College of Engineering, Pune - 5
Dept. of Computer Engineering and IT
B.Tech. Project: VII Semester - Progress Report

Adding Features To BTRFS

Gautam Akiwate
Information Technology
akiwategs08.it@coep.ac.in
Roll no: 110808006

Sanjeev Mk
Information Technology
sanjeevmk@gmail.com
Roll No: 110808019

Shravan Aras
Computer Engineering
arassg08.comp@coep.ac.in
Roll no:110803073

15-11-2011

Abstract

The Operating System Kernel is probably one of the most fundamental piece of software that is instrumental in keeping a system running. Besides providing the general tasks of scheduling,interrupt handling and resource management, the kernel also has a section dedicated to storage of data, the section that keeps track of files stored onto the hard disk. This section of the kernel is called the file system. It maintains various data structures which keep track of the data on the disk. Traditional file systems were supported by Logical Volume Manager (LVM) which managed the disk partitions on which the file systems were mounted. However, the move to next generation file systems comes with the amalgamation of the LVM with the actual file system. The B-Tree File System also called btrfs is a next generation file system that implements this idea in combination with the concept of storage pools and copy on write system, ideas which make it the file system of choice in the foreseeable future. The main purpose of our project is to add features to btrfs which will allow the system administrator to adjust various tunable filesystem parameters. This needs to be done via the `sysfs` interface which is not available as of now on btrfs. Thus the project first aims at developing a `sysfs` interface for btrfs using which the additional features will be added.

Contents

1 Definition of Problem Statement

The project aims to enable users to configure various BTRFS parameters after mounting the filesystem. It's a parallel to tune2fs which provides the same feature to ext 2/3/4 filesystems. For example, using tune2fs the user can enable/disable journaling in a mounted ext4 FS. We aim to provide a similar BTRFS tool that users can use to configure BTRFS parameters dynamically.

This can be done by adding sysfs support to BTRFS. The user space application will write the desired values of the configurable parameters in the approriate sysfs files. The kernel reads the values from these files and sets the values to these BTRFS parameters. For example, we can turn on and off debugging support.

2 Literature Survey

2.1 Study of Linux Kernel

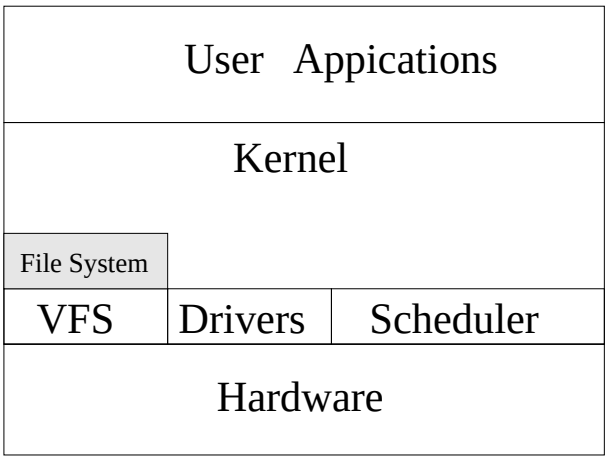

Figure 1: Basic System Block Diagram

The diagram 3.2 shown above depicts various components of a normal system. We shall cover the following topic - VFS, Scheduling, Drivers and Process Management in detail in next few sections. However it is worth differentiating between User Mode and Kernel Mode when it comes to kernel operations. The x86 architecture offers four different privilege levels for programs to run on with privilege level 0 housing the Linux Kernel(Kernel Mode) and privilege level 3 housing user processes(User Mode). The user processes interact with the kernel using `system calls`.

2.2 Brief Overview of System Calls

System calls are handled by the kernel as a special type of exception and exceptions in turn are handled by hardware interrupts. The hardware interrupt `0x80` is invoked when a system call is made, this interrupt points to the vector 128 in the IVT(Interrupt vector table). However to determine the exact type of system call a system call number is stored in the `eax` register. A example 2.2 depicting the exact path taken by a system call is show below. As depicted in the example when a call to `write` is made a `libc` wrapper

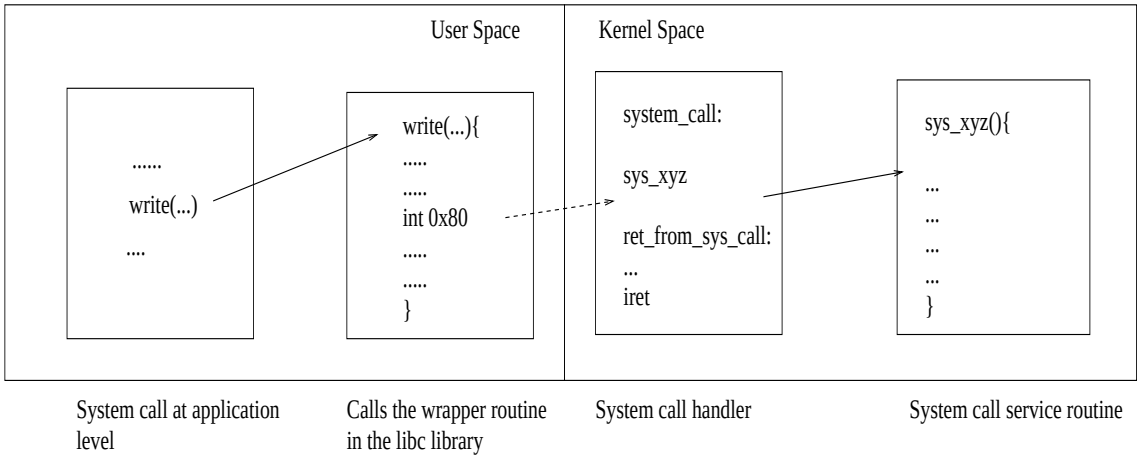

Figure 2: Path traced by a system call

is called. This wrapper then invokes the interrupt through the assembly instruction `int 0x80` and the switch to kernel mode is made. The control is then passed to `system_call`, who's structure is similar to that of an exception handler. The system_call looks through the system call dispatch table, which for security reasons is located inside a page having read only permissions, to find the system call service routine which must be invoked for that system call number.

2.3 Process Management in Linux Kernel

Every user space application runs as a process in the linux kernel. The processes are linked to each other by a parent child relationship, `init` being the first process which spawns other processes. The `tast_struct` represents the complete process descriptor, containing pointers to open files, pending signals, array of pointers to maintain parent child relationship, etc. All process are linked together in a doubly linked know as process list. The kernel also maintains a list of running process. Every process is given a `pid` which is stored inside a `pidhash` table. The `tarray_freelist` points to a chain of pointers which links all the free slots of the `pidhash` table. This enables new pid's to be allocated to a process in a fast and efficient manner. Processes which are in a state of waiting are stored in the `wait_queue` state making it simpler for the schedule to resume these process.

2.4 Interrupts in Linux Kernel

Devices usually need to interact with external devices and more often than not in a time frame that is significantly different to its own. Due to this significant differences in the time frames it is undesirable that the faster devices wait for slower devices or external events. To implement this functionality interrupts were designed. An interrupt is simply a signal that the hardware can send when it wants the devices attention.

To handle an interrupt a driver needs to register the handler and invoke it appropriately as and when the interrupt arrives. However, interrupt handlers are somewhat limited in their actions due to the way they are run. Interrupt handlers, by their nature, run concurrently with other code and hence raise concerns of concurrency and contention.

As mentioned above a software handler must be configured in the system to handle an interrupt. If this hasn't been done the kernel acknowledges the interrupt and ignores it.

Moreover, interrupts are received on interrupt lines which are a precious and often limited resource. The kernel keeps a registry of interrupt lines and a module when it defines a handler is expected to request an interrupt channel to use it and must also release it once it is done. It is also possible that the interrupt lines are shared with other drivers.

The interrupt handler depending on its use can be installed either at driver initialization or when the device is first opened. Due to the limited number of interrupt lines it is important to install the handlers as and when needed. This becomes even more important in case of drivers which do not share interrupts. If a module requests an interrupt line at initialization, it will prevent any other driver from using the interrupt line, even if the device holding it has never used it. Thus requesting the interrupt at device open allows sharing of resources and should usually be adopted.

The other major problem with interrupt handling is the question of performing lengthy tasks within the handler. Usually, a significant amount of work needs to be done in response to the interrupt, but the interrupt handlers need to return quickly and not block the interrupt for long. This conflict between work and speed is resolved by splitting the handler into two halves. The top half is invoked on the interrupt. The bottom half is scheduled by the top half to be executed later, at a safer time. This split allows the top half to service new interrupts even when the bottom half is working.

2.5 Introduction to VFS

The Virtual File System is the access point/interface to the various file system provided in the linux kernel. All filesystem accesses go through the VFS. The VFS provides a generic interface for applications to perform common file operations. Different file systems provide different implementations of file operations. But the layers above the file system need not be aware of this diversity. The VFS provides a common file system layer for everyone to use, without being aware of what exact file system they are using. If seen as a layered architecture, the VFS is placed above all the file systems. All file system requests are intercepted by the filesystem, and appropriately translated to specific file system calls. For example, if mkfs.ext4 is called, the VFS intercepts this request, realizes that this call is for ext4 and asks ext4 to perform the desired task. Thus, other components of the kernel and all other higher layer softwares, do not make SF-specific calls. A generic call interface is provided by the VFS and this is what is used to request SF services.

Some important data structures used in the VFS include the structures for the superblock, the inode, file and dentry. The superblock structure contains all information about a mounted filesystem. The inode object has information about every file. The file structure has information of the currently open files. Lastly, the dentry object represents all the directories.

2.6 `sysfs`

`sysfs` is an in-memory file system. The file system provides a set of configuration files for different subsystems of the kernel. These files provide an interface to user space to configure various parameters of these file systems. User space applications write values of parameters in these files, which the kernel reads later an sets the desired values. Each subsystem that wants to use sysfs firsts registers to sysfs. After registering, the subsystem

can create it's own directory in `/sys` which will have the configuration files in it. For example, the block device subsystem has a directory `block` in `/sys`. This directory has various configuration files, which can be written to, if the user wishes to tune some parameter. `sysfs` basically provides an interface between the kernel and user space. The first task of our project would be to add sysfs support to BTRFS.

2.7 Compiling Linux kernel-Classic way vs dpkg

We mainly used two different ways to compile the kernels we experimented on. Here we provide pros and cons of both the methods. All kernel source code used to compile was taken from kernel.org website. The kernel needs information about its surrounding and the way in which to compile the modules. This information is feed to the kernel compilation scripts (which recursively call the makefiles from each directory) via the .config file. There are various options in which new configuration files can be populated.

1. If you wish to use your old configuration to build the new kernel
   `make oldconfig` can be used.
   However new features are added with with the launch of newer kernels. Their entries will not be present in the old config and hence the kernel scripts will prompt the user to configure these features when make oldconfig is run.

2. Another way of populating the configuration file is to run
   `make menuconfig`
   This provides a ncurses based UI which enables the user to select device drivers, what modules to compile into the kernel itself and which ones to compile as external ones, putting them in /lib/modules. The menu allows the programmer to opt for cross compiling his kernel if the appropriate cross compilation tool chain has been set up.

The process in which the configuration files are populated are common to both the classical approach and the dpkg based one.

2.8 Classic way

Largely 4 command are instrumental in compiling the kernel and installing it.

```
make
make bzimage
make modules_install
make install
```

However the user has to manually add the kernel entries into his boot loader. In case of grub this can be done by passing the compressed kernel image along with its parameters(root filesystem location and device on which root filesystem is located) to the `linux` command and passing the initial ram fs image to the initd command.

2.9 dpkg way

The dpkg method creates a kernel .deb package which can then be easily installed using `dpkg -i`. This process is relatively easy and does not require the user to edit his boot loader configuration script. `make-kpkg clean`

```
make-kpkg -initrd kernel_image kernel_headers
```

The above commands will create 2 deb files, one for the image and other for the header, which can then be installed using `dpkg -i`. Because the package manager keeps track of these files as packages the user can easily remove a kernel using `dpkg -r`.

2.10 Linux Kernel Drivers

As nearly every system operation that is executed on the system maps to a physical device it is imperative that the device drivers be designed with care as loopholes could prove to potential security hazards. In most of the systems nearly all the operations on the particular devices are done by code that is specific to the device. This code is called as a device driver. For a system to be functional the kernel must have embedded in it a device driver for every peripheral present on the system.

Device drivers play a very special role in the Linux kernel. They are considered to be an interface the hardware with the software programming interface. The programming interface is abstracted by an API which ensures that the calls are independent of the driver. The device driver maps these calls to the device specific operations. Device Drivers is the Linux kernel are highly modular in design that makes them robust.

Essentially it performs the role of layer that lies between the applications and the actual device. This significant role of the driver gives it a lot of privilege and the driver can determine the capabilities it can offer to the applications. There are a lot of considerations that go into the driver design and depends on the policies that need to be implemented.

2.11 Block Drivers

Block drivers interact with their devices by transferring data in blocks of fixed size. Block drivers provide an interface to work with block devices like disks. Each access to a block device is seen by the kernel as a request. For each request, the kernel builds a `request` structure, which contains all the information about the request, like starting sector, number of sectors etc. Each `request` structure contains a linked list of `bio` structures. Each `bio` structure on the other hand, contains an array of `bio_vec` structures. The `bio_vec` structure contains information about the memory page for the request, number of bytes to transfer into or from this page etc. This is how each `request` structure is built from the very basic memory page information.

Further, each such request structure is then queued by the kernel in `request_queue`. The `request_queue` is a queue of requests made to use the block device. This queue is associated with a request function which is defined in the driver. The kernel calls the request function whenever it thinks that it is time to service the requests in the queue. The request function, when called with a pointer to the queue as a parameter, can then do it's job. This job is to process the requests in the queue one after the other. What the function does, and how it does it, varies from device to device, driver to driver.

2.12 kernel OOPS and bugfixing

Whenever the kernel detects the presence of a bug, it generates an OOPs. An OOPs kills the process that triggered the bug, and the kernel exits the process, followed by printing

as many details as possible about the bug.
An OOPs includes the details of all the registers of the processor at the time of the bug, the contents of the stack, the contents of the program counter, the opcodes of the instructions in the vicinity of the bug instruction, and a call trace showing the function calls that led up to the bug. All this information can be used to find the exact location of the bug in the code. Things that could help find the exact location in the C code include doing an objdump on the said C file, and having the assembly code for that C code output in a .s file. Analyzing the C code and the output of objdump (disassembled code), we can find the line of bug in the kernel code.

Based on this knowledge, we worked on bug #36172 of the linux kernel. The occurance of this bug causes a kernel OOPs. The bug is produced when an already mounted read-only filesystem, is remounted after disabling journaling and changing the permission to read-write. The remounting is done as:
`mount -o remount,rw device_file`
The device was earlier mounted as read-only and with journaling. The journaling was disabled by using tune2fs, and then remounted by the above command. This causes the kernel to kill mount and print an OOPs status.

Our analysis of the bug:
`sys_mount` call is redirected to `do_mount`. In `do_mount` the various mount options are parsed. This detects the `remount` option we had given, and calls `do_remount`. `do_remount` after doing some flag checking (or options parsing), calls `do_remount_sb`. `do_remount_sb` changes the parameters of a mounted FS, and calls `fs_remount`, which is a VFS call. This call is translated directly to `ext4_remount`, as we are working with ext4 FS here. `ext4_remount` does a remount with the new parameters. While remounting, it notices that we have set the read-write option. Changing of permissions requires clearing the journal of any previous errors, before actually making it read-write. Thus it calls `ext4_clear_journal_err`. In this function, there is a `BUG_ON` macro, which checks a flag that indicates compatible features. In this check, it is noticed that journaling has been disabled, thus the clearing of journal errors can't proceed. This triggers a bug or OOPs. The kernel kills the mount process.

One easy fix, is to check for the compatible features, before actually calling, `ext4_clear_journal_er`
So if journaling is disabled, there is no point in calling this function, as there is no point in clearing old journal errors,as the journal is anyway not going to be used. Another fix would be not let tune2fs disable journaling in a mounted file system.

3 BTRFS

3.1 An Overview

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 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.

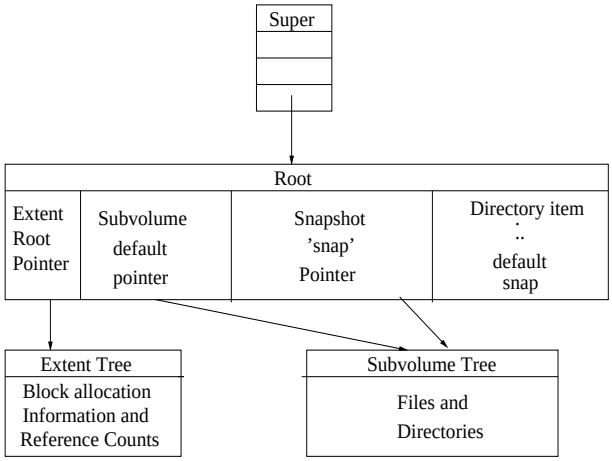

Figure 3: BTRFS Tree

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.

BTRFS Data structures

```
struct btrfs_header {
u8 csum[32];
u8 fsid[16];
__le64 blocknr;
__le64 flags;
u8 chunk_tree_uid[16];
__le64 generation;
__le64 owner;
__le32 nritems;
u8 level;
}
struct btrfs_disk_key {
__le64 objectid;
u8 type;
__le64 offset;
}
struct btrfs_item {
struct btrfs_disk_key key;
__le32 offset;
__le32 size;
}
```

4 Acknowledgements

We would like to thank College of Engineering, Pune and our project mentor Professor Abijit M. for providing us with an opportunity to pursue this project. We extent our gratitude towards Nishchay Mhatre, Neependra Khare and Amit Shah for providing us with valuable suggestions and support.

References

[1] Understanding the LINUX KERNEL
Daniel P. Bovet and Marco Cesati.

[2] Linux Device Drivers
Jonathan Corbet, Alesandro Rubini and Greg Kroah-Hartman

[3] Unix Internals
Uresh Vahalia

[4] Workload Dependent Performance Evaluation of the Btrfs and ZFS Filesystems", DHTechnologies
Heger Dominique
http://www.dhtusa.com/media/IOPerf_CMG09DHT.pdf

[5] Btrfs Design, Oracle
http://oss.oracle.com/projects/btrfs/dist/documentation/btrfs-design.html

[6] The Btrfs Filesystem, Oracle
http://oss.oracle.com/projects/btrfs/dist/documentation/btrfs-ukuug.pdf