Monitoring Framework for B-Tree File System

A Project Report

Submitted by

Gautam Akiwate	**110808006**
Shravan Aras	**110803073**
Sanjeev M.K.	**110808019**

in partial fulfilment for the award of the degree

of

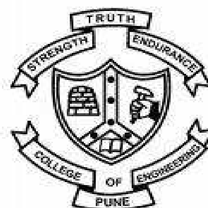
B.Tech Computer Engineering/

Information Technology

Under the guidance of

Prof. Abhijit A.M.

College of Engineering, Pune



DEPARTMENT OF COMPUTER ENGINEERING AND

INFORMATION TECHNOLOGY,

COLLEGE OF ENGINEERING, PUNE-5

May, 2011

DEPARTMENT OF COMPUTER ENGINEERING AND INFORMATION TECHNOLOGY,

COLLEGE OF ENGINEERING, PUNE

CERTIFICATE

Certified that this project, titled ”Monitoring Framework for B-Tree File System” has been successfully completed by

Gautam Akiwate	**110808006**
Shravan Aras	**110803073**
Sanjeev M.K.	**110808019**

and is approved for the partial fulfilment of the requirements for the degree of “B.Tech. Computer Engineering/Information Technology”.

SIGNATURE

Prof. Abhijit A.M.

Project Guide

Department of Computer Engineering

and Information Technology,

College of Engineering Pune,

Shivajinagar, Pune - 5.

SIGNATURE

Dr. Jibi Abraham

Head

Department of Computer Engineering

and Information Technology,

College of Engineering Pune,

Shivajinagar, Pune - 5.

Abstract

The Operating System Kernel is one of the most fundamental piece of software that is instrumental in keeping a system running. Besides providing for tasks like scheduling, interrupt handling and resource management, the kernel also handles storage of data on physical devices. The File System is responsible for these activties in the kernel. The File System maintains various data structures which keep track of of data on the disk. Traditional File Systems are supported by Logical Volume Manager (LVM) which manages the disk partitions on which the File Systems are mounted. However, the Next-Generation File Systems amalgamate the LVM with the actual file system along with other features like Copy-on-Write (COW), RAID, Snapshots, Storage Pools, Checksums etc. Considering the importance of the role that the File System plays and the complexity of the task it is essential to have a monitoring system for the File System that monitor the health of the File System and other parameters so as to indicate the well-being of the File System. In case of the Next-Generation File Systems the need for a monitoring system becomes even more paramount. The B-Tree File System also called BTRFS is one such next generation file system and is touted as the file system of choice in the future. This work deals with design and implementation of a Monitoring Framework for BTRFS. This will enable system administrators to analyze various parameters of File System. The key characteristic of the framework developed which makes it highly desirable is that the framework is flexible and easy to extend.

Contents

List of Tables ii

List of Figures iii

List of Symbols iv

1 Introduction 1

1.1 Background . . . 1

1.2 Definition of Problem Statement . . . 2

2 File Systems - Present and Next Generation 3

2.1 The need of a new generation FS . . . 3

2.2 Future Features . . . 4

2.2.1 Snapshots and Copy-On-Write . . . 4

2.2.2 Integrated Volume Manager . . . 5

2.2.3 Multiple RAID levels . . . 5

2.2.4 Online fsck,resizing,device management . . . 6

2.2.5 Checking for data corruption . . . 6

2.3 File system comparisons . . . 6

3 BTRFS 8

3.1 An Overview of BTRFS . . . 8

3.2 BTRFS Design . . . 8

3.3 Representation of BTRFS meta data . . . 10

3.3.1 Overview . . . 10

3.3.2 BTRFS Key . . . 13

3.4 Case study - BTRFS *debug-tree* output . . . 14

3.5 Important In Memory Structures . . . 17

3.5.1 Super Block . . . 17

3.5.2 Devices . . . 17

3.5.3 Relationship between various In Memory data structures . . . 18

4 Generic Framework **19**

4.1 Overview . . . 19

4.2 Design . . . 20

4.2.1 Design Requirements . . . 21

4.3 Implementation . . . 22

4.3.1 Implementation for BTRFS . . . 23

5 BTRFS Monitoring Framework **24**

5.1 Design Framework . . . 24

5.1.1 Static and Dynamic Phases . . . 24

5.2 The BTRFS sysfs Interface . . . 26

6 Device Statistics **28**

6.1 Keeping count of the disk errors . . . 28

6.1.1 IOCTL vs Sysfs . . . 29

6.1.2 Patch Details . . . 29

7 Device Statistics Patch - Testing 31

7.1 Approach and Design - Use of Layered OS structure 31

7.1.1 Ramdisk 32

7.1.2 Final Testing 33

List of Tables

2.1 Feature Comparison of File Systems 7

4.1 Need for a New Framework 19

4.2 Requirements for Generic Framework 21

4.3 OS Compatability 22

List of Figures

3.1 BTRFS Tree 9

3.2 Disk representation of a generic block in BTRFS 11

3.3 Disk representation of BTRFS Tree Node and Leaf 11

3.4 Leaf Item and the BTRFS key/index 12

3.5 BTRFS Super Block 17

3.6 Relationship between various BTRFS data structures 18

4.1 Monitoring Framework Design 20

5.1 Connection between core BTRFS components and the Sysfs interface . . 26

5.2 sysfs Structure 27

6.1 Device Stats 29

7.1 Layered OS Structure 31

List of Symbols

Chapter 1

Introduction

1.1 Background

File Systems play an essential role in the working of the kernel. Advancements in the area of file storage leading to cheaper storage along with the explosion of data generated have lead to serious rethinking on the design of File Systems. This has lead to the development of the Next-Generation File Systems like ZFS[5] and BTRFS[2].

The new File System are designed primarily with requirements for:

- High Capacity
- Data Integrity
- High Performance
- Simplified Administration
- Reduced Costs
- Compatibility
- General Purspose File System

These File Systems have implemented many novel ideas and features. However chief among them is the concept of storage pools. The storage pool is essentially a collection of all the block devices. This storage pool allows hot swapping of devices. More importantly it does away with the need of a Logical Volume Manager thereby greatly simplifying the administration. This switch to the new architecture optimizes and simplifies code paths from the application to the hardware, producing sustained throughput at exceptional speeds. Furthermore, usage of the Copy on Write Model that writes the data immediately on request by allocatiing a new block allows it to easily take snapshots and restore files as the old data is available too. Not only this but these File Systems also supports variable block sizes which essentially allow to improve performance and storage and some of them also allow for different endianess.[2][5]

1.2 Definition of Problem Statement

Considering the importance of the role that the File System plays and the complexity of the task it is essential to have a monitoring system for the File System that monitor the health of the File System and other parameters so as to indicate the well-being of the File System. However, the new architecture of the Next-Generation File Systems leads to most of the monitoring framework designs for Traditional File Systems to be defunct. Hence, a need is felt to design a generic framework for Monitoring Systems in these new File Systems.

The problem statement comprises of two aspects. First is designing a generic framework for Monitoring Systems in the new File Systems. The second is to implement this design and develop a Monitoring Framework for one of the Next-Generation File Systems. BTRFS was chosen as it was still under active development and the fact that its source is released under GPL License and is hence open to contribution from anyone.[2]

Chapter 2

File Systems - Present and Next Generation

2.1 The need of a new generation FS

Kernel 2.6 has been host to a number of file systems with various underlying concepts. Each FS has enjoyed varying degrees of popularity and user base. While the log ased Extended family of file sytems - ext 2/3/4 - has met almost every need of the day-to-day user and the system administrator, the need for a new breed of file systems that overcome the shortcomings of the ext group, has ever been felt. No file system fulfils every need though. The system admin often has to trade-off on various points when selecting the best file system that suits their purpose. While XFS is designed for multithreading and heavy workloads, the EXT group is designed to occupy minimum disk space and the newer breed like ZFS, BTRFS are known for providing recoverability and newer RAID levels. Which FS to choose depends entirely on the task at hand.

2.2 Future Features

What users expect from file systems: 1. Snapshots - The ability to revert the entire file system back to a previous state at any time. 2. Integrated Volume Manager - The ability to create and manipulate volumes on the fly. 3. Mulitple RAID level support - The ability to support a pool of physical devices with maximum flexibility and control. 4. Online FS resize - Changing the size of a mounted FS. 5. Dynamically adding and removing devices from a common pool without any additional device management. 6. Online fsck - Repairing and modifying a mounted file system. 7. Online defragmentation - Defragmenting the data on a mounted file system 8. Corruption checking - Having the FS associate data with checksums.

We might as well call these the most wanted features in the newer breed of file systems. The new file sytems like ZFS,BTRFS,NiLFS,EXOFS,ReFS do provide some or all of the above five features.

2.2.1 Snapshots and Copy-On-Write

Current file systems guarantee the protection of only the associated metadata in cases of crashes and power outage. BTRFS, ZFS, NiLFS have made it possible to recover entire chunks of data with just a command or two. This is owed to snapshots - copies of previous states of the file system. Snapshots can be mounted like normal partitions and are created when certain checkpoints are reached. One can also manually create these snapshots. Recovering lost data, deleted files, correcting the file system has become as simple as mounting a previous working version of the file system. There are a number of ways of creating snapshots, one of them being Copy-on-Write (COW). The snapshot keeps track of the changing blocks in the original data. Whenever a write operation occurs on the original data, before the write is actually done,the previous copy is written to the

snapshot, and then the new data is written.Thus only changing blocks are copied to the snapshot. Hence the name Copy-on-Write. Read requests to unchanged data blocks read data from the original block, while requests to the changed blocks are fulfilled from the snapshot version.

The impact on the performance is there, as all write operations have to wait for old data to be copied, but read operations are more or less unaffected and there are always more reads than writes. Also the extra space requirement is minimum, as only the changed blocks are copied.

2.2.2 Integrated Volume Manager

This feature makes LVM tools obsolete. Volume management is now desired within the file system itself. With this, one can create and delete volumes within a device on the fly with normal FS tools, without the need of additional tools and management. The FS itself takes care of managing all the volumes. The user simply has to use the FS tools to create the volumes.

2.2.3 Multiple RAID levels

For servers which typically host a pool of devices, it has always been desirable to have a file system that manages data across the multiple devices flexibly. This data managment pertaining to multiple devices is provided by having the file system support different RAID levels. Each RAID level provides different features regarding management of data across a pool of devices. The ability to configure the RAID level during FS creation, enables the system administrator to tell the FS how he wishes the FS to manage data - Whether he wants his data mirrored on all devices, or he wants data to have parity bits associated etc.

2.2.4 Online fsck,resizing,device management

There has been and increasing demand among server administrators for online tools - tools that can operate on a file system while it is mounted. Previously, if one wanted to run `fsck` on a server disk , it had to be unmounted before running the check. This meant unavailability of some server resource for a period of time. This has been done away with, thanks to the invasion of online tools - online FS checking, resizing mounted devices, adding devices to existing pool, removing devices from them and other complex pool management services while the pool is active.

2.2.5 Checking for data corruption

New file sytems now associate blocks of data with their corresponding checksums. This has enabled detection of data corruption soon after it's occurance. The device statistics patch of BTRFS, in fact, uses the checksum data to detect corruption and update the corresponding variable that is exported to userspace via sysfs.

2.3 File system comparisons

While ext4 is still the default file system in majority of the linux distributions, the younger file systems like ZFS and BTRFS may soon take the default spot, given their plethora of features. ZFS and BTRFS have a similar feature list with: snapshots,online defrag,resize,fsck,flexible RAID levels etc. There are very subtle differences however, for example, in ZFS, the snapshots are read-only but for BTRFS, the snapshots can also be written to.

Most newer file systems including ZFS,BTRFS, Windows ReFS do support data integrity checks with checksum and parity bits. XFS has the added feature of providing high performance on parallel systems. While ZFS, BTRFS provide data deduplication

File System	Snapshot	Encryption	COW	Integrated LVM	Data Deduplication
BTRFS	Yes	Yes	Yes	Yes	Yes
ZFS	Yes	Yes	Yes	Yes	Yes
XFS	No	No	No	No	No
NiLFS	Yes	No	Yes	Yes	Yes
ext4	No	No	No	No	No
ReFS	Yes	No	Yes	No	No

Table 2.1: Feature Comparison of File Systems

- reducing redundant data - ReFS has no such feature planned. ZFS and BTRFS also provide data encryption.

It is clear that ZFS and BTRFS are the future with a kingly set of features. Windows systems look forward to ReFS which has a feature set similar to BTRFS but it does not have data deduplication. ReFS however, does have writable snapshots like BTRFS and unlike ZFS. ZFS on the other hand can also work over networks , much like NFS. BTRFS has planned something like that for the future.

However, there is a set of common features binding the new file systems: 1. Snapshots 2. Data integrity and security 3. Mulitple RAID Level support 4. Online FS tools - Defragmentation,resizing,fsck 5. Dynamic volume management

Chapter 3

BTRFS

3.1 An Overview of BTRFS

Btrfs, also called the B-tree File System is a next generation file systems that amalgamates the role of the Logical Volume Manager (LVM) with the actual file system. The Btrfs implements this idea in combination with the concept of storage pools and copy on write system. Btrfs is intended to address the lack of pooling, snapshots, checksums and integral multi-device spanning in Linux file systems.

Btrfs uses on the fly compression for efficient utilization of space with tight packing of small files. This combined with the hash tables allows for a very fast search capability. For improved integrity, Btrfs has checksums for writes and even supports snapshots. Efficient incremental backups, live defragmentation, dynamic expansion of the file system to incorporate new devices, and support of massive amounts of data.

3.2 BTRFS Design

The key feature of Btrfs is that it is implemented with simple and well known constructs. It focuses on maintaining performance of the filesystem. Btrfs essentially is structured as several layers of trees, all of which use the same B-tree implementation to store data

sorted on a 136-bit key. The B-trees are composed of three data types: keys, items and block headers. The data type of these trees is referred to as items and all are sorted on a 136-bit key. The key is divided into three chunks. The first is a 64-bit object id, followed by 8-bits denoting the item's type, and finally the last 64 bits have distinct uses depending on the type. The unique key is to help with quick searches using hash tables. It is also necessary for the sorting algorithms that are designed to keep the tree balanced. The first 64 bits of the key are a unique object id. The middle 8 bits is an item type field. The remaining 64 bits are used in type-specific ways.

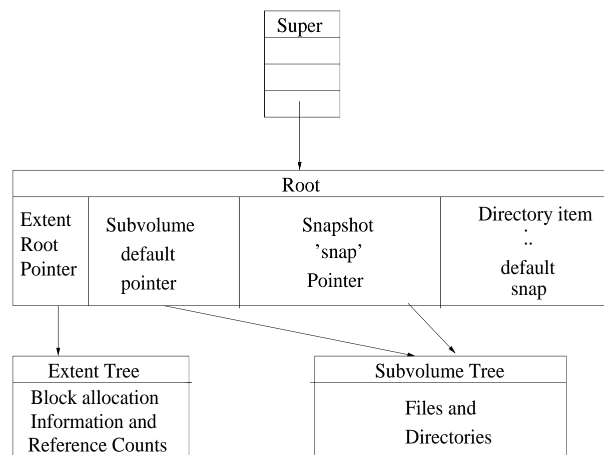


Figure 3.1: BTRFS Tree

The items contain a key and information on the size and location of the items data. Block headers contain various information about blocks.The trees are constructed of these primitive items with interior nodes containing keys to identify the node and block pointers that point to the child of the node. Leaves contain multiple items and their data. At a minimum the Btrfs contains three trees. The first tree contains other tree roots. Second tree contains a subvolumes tree which holds files and directories and third contains an extents volume tree that contains information about all the allocated extents files. Btrfs also has a data structure called as superblock that points to the root of roots. Btrfs can

have additional trees needed to support other features.

The copy on write method of the system is a pivotal aspect of Btrfs, once which affects a number of its features. Writes never occur on the same blocks. A transaction log is created and writes are cached. The file system then allocates sufficient blocks for the new data and the new data is written there. All subvolumes are updated to the new blocks in case of replication. The old blocks are then removed and freed at the discretion of the file system. This copy on write combined with the internal generation number allows the system to create snapshots of the data to be made. This is so because the old data is still available after the write as the old data is not overwritten. After each copy the checksum is also recalculated on a per block basis and a duplicate is made to another chunk. These actions combine to provide exceptional data integrity.
Thus, Btrfs with a very simple underlying implementation of B-trees provides a number of features which make it a robust and easy to use file system along which deems it to be file system of choice in the foreseeable future.

3.3 Representation of BTRFS meta data

3.3.1 Overview

All the meta data pertaining to the filesystem are stored in the B-Tree with the intermediate nodes containing keys and pointers to other block, while the actual data is stored only in the tree leaves. The nodes and the leafs when stored onto the disk are packed into block, each of 4KB, out of which 101 Bytes are reserved for the header and the remaining 3995 Bytes are used to store data. A diagram representing a on disk block is show in figure 3.3.1

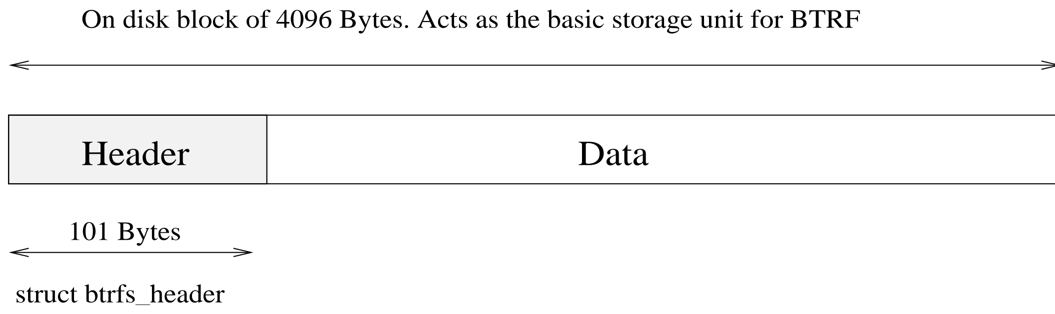


Figure 3.2: Disk representation of a generic block in BTRFS

Figure 3.3.1 represents a generic on disk block while figure 3.3.1 depicts specific examples when the data being represented is a tree node and tree leaf respectively.

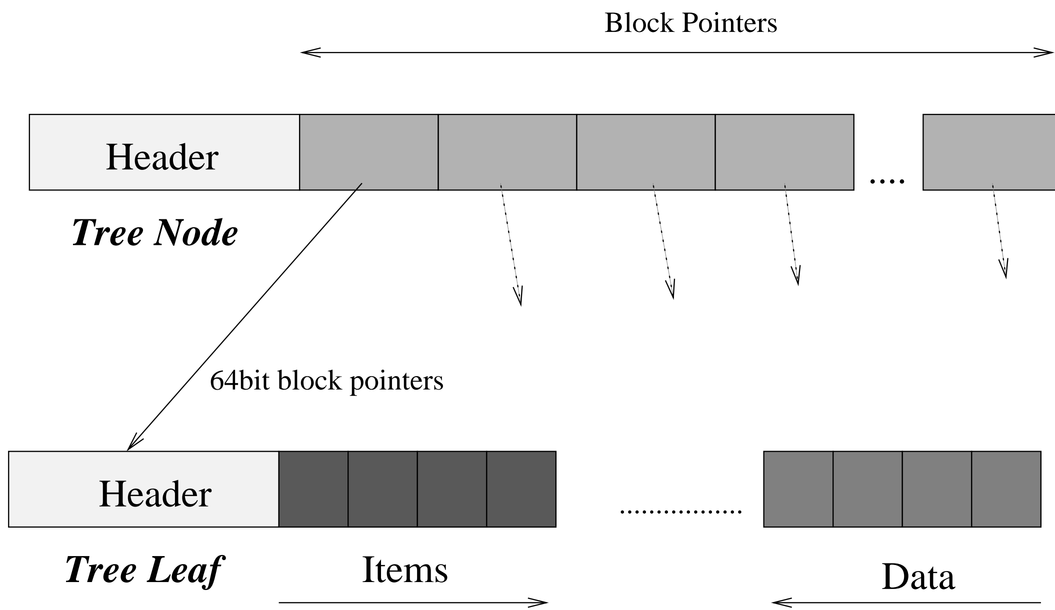


Figure 3.3: Disk representation of BTRFS Tree Node and Leaf

As shown in figure 3.3.1 every element is keyed, like in a regular B-tree. In the case of BTRFS trees the items and the data corresponding to them are both stored in the same block. The items containing the keys (index in B-Tree) are packed together making

it faster to search amongst them. The items start right after the *header* and expand towards the data which starts from the end of the block and expands towards the center. A stack and heap analogy can be used to understand this concept better.

A BTRFS item can be further divided into 3 sections -

- BTRFS key - A 136 Bits tuple composed of a 64 Bit objectid, 8 Bits of type information and 64 Bits of offset. The tuple is always stored in CPU native order and are used as indexes in navigating and forming the B-tree.
- offset
- size - The offset and size together determine the position of the item data relative to the start of the data section (right end of the block).

Figure ?? gives a diagramatic representation of the item and key structure.

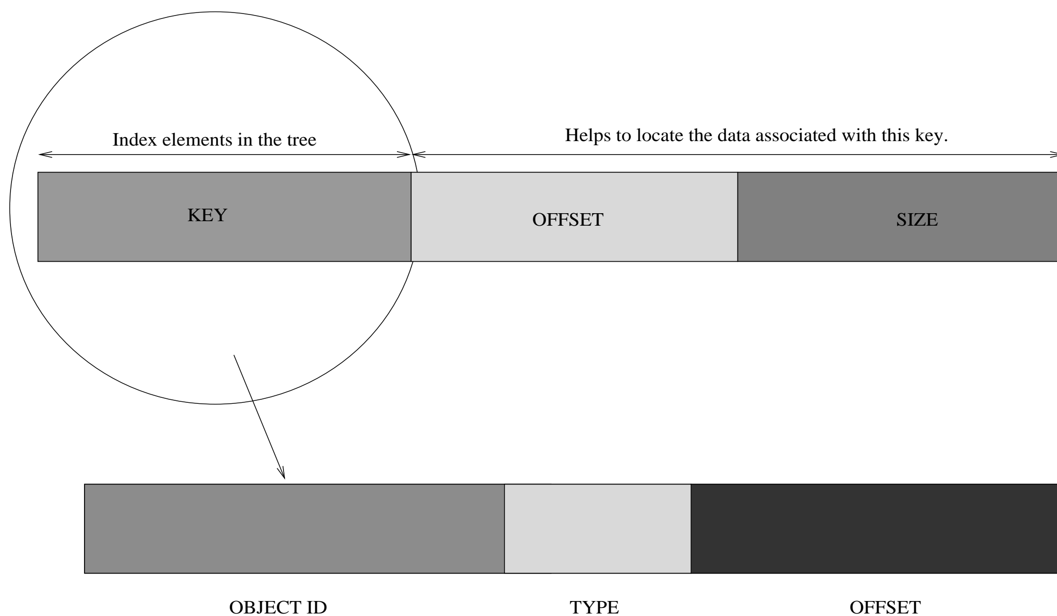


Figure 3.4: Leaf Item and the BTRFS key/index

3.3.2 BTRFS Key

The BTRFS key is used to index the elements in the BTRFS tree, with the *object id* and *type* together used to determine the nature of data represented by this key. As a reminder, keys are only present in the leafs of the BTRFS tree, hence the intermediate nodes contain no such structures.

The concept of *keys* can be better understood with the help of the following examples. The notations used in the below examples are - *[objectid, type, offset]*

1. *[parent_inode, DIR_ITEM, crc32 hash of filename]* - Represents a key which can be used to point to a directory entery (data represented by this key), which in majority of cases is a *INODE_ITEM*. If the directory item is present in the root directory then it's parent inode number is 256. Object id's below 256 are reserved for special keys while those above 256 can be used by inodes.

2. *[inode_number, INODE_REF, parent_inode]* - This key points to a *INODE_REF* which is used to store the filename (in UNIX like systems even directories are files) and name length, amongst other parameters. The inode number can be used to link this entry with its respective *INODE_ITEM* entry. The prime importance of this key is to link a inode (can be a dictory of a file) with its parent.

3. *[inode_number, INODE_ITEM, 0]* - The data represented by this key points to the actual chunk of data in the chunk tree. The offset field for this key is not implemented yet and contains the value 0.

Important - In BTRFS, the data (contents of the file) is stored inside the meta data objects itself if the data is small enough. Hence for small files the *INODE_ITEM* does not point to a valid entry inside the chunk tree as the data is stored inside the meta data block itself. Such entries are denoted by the type *BTRFS_FILE_EXTENT_INLINE*

Having looked at the basic elements which constitute BTRFS tree, in the next section we take up a case study were we look at how the meta data changes as new files and directories are added and deleted from the file system.

3.4 Case study - BTRFS *debug-tree* output

The offset as explained in **??** is The *btrfs-debug-tree* tool can be used print the filesystem meta data. The output is largely divided into 6 sections -

1. *root tree* - This section correspond to the BTRFS *root tree* and contains no intermediate nodes but only leafs. The leaves in turn point to various other trees of the BTRFS filesystem including the *root tree*.

2. *chunk tree* - This section corresponds to the BTRFS *chunk tree*. User data is stored in chunks and the mapping of various chunks to their respective entries in the device trees is taken care by the *chunk tree*.

3. *extent tree* - Corresponding to the *extent tree*.

4. *device tree* - Used to keep track of devices and corresponds to BTRFS *device tree*.

5. *fs tree* - This section shows us the files in the filesystem and directly relates to the BTRFS *fs tree*.

6. *checksum tree* - Corresponds to the *checksum tree*.

We begin our study with *fs tree* section and slowly progress towards the device tree.

```
fs tree key (FS_TREE ROOT_ITEM 0)

leaf 29528064 items 14 free space 2700 generation 134 owner 5

fs uuid 4b735dc7-e855-48cd-9855-31ade9a46031

chunk uuid 30525af3-ee43-4b8e-af9c-411fa97024c6
```

```
item 0 key (256 INODE_ITEM 0) itemoff 3835 itemsize 160

inode generation 3 size 30 block group 0 mode 40555 links 1

item 1 key (256 INODE_REF 256) itemoff 3823 itemsize 12

inode ref index 0 namelen 2 name: ..

item 2 key (256 DIR_ITEM 1487203495) itemoff 3786 itemsize 37

location key (287 INODE_ITEM 0) type 1

namelen 7 datalen 0 name: myfile1

item 3 key (256 DIR_ITEM 2801880295) itemoff 3748 itemsize 38

location key (293 INODE_ITEM 0) type 1

namelen 8 datalen 0 name: myfile12

item 4 key (256 DIR_INDEX 38) itemoff 3711 itemsize 37

location key (287 INODE_ITEM 0) type 1

namelen 7 datalen 0 name: myfile1

item 5 key (256 DIR_INDEX 45) itemoff 3673 itemsize 38

location key (293 INODE_ITEM 0) type 1

namelen 8 datalen 0 name: myfile12

item 6 key (287 INODE_ITEM 0) itemoff 3513 itemsize 160

inode generation 133 size 205268 block group 0 mode 100644 links 1

item 7 key (287 INODE_REF 256) itemoff 3496 itemsize 17

inode ref index 38 namelen 7 name: myfile1

item 8 key (287 XATTR_ITEM 2038346239) itemoff 3415 itemsize 81

location key (0 UNKNOWN 0) type 8

namelen 23 datalen 28 name: system.posix_acl_access

item 9 key (287 EXTENT_DATA 0) itemoff 3362 itemsize 53

extent data disk byte 13246464 nr 208896

extent data offset 0 nr 208896 ram 208896

extent compression 0

item 10 key (293 INODE_ITEM 0) itemoff 3202 itemsize 160

inode generation 134 size 205321 block group 0 mode 100644 links 1

item 11 key (293 INODE_REF 256) itemoff 3184 itemsize 18

inode ref index 45 namelen 8 name: myfile12

item 12 key (293 XATTR_ITEM 2038346239) itemoff 3103 itemsize 81
```

```
location key (0 UNKNOWN 0) type 8
namelen 23 datalen 28 name: system.posix_acl_access
item 13 key (293 EXTENT_DATA 0) itemoff 3050 itemsize 53
extent data disk byte 12582912 nr 208896
extent data offset 0 nr 208896 ram 208896
extent compression 0
```

We start by focusing our attention on the line number 3 and 4. These 2 lines contain the FS uuid and the chunk tree uuid. Both these values come after the block header (refer figure **??**), the FS uuid is same as the one present in the super block. We now look at some of the prime items which help us understand how the content is distributed.

1. item0 and item1 - As described in **??**, a combination of *INODE_ITEM* and *INODE_REF* can used to describe a file. The *parent inode* and the *inode* number of this item are the same as the file always points to its parent directory. In this case however we are in the root directory, the parent directory of the root directory is the root directory itself. It can also be seen that the object id here is 256. This is the inode number given to the first file created on disk, with *object id's* below 256 reserved for BTRFS objects.

2. myfile1 - There are a total of 6 items associated with *myfile1*

    - item2 - We start of with the *directory item*, as explained in 3.3.1 the object id in this case points to the inode of the parent. In this example it is 256, the parent of this file is the root directory. The offset in our case is the crc32 hash value of the filename. The next 2 lines below the item2 describe the data pointed to by this key. In this case it points us to *inode item* data type with the inode number 287 and also contains some auxillary information like the filename and its length. We look at the *inode item* next pointed to by this directory entry (item6).

    - item6 - We jump directly to item6. Here it can be observed that object id is the inode number of the file, which is 287. The offset as explained in **??** is 0. As mentioned earlier the data in *inode item* points directly into the *chunk tree* indexed by the key *[287, INODE_ITEM, 0]*

    - item7 - This is a *inode reference* item which links the current inode to its parent. It can

clearly be seen from the tuple that the parent of this file is the root directory with the inode number as 256.

3.5 Important In Memory Structures

This sections tries to explain the various in memory data structures and the way they are linked together in the BTRFS filesystem.

Please note that only those data structures used in the sysfs implementation have been explained here, a detailed explaination of all the data structures is beyond the scope of this document.

3.5.1 Super Block

The super block of any filesystem is the starting point for a host of operations and data structure, hence to maintain redundancy it is mirrored at multiple locations on the disk. BTRFS is no different and the maximum number of super blocks can be 3, as shown in the figure. 3.5.1

Please note that the exact number of super block copies on a device greatly depend on the size of the device. Super block 1 and 2 are commonly found in majority of the devices however super block 3 is found on larger devices.

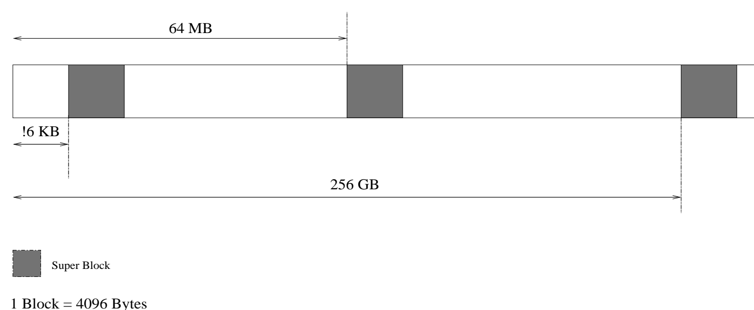


Figure 3.5: BTRFS Super Block

3.5.2 Devices

The BTRFS filesystem is capable of working on a pool of devices. Hence it is necessary for the filesystem to keep track of all these devices, which is does nicely in the form of a double linked list. As our device monitoring parameters are per device, we have placed our device statistics monitoring variables also

inside this in memory representation.

3.5.3 Relationship between various In Memory data structures

A diagramatic representation depicting the relationship between various important in memory data structures is shown in figure 3.5.3

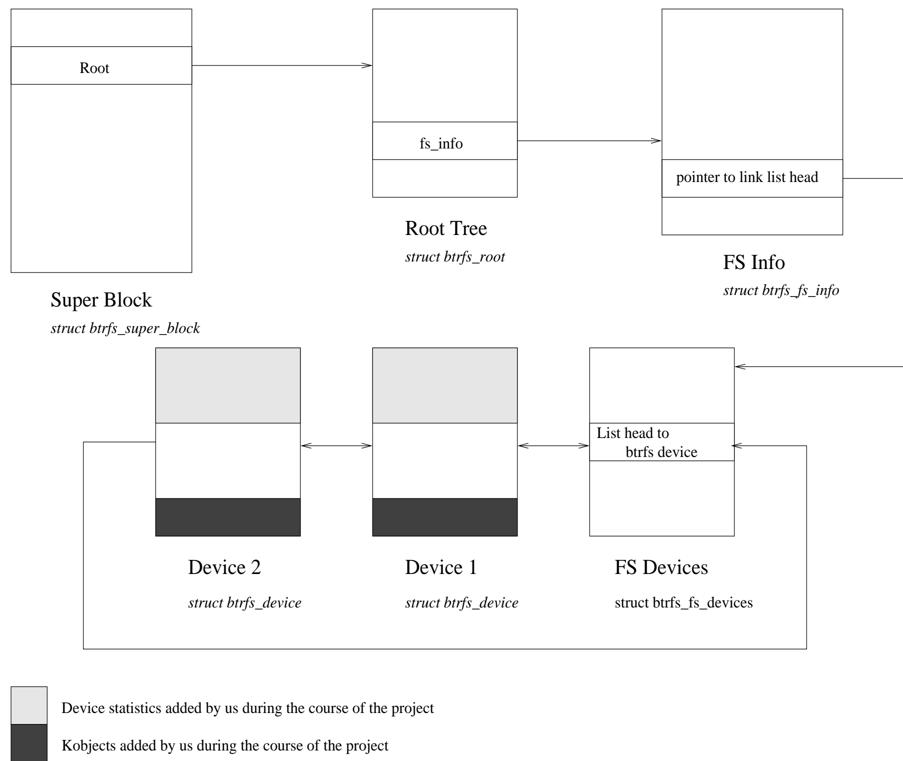


Figure 3.6: Relationship between various BTRFS data structures

Having looked at the basics of BTRFS, we look at how the SysFs interface has been interfaced with BTRFS in the subsequent chapters.

Chapter 4

Generic Framework

4.1 Overview

Issues of Next-Generation File Systems
New Architecture
Added Features
Increased Functionality
Component Interdependency
Data Duplication

Table 4.1: Need for a New Framework

The new architecture of the Next-Generation File System does not fit within the scheme of the framework of the monitoring systems for traditional file systems. This is essentially because the new File Systems have taken on additional responisbility and quite a lot of features too. For example, the new file systems are responsible for the management of devices that come under the file system and also the distribution of data among these devices. Thus the framework also has to account for interdependency between the components and the need for the same data at multiple locations.

4.2 Design

The Design for the Generic Framework basically means the overview of the information that needs to be collected and its categorization. In this aspect we studied two of the best examples of the new genre of File Systems that is ZFS and BTRFS. After a brief study we listed down for major aspects that seemed critical for a Generic Framework. As of now these aspects only cover the critical aspects of these File Systems but nevertheless cannot be deemed complete. But one of the major design consideration is that this framework should be easily extendible and hence adding other aspects to this shouldn't be a major hurdle.

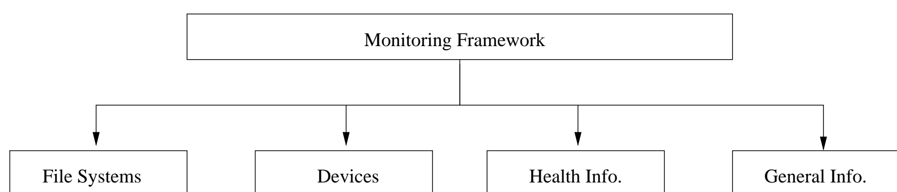


Figure 4.1: Monitoring Framework Design

In Figure 4.1 we have a broadview of the categorization of information that needs to be monitored. The Categorization is done as follows:

1. **General Information**

   This category includes the general information like the number of devices, the number of devices opened and possibly other information about the File System module like the block size, worker threads etc.

2. **Health Information**

   This category includes the health specific information of the file system and devices. It is important to note that information about the devices like its error statistics etc. maybe duplicated from the devices entry which will also be listed in the Devices category.

3. **Devices**

   This category will contain specific information about the devices. This will contain statistics about the device including error statistics etc.

4. **File System**

   This category contains the File System instances presently on the system. This will also keep

information about the aspects of the File System like its label, id if any etc. Again it is important to note that every File System instance will duplicate information about the devices that are under the said file system.

4.2.1 Design Requirements

Now that we have a skeleton in place let us try to understand the design requirements that will be needed to ensure correct implementation of the design. The first and foremost is that of the Userspace-Kernel Interaction. It is understood that the monitoring activity will take place from the Userspace. However, the monitored data will be part of the Kernel. Hence, the design should account for interaction between the Userspace and the Kernel. However, at the sametime care must be taken that this does not lead to security issues.

Design Requirements
Userspace-Kernel Interaction
Hierarchical Structure
Linking of Data
Easy to Use
Extensible
Secure

Table 4.2: Requirements for Generic Framework

Furthermore, the information being monitored should be easily organised into an hierarchical structure preferably one that follows the broad categories defined earlier. Also, as listed earlier in Table 4.1 there is a lot of interdependent components and hence there is a chance that information needs to be duplicated. As duplication of data is not always such a good idea there should be a provision of linking data or other means by which we can avoid duplication. More importantly this should be easy to use and also extensible. As the File System matures there will be a need to extend the framework for other such information and this should be easily possible by designing the framework correctly.

4.3 Implementation

As far as implementation is concerned there are to two options. First, is to implement our own custom interface with which to implement the framework. Alternatively, an existing interface can be used which best suits our requirements. After a preliminary study we shortlisted a few possible interfaces that we could use to design the framework.

Existing Interfaces that can be used are

1. *sysctl* It is an interface for examining and dynamically changing parameters. A system call or system call wrapper is usually provided for use by programs, as well as an administrative program and a configuration file for usage of *sysctl*

2. *sysfs* It is a virtual file system that exports information about devices and drivers from the kernel device model to user space, and is also used for configuration. It is similar to the sysctl mechanism but is implemented as a file system instead of a separate mechanism.

3. *procfs* It is a special filesystem that conveys information about processes and other system information in a hierarchical file-like structure, providing a more convenient and standardized method for dynamically accessing process data held in the kernel than traditional tracing methods or direct access to kernel memory.

4. *ioctl* It is a system call for device-specific input/output operations and other operations which cannot be expressed by regular system calls.

Possiblities to Implement Framework			
Interface	**FreeBSD**	**Linux**	**Windows**
sysctl	Yes	No	No
sysfs	No	Yes	No
procfs	Yes	Yes	No
ioctl	Yes	Yes	Yes

Table 4.3: OS Compatability

4.3.1 Implementation for BTRFS

For purposes of implementation we had chosen BTRFS. As BTRFS works on Linux are main area of focus was interfaces in Linux. After interacting with the BTRFS Development Community the choice was narrowed down to *ioctl* and *sysfs*. After further delibration this was finalized to *sysfs* as *ioctl* was already being heavily used in the BTRFS code and were becoming too bulky. Also *sysfs* satisfied all our design requirements and hence was suitable to develop the framework.[4]

Chapter 5

BTRFS Monitoring Framework

If there is one thing that system administrators regret is the circumstance when they loose all their data. Regular backups and RAID configuration is all nice, however system administrators need a concrete way to monitor the filesystem and for application developers to easily build on existing tools. In this chapter we deal with the various design alternatives we considered and the actual Sysfs implementation for BTRFS.[1][3]

5.1 Design Framework

In this section we look at the various design decisions that went into the design and implementation of the monitoring framework.

5.1.1 Static and Dynamic Phases

We have split the Sysfs implementation into Static and Dynamic parts with the static part being called at module initialization phase while the dynamic part is called every time a device is added/removed to/from the device tree.

Static Phase

The static phase of the Sysfs code is called as soon as the module is initialized and freed when the module is removed from the system. The major objective of the static phase are as follows -

1. Register the *btrfs* kset with top level *fs* kobject

2. Initialize the 3 basic top level directories - devices, health and info.

3. Register these kobjects with the *btrfs* kset.

It must be noted here that the above directory structure can easily be extended or modified as required by specific users of the filesystem.

Dynamic Phase

The dynamic phase is responsible for creating entries inside the *devices* as and when the devices are mounted and cleaning up the entries when the devices are no longer in use. The following three designs were implemented and tested for the dynamic phase -

1. Allocating memory using kzalloc/kmalloc when the device is mounted.

2. Using the kobject from *btrfs_fs_info* data structure.

3. Adding a kobject to *btrfs_device* data structure and using it.

After implementing and testing both the approaches, we came to the conclusion that the third approach was more feasible. The prime reason for this decision was the fact that both *kzalloc* and *kmalloc* allocate memory on behalf of the process calling it. In our situation the process allocating memory was *mount* while the process freeing the memory was *umount*. However the memory requested for freeing could not be loaded after a page fault. Hence the first approach was eliminated.

After working with the second approach for a considerable amount of time, we realized that there was a flaw in our implementation. If we used the *kobject* located inside the *btrfs_fs_info*, it would be common for the whole filesystem. This posed a serious problem as

- BTRFS allows us to create a pool of devices associated with a filesystem. Hence the kobject will be common amongst all the devices.

- Our monitoring framework was required to work on device level. The kobject was of prime important as it was the only way we could decipher on which device the Sysfs *show* and *store* functions were called. The reader must note that in our implementation we have set of generic Sysfs *show* and *store* function for all devices as the number of devices are not know before hand and are created on the fly dynamically.

Due to the above reasons we concluded that it would be best to add a *kobject* inside *btrfs_device*.

Note: The second approach of placing the *kobject* can be used in the case when monitoring on filesystem level is required. So it is proposed that the *kobject* must not be removed.

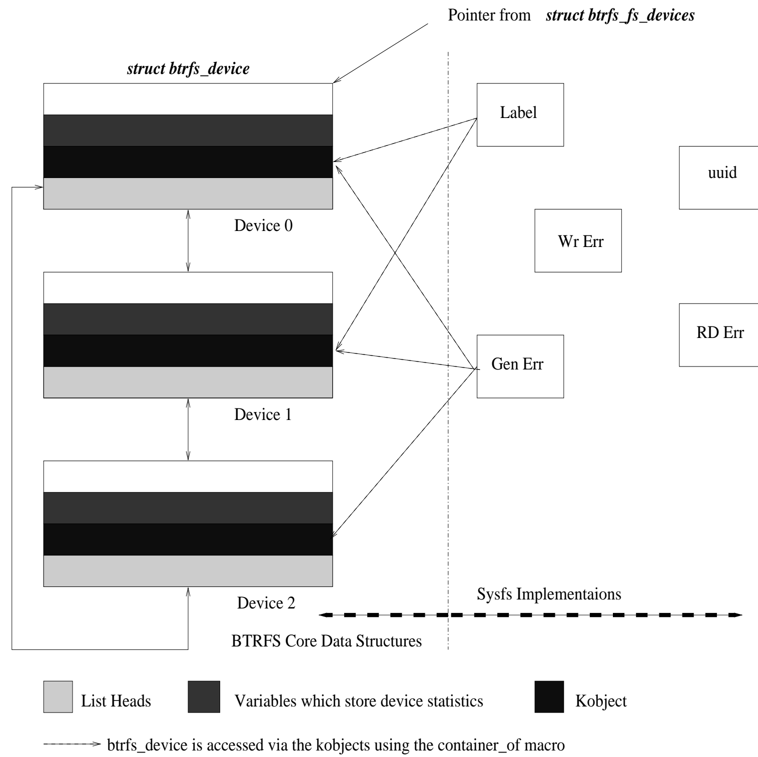


Figure 5.1: Connection between core BTRFS components and the Sysfs interface

5.2 The BTRFS sysfs Interface

The primary goals while designing the Sysfs structure were

- It should be possible to extend it easily in the future.
- Flexibility

Keeping in mind the above points we came up with the following directory structure for Sysfs

- BTRFS
  - Devices
    - hdb1
    - hdb2
      - uuid
      - label
      - cnt_corruption_errs
      - cnt_flush_io_errs
      - cnt_generation_errs
      - cnt_read_io_errs
      - cnt_write_io_errs
  - Info
  - Health

In the above representation *hdb1* and *hdb2* are BTRFS devices which have been mounted. Every time a device is mounted/unmounted its respective entry from the Device directory is dynamically added/removed. The attributes inside each device entry represent the type of errors which this framework in capable of currently monitoring. Figure 5.2 shows a diagramatic representation of the sysfs directory structure.

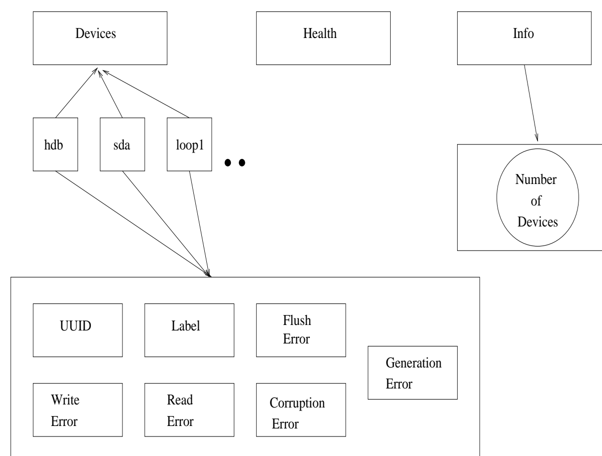


Figure 5.2: sysfs Structure

Chapter 6

Device Statistics

6.1 Keeping count of the disk errors

The device statistics patch provided by Mr. Stefan Berhans aims to enable the system administrator using BTRFS to keep track of device errors. Knowing the number of disk errors that have occured, the administrator can decide whether the disk is robust or not, and accordingly decide whether to replace the system disk or not. This feature is very much required, as not knowing the health of the disk could result in serious loss of data at a later stage, beyond which the loss would not be recoverable.

The errors being counted are:

1. Read Errors
2. Write Errors
3. Flush Errors
4. Corruption Errors
5. Generation Errors

Each of the above errors do not necessarily imply that the disk is failing, but frequent and high number of occurrances of these errors does indicate that the problem is in the hardware and a change might be needed.

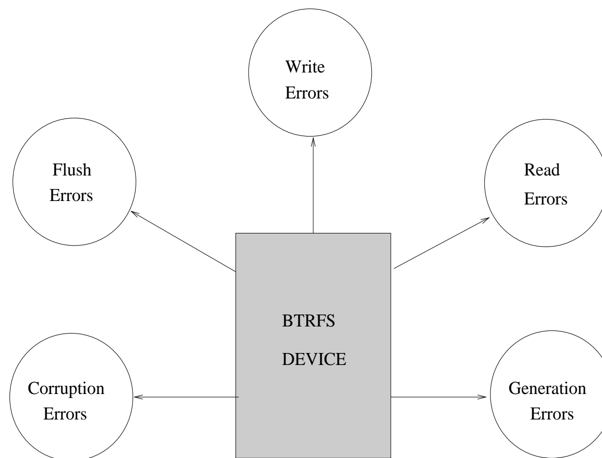


Figure 6.1: Device Stats

6.1.1 IOCTL vs Sysfs

The errors are being tracked within the kernel, and for it to be useful, it needs to be exported to userspace, so the system administrator can read them. This was initially done using an IOCTL interface, where each error was associated with an IOCTL command. A simple C program can use this interface to read the error counts. It is inconvenient to always have to write a C program to import any kind of data into userspace. Thus, BTRFS developers decided to do away with IOCTL and instead have a simpler and flexible interface via Sysfs.

Sysfs simply provides a directory structure with files associated with kernel data. The kernel writes to these files via the interface and the user reads these files for the exported kernel data. The reading of the data also invokes the Sysfs interface. Each file in this structure contains one value. The user simply performs a *cat* operation on the file to read the value.

In our *sysfs* implementation for device statistics, we associate each type of error with one file in the directory structure. Thus, if the administrator wants to know the number of write errors for a particular BTRFS device, he simply has to run *cat* on the appropriate file in the *sysfs* structure.

6.1.2 Patch Details

All the five errors - read,write,flush,corruption,generation - are kept track of via five variables in *struct btrfs_device*. Each occurance of the error simply triggers an increment operation on the appropriate variable.

The error is counted whenver any of the above operations in BTRFS returns an input/output error. An input/output error is indicated by a return value of *-EIO*. For example, if a write on a BTRFS device fails, the operation or it's function returns *-EIO*, a negative value. When this happens the variable keeping track of write errors is incremented by 1. Whenever the system administrator performs a read operation on the *sysfs* file for write errors, the sysfs interface code is invoked, and the newest value is shown. When the device is unmounted, the updated values are written to the device tree on the disk. When mounted again, the sysfs initialization code for the device is invoked, which takes care of retrieving the previously stored values.

On the sysfs interface side, each error variable has associated with it a *show* function. This function is invoked when the user performs a read operation on the file associated with that particular variable. It is passed as parameter a *kobject* variable,from which the function accesses the error variables within *struct btrfs_device*. The value that is read is the latest value of that variable, and is printed for the user to see.

Within *struct btrfs_device* are the five variables associated with the five error counts. We simply have print this value for the user to see. This printing is invoked when the user does a read operation on the files associated with the error variables. Specifically, a generic *show* function is invoked which calls the *show* function of the specific variable. *show* functions are the essence of the *sysfs* interface.

As said before, the *sysfs* initialization function takes care of setting up the interface, creating the directory structure, retrieving previously stored values, 'registering' the *show* functions etc.

Chapter 7

Device Statistics Patch - Testing

7.1 Approach and Design - Use of Layered OS structure

The device statistics patch counts the number of read,write,flush,corruption and generation errors. In-order, to test the correctness of the patch, these errors need to be generated on a BTRFS device and check if the values returned in the Sysfs files are correct.

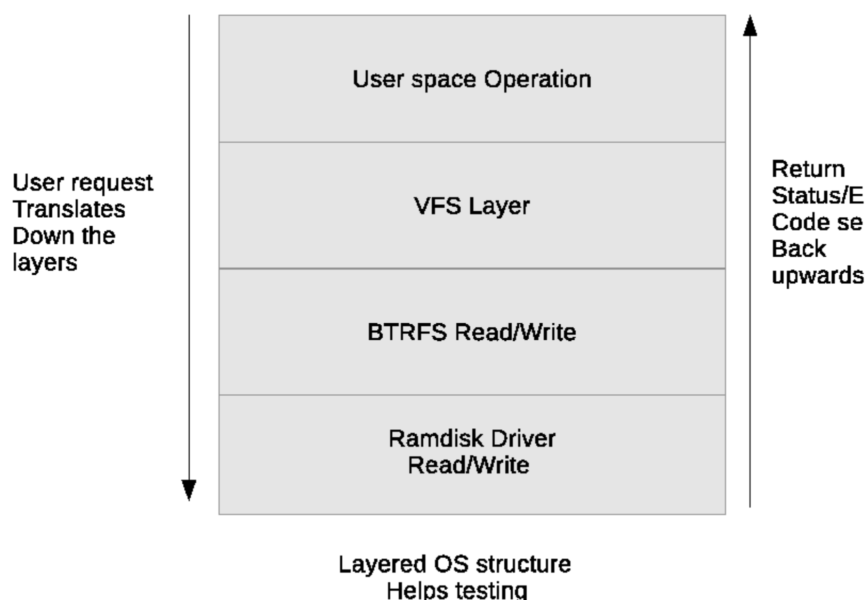


Figure 7.1: Layered OS Structure

As shown in Figure 7.1The operating system has a layered structure. It means that the file system uses the services of the software beneath it in the structure. The layer below the file system, is the block

device layer. Thus, whenever the FS performs a read or a write, it invokes the services of the block device layer. A block device is a device that communicates data as fixed-size blocks. Each block device, like a hard disk, has a driver that interacts with the file system layer.

Typically, a read/write operation on the FS translates to a corresponding read/write operation in the driver. If the read/write in the driver results in any error, the error code is returned back to the FS layer. The FS layer will either act upon the error, or choose to simply return the error code to the layers above, where it might get handled.

Thus, inorder to test the device statistics patch, we choose to force the layer below - the block device layer - to return an IO error code, so that BTRFS registers it as a genuine read/write error and updates the appropriate error count. Effectively, we simulate a read/write error at the block layer, fooling the FS layer to think that a genuine error has occured.

7.1.1 Ramdisk

The simulated-error method described above can be done using a simple ramdisk implementation. A ramdisk is a block device just like a hard disk, but it uses the RAM as it's memory. In other words, a read operation on the ramdisk fetches data from the RAM allocated to the ramdisk, similarly, a write operation writes data to the RAM. In effect, a section of RAM behaves like normal hard disk. In this section of memory read/write occurs in units of block sizes as dictated by the FS above it, and the read/write in this area is handled by the corresponding block driver written for it. A ramdisk of 1 Megabytes, will use 1 MB of contiguous memory from RAM for it's operations. This 1 MB of RAM is viewed by the higher layers just like any other hard disk, and likewise it gets an appropriate entry in */dev*. Just like any other hard disk, we can format it with a file system, mount it, create and delete files in it etc. The user is abstracted from the fact that he is working on a section of RAM and not a true block device.

For testing our patch, we fashioned a simple ramdisk which utilizes 256 MB of contiguous RAM. The driver of the ramdisk has operations to read/write data to and from this 256 MB section. These operations are carried out in blocks of size 512 Bytes. Under normal operation, this disk acts like any hard disk and can be formatted with any FS. We however, need this disk to act as an erraneous device, so that the FS layer above it, thinks that the disk is at fault, and records the read/write errors. If the patch

is working, the Sysfs interface files should show the correct number of errors that were simulated this way.

Note: The minimum device size for BTRFS is 256 MB. Hence in order to obtain a contiguous memory of 256 MB using *vmalloc* we were required to pass *vmalloc=256 MB* as a boot parameter to linux kernel at boot up. This had to be done because the kernel always has a upper bound on the amount of virtual memory that can be allocated. This threshold was too small for us and had to be changed.

But at the same time, we also need to control the frequency of these errors. We don't want the ramdisk to always return an error code, because that would also cause basic tools like *mkfs.btrfs* to fail . We wish to be able to command the ramdisk when to return an error and when to function properly. This we achieve by adding a simple IOCTL interface to the ramdisk driver. We run a C program, that uses this interface to toggle on/off certain flags in the driver code, via IOCTL commands. Based on the state of the flags, the driver behaves erraneously or behaves as a normal device.

For example, if we use a C program to use the IOCTL interface of the ramdisk to set the *write_error_flag* to 1, then for all future write operations on the ramdisk, the driver will return *-EIO*, indicating an IO error. We can use the same interface to clear *write_error_flag* to 0. When this is done, the ramdisk will operate correctly,without returning errors. Similarly we have flags associated with each type of error. Thus we can command the ramdisk to simualate any error at any time we want.

7.1.2 Final Testing

Read/Write/Flush Errors Testing steps

1. Add *vmalloc=256 MB* or greater to boot options of the linux kernel. For example, the options to your linux kernel and boot time would look something like

   linux /vmlinuz-3.0.4 root=/dev/mapper/SH-root vmalloc=256M ro single Kindly refer to note in 7.1.1 for explaination.

2. Insert the ramdisk module. The ramdisk initially performs normally. When the module is inserted, the ramdisk, like any block device becomes visible under */dev* directory of linux.

3. Format the ramdisk as a BTRFS device using *mkfs.btrfs*. Thus we have made the ramdisk a BTRFS device. What this means is, all read/write operations on the ramdisk will call the corresponding

read/write operations of BTRFS, which then translates to read/write operation implemented within the ramdisk driver.

4. Mount the ramdisk device. At this stage, the Sysfs error files should show zero errors. Mounting the device triggers creation of *Sysfs* file structure of the mounted device. Associated with every device is a directory, under which we have the files that store the error counts maintained by the patch.

5. Run the C program to toggle the error control flags of the ramdisk via IOCTL commands. Having set the appropriate flags, the driver will now return IO error code for read/write operations.

6. Perform random read/write operations on the mounted device. Since the driver returns IO error codes, the BTRFS layer thinks that the device is at fault, and thus starts counting the read/write IO errors of the device. This can be verified via the Sysfs interface files for read and write errors.

This testing procedure can be used to simulate read/write/flush IO errors.

Thus we have used the layered structure of the OS, to test the patch.

User operations on the ramdisk first invoke the functions in the *Virtual File System* or *VFS*. This call invokes the appropriate BTRFS function because we have formatted the ramdisk as a BTRFS device. The BTRFS function then invokes the read/write operations within the driver code of the ramdisk. It is at this point that we simulate the error. The driver functions can be made to work normally or be forced to return an error code, at our will.

Corruption Errors Testing steps

1. Create a BTRFS device by using *mkfs.btrfs* and mount it.

2. Overwrite the device file of the device,under */dev* by using the *dd* command. This dumps random data on the device file, while it is mounted. This causes the data on the device to get corrupted. The BTRFS layer records these corruption errors and increment the appropriate variable for corruption error.

This is a corruption error, because while the device is mounted, the data in it gets written on, from a source of which the BTRFS layer is not aware. The disk dump operation thus corrupts the data. This corruption is detected using the checksum values. Sine the data is overwritten, the data and the

checksum fail to match, causing the BTRFS layer to think that the data is now corrupt. If the patch is working, this causes an increment in the variable that counts the number of corruption errors. This change become visible in the Sysfs interface file for corruption error.

The above 2 testing procedures verify the device statistics patch and it's Sysfs interface.

Bibliography

[1] Btrfs *sysfs* git repository. http://goo.gl/uLWgE.

[2] Btrfs homepage. http://btrfs.ipv5.de.

[3] Linux source code git repository. http://git.kernel.org/?p=linux/kernel/git/torvalds/linux.git.

[4] Patrick Mochel. The sysfs filesystem. www.kernel.org/pub/linux/kernel/people/mochel/doc/papers/ols-2005/mochel.pdf.

[5] Oracle Solaris. Zfs homepage. http://hub.opensolaris.org/bin/view/Community+Group+zfs/.