Crossmedia File System *MetaFS* Exploiting Performance Characteristics from Flash Storage and HDD

Bachelor Thesis

Leszek Kattinger

12. November 2009 – 23. März 2010

Betreuer: Prof. Dr. Thomas Ludwig
Julian M. Kunkel
Olga Mordvinova

Leszek Kattinger
Hans-Sachs-Ring 110
68199 Mannheim

Hiermit erkläre ich an Eides statt, dass ich die von mir vorgelegte Arbeit selbstständig verfasst habe, dass ich die verwendeten Quellen, Internet-Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Mannheim, den 22. März 2010

_____

Leszek Kattinger

Abstract

Until recently, the decision which storage device is most suitable, in aspects of costs, capacity, performance and reliability has been an easy choice. Only hard disk devices offered requested properties. Nowadays rapid development of flash storage technology, makes these devices competitive or even more attractive. The great advantage of flash storage is, apart from lower energy consumption and insensitivity against mechanical shocks, the much lower access time. Compared with hard disks, flash devices can access data about a hundred times faster. This feature enables a significant performance benefit for random I/O operations. Unfortunately, the situation at present is that HDDs provide a much bigger capacity at considerable lower prices than flash storage devices, and this fact does not seem to be changing in the near future. Considering also the wide-spread use of HDDs, the continuing increase of storage density and the associated increase of sequential I/O performance, the incentive to use HDDs will continue. For this reason, a way to combine both storage technologies seems beneficial. From the view of a file system, meta data is often accessed randomly and very small, in contrast a logical file might be large and is often accessed sequentially. Therefore, in this thesis a file system is designed and implemented which places meta data on an USB-flash device and data on an HDD. The design also considers, how meta data operations can be optimized for a cheep low-end USB flash device, which provide flash media like fast access times but also characteristic low write rates, caused by the block-wise erase-before-write operating principle. All measured file systems show a performance drop for meta data updates on this kind of flash devices, compared with their behavior on HDD. Therefore the design focused on the possibility to map coherent logical name space structures (directories) close to physical media characteristics (blocks). To also check impacts by writes of data sizes equal or smaller then the selected block size, the option to write only full blocks or current block fill rates was given. The file system was implemented in the user space and operate through the FUSE interface. Despite of the overhead caused by this fact, the performance of write associated meta data operations (like create/remove) was better or equal than of those file systems used for benchmark comparison.

Contents

Chapter 1

Introduction

A computer system is primarily designed to store, retrieve and process data. Before and after data is manipulated in the main memory (RAM) by the central processing unit (CPU), it has to be retrieved from and stored on a persistent storage device. These input and output devices (collectively termed I/O) can keep huge amount of data over a long period of time. In a modern computer system the CPU can process data much faster then the I/O subsystem can deliver it. While storage capacity and the amount of data increases, this performance gap tends to be more and more significant and forms a bottleneck for applications associated with data storage.

1.1 Motivation

Today persistent data storage is commonly based on magnetic disks (platters), of which a so called hard disk device (HDD) is composed. These devices have a long history in computer data storage, and are therefore widespread. It is also fair to say that they are well-engineered, durable and much cheaper in comparison to other other storage devices, like flash storage. Furthermore they show better I/O performance, e.g. in sequential writes, than average flash storage devices. But their performance is limited by the fact that they consist of several mechanical parts, which have to be moved during I/O operations. The process of moving the read/write head from one position of the disk to another, an operation known as a seek, is time-consuming. This seek, or more general access time, keeps nearly constant over the last years, and it seems to be not possible to decrease this delay without disproportional effort of electrical power and risks of pushing things to the limit, like speedup disk rotation or head acceleration. Nowadays a single disk access can take about 10 milliseconds $(10^{-2}\ s)$ – still a small number, but about 1 million times longer than a single RAM cycle, which takes about 10 nanoseconds $(10^{-8}\ s)$.
Given the disparity in performance, it would be inefficient to send all I/O requests to the disk in the order in which they are issued. Therefore, operating systems and disk controllers implement caches and I/O scheduling algorithms, which try to minimize the number and size of disk seeks by manipulating the order in which I/O requests are serviced. These hard- and software schedulers lessen the performance penalties associated with disk access, but their capabilities are restricted by combinations of unpredictable demands and various physical data layouts.

In the last few years flash devices became more and more popular as persistent storage components in computer systems. They offer currently much lower access time than hard disks drives, but also often lower write performance. Today USB flash storage devices with low capacity are quite affordable. Furthermore they have a linear access time of approximately 0.1 ms [1], which is one hundred times faster than average disk access time. Flash storage also provides good read performance, some solid state drives outperform hard disks in several factors [2]. Write throughput on flash media differs and is up to the technology and model. Some SSD models can compete with HDDs or outperform them. Others, e.g. the commodity SSDs, show less performance in this case. The cost/capacity ratio decreases in the meantime, but from the vantage point of the present, it will take some time to replace hard disks entirely [1].
To achieve better overall I/O performance and cost benefits, a combination of both storage media types seems to be encouraging. One way to exploit the advantages and reduce penalties of flash and disk storage is to split data by request patterns. Two basic data access types fit the characteristic distinctions: random and sequential. This kind of data usage is for a multi purpose, multi user data system is hardly predictable, but some file system statistics say:

Most files are read, not written and the ratio of reads to writes is often 4-6. The most popular file system operation is `stat`, which results in a file `lookup` and `lookups` account for 40-50% of all file system operations[3].

Technically spoken a `lookup` of a file system's object results, if not found in cache, in a random read operation on the storage device. The amount of meta data associated with a single file object is small and limited, seldom over 100 Bytes. On the contrary, file objects content is often big and (in newer file systems) only limited by the storage device capacity. Furthermore, rather sequential `read/write` operations on regular files are usual. To speedup file access and data access as well, it would be beneficial to store meta data on a flash device and data on a hard disk.

Since I/O requests themselves are complex operations, in which many different soft- and hardware systems and their interfaces are involved, general performance optimization is often a difficult task and the impact is generally difficult to predict, especially when data safety is also an issue. Assumed, benchmark environment stays the same, two major factors determine I/O performance: file systems and their underlaying storage devices. To validate this concept a flash-optimized meta data file system needs to be created.

1.2 State-of-the-Art and Related Work

There are various different file system approaches to store data. In the following are mentioned the most common. Popular file systems are especially designed to map data on hard disks. These are for example: NTFS, FAT32, XFS and the Ext-family. It is also possible to use them with flash storage devices, but the performance is in some cases, like meta data updates, not adequate. Especially the FAT32 file system comes along with every new USB-flash device, but this is more due to market dominance than suitability. The main performance problem for this class of file systems is that they do not consider flash specific technology. As flash

[1] `http://www.storagesearch.com/easyco-flashperformance-art.pdf`
[2] `http://www.intel.com/design/flash/nand/extreme`
[3] `http://www.filesystems.org/docs/sca/node4.html`

storage becomes more available in the recent years, it is deployed by many systems. Flash storage is mainly made of NAND-Gates, which requires to erase a whole block before writing any data to it. This slows down the process of writing data that fits in one block but is written in two or more, by the same factor. To exploit NAND properly, specific flash file systems exist [2, 3, 4, 5] that are designed to use blocks, a unit the same size as the erase segment of the flash medium. Flash file system differs from regular file systems by operating directly on a flash device (e.g. MTD in Linux) instead of an intermediate block device.

JFFS2 (Journalling Flash File System version 2) [2] is a log-structured file system. All changes to files and directories are "logged" to flash in nodes, which can be either inodes and direntry nodes. Inodes contain a header with file meta data followed by file data. Direntry nodes contain directory entries, each holding a name and an inode number. JFFS2 does not guarantee 100% optimal use of flash space due to its journalling nature, and the granularity of flash blocks. It is possible for it to fill up even when not all file space appears to have been used, especially if files have had many small operations performed on them and the flash partition is small compared to the size of the flash blocks.

LogFS (Scalable Flash File System) [5] has a log structure based on a tree. Aims to replace JFFS2 for most uses, but focuses more on large devices. Wandering trees are used to allow out-of-place writes.

UBIFS (Unsorted Block Image File System) [3] is a successor to JFFS2, and competitor to LogFS, as a file system for use with raw flash memory media, it works on top of UBI devices, which are themselves on top of MTD devices. UBIFS supports write caching.

YAFFS2 (Yet Another Flash File System version 2) [4] is the latest of this file systems. Like the others it uses journaling, error correction, and verification techniques tuned to the way NAND typically fails to enhance robustness.

There are also several approaches that try to exploit beneficial features of different storage devices and data semantics. The Ext3 FS has the facility to store the Journal on a separate volume or device what discharged the I/O load of the file system volume[4].

Another approach of using advantages of flash storage, apart from file system implementation, and follow more a cache concept, is "Ready Boost" invented by Microsoft to improve performance without having to add additional memory [5]. It supposes a USB flash drive, to access data much more quickly than it can access data on the hard drive.

USB flash drives and SSDs consist of NAND flash storage accessed through a device controller. This Controller takes care about wear leveling and bad blocks. This technology provides a more convenient way to store data on a flash media. This work tries to use this advantage of a cheap and easy to handle USB flash storage device to combine following approaches.

[4] `http://e2fsprogs.sourceforge.net/journal-design.pdf`

[5] `http://www.microsoft.com/windows/windows-vista/features/readyboost.aspx`

1.3 Goals

The goal of this work is to design and implement a fast and secure data management system with a standard Unix file system interface (defined by POSIX[6] and provided by FUSE[7]), by exploiting the combination of data semantics and storage media characteristics in a useful manner. More precisely, to use a common USB flash storage device for storing file systems meta data. From a user perspective, every common file operation, like `create, delete, readdir`, should be possible and persistent with the exception of storing file content. Additional implementation and linkage of a data storage subsystem should be flexible and easy to add to existing code base.

1.4 Structure of the Thesis

This thesis is structured as follows. After the introduction to the topic and motivation of this work in chapter 1.1 we gave a short overview over the state-of-the-art on the field of the file systems relevant for our development. Subsequently, in chapter 2, we introduce some fundamental details necessary for a better understanding of the developed concept. Chapter 3 describes design and implementation details, followed by a description and discussion of test cases and benchmark results, of the developed file system, in chapter 4. Finally, chapter 5 summarize and conclude this work, and chapter 6 provides an outlook on future development.

[6] `http://www.pasc.org/whatispasc.html`
[7] `http://fuse.sourceforge.net`

Chapter 2

Background

This chapter gives a short overview about the background knowledge needed to design and implement this file system. It covers four major aspects: storage media, information representation, file system concepts and environment.

2.1 Storage Media Characteristics

Many different forms of storage, based on various natural phenomena, have been invented. Multiple storage devices can be generally divided in primary and secondary storage, each with an individual purpose. Primary storage (also referred to as main memory) contains actual processed data. It builds the short-term memory, where data can be easy and fast manipulated by reasons of a homogeneous address space with low latency and a high data bandwidth connection. Secondary storage devices are, by reason of efficiency and handling, accessible to an operating system via primary storage. They appear in contrast as long-term memory and are located in the peripheries of a computer system and connected through external buses. I/O refers mainly to data transfers to or from them. On many systems, main memory acts as an I/O buffer. Figure 2.1 shows the I/O component layout. All available storage media builds, concerning data accessibility, a memory hierarchy.

2.1.1 Memory Hierarchy

Memory hierarchy refers to a CPU-centric latency. The primary criterion for designing a placement in storage is a memory hierarchy, that fits the storage device into the design considerations. Most computers have a memory hierarchy, with a small amount of very fast, expensive, volatile cache memory, gigabytes of medium-speed, medium-price, volatile main memory, and hundreds or thousands of gigabytes of slow, cheap, nonvolatile storage on various devices.

2.1.2 Main Memory

The main memory also called random access memory (RAM) is constructed from integrated circuits and needs to have electrical power in order to maintain its information. When power

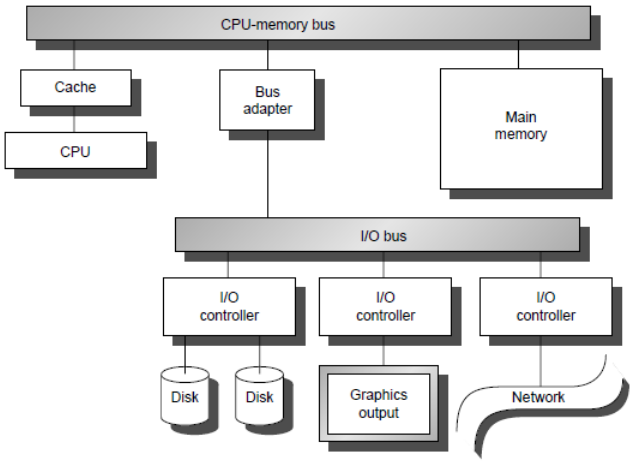

Figure 2.1: A typical interface of I/O devices and an I/O bus to the CPU-memory bus [6]

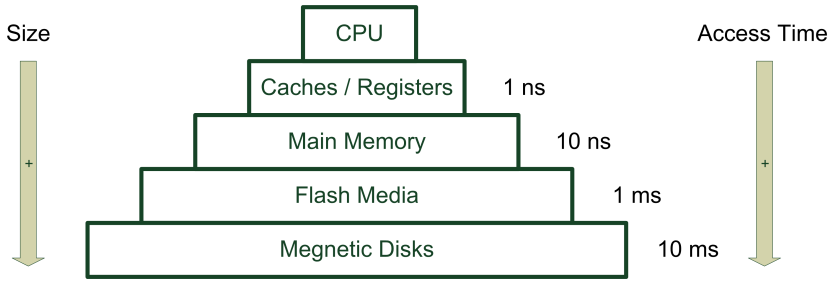

Figure 2.2: Memory Hierarchie

is lost, the information is lost too. It can be directly accessed by the CPU. The access time to read or write any particular byte are independent of whereabouts in the memory that byte is, and currently is approximately 10 nanoseconds (a thousand millionth of a second). This is broadly comparable with the speed at which the CPU will need to access data. Main memory is, compared to external memory, expensive and has limited capacity.

2.1.3 Flash Memory

Flash memory is a non-volatile computer storage that can be electrically erased and reprogrammed. It is a technology that is primarily used in memory cards, USB flash drives or solid state disks (SSDs) for general storage and transfer of data between computers and other digital products. It is a specific type of EEPROM (Electrically Erasable Programmable Read-Only Memory). It has a grid of columns and rows with a cell that has two transistors at each intersection.

The two transistors are separated from each other by a thin oxide layer. One of the transistors is known as a floating gate, and the other one is the control gate. The floating gate's only link to the row, or wordline, is through the control gate. As long as this link is in place, the cell has a value of 1. To change the value to a 0 requires a process called Fowler-Nordheim tunneling. The difference between Flash Memory and EEPROM are, EEPROM erases and

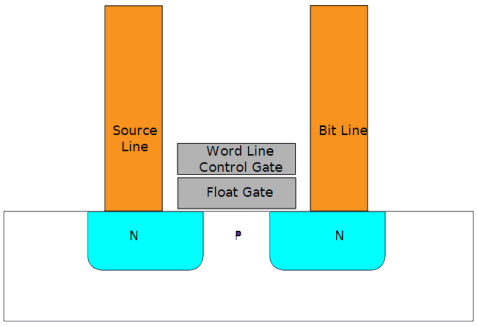

Figure 2.3: Flash cell structure([7], NOR cell structure)

rewrite its content one byte at a time, Flash Memory is erased and programmed in large blocks, which makes it a very fast memory compared to EEPROM.
Flash memory stores information in an array of memory cells made from floating-gate transistors. In traditional single-level cell (SLC) devices, each cell stores only one bit of information. Some newer flash memory, known as multi-level cell (MLC) devices, can store more than one bit per cell by choosing between multiple levels of electrical charge to apply to the floating gates of its cells.

Two different flash memory types exist - NOR and NAND. They can be distinguished by connections of the individual memory cells and interface provided for reading and writing as shown in figure 2.4 and 2.5.

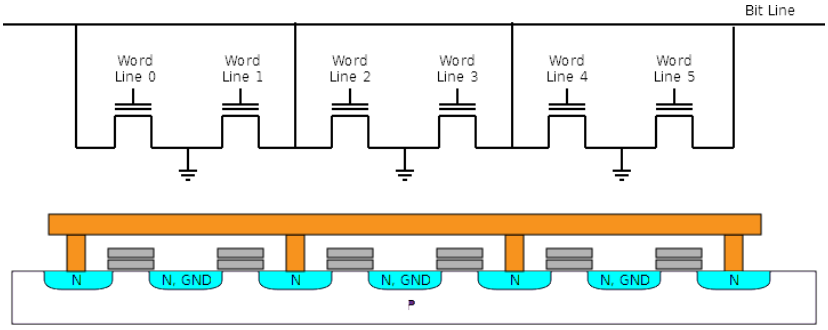

Figure 2.4: NOR Flash Layout ([7], NOR Flash Layout)

While NOR memory provides an external address bus for read and program operations (and thus supports random-access), unlocking and erasing NOR memory must proceed on a block-by-block basis. With NAND flash memory, read and programming operations must be performed page-at-a-time while unlocking and erasing must happen block-wise.

This type of flash architecture offers higher densities and larger capacities at lower cost with faster erase, sequential write, and sequential read speeds, what makes it the dominant technology for non-volatile solid-state storage.

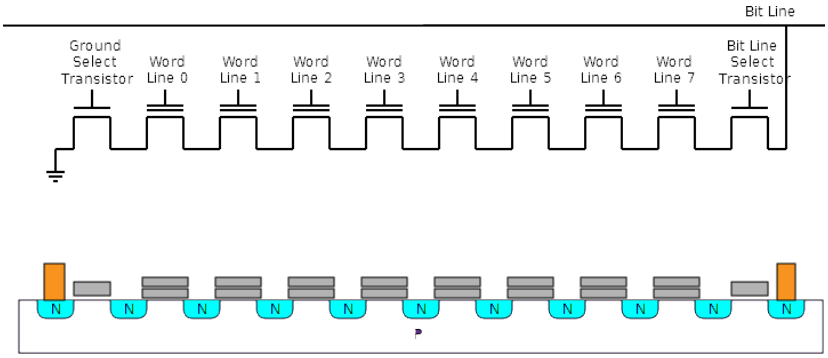

Figure 2.5: NAND Flash Layout ([7], NAND Flash Layout)

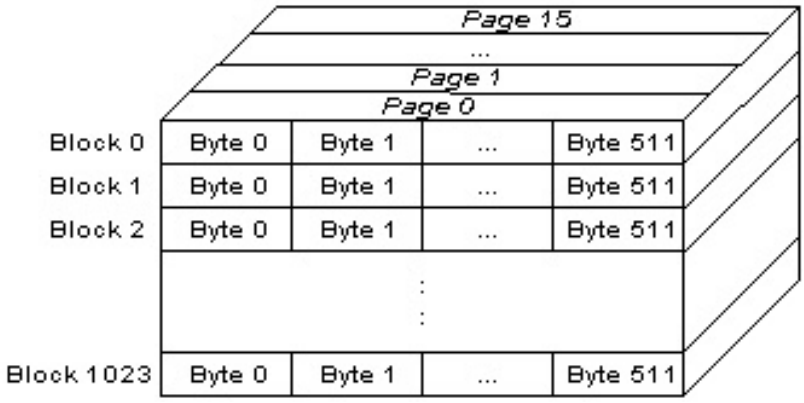

Figure 2.6: NAND Flash Map ([7], NAND Flash Map)

2.1.4 Magnetic Disks

Magnetic storage is also a form of non-volatile memory. It uses different patterns of magnetization in a magnetizable material to store data. Two methods of magnetic recording exist: longitudinal and perpendicular magnetic recording. Perpendicular recording can deliver more than three times the storage density of traditional longitudinal recording. Differences of both recording techniques as shown in figure 2.7.

All hard disks are organized into cylinders, each one containing as many tracks as there are heads stacked vertically. The schematic buildup is shown in figure 2.9.

The tracks are divided into sectors as shown in figure 2.9.

On a modern disk the surface is divided into zones. Zones closer to the center of the disk have fewer sectors per track than zones nearer the periphery. Thus sectors have approximately the same physical length no matter where they are located on the disk, making more efficient use of the disk surface. Internally, the integrated controller addresses the disk by calculating the zone, cylinder, head, and sector. But this is never visible to the user or file system developer.

Data is written to the drive in concentric circles, called tracks, starting from the outer diameter of the bottom platter, disc 0, and the first read/write head, head 0. When one complete circle on one side of the disc, track 0 on head 0, is complete the drive starts writing to the next head on the other side of the disc, track 0 and head 1. When the track is complete

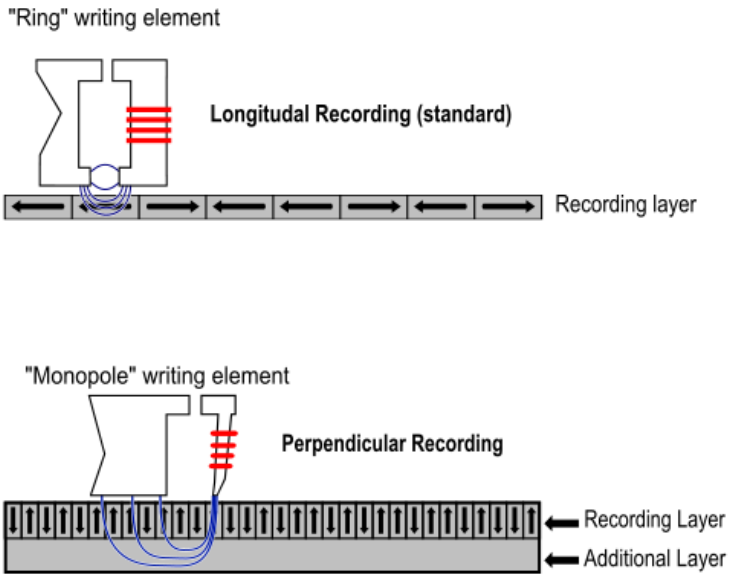

Figure 2.7: Modifying Information on magnetic media ([7], Perpendicuar recordig).

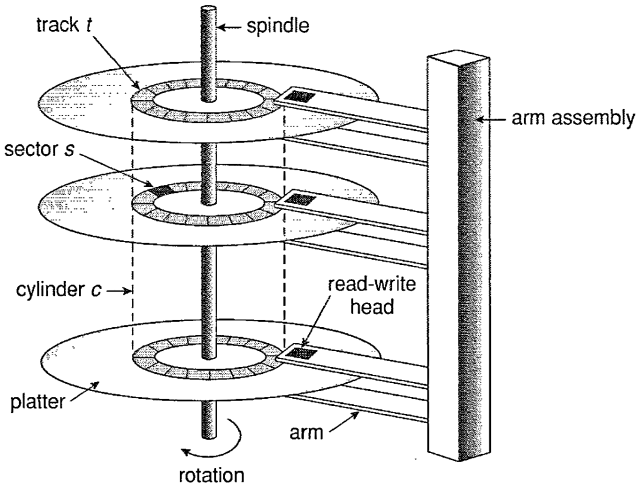

Figure 2.8: HDD Buildup [8]

on head 1 the drive starts writing to the next head, track 0 and head 2, on the second disc. This process continues until the last head on the last side of the final disc has completed the first track. The drive then will start writing the second track, track 1, with head 0 and continues with the same process as it did when writing track 0. This process results in a concentric circles where as writing continues the data moves closer and closer to the inner diameter of the discs. A particular track on all heads, or sides of the discs, is collectively called a cylinder. Thus, data is laid out across the discs sequentially in cylinders starting from the outer diameter of the drive.

The raw disk performance is physically determined by three major factors:

1. The seek time (time to move the arm to the proper cylinder).

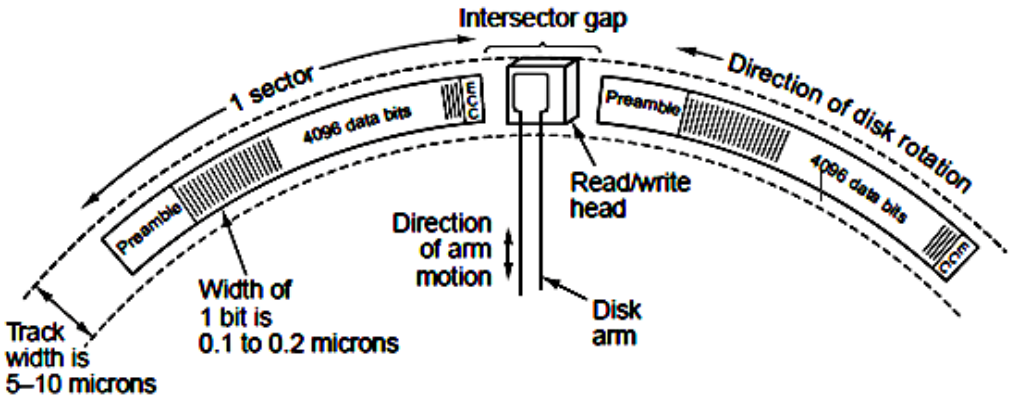

Figure 2.9: HDD Data Track [8]

2. The rotational delay (time for the proper sector to rotate under the head).
3. The actual data transfer time (time the data is transfered from read head to cache)

For most disks, the seek time dominates the other two times, so reducing the mean seek time can improve system performance substantially. Particularly with hard disks, the transfer time for consecutive sectors within a track can be very fast. Thus reading more data than requested and caching it in memory can be very effective in speeding disk access. Disk devices are prone to errors. Some kind of error check, a checksum or a cyclic redundancy check, is always recorded along with the data in each sector on a disk. Even the sector addresses, recorded when the disk is formatted, have check data.

2.2 Information Representation

After discussing how data can be physically stored, a closer look at how data can be structured to represent information is useful, to introduce a file system concept described in the next chapter. In general humans deal with information in form of symbols, computers with data in form of bits. To make them work together requires a mapping between this two distinct systems.

2.2.1 Data Structures

As already mentioned, the smallest base unit of data is called a Bit, physically represented by two distinguishable conditions of the same object or position. A bit of data can logically represent various kinds of information, depending on interpretation of his current status. For Example: `Yes/No, 0/1, True/False, Blue/Green`. To manage more complex information content more Bits and an arrangement are needed. Bits are grouped together in bit patterns to represent all information items of interest. There are 2^n possible bit patterns for a sequence of n bits. The next data unit is called a Byte, which is, by convention, a collection of 8 Bits, and has the power to distinguish 256 information items (symbols). It is often used to represent a standardized character set like ASCII (American Standard Code for Information Interchange) or numbers in e.g. two's complement. By the similar procedure, of grouping

smaller units, a byte filed can be defined by the number of bytes. This structure is now able to represent names, so called strings, with (theoretical) unlimited length and numbers with a greater precision or size. Finally the last data structure hierarchy level, at which every kind of formalized information can be stored, is often called a record. This structure consists of a number of fields and has the advantage to link data about quantity and quality of an information item, by means of represent strings and numbers simultaneously. For example personal data of an employee: first name, last name, salary and working time. To manage this records, or data in general, additional management information, also in form of data, needs to be stored and distinguished. This distinction is made at the semantic level.

2.2.2 Data Semantics

Data semantics deal with the "meaning and the use of data". Data turns to information by interpretation and interpretation depends on the current context. Because the context can change and often do so, it can be seen as an additional data description level above physical storage techniques. A semantic data model define the meaning of data within the context of its interrelationships with other data. Often, only the binary distinction between "Data" and "Meta data" is used. Data, which is also called user data, represents the content – the data itself. User programs know how to interpret this data stream in a very specific manner, and the bit pattern may represent any kind of information like pictures, text documents, music, etc. and also nothing at all as well. Meta (from Greek: $\mu\epsilon\tau\alpha$) means: after, beyond, with, adjacent, self. Meta data describes data or can be explained as data about data. It provides information about a content. For example, an image may include meta data, which describes how large the picture is, the color depth, the image resolution, when the image was created, and any other attributes. Many levels of meta information can exist, and their content or meaning can change in time and space, without affect the original data content they refer to. Meta data can also be used to describe and manage other meta data. With this preliminary considerations about different aspects of data, we can go on with the description of a system to store and manage data.

2.3 File System Concepts

A system describes abstractly a functionally related group of elements and a set of methods for organizing something. An operating system (OS) provides a uniform view of information storage and abstracts from the physical storage by defining a logical storage unit, the file. To handle this objects the OS defines a standard set of operations. The specific physical representation and organization is the job of a file system (FS). Since a file system is normally used as a subsystem of a operating system it has to provide operations on file system objects, or at least, implement a subset of these. Various file systems exist, like pseudo, virtual, read-only or network file systems, each with a specific application and usage restriction. Apart from that, real file systems build a class of systems for permanent storage. These kind of file systems will be, in the following, referred to simply as file systems. File systems turn a vast homogenous array of bytes on a storage medium into a highly organized hierarchical structure more akin to humans categorically oriented thinking. They are expected to be fast and efficient, making good use of the underlying technology to provide rapid responses to

user requests. File systems access block devices through the block driver interface for which the device driver exports a strategy function that is called by the operating system.

To provide data storage and access, a file systems must fulfill two main objectives:

1. Data Management
2. Meta Data Management

The first organize physical data storage and implement functions to store and receive file content. The second is responsible for maintaining the logical name space, what means to perform a logical to physical mapping, and build structures or rebuild them, if they have been modified. This distinction is not always that clear and both have to keep track of free space, block allocation, data consistency and so on. The linking of data and meta data (content and description) is achieved by a file.

2.3.1 File System Object

A file system object, or just abbreviated as file, can be considered as a universal package of information with a name attached to it. Universal in the sense that it is used on many system for various purposes like: store and organizing data, program execution, communication, access restriction, configuration, and many more. Nearly any aspect and feature of an operating system can be associated with a file. Unix like systems follow up the idea "everything is a file", and provide the same access semantic for a device as for a any other file. But that is only partly true. Files are distinct objects, and can be distinguished and described not only by file name, but also by file attributes. Only the storage structure and possible content of this additional information (called meta data) is uniformly defined for every file object. Therefor various file types exist. Regular files the most common case. They act as information container structured as a sequence of bytes and the OS does not interpret their content [9]. A regular file needs data blocks only when it starts to have data.

2.3.2 File Name

A file name is an arbitrary, directory wide unique, character string, with an implementation dependent maximum length. It provides therefore the identification of a file object inside a directory name space. On Unix systems the file name may consist of any characters (supported by the used character set), except the slash character, which is used to separate file name components in a path name (as described later in2.3.6). While specifying file names, the notations "." and ".." are also already in use, to denote the current working directory and its parent directory, respectively. If the current working directory is the root directory, "." and ".." coincide. File names together with the file type attribute (described in the next subsection) build the file system name space structure.

2.3.3 File Attributes

File Attributes (also called regular attributes) define the standard part of files meta data. They have a uniform and general structure for every file system object to provide administrative information for high level file management (at operating system level and above). A

typical command line listing of the most common information about a file on a Unix system is shown in figure 2.10

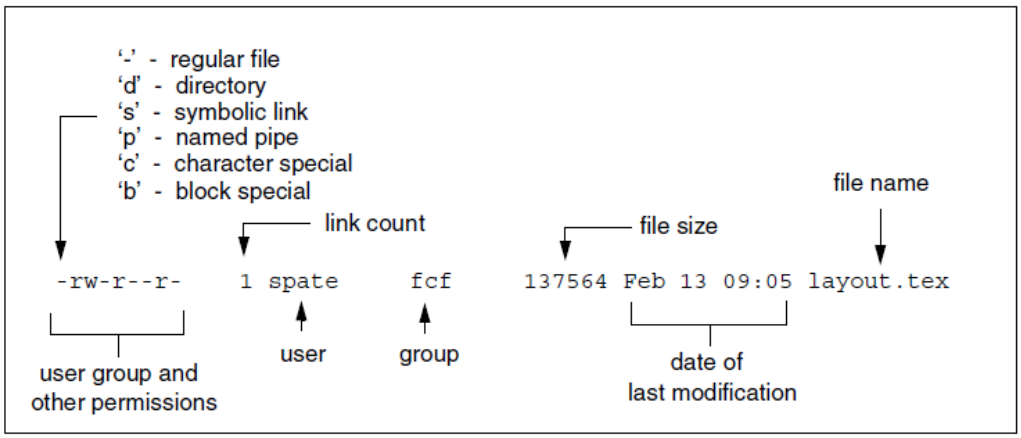

Figure 2.10: File name and attributes shown by typing `ls -l` in a shell [10]

File systems may implement, in principle, any kind of file attributes but need, to make them available, a correspondent caller function and structure in the interface, they are accessed through. A method to access not strictly predefined attribute types is, to use the extended file attributes feature. Extended attributes (xattrs) can be set in different name spaces. Linux uses `security, system, trusted,` and `user` [1]. This is a flexible extension to the restricted, but regular attributes, and many systems use them for different reasons but also in different ways[2]. With the `user` attribute space the distinction between data and meta data begins to blur. Perhaps in some circumstances it would be of interest to store weather information along with each file, and sort them by sunshine, but the coverage of file attributes is always a compromise between space consumption and the amount of useful information about a file.

This selection contains only a standard set of attributes but enables nearly all common meta data operations.

Mode

The file mode stores file types and permissions compactly in a bitmap.
File type describes the kind of file usage and how the data stream is handled. Common file types in Unix are:

- Regular file - data container
- Directory - file index
- Symbolic link - reference to another file
- Socket - interprocess communication
- Device file - character or block device

[1] `http://www.freedesktop.org/wiki/CommonExtendedAttributes`
[2] `http://en.wikipedia.org/wiki/Extended_file_attributes`

- and some more ...

File permissions describe access and modification rights of a file. They have names like:

- r = read permission
- w = write and modify permission
- x = execute permission

Permissions take a different meaning for directories

- r = read - determines if a user can view (ls) the directory's contents
- w = write - determines if a user can create new files or delete file in the directory
- x = execute - determines if the user can enter (cd) the directory.

Various combinations of following file permissions are possible:

- read, write, execute permission for user (owner)
- read, write, execute permission for group
- read, write, execute permission for other

Ownership

Every user on a Unix system has a unique user name, and is a member of at least one group. To provide permissions every directory and file on the system has a field to store this user and group ID. This ID is represented by an integer number, and their size determines the maximum number of system users.

Time Stamps

Each file has fields where main historical data manipulation event dates are saved. This dates are important for differentiation of obviously same files or any other applications which operates on this informations (like compilers or synchronization programs). Beside this they also provide convenience for users.

Three different timestamps are used:

- Creation time - when was the file created.
- Access time - last access for read.
- Modification time - last access for write, even if no writes were actually performed.

Size

Physical on-disk size occupied by file data. Some file systems stores meta data, like file names as data, or preallocate storage space.

2.3.4 Directory

Directories (also called folders) provide a logical file organization structure. The content of a directory are files (and so directories too). It can also be described as a file index file, because the distinction between directories and other file system objects, is in the first instance, made by interpretation of the file type attribute. This model gives rise to a hierarchical name space.

2.3.5 Hierarchical Name Space

From the user's point of view, files are organized in a tree-structured name space. The hierarchical directory system concept linked directory name spaces together in a way files are not only identified by names, but also by position. The logical position is defined by the parent directory position relative to the root directory (measured in names). This extends the name space for finite file names, and provides a better possibility of data organization and administration.

The structure looks like a tree with a root directory, sub directories as inner nodes and directories or files as its leaves.

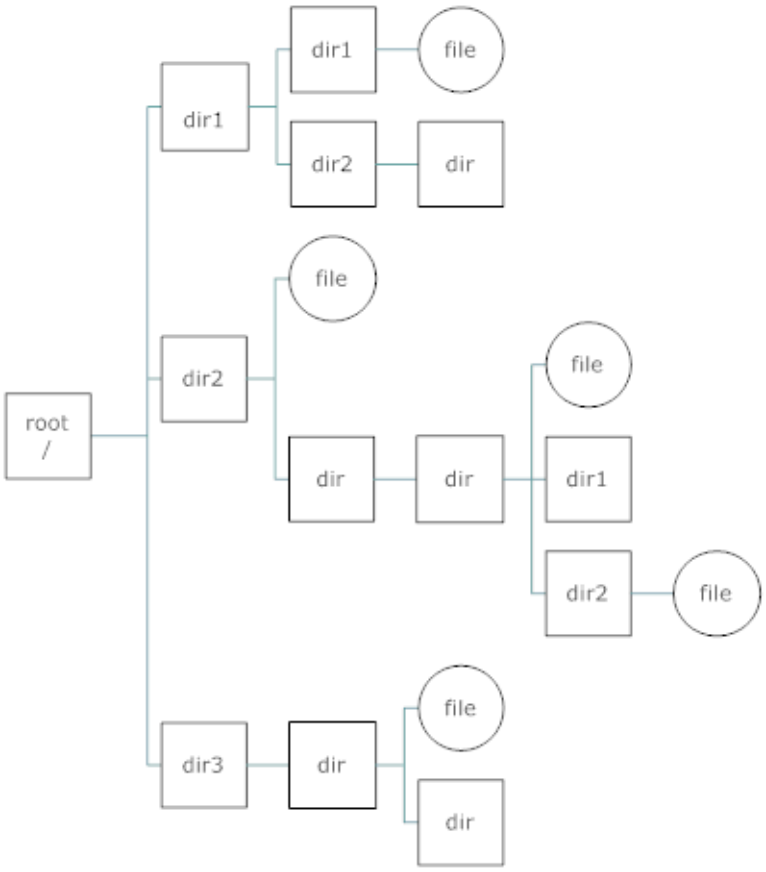

Figure 2.11: Hierarchical Name Space

2.3.6 Path - Name and Lookup

A path name is a sequence of names separated by a separator character (in Unix Systems a slash "/"). This well-ordered string describes the path to, and the position of a particular file system object in the hierarchical file system name space. The path name itself consist logically of two parts a path and a name, also known as dir name and base name. For a path name like: `/fbi/home/scully/xfile` the dir name would be `/fbi/home/scully`, the base name `xfile` If the path name starts with a separator character, it is called absolute otherwise relative path name. The Unix file name convention forbids the use of the slash character, because otherwise the path lookup algorithm will misinterpret the path string. To find a file position in this system and so the file itself, the path name must be processed name component by name component, comparing current file name with file names indexed in the directory with the last preceding component name. If the path name is absolute the lookup starts in the root directory if is relative, at the current directory position.

2.4 File System Environment

Unix-like operating systems are built from a collection of libraries, applications and developer tools, plus a kernel to allocate resources and talk to the hardware [3]. Figure 2.12 illustrates this composition.

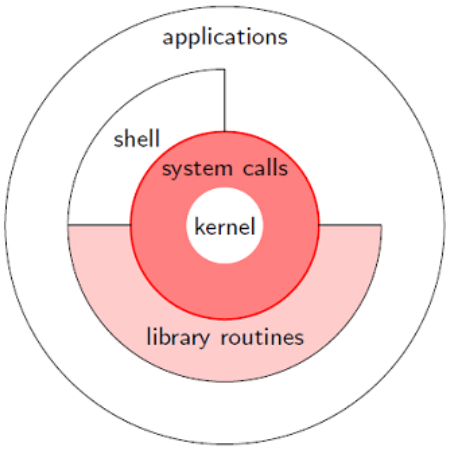

Figure 2.12: Abstract OS Layers

The operating system communicates natively with file systems through a well defined set of functions and structures. Every OS can integrate and manage a file systems name space in various ways, and access the stored data through it. Therefore standards for general-purpose concepts and interfaces had been introduced.

[3] `http://www.gnu.org`

2.4.1 POSIX API and System Calls

The Portable Operating System Interface for Unix (POSIX) defines standard application programming interface (API), along with shell and utilities interfaces for software compatible with variants of the Unix operating system. The difference between an API and a system call is, that the former is a function definition that specifies how to obtain a given service, while the latter is an explicit request to the kernel made via a software interrupt. Unix systems include several libraries of functions that provide APIs to programmers. Some of the APIs defined by the standard C library (libc) refer to wrapper routines (routines whose only purpose is to issue a system call). Usually, each system call has a corresponding wrapper routine, which defines the API that application programs should employ. POSIX.1 Core Services define standard file and directory operations which are itself standard libraries and system calls and provide complete control over the creation and maintenance of files and directories.

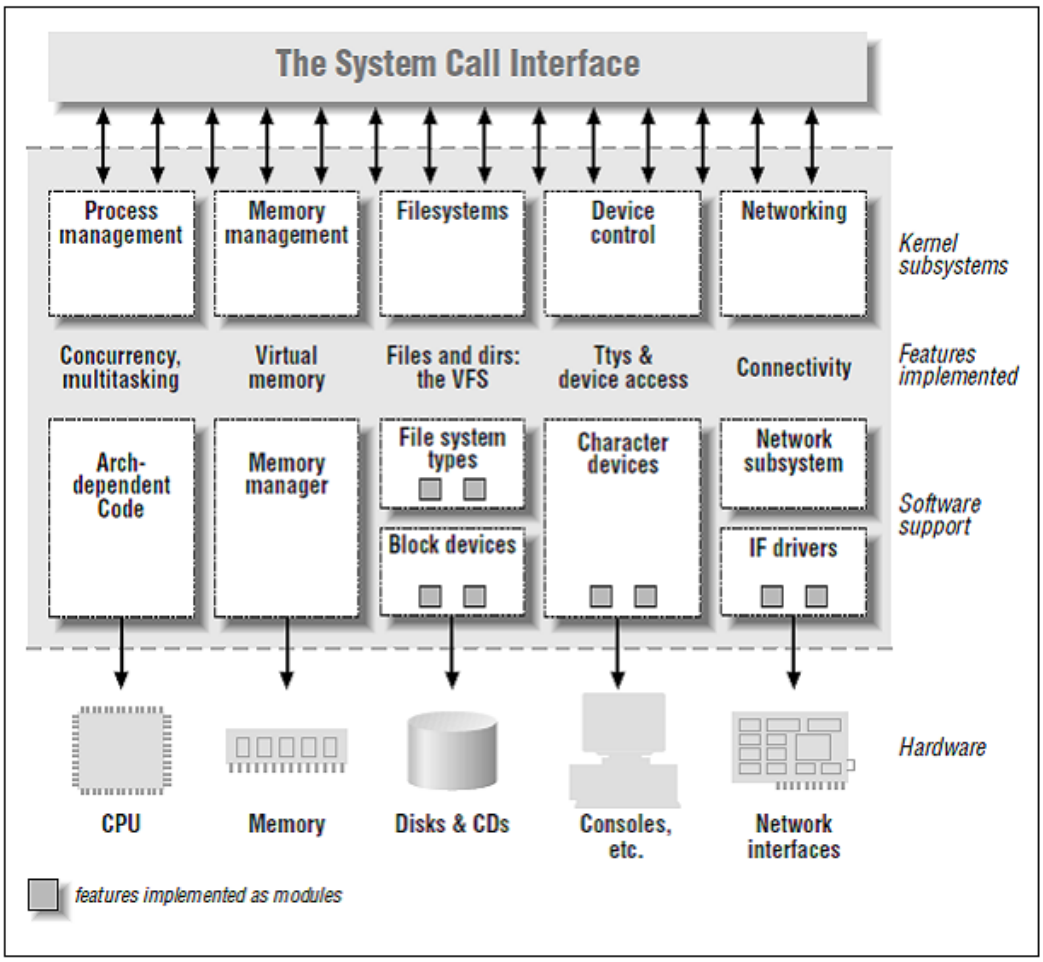

Figure 2.13: The System Call Interface [11]

2.4.2 Virtual File System Switch

A virtual file system (VFS) or virtual file system switch is an abstraction layer on top of a more concrete file system or layer. It is used to manage logically various file system types mounted to the OS, as shown in 2.14 .

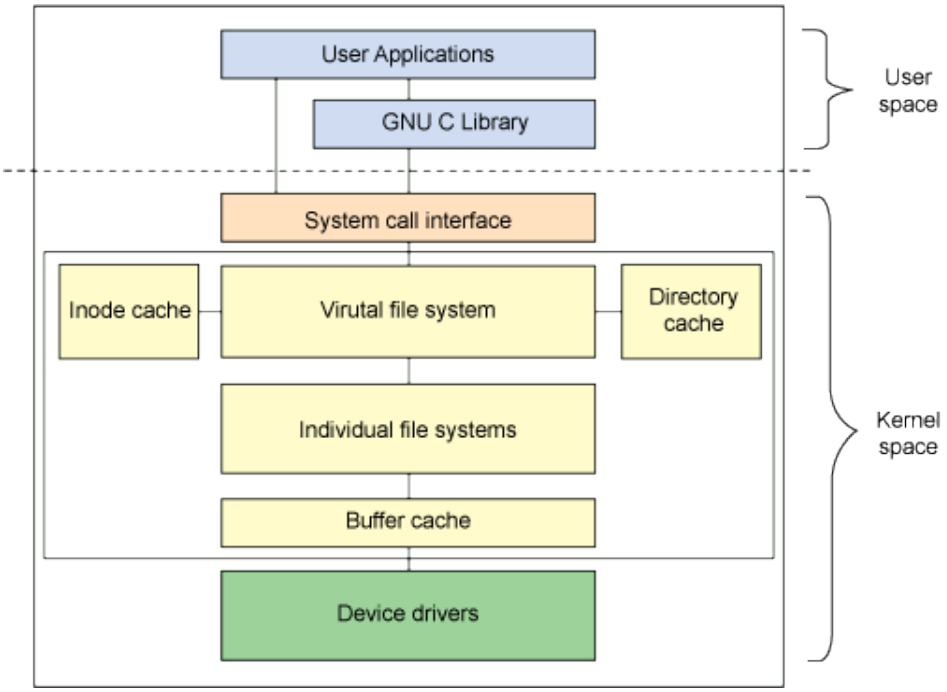

Figure 2.14: Architectural view of the Linux file system components [12]

The VFS accomplishes this abstraction by providing a common file model, which is the basis for all file systems in Linux. Via function pointers and various object-oriented practices, the common file model provides a framework to which file systems in the Linux kernel must adhere. This allows the VFS to generically make requests of the file system, but contains for this reason only a minimal subset of possible capabilities of a file management system.

The framework provides hooks to support reading, creating links, synchronizing, and so on. Each file system then registers functions to handle the operations of which it is capable and the VFS-API defines. The VFS knows about file system types supported in the kernel. It uses a table defined during the kernel configuration. Each entry in this table describes a file system type. It contains the name of the file system type and a pointer on a function called during the mount operation. When a file system is to be mounted, the appropriate mount function is called. This function is responsible for reading the super block from the disk, initializing its internal variables, and returning a mounted file system descriptor to the VFS. After the file system is mounted, the VFS functions can use this descriptor to access the physical file system routines. Two other types of descriptors are used by the VFS: an inode descriptor and an open file descriptor. Each descriptor contains informations related to files in use and a set of operations provided by the physical file system code. While the `inode` descriptor contains pointers to functions that can be used to act on any file (e.g. `create`, `unlink`), the file descriptors contains pointer to functions which can only act on open files (e.g. read, write).

2.4.3 File System in User Space

FUSE is a further abstract file systems layer. It provides a framework for building user space file system servers, i.e. implement file systems with user space code. The basic idea is to integrate information behind the file system name space. From a programmer's perspective FUSE provides a library and defines a standard and a low-level interface to use it. FUSE implements a library which manages communications with the kernel module. A FUSE-supported system integrates kernel and user level components. Attach an in-kernel file system (component/module) to the kernel's virtual file system layer. These functions have names like `open()`, `read()`, `write()`, `rename()`, `symlink()`, etc. The FUSE kernel module and the FUSE library communicate via a special file descriptor which is obtained by opening `/dev/fuse`. This file can be opened multiple times, and the obtained file descriptor is passed to the mount system call, to match up the descriptor with the mounted file system.

Short description of how it works:

- accepts file system requests from the FUSE device, prepares incoming requests for delivery to user space
- translates them into a set of function calls which look similar (but not identical) to the kernel's VFS interface
- sends the request to user space
- waits for a response
- interprets the answers
- feeds the results back to the caller in the kernel

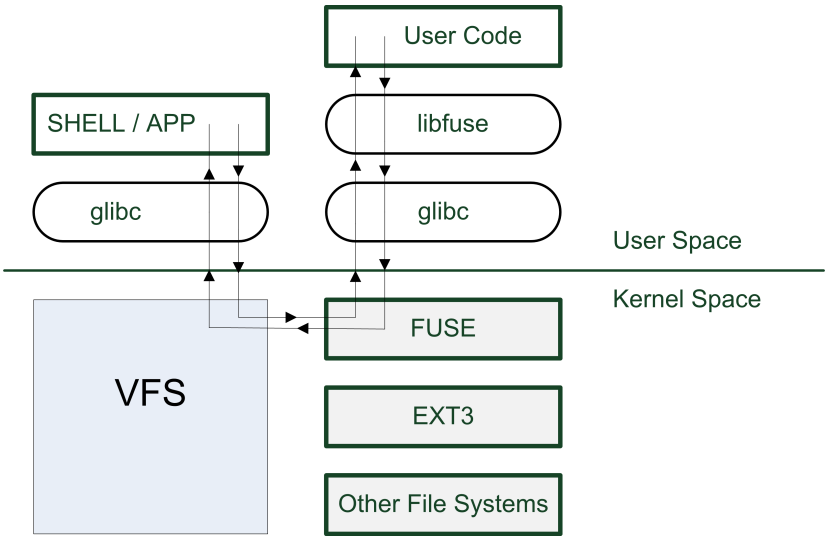

Figure 2.15: FUSE structure

The sections of this chapter deal with complex issues and some books are necessary to explain them in detail. So the goal was to put the most focal issues together and point out the key

details for practical application and basic understanding of soft- and hardware components of the I/O subsystem. It should becomes clear how information, in form of digital data, is stored and retrieved, and what the particular tasks of a file system are. Furthermore, a possible embedding of a file system in an operation system is described.

Chapter 3

Design and Implementation

This chapter describes the design and implementation aspects of the implemented file system (metafs), the media usage, on-disk layout, data structures and algorithms.

3.1 General Aspects

Design and implementation decide about the performance of a system, which can be measured as the time it will take to complete an operation (e.g. `create/remove` or `lookup/stat` a file). System design decisions, determine general data flow and structure. They can not be changed easily afterwards, so they have to be considered carefully for all possible application cases. Implementation on the other hand, can be improved in details over time. Some details can have a negative performance impact for frequently called functions (e.g. `loopup` or `readdir`) and in application scenarios with lots of files, small delays may accumulate to a significant latency. To put it pointedly: "implementation wins battles, design wins wars".

In this case, the design consideration refers to a persistent file system, dedicated to store meta data – therefore named "metafs". To implement this, we need first to specify basic system properties and provided features:

- file system environment and interface
- coverage of file attributes and operations
- storage media usage
- data flow and mapping

To keep the implementation effort low, the file system code will reside in the user-space and use the file system interface provided by FUSE (see section2.4.3), to communicate via POSIX compliant meta data operations. This has obviously the disadvantage of lower performance caused by numerous context switches, but should be acceptable to verify the basic concept. Portability between 32/64 Bit systems, which is critical for address operations on persistent data structures, is accomplished by using fixed size variables. The coverage of file attributes complies with the regular file attributes described in subsection 2.3.3, and the file name length is restricted to 255 characters. A possibility to log meta data transactions should be given to avoid file system corruption, which can occur by aspects of a removable flash storage device.

File system performance depends on the exploitation of storage media specific performance characteristics, mainly data throughput and access time. As described in the section 2.1 HDDs perform well in sequential, and flash devices in random I/O operations. Well, not the device alone decides about I/O operation patterns, but the data layout, in combination with application, defines them for the most part. At the file system level, three different types of data can be distinguished:

- meta data
- data
- journal data

The file system should therefore have the ability to use (transparently) three storage locations (files/devices). The data storage will be left unimplemented but prepared to be added later.

For the meta data part of a file system, the I/O requests are often random and have small size, e.g during a lookup. File attributes take about 50 Byte and a filename is seldom over 255 Bytes in size, mostly smaller. In this case the access time is the deciding factor and indicates the use of a flash storage device, but only for a read operation. During creation or modification of files, write operations on meta data are performed, which are not emphatic fast, due to the need to erase an entire block before writing to it. So, if the meta data throughput should be reasonable in average, the write operation has to be accelerated. The assumption made here is that writing blocks of a device specific block size (called erasure block size) is the optimal strategy. Different devices may have different erasure block sizes, so the block size needs to be variable to support a wide range of devices. To pass this physical storage property to the logical file system name space level and benefit from it, all entries of a directory are stored in the same block, or if the amount of them exceeds the block capacity, an additional extend block will be allocated and so automatically associated with the directory.

The data (content) and the journal part are mostly bigger in size and rather sequentially requested.

In overview, the decision of what to store and where tends to be as follows:

- meta data on flash storage (random)
- data on hard disk (most sequential, seldom random)
- journal on hard disk (strictly sequential)

This is only a proposal for a storage division, which is provided by metafs. It is also possible to store everything on one device, by partitioning the available address space into volumes or using files (stored on other file systems). Besides this, it should also be possible to store just meta data on a single device or file, which is the current implementation focus. The concrete media usage decision is made during file system creation.

After discussing basic aspects, we need to define persistent data structures which build the so called on-disk file system.

3.2 Persistent Data Structures

The linear byte address space of the media is logically divided into blocks, (as mentioned above) ideally with the controller specific "erasure block" size. File system management structures (super block and bitmap) reside at the beginning of the address space followed by blocks containing meta data of file system objects. The block layout schema 3.1 gives an overview.

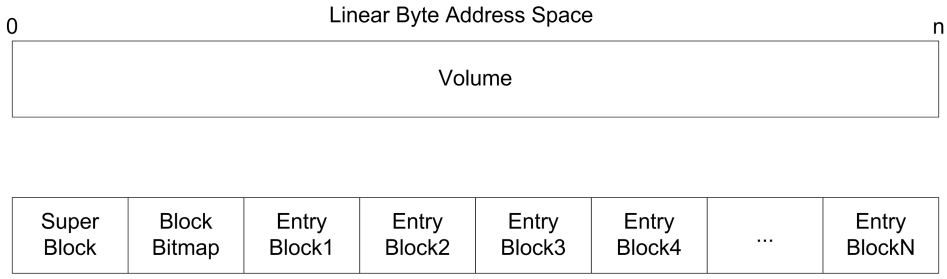

Figure 3.1: Block layout.

3.2.1 Super Block

This is the entry point to the file system when the volume is mounted (see subsection 2.4.2) The information within the super block is used to maintain the file system as a whole. It contains the description and shape of metafs, like position and size of main data structures, selected volume and block size and the status in which the file system has been disconnected the last time. Whenever the file system has not been proper unmounted (e.g. system crash, device removal), some meta data inconsistencies, or structure corruptions may occur.

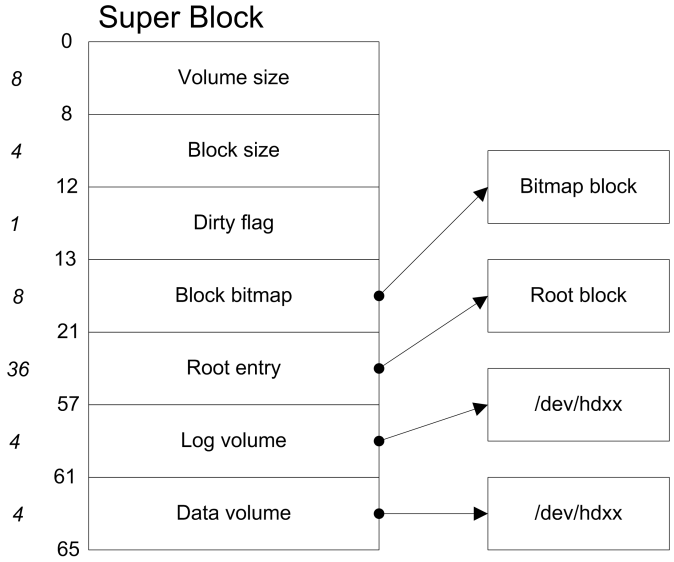

Figure 3.2: Super block structure.

3.2.2 Block Bitmap

A bitmap stores compactly individual logical (boolean) values represented by bits. Block allocation is performed by setting a bit in the bitmap at the according position to the block number. Setting a bit to 1 means a block is used and 0 indicates a free block. The bitmap must have, at least, the same number of bits as the volume has blocks, to manage the block address space. While the volume size and block size are variable parameters, in the file system creation process, the bitmap varies in size, but occupies at least on block.

3.2.3 Block

The block is, in this case, an abstraction of the file system, not of the physical media on which the file system resides, but for a flash media they should correspond as described in section 2.1. An entry block is a physical directory representation structure. The block number multiplied with the block size results in the offset to which the file pointer seeks in a `read/write` operation. Every directory allocates at least one block. It is a container for contents of directories. Directories can contain subdirectories and other file entries. Every block has a header, which contains his current fill position, measured in bytes, to keep track of how much of the block space is occupied by entries, and a pointer to the next extent block, which is a block number, or 0 if no further block is allocated for this directory.

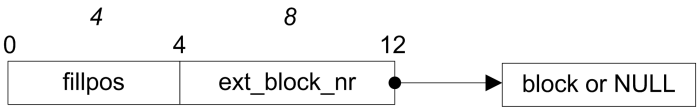

Figure 3.3: Fields of the block header

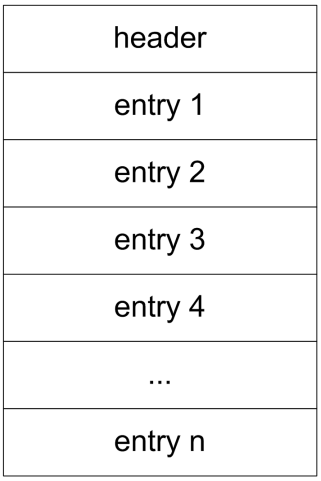

Figure 3.4: Block with entries

3.2.4 Entry

An entry is a file system object representation structure. It stores file attributes (as described in subsection 2.3.3) and file names. The file name length is variable, so the size of each entry is variable too. To catch this variability an additional variable which stores the name length is used. Another possibility, which is a little bit more complex but also more flexible, would be to test the string for an "end of string character" - but this then needs to be saved as well, and the string parsing would not speed up the operation. Each entry has a pointer, which points (file type dependent) to an additional address. This address has a 64 Bit range and represents for a directory a block address, where entries of this directory are stored. For any other file type this pointer can be used as a multi purpose link to any kind of data, and is left for this reasons undefined. Figure 3.5 shows the fields of an entry in detail.

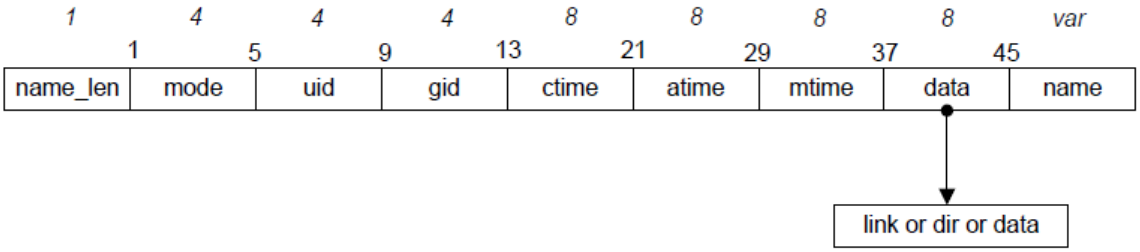

Figure 3.5: Fields of an entry.

3.3 Logical-to-Physical Address Mapping

From the user view a path string is the logical address of a file in the hierarchical file system name space. This string consists, as described previously, of path components (names) separated by "/". This logical address can be translated to a physical address by parsing the path-name and loading each block of each directory path component into the cache. It can be simply described as a directory-to-block mapping. Only the last component of the path could be a non directory file object, then the decision on the next step depends on the file type (as described in 2.3.3). For a regular file this could be a read or write data request or for a link a new path-name lookup. When the last path component is a directory, the content is loaded into the cache. For a path name like: "/usr/local/bin/file" the lookup on persistent storage looks as illustrated in figure 3.6.

When the directory is already cached, the storage device is not involved and the lookup runs only in the main memory. The cache has a different structure to hold the meta data and is optimized for faster access and operations, as described in the next section.

3.4 Cache

All data has to be present in main memory to be visible to the CPU. When the data is kept there longer than really needed we refer to it as a cache. Cache is a very important I/O

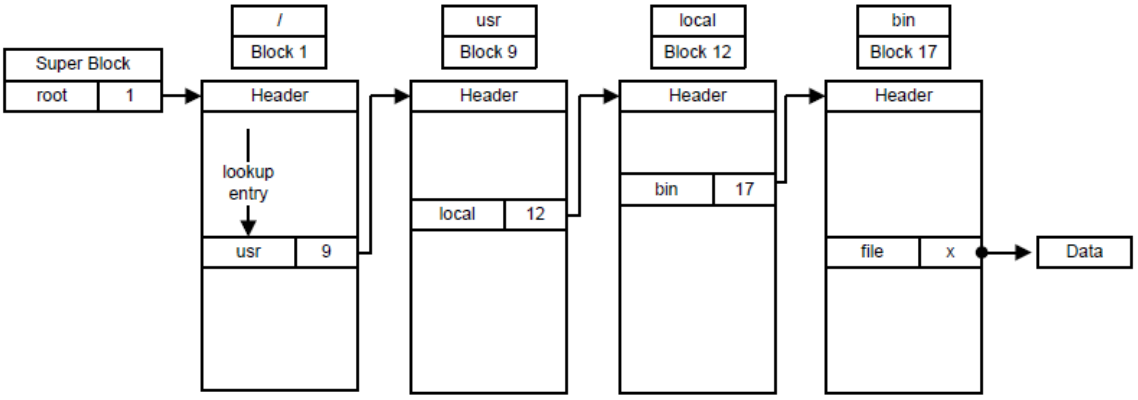

Figure 3.6: Simplified path-lookup (without extended blocks).

performance feature. It can speedup a repeated request nearly by the factor of access time or throughput difference between storage media types. For slow write operations on flash a write-back-cache type is advisable. It delays the write-back-block operation to a specific point in time when a write-back-thread starts to process the list with modified blocks. The right write-back-time is a compromise between data safety and performance aspects. The longer data is kept unwritten (from main memory to storage device) the greater is the risk of data loss. But on the other side, when a block is modified twice and not written back in the meantime, this would safe one write for example. For file systems with high meta data modification rates, this is a big performance issue, because during the write-back-process the content needs to be locked, to avoid inconstancy. Without locking, it may happen that only partially modified entries are stored. A reasonable default time for multi purpose application would be for example a time range of 5 to 10 seconds. Not to fail to mention for the sake of completeness is that the durability of a flash device will also benefit from fewer write operations, because they outwear the flash cells and reduce so the lifetime of a block.

Besides theoretical aspects there are two points in time when dirty blocks must be written back immediately: when the file system is unmounted, and when the cache runs full (reached defined maximum size). The second point leads to the question of how much memory should be designated for caching, and which entries should be unloaded first and when. This question has, like the previous, not a single answer and can be discussed from many points. Possible strategies could be: last recently used (LRU), first in first out (FIFO), the biggest or deepest directory first, and so on. The easiest way would surely be to unload the whole cache at once and let it grow again from the root directory. In some cases this is not as bad as it looks at the first glance, and has the advantage of low implementation and processing overhead. It takes only one giant lock, write back and unload all - no comparisons or elaborate bookkeeping.

The cache as a temporary file system needs, like the persistent one, a structure layout in the main memory. This layout must map the data layout on disk on the one hand and provide the name space and operations on the other hand – in best case efficiently in terms of space and time complexity.

A file system provides many different operations on a variable number of elements, what makes it difficult to find the best suitable data structure for all. Lists are better for adding

and deleting elements, trees are advantageous for searching big data sets. In the first place we need a structure which keeps track of the loaded directories. Assumed more lookups and fewer adds or removes run on it, a tree structure is fitting – a binary tree with directory block numbers as keys and pointers to the directory content. The directory numbers, which correspond to the first block occupied by a directory, tend to be more and more random during file system usage. A directory structure has to manage entries and keep their association to blocks. Since a directory can have a variable number of blocks allocated, and a block can store a variable number of entries we need two lists, one for blocks and one for the entries of each block. Whenever an entry inside a directory is modified (created or removed as well) only the corresponding block, in which it resides, is marked for `write-back`. To speedup `lookups` and provide native directory listing in alphabetic order, a directory wide tree with entry names as keys and pointers to the attributes is used. The schematic structure of an into memory loaded directory is shown in figure 3.7

3.5 Journal

Journaling is a compromise between reliability and performance which stores only meta data changes in a log. Due to the fact metafs stores only meta data, logging is equivalent to journaling. Since all meta data I/O is block oriented, the recording of changes is also carried out block wise. The whole block content is (first) written to the log and (then) written to the storage device. Broadly spoken, a copy of the modified content is made. After the write operation (to the meta data storage device) is completed successfully, the data in the log will be deleted and the recording starts from the beginning. The only overhead that has to be added, is the block number, which maps the data part in the log space, to the position on the storage media. This is needed to implement a recovery function that writes back outstanding blocks after unclean unmounts. File system status is indicated in the super block, and is set from clean to dirty at mount time. As the write-back super block operation is the last one performed during an unmount procedure, the resetting back to clean assures correct file system state indication.

3.6 File System Operations

The description of file system operations in this section can be subdivided into four major categories: high-level, path-lookup, low-level and I/O-operations.

3.6.1 High-Level

At the High-Level we have functions, which perform operations invoked directly by the user or application decisions. They are defined in all file system layers above. These functions build the standard interface, through which the file system is used. Shell commands/programs, like for example `ls, cp, rm, mv, find`, provide a convenient way to manage file system objects, by calling these functions, according to desired operations.

These are the standard file system meta data operations known as:

- `create, unlink` - create/delate regular file

- `mkdir, rmdir` - create/delate directory
- `readdir` - read directory content
- `rename` - change file name and position in the name space
- `getattr` - get file attributes
- `chmod` - set file permissions
- `chown` - set owner and/or group
- `utimes` - set file timestamp
- `statfs` - get file system parameters
- `destroy, release` - unmount file system

3.6.2 Path-Lookup

Functions, which provide path-lookup, act as a mediator between functions at High- and Low-Level. As the path name parameter is passed along with (almost) each of the above listed functions, it has to be parsed and the data for each disassembled component has to be located in the cache structure or, if not found there, loaded into it. Each file system object resides in a directory (called the parent directory - except the root directory), so the lookup for each path component consist mainly of two functions:

- `lookupDirectory`
- `lookupEntry`

A recursively called `ls` shell command (`ls -Rl`) at the top directory would, for example, load all file system meta data into the cache (assumed the cache size can hold it).

3.6.3 Low-Level

At the Low-Level are functions, which perform all needed operations automatically in the background - not visible or directly accessibly to the user.

Generalized they are responsible for the following operations:

- File system (un)loading - during mount and unmount
- Directory loading - during lookups
- Bitmap manipulation - block (de)allocation
- Caching - insert or remove nodes in cache structure
- Locking - secures modification consistency
- Syncing - starts write operations to persistent meta data storage
- Journaling - log meta data transactions for recovery

3.6.4 I/O-Operations

Last but not least we have functions which perform concrete data input and output operations from and to storage devices. All read and write operations are buffered, which means, before I/O can take place, an appropriate main memory space amount needs to be allocated, to accommodate the binary data. The procedure of reading or writing a block at once reduces the amount of system calls (software interrupts due to context switching from user to kernel mode) and the penalty of device access operations (hardware interrupts and access time). Reading and writing a whole block, can also be wasteful, especially when the block size is big and the block space is barely used (by entries). Imagine a block size of 500 KB and a block occupation of 500 Byte, which is not an unusual situation and would cause, in this case, a data overhead of 99.9 %. For this reason, we can break down the read operation in two parts. First we read the block header, which contains the block fill position, and allocate, according to this, the buffer space for block entries. Then we read all entries into the buffer. This has obviously the disadvantage of double access times, and would surely be not advisable for a hard disk device, but since flash devices offers linear access times (lower than 1 ms) and (connected to USB2.1) low bandwidth, this strategy seems to be faster. For the write operation, the situation is the same but needs a different consideration. Flash devices need to erase a whole block before writing to it. If only a small amount of data is written (smaller then the erasure block size), and old data in a block should be kept (the device does not know what data is only garbage), it has to be copied to a temporary place and then written back with the new data (this is the assumption made in section 3.1). In this case, writing a whole block seems to be more efficient, to avoid copy on write. To test the performance impact the option to write back just the current block load, we implement this option as well.

To give an overview of how metafs works, some simplified algorithms are described in the following:

3.6.5 Algorithm for Creating a File

1. lookup parent directory
2. find free block in the parent directory block space
3. create a file entry in this block
4. increase the fill position of the entry block
5. append block to dirty block list

3.6.6 Algorithm for Creating a Directory

1. lookup parent directory
2. get free block from bitmap - allocate block
3. create a file entry in the parent directory block space
4. insert allocated block number in the data field
5. increase the fill position of parent directory block

6. append new and parent block dirty block list

3.6.7 Algorithm for Removing a File

1. lookup parent directory
2. lookup file entry
3. remove file entry in parent directory block
4. decrease the fill position of parent directory block
5. append block to dirty block list

3.6.8 Algorithm for Removing a Directory

1. lookup parent directory
2. lookup directory entry
3. set block free in bitmap
4. remove file entry in parent directory block
5. decrease the fill position of parent directory block
6. append parent block to dirty block list

3.7 File System Creation

Before a file system can be used for the first time, it has to be created. This requires an external program, which writes file system specific management structures (super block and bitmap) to the meta data storage device. The program, named `create_metafs`, expects at least 3 parameters:

1. meta data space
2. volume size
3. block size

These parameters define the location and shape of metafs. The definition of the location for the log is optional. According to them, the super block entries and the bitmap are calculated and initialized. An optional parameter to format the meta data storage device (overwrite all blocks with zeros) and check for errors is also provided. This can also be used to benchmark block I/O of the device.

3.8 Mount Procedure

When the file system is mounted, three automatic I/O operations are performed to make the file system ready to use.

1. read Super Block - get file system parameter
2. set file system status flag - from clean to dirty
3. read Bitmap - load address space management
4. read Root Directory content - enable lookup and file operations

The last step could also be executed when the first lookup is performed, but assuming the file system is mounted for using and the root directory should not have too many entries, this procedure is used and the data flow is illustrated in figurure 3.8

The unmount procedure is analogous but inverted. First all loaded directories get unloaded (modified blocks, including bitmap, are written back to storage), the file system status flag is set to clean and written back with the super block.

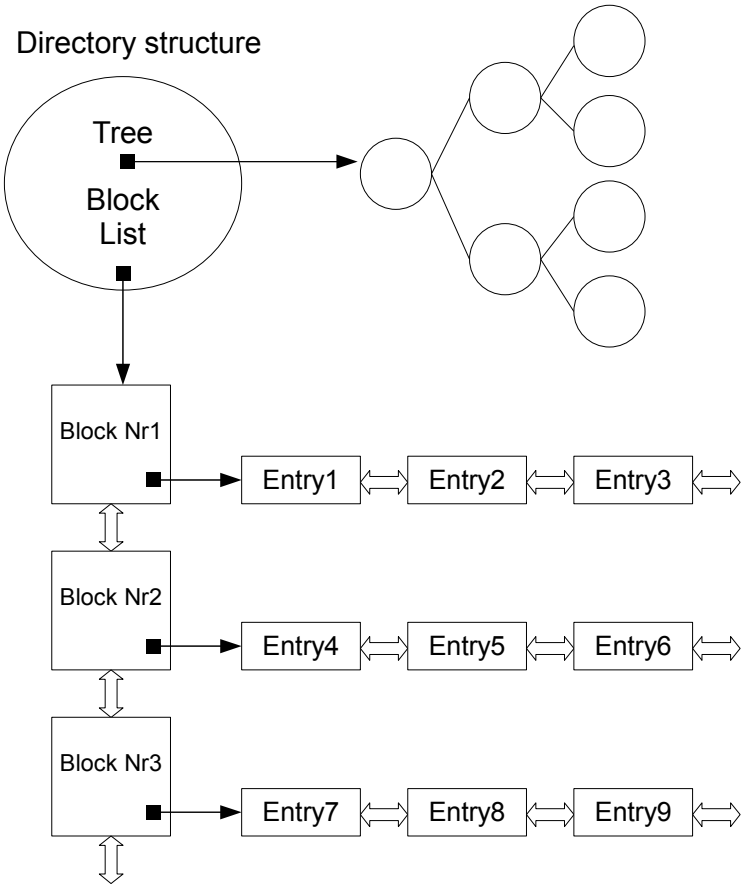

Figure 3.7: In-Memory directory structure.

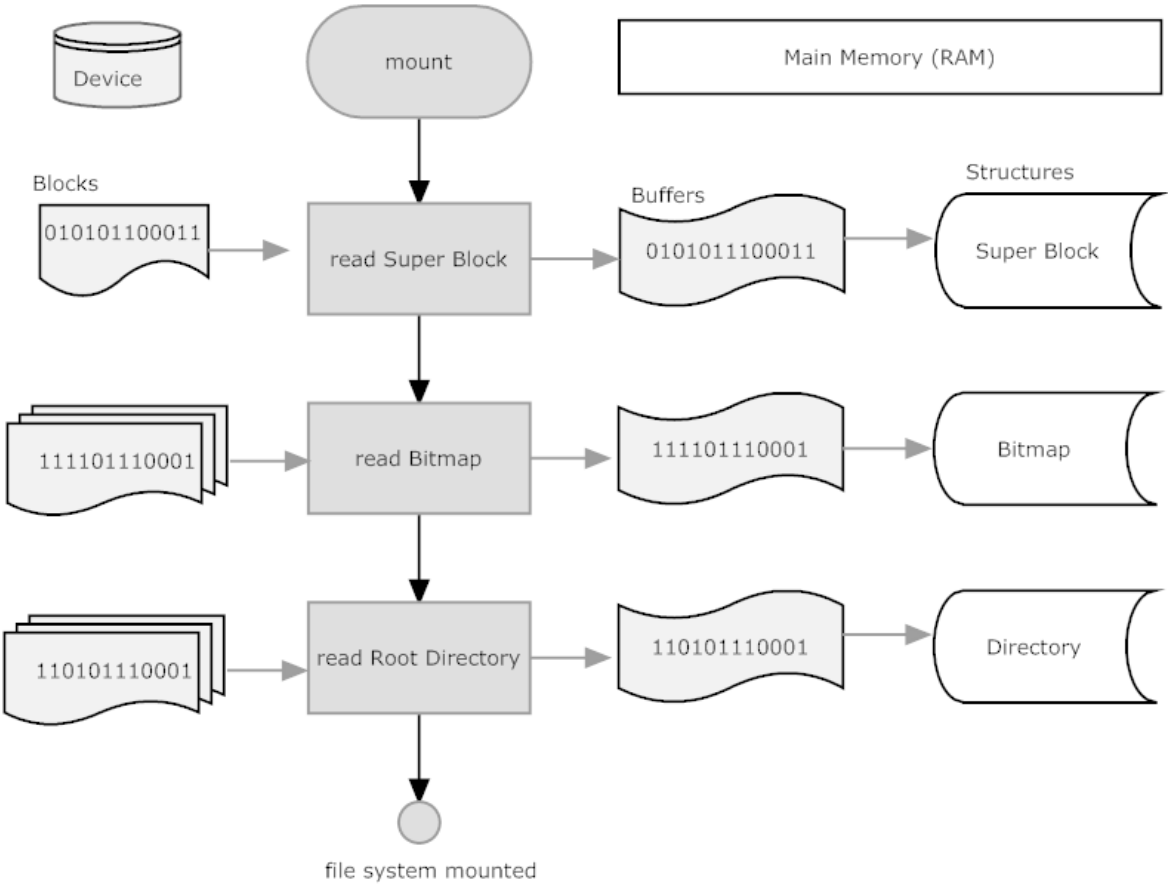

Figure 3.8: Mount procedure

Chapter 4

Evaluation

This chapter describes the evaluation for the implemented file system. After discussing some general file system performance evaluation aspects, we describe used benchmark procedures and environment, followed by a study and comparison of the results.

4.1 General Evaluation Aspects

File system performance depends on various factors such as:

- operations e.g. create, remove
- access patterns e.g. random or seqential
- selected file system create/mount parameters
- used name space structure
- active caches and buffers
- and some more

To decide, if the measured times for a particular operation are high or low, they have to be compared with some other file systems. The comparison is only useful, if the test environment for all tested systems, is exactly the same. To reduce measurement errors, each test has to be run several times, and calculated as the average of the results. All this circumstances, and a the amount of benchmark results makes performance analysis time consuming and often difficult to summarize. The correct interpretation of measured differences requires some knowledge about used file system concepts, and data layouts. Configuration of the test environment (like used OS and device models), are also factors, which influence the results.

4.2 Benchmark Environment

All tests were made using Fedora-12 Linux Distribution with kernel 2.6.32. Testmachine was an Intel Core 2 Duo (T7200) at 2000 MHz with 2GB RAM at 667 MHz. Testmedia was a

16 GB flash storage device (SanDisk - micro cruzer SDCZ6-016G)[1] and, for comparison, a 160 GB HDD (Samsung - SP1614C)[2] with 8 MB internal buffer and 7200 rpm. Both devices were connected via an USB-2.0 bus.

4.3 Benchmarks and Results

To measure the (meta data) file system performance, we selected some basic and common shell commands, used for file management on Unix like operating systems. These commands execute standard programs for file manipulation (included in the GNU/Coreutils package[3]), which provide, via parameter selection, a convenient way to perform complex operations on file system objects. They navigate and manipulate the name space, by calling appropriate file system functions listed in subsection 3.6.1.

The meta data content from the linux-kernel-2.6.32.2 source archive, was used as test data and template for the name space structure. To extract the meta data, it was copied to metafs, which act as data content filter and stores only meta data informations (directories and empty files). This version of the linux-kernel sources contains 30468 entries, which represent a (more or less common) hierarchical name space with 28590 files in 1878 folders, of various name lengths and nesting. To avoid temporary effects from other file systems or devices, the test data was placed on `tmpfs` in RAM and copied from there, to the file system on the tested device. The call to mount metafs can be described generally as follows:

```
fuse_metafs <optional FUSE options> <mount point> <device>
```

The concrete example is:

```
fuse_metafs /media/flash/metafs /dev/sdb
```

there metafs is mounted under /media/flash/metafs and meta data is stored on device referred to as /dev/sdb.

To create, list and remove file system objects, the shell commands (`cp, ls, rm`) were used as follows:

1. Create files and directories - copy recursively from `tmpfs` to FS `fs_y` on device `dev_x`
   `cp -r /media/tmpfs/linux-2.6.32.2 /media/dev_x/fs_y`
2. Lookup and stat all file system objects recursively
   `ls -lR /media/dev_x/fs_y/linux-2.6.32.2 > /dev/null`
3. Remove files and directories recursively
   `rm -rf /media/dev_x/fs_y/linux-2.6.32.2`

Each particular test was run 10 times under same conditions, to improve validity of the results. The tested file systems were created before each test series and empty, that no previous fragmentation took place. To eliminate impacts from caches, all file systems were mounted, before each test case, and no other operation, except the measured one, was executed. The

[1] http://sandisk.de/Products/Item%282660%29-SDCZ6-016G-E11-SanDisk_Cruzer_Micro_16GB_Black.aspx
[2] http://www.samsung.com/me/products/hdd/sata/sp1614c.asp?page=Specifications
[3] http://www.gnu.org/software/coreutils

time for the unmount was counted to the measurement of execution time, because a file system might defer write operation. By unmounting the file system write back is enforced.

Metafs was created with block sizes of 32 KB and 64 KB. The general representation of the call is:

```
create_metafs <device> <volume_size in MB> <block_size in KB>
```

and in concrete (for a 4 GB volume with 32 KB blocks on a device referred to as /dev/sdb):

```
create_metafs /dev/sdb 4000 32
```

Two different block-write-back strategies were measured: `full_block` and `fill_rate`. In the first case only the used block space was written back (free block space was filled with zeros), and in the second the data size corresponded to the block size.

Reference file systems were: ext2[4], ext4[5], vfat[6], ntfs[7] ; all created with default parameters. This are some common used file systems for block devices. The ext2 and ext4 are native Linux file systems and ext2 and vfat do not use a journal.

The evaluation of the test cases led to the following overview of averaged execution times. Tables 2.1 and 2.2 summarize the results.

Times in [s]	File System							
Operation	ext2	ext4	vfat	ntfs	metafs$^{(1)}$	metafs$^{(2)}$	metafs$^{(3)}$	metafs$^{(4)}$
cp	2.1	2.6	3.4	10.3	5.1	5.6	5.7	5.4
ls	5.7	4.7	10.8	22.2	3.0	2.8	3.2	3
rm	3.4	4.9	9	26.7	5.4	5.4	5.3	5

Table 4.1: Meta data performance on HDD.

Times in [s]	File System							
Operation	ext2	ext4	vfat	ntfs	metafs$^{(1)}$	metafs$^{(2)}$	metafs$^{(3)}$	metafs$^{(4)}$
cp	22.3	10.9	7	167.6	5.1	9.3	12.3	10.3
ls	4.3	4	12	10.6	7.4	11.3	19.2	15.3
rm	14.4	10.1	31.7	167.7	10.4	8.8	17.7	8.9

Table 4.2: Meta data performance on flash storage.

$^{(1)}$block-size = 32 KB, write-back = full_block $^{(2)}$block-size = 32 KB, write-back = fill_rate
$^{(3)}$block-size = 64 KB, write-back = full_block $^{(4)}$block-size = 64 KB, write-back = fill_rate

The benchmark results show a significant performance drop for meta data updates (write operations performed by the `cp` and `rm` commands), on the flash device compared to the behavior on HDD. The referenced file systems, except vfat, execute the lookup of all file system objects (with full attribute stat performed by the `ls` command) a little faster on flash storage. Metafs performs better on HDD in all test cases, but shows also better or comparable

[4]http://e2fsprogs.sourceforge.net/ext2.html
[5]http://ext4.wiki.kernel.org
[6]http://www.microsoft.com/whdc/system/platform/firmware/fatgen.mspx
[7]http://www.linux-ntfs.org/doku.php

times for meta data updates in the metafs$^{(1),(2),(4)}$ configurations. The times for lookups on flash, in the metafs$^{(1)}$ configurations, are averaged between those of the native file systems (ext2, ext4) and those of the non-native (vfat, ntfs). To enable a better interpretation of the measured times for metafs, a closer look at their composition, for each test case, is needed.

4.4 Analysis and Interpretation

Each of this three test cases acts in different ways, so we have to consider each one separately, to find out what happens and why it consumes the measured amount of time. All test cases operate at the same name space name space structure, which results (for the selected block sizes of 32/64 KB) in near the same number of allocated blocks (1880 and 1881) with block fill rates as shown in appendix A. This output was generated with an additionally implemented utility named `check_metafs` to test the file system and benchmark the storage device according to selected parameters and current block utilization – data distribution. It can be run with the following options:
`check_metafs <device> <test_case>`. Four test cases, selectable by number (1, 2, 3, 4), are are available, and perform following actions on used blocks:

1. show block fill rates and block-header read-times
2. show read-times for full blocks in turn
3. show read-time for block-headers + block-content
4. show read-times for full blocks in random order

The detailed output is shown in apendix A and B.

To test the block-write performance, the file system creation program can be run with the `f` option as follows: `create_metafs <device> <volume_size in MB> <block_size in KB> <f>` Then the selected volume space is overwritten, block by block, with zeros, and the write-times for each are printed and summarized. Some interesting plots of the generated output are shown in appendix B. With this raw read/write performance benchmarks we can better evaluate the benchmarks. As the shell commands perform the same operations (read/write) on all file system objects (all blocks), the summarized times for the I/O are sufficiently meaningful. These are in best case (for 32 KB blocks):

1. min 1.5 s to read all block-headers and contents (in turn)
2. min 1.4 s to write all blocks (for this amount of data they all fit in the OS buffer cache)

and in worst case:

1. min 4.8 s to read all block-headers and contents (randomly)
2. min 8.4 s to write all blocks (they do not fit in the OS buffer cache)

Chapter 5

Conclusion

Optimizing I/O by implementing a file system is an interesting, but sometimes also a difficult and time consuming task. Apart from the fact that user-space implementation, provided by the FUSE interface, reduces the effort, many aspects need to be considered. Depending on previous knowledge in used operating systems I/O concepts and how a file systems can manage the physical and logical space by bit patterns and abstract objects, the induction overhead can be higher than in the case of a comparable stand-alone application. But once these obstacles are overcome, and the set of standard file operations is implemented, you will be able to use a whole "GNU Herd" of existing file utilities. Well, the improvement of reliability and performance is, in principle, a never ending process. Especially for the I/O subsystem there are many different ways, methods and parameters, which can be used or changed to speedup the process of storing and retrieving data. One of the difficulties, according to efficient data mapping to persistent storage, is that storage devices often act as "black boxes". Especially flash devices represent in this case a difficult to handle storage. The build-in controllers often uses different strategies to internally organize received data. So it can be pointless to exploit their advantages, without having detailed information about the device-specific features. The attempt, which was made in this work, to test the concept of a file system with "erasure block" oriented I/O, shows some benefits, compared to some other file systems on the tested flash device. To appraise the whole performance increase, by splitting file systems data and meta data storage, synergetic performance effects, caused by the reduced number of disk seeks, need also to be taken into account.

Chapter 6

Future Work

To use the implemented metafs for purposes other than those associated with file system name space manipulation, the code needs a corresponding extension. This can be accomplished by defining the pointer, which is stored in each regular file and left undefined for individual purpose. It can represent, in the simplest case, a 64 Bit address of a data content block, or a key to a more sophisticated data management. This could be an existing file systems data management or a new written one. It is also possible, to use the already implemented (meta data) address space management, for data storage. In this case, a second bitmap would manage free data blocks, and the block-header would indicate block fill-level and extent-block usage. The possibilities to implement various `read/write` functions are unlimited.

Apart from application aspects there is some work left to improve basic procedures like file system creation and maintenance. To provide the best possible write performance on flash media, an automatic benchmark which detects the optimal device-specific block size, could be executed during file system creation. An implementation of a recovery function would also be useful.

Appendix A

Block Fill Rates

All percentages are integer rounded and if the percentage of used block space is fewer 1 % – 1 is indicated (to avoid confusion).

```
[root@duo metafs]# ./check_metafs /dev/sdb 1
------------------------------------------------------------
<displaySuper>  File System Parameter: status = clean
volume_size: 4194304000 Byte = 4000 MB
block_size : 32768 Byte = 32 KB
blocks : 128000 | free : 126117 | used : 1883
-------------------------------------------------------------------------------------------------------------
Start read allocated block-header (according to bitmap): block load in [%] on:  /dev/sdb
-------------------------------------------------------------------------------------------------------------
  1  4 36  1  2  1  1  1  3  2  1  1  1  2  1  1  9  1  1  1  1  2  1  1  3  1  1  1  1  1  2  1
  1  9  1  1  2  1  1  1  1  1  1  1  1 12  1  1  1  1  4  2  3 10  1  1  1  1  2  1  1  1  7  1
  1  1  1  1  1  1  1  2  1  1  1  2  1  1  3  3  1  5  2  1  1  1  1  1  6  2 20  2  1 18  1
  1  1  1  1  1  1  1  1  1  3  1  1  1  1  2  4  1  1  1  1  1  3  1  1  1  1  1  1  4  1  1  1
  2  7  1 20  2 20  1  1  1  4  1  1  1  5  1  1  1  1  1  1  3  1  3  1  8  1  1  2  6 14  4  1
  1  1  1  2  1  1 99  1  1  4  3  1  1  1  1  5  1  1 11  1  1  1  1  1  1  1 64  5  2  1  3
  1  1  1  1  1  1  1  1  3  2  1  1  3  8  2  1  1  1  1  2  1  4  2  1  4  3  1  8  1  4 10  1
  1  3  2  1  1  1 15  4  7  4  1  1  1  1  1  1  3  1 13  8  5  3  4  1  3  1  1  1  2  1  4  1
  1  1  2  1  1  1  7  1  2  8  3  1  2 13  1  1  1  3  1  1  6  1  1 36  2  1  2  1  2  1  1  1
  2  1  1  4  1  1  1  1 15  2  1  1  1  1  1  5  1  1  1  1  2  1  1  1  1  1  7  1  3  2  5  2
  3  1  6  1  1  8  1  1  7  1  1  1  2 15  1  1  1  6  1  1  1  6  1 15  1 56  4  2  2  1  1  1
  2  8  8  2  9  3  3  4  1  1  3  2  5  6  5  3  5  2  3  2  2  1  1  4  5  2  3  1  1  2  1  2
  5  1  8  3  1  1  3  1  1  3  7  1  3  1  1  1  1  1  2  3  3  1  2  2  1  1  2  1  7 15  2  3
  1  1 21  1  1  1  1  1  2  3  1  4  6  1  1  1  7  3  3  5  5  2  5  1  1  1  1 13  2  1  1  1
  2 14  1  1  1  1  4 13  1  1 12  3  1  4  1  1  7  2  1 11  1  1 14  1  7  1  4  2 13  4  2  1
  4  1  3  3 24  9  1  1 10  1  1  5  6 12  1  1  1  1  1  1  5  4  2  2  3  4  5  3  1  1  1  1
  1  1  1  1  4  3  5  3 13  4  1  1  1  2  2  1  1  1  1  1  1  4  1  3  1  4  4  7  6 15  1  1
  3  1  2  1  1  1  3  1  1  7 22  1  4  1  2  1  1 11  2  3  1  3  1  1  1 28  3  1 26  1  3  1
  6  1  3  1  7  1  3  3  1  2  2  1  1  2  1  2  5  2  3  1  2  1  8  3  2  1  7  1  1  1  2  1
 27  3  1  2  5  3  1  4  1  1  4  5  1  2  1  1  1  1  1  3  1  2  1  3  2  1  1  1  4  1 11  1
  7  1  1  1  7  1  4 13  5  1  1  7  1  2  9  3  2  1  1  7  1  2  5  8  6  2  3  1  1  1  2  3
  1  2  1  2  2  1  1  6  1  2  1  1  3  3  1  2  2  3  1  1  1  1  2  2  3  1 16  2  1  2  1
  1  2  5  1  1 13  3  1  3  1  1  1 17  1  6  3  1  4  1  2  1  1  3  6  6  1  4  1  5  6  1  4
  1  1  4  1  1  1  1  3  1 13  1 22  1  4  1  1  1 12  3  3  2  1  1  1  2  7  2  1  1  1  3  2
 18  2  6  1  1  7  1  2  1  1  1  7  1  2  3  1  1  1  1  2  3  5  7  2  1  5  1  1  2  1  3  2
  7  1  9  8  6  2  1  1  1  2  6  5  4  3  1  1  4  7  2  4  1  3  1  2  1  1  1  2  1  2  3  3
 12  2  3  1  1  1  1  1  1  4  1  1  1  1  1  1  1  1  1  3  1  1  8  1  1  1  1  1  1  3  2  1
  4  1  1  5  1  1  1  1  1  6  4  4  1  1  1  3  1  1  3  1  1  1  1  2  1  1  1  1  1  2  1  1
  1  1  1  1  2  1  1  1  1  1 12  1  2  5  1  5  3  1  2  1  1  2  1  2  3  3  1 15  1  4  1  1
  1  1  1  1  1  1  1  1  1  1 22  1  6  2  2  2  4  8  1  1  1  1  1  1  1  1  1  1  1  1  1  1
  1  1  1  5  2  1  2  1  1  1  1  1  1  1  1  1  1  1  1  1  1  3  3 10  1  1  1 11  2 20  1  1
  1  1  3 13  1  1  4  1  1  1 28  1  4  4  2  1  2  1 23  4  1  3  1  6  1  2  1  2  1  1  1  2
  1  2  2  1  1  1  1  1  2  3  1  1  6  2  1  2  2  1  1  2  2  1  1  1  1  1  2  1  1  2  1
  4  1 11  3  1  1  1  1  1  2  1  1  2  1  1  1  2  1  2  1  1  1  3  1  4  9  1  1  1  1  1  1
 11  1  8  1  1  3  1  1  1  1  1  1  7  1  1  1  1  1  1  1  1  1  2  2  1  2  2  2  1 10  1  1
  2  1  1  2  1  3  1  3  3  1  2  1 16  1 14  5  1  4  6  1  4  2  1  4  3  1  2  1  1  2  1  1
  2  1  2  1  4  1  2  1  5  1  3  1 14  5  1  2  2  2  1  4  1  1  1  1  3  1  3  1  1  2  9  4
  1  8  5  1  5  2  1  2 14  1  1  2  2  4  1  6  1  8  8  3  1  2  1  3  1  2  1  1  5  1  3  1
  1  1  1  1  1  1  1  1  1  2  1  1 17  1  1  1  1  7  1  1  1  1  1  1  1  1  1  2  2  1  1 19
  1  7  1  2  3  1  1 21  1 13  1  1  1  1  9  2  7  1  1  1  2  2  2  1 45  2  1  1 22  1  1  3
 10  3  1  2  3  1  9  4  1  3  1  1  1  2  2  1  2  1  3  5  1  1  1  3  2  6 19 20  1  7  6  3
  1  2  1 17  1  1  1  1  1  1  3  1  1  1  1  1  1  1  1  1  1  1  1  1  2  1 16  1  1  1  1  1
  1  1  1  2  1  1  1  1  1  1  1  2  1  1  1  1  1  1  1  1  1  1  1  1  1  2  1  2  1  2  1
  1  1 14  2  1  1  1  1  2  1 15  1  1  1  1  1  1  1  1  4  3  1  2  1  1  1  1  3  3  4  1
  1  3  1  1 21  1  1  2  1  1  2  1  1  1  1  1  1  1  1  5  1  1  1  1  3  1  1  1  1  1  2  1
  1  1  1  1  3  1  1  1  1  1  1  1  1  1  1  1  1  1 17  3  1  1  1  1  1  1  1  1  1  1  1  1
  1  1  1  1  1  1  1  1 17  1  1  5  1  1  1  1  1  6  1  1  1  2  1  1  2  1  8 11  1  1  1 39
  1  2  2  1  1  1  1  1  1  2  1  1  1  1  1  1  2  3  1  1  1  4  2  1  1  1  1  1  1  1  5  1
  1  1  1  1  1  1  1  1  1  1  1  2  1  1  1  1  1  1  2  1  1  1  1  2  1  2  1  1  1  1  1  1
  1  1  1  1  1  2  1  1  1  1  1  1 14  2  1  2  1  2  1  1  3  1  1  1  1  2  7  1  1  1  1  1  1
  1  1  1  1  2  1  1  5  1  2  1  1  1  3  1  1  1  1  1  1  1  1  2  1 23  1  1  2  1 10  1  1
  1  1  1  1  1  1  1  1  1  4  1  1  1  2  1  1  1  1  1  2  1  4  3  4  4  4 15  1  1  2  1  1
  1  1  1  1  1  1  1  1  3  1  1  2  1  1  1  1  1  3  2  1  1 38  1  1  3  1  1  1  1  1  3  1  1
  1  1  1  1  2  1  1  1  7  1  1  1 47  1 24  1  1  4 13  3  3  1  1  1 26  1  1  2  5  1  1  1
  1  1  1  2  1  1  1 14  1  1  1  3  2  1  1  1  1  1  1  2  1  3  2  1  1  1 20  1  7  5  1  1
  2  9  2  1  1  1  1  1  1  1  1  1  2  1  1  1  1  1  1  2  1  1  1  1  1  1  1  1  1  1  1 30
  1  4  1  1  1  2  5  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
  1  1  1  1  1  1  9  2  1  1  6  1  1  2  1  2  1  1  1  1  1  1  5  4  3  1  2  2  1  1 48  1
  1  1  1  1  2  1  1 24  1  2  4  1  1  3  2  3  3  1  5  1  1  7  5  1  5
-------------------------------------------------------------------------------------------------------------
 Stored metadata size: 1774707 Byte [ 1.69 MB] - Space used: 0.04 %  of 4000 MB
-------------------------------------------------------------------------------------------------------------
 Read 1881 block-headers in 1s 271ms   => average block-header read-time: 675 us/block
```

Figure A.1: 32 KB block fill rates.

```
[root@duo metafs]# ./check_metafs /dev/sdb 1
------------------------------------------------------------------
<displaySuper>  File System Parameter: status = clean
volume_size: 4194304000 Byte = 4000 MB
block_size : 65536 Byte = 64 KB
blocks : 64000 | free : 62118 | used : 1882
-------------------------------------------------------------------------------------------------------------
Start read allocated block-header (according to bitmap): block load in [%] on:  /dev/sdb
-------------------------------------------------------------------------------------------------------------
  1  2 18  1  1  1  1  1  1  1  1  1  1  1  1  1  4  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
  1  4  1  1  1  1  1  1  1  1  1  1  1  6  1  1  1  1  2  1  1  5  1  1  1  1  1  1  1  1  3  1
  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  2  1  1  1  1  1  1  1  3  1 10  1  1  9  1
  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  2  1  1  1  1  1  1  1  1  1  1  1  2  1  1  1
  1  3  1 10  1 10  1  1  1  2  1  1  1  2  1  1  1  1  1  1  1  1  1  1  4  1  1  1  3  7  2  1
  1  1  1  1  1  1 82  1  1  2  1  1  1  1  1  2  1  1  5  1  1  1  1  1  1  2  1  1  1  1  1  1
  1  1  1  1  1  1  1  1  1  1  1  1  4  1  1  1  1  1  1  1  2  1  1  2  1  1  4  1  2  5  1  1
  1  1  1  1  1  7  2  3  2  1  1  1  1  1  1  1  1  6  4  2  1  2  1  1  1  1  1  1  1  2  1  1
  1  1  1  1  1  3  1  1  4  1  1  1  6  1  1  1  1  1  1  3  1  1 18  1  1  1  1  1  1  1  1  1
  1  1  2  1  1  1  1  7  1  1  1  1  1  1  2  1  1  1  1  1  1  1  1  3  1  1  1  2  1  1  1  1
  1  3  1  1  4  1  1  3  1  1  1  1  7  1  1  1  3  1  1  1  3  1  7  1 28  2  1  1  1  1  1  1
  4  4  1  4  1  1  2  1  1  1  1  2  3  2  1  2  1  1  1  1  1  1  2  2  1  1  1  1  1  1  1  2
  1  4  1  1  1  1  1  1  1  3  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  3  7  1  1  1
  1 10  1  1  1  1  1  1  1  1  2  3  1  1  1  3  1  1  2  2  1  2  1  1  1  1  6  1  1  1  1  1
  7  1  1  1  1  2  6  1  1  6  1  1  2  1  1  3  1  1  5  1  1  7  1  3  1  2  1  6  2  1  1  2
  1  1  1 12  4  1  1  5  1  1  2  3  6  1  1  1  1  1  1  2  2  1  1  1  2  2  1  1  1  1  1  1
  1  1  1  2  1  2  1  6  2  1  1  1  1  1  1  1  1  1  1  1  2  1  1  1  2  2  3  3  7  1  1  1
  1  1  1  1  1  1  1  1  3 11  1  2  1  1  1  1  5  1  1  1  1  1  1 14  1  1 13  1  1  1  3
  1  1  1  3  1  1  1  1  1  1  1  1  1  1  1  2  1  1  1  1  1  4  1  1  1  3  1  1  1  1  1 13
  1  1  1  2  1  1  2  1  1  2  2  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  2  1  5  1  3
  1  1  1  3  1  2  6  2  1  1  3  1  1  4  1  1  1  1  3  1  1  2  4  3  1  1  1  1  1  1  1
  1  1  1  1  1  1  3  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  8  1  1  1  1  1
  1  2  1  1  6  1  1  1  1  1  1  8  1  3  1  1  2  1  1  1  1  1  3  3  1  2  1  2  3  1  2  1
  1  2  1  1  1  1  1  1  6  1 11  1  2  1  1  1  6  1  1  1  1  1  1  1  3  1  1  1  1  1  1  9
  1  3  1  1  3  1  1  1  1  1  3  1  1  1  1  1  1  1  1  1  2  3  1  1  2  1  1  1  1  1  1  3
  1  4  4  3  1  1  1  1  1  3  2  2  1  1  1  2  3  1  2  1  1  1  1  1  1  1  1  1  1  1  1  6
  1  1  1  1  1  1  1  1  2  1  1  1  1  1  1  1  1  1  1  1  1  4  1  1  1  1  1  1  1  1  1  2
  1  1  2  1  1  1  1  1  3  2  2  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
  1  1  1  1  1  1  1  1  1  6  1  1  2  1  2  1  1  1  1  1  1  1  1  1  1  7  1  2  1  1  1
  1  1  1  1  1  1  1  1  1 11  1  3  1  1  1  2  4  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
  1  1  2  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  5  1  1  1  5  1 10  1  1  1
  1  1  6  1  1  2  1  1  1 14  1  2  2  1  1  1  1 11  2  1  1  1  3  1  1  1  1  1  1  1  1  1
  1  1  1  1  1  1  1  1  1  1  1  3  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  2
  1  5  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  2  4  1  1  1  1  1  1  5
  1  4  1  1  1  1  1  1  1  1  1  3  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  5  1  1  1
  1  1  1  1  1  1  1  1  1  1  1  8  1  7  2  1  2  3  1  2  1  1  2  1  1  1  1  1  1  1  1  1
  1  1  1  2  1  1  1  2  1  1  1  7  2  1  1  1  1  1  2  1  1  1  1  1  1  1  1  1  4  2  1
  4  2  1  2  1  1  1  7  1  1  1  1  2  1  3  1  4  4  1  1  1  1  1  1  1  1  1  2  1  1  1  1
  1  1  1  1  1  1  1  1  1  1  1  8  1  1  1  1  3  1  1  1  1  1  1  1  1  1  1  1  1  1  9  1
  3  1  1  1  1  1 10  1  6  1  1  1  1  4  1  3  1  1  1  1  1  1  1 22  1  1  1 11  1  1  1  5
  1  1  1  1  1  4  2  1  1  1  1  1  1  1  1  1  1  2  1  1  1  1  1  3  9 10  1  3  3  1  1
  1  1  8  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  8  1  1  1  1  1  1
  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
  1  7  1  1  1  1  1  1  7  1  1  1  1  1  1  1  1  2  1  1  1  1  1  1  1  1  1  1  2  1  1
  1  1  1 10  1  1  1  1  1  1  1  1  1  1  1  1  1  1  2  1  1  1  1  1  1  1  1  1  1  1  1  1
  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  8  1  1  1  1  1  1  1  1  1  1  1  1  1  1
  1  1  1  1  1  1  1  8  1  1  2  1  1  1  1  1  3  1  1  1  1  1  1  1  1  4  5  1  1  1 19  1
  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  2  1  1  1  1  1  1  1  1  2  1  1
  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
  1  1  1  1  1  1  1  1  1  1  7  1  1  1  1  1  1  1  1  1  1  1  1  1  3  1  1  1  1  1  1  1
  1  1  1  1  1  1  2  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1 11  1  1  1  1  5  1  1  1
  1  1  1  1  1  1  1  2  1  1  1  1  1  1  1  1  1  1  1  2  1  2  2  2  7  1  1  1  1  1  1  1
  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1 19  1  1  1  1  1  1  1  1  1  1  1  1
  1  1  1  1  1  1  1  3  1  1  1 23  1 12  1  1  2  6  1  1  1  1  1 13  1  1  1  2  1  1  1  1
  1  1  1  1  1  1  7  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1 10  1  3  2  1  1  1
  4  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1 15  1
  2  1  1  1  1  2  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
  1  1  1  1  1  4  1  1  1  3  1  1  1  1  1  1  1  1  1  1  1  2  2  1  1  1  1  1 24  1  1
  1  1  1  1  1  1 12  1  1  2  1  1  1  1  1  1  1  2  1  1  3  2  1  2
 done!
-------------------------------------------------------------------------------------------------------------
 Stored metadata size: 1774707 Byte [ 1.69 MB] - Space used: 0.04 %  of 4000 MB
-------------------------------------------------------------------------------------------------------------
  Read 1880 block-headers in 1s 229ms   => average block-header read-time: 654 us/block
```

Figure A.2: 64 KB block fill rates.

Appendix B

Banchmarks

```
[21280]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  526  0
 [21296]  0  0  0  0  0  0  0  30  0  0  726  0  0  0  0  0
 [21312]  0  0  0  31  0  0  0  0  0  0  7  0  0  30  0  0
 [21328]  0  0  0  15  0  0  16  0  14  0  0  30  0  0  0  0
 [21344]  0  0  0  0  15  15  0  0  30  0  0  14  0  0  2  0
 [21360]  0  14  0  0  0  0  0  0  126  0  0  0  0  0  0  0
 [21376]  62  31  0  0  0  0  2  0  62  0  0  0  0  0  0  0
 [21392]  30  31  0  0  63  0  0  14  0  0  30  0  0  0  0  0
 [21408]  30  0  0  14  0  0  0  0  0  0  6  0  0  30  0  0
 [21424]  6  0  0  30  0  0  6  0  0  0  30  0  0  14  0  0
 [21440]  62  0  0  0  0  0  14  0  0  30  0  0  0  2  0  0
 [21456]  14  0  0  62  0  0  0  0  0  0  0  30  0  0  6  0
 [21472]  0  62  0  0  0  0  0  0  30  0  0  30  0  0  30  0
 [21488]  0  0  0  0  127  0  0  0  726  0  0  0  0  0  0  0
 [21504]  0  0  0  0  0  0  0  0  0  0  30  0  0  0  6  0
 [21520]  0  31  0  0  0  0  15  0  0  15  0  0  14  0  0  0
 [21536]  30  0  0  0  0  0  33  0  0  0  31  0  0  0  0  0
 [21552]  0  0  0  62  0  0  0  0  0  0  0  0  14  0  0  14
 [21568]  0  0  62  0  0  0  14  0  0  14  0  0  62  0  0  0
 [21584]  0  0  62  0  0  0  0  0  62  0  0  14  0  0  0  0
 [21600]  0  30  0  0  0  0  0  0  30  0  0  0  0  0  30  0
 [21616]  0  0  0  0  0  0  0  30  0  0  0  6  0  0  15  0
 [21632]  0  30  0  6  0  0  30  0  0  30  0  0  0  0  0  0
 [21648]  0  0  0  30  0  0  0  0  30  0  0  0  0  0  0  30
 [21664]  0  0  0  0  0  14  0  0  6  0  0  30  0  0  14  0
 [21680]  0  7  0  0  0  62  0  0  30  0  0  0  0  0  14  0
 [21696]  0  30  0  0  0  0  0  0  14  0  0  0  30  0  0  0
 [21712]  0  31  0  0  0  0  0  14  0  0  0  14  0  0  30  0
 [21728]  0  0  0  0  62  0  0  0  0  0  0  0  0  0  14  0
 [21744]  0  15  0  0  30  0  0  0  0  0  6  0  0  30  0  0
 [21760]  0  14  0  6  0  15  6  0  0  0  0  0  32  0  0  0  0
 [21776]  30  0  0  0  30  0  0  30  0  0  62  0  0  0  0  0
 [21792]  0  0  0  14  0  0  0  2  0  0  62  0  0  0  0  0
 [21808]  0  0  0  30  0  0  0  30  0  0  14  0  0  14  0  0
 [21824]  30  0  0  0  0  0  0  0  0  31  0  0  0  0  0  3
 [21840]  0  0  31  0  0  14  0  0  6  0  0  30  0  0  0  0
 [21856]  0  32  0  0  0  3  0  0  62  0  0  0  0  0  30  0
 [21872]  0  14  0  0  0  15  62  0  0  62  0  0  0  0  0  0
 [21888]  0  6  0  0  14  0  0  14  0  0  30  0  0  14  0  0
 [21904]  14  0  0  14  0  0  30  0  0  0  0  0  0  30  0  0
 [21920]  0  0  0  31  0  0  0  0  0  14  0  0  0  30  0  0
 [21936]  14  0  0  0  0  0  2  0  0  30  0  0  14  0  0  0
 [21952]  30  0  0  0  0  0  14  0  30  0  0  0  0  0  62  0
 [21968]  0  0  0  0  0  0  0  0  6  0  0  30  0  0  6  0
 [21984]  0  30  0  0  0  0  0  0  14  0  0  14  0  0  30  0
 [22000]  0  0  0  0  30  0  0  0  6  0  0  14  0  0  6  0
 [22016]  0  62  0  0  0  0  0  0  0  14  0  0  0  2  0  0
 [22032]  30  0  0  0  0  0  30  0  0  0  0  0  30  0  0  0
 [22048]  0  0  0  30  0  0  14  0  0  14  0  0  14  0  0  14
 [22064]  0  0  0  30  0  0  0  0  0  6  0  0  30  0  0  6
 [22080]  0  0  30  0  0  14  0  0  6  0  0  30  0  0  0  0
 [22096]  0  30  0  0  0  30  0  0  2  0  0  30  0  0  0  0
 [22112]  0  30  0  0  0  0  0  0  14  0  0  30  0  0  14  0
 [22128]  0  3  0  0  0  0  0  30  0  0  0  0  0  0  30  0
 [22144]  0  0  0  0  32  0  0  0  0  0  30  0  0  0  0  0
 [22160]  30  0  0  15  0  0  6  0  0  30  0  0  0  30  0  14
 [22176]  0  0  14  0  0  2  0  0  30  0  0  0  0  0  14  0
 [22192]  0  0  6  0  0  62  0  0  0  0  0  2  0  0  30  0
 [22208]  0  0  0  0  30  0  0  14  0  0  30  0  0  0  0  0
 [22224]  30  0  0  14  0  0  6  0  0  0  0  0  0  30  0  0
 [22240]  0  0  0  15  0  0  7  0  0  30  0  0  0  14  0  0
 [22256]  30  0  0  7  0  0  63  0  0  0  0  0  30  0  0  6
 [22272]  0  0  30  0  0  0  0  0  0  0  0  0  6  0  0  30
 [22288]  0  0  14  0  0  6  0  0  30  0  0  14  0  0  30  0
 [22304]  0  0  0  0  30  0  0  0  0  0  0  30  0  0  0  0
 [22320]  0  30  0  0  0  0  6  0  30  0  0  0  0  0  0  0
 [22336]  0  14  0  0  30  0  0  6  0  0  0  30  0  0  14  0
 [22352]  0  6  0  0  14  0  14  0  0  62  0  0  0  62  0  0
 [22368]  0  0  0  0  0  0  0  0  0  31  0  0  14  0  0  14
 [22384]  0  0  14  0  0  30  0  0  0  0  0  0  14  0  0  14
 [22400]  0  0  14  0  0  62  0  0  0  0  0  0  14  0  0  14
 [22416]  0  0  14  0  14  0  0  6  0  0  6  0  0  0  30  0
 [22432]  0  14  0  0  62  0  0  0  0  0  30  0  0  14  0  0
 [22448]  0  2  15  0  0  0  0  31  0  0  0  0  0  30  30  0
```

Figure B.1: 32 KB block write times part 1.

```
[27312]  0  628  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [27328]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [27344]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [27360]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [27376]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [27392]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [27408]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [27424]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  628
 [27440]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [27456]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [27472]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [27488]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [27504]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [27520]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [27536]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [27552]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  726  0
 [27568]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [27584]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [27600]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [27616]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [27632]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [27648]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [27664]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [27680]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  727  0  0  0
 [27696]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [27712]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [27728]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [27744]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [27760]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [27776]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [27792]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [27808]  0  0  0  0  0  0  0  0  0  0  0  726  0  0  0  0  0  0
 [27824]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [27840]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [27856]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [27872]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [27888]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [27904]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [27920]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [27936]  0  0  0  0  0  0  0  0  0  727  0  0  0  0  0  0  0  0
 [27952]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [27968]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [27984]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [28000]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [28016]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [28032]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [28048]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [28064]  0  0  0  0  0  0  827  0  0  0  0  0  0  0  0  0  0
```

Figure B.2: 32 KB block write times part 2.

```
[15568]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [15584]  0  0  0  0  0  0  0  0  0  0  0  0  0  826  0  0
 [15600]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [15616]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [15632]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [15648]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [15664]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [15680]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [15696]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [15712]  0  0  0  0  0  0  0  0  0  0  0  0  826  0  0  0  0
 [15728]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [15744]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [15760]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [15776]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [15792]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [15808]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [15824]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [15840]  0  0  0  0  0  0  0  0  0  827  0  0  0  0  0  0  0
 [15856]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [15872]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [15888]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [15904]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [15920]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [15936]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [15952]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [15968]  0  0  0  0  0  0  0  0  0  727  0  0  0  0  0  0  0
 [15984]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  done!

 writen: 16000 blocks in 70.99 seconds
 => average block-write speed: 7.00 MB/s
 ----------------------
 Write data structures
 ----------------------
<mefs_initEntry><displayEntry>[/] data [0] mode [16895]
<writeSuper>
<writeBitmap> written Bytes  2004  of 2004

 done!
```

Figure B.3: 32 KB block write times part 3.

```
[root@duo metafs]# ./check_metafs /dev/sdb 2
---------------------------------------------------------------
<displaySuper>  File System Parameter: status = clean
volume_size: 4194304000 Byte = 4000 MB
block_size : 32768 Byte = 32 KB
blocks : 128000 | free : 126117 | used : 1883
-----------------------------------------------------------------------------------------------
Start read allocated full-blocks (according to bitmap): block read time in [ms] on:  /dev/sdb
-----------------------------------------------------------------------------------------------
 1 3 0 2 0 2 0 2 0 2 0 2 0 2 0 3 0 1 0 2 0 2 0 2 0 2 0 2 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 3 0 2 0 2 0 2 0 2 0 2 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 2 0 0 2 0 3 0 2 0 2 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 3 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2
 0 2 0 2 0 4 0 0 0 2 0 3 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 3 0 2 0 2 0 2 0 2 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2
 0 2 0 2 0 5 0 0 0 2 0 3 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 3 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 3 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 3 0 1 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 3 0 1 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 1 0 2 0 2 0 2 0 2 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2
 0 2 0 2 0 3 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2
 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0 2 0
 ---------------------------------------------------------------------------
 Read 1881 blocks in in 2s 490ms   => average block read-time: 1324 us/block
```

Figure B.4: 32 KB block read times.

```
[root@duo metafs]# ./check_metafs /dev/sdb 2
---------------------------------------------------------------
<displaySuper>  File System Parameter: status = clean
volume_size: 4194304000 Byte = 4000 MB
block_size : 65536 Byte = 64 KB
blocks : 64000 | free : 62118 | used : 1882
----------------------------------------------------------------------------------------------------
Start read allocated full-blocks (according to bitmap): block read time in [ms] on:  /dev/sdb
----------------------------------------------------------------------------------------------------
 2 3 4 2 2 2 2 5 0 2 6 3 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 3 2 2 2 2 2 5 0 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 3 2 2 2 2 2 5 0 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 3 2 2 2 2 2 4 0 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 3 2 2 2 6 0 2 4 2 2 0 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 1 5 0 2 2 2 2 2 2
 2 2 4 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 3 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 4 0 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 4 0 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 3 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 5 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 3 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 3 1 3 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 4 0 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 5 0 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 3 2 2 2 2 2 4 0 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 4 0 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 4 0 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 3 3
 2 2 2 2 3 2 2 2 2 5 0 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 4 1 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 3
 done!
----------------------------------------------------------------
 Read 1880 blocks in in 5s 47ms   => average block read-time: 2684 us/block
```

Figure B.5: 64 KB block read times.

List of Figures

List of Tables

Bibliography

[1] O. Mordvinova, J. M. Kunkel, Ch. Baun, Th. Ludwig, and M. Kunze. USB flash drives as an energy efficiency storage alternative. In *Proc. to GRID 2009*, pages 175–183, Banff, CA, 2009.

[2] JFFS2. Journalling flash file system. (version 2), 2009. `http://sourceware.org/jffs2`.

[3] UBIFS. Unsorted block image file system, 2009. `http://www.inf.u-szeged.hu/sed/ubifs`.

[4] YAFFS2. Yet another flash file system. version 2, 2009. `http://www.yaffs.net/yaffs-2-specification-and-development-notes`.

[5] LogFS. Scalable flash file system, 2009. `http://logfs.org/logfs/`.

[6] S. Danamudi. *Fundamentals of Computer Organization and Design.* Springer, 2002.

[7] Wikipedia – the free encyclopedia, February 2010. `http://en.wikipedia.org/`.

[8] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts, 7th Edition.* Wiley & Sons., 2005.

[9] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel.* O'Reilly, 3 edition, 2005.

[10] S. D. Pate. *UNIX® Filesystems, Evolution, Design, and Implementation.* Wiley Publishing, 2003.

[11] A. Rubini and J. Corbet. *Linux Device Drivers.* O'Reilly, 2 edition, 2001.

[12] M. Tim Jones. Anatomy of the linux file system, 2007. `http://www.ibm.com/developerworks/linux/library/l-linux-filesystem`.

[13] R. Love. *LINUX System Programming.* O'Reilly, 2007.

[14] R. Love. *Linux Kernel Development Second Edition.* Sams Publishing, 2005.

[15] N. Matthew and R. Stones. *Beginning Linux® Programming 4th Edition.* Wiley Publishing, 2008.

[16] W. R. Stevens and S. A. Rago. *Advanced Programming in the UNIX® Environment: Second Edition.* Addison Wesley Professional, 2005.

[17] J. Clark and A. Huffmann. Native command queuing. Technical report, Intel Corporation and Seagate Technology, 2003.

[18] A. S. Tanenbaum. *Operating Systems Design and Implementation, Third Edition.* Prentice Hall, 2006.

[19] S. C. Tweedie. Journaling the linux ext2fs filesystem. 1998.