-183:/home/sbhatia3/FileSystem/TestFS/TestFS-mount/FirstDirectory/nestedDirectory# cd ..
root@vclv13-183:/home/sbhatia3/FileSystem/TestFS/TestFS-mount/FirstDirectory# ls
nestedDirectory
root@vclv13-183:/home/sbhatia3/FileSystem/TestFS/TestFS-mount/FirstDirectory# cd ..
root@vclv13-183:/home/sbhatia3/FileSystem/TestFS/TestFS-mount# ls
FirstDirectory  FirstFile
root@vclv13-183:/home/sbhatia3/FileSystem/TestFS/TestFS-mount#
```

Figure 8: Basic operation: Creation

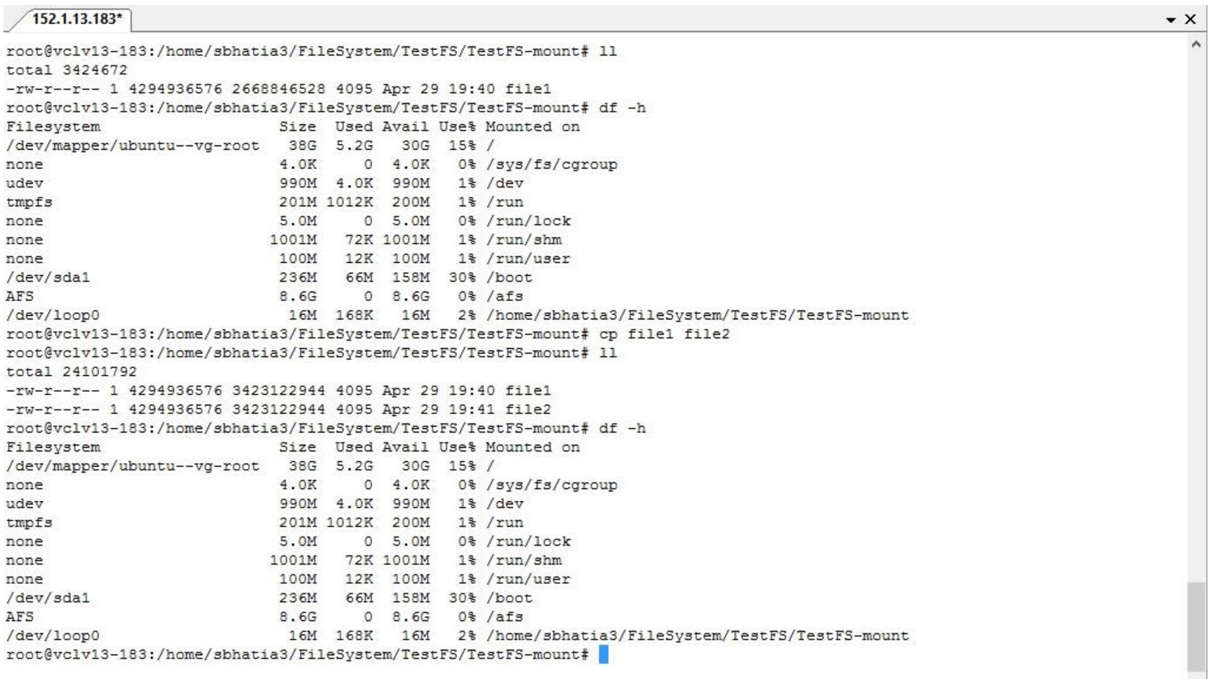
Figure 9 shows basic file/directory deletion operation.



```
root@vclv13-183:/home/sbhatia3/FileSystem/TestFS/TestFS-mount/FirstDirectory/nestedDirectory# ll
total 9965984
-rw-r--r-- 1 4294936576 2263734720 0 Apr 29 19:34 SecondFile
root@vclv13-183:/home/sbhatia3/FileSystem/TestFS/TestFS-mount/FirstDirectory/nestedDirectory# rm SecondFile
root@vclv13-183:/home/sbhatia3/FileSystem/TestFS/TestFS-mount/FirstDirectory/nestedDirectory# ll
total 0
root@vclv13-183:/home/sbhatia3/FileSystem/TestFS/TestFS-mount/FirstDirectory/nestedDirectory# cd ../..
root@vclv13-183:/home/sbhatia3/FileSystem/TestFS/TestFS-mount# ls
FirstDirectory  FirstFile
root@vclv13-183:/home/sbhatia3/FileSystem/TestFS/TestFS-mount# rm -rf *
root@vclv13-183:/home/sbhatia3/FileSystem/TestFS/TestFS-mount# ll
total 0
root@vclv13-183:/home/sbhatia3/FileSystem/TestFS/TestFS-mount#
```

Figure 9: Basic operation – Deletion

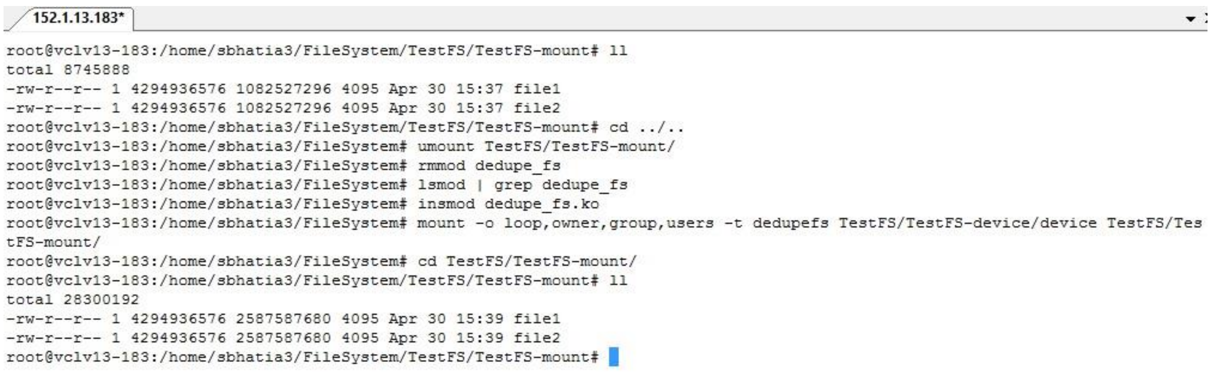
Figure 10 illustrates deduplication at work. Two files with same content are created and it is observed that the disk space utilization equals the size of one such file.



```
root@vclv13-183:/home/sbhatia3/FileSystem/TestFS/TestFS-mount# ll
total 3424672
-rw-r--r-- 1 4294936576 2668846528 4095 Apr 29 19:40 file1
root@vclv13-183:/home/sbhatia3/FileSystem/TestFS/TestFS-mount# df -h
Filesystem                   Size  Used Avail Use% Mounted on
/dev/mapper/ubuntu--vg-root   38G  5.2G   30G  15% /
none                         4.0K     0  4.0K   0% /sys/fs/cgroup
udev                         990M  4.0K  990M   1% /dev
tmpfs                        201M 1012K  200M   1% /run
none                         5.0M     0  5.0M   0% /run/lock
none                        1001M   72K 1001M   1% /run/shm
none                         100M   12K  100M   1% /run/user
/dev/sda1                    236M   66M  158M  30% /boot
AFS                          8.6G     0  8.6G   0% /afs
/dev/loop0                    16M  168K   16M   2% /home/sbhatia3/FileSystem/TestFS/TestFS-mount
root@vclv13-183:/home/sbhatia3/FileSystem/TestFS/TestFS-mount# cp file1 file2
root@vclv13-183:/home/sbhatia3/FileSystem/TestFS/TestFS-mount# ll
total 24101792
-rw-r--r-- 1 4294936576 3423122944 4095 Apr 29 19:40 file1
-rw-r--r-- 1 4294936576 3423122944 4095 Apr 29 19:41 file2
root@vclv13-183:/home/sbhatia3/FileSystem/TestFS/TestFS-mount# df -h
Filesystem                   Size  Used Avail Use% Mounted on
/dev/mapper/ubuntu--vg-root   38G  5.2G   30G  15% /
none                         4.0K     0  4.0K   0% /sys/fs/cgroup
udev                         990M  4.0K  990M   1% /dev
tmpfs                        201M 1012K  200M   1% /run
none                         5.0M     0  5.0M   0% /run/lock
none                        1001M   72K 1001M   1% /run/shm
none                         100M   12K  100M   1% /run/user
/dev/sda1                    236M   66M  158M  30% /boot
AFS                          8.6G     0  8.6G   0% /afs
/dev/loop0                    16M  168K   16M   2% /home/sbhatia3/FileSystem/TestFS/TestFS-mount
root@vclv13-183:/home/sbhatia3/FileSystem/TestFS/TestFS-mount#
```

Figure 10: Deduplication illustration

Figure 11 illustrates persistence at work. The kernel module is un-mounted, removed, re-inserted and re-mounted. After re-mounting the files created initially are again available.



```
root@vclv13-183:/home/sbhatia3/FileSystem/TestFS/TestFS-mount# ll
total 8745888
-rw-r--r-- 1 4294936576 1082527296 4095 Apr 30 15:37 file1
-rw-r--r-- 1 4294936576 1082527296 4095 Apr 30 15:37 file2
root@vclv13-183:/home/sbhatia3/FileSystem/TestFS/TestFS-mount# cd ../..
root@vclv13-183:/home/sbhatia3/FileSystem# umount TestFS/TestFS-mount/
root@vclv13-183:/home/sbhatia3/FileSystem# rmmod dedupe_fs
root@vclv13-183:/home/sbhatia3/FileSystem# lsmod | grep dedupe_fs
root@vclv13-183:/home/sbhatia3/FileSystem# insmod dedupe_fs.ko
root@vclv13-183:/home/sbhatia3/FileSystem# mount -o loop,owner,group,users -t dedupefs TestFS/TestFS-device/device TestFS/Tes
tFS-mount/
root@vclv13-183:/home/sbhatia3/FileSystem# cd TestFS/TestFS-mount/
root@vclv13-183:/home/sbhatia3/FileSystem/TestFS/TestFS-mount# ll
total 28300192
-rw-r--r-- 1 4294936576 2587587680 4095 Apr 30 15:39 file1
-rw-r--r-- 1 4294936576 2587587680 4095 Apr 30 15:39 file2
root@vclv13-183:/home/sbhatia3/FileSystem/TestFS/TestFS-mount#
```

Figure 11: Persistence Illustration

Related Work

The Log structured file system addresses the disk bottleneck issues generally observed in traditional Unix File systems. It buffers all data and metadata writes in a memory cache before eventually saving it to disk as a single log entry. This results in small number of large writes and thus reduces the seek time. Also, in a log structured file system all these writes are asynchronous and thereby it amortizes the seek cost over a large amount of buffered data [8]. Write Anywhere File Layout (WAFL) is a file system designed by NetApp Inc. for use in its storage appliances. Similar to Unix's Berkely Fast File System, WAFL is a block based file system, using 4 KB blocks with no fragments. It uses files to store meta-data. The two most important meta-data files are inode files and free block bitmap file. Keeping meta-data in files allows meta-data blocks to be written anywhere on the disk. WAFL treats data and metadata blocks equally and writes go to nearest free block thus reducing the seek time [9].

Extreme Binning [10] is a deduplication technique that exploits the file similarity for deduplication to apply to non-traditional backup workloads with low-locality. It stores a representative index of each new file in RAM and groups many similar files into bins that are stored on the disks. A study of all these approaches helped in deciphering all the low level intricacies of file system design and deduplication features. It paved the way for designing the data structures for this project.

Concluding Remarks

This project focused on implementing an enterprise level file system with support for basic file system operations and provides deduplication feature to economize storage cost. The main focus of this project was to engineer a working file system and it intentionally compromises on certain minor lookup efficiencies in the lieu of the simple implementation. The deduplication feature ratified the understanding that an intelligent compression technique helps enterprises save a large amount of disk space in an age where there is an exponential data growth. The work accomplished in this project can be extended by introducing features like support for hard and soft links, support for indirect blocks to allow creation of files with arbitrary large sizes, support for arbitrary updates to a file, implementation of permissions for regular files and directories. Deduplication using variable sized blocks at the kernel level can be explored to further economize storage space.

Acknowledgments

We would like to thank Dr. Vincent W. Freeh for his support and encouragement that allowed us to do this project and gain invaluable learning about the structure and organization of filesystems. We also extend our appreciation to Tyler Bletsch for his insights in certain aspects like WAFL filesystem and the concepts of choosing fixed size chunks for deduplicating data. Dr. Erez Zadok's web homepage and manuals on his File System Lab's page provided a good understanding of various kernel functions and their intricacies which helped debug certain critical issues in the development process. Finally, the article by Sankar P., proved invaluable in understanding how to create a filesystem as a pluggable Linux kernel module.

References

[1] R. Love. Linux Kernel Development, Third Edition, Addision-Wesley Professional, June 22, 2010

[2] http://www.emc.com/corporate/glossary/data-deduplication.htm

[3]http://www.storage-switzerland.com/Articles/Entries/2013/1/2_Source_vs _Target_ Based_ Data _Deduplication.html

[4] http://www.emc.com/corporate/glossary/fixed-length-deduplication.htm

[5] https://tools.ietf.org/html/rfc3174

[6] http://www.openauthentication.org/files/download/oathPdf/TechnicalWhitePaper.pdf

[7] T. Xie, F. Liu, and D. Feng, "Fast collision attack on MD5", IACR Cryptology ePrint Archive, Report 2013/170, 2013. [Online]. Available from http://eprint.iacr.org/2013/170 2014.11.30 [6] http://www.emc.com/corporate/glossary/fixed-length-deduplication.htm

[8] M. Rosenblum and J. Ousterhout, The design and implementation of a Log-Structured File System, ACM Transactions on Computer Systems, 10(1),26-52,1992

[9] http://community.netapp.com/fukiw75442/attachments/fukiw75442/data-ontap-discussions/2334/1/WAFL.pdf

[10] D. Bhagwat, K. Eshghi, D. Long, and M. Lillibridge. Extreme Binning: Scalable, Parallel Deduplication for Chunk-based File Backup. In MASCOTS, 2009.