VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

AN EFFICIENT WAY TO ALLOCATE AND READ DIRECTORY ENTRIES IN THE EXT4 FILE SYSTEM

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. RADEK PAZDERA
AUTHOR

BRNO 2013

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

EFEKTIVNÍ METODA ALOKACE A ČTENÍ ADRESÁŘOVÝCH ZÁZNAMŮ PRO SOUBOROVÝ SYSTÉM EXT4

AN EFFICIENT WAY TO ALLOCATE AND READ DIRECTORY ENTRIES IN THE EXT4 FILE SYSTEM

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. RADEK PAZDERA
AUTHOR

VEDOUCÍ PRÁCE — Ing. TOMÁŠ KAŠPÁREK
SUPERVISOR

BRNO 2013

Abstrakt

Cílem této práce je zvýšit výkon sekvenčního procházení adresářů v souborovém systému ext4. Datová struktura HTree, jenž je v současné době použita k implementaci adresářu v ext4 zvládá velmi dobře náhodné přístupy do adresáře, avšak není optimalizována pro sekvenční procházení. Tato práce přináší analýzu tohoto problému. Nejprve studuje implementaci souborového systému ext4 a dalších subsystému Linuxového jádra, které s ním souvisí. Pro vyhodnocení výkonu současné implementace adresářového indexu byla vytvořena sada testů. Na základě výsledků těchto testů bylo navrženo řešení, které bylo následně implementováno do Linuxového jádra. V závěru této práce naleznete vyhodnocení přínosu a porovnání výkonu nové implementace s dalšími souborovými systémy v Linuxu.

Abstract

The aim of this thesis is to improve the performance of sequential directory traversal in the ext4 file system. The HTree data structure that is used to store directories in ext4 at the moment works very well for random accesses, however, it is not optimal when it comes to traversing a directory sequentially. This thesis investigates the issue; it explores the implementation of ext4 and the associated Linux kernel subsystems. To assess the performance of the directory index, a set of test cases and benchmarks was implemented. Based on the analysis, an optimisation was designed and implemented to the ext4 driver within the Linux kernel. The implementation was tested, evaluated, and compared to other native Linux file systems in the last chapter of this document.

Klíčová slova

souborový systém ext4, jádro operačního systému, Linux, adresářový index, adresářová metadata, HTree, itree, sekvenční průchod adresářem, optimalizace

Keywords

ext4 file system, the Linux kernel, operation systems development, directory index, directory metadata, HTree, itree, sequential directory traversal, optimisations

Citace

Radek Pazdera: An Efficient Way to Allocate and Read Directory Entries in the Ext4 File System, diplomová práce, Brno, FIT VUT v Brně, 2013

An Efficient Way to Allocate and Read Directory Entries in the Ext4 File System

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Tomáše Kašpárka

....
Radek Pazdera
22. 5. 2013

Poděkování

Mnohokrát děkuji svému vedoucímu práce Ing. Tomáši Kašpárkovi a konzultantovi Ing. Lukáši Czernerovi ze společnosti Red Hat za jejich cenné rady, odbornou pomoc a vedení, jenž mi poskytli při řešení této práce.

Contents

Introduction

This thesis aims to improve the ext4 file system. The extended file system family has withstood many challenges since its introduction to the Linux kernel in 1992. It is only due to the continuous effort of its developers and maintainers, that the file system meets the current standards and the requirements determined by the constantly evolving hardware.

The computer industry has come a long way since 1992. Nevertheless, file systems still play a crucial role in operating systems and computer storage in general. The vast majority of applications need to store some information persistently over a long period of time. That is why we use them in the first place. Having our data disappear each time the computer is turned off is simply unacceptable.

The amount of information the computers are able to process increases as the technology advances further, which only emphasizes the need for a persistent and reliable data storage. Moreover, a file system is even considered to be the very heart of an operating system in the Unix-like environment, which is very popular for server deployments these days.

In the Linux kernel, the file system of choice has always been one from the extended file system family. Even though the Linux kernel supports a variety of different file systems, ext4 is (at the time of writing) the default option for many distributions. It is considered to be what's called the native Linux file system. The administrators choose it for various reasons. Either it's reliability, rich feature set, high level of maturity or strong compatibility between different versions. Long tradition within the Linux kernel and the conservative approach to development come as benefit to the reputation of ext4 as well.

Motivation

This work is motivated by a series of discussions held by the developers on the ext4 mailing list [1][2]. Concerns have been raised about the efficiency of the sequential directory traversal that occurs, for example, when the `ls -l` command is executed. Similar issues has been observed with other operations, that iterate through the directory index and retrieve the files from disk in order as returned by the `getdents()` system call.

A substantial decrease of performance has been reported in this particular scenario, when measured in comparison to the performance of other file systems also available in the Linux kernel. The issue is apparent especially with very large directories containing hundreds of thousands to millions of files.

The current implementation seems not to scale well to large directories, while scalability is becoming increasingly more important asset of all software solutions. Especially in the context of high-throughput machines, such as mainframes, servers and other enterprise deployments. To make a file system a viable choice, it must perform well for a large variety of workloads and use-cases of different sizes.

Goals

My goal in this thesis is to explore and analyse the above mentioned issue. To assess the existing implementation and identify the weak spots that lie behind these performance problems, and make a comparison with other contemporary Linux file systems, such as XFS and btrfs to see how it performs against its competitors in this aspect.

I would like to use the acquired information to propose an optimisation of the way the ext4 file system stores directories to remove the performance limitations that have been recognised in the current implementation.

My next goal is to implement a working solution of the problem to the ext4 file system, so it can be tested and further evaluated. If implementation works well, and the solution is accepted by the community of ext4 developers, I would like to cooperate with the upstream developers and work on merging the changes into the main branch of the Linux kernel tree.

The last goal of this master's thesis is merely a personal one. I have been a user of Linux-based operating systems and the ext4 file system for several years now. However, my experience with programming for Linux comes only from the user-space. With this project, I would like to get familiar with the internals of the Linux kernel, engage in its development, and hopefully to contribute something back upstream.

Chapter 1

File System Development

Chapter one of the thesis will provide the reader with the necessary background of file system development in Linux required to fully understand what will follow.

The first section will explain what “file system” exactly means in terms of the Linux kernel. The one after that will introduce the very heart of the storage subsystem of the Linux kernel – the *virtual file system* layer. The last two sections will describe the way the kernel interacts with the underlying block devices.

1.1 File Systems in the Linux Kernel

Linux kernel supports a variety of different file systems. Some of them are developed specifically for Linux, for instance the extended file system family, btrfs, or ReiserFS. Others were ported to Linux from different Unix-like operating systems and they are now maintained as a part of the Linux kernel tree. These include XFS from Irix, JFS from IBM’s OS/2, UFS from BSD, or even Microsoft’s VFAT. Although many of these examples are based on the same abstractions and use very similar interfaces (simply due to the fact they come from Unix-like operating systems), the concepts and storage algorithms can differ dramatically from one file system to another.

Supporting a wide range of file systems is certainly an important asset of any modern operating system. Each file system performs differently under different circumstances. For certain workloads, some are better than others. It is therefore desirable to let the administrator choose an appropriate solution based on his own needs.

File systems in Linux are not used purely for storage. As many other Unix-like operating systems, Linux employs the famous “everything is a file” principle to some extent. Although it does not apply entirely (network devices are a notable exception), there are many more services that are made available using the file abstraction in a form of various special file systems. For instance, system information have been traditionally exported through `procfs`, information about devices through `sysfs`. These two are called *virtual file systems*, because they do not operate on any storage media. The data they provide are generated on the fly. On top of that, many device drivers are designed to provide access to the devices as if they were files. The use of the file abstraction is very powerful indeed.

However, this goal is not as easily attainable as it might seem. To support a new file system, a simple addition of the code implementing it is not sufficient. There is a number of issues to consider which results in many problems to be addressed. Most notably the ways of coexistence and cooperation between the file systems.

To support this sort of variety of implementation, there is an additional layer between the implementation of the individual file systems and the rest of the kernel called *virtual file system*.

1.2 Virtual File System

Virtual file system, sometimes also called the *virtual file-system switch*, is an abstract interface that specifies what operations have to be implemented in a file system. The implementation is then accessed exclusively through this interface. Even though it is programmed in plain C, there are recognizable traits of object-oriented approach in its design.

The motivation is to have the structures in memory separated from the structures actually written to disk. These are in many cases quite similar to each other. For example, VFS defines `struct inode` while ext4 has `struct ext4_inode` and XFS `struct xfs_inode`. In respect to the abstraction, all three structures represent the same thing, but each one of them is implemented differently. This gives file system's developers the freedom to change and optimise the on-disk structures without breaking the rest of the code. File abstraction layer enables the Linux kernel to support not only the coexistence of file systems built on different principles, but also cooperation between them [3].

VFS also defines a set of operations to manipulate the structures. These operations are used by the rest of the kernel code as well as the users and programs from user-space that can access a subset of them via the system call interface. The hierarchy in which VFS takes part is illustrated in Figure 1.1. The relationship between system calls, the individual file systems' implementations, and VFS is also demonstrated in section 1.2.7, where the `getdents()` system call is examined.

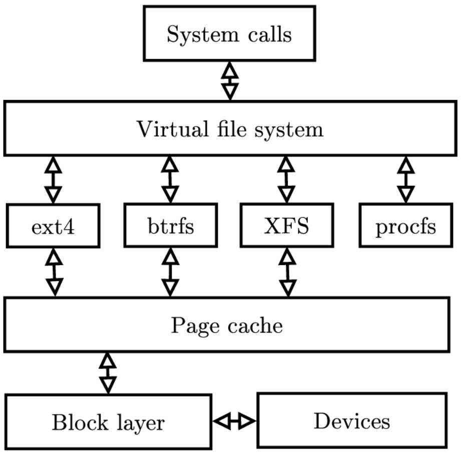


Figure 1.1: Illustration of the relationship of VFS to different parts of the kernel.

1.2.1 VFS Abstractions

The abstractions used in the virtual file system are based on the traditional Unix-style file system implementation. Despite this bias, the file model employed in Linux is common enough to represent any file system's general feature set and behaviour [3].

Apart from the file system *itself*, Unix has traditionally provided four basic abstractions related to storage: files, directory entries, inodes, and mount points [3]. The structures which represent these abstractions are often referred to as *metadata*, because they provide information about the data actually stored. All five of them will be discussed in the next few sections.

1.2.2 The Super Block

File system as a whole is represented by an object called the super block. It is defined in `include/linux/fs.h` as `struct super_block`. The Linux kernel uses it to store control information describing a specific instance of a file system, its parameters, usage statistics and more. In most cases it corresponds directly to a control block that is stored in a well-known location on disk.

Typical operations performed on this structure are creation, deletion, and mounting [3]. Pointers to the operations available for a particular file system are stored in an operation table represented by `struct super_operations`. This structure is embedded in the super block structure as `s_op` member.

The kernel maintains a list of the super block instances of all currently active file systems [4]. Each member of the list corresponds to a single mounted file system, unless the same partition is mounted multiple times.

From the user-space perspective, the super block structure is allocated when the `mount()` system call is invoked. The kernel will retrieve the file system specific super block structure from disk and use the data to fill the in-core super block.

1.2.3 Index Nodes

Index nodes, often abbreviated as *inodes*, represent the information the kernel needs to know about a file stored on disk. At this point, the kernel makes no difference between a regular file and a directory. From the user's perspective, these two are quite different, but the kernel treats them as a mere sequence of bytes or, more precisely, blocks. The differences between them are handled later on.

The indode structure defined by the virtual file system is often called *in-core*, while the file system specific structure is referred to as *on-disk*. The in-core inodes are instances of `struct inode` defined in `include/linux/fs.h`. Each structure contains information about a single file or directory including information about the owner of the file in question, its permissions, times of the last access or modification, size of the file, pointers to the file's data, and more. All inodes also have an unique identifier – an inode number.

The structure also contains a table of operations that can be used to manipulate the respective inode. The table can be obtained from the `i_op` member pointing to an instance of `struct inode_operations` (also defined in `include/linux/fs.h`). As in the previous case of the super block, each file system can provide its own implementation of each operation or use the generic one that is a part of VFS.

It is also common that inodes representing directories have different operations from the inodes representing regular files. There are, in fact, more than two types of files the kernel can work with. For example, symbolic links, device files, sockets and pipes [5]. File type is determined from the `i_mode` member. Each one of these types can have different operations associated with it and provide different functionality.

An in-core inode is allocated in memory from the corresponding on-disk inode from the underlying medium. This happens when a file from disk is requested. In-core inodes are cached in the inode cache, so any subsequent requests for a single file does not result in a series of unnecessary reads from disk.

Inodes also can be marked *dirty* to indicate that changes has been made and the respective inode should be written back to disk. It will effectively add the inode to the list of dirty inodes. Then it will be written to disk by the `write_inode` function from the super block's operation table.

1.2.4 Directory Entries

Directories are an essential file system concept. They themselves do not contain any data directly. Instead, they carry links to other files or directories. This allows users to build hierarchical structure of folders and organize their files in them. As mentioned above, Linux kernel treats directories as files. Directories are stored as *directory files*. That means each directory is a file that contains a list (or any other data structure) that maps file names to inodes. In VFS, the mapping is represented by *directory entries*.

Directory entries (often shortened to just *dentry* or *dirent*) are constructed in memory on the fly during path lookups. A directory entry is allocated for every component of a path, including the file itself. The on-disk counterparts are located within the so called *directory files*. Each file system implements the file differently, so the format of the on-disk entries can vary quite a lot.

Directory entries (dentry objects) are represented by `struct dentry` and defined in `include/linux/dcache.h` [3]. The structure contains, apart from some lists, locks, and flags, a name and the inode it points to. As the other VFS objects, also dentries have a set of operations associated with them in the operation table stored in the `d_op` member. The operation table is represented by `struct dentry_operations` defined, just as the dentry structure itself, in `include/linux/dcache.h`.

Path lookups are indeed a very common and at the same time very expensive operation. The kernel has to search whole directories and do a lot of string comparison in the process. Repeated access to a single file is not uncommon either. For these reasons, Linux implements dentry caching facility called `dcache` to speed the process up.

1.2.5 Files

Traditionally, data storage has been based on a simple abstraction of *storing files*. Due to this fact, files are in the core of persistent data manipulation. However, from the storage perspective, the kernel works only with inodes. Thus it must provide an abstraction of a file for user-space.

A file is effectively an ordered string of bytes and it is up to the application to interpret them correctly according to their internal formatting. A file opened by a process is represented by an instance of `struct file` defined in `include/linux/fs.h`. It is initialized by the `open()` system call. The kernel keeps the structures in the per-process file descriptor table. The table is basically an array of file structures and a *file descriptor* is an integer index to it. Although the actual implementation is a bit more complicated, it adheres to these principles.

The file structure again contains a pointer, `f_op`, to its associated operations table. The table is defined as `struct file_operations` in `include/linux/fs.h` header file. The correct set of operations based on the underlying file system driver is placed there during the object initialization. However, not all of the operations require the attendance of the driver, so VFS also implements generic functions of some operations to reduce unnecessary code duplication.

Most of the common file operations are well-known from user-space programming. They include, for example, reading and writing data, seeking through the file, mapping it to the memory, and more.

As the dentry objects, file objects do not correspond directly to any data stored on disk. Instead, the file object points to its associated `dentry` object via the `f_dentry` pointer [3]. The directory entry then point to an index node.

1.2.6 Mount Points

Unix employs a single file system hierarchy into which new file systems can be integrated [4]. There is one distinguished root file system and the data contained in any other file systems are accessed through *mounting*.

Each file system has its own local root directory which is, during the mounting process, attached to a *mount point* – a directory within the parent file system. The fact a directory serves as a mount point is indicated by the `d_mounted` member of the `dentry` object associated with it. Its value is incremented every time a file system is mounted to the respective directory entry. This allows the user to mount multiple file systems to a single directory, however, only one of them (the last one) can be accessed at a time.

Apart from the `d_mounted` flag, the VFS stores the information about each mount point in an instance of `struct vfsmount` defined in `include/linux/mount.h`. It links the mount point's dentry structure together with the mounted file system's super block. On top of that, it contains the mount flags (in `mnt_flags`) if any were passed, the usage count, and more.

1.2.7 `getdents()` system call

To illustrate how the kernel works with the individual file systems through the virtual file system layer, let us have a closer look at one of the system calls that interact with VFS – `getdents()`. This call is very important for this work. It is one of two ways of sequential walking through the contents of a directory in Linux kernel. The second one is the `readdir()` system call, which is now, according to the Linux man-pages, superseded by the former and should be avoided. Either way, both calls use the same underlying VFS interface.

```
int getdents(unsigned int fd, struct linux_dirent *dirp, unsigned int count);
```

Listing 1.1: Signature of the `getdents()` system call.

The system call requires the following arguments: a file descriptor of an open directory file, a buffer to store the directory entries, and a size of the buffer. It will retrieve several directory entries represented by `struct linux_dirent` and store them into the buffer. On success, it returns the number of bytes read. Zero is returned when it reaches the end of the directory.

The call is defined in `fs/readdir.c`. The implementation is quite short, in fact it is only a wrapper for a VFS function called `vfs_readdir()`. It retrieves the corresponding file object for the descriptor passed and after an successful memory access check for the buffer, it calls `vfs_readdir()`.

The job done by `vfs_readdir()` is no harder than of the parent function itself. It performs a file permission check, acquires a lock for the inode of the directory file and proceeds

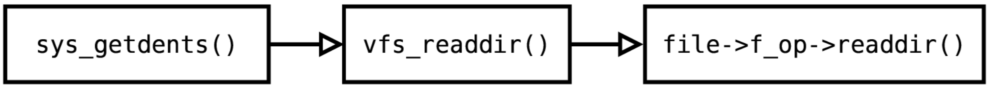


Figure 1.2: Illustration of **getdents()** system call operation through VFS.

to call the file system specific **readdir()** function from the operation table of the associated file object. This function is responsible for filling out the buffer with an appropriate amount of directory entries. This chain of function calls is illustrated in Figure 1.2. The system call layer and the virtual file system layer are just interfaces defining the required behaviour for the underlying file system implementation, where the real job is done.

1.3 Block I/O Layer

Another subsystem that surrounds the development of file systems in the Linux kernel is the *block I/O layer*. While the virtual file system interface discussed in the previous chapter is there to facilitate the communication of the file system code with the rest of the kernel, the block layer provides interface for accessing block devices, such as hard disks, floppy drives, USB sticks, or CD/DVD readers. The block layer provides an abstract interface, allowing the file system to work on any block device supported by the kernel regardless of its type or its driver.

This section will explain the principles behind the block I/O subsystem and describe the structures and function used for this purpose in the Linux kernel.

1.3.1 Block Devices

File systems are designed to work with *block devices*. The main characteristic of a block device is its ability to access data randomly (that is, at any point in the data) in fixed-size units called blocks. Block devices constitute a fairly large group of supported peripherals in Linux. A common example of block devices are hard drives.

There are two terms associated with block devices – *block* and *sector*. A block represents the minimal amount of data transferred between the kernel and a device driver. It is the smallest unit with which the Linux's block layer work with and its size is configurable. File systems work in terms of blocks. On the other hand, a sector is a fixed hardware unit. It specifies the smallest amount of data that can be transferred by the device driver from or to a physical device. [4]

A typical size of a single sector is 512B and usually, the kernel can do a little about it. The size of a block depends on the sector size; it must be an integer multiple of it. On top of that, the block size must be smaller or equal to the page size of the current architecture. This constraint is artificial; however it simplifies the kernel [3]. Typical block sizes are 512, 1024, and 4096 bytes.

Block devices are represented by instances of the **block_device** structure defined in **include/linux/fs.h**. This structure contains the device-driver oriented information about a block device [4]. It is associated with an inode that is stored in a special **bdev** pseudo file system. The **block_device** structure has a pointer to another structure called **gendisk**, which represents a *generic hard disk* – a device that comprises of one or more partitions.

A pointer to the block_device structure is kept in the super block of every file system (available through s_bdev).

Due to the nature of block devices and the performance limitations of common hard drives, the access to them is a subject to extensive caching and optimisation. This applies to the read as well as the write operations. When a block is read it is kept in memory for possible future accesses. Some read-ahead algorithms can be employed to speculatively pre-load blocks that will most likely be required shortly. Writing is usually delayed to allow the driver to dispatch multiple blocks to disk at once.

1.3.2 Buffers

Buffers can be used to describe a contiguous sequence of bytes within a page of memory. They are commonly used to identify disk blocks within memory pages. It is necessary, because a block can be equal to or smaller than a page of memory and working in page-size resolution would be a tremendous waste of available space.

Each buffer has a descriptor associated with it. The descriptor object is represented by **struct buffer_head** defined in **include/linux/buffer_head.h**. It contains all the information the kernel needs to know to be able to handle the buffer. This include its size, pointer to data within the memory page, pointer to the associated page descriptor represented by struct page, and another pointer to the block_device structure describing the underlying device.

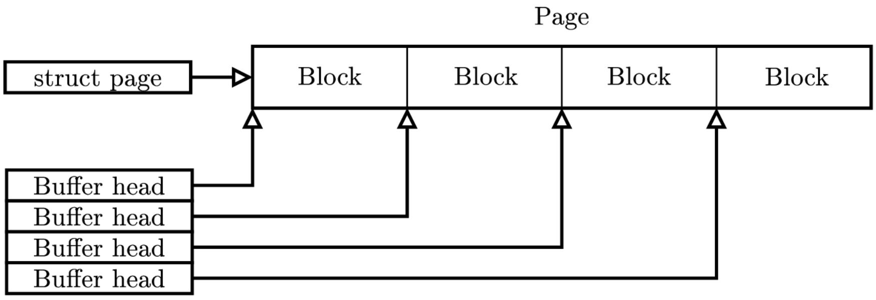


Figure 1.3: Illustration of mapping four 1 kilobyte blocks into a single 4K page.

The purpose of buffers and buffer heads in the kernel is solely to provide a mapping between disk blocks and memory pages. However, in the past its use was much broader. It used to be the unit of I/O through the file system and block layer. This task has been taken over by the bio structure which will be discussed in the following section.

1.3.3 The bio Structure

The bio structure was introduced to the Linux kernel during the block layer revamp in 2.5 development kernel. Instances of **struct bio** represent ongoing I/O operations. Its definition can be found in the **bio.h** header file located under the **include/linux/** directory.

When the kernel, in the form of a file system, the virtual memory subsystem, or a system call, decides that a set of blocks must be transferred to or from a block I/O device; it puts together a bio structure to describe that operation. [6]

The bio structure holds a list of *segments*. Each segment is a continuous part of memory; however it is not necessary for the segments to be stored next to each other. The segments are represented by the bio_vec structure, which is a 3-tuple **<page, offset, length>**.

Each vector refers to a memory page and specifies an offset from which the buffer starts along with a length of the buffer. Therefore, one segment cannot exceed the size of a page on the target architecture.

The relationship between the above described structures is illustrated in Figure 1.4. Structure `bio` has a list of `bio_vec` structures available through the `bi_io_vec` member. The number of them is indicated by the `bi_vcnt`. There is also an index `bi_idx` field available to keep track of the current position in the vector list. Vectors `bio_vec` reach to their associated page via the `bv_page` pointer.

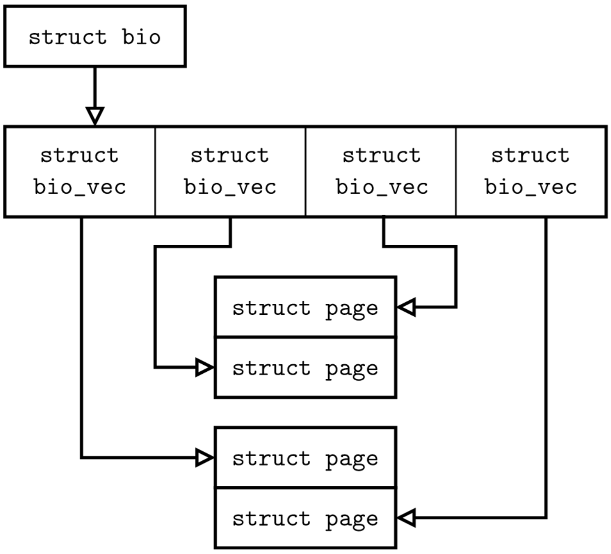


Figure 1.4: Relationship between the `bio` structure, I/O vectors, and memory pages.

Instances of the `bio` structure can be submitted for processing to the block layer via a call to the `submit_bio()` function. The request will be queued into a *request queue* for scheduling.

1.3.4 I/O Scheduling

As mentioned earlier, block devices are often slow and access times to various sections of the physical device are not always uniform. In case of the traditional hard drives, seek times depend on the position of the last read or write and they can differ substantially. To optimise the access costs even further, Linux implements several ways of scheduling the input and output requests.

Each I/O queue is managed by an I/O scheduler (or *elevator*), which is responsible for submitting requests to the device driver. It can reorder the requests in the queue to better match the needs of the underlying device and even merge some if the device supports it. For this, the Linux kernel implements several algorithms. Each device can use a different elevator. Device drivers can overwrite some functions of the schedulers and even supply their own.

The default elevator is the *Completely Fair Queuing* scheduler. It attempts to provide a fair share of the device's bandwidth for each process. It is recommended for desktop workloads, although it performs reasonably well in nearly all workloads [3].

The former defaults include the *anticipatory* scheduler, and the *deadline* scheduler, which offers good latency reduction, unfortunately at the cost of lower overall throughput.

The simplest one is the *noop scheduler* that does not sort the requests at all, it only merges them if possible.

1.4 Page Cache

This section will introduce page cache; the mechanism the kernel uses to speed up file data access. It plays a crucial role in file system development in Linux. As the name suggests, the page cache works with the whole pages of data.

Its primary task is to cache chunks of files as the user-space application access them through reads and writes. Starting from the stable version 2.4.10 [7], the page cache also takes up the role of the buffer cache and is used to keep the contents of the meta-data blocks accessed by the VFS in memory as well.

The ultimate goal is to minimize disk I/O by storing data in memory that would otherwise require disk access [3]. Accessing data on a hard drive is quite expensive. On the other hand, accesses to memory are be by several orders of magnitude faster than that.

All file reads and writes go through the page cache first, unless the file was open for direct I/O by passing the `O_DIRECT` flag to the `open()` system call [8]. In that case the cache is bypassed and the request is sent directly to the underlying device driver.

1.4.1 Cache Structure

The core data structure of the page cache is the `address_space` object, a data structure embedded in the inode object that owns the page [7]. It establishes a link between pages in the cache and the underlying device the data originates from. The structure is defined in `include/linux/fs.h`. It contains a reference to the owner inode, pointers to all the pages associated with the mapping stored in a *radix tree*[1], and also a set of operations to manipulate the mapping. The operations handle reading and writing back the pages from the underlying device, invalidating them, etc.

As was already mentioned, the unit with which the page cache works is a *page*. Each page in the system has a page descriptor associated with it represented by a `struct page` instance. This structure is actually a part of the memory management subsystem; it is defined in `include/linux/mm_types.h`. It contains two fields that are important from the page cache perspective – `mapping` and `index`. The former is a pointer to the `address_space` object the page belongs to. The latter field specifies the offset in page-size units within the owner's "address space", that is, the position of the page's data inside the owner's disk image [7].

1.4.2 Buffer Pages

In some cases, usually in association with file system meta-data, the kernel needs to work in terms of blocks, instead of pages. Blocks can be smaller than a page; and to be able to address them, the kernel uses buffer descriptors represented by the `buffer_head` structure (described earlier in this chapter in section 1.3.2). Pages that have buffer descriptors associated with them are called *buffer pages*. Each buffer page can contain one or more blocks, depending on the block size of the device and, of course, the page size available on the target architecture.

[1] http://lwn.net/Articles/175432/

When a page acts as a buffer page, all buffer heads associated with its block buffers are collected in a singly linked circular list. The `private` field of the buffer page points to the buffer head of the first block in the page [7]. Consequentially the `PG_private` flag of the buffer page is set; and, in context of the page cache, it indicates that the page has buffers associated with it.

All buffer pages are stored in the address space of the master `bdev` inode of the device they are associated with. The file system implementation can use a function called `sb_bread()` from `include/linux/buffer_head.h` to access individual blocks on a file system. The function takes two arguments, a super block and a logical block number of the block to read and returns a buffer descriptor for the requested block.

1.4.3 Cache Eviction

The size of files stored on disk usually vastly exceeds the size of available system memory. Additionally, the memory cannot be used only for caching. The kernel can keep only a very limited amount of pages in the cache. A so-called *cache eviction* algorithm must be implemented in the kernel to remove unused pages from the cache and free the memory.

To facilitate proper cache eviction, Linux implements a daemon called *pdflush* running in the background as a one or more kernel threads (the number is dynamically adjusted).

A modified version of the least recently used algorithm called *LRU/2* is used. There are two queues to keep track of the usage of the page. Freshly accessed pages are put in front of the first queue, the *passive* one. In case an entry is accessed again while it is in the passive queue, it is moved to the second one, the *active* queue. The pdflush threads always remove the pages from the back of the passive queue.

Chapter 2

The ext4 File System

The following chapter gives an overview of the design and implementation of the fourth incarnation of the extended file system. Apart from the very first section, which presents shortly the history of the extended file system family, this chapter will introduce the physical data structures, the related algorithms, and discuss the current directory file indexing approaches used in ext4.

All references to the Linux source tree in this chapter are relevant to the stable upstream Linux kernel of version 3.6.3.

2.1 Brief History

The first development version of ext4, the *ext4dev*, was accepted to Linux kernel during the merge window of 2.6.19 in 2006. It was simply a copy of the existing ext3 code with multiple patches that meant to improve the storage limits imposed from the use of 32-bit block numbers and the 32,768 limit on subdirectories, increase the resolution of timestamps, and address some performance limitations [9], which were rejected for inclusion to ext3 due to concerns the developers raised about the stability of such changes.

The works on the development branch went on for approximately two years and in Linux 2.6.28 (released in December 2008) ext4 was marked stable and ready for adoption.

The fourth extended file system is, as his predecessor, backward compatible with the other members from the extended family. It is therefore possible to mount ext2 or ext3 file systems as ext4. Additionally, ext3 is partially forward compatible with ext4 meaning that ext4 can be mounted as ext3 unless it uses extents instead of indirect blocks for mapping file's data blocks.

In fact, *extents* are one of the most notable features introduced in ext4, finally replacing the traditional and largely inefficient block mapping scheme. Managing blocks using extent trees helps to improve the performance of large files and to reduce fragmentation. Use of extents enables ext4 to support volumes with sizes up to 1 EiB and files with sizes up to 16 TiB. Similar improvements of performance and decrease in fragmentation are consequential to implementing the *delayed allocation* technique. Additionally, the ext4 directory indexing feature `dir_index` is now enabled by default.

It is expected, that ext4 will be replaced by btrfs as the default Linux file system in the future. The developers work on stabilizing it. However, btrfs is (at the time of writing) still not ready for deployment in production environment.

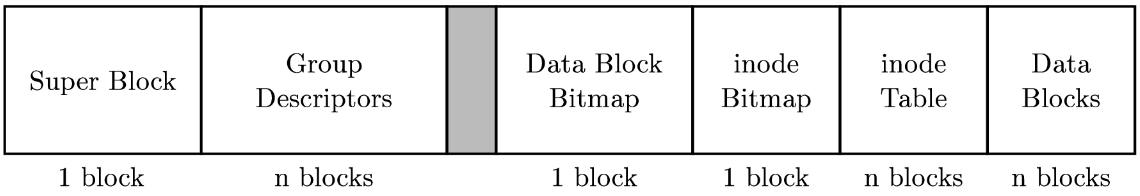
2.2 On-disk Structures

This section introduces the data structures actually written to disk when ext4 stores data. It is based on the exhaustive description of the on-disk layout from ext4's wiki page by Darrick Wong [10] as well as on exploring the ext4 code base and experimenting with utilities such as `debugfs` and `dumpe2fs` from the e2fsprogs package.

2.2.1 Layout

Every instance of the ext4 file system is divided into a series of *block groups*. Each block group can contain a copy of the super block and group descriptors followed by data block and inode bitmaps (the exact conditions will be explained later on). After them comes only the inode table and data blocks. Figure 2.1 illustrates a single ext4 block group layout.

As opposed to placing all meta-data to the beginning of the file system, this approach guarantees that the meta-data are distributed evenly across the file system and it makes it possible for the allocator to optimise placement of both data and meta-data to achieve better performance. All the meta-data structures will be described in detail in the following sections.



Super Block	Group Descriptors		Data Block Bitmap	inode Bitmap	inode Table	Data Blocks
1 block	n blocks		1 block	1 block	n blocks	n blocks

Figure 2.1: Layout of a single block group in ext4.

All fields in ext4 are written to disk in little-endian order, with the exception of the journal jbd2 which uses big-endian order. There are also some important values and constants (usually stored in the super block) that influence the layout.

Block Size

File system block is the smallest unit of data that can be manipulated at once. The size of a single block can have a serious impact on the layout decisions made by the file system. In ext4, several other important values, such as the size of block and meta-block groups, depend on the base logical block size. Minimum block size in ext4 is limited to 1024 Bytes while the maximum can be set up to 65KiB. In practice, it is often configured accordingly to the *page size* of the target machine, as it is the upper limit.

It can be determined from the value of `s_log_block_size` member of the super block structure by substituting it for n to the following formula 2^{10+n}. This form of recording the block size is used to make sure sane values are used.

Size of a Block Group

The size of a single block group in an individual instance of ext4 is specified by a value in super block called `s_blocks_per_group`. Alternatively, it can be calculated as 8×block_size. Very common number of blocks in a single group is 2^{15}, which corresponds to block size of 4096B. It equals to 128MiB of space in a single block group.

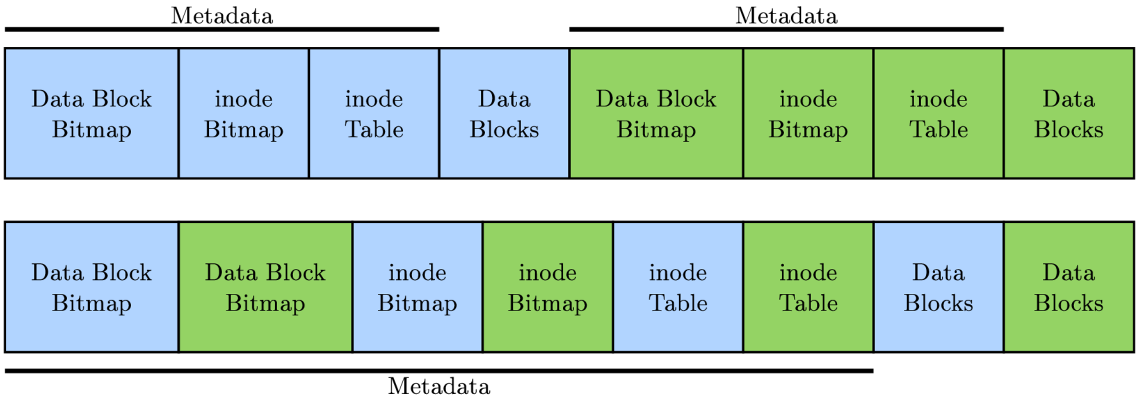


Figure 2.2: Illustration of the flex_bg feature principle on two adjacent block groups.

Flexible Block Groups

With the introduction of extent trees to ext4 as a way of addressing data blocks, fairly small block groups started to become more of a limiting factor. Extents represent a continuous series of adjacent data blocks and with traditional block groups the space is partitioned to relatively small parts (128MiB on a file system with 4 KiB block size).

To overcome this problem, *flex_bg* feature was added to ext4. If enabled, several block groups are tied together and form a logical block group called *flex*. The number of block groups that make up a flex block group is given by 2^n where n is determined by a value of `s_log_groups_per_flex` from the super block.

Both bitmaps and the inode table from all the groups in a flex are packed together and stored in the first block group, thus allowing to use the remaining space of the flex block group to store files' data blocks. The layout change is illustrated in Figure 2.2.

However, the redundant copies of the super block and group descriptors will not be moved, so to take a full advantage from flex_bg, sparse_super feature should be enabled as well.

Meta Block Groups

Normally, a complete copy of the entire block group descriptor table is kept after every copy of the super block. When the file system contains enough block groups (volumes over 256 TiB), they will fill the entire block group leaving no space for anything else.

With the meta_bg feature, the ext4 file system will be partitioned into many meta block groups. Each meta block group is a series of block groups whose group descriptor structures can be stored in a single disk block. For ext4 file system with 4 KiB block size, a single meta block group partition includes 64 block groups, or 8 GiB of disk space. The meta block group feature moves the location of the group descriptors from the first block group of the whole file system into the first group of each meta block group itself. The backups are kept in the second and last group of each meta block group. [11]

2.2.2 The Super Block

The super block data structure of ext4 contains various information about the file system, settings and parameters of the specific file system instance. This includes the block size used, the total number of blocks and inodes available, maintenance information, such as mount count, the time of the last fsck, and more. In the kernel, the super block is represented by `struct ext4_super_block` defined in `fs/ext4/ext4.h`.

On disk, a copy is kept in the beginning of the file system, right after the boot sector from offset 1024. Apart from that, backup copies of the super block are kept in the beginning of every block group. It carries data very important for the existence of the file system, so maintaining multiple copies at various places is more than desirable to be able to recover at least some bits of data after losing the beginning of the file system (which is not that uncommon).

The backup copies are not used or even accessed during the normal operation of the file system. They are updated and synchronized with the active ones during the consistency checks done by the *e2fsck* program from *e2fsprogs*.

However, having so many copies can be quite inefficient with today's volume sizes measured in terabytes. For instance, on 2 TiB volume with the block size equal to 4 KiB, there will be 16,384 copies of the super block. If the sparse_super feature is enabled, redundant copies of the super block are kept only in the groups whose group number is either 0 or a power of 3, 5, or 7. In the case of the previous example, the number of copies will be reduced to 19.

2.2.3 Block Group Descriptors

Apart from the global information that ext4 keeps in its super block, some additional meta-data have to be maintained on a per-block-group basis. Each block group has a group descriptor associated with it. A table of all the block group descriptors, called the group descriptor table, is located after every copy of the super block in a file system. The `sparse_super` feature discussed above applies to the block group descriptor tables as well.

An on-disk block group descriptor is represented by a structure called `ext4_group_desc` that is defined in `fs/ext4/ext4.h`. It contains block numbers of both the block and inode bitmaps, and the inode table of the corresponding block group. Besides those pointers, each descriptor also contains the numbers of free blocks, free inodes, and used directories in the group. These numbers help the block and the inode allocators make better decisions while looking for the optimal placements for new allocations.

2.2.4 Block and Inode Bitmaps

The bitmaps track the use of data block and inodes within a block group. A zero bit within the bitmap indicates that the corresponding block or inode are not currently in use and are free for allocation.

The location of the bitmaps within a block group is not fixed. They can float depending on whether the block group contains a copy of the super block and the group descriptors. They can even be located out of the block group completely (as illustrated in Figure 2.2) if the `flex_bg` feature is enabled.

Each bitmap takes up exactly one file system block. This fact determines the size of each block group of a file system. For example, if the block size is 4096 bytes there will be 32,768 bits in the block bitmap. Consequentially, the size of each block group is 32,768 blocks times the block size

$$32768 \times 4096 = 128 \text{ MiB}.$$

2.2.5 Inode Table

Inodes in ext4 are evenly distributed along the whole file system. It is beneficial for performance reasons, because then the allocator can place the data and meta-data closer together. An *inode table* is a part of each block group on the file system. Its size is again limited by the block size. Due to the fact that the inode bitmap discussed in the previous section takes up only one block, the maximum number of inodes per block group is 32,768. This consequentially limits the number of files that can be created on a file system.

The number of inodes per group is specified in the super block by a member called `s_inodes_per_group`. It is decided at file system creation time and from that point on, it is fixed and it cannot be changed.

Inodes within a file system are numbered sequentially from number 1, so in order to locate one, the driver does not need to search, instead it can use a couple of very simple calculations to convert an inode number to a block number and an offset to its inode table. The index of a block group containing an inode of a specific number can be calculated as

$$\frac{\mathit{inode\ number} - 1}{\texttt{s\_inodes\_per\_group}}$$

and the offset into the group's table is;

$$(\mathit{inode\ number} - 1) \bmod (\texttt{s\_inodes\_per\_group}).$$

Physically, the inode table is an array of the ext4's inode objects. The structure `ext4_inode` is very similar to the in-core inodes that are a part of VFS (described in chapter 1). Its definition can be found in `fs/ext4/ext4.h`. It contains values such as, the size of file in bytes, access and modification times, user permissions, the UID and GID of the file's owner, and more. However, the most interesting field is the `i_block` which contains a block map or an extent tree to mark which blocks carry the data.

The default size of an inode in ext4 is 256 bytes; however, effectively used is only 156 bytes. The extra space can be used for extended attributes [10].

2.2.6 Indirect Block Mapping

The ext2 and ext3 file systems both use a direct/indirect block mapping scheme to address data blocks associated with an inode. This scheme is very efficient for sparse or small files, but it has high overhead for larger allocations [11]. In ext4, this scheme has been superseded by the extent trees, nevertheless, it is still supported for backwards compatibility reasons.

Inside ext4's inode, under the `i_data` field, there is space for 15 block numbers (total of 60 bytes). These block pointers specify, in which blocks the file system keeps the data associated with this inode. The first 12 of them points to data blocks directly and the last 3 are used for indirect mapping. The 13th entry of the `i_data` field points to one indirect block, the 14th to a double indirect block, and the 15th to a triple indirect block. The indirect block contains references data blocks directly, the double indirect block points to indirect blocks, and the triple indirect block contains pointers to double indirect blocks.

The indirect and double indirect blocks are illustrated in Figure 2.3. Blue fields in the picture represent entries that point directly to data blocks. Indirect pointers are painted green, double indirect red, and triple indirect purple. Note that the triple indirect mapping is not pictured whole (to save space).

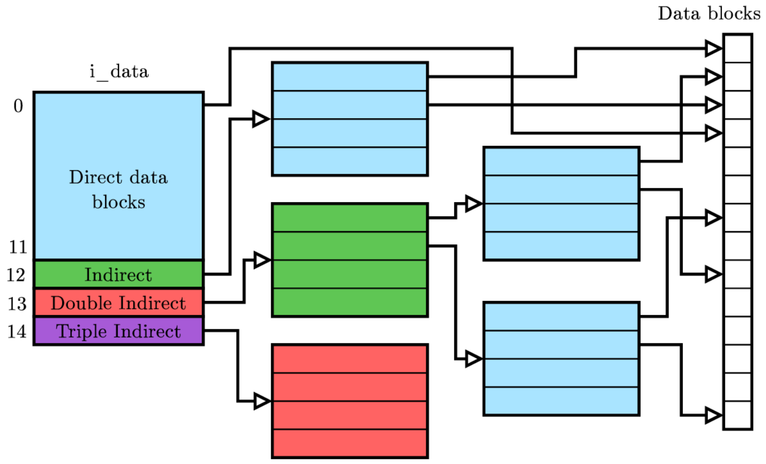


Figure 2.3: Example of the indirect block mapping.

All three types of indirect blocks contain a simple array of 32-bit block numbers referring to different blocks on the file system (indirect or data). Considering a block size of 4096 bytes, a single indirect block can carry up to 1024 pointers to other blocks. Due to this fact, the indirect mapping scheme cannot work with block numbers that exceed the traditional 32 bits.

Small files (up to 48 KiB with 4096 bytes block size) usually do not require any indirect blocks, so the mapping is very efficient. Unfortunately, as the file size grows, the number of indirect blocks required to keep track of all the data blocks grows very rapidly.

2.2.7 Extents

As the size of common file systems and the size of the files computers can handle increases, the indirect block mapping scheme becomes more and more inefficient. To address this issue, the developers introduced *extents* as the major new feature of the transition from ext3 to ext4.

An *extent* is a single descriptor which represents a range of contiguous file system blocks [11]. Single extent can cover up to 32768 blocks (128 MiB with 4K block size). This is very efficient in comparison to the indirect block mapping scheme. In practice, the vast majority of files on a standard file system will take no more than 3 extents [9]. Extents also benefit from the improvements of the block allocator that were also a part of ext4's development. Figure 2.4 shows the structure of the descriptor. Basically, it contains three integer numbers, the first logical block number, size of the extent, and the physical block number from which the extent is stored on disk.

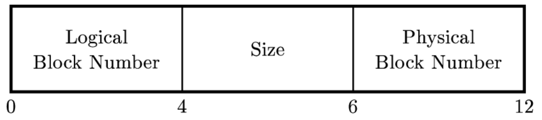


Figure 2.4: Extent descriptor structure layout.

Extent Tree

Extents are stored in a B^+ tree data structure called the *extent tree*. Each node of the tree consists of a header followed by an array of entries. The header contains information about the depth of the node within the tree, number of entries in the node, and its capacity. The entries can be of two types – index entries and extents.

Interior nodes of the tree, marked by depth greater than zero, contain exclusively index entries, only the leaves can carry the extent descriptors. Index nodes simply point to either leaf nodes with extents or alternatively another index nodes. The root of the tree is stored directly within the inode; each of the other nodes takes up a full disk block. Figure 2.5 shows the the structure of the ext4's extent tree.

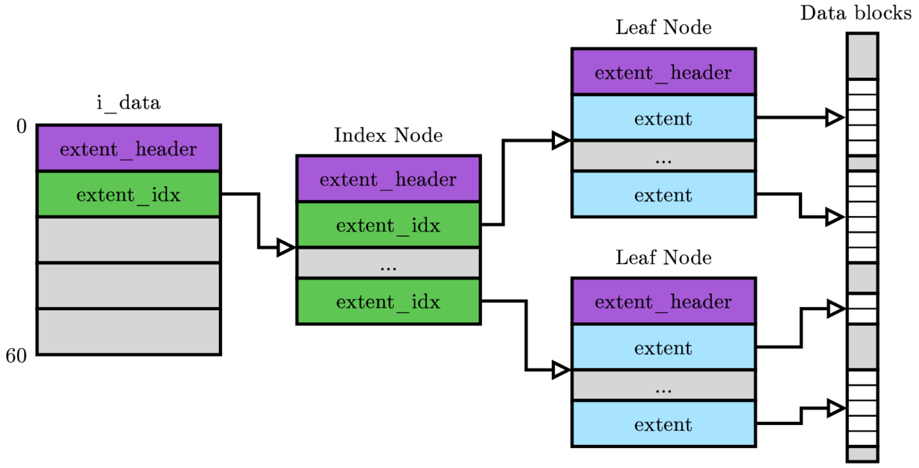


Figure 2.5: Structure of the extent tree.

Three different data structures are used in ext4 to represent the extent tree. The header is the same for all node types and it is laid out in `struct ext4_extent_header`, index entries are represented by `struct ext4_extent_idx`, and extent descriptors are instances of `struct ext4_extent`. All three are defined in `fs/ext4/ext4_extents.h`. They all are conveniently of the same size of 12 bytes.

2.3 Directory Index

The ext4 file system supports two ways of storing a directory. The first available option is to populate the directory file with a linear list of entries. This approach is simple, but not very efficient, especially with large directories with many files. The linear approach has been replaced in ext3 by a special indexing tree, which offers much better performance in most cases. Ext4 supports both schemes for backward compatibility reasons. The linear approach is still used for very small directories that do not exceed the size of a single file system block. Both approaches will be explained in the following sections.

2.3.1 Linear Directories

Using this approach, the directory file will contain a simple linear array of directory entries. There are two versions of the structure that represents each directory entry within the list. The original directory format used the `struct ext4_dir_entry`, but the structure was slightly altered and **`struct ext4_dir_entry_2`** was created later. Although, both of them are supported for compatibility reasons. The latter is the default one, unless the file type feature is disabled (by omitting the flag in the super block).

Both structures contain the name of the file and a block pointer to the inode on disk that the name is associated with. The difference between these structures is in the length of the **`name_len`** field that contains, as its name suggests, the length of the file name. The former version of the structure reserves two bytes for it. However, file names are limited to 255 characters in Linux. Therefore, the developers decided to shorten the field and use the additional byte to store file type.

```
struct ext4_dir_entry_2 {
        __le32  inode;                  /* Inode number */
        __le16  rec_len;                /* Directory entry length */
        __u8    name_len;               /* Name length */
        __u8    file_type;
        char    name[EXT4_NAME_LEN];    /* File name */
};
```

Listing 2.1: Full definition of the directory entry structure.

Finally, both structures also contain the **`rec_len`** field that stores the length of the whole entry. This is used by entries located at the end of each block of the directory file. Occasionally, gaps can appear when there is still space in the block, but it is not enough to store the whole next entry. In these cases the size of the last directory entry in a block is adjusted to take up the remaining space.

This, although fairly easy to implement, approach has a significant performance disadvantage: each directory operation (create, open and delete) requires a linear search over an entire directory file [12]. This results in quadratically increasing the cost of operating on all files of a directory, as the number of files in the directory increases [12].

On the other hand, due to its simplicity, it is quite efficient for very small number of entries. This makes it still a viable choice for directories smaller than one block, where the advantages of more complex solution do not yet outweigh its additional overhead.

2.3.2 Indexed Directories

When a directory file exceeds the above mentioned one block of size, an index will be added to speed up the lookup of files in the directory. The use of this approach is indicated by an inode flag `EXT4_INDEX_FL`.

The indexing feature was introduced in 2001 by Daniel Phillips. A data structure crafted specifically for the use in ext4 called the *HTree* is used, which reduces the complexity of individual directory operations from $O(n)$, i.e., linear search over the directory to $O(\log(n))$. The entries are still stored using the same structure **`ext4_dir_entry_2`**. However, in this case, they are not arranged into a linear array. They are stored inside the leaf nodes of the tree. The principles of HTrees will be explained in the following section.

One of the most interesting features of the index is that it is backwards compatible with older versions of the file system. The tree is hidden inside the directory file, masquerading as empty directory data blocks [10]. For the previous versions of the extended file system, the directory appears as a well formed. It works, because an empty entry in the directory table is signified by setting the inode number of an entry to 0 (there is no inode zero).

2.4 HTree

The HTree data structure is in principle a B$^+$ tree. The key used as an index in the tree is a hash value of the name of the associated directory entry. Its characteristics lie somewhere between those of a tree and a hash table [12]. It offers a good performance while retaining some of the simplicity of implementation.

The B$^+$ tree is an ideal data structure for this purposes in ext4. It separates the sequence set (in our case the directory entries) from the index. The sequence set is contained only in the leaf nodes, while the index provides information to speed up random accesses into the sequence.

In this section, we will not explain the principles behind B-trees and B$^+$ trees alone. Instead, we will focus on the particular implementation of the tree used in the extended file system. The principles of multiway trees were described by Donald Knuth in his famous series about computer programming [13] and specifically B-trees are very well explained in an article by Douglas Comer [14].

There are 2 types of nodes used in HTree. Each node takes up a whole file system block. The very first block (logical block 0) of a directory file is always occupied by an *index node*. The root can point either to more *index nodes* or to *leaf nodes*. Index nodes contain block pointers to other nodes within the tree (either index or leaf). The leaf nodes contain an array of directory entries.

The depth of the tree is currently limited to a maximum of 2 levels. Two level directory index can accommodate more than 30 million directory entries, which is more than sufficient [12].

2.4.1 On-disk Structures

The on-disk layout of the directory index is a bit more complicated due to the efforts to maintain backwards compatibility. The index blocks act as if they were unused directory entries in order to remain ignored by the older code. The full structure of the tree is shown in Figure 2.6.

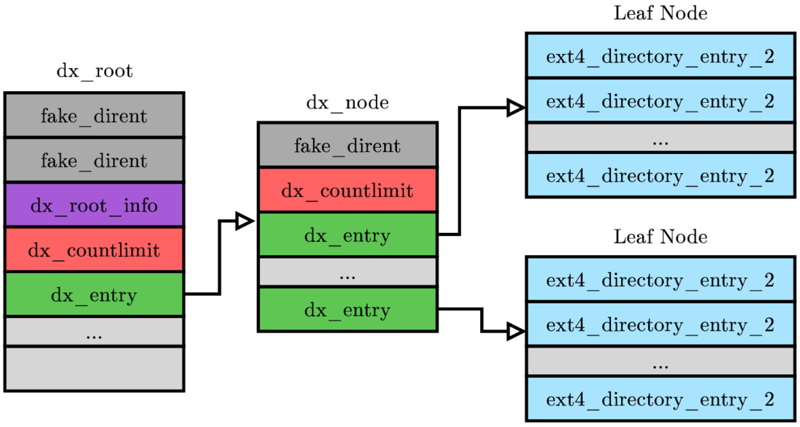


Figure 2.6: Structure of the HTree illustrated.

The root node is represented by `struct dx_root`. The structure contains two fields of type `struct fake_dirent` from the start. This is because of the convention that first two entries are always the "." and ".." pointing to the current directory and its parent respectively.

These are therefore excluded from the tree and arranged from the beginning of the structure. The latter, the dot-dot entry, has its size set as if it was stretched all the way to the end of the block. This will effectively hide the rest of the block from the legacy code.

These two directory entries are followed by the `dx_root_info` structure, which holds the control information about the directory index, such as the type of the hash function used, and the number of indirect levels in the tree.

Directly after the info structure, there is an array of *index entries* represented by `struct dx_entry`. Each index entry has two fields a `hash` and a `block`. Each entry points to a subdomain of the B^+ tree and contains a block number of the block in which the subdomain is stored. The entries are ordered by the hash value. It is important to note, that all block pointers within the directory index contain logical block numbers within the directory file, not physical block numbers.

The first index entry of the list is unused. It is overlaid by `struct dx_countlimit`, which contains the maximum number of index entries that can follow, and the actual number of entries that follow this header. The limit field determines the order of the B^+ tree and it depends on the block size of the individual file system.

The second type of node, the *index node*, is quite similar to the root. It masquerades itself as an empty directory entry. The first field of a directory entry (as showcased in Listing 2.1) is a pointer to an inode, therefore starting with four zeroed bytes will trick the older code to assume the entry is empty. Size of this entry is again set as if it stretched all the way to the end of the block. After comes a list of index entries, as in the case of the root node.

All the structures mentioned above are defined locally in `fs/ext4/namei.c`. They are not available anywhere else in the code.

2.4.2 Tree Operations

There are three basic operations commonly performed on a HTree. Inserting a new entry to a directory, deleting an entry from a directory, and searching for a particular entry within a directory. At the first glance, these three operations may seem very different from each other, but they are in fact very similar. When inserting an entry to a tree, it is necessary to find the right place for it. The same applies to the delete operation – when deleting an entry from a tree, one needs to search for it first. This means that the first two operations are basically the same as the last one, except there is an action performed after an entry is found.

A search through the index can be performed with logarithmic time complexity

$$O(\log_b(n))$$

where b is the order of the tree, i.e., the number of entries in each node and n is the total number of entries in the tree. The time complexity of an insert or removal of an entry from a leaf node is constant as the nodes are of a constant size. Therefore the complexity of all the operations performed on the HTree is logarithmic.

Chapter 3

Analysis and Benchmarks

This chapter contains the analysis of the issues which cause the observed decrease in performance of ext4, while working with large directories. Based on the carried out experimentation, I designed and implemented a set of test cases and benchmarks to assess the current implementation of the directory index in the ext4 file system. The tests will be useful not only to identify the issue; they will also help us to evaluate the future optimisations and possibly also find regressions in the new code.

The first section contains a description of the issue, based on the description that has been presented in discussions on the linux-ext4[1] mailing list. The following section contains an analysis of the file system operations related to directories in ext4 and their behaviour with respect to the description of the issue. The following section describes the design of the tests and benchmarks that were developed to be able to verify and further investigate the issue. In the last two sections of this chapter, the results of these tests along with a brief comparison with a few contemporary Linux file systems are presented; concluding the analysis.

3.1 The Issue

The implementation of the HTree directory index has brought an interesting side effect to the file system. It was discussed on numerous occasions on the ext4 developer's mailing list [1, 2]. Daniel Phillips discussed this problem in the original paper presenting the initial directory index implementation in 2002 [12]. Cao et al. pointed it out again in a paper discussing the status of ext3 in 2005 [15].

The HTree data structure orders directory entries by their hash. Consequentially, when a directory is read using the **`readdir()`** library call, the entries are returned in hash order. However, due to the random characteristics of the hash function used, the ordering is practically random. This is desirable from the perspective of the directory index implementation and it works very well. However, it is far from optimal in case we would like to manipulate the directory as a whole. The ext4 file system stores the inodes in an ascending order, so the optimal way of reading them to avoid seeks is precisely in that order.

Currently, when an application attempts to access all the files in a directory in the order in which `readdir()` returns them, it will access the inode table blocks on the file system in a random manner. A single inode table then can be retrieved from disk multiple times during the directory traversal. In the worst case scenario, the block can be retreived from

[1] linux-ext4@vger.kernel.org

disk for every inode in it. This is certainly not ideal and it can lead to disk head thrashing when the directory is large enough.

3.2 Analysing Directory Operations

This section will analyse the possible impact of this issue on various file system operations. First, the `readdir()` library call is described, followed by a discussion of various operations that can be performed over a directory. I used `strace`[2] and `ltrace`[3] call tracers to examine what actually happens during the execution of different commands.

3.2.1 readdir() library call

This library call provides the user-space interface for reading a directory file. It is a part of the GNU C library and it is in compliance with the POSIX.1-2001 standard. It accepts a pointer to a directory handle of type `DIR` and returns a pointer to the next directory entry represented by `struct dirent`. When the end of the directory stream is reached, null pointer is returned. Internally, the function uses the `getdents()` system call to retrieve directory entries from the kernel. The system call was described earlier in section 1.2.7.

It is important to point out, that this function and therefore also the underlying system call do not access the inodes of the files contained in the directory on disk. It only reads the inode of the directory file. Therefore, calling this function alone is not necessarily slow nor suboptimal.

3.2.2 Creating Files

The operation of inserting a file to a directory is in case of ext4 quite efficient. Traversing the HTree to find a suitable leaf block is done in logarithmic time. The tests I performed show an outstanding performance, even in comparison to other Linux file systems. The results will be discussed in section 3.4. Creating new files certainly is the type of operation that benefited the most from the directory index implementation.

3.2.3 Deleting Files

Random file deletion is an operation similar to the file creation discussed above. File system driver must locate the leaf block for a particular entry and remove it. Also with the logarithmic complexity.

However, the deletion of a whole directory can be affected by disk thrashing, provided the directory file is large enough to exceed the size of the available page cache. During file deletion, the file system driver has to touch the inodes in order to decrement their link count [12].

3.2.4 Listing directories – the `ls` command

Listing the contents of directories is one of the most common and the most well known directory operations from the user's perspective. The `ls` command has been traditionally used for these purposes in the Unix-like operating systems. By default, it only prints a sorted list of file names. The names can be attained directly from the directory file. Time

[2] http://sourceforge.net/projects/strace/

[3] http://sourceforge.net/projects/ltrace/

complexity of listing the file names from the whole directory is therefore equal to the of a single call to `readdir()`.

The situation is different in case the user requests more information about each individual file contained in the directory. For instance by executing the list command with the `-l` option to print file permissions, sizes, and more. The `ls` command uses the `stat()` system call to get the information about each file as it reads the directory. Ultimately, it leads to the retrieval of the inodes of all the files in the directory. Returning inodes in random order in this case can lead to suboptimal performance.

3.2.5 Copying – the `cp` command

The copy operation is quite common as well, especially in production environment where full backups are performed on a regular basis. When copying a whole directory, the kernel needs to retrieve each file and write it one by one to a different location. This is done as the program traverses the directory using `readdir()`.

The situation is similar to directory listing, but the kernel needs to retrieve the file's data as well as their inodes. The amount of blocks read depends on the size of the directory. However, reading lots of additional blocks can have a negative impact on the page cache, evicting the pages with inode tables. If the kernel accesses them in a random manner, it means the disk might have to seek for every inode.

Copying large directories, in terms of number of files as well as their size, is the operation that suffers the most when reading the files in random order. Substantial slowdown, in terms of multiple hours in comparison to reading the files in inode-order can be observed in the results of my tests.

3.3 Tests and Benchmarks

This section will cover the design and the implementation of the test suite I developed specifically for assessing and measuring the performance of the ext4's directory index. The conditions required to reproduce the issue are described first, followed by a description of the two groups of test cases that are a part of the dir-index-test[4] suite.

3.3.1 The Minimal Reproducer

Several factors can affect the occurrence of disk head thrashing from accessing inodes in random order. If the directory is small or conversely, if the target machine has a large amount of memory, the page cache will reduce the number of random seeks by keeping the inode tables cached. The slowdown will become more severe either with an increase in the size of the directory or decrease in the amount of available memory for page cache. This is determined by the configuration of the target system and also by the utilization of system resources at the time of the test.

Based on the carried out experimentations, I identified the easiest and at the same time the most effective way of reproducing the issue to be copying files. When the file system driver also has to read the content of each file, it will evict the inode tables from cache so the disk has to retrieve them again and seek.

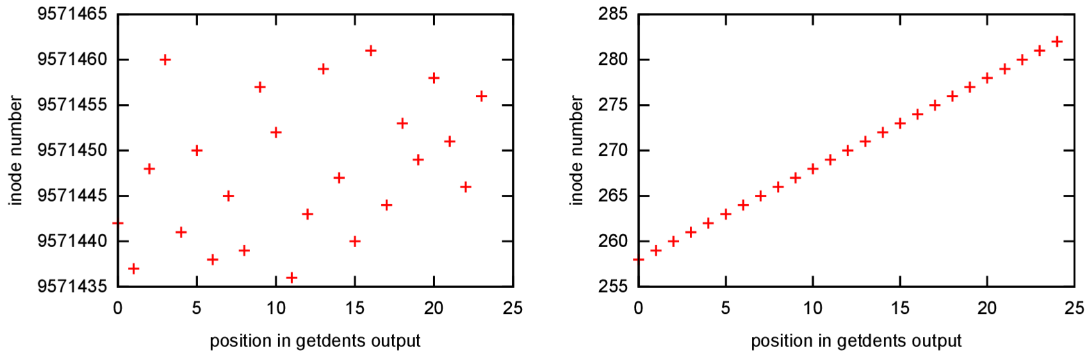
[4] https://github.com/astro-/dir-index-test

3.3.2 Order Tests

As was discussed in the sections above, the order in which inodes are returned when reading a directory file from disk is crucial for the performance of the subsequent operations on the files; especially within large directories. The purpose of these tests is to capture and visualize the order in which a file system driver returns directory entries to the user-space.

I use multiple approaches. The first one is based on a test script posted to the ext4 development mailing list by Phillip Susi [2], which calculates the correlation between the number of inodes that were preceded by inodes with smaller number and the total number of inodes returned. In an ideal case, the correlation would be very close to 1.

The second approach is to visualize the ordering in a plot. The X-axis variable will be the sequence number with which was the inode retrieved from the kernel and the Y-variable will be the inode number on disk. This will effectively visualize the theoretical path the disk head has to take in order to retrieve the sequence of inodes in this particular order. An example of this plot is shown in Figure 3.1.



(a) Ext4

(b) Btrfs

Figure 3.1: Sequences returned by `getdents()` for a directory of 25 files.

3.3.3 Benchmarks

The second part of the test suite is comprised of benchmarks to measure the performance of various directory operations in different conditions. The purpose of these tests is to ascertain what operations suffer from this problem, and to what extent are they slowed down. These benchmarks are also used for the comparison with other Linux file systems.

The performance of several operations is measured, including copying files between two physical devices, listing files in a directory, creating a tar archive from a directory, creating and deleting a whole directory. Apart from those, I also created an isolated test case to measure the performance of just `getdents()` followed by calling `stat()` on every file in the directory. This test case helps to isolate the problem from another software layers build on top of those system calls.

The benchmarks are focused on measuring the performance of operations that read a directory and then manipulate the data in it in some way as these are the ones to be affected by the issue described earlier. However, we must not forget to measure the performance of the remaining operations related to directories (file creation and deletion). These benchmarks will be useful later on to verify that the optimisations have not caused any harm to other parts of the file system.

3.3.4 Testing Conditions

To get a better picture of what exactly is the cause of the above mentioned performance problems, the benchmarks will be performed in various circumstances.

To determine how the performance of the operation changes with the size of the directory, I used a number of different values (up to 5 million files in a single directory).

Also, two different sizes for the files in the testing directory will be used; 0 bytes, i.e., empty files and 4096 bytes, i.e., files taking up a single file system block. The more blocks a file system has to handle during an operation the less effective the page cache is to compensate for the unnecessary I/O operations. Pages are evicted more often in case of the increased load.

To simulate heavy load even more, the benchmarks can be run in memory pressure. This can be simulated by allocating large amounts of system memory during the test.

3.3.5 Tests Implementation

Most of the tests and benchmarks were implemented as a combination of shell and python scripts. Some test cases are programmed directly in C to avoid any distortion caused by other software layers on top of system calls. The test suite with all its sources is publicly available on Github[5] under the terms of GNU General Public License.

Apart from the test cases themselves, multiple processing scripts are also a part of the test suite. These are used to calculate statistics, format them into tables, and plot the results in graphs using the GNU plot utility.

The performance test cases are run twice. During the first run, only the time of execution is measured. During the second run the Seekwatcher utility[6] is used to collect statistics about which blocks were read from disk and to visualize the I/O patterns.

The scripts are arranged so they can be run in batch by executing the main script called `run_tests.sh`. This script will run all the test cases and benchmarks, and gather their results. Everything is processed automatically, so the results of the tests are available in the form of tables and graphs.

3.4 Test Results

This section presents the test results that were observed. All tests were executed using a desktop system with Intel Core 2 Duo 6600 processor running at 2.40GHz, 4GB of memory, and two SATA hard drives – 250GB Western Digital WD2500AAKX-0 and 320GB Segate ST3320620AS. The first one was used as a primary test device, while the second one served as a scratch device for copy tests.

The results show that the performance of ext4 during file creation and deletion is outstanding. It easily outperforms XFS, JFS, and is almost twice as fast as btrfs.

When it comes to listing a directory (i.e., accessing only its meta-data), ext4 still outperforms btrfs, due to caches that from certain point contain all necessary inode tables and compensate for the seeks. However, it is roughly 25% slower than XFS in this case.

Table 3.1 shows a detailed overview of durations when copying files between two file systems. All files contained a single block of data, in this case 4096B. Copying is certainly

[5] https://github.com/astro-/dir-index-test

[6] https://oss.oracle.com/~mason/seekwatcher/

Files	btrfs	ext4	ext4-spd	JFS	XFS
250,000	224.269	2,348.615	115.579	165.895	121.202
500,000	426.306	5,129.581	224.902	343.293	257.047
750,000	596.551	7,670.962	371.185	562.602	399.921
1,000,000	803.226	10,605.363	728.488	782.527	552.79
1,250,000	1,028.372	13,569.547	687.927	1,008.731	704.262

Table 3.1: Comparison of the copy times on different file systems. The ext4-spd column contains the results for ext4 with the `readdir()` library call patched to cache and sort the entries.

where the issue manifests the most. Ext4 falls long behind all other file systems. It is almost twenty times slower than XFS.

Interestingly enough, if we sort the inodes in the user-space, using the modified version of `readdir()` mentioned in section 3.2, ext4 performs quite well in comparison to its opponents. This shows that there certainly is a space for improvement.

This performance issues were also confirmed by the order tests. Ext4 with the HTree accesses the inodes during the copying in virtually random order. On the other hand, btrfs and XFS go through the sequence in an ascending order. This is crucial to avoid unnecessary seeks. Based on the tests I did on an aged file system, btrfs is able to maintain the perfect ordering for I/O even after random deletions and creations. However, XFS suffers from fragmentation and the ordering tends to degrade over time, which results in a substantial drop of the performance of directory traversal. Ext4 still returns entries in a random order, no change there.

Full test reports and results summaries including the plotted data are available on the media attached to this document.

Chapter 4

Possible Solutions

Now that we have analysed the issue and tested the impact of it on various directory operations and their performance, we will discuss the possible solutions of the problem. In this chapter, three proposals will be presented.

The first section describes the least complicated solution to the problem – mere sorting of the inodes returned from the `readdir()` system call. The following section explains a proposal from Andreas Dilger and Daniel Phillips that incorporates making changes to the ext4's inode allocator to make pre-allocations for directories and then allocate new inodes in hash order, rather than allocating them sequentially. The last approach that is described in this chapter constitutes the adding of an additional tree to the file system that would provide a different view on the data that is more suitable for sequential access while retaining the current tree for random accesses.

A brief comparison of the three proposals is made in the last section of this chapter.

4.1 Sorting Directory Entries in User-space

The very first solution that comes to mind is to simply sort the inodes before they are returned to the user so the following accesses to disk are in the ideal order from the file system's perspective. Some applications have been using this workaround to improve their performance when used with the ext4 file system. However, forcing the user to sort the inodes before accessing them is not very convenient and not very efficient either, as every application would have to contain this sorting code. Nevertheless, there are a few places on the path from the kernel to the user-space code where the sorting could possibly be done.

We cannot sort the entries directly in the kernel, because the amount of kernel space memory is limited and it cannot be swapped out. This brings us to one of the biggest downsides of sorting the entries – to be able to sort them, we need to read all the entries from disk first, and store them in memory. This can lead to using quite a lot of RAM, as the program might need to load millions of entries.

In the worst case scenario, each directory entry can be 263 bytes long (file names are limited to 255 characters in Linux) so a directory with a million files can take up as much as 250MB of memory. Reading 5-million directory entries would require more than 1GB of free system memory. An operation as simple as reading a directory should not require that much memory. In addition, the memory used for directory entries for the time of the operation would not be available for the page cache. Smaller available page cache could slow down the operation on the directory, because of the increased amount of I/O.

Besides sorting the entries within the kernel, the sorting could be done in the user-space as well. The `getdents` system call is rarely used directly; most of user-space code accesses this functionality through the `readdir` library call. This function could be changed to sort the inodes before giving them to the user. In fact, there is a modified version already available in *e2fsprogs*[1] that will sort the directory entries before passing them to the user (available in `contrib/spd_readdir.c`).

Based on the benchmarks I did, using this library pre-load works great as a workaround for some cases where there is enough memory available for storing the directory entries. The CPU overhead of sorting the entries before they are returned is insignificant compared to the great increase of performance of the subsequent operation on the directory.

Nevertheless, the same concerns about space complexity apply for this approach as well. This could even become a security issue, as a user with malicious intentions could force a system to create one or more large directories and then cripple the system by using all of the memory by simply accessing them.

One way to overcome these limitations would be not to sort the whole directory, but use a fixed-size buffer for directory entries and sort it in parts. This approach was proposed on the linux-ext4 mailing list by Andreas Dilger.

I made a patch[2] for the `spd_readdir` pre-load to allow limiting the size of the buffer and performed benchmarks of it with different buffer sizes including 1,000, 10,000, and 50,000 directory entries. This approach helps to increase the performance to some extent. Naturally, the bigger the buffer, the bigger the improvement in performance.

When the sequence is sorted in several steps, the I/O operations are performed in *waves* throughout the file system. The performance increase then depends on the number of waves, i.e., the number of times the disk has to go through the whole sequence. The performance increase then depends primarily on the size of the sequence and the size of the buffer used. The layout of inodes in the sequence is important as well. The more scattered the inodes in the directory through the file system, the less effective this optimisation will be.

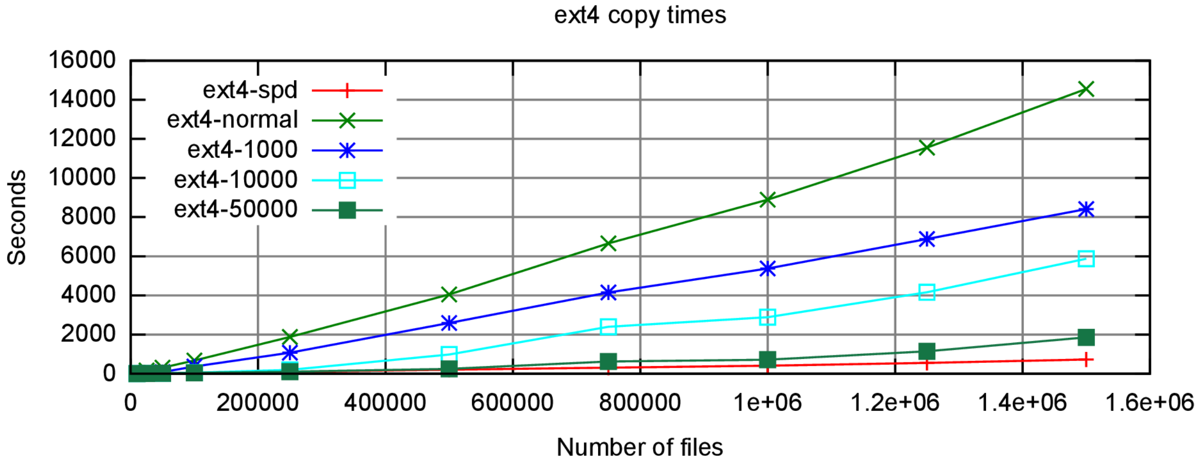


Figure 4.1: Copy durations for different buffer sizes. The worst case in the graph was measured without the pre-load entirely; the best case is with unlimited buffer size.

An example of this benchmark is shown in Figure 4.1. The graph displays the increase in copy times based on the size of the directory being copied. The performance increase

[1] http://e2fsprogs.sourceforge.net/

[2] https://github.com/astro-/dir-index-test/blob/master/src/spd_readdir.c

is substantial, especially in the case of buffering 50000 files, where the kernel has to go through the inode tables about 30 times. This is much better for the cache compared to random access, but there still can be a lot more I/O operations than necessary.

4.1.1 Evaluation of the Proposal

This approach can be used to solve the issue and greatly increase the performance of the ext4 file system, especially for full backups and similar workloads. It has some limitations and downsides, such as excessive memory usage and increased CPU usage during the sort that might negatively impact the performance of the system in the worst case scenarios.

In my opinion, this approach is more of a workaround that can be used in case the system has plenty of memory available. Alternatively, the patched version `spd_readdir` implementation can be used, where the administrator can fine-tune the limits on the buffer to fit the needs of the particular system.

Applying this solution generally as a part of the GNU C library might be problematic. The library is used on a large variety of devices ranging from embedded devices (such as Raspberry Pi) to large production servers. And while it is unlikely to encounter directories containing millions of files on a system-on-a-chip computer, it is not impossible. The smaller device naturally do not have that much resources to handle the additional processing as easily as large production machines, so the algorithm would most certainly need to adjust the buffering parameters accordingly to the resources currently available on the system.

Besides, there are other contemporary file systems to ext4, such as XFS or btrfs that return inodes in the optimal order already. In fact, it seems as something the file system driver should do, instead of depending on external library code to perform well.

4.2 Inode Preallocation

The idea behind this solution was introduced by the Linux kernel developers Andreas Dilger and Daniel Phillips in 2003. Phillips presented the proposal in an email posted to the linux-ext4 mailing list [16].

This proposal was picked up later by a kernel developer Coly Li who worked on implementing it into the extended file system [17, 18]. The work seems to be finished, but, for some reason, the project was abandoned and the code was not merged to the mainline.

This approach incorporates making modifications to the inode allocator of ext4 to increase the correlation between the hash order and the order in which the inodes are allocated. Normally, the inodes within a directory are allocated linearly as close to the directory inode as possible. This leads to increased cache pressure on directory traversal as some inode table blocks might be retrieved repeatedly from the disk [17].

In the ideal case, the inode allocator would return the inodes based on the position of the associated directory entries within the directory file. So when the directory is read sequentially in hash order, the file system could process all the entries within a single inode table block while it is still in the cache.

Unfortunately, this is not possible, because we do not know the exact size of the directory nor all the file names ahead of time. Therefore, we cannot make a precise decision where to put the inodes while they are allocated. However, Dilger and Phillips propose that this decisions can be approximated rather well using pre-allocated sections of the inode table for each directory.

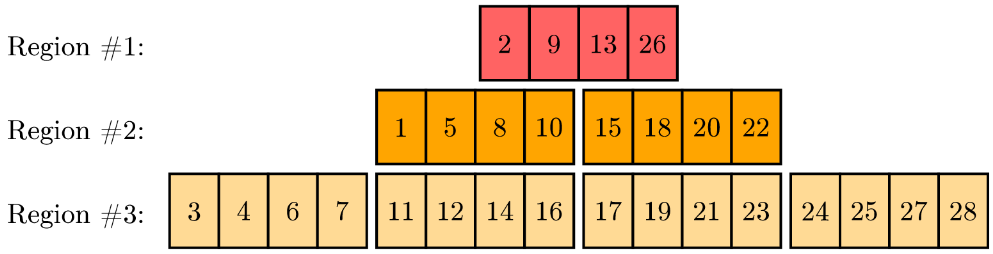


Figure 4.2: Three hash ordered regions of the inode table

The main idea is to reserve a region in the inode table for each directory when it is created. This space would be used to allocate the inodes from in an approximate hash order. A new, twice as big, area would be reserved when the directory outgrows the space of the pre-allocated region and the hash order allocations would continue from there.

This would then result in several regions within the inode table and each one of them would be roughly in hash order. Occasionally, some inodes might have been allocated out of order, because of an unfortunate sequence of hash values. But this shouldn't happen too often due to the random nature of the hashing functions used. Each region then represents a group of non-overlapping blocks. Therefore, when reading the directory in hash order, there would be at most a single block required from each sorted group in which the next inode could reside. This would decrease the cache load from $O(n)$ to $O(log_2(n))$ [16].

Figure 4.2 shows how three pre-allocated regions could look like. The first region (a single inode table block) would have to be in cache for the whole directory traversal. The two blocks from the second region would spend in cache only a half of the time of the traversal each, as the first block of the second region would not be accessed after moving to hash values bigger than 10. Each additional region would increase the number of blocks in cache by only one additional block.

4.2.1 Evaluation of the Proposal

The inode pre-allocation seems to be a very elegant solution to the problem. It does not require any changes to the disk format, so it is completely compatible with older kernels and file systems. It would also require only a small number of lines of code to implement [16]. Andreas Dilger predicts the number of lines to roughly 500 [17].

However, the pre-allocations could bring some negative side effects to the file system that could outweigh its benefits. Due to the reserved regions in the inode table, the directories would tend to spread apart more quickly than before. The inode table could become fragmented when creating too many directories and the file system would need to fall back to the original allocator. This might be a problem especially with the big directories where, at some point, creation of a single file would result in pre-allocating a million of inodes at once.

It would also take some time to find and mark the new regions to reserve during directory creation or in case the previous region has been filled. This operation could potentially be very time consuming especially with large directory files.

4.3 Adding an Auxiliary Tree

The directory data structure is handled in two different ways. As a part of the file system tree, it acts as a container of other files while it is an entry of its parent directory at the same time. While manipulating the directory, we might want to do two types of operations; either to manipulate the entries within the directory or to manipulate the whole directory as if it was a single file.

The first view of directories is utilized typically for path lookups, inserting, deleting, or otherwise manipulating individual files within the directory. For these operations, the most effective way of indexing the entries is by the file name. In case of ext4, this is achieved by using the hash generated from the file name that is associated with the entry as a key to the B^+ tree that contains the entries. These operations are without a doubt very important and their performance plays a crucial role in the overall usability of the whole file system in the majority of workloads.

The other view comes to effect when the directory is manipulated as a whole, including operations, such as the creation, deletion, and traversal of the entire directory. The typical workload from this category is doing full backups. In that case, the whole directory file and all the files stored under the directory must be read and copied to a different location. For the best performance of this group of operations, the ordering of entries must reflect the way the directory is laid out on the file system to avoid unnecessary I/O.

Merging these two views of the directory so the file system performs well in all the cases described is a very difficult task, provided that there is no relationship between the file's name and its location on disk.

However, to get a good performance of all the operations, the file system could use two separate indexes with the directory file, one in the hash order for quick file lookups and an auxiliary index for fast manipulation of the whole directory. This way, we would get the full benefit of both of the indexing approaches for all directory operations. The operations involving only individual entries in the directory would use the hash-ordered data structure while the directory oriented operations would use the other one.

One problem of this approach is the fact that every modification of the directory would require modifying both data structures, which could potentially introduce some delays. The ext4 file system uses a B^+ tree on which all the operations can be done in logarithmic time. Therefore, we cannot use any structure simpler than a tree in order not to affect the performance of inserts or deletes.

Another problem with the auxiliary tree comes from its key – the inode number. The ext4 file systems allows creating up to 65,000 hard links to a single inode. The tree would not be able to differentiate between these colliding entries and would handle them as a single sequence. This would result in a severe performance drop in this particular scenario. The creation of thousands of links to a single file is not very common, yet it is not impossible.

Fortunately, this problem can be easily overcome by using a mixed key comprising of the inode number along with the hash associated with the directory entry. Therefore, we will reduce the problem of hard link collisions to collisions between 32-bit hash values which are very rare indeed.

Adding a second tree to the directory would roughly double the size of the meta-data associated with the directory file, as the same information must be stored on the disk again only in a different order. Making the new data structure use the directory entries that are stored in the original tree would introduce different problems with performance during the traversal.

4.3.1 Evaluation of the Approach

The biggest drawback of this solution is the necessity to change the on-disk format. The changes could be made backwards compatible quite easily. However, achieving full forward compatibility would be problematic. The ext4 partitions with the new feature could be mounted by older kernels only for reading, because the older kernels couldn't update the newer tree while adding files to a directory.

The directory file would be bigger and inserting or deleting a file from the index might be affected, because of the necessity to insert the entry to both of the trees. The page cache should compensate for this to some extent, given that both trees will reside quite close to each other, so these delays shouldn't be too big.

The trade-off for these possible drawbacks is solving the problems entirely for all possible cases. With the additional tree, the inodes would always be returned in the ideal order for sequential reading. The order would not suffer from degradation caused by file system ageing or fragmentation.

4.4 A Solution to Implement

Three possible approaches to improve performance of the ext4 file system while manipulating large directories were presented in this chapter. The first and the least complex solution involves sorting the entries at some point after they were read from the directory file. The second approach is an attempt to create some degree of correlation between the ordering of the directory file and the order in which the inodes are allocated on the file system. The last presented solution goes the other way around the problem. It involves the addition of a second tree to the directory file that would provide information about the placement of the files on the disk.

From the perspective of this thesis, the first approach is more of a workaround, rather than a proper solution. I would like to try to solve the problem directly in the file system driver, rather than patching several user-space applications to do the sorting.

On the other hand, the last two approaches both seem as viable options that could fix the problem in ext4. An attempt to implement the inode pre-allocation idea has been made in the past already, but the development work stopped before it could be merged to the mainline.

Therefore, I decided to work on the last presented approach and implement an additional tree to the directory index of the ext4 file system. The expected benefits and downsides of this solution are clear. This idea has not been tried yet in the ext4, and by implementing it, I would like to test whether the trade-off is viable. It will also serve as an alternative to the inode pre-allocation patches that exist already. The design and implementation of this solution will be described in the following chapter.

Chapter 5

The Design and Implementation of the Inode Tree

This chapter describes the design and the implementation of an additional indexing tree to the ext4 directory file. The inode tree should be used by the file system to improve performance while working with very large directories.

The first section is dedicated to the design of the *itree* feature from the perspective of the user of the file system. It describes the file system flags that are associated with this feature and its level of backward and forward compatibility. It is followed by two sections that explain the design of the new tree and how it will affect the existing directory indexing implementation. These sections are still focused more on the design and of high-level view, unlike the very last two sections of this chapter that describe the low-level bits from the implementation of the inode tree in the Linux kernel.

5.1 The File System Feature Design

The first decision to be made concerns the level of compatibility of the new feature. To assure the backward compatibility will be easy, because the newer kernel can work with the older file systems simply as it did previously and ignoring the new code. On the other hand, maintaining a full forward compatibility will not be possible. An older kernel will be able to read the new on-disk format without noticing the new blocks, but it won't be able to modify it correctly. It would add the entry to the old directory index, but, for the obvious reasons, it would not be able to change the new tree. Any modifications to a directory with this feature enabled from an older kernel would lead to diversion between the two trees. The index would be destroyed and it would have to be rebuilt during a file system check.

With these things in mind, I will add a new read-only compatible feature flag and also a new inode flag. The file system flag will indicate that the ext4 instance in question, may contain directories with the itree enabled, and the inode flag will mark all the directories with the tree. The flags and their assigned bits are in Table 5.1.

Adding a file system flag is convenient, because it makes the feature optional. The optimisation will have its downside and the trade–off may not be favourable for every possible use-case of the file system. It is best to let the administrator decide whether the feature is viable for the particular workload or not.

Users will be able to enable or disable the *itree* feature when creating the file system by passing a flag to `mke2fs`. The feature can also be turned on or off later on using `tune2fs`.

Flag Name	Bit	Mask
`EXT4_FEATURE_RO_COMPAT_ITREE`	13	0x1000
`EXT4_ITREE_FL`	30	0x20000000

Table 5.1: The file system and inode flags associated with the new itree feature

I created a patch set[1] for `e2fsprogs` that adds support for these options to the `mke2fs`, `tune2fs`, and `dumpe2fs` utilities.

5.2 The Design of the Inode Tree

Now let us proceed to the design of the inode tree itself. From a theoretical standpoint, it is a B^+ tree. There are actually two reasons why I have chosen this type of data structure. The first one is to maintain the time complexity bounds of the directory operations. The original HTree is a kind of B^+ tree too, so it is necessary to use a data structure with at most $O(\log(n))$ time complexity for the insert, delete, and search operations. Apart from that, I would like to add as little complexity to the existing code as possible. B^+ trees work very well with block-oriented storage devices and given that there is one already implemented in ext4, there is a chance I will be able to reuse some code as well. The order of the tree will be determined by the block size of the individual file system instance. With 4kB per block, each non-leaf node will have room for up to 227 entries.

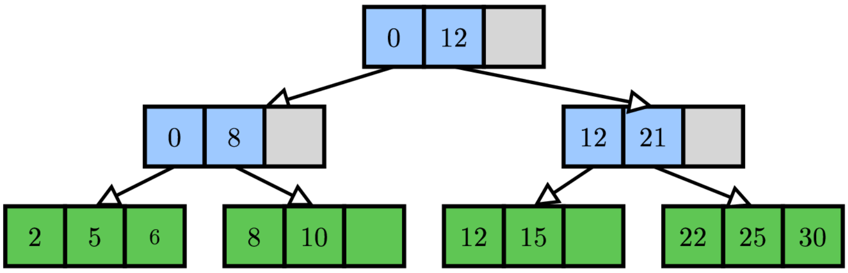


Figure 5.1: An example of a B^+ tree of the order 3.

The entries in the tree are ordered by the inode number, but the actual key is compounded. It is a 64 bit vector, comprising of an inode number and a hash, which are associated with the entry. The hash value is there to resolve the collisions between multiple hard links to the same inode within a single directory. With ext4, the user can create as much as 65,000 names for a single file. All these hard links would collide in case they all reside within the same directory.

Working with collisions requires special care from the kernel, because it cannot differentiate between them using the key. That is a problem, because we cannot use the key to remember the position within the sequence. We could try to use a byte offset to the entry on disk. Unfortunately, this is not possible, because, in the concurrent environment that is an operating system, the entries can easily move during a node split, invalidating any offset we might have kept to be able to continue processing.

[1] http://marc.info/?l=linux-ext4&m=136770333515230

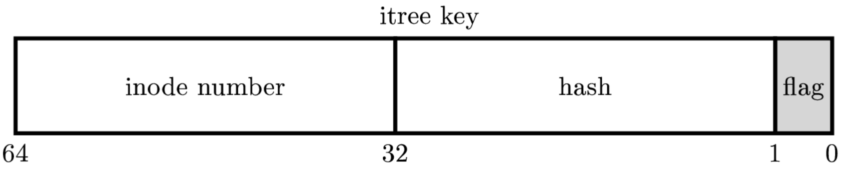


Figure 5.2: The structure of the key used for the inode tree.

Therefore, the kernel must manipulate all the colliding entries at once, read them all from disk and cache them in memory. However, working with a sequence of 65,000 entries at once would be extremely inefficient and it might cause severe performance issues. Even though it is unlikely for the users to create as many names for the same file from a single directory, it is not impossible and the kernel must be able to work well, even in the corner-case scenarios.

To eliminate these collisions, a 16 bit unique sequence number could be added to the key. The problem here is how to effectively assure the uniqueness of this number. Having to walk through the whole sequence, in order to find the first free number in it, would again have a negative effect on the performance of inserting to the tree. Therefore, I decided to use the hash number which is already computed by the original itree code, and the values are almost unique. There still will be some key collisions, but they will happen far less often, and the kernel won't have any problems with caching only several directory entries.

The collisions pose another problem when it comes to splits. If a node should be split just between two colliding entries, the entries before the collision would be skipped during a traversal. The situation is shown in Figure 5.3.

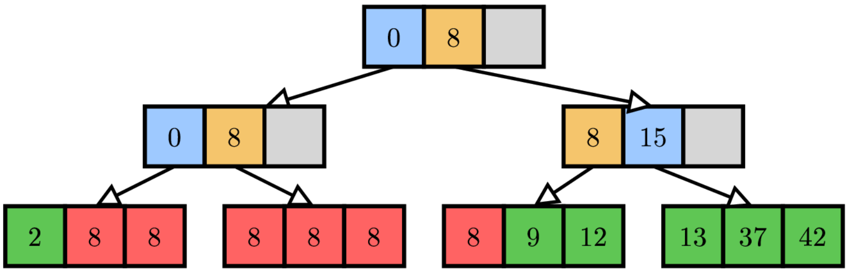


Figure 5.3: A split collision in a B$^+$ tree. This will not work correctly. All the entries with key=8 except the last one will be skipped during a traversal.

To overcome this issue, we could either disallow splits within collisions altogether or make the tree traversal algorithm aware of the fact. I decided to use the second approach to stay close with the HTree, which uses flags in the index nodes to indicate collision splits. It is convenient, because the least significant bit in the hash value is actually unused, so we can use it for the flag the same way as the original tree.

The depth of the tree is limited to a maximum of three levels. The limitation is there only to avoid heap memory allocations for certain variables. Having a rather small upper limit on the number of levels allows us to allocate the memory statically on the stack. A three level tree can take around 12 million of entries in the worst case scenario of 255 characters per file name and each node of the tree only half-full. In a much more likely case of 20 bytes per name and 75% average fullness, the tree capacity grows up to 500 million of entries. These values are sufficient for now, but in case they are exceeded in the future, the limit can be increased by changing a constant in the source files.

5.3 Integration with the Existing Tree

One of the most important parts of the implementation is the integration of the new tree with the existing directory index, which will be described in this section. While thinking of integration, it is necessary to keep in mind the backward and the forward compatibility. To keep the file system backwards compatible, we cannot remove any functionality, we can only add more. Additionally, while adding things, we must consider the forward compatibility of the older kernels, especially while changing the on-disk format.

Currently, all parts of the directory index, i.e., the index blocks along with the leaves are stored within the directory file using logical block numbers. The index nodes are formatted in a clever way as if they were empty directory entries, so a kernel without the support for directory indexing can still read the entries off the directory. The question is, how to fit a new tree here with only minimal changes to the structure of the existing index?

The first approach that comes to mind is to hide the blocks that belong to the new tree in the directory file, the same way the original tree is hidden. In this case, it would be necessary to hide all the blocks, not only the non-leaf nodes. Otherwise, the same directory entries would appear twice on a sequential traversal through the directory file. This approach seems appealing, for it keeps everything in one place on the disk. However, we would then need to have a way of differentiating between the blocks that belong to the hash tree and those of the inode tree. Apart from that, using logical block numbers means, that the file system has to keep a block map or an extent tree to resolve the mapping of the logical blocks to physical ones. This is not necessary, because the B^+ tree stores the pointers to blocks anyway. And especially with millions of files, it is better to avoid any unnecessary processing.

The second alternative is to store the new blocks for the inode tree outside of the directory file using meta-data blocks addressed directly by physical block numbers. This approach was proposed by Theodore Ts'o on the linux-ext4 mailing list [19] and it is the approach that I decided to use. It solves both of the problems described above. The directory file will stay unchanged, and because we are using the 64 bit physical block numbers to address the blocks in the tree, there is no need for any translation.

One problem with this approach is, that we will not be able to reuse the majority of the existing code, because it works with structures that cannot accommodate for block addresses twice as big. However, we need to use bigger key as well, due to the hard link collisions, so we would not be able to reuse much of the existing code without many substantial changes anyway.

5.3.1 On-disk Format Changes

Even though the tree will be stored completely outside of the directory file, a small change to the current format of the directory file is still required. The address of the root block of the inode tree (a 64 bit number) must be stored somewhere so the ext4 driver can find it.

The ideal place for storing the pointer would be the root block of the directory index, which is always the very first block of the directory file. To retain maximum compatibility, the address will be stored at the end of the block. We can use the fact, that there is a number that says, how many entries can be stored within this particular block. We can lower the *limit* of the root block and hide the root pointer behind the entries.

The same approach is used by the `metadata_csum` feature which stores the checksum in the tail space of the root block. The pointer will be added to the tail behind the checksum

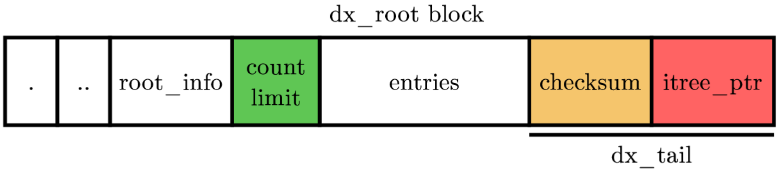


Figure 5.4: The placement of the pointer in the dx_root block. The whole structure is described in detail in section 2.4.1.

entry. The entries must be handled carefully, because none of the features is mandatory and one or the other entry can be missing. The file system driver must handle all of these cases correctly. The structure of the HTree root block with the itree pointer added is shown in Figure 5.4.

5.3.2 File System Operations

And finally, a modification some of the existing file system operations is required. We need to make sure that the entries are inserted, removed, and modified in the auxiliary inode tree at the same time as they are in the directory index. This can be achieved rather easily by adding an additional calls next to the existing ones that handle the itree. However, we need to pay a special attention to error handling in these cases, as an entry could be added to one tree and not to the second one due to a failure. This would lead to a diversion between the two trees that would require a file system check to resolve.

The tree will be initialized along with the hash tree when the size of the directory file exceeds a single block. This is done by the `make_indexed_dir()` function, so the itree initialisation code can be called from there. Apart from that, a call to either itree insert or itree delete will have to be added to the `ext4_dx_add_entry()`, `ext4_rmdir()`, `ext4_unlink()`, and `ext4_rename()` functions.

5.4 On-disk Structures

This section describes the structures that represent the nodes within the inode tree. As we already know from section 5.3, the nodes of the inode tree will be placed in blocks outside of the existing layout. That means, we are not limited in their design in any way in order to stay compatible with the current disk format. Unfortunately, we cannot reuse any structures that are used by the HTree, for both the key and the block pointers used there are only 32 bits long. The inode tree, on the other hand, requires 64 bits for each one. Nevertheless, I still tried to keep the design of the nodes as close as possible to the original tree, at least on the conceptual level.

There are only two types of nodes in the inode tree, the index nodes and the leaf nodes, as opposed to the hash tree, which works with three types. In this case, we do not need to keep any additional information about the tree in the root block, so the root is the same as any other index node. This is convenient, because we can make the number of levels in the tree easily extensible without the necessity of changing the code.

The structure of the tree is visualised in Figure 5.5, showing both the index nodes and the leaf nodes. Their properties are then discussed individually in the sections that follow.

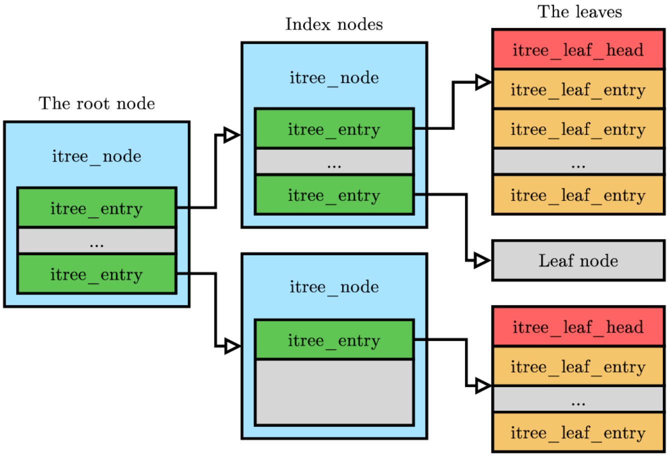


Figure 5.5: The structures that constitute the nodes of the itree.

5.4.1 Index Nodes

There are two structures used within each index node of the inode tree. The whole node is represented by **struct itree_node** and the entries that are a part of the node are represented by the **itree_entry** structure. The exact layout of both of the structures is shown in Figure 5.6.

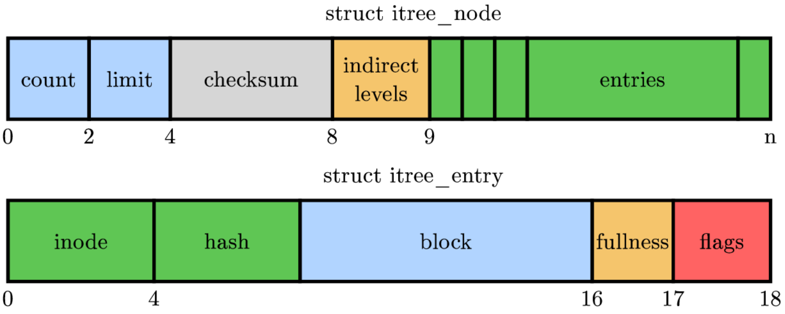


Figure 5.6: The layout of the `itree_node` and `itree_entry` structures.

The `itree_node` is very similar to the `dx_node` structure that is used in the hash tree. It contains the number of entries that currently reside within the node (`count`), the maximum number of entries that can fit into this particular node (`limit`). Apart from that, there is a field for a checksum and a single byte number, that is used to determine on which level of the tree does this node reside. This way, we can use the same nodes to point to both index nodes and leaf nodes as well. When the `indirect_levels` field is set to zero, the entries within this index node point directly to the leaves, otherwise they point to another level of index nodes within the tree.

The very last item is an array of the `itree_entry` structures, which stretches all the way to the end of the block. These structures carry the keys and pointers to other blocks. Apart from the `inode` and `hash` fields, that constitute the key, and the `block` field, that stores the pointer to the next block in the tree, there are two additional single byte numbers. The first one is to store various flags. However, there is only a single flag at the moment for which I found a use for during the development – `ITREE_NODE_FL_CONT` to indicate collision splits. There is a one bit unused in the hash part of the key, which could serve as this flag, so the flags field is very likely to be removed in the future, provided there will be no other flags required.

And the last field stores the *fullness* of the node, which is this entry pointing at. This value is very important while the tree is coalesced as files are removed from the directory. Without keeping this statistic here, we would have to read a multitude of additional blocks from the disk during the delete operation and walk through them to determine whether they can be merged with the current block. Doing this would be extremely inefficient.

5.4.2 Leaf Nodes

The leaf nodes are the place where the directory entries are stored. Again, the precise layout of this type of node is shown in Figure 5.7. In the inode tree, each leaf node is started with an instance of `struct itree_leaf_head`. The head is only two entries long. It contains the checksum for this node and also the number of bytes in this block that are used by the directory entries. Keeping this value here is convenient, because then we do not need to count it every time it is time to update the fullness of this node in the index.

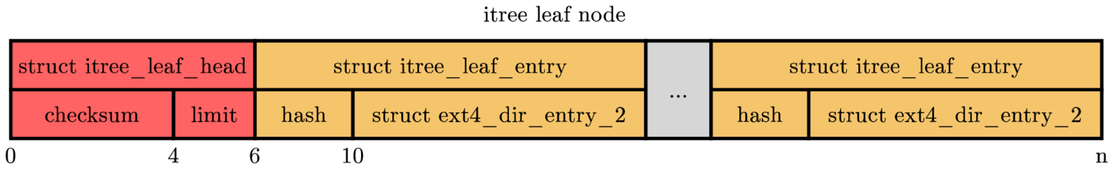


Figure 5.7: The layout of an itree leaf block.

The head is followed by a linked-list of directory entries. The entries in the inode tree are represented by a different structure than they are in the original tree. Unlike in the hash tree, the directory entries are kept sorted within the leaves of the inode tree. Each leaf entry is represented by an instance of the `itree_leaf_entry` structure, which is an ordinary directory entry, but it is accompanied by the hash value that is associated with this entry.

I decided to store the hash values with the directory entries in the leaves to optimise the processing of large collision sequences, where the ext4 driver would have to compute the values every time a new colliding entry were to be added to the sequence. This would be done for the whole sequence in case the entry would fit as the very last entry in it.

However, as was mentioned above, the inode collisions are very rare in the common file system workloads, so it might be a subject to a discussion in the future, whether it is necessary to do such optimisations. On the other hand, a dramatic decrease of performance in certain easy-to-reproduce cases might leave space for possible exploitation, as it could be used by users to intentionally paralyse the machine rather easily.

5.5 Tree Operations

Now is the time to describe the implementation of the operations that are able to search or modify the inode tree. The set of operations is almost identical to the actions that are performed on the HTree, as both of them are B^+ trees in nature. This section will focus on briefly describing the implementation of these operations for the inode tree, rather than their general properties.

5.5.1 Tree Traversal

The very basis of each operation is the traversal through the tree. If we would like to insert a new entry, we must search the tree for the right place to put it. The same applies on deletion. The entry must be found first, in order to be deleted. The tree traversal is implemented in a function called `itree_probe()`. The design of this function is very similar to its counterpart from the hash tree – `dx_probe()`. It starts with a key and a block pointer to the root and it returns the path through the tree, leading to the entry we're looking for. This function does not search through the leaves, so the result is in fact a block, in which the entry should be located. There are multiple reasons for this. The format of the leaf nodes is different, so representing the path would not be as simple. And in case the traversal is used for the insert operation, the entry we are looking for does not exist yet. This would again require some additional processing, therefore the last step in the traversal is simply left to the caller to do.

During the leaf searches, it is important to keep in mind, that there might have been a collision and the entry could as well reside in the following block. If the search fails, the collision flag of the next entry in the index must be checked and the search continued from there, in case the flag is set. This check is implemented in the `itree_next_frame()` function. For searching a leaf block there are two options with a slightly different semantics. The `scan_sorted_buf()` does the complex search through the leaf block, that is required while inserting an entry to the tree. The function used for deletes, `itree_search_leaf()`, does just a basic search, but it performs an additional verification of the result to make sure not to erase a wrong entry.

5.5.2 Inserting Entries

The insert operation is implemented by a function called `itree_add_entry()`. It starts with a search through the tree using the probing function we described earlier. Then the `scan_sorted_buf()` function is called to find the right spot in the leaf for the new entry. Provided there is enough space within the leaf node, the entry is stored in the right place by the `put_entry_to_sorted_buf()` function. In case there is not enough room available for the entry within this node, it will be split.

The split is quite a complicated procedure, because it can affect the whole tree. It has two parts, splitting the leaf node, which is handled directly in the `itree_add_entry()` function and inserting a new entry to the associated index node, which is taken care of in the `itree_node_insert_entry()` function. However, the index node may be full as well, so the algorithm might end up repeating the split procedure a few times over until it reaches the root. In case the root is full, and the tree is not yet of the maximum depth, a new level of the tree will be created. The split operation in case of the whole tree being full is illustrated in Figure 5.8.

5.5.3 Deleting Entries

Entry deletion is handled by the `itree_delete_entry()` function. The first part again, as in the insert operation, consists of a search through the tree. When the entry is found, it is not removed, but it is rather hidden by increasing the length of the previous entry. This makes the file system driver ignore the file later on and the free space may be allocated for a new entry in the future.

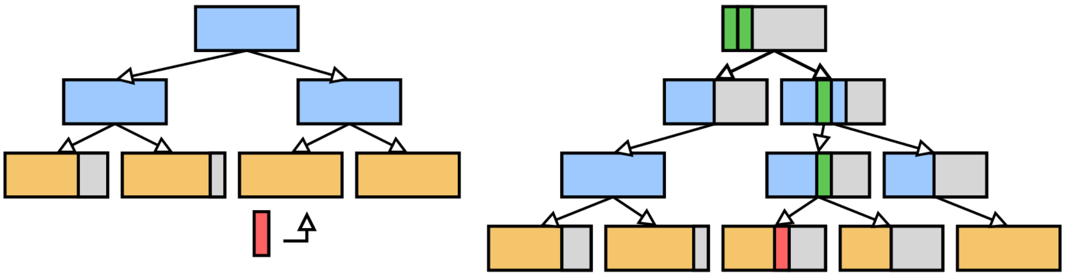


Figure 5.8: A node split in a full tree. The new leaf entry is pictured in red, the newly added index entries are green, and the grey spaces within nodes indicate free space.

When this is done, the delete operation continues with a check if the current leaf could be merged with any of its neighbours. This coalesce on delete functionality is very important, because the inode tree could get very easily fragmented during file moves. The kernel will look at the fullness of the neighbour blocks, which is stored in the associated index node and it will merge the leaf node with either of them, in case the entries from the neighbour block would fit within the leaf.

After the merge, the kernel must remove the entry that is associated with the block being freed from the index node. The procedure is very similar to node splits. The kernel goes through the index blocks of the tree, removes an entry and attempts to merge the index node with one of its neighbours as well. The root block is removed in case it only has a single child node left. The leaf nodes are merged by the `itree_do_delete_entry()` function. The index nodes are coalesced within the `itree_remove_from_index()` function, in case it is necessary.

5.5.4 The `getdents()` System Call

To actually receive the benefits from having an auxiliary tree with the directory entries ordered by the inode numbers, an additional implementation of the `getdents()` system call must be added to the file system driver, which will use the inode tree while reading the entries from disk. In the ext4 driver, this system call is implemented within the `ext4_readdir()` function. I added a new implementation, that can be found in the `ext4_itree_readdir()` function. This function is used instead of the original one in case the directory has the itree flag set.

The new implementation is in many ways similar to the previous one. It is a bit simpler though, because we do not have to sort the entries within each leaf any more. The position within the directory file is represented by the value of the last key read. The key is stored within the file's offset (`filp->f_pos`), in the `file` structure. This is crucial, because the directory might have changed between the calls to `getdents()` and the kernel must be able to restart the operation from the exact point where it previously stopped. We cannot keep any offsets, because the node we were working with might have been split and a half of the entries moved to a different block.

Because we can keep track of the entries only by their key, any collisions within the sequence are problematic. The kernel cannot differentiate between the colliding entries using the key. All the entries with the same value for the key must be then read at once, and they are cached in memory until the next call to `getdents()`.

Chapter 6

Evaluating the Implementation

The previous chapter introduced the design and the implementation of a new feature to the ext4's directory index. In the one that follows, we aim to assess this feature and evaluate, how it affects the whole file system. We will focus mainly on assessing the performance of the operations that manipulate directories. The same set of tests and benchmarks that we used to do the analysis in chapter 3 will be used to compare the new implementation to btrfs, XFS, and also to the upstream version of the ext4 file system.

6.1 Tests and Benchmarks

This section will describe the set of benchmarks that were performed to assess the implementation. Apart from that, a brief overview of the environment in which the tests were performed will be given, including the description of the hardware and also the software and the tools that were used. It is important to keep in mind, that the tests performed on a different system might yield very different outcomes, because the results depend heavily on a number of factors, for instance the number of storage devices, their speed, and partitioning. The tests can also be affected by the amount of available memory, which then determines the size of the page cache. Chapter 3 offers a full analysis of the conditions that can affect various directory operations.

The same set of tools and scripts that I developed during the first part of this project was used to conduct the testing (the design of the test suite itself was described in detail in section 3.3). The benchmarks included measuring of the performance of various directory operations, such as file creation, deletion, and, of course, directory traversal in various circumstances. The first two operations are not directly related to the case that we aim to optimise, but nevertheless, we must test how the new implementation affected them and make sure the implementation did not introduce any regressions to the file system.

To assess the directory traversal, three different test cases were used. The first benchmark measures the performance of copying a whole directory to a different location on a different physical device. This test represents a workload that is very common in practice, for example while storing snapshots or doing full backups. The other two test cases measure the performance only of an isolated directory traversal. They might not be as common in practical applications, but they are important so we can compare only the properties of the directory indexing code without any distortion, that can be added by different parts of the file system. Also the cache load for the isolated traversal is very much different from the load of the copy operation.

These few test cases were performed on a number of directories of different sizes ranging from 10,000 files all the way to 5,000,000 files per directory. The size of each individual file affects the cache pressure, so the tests were performed two times, once with empty files and again with each file exactly of 4kB of size. The last parameter of the tests was the fragmentation of the test directories. The first batch of tests was done on clean directories, which were populated by files created by a single process. In the second run a simple simulation of file system ageing was used while populating the directories with files.

6.1.1 Testing Environment

The hardware used to perform these tests was a desktop machine with an Intel Core 2 Duo E7600 processor running at 3.06GHz. The machine had 2GB of memory and three physical disk drives. One disk served as a system drive and the remaining two were used for testing. The disks were both 150GB Western Digital Raptors (WD1500ADFD) running at 10,000 RPM. Each one had only a single partition that stretched all the way across the disk. The second testing device served only as a scratch for the copy test. The XFS file system was used on the scratch device for all the tests. The operating system installed on this machine was Fedora 18 with the upstream 3.9.0-rc7 kernel built directly from git.

6.2 Results Summary and Discussion

With the tests and the testing environment explained, let us now proceed to the results of the tests. This section contains just a summary and a brief evaluation of a subset of the actual results, as there are way too many of them. The full results of all the tests I did are available on the media attached to this document. Each benchmark is available in two representations – graphical and text. The graphs visualise the differences between the individual file systems, while the tables contain the precise values that were measured during the tests.

The results of the order tests (described in section 3.3.2) show a clear improvement over the previous implementation. Now that the entries are read from the auxiliary tree, they are always returned perfectly in order. This is true for all the cases that were tested, even for aged and fragmented directories.

The different ordering of the entries lead to a great improvement in performance of the copy benchmark, which was more than 14 times faster in comparison to ext4 without the inode tree. Copying 5 million of empty files took previously 6 hours and 28 minutes to complete, while the new implementation does the very same task with a 6-hour difference just under 28 minutes. A very similar improvement was observed in the same test, but with 20GB of data (each file with 4kB of data), where the time required to copy the directory went down from 22 hours to only 2 hours, saving almost a whole day of computation. The results from this test case are shown in Figure 6.1.

The isolated directory traversal tests are no different in this aspect. For the directory size of 5 million, the traversal was almost 8 times faster for both the empty and the 4kB files. The improvement here is not as big as it was in the case of the copy test, because the cache load is much lower, as the directory is only read, we do not need to write the blocks to another file system. Therefore, the page cache is able to compensate for the random accesses to some extent.

In both benchmarks, the copy and the isolated directory traversal, the differences between the results grow with the number of files in the directory. This tendency is caused by

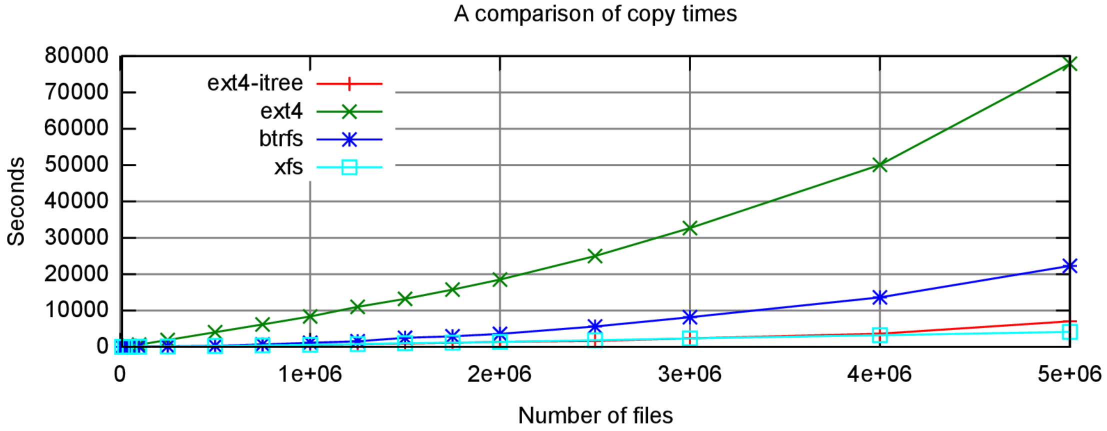


Figure 6.1: This graph shows the improvement in copy times. With the inode tree the ext4 performs as well as the XFS. Btrfs is much slower in this case and the upstream ext4 falls far behind. These values were measured on cleanly created directories with 4kB per file.

two reasons. As a directory grows, more inodes are allocated and even though the directory was created cleanly, the sequence is so big, that the first and the last inodes are very far from each other on disk. Accessing the inode tables randomly then leads to even longer seek times. Also the longer the sequence, the less effective the page cache is here, so the disk has to seek even more often.

Let us now have a look at the other operations – creating and deleting files from a directory and how the addition of the auxiliary tree affected them. The time required to populate a directory with 5 millions of empty files increased by 40% from 34 minutes to 47 minutes. Roughly the same percentual increase can be observed in case the files are of a 4kB of size. This difference again gets bigger as the number of file in the directory increases. This is caused by the increased size of the directory file, which leads to decrease in cache hits. A comparison of delete times of ext4 and XFS is displayed in Figure 6.2.

And finally, the delete times. A substantial decrease was observed up to the directory size of 1,500,000 files. In that case, the deleting the whole directory took 3 minutes and 25 seconds with the inode tree feature enabled. Without it, the operation run for 11 minutes and 50 seconds, being more than 3 times slower. Behind this point, the new implementation is still faster, but the differences are far less obvious as the directory size increases. However, the exact opposite tendency was expected. This is most likely to be caused by the implementation of the coalesce-on-delete algorithm, which is at the moment very aggressive. The nodes in the tree are probably merged too often, which results in more optimal utilisation of disk space, but it also leads to decrease in performance. This issue will certainly be looked into during the next development iteration.

6.2.1 Comparison to Other Linux File Systems

This section offers a brief comparison of ext4 with the itree feature enabled to two other Linux file systems – btrfs, and XFS. I picked these two, because they are among the most well-known. The XFS is used quite often for server deployments and btrfs is considered to be the next default Linux file system. In case of btrfs, it is important to keep in mind that it is still, at the time of writing, not stable and it is under a lot of development which is not yet focused on performance.

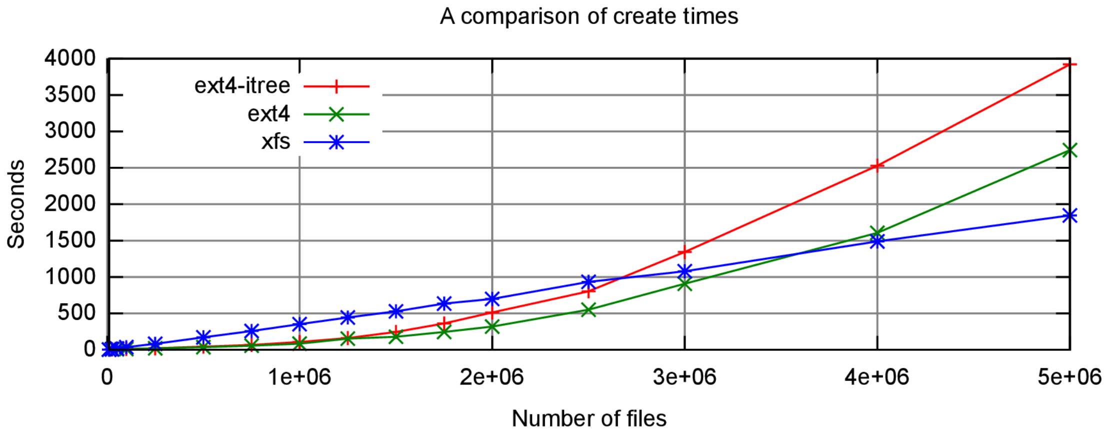


Figure 6.2: The create times of directories of different sizes are compared in this graph. Ext4 performs well both with and without the inode tree up to 2 million files. The create times tend to grow much less on the XFS after that point. Btrfs was left out from this comparison, as the results were completely out of this scale, worsening the readability of the rest of the graph. The values were measured on cleanly created directories with 4kB per file.

In the cases we tested, btrfs performs worse than both its stable opponents. The performance of ext4 during the copy benchmark was even worse, but now with the inode tree, it moved to the other end and it is even a slightly faster than XFS with empty files on a clean directory. It is a bit slower with 4kB files, but when the directory ages, the ext4 maintains its performance, while XFS suffers more from fragmentation. With 5 million empty files in a fragmented directory, ext4 was measured to be more than two times faster to perform the copy operation.

The results of isolated directory traversal are much the same as the copy benchmarks, the XFS being slightly faster, but when the directory is fragmented, ext4 becomes almost 10 times faster to walk the whole directory.

The XFS file system is much better in the file creation. Creating cleanly 5 million of files is more than 3 times faster on the XFS than on the ext4. However, while the directory is fragmented the times are approximately the same for both file systems. Btrfs is again far behind, being almost 4 times slower than both its opponents.

The delete results are again much better on the XFS. The difference is smaller for the fragmented tests, ext4 is actually faster there up to 2 million of files, but it gets approximately 1.5 times slower than XFS for 5 million files due to the above mentioned coalesce on delete algorithm.

6.2.2 The Final Discussion

Based on the results presented above, the implementation of the itree feature fixes the problem of the inefficient directory traversal rather well. However, as with many other optimisations it is a trade-off. An additional tree takes up more space on the disk and maintaining both of the trees takes more time. The insert operation is roughly 40% slower for mass creation of files. However, there still might be space for improvement.

During the development, I made a decision to keep the leaf nodes of the inode tree sorted, so they do not need to be sorted every time during a traversal through the directory.

Test Results for 250,000 Files				
Operation	btrfs	ext4	ext4-itree	XFS
Create	26.8	21.6	20.3	82.7
Delete	53.1	13.1	9.6	43.3
Copying	135.4	1911.5	96.8	110.7
Isolated Traversal	22.8	6.7	4.9	10.1

Table 6.1: A comparison of the durations of a few directory operations for a directory of 250,000 files. All the values within the table are in seconds.

However, that may require more processing during file addition in certain cases. Not sorting the leaf nodes could speed up the insert operation. The directory traversal should not be slowed down very much by the necessity to sort the leaves, as it is already fast enough in comparison to the previous state of affairs. However, it would require substantial changes in the implementation, so further testing and investigation is necessary, before we proceed with such a step.

The performance of file deletion has improved, but not as much as was hoped for. This is most likely caused by too aggressive coalescing of the tree nodes. During the next stage of development, the coalesce-on-delete algorithm must be fine-tuned to provide a good compromise between the fragmentation of the tree and the performance burden it imposes on the delete operation.

Overall, the current implementation of the inode tree works best for directory sizes up to 1.5 million files. For instance, virtually no decrease of performance of the create operation was observed up to 250,000 4kB files, while the copying was almost 20 times faster in this case (the precise results of this particular test case are shown in Table 6.1). The deletion time was also around 40% faster. With more than 2 million files, the directory traversal times are still excellent, but the create and delete operations seem to suffer more from the extra processing that is required to manage the additional tree. Nevertheless, exchanging 40% decrease of file creation for 14 times faster directory traversal is more than a viable trade-off for many workloads. The situation might get even better in the future when the coalesce on delete algorithm is optimised.

Conclusion

The goal of this project was to analyse the current implementation of the directory index in ext4, to find out the reasons for the enormous drop of performance that had been observed during sequential directory traversals, and ultimately to implement a better solution.

I started off by studying the internals of the Linux kernel. I explored the subsystems associated with file systems, such as the virtual file system layer, the block I/O layer, buffer and page caches, and the ways these parts cooperate with each other. Understanding of the principles behind those subsystems was crucial for further comprehension of ext4 and its codebase.

After that, I focused on the implementation of the ext4 file system itself. Its on-disk layout as well as the algorithms used to store files and directories on disk. During this part, I developed an understanding of the way ext4 stores and manipulates directory files reasonable enough to be able to work on the issue.

An analysis of various directory operations was carried out next. During this phase of the project I developed a series of test cases to for evaluating and benchmarking the performance of directory operations. This testing suite is now publicly available on Github[1] under the GNU GPL v3 license. The experimentation and further testing identified the weak spots in the current implementation. A comparison of ext4 to btrfs and XFS shown to which extent the issue affects each operation.

Following the analysis, I researched the existing proposals to solve this problem and worked on one of my own. The different approaches were evaluated and a decision was made to implement an auxiliary tree to the ext4's directory index, an approach that has, to my knowledge, never been tried before in the ext4 file system, as opposed to the other solutions.

A new feature adding the auxiliary tree called *itree* was implemented to the ext4 file system. It can be enabled optionally at the file system's creation time, in case the administrator expects to work with large directories. The patches with the implementation of this feature were posted to the ext4 development mailing list[2].

Later on, the implementation was tested and evaluated using the same set of benchmarks that were developed during the analysis. Based on the test I did, the directory traversal is roughly 14 times faster than before with the itree feature enabled. Copying 5,000,000 files, an operation that took previously more than 6 hours to complete is now done in 28 minutes. This is for the cost of an increased size of the directory file and approximately a 40% increase of file creation times, as the file system now has two trees to maintain. This optimisation is, as many others, a trade-off, however, a viable one in many situations and workloads (such as doing full backups, processing large data sets, mail server workloads, and more).

[1] https://github.com/astro-/dir-index-test

[2] http://marc.info/?l=linux-ext4&m=136770326315215

There are several areas of this project where further work is planned. One of them is the insert operation to the inode tree. The leaf nodes of the inode tree are kept sorted, which could, in some cases, be less efficient than sorting the entries later on during the traversal. A series of benchmarks could be made to analyse the insertions to see whether the sorting is viable. A similar series of tests will be necessary for the delete operation to find the most suitable configuration for the coalesce-on-delete routines.

The last downside of this solution is the increased size of the directory meta-data, because the entries must reside in both trees at the same time. A further research could be made to the possibilities of storing both indexes outside of the directory file and using pointers into a single sequence of entries shared by both the trees.

And ultimately, as it is with any new feature, the code must be reviewed, tested, debugged, and refined again, until it meets the requirements for production, especially in the environment as critical as data storage.

Bibliography

[1] Jacek Luczak. difrost.kernel@gmail.com. getdents - ext4 vs btrfs performance. Feb 29, 2012. Linux Kernel Mailing List Archives: linux-ext4@vger.kernel.org (Aug 25, 2012).

[2] Phillip Susi. psusi@cfl.rr.com. Large directories and poor order correlation. Mar 14, 2011. Linux Kernel Mailing List Archives: linux-ext4@vger.kernel.org (Aug 25, 2012).

[3] Robert Love. *Linux Kernel Development, Third Edition*. Addison-Wesley Professional, 2010.

[4] Wolfgang Mauerer. *Professional Linux Kernel Architecture*. Wrox Press Ltd., Birmingham, UK, 2008.

[5] Claudia Salzberg Rodriguez, Gordon Fischer, and Steven Smolski. *The Linux(R) Kernel Primer: A Top-Down Approach for x86 and PowerPC Architectures*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[6] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc., 2005.

[7] Daniel Bovet and Marco Cesati. *Understanding The Linux Kernel, Third Edition*. O'Reilly Media, 2005.

[8] Lukáš Jelínek. *Jádro systému Linux: Kompletní průvodce programátora*. Computer Press, a. s., Holandská 8, 639 00 Brno, 2008.

[9] Theodore T'so. Ext4. Presented at *The Free and Open source Software Developers' European Meeting* (FOSDEM), Brussels, 2009.

[10] Darrick J. Wong. Ext4 disk layout. Aug 3, 2012. https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout (Aug 25, 2012).

[11] A. Mathor, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux Symposium*, pages 21–33, Ottawa, ON, Canada, 2007.

[12] D. Phillips. A directory index for ext2. In *Proceedings of the Linux Symposium*, pages 426–438, Ottawa, ON, Canada, 2002.

[13] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The art of computer programming*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, second edition, 1998.

[14] Douglas Comer. Ubiquitous B-Tree. *ACM Comput. Surv.*, 11(2):121–137, June 1979.

[15] Mingming Cao, Theodore Y. Ts'o, Badari Pulavarty, Suparna Bhattacharya, Andreas Dilger, and Alex Tomas. State of the art: Where we are with the ext3 filesystem. In *In Proceedings of the Ottawa Linux Symposium*, pages 69–96, 2005.

[16] Daniel Phillips. phillips@arcor.de. [RFC] Improved inode number allocation for HTree. (Mar 10, 2003). Linux Kernel Mailing List Archives: linux-ext4@vger.kernel.org (Nov 18, 2012).

[17] Coly Li. coyli@suse.de. [RFC] Designing and Implementation of Directory Inode Reservation. (Mar 27, 2007). Linux Kernel Mailing List Archives: linux-ext4@vger.kernel.org (Jan 05, 2013).

[18] Coly Li. coyli@suse.de. [PATCH] ext4: dir inode reservation V3. (Nov 13, 2007). Linux Kernel Mailing List Archives: linux-ext4@vger.kernel.org (Jan 05, 2013).

[19] Theodore Ts'o. tytso@mit.edu. Re: [RFC] Optimizing readdir(). (Jan 14, 2013). Linux Kernel Mailing List Archives: linux-ext4@vger.kernel.org (Jan 25, 2013).

Contents of the Attached CD

- `/dir-index-test/` – test cases and scripts
- `/ext4-tests/` – detailed results of the tests
- `/ext4-tests/README` – description of the test results
- `/patches/kernel/` – the itree patch sets for Linux 3.9 and 3.10-rc1, full sources of both kernels included
- `/patches/e2fsprogs/` – patches adding the itree feature to the ext4 user-space utilities, full sources of e2fsprogs 1.42.7 included
- `/xpazde00-ext4.pdf` – the pdf version of this document