Die approbierte Originalversion dieser Diplom-/Masterarbeit ist an der Hauptbibliothek der Technischen Universität Wien aufgestellt (http://www.ub.tuwien.ac.at).

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology (http://www.ub.tuwien.ac.at/englweb/).

TECHNISCHE UNIVERSITÄT WIEN

VIENNA UNIVERSITY OF TECHNOLOGY

DIPLOMARBEIT

LLFS
A Copy-On-Write File System For Linux

ausgeführt am Institut für Computersprachen
Abteilung für Programmiersprachen und Übersetzerbau
der Technischen Universität Wien

unter Anleitung von
Ao.Univ.Prof. Anton Ertl

durch

Rastislav Levrinc
Kaltenbäckgasse 3/4
1140 Wien, Austria

Wien am May 6, 2008

Zusammenfassung

Diese Diplomarbeit beschreibt das Design und Implementation von LLFS, einem Linux-Dateisystem. LLFS kombiniert Clustering mit Copy-On-Write. Copy-On-Write überschreibt belegte Blöcke nicht zwischen Commits; Durch das Clustering bleibt die LLFS-Geschwindigkeit vergleichbar mit Clustered-Dateisystemen wie Ext2. Copy-On-Write ermöglicht neue Features wie zum Beispiel Snapshots, beschreibbare Snapshots (Clones) und schnelle Crash-Recovery zu einem konsistenten Dateisystem-Zustand. Gleichzeitig hilft das Clustering, die Fragmentierung niedrig und Geschwindigkeit hoch zu halten.

Clustering wird mit Ext2-ähnlichen Gruppen und Free-Blocks-Bitmaps für das Belegen und Freigeben von Blöcken erreicht. Journaling Dateisysteme wie Ext3 brauchen ein Journal und schreiben Blöcke doppelt. Mit Hilfe von Copy-on-Write vermeidet LLFS diese Kosten. Aufgrund der Free-Blocks-Bitmaps braucht LLFS keinen Cleaner wie Log-Strukturierte Dateisysteme. Trotzdem bietet LLFS die kombinierte Funktionalität von Journaling und Log-Strukturierten Dateisystemen.

Ich habe LLFS aufbauend auf Ext2 implementiert, und ich habe die Performance getestet. Die Benchmarks zeigen, dass LLFS ähnliche und in einigen Fällen bessere Resultate als Linux-Journaling-Dateisysteme erreicht.

Abstract

This thesis discusses the design and implementation of LLFS, a Linux file system. LLFS combines clustering with copy-on-write. With copy-on-write no allocated blocks are overwritten between commits, and thanks to the clustering the speed of LLFS remains comparable with clustered file systems such as Ext2. Copy-on-write opens new possibilities for features like snapshots, writable snapshots (clones) and fast crash recovery to the consistent state of the file system, while the clustering helps to keep fragmentation low and speed high.

Clustered reads and writes are achieved with Ext2-like groups and free-blocks bitmaps for allocating and freeing of blocks. Journaling file systems like Ext3 need to keep a journal and write blocks twice; By using copy-on-write, LLFS avoids these overheads. By using free-blocks bitmaps, it does not need a cleaner like log-structured file systems. Yet LLFS offers the combined functionality of journaling and log-structured file systems.

I have implemented LLFS starting from the Ext2 file system and tested the performance. The benchmarks have shown that LLFS achieves similar performance and in some cases better than Linux journaling file systems.

Contents

Part I

Design

Chapter 1

File Systems

A file system is a part of an operating system that takes care of storing and reading data on the storage device such as a hard drive or CD-ROM. From a user perspective, data are organized as a collection of files. Files are not only text files, but also images, executable programs and so on. Another important abstraction in a file system is a directory. A directory can hold not only files but yet another directory called a sub directory that allows to organize files in a tree structure. Thanks to this it is possible to create a hierarchy of directories containing related files. A file system also provides means to create, move and delete files and directories, change permission who can read or modify these files and provides information about a file such as its length and creation time.

Another task for a file system is to optimize reading and writing of files. Data are stored on a disk in units of blocks, and it is desirable that blocks belonging to one file and some other related blocks are not scattered around the disk, because reading of adjacent blocks is much faster than reading blocks on different parts of the disk divided by holes. Skipping the holes involves seek times, and the seek times are bad for performance with the current hard drive technology. A typical seek time on a normal hard disk is several milliseconds and is to be avoided as much as possible.

1.1 File System Design Issues

There are many issues that a file system designer has to take into a consideration. I list the most important of them in this section. Although file systems improved much over the years and many issues are solved, there are still open questions about data consistency and trade-offs that a file system designer must make.

1.1.1 Fast Crash Recovery

A computer can crash because of faulty hardware, or a mistake in an operating system code. The computer can also come down by sudden loss of power or can be switched off by mistake. This sudden interruption of the operation of a running computer is a particular problem for file systems because they can end up loosing some data or become flat out unusable.

This inconsistency issue is caused by the fact that file systems heavily use caches to immensely speed up a performance of writing. The blocks are gathered in the main memory of a computer, where they can be reordered and can be written out sequentially or at least more sequentially than they would have been if they would be written out one by one. On the other hand, if some file system operations consist of multiple steps, the steps can be written out out of order, some of the blocks can be already written to the disk and some were only in the memory, waiting to be written and are lost.

After a sudden interruption when some of the blocks were not written, a disk partition can contain data that do not belong to any file, and even worse, files can contain wrong data.

Other inconsistencies include changed directory entries that point to files that do not exist and vice versa.

Before journaling was introduced, such mishap was resolved by running fsck (file system check) utility. This utility checked and repaired the whole disk partition if possible[1]. This was time consuming task and with the ever increasing sizes of disks it becomes unacceptable for availability of servers for example. Traditional file systems, among them Ext2, need to execute whole structural verification after a system failure. Current file systems try to avoid this task.

1.1.2 Data Consistency

Fast crash recovery is nowadays standard in the Linux file systems. What is not clear is file system data consistency after a crash or a power failure. Usually only a meta-data consistency is guaranteed. That means that directory structure is recovered, but the data may be lost. This can be disastrous for applications especially if they require data consistency between files.

Ideally all the data that are written end up on the disk. The best way to achieve this would be to write all the data synchronously without using a write cache. This would also be a very slow way.

Some Linux file systems offer data consistency. They achieve it with journaling of the data. This beats the synchronous writes but it is still way

[1]Sometimes repairing of file system is impossible.

too slow, because the data have to be written twice, once to the journal and once to their proper place.

Data consistency that will be dealt with in this work, uses in-order semantics and refers to a state of a file system after a recovery where not only directory structure, but also files contain all the data that were written before a specific point in time, and no writes and other changes that occurred afterwards [Cze00].

Currently only copy-on-write file systems with in-order semantics can potentially offer data consistency with acceptable performance.

1.1.3 Undo

To undo changes in file systems would be nifty feature that most Linux file systems do not offer. In today's Linux file systems, retrieving a removed file is sometimes possible, so long it is not overwritten by some other data. If data in a file are changed the old data are lost. Multiple undo could help to recover specific version of the data as they were changed over and over.

Some other solutions exist, but they exist in the user space and are not a part of a file system.

1.1.4 Consistent Backups

Backup is an activity to copy important data to another place, the more remote the better, where in case of need all of the data or some of them can be copied back. The place where the backup is stored can be any kind of a storage device, for example a magnetic tape or hard drive in another computer.

Making of a backup can take a long time and during this time data on the disk can change, so the data reflect the state as they were in different points in time. This can cause some of the backups to be unusable. For example databases cannot be backed up just by copying the files, while the database is used. Although databases use their own methods to ensure consistent backup, it would be nicer if the underlying file system could do this and it would not matter which application is using the data.

Creating a snapshot of the file system at one exact point in time and backup the data from the snapshot, while the file system is used without affecting the snapshot, would effectively solve this problem.

1.1.5 Fragmentation

Although a hard drive is a block device with random access, it can access blocks in any part of disk in any order, in reality the hard drives perform best if data are read and written sequentially. The task of the file system is to store data that are likely to be accessed at the same time next to each other as much as possible. This speeds up writing as well as reading. Data are likely to be accessed at the same time, either if they belong together logically, for example if they are in the same directory, or when they are modified at approximately the same time, it is more likely they will be accessed at the same time in future.

To avoid fragmentation is easy if the file system is almost empty, but over time as files are created and removed, it is increasingly difficult to find large free regions that can hold the whole file and seek time and rotational delay of the read/write head will deteriorate the over-all performance of the system.

Some file systems solve the fragmentation problem with defragmentation utilities that have to be run for time to time for example once a day, but a modern file system should try to minimize the fragmentation during writing of the data.

This is not the only kind of fragmentation that is relevant to the file system design. This kind of fragmentation is called an external fragmentation. Another kind of fragmentation is internal fragmentation and it is about an empty space between an end of a file and a block boundary. This is called internal fragmentation. This fragmentation is getting bigger with bigger block sizes. This is more a wasted disk space than a performance problem. A performance impact can be possibly noticed only on a system consisting of many small files. Solutions to the internal fragmentation reduce wasting of the disk space, but add yet more computational overhead.

1.1.6 Scalability

The file system is one among many systems in a computer that should be scalable. In the case of file system the scalability is understood as ability of the file system to handle ever increasing sizes of disks, bigger files and number of directory entries.

Some file systems have hard limits that cannot be overcome, some will hit a performance bottleneck sooner or later and are no more usable. A file system should be designed, so that even today unimaginable capacities and sizes of files are possible.

1.2 Clustering

Clustering of reads and writes is a solution for decreasing the disks seeks between adjacent blocks in a file, thus decreasing the overall fragmentation of the system. Once the blocks that are likely to be accessed at the same time are stored in one cluster or neighboring clusters, the read and write performance can increase significantly. File systems using clustering are FFS and in the Linux world the Ext2 file system.

The clustering in the Ext2 file system is achieved with dividing the disk into block groups, where related data and meta-data are allocated in one block group or some block group nearby. During allocation procedure the file system detects sequential writes and files and meta-data are written sequentially on a disk if possible. That way the data that are likely to be read at the same time are stored next to each other.

Although the external fragmentation degrades somewhat the performance of such file system, the research showed that active FFS file systems function at approximately 85–86% of their maximum performance after two to three years[Sel95].

The clustering does not solve all the problems though. The consistency after a system failure is normally not guaranteed in a file system without a journaling and file system check is required. Although this need could be removed by synchronous meta-data writes, there is a huge performance penalty. Journaling is described in the next section. There is one more solution called soft-updates that marks order of meta-data updates and syncs them to a disk in that order. Performance of soft-updates enabled file system in some cases when deletes are delayed is better than that of the journaling file systems, but in other cases the performance suffers up to 50% degradation[Sel00].

1.3 Journaling File Systems

The most popular file systems on Linux are the journaling file systems. Journaling file systems use database transaction and recover technologies to solve the inconsistency problem after a system crash or power failure.

Journaling file systems keep a journal of file system changes in order to avoid the time-consuming task of the full file system check. Journaling file systems need to replay changes from the journal when recovery is needed. The journaling file systems keep a journal of disk changes in a reserved space on disk. The journal is written before the actual changes are made on the disk. After a system failure the journal is analyzed and the disk is brought to a consistent state. Scanning of the whole disk is not required anymore and not surprisingly the journaling file systems took over on production servers and elsewhere.

Oddly enough, file system research did not stop here, because there is a problem. Journaling file systems need to write data blocks twice, once in the log and once to their place on the disk. This full journaling is a big performance hit. That is the reason why the journaling file systems normally log only meta-data changes and disable data-logging or do not implement it at all.

Log-structured file systems, as well as LLFS, on the other hand, with different approach, ensure a full data consistency without writing the data blocks twice.

1.4 Log-Structured File Systems

The central principle behind log-structured file systems is to perform all writes sequentially, thus increasing the write performance. The no in-place updates allow for the fast crash recovery and data consistency. Log-structured file systems can get to the point of the last check point and get to the consistent point. From that point a roll forward can be performed to save some of the data that were written after the checkpoint.

The Log-Structured File System (LFS) was introduced in 1991 and was available for comparison.

Early research on the log-structured file systems promised order-of-magnitude improvement of performance and for small files allowed LFS to write at an effective bandwidth of 62 to 83% of the maximum[Ros91]. Later research showed that high hopes for the log-structured file system were not realized.

For full utilization of the disk's bandwidth, a log-structured file system needs to maintain large free areas on the disk. For that a garbage collector is needed called cleaner that collects small free areas into the large ones. Paper comparing FFS with LFS by Seltzer et al.[Sel93] showed that cleaning overhead degraded transaction processing performance by as much as 40%.

Further research by Seltzer et al. [Sel00] comparing LFS and FFS showed that even ignoring the cleaner overhead, the order-of-magnitude improvement in performance claimed for LFS applies only to meta-data intensive activities, specifically the creation and deletion of small files. For large files the performance was comparable with clustering file systems. Cleaner overhead reduced LFS performance by more than 33% when the disk is 50% full. LLFS is in a way a log-structured file system that does not need the cleaner, but uses clustering to group related data and meta-data together.

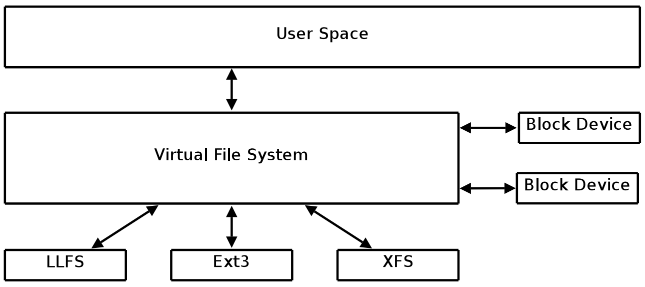

Figure 1.1: Virtual File System

1.5 Linux Virtual File System

Linux file systems are all implemented or ported on top of the VFS (Virtual File System). VFS is a layer that takes care of interoperability between different file systems themselves and user applications. Consequently the user applications talk to the virtual file system that hides the specific file system implementation. VFS provides a directory entry and an inode cache of last used files and directories.

VFS is also an interface between file system and the lower block level of the kernel. Thanks to this, file system reads and writes data to the buffer cache or page cache and does not have to care how the underlying media looks like. Figure 1.1 shows how a user space application writes and reads data to different file systems.

1.5.1 VFS Data Structures

VFS contains data structures that describe a common file system. They range from data structures that describe the file system as a whole to the data structures that describe every file. These data structures contain data and function pointers that can be redefined by any file system or the file system can use function implementations from VFS. This is in a way a kind of object programing in C. Furthermore these data structures can be extended by every file system with their own member variables. In Linux as well as in Unix the file system is organized into two distinct subsystems. Names of directories, their hierarchy and file names are stored independently of inodes that represent files, their sizes, permissions and data.

Super Block

The super block holds information about one specific instance of a file system. Normally a file system stores its super block on a special place on a disk partition where it can be read during mounting. It also includes pointers to functions that read, write, remove, allocate inodes and so on.

Inodes

One of the most important structures in VFS is an inode. One inode corresponds to one file in the file system[2].

Linux file systems normally have their own representation of an inode that corresponds to the VFS inode. File systems that do not represent files and directories through the inodes, have to assemble inodes in memory, so that they can work with VFS. Inodes contain pointers to the data blocks, access permissions, owner, type of the file and so on. Every inode is identified by a unique inode number.

Directory Entries

Directories are files that contain a list of file names and names of sub directories with a corresponding inode number. When a user opens a file, its inode has to be determined. It can be found either in a cache or in the parent directory file with directory entries with inode numbers have to be read. The parent directory is again obtained from its parent directory or the cache if it is there. So it goes recursively to the root directory if necessary. The root directory is always in the cache. If some entry is not in the cache, it stored there in order to speed up the next look ups of the same file or files in same or nearby directories.

VFS assumes that there is only one root directory per file system, which is not true for LLFS.

Other Data Structures

There are several other data structures associated with VFS. The File structure represents an open file, its attributes like permissions and position in a file and pointers to functions that perform operations like open, seek, read and write. This data structure is the most familiar for users of the file system. It also contains a link to a directory entry with resolved name.

[2]File is meant here in the broad definition of the word *file*. In this context file can be directory, symbolic link, named pipe or an ordinary file with data. Remember that everything in Unix is a file.

The file_system_type data structure describes a file system, its name and type. There is only on such structure per file system.

When a file system is mounted vfs_mount data structure is populated. It describes a mount point. This structure stores for example options with which the file system was mounted and the directory entry of the mount point.

The files_struct and namespace data structures map every process with its open files, current working directory, and so on.

Chapter 2

LLFS Basic Idea

2.1 Introduction

There are two types of recently popular approaches to the file system design – journaling and log-structured file systems. Log-structured file systems offer a data consistency but require a garbage collector called cleaner that gathers allocated blocks together if there are holes between them. Log-structured file system do not perform very well because of the cleaner overhead.

Journaling file systems do not offer data consistency in any efficient way unless the data blocks are written twice.

Traditional file systems like Ext2 are still kept around with their good performance with clustered reads and writes.

LLFS's idea is to combine clustering and the part from log-structured file systems where blocks are not overwritten right away, but doing away with the idea of one never-ending sequential log. That way LLFS is a file system that makes use of clustering to achieve good performance, but still offers features of log-structured file systems like the fast crash recovery and data consistency.

The key feature of LLFS is no in-place writes or copy-on-write. If data in a file are modified, the blocks that were affected are not written to the same place where they had been before, but a new place on a disk is allocated for them. The previous location of this blocks is not freed until the block is committed.

A consequence of this is that blocks do not get overwritten, not until they are committed or optionally not even after that. This is used for snapshots and clones.

When a clone or snapshot is made, its blocks should not be overwritten, even if they were freed in the clone from which the clone or snapshot was made. When the clone or snapshot is destroyed, the blocks should be made available again.

The last committed state can be seen as an automatic snapshot that can be (and is) recovered after a system failure. This has an advantage that directories, data, and meta-data are always consistent between commits.

For allocation of blocks LLFS uses Ext2-like allocation, where blocks are clustered to the groups with whatever algorithm Ext2 is using.

2.2 LLFS Requirements / Goals

Several requirements and goals were identified for the new file system. Some of them like fast crash recovery are common by today's file systems, some of them like clones and snapshot functionality are just emerging and are either not efficient or not completely implemented.

2.2.1 Fast Crash Recovery / File System Check

LLFS should implement an instantaneous crash recovery to the state that the file system was after the last commit, any committed snapshot or clone. This is similar to the log-structured file system crash recovery. It should be possible to make a complete file system check in a background, on one clone, while some other clone is mounted.

2.2.2 Data consistency

LLFS should implement the in-order semantics that guarantees that after a recovery the file system represents the state of all files and directories as they were in one specific point in time.

Most journaling file systems today do not give in-order semantics, they give only meta-data consistency or there is a performance penalty, as all data has to be written twice. If only changes to a directory structure are logged, directories are preserved, but the data may be replaced by garbage.

2.2.3 Snapshots

A snapshot of a file system is a state of the file system as it was in one specific point in time.

One of the uses of the snapshot is for enabling of consistent backups. A backup can get inconsistent if during the backup the file system is used. In

LLFS a snapshot can be taken instantaneously and the file system can be used after taking the snapshot without affecting it.

Another use of the snapshot is that the snapshot is kind of easy backup that allows for retrieving of removed or changed data. Taking of snapshots can be set up in this way that file system could do multiple level undoes. Especially if taking of snapshots does not affect performance of the system.

Although LVM (Logical Volume Manager) offers snapshot functionality, there is a performance and space penalty. When a block is written, LVM copies the blocks in a for this reason allocated area. Because of this a block must be written twice, which is time consuming and there must be the allocated area on the disk that can not be used by the file system.

2.2.4 Clones

A clone is a snapshot that can be written to or another way to view a snapshot is as read-only clone. A clone should be created instantaneously just like a snapshot. It can be mounted at the same time as its parent and it can be used and then discarded or kept. This can be useful in many ways. Software can be installed and tested on a clone during the production and then this clone can be switched to the production or it can be discarded if it went wrong.

A goal of LLFS is to provide efficient creation of snapshots and clones without copying of blocks. A new clone starts with the same blocks as the cloned file system and only with time as the data change the copies of blocks will be created. Destroying the clones in LLFS should be also cheap.

2.2.5 Performance

The requirement for LLFS in terms of performance is to stay competitive with file systems like Ext2 and Ext3 in a typical operation.

LLFS uses Ext2-like allocation policies that are tuned to perform very well in comparison with log-structured file systems.

2.2.6 Fragmentation

The goal for fragmentation is to keep it low. LLFS can create an additional fragmentation, because when parts of a file are modified, they do not remain adjacent to the other blocks of the same file, but are copied some place else[1], unlike the file systems that modify the blocks in place. File systems like

[1]But still there is effort to put these blocks nearby if possible

Ext2 do not experience much of a fragmentation, so hope is that additional fragmentation in LLFS will be not so tragic. On the positive side of things, LLFS does not have predetermined positions of block bitmaps, inodes and group descriptors and they generally they could be allocated nearer to the data blocks than it is in case in the Ext2 file system and that could reduce fragmentation somewhat.

2.2.7 Scalability

LLFS should be scalable. Hard disk sizes will continue to increase in the foreseeable future as they have been doing till now. LLFS should scale with bigger sizes of disks and should be able to store any number of files and directories with any sizes that are reasonable in the considerable future.

2.2.8 Portability

LLFS should be portable. LLFS should work on wide range of computer architectures like other Linux file systems and the on-disk structure should be portable between different architectures.

2.3 Implementation of these Goals

2.3.1 Fast Crash Recovery

Fast crash recovery in LLFS is part of the design. The last consistent state of every clone is kept on the disk as a snapshot. After a crash these snapshots will be mounted and possibly inconsistent partly written clones discarded. This is really fast, faster than replaying of logs.

2.3.2 Data Consistency

LLFS implements in-order semantics. No blocks with exception of super block are written to the same place. When the file system is committed, the committed blocks are not overwritten. This last committed state can be recovered after a system crash or power failure and this recovered state represents a point-in-time data consistency.

2.3.3 Snapshots / Clones

In LLFS there is no difference between snapshots and clones, only that clones can be called snapshots if they are mounted read-only.

There is only one pointer that is needed for file system to know that there is a clone or snapshot in use. This pointer points to all the meta-data, starting with inodes to group descriptors and free block bitmaps. When a clone or snapshot is discarded, it is enough to overwrite this pointer.

From point of view of other clones, while they are being written to, they, use the pointer from this clone to read this clone block bitmap to not overwrite its blocks. The more clones there are, the more bitmaps have to be read by allocating a block in a group. Discarding of the clone is to remove the pointer to this clone's meta-data. From this point on, other clones can allocate their blocks where the discarded clone used to be.

It is also possible to clone a previously created clone or make more clones from one clone. This creates a kind of tree as seen in figure 2.1, where every clone except the clone 0 has exactly one parent clone a can have more child clones. At the same time any clone can be discarded, so the tree structure is broken.

The figure 2.1 shows making a clone from the master *clone 0* where two clones *clone 1* and *2* were created. The same way *clone 3* and *4* were created from *clone 2*. After that although *clone 2* was destroyed, *clone 3* and *4* can be normally used. *Clone 0* is not special in anyway, only that when file system is first created, it is created as *clone 0*. When *clone 0* is cloned, *clone 0* can be removed, and can be overwritten with another clone and so on.

It is possible to create a clone over already existing clone, which equals destroying the clone and creating a new one on its place. The way it is implemented, it is also possible in this way to exchange a clone while it is used. Although this seems interesting and may have some uses, I cannot think of any and no Linux application expects this behavior from file system and would hopelessly break.

2.3.4 Performance

LLFS is designed not to perform much worse than the Ext2 file system. With increasing number of clones the write performance is decreasing because every clone has its own bitmap blocks and they have to be read for every clone. Read and write performance is also influenced by a bit more fragmentation than it is in the Ext2 file system.

2.3.5 Scalability

While working on LLFS, I did not focus on scalability much. The LLFS scales with bigger block sizes and indirection, but further work should be done in this area.

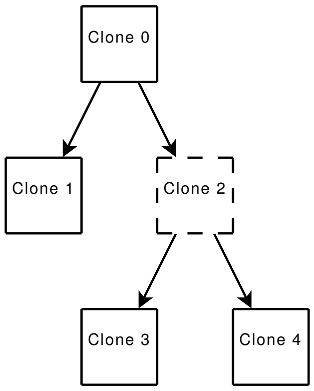

Figure 2.1: Clones

2.3.6 Portability

LLFS inherited its portability from the Ext2 file system. Portability issues come down to different sizes of integer types and different byte order on some architectures. Using explicitly-sized data with fixed byte order for on-disk structures solve this problem. LLFS is portable, but since it was tested only on 386 architecture, some easy to fix errors can be still in there.

Chapter 3

LLFS Design

3.1 Inodes and Free Blocks Bitmaps

The Ext2 file system keeps its inodes, free inodes bitmap and block bitmap in the same locations on the disk. This is not so with LLFS, because no blocks except of super block are written on the same place.

To find an inode in the Ext2 file system is enough to know the inode number and the location of the inode is computed and the block is found that contains the inode. There are several inodes stored in one block. For example in 4096 byte block there are 16 inodes stored next to each other. In LLFS the inodes are stored in blocks in the same way but the blocks containing inodes are not stored continuously in a predefined place but are stored all around the disk. In order to find them, a structure is needed that contains pointers to the blocks with inodes, in other words it maps an inode number to the block on a disk. Such structure is again an inode. For that reason an inode with pointers to the blocks with inodes is used. This is in a way a file and it is called *ifile*. The inode that maps the blocks of this *ifile* is called an ifile inode, instead of awkward an inode of inode of inodes. Free inodes bitmaps and group descriptors are also stored in the *ifile*. More about this later.

The same problem faces the free-blocks-bitmap. In the Ext2 file system, the free-blocks-bitmap is stored in the same place for every group, but LLFS must move these free-blocks-bitmap blocks around. For 4096 byte blocks, one free-blocks-bitmap block contains 32768 bits with information which of 32768 blocks are free. These 32768 blocks compose one group.

I have chosen again an inode to represent a mapping from a group number to a location of the free-blocks-bitmap block on the disk. Another benefit of this bitmap inode is that the code that manages the no-in-place writes for ordinary inodes can be reused.

The only block that is stored in LLFS on the same place as in Ext2 file system is the super block. The super block contains a pointer to the block where the ifile inode is stored and from this point all inodes including the bitmap inode and from there everything that is needed can be found. The fact that *.ifile* and *.bitmap* files are not fixed allow for having more clones in one file system.

For 4096 byte blocks, there are 512 group descriptors in one block. All descriptors occupy part of the *.ifile* from the sixth block. This is to avoid indirection for inodes that are accessed all the time and group descriptors.

For the clone support the super block keeps not only one pointer to the *.ifile*, but an array of pointers to many *.ifile*s. The challenge is that the different clones do not overwrite each other's blocks.

Let's consider for a moment that we have created more clones. They all have their free-blocks-bitmap and to find a free block, free-blocks-bitmaps of all clones must be checked. When a free block is found it is marked only in the free-blocks-bitmap of the current clone as taken. When a block is freed, the block it is marked as freed only in the current clone. And that is all in order this to work. Destroying the clone means that its bitmap is not available anymore and its blocks are free to be taken by any clones.

3.2 Allocating Blocks

An LLFS partition is divided into groups. One group consists of $8 * blocksize$ blocks, which is the number of bits in one block. That way a bitmap that fits in one block, can map exactly one group. The first block in a group is the super block. In the second block pointers to the clones could be stored. This is would allow for yet more clones, but it is not currently implemented.

The super block is overwritten in place. All other blocks in the group are like data blocks. Unlike the Ext2 file system, in LLFS the inode bitmap, inode table and data block bitmap, group descriptors are not fixed, but belong to the data area.

Other structures that are used for allocating blocks are group descriptors. Group descriptors hold information that helps to decide in which group of blocks there is enough space for file and meta data to be written. LLFS tries to put a file in a single group if it is possible.

The data block bitmap is a file with information one bit per block, which blocks are free in the group. Every time a new block is allocated, bitmap blocks are searched until a zero bit is found.

LLFS complicates the matter in that data block bitmaps are not fixed

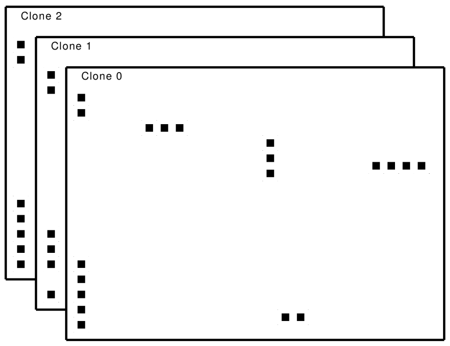

Figure 3.1: Free blocks bitmaps

on one location. When a new block is allocated, a new data block bitmap block can be also allocated if it was not already for some other block from this group. It is possible that the bitmap block for some group is in another group. The location of the bitmap block for every group is stored yet in another block, as part of an inode structure, that can be allocated too if it has not been after the last commit. See figure 3.1.

3.3 Freeing of Blocks

When a block is freed its corresponding bit in the bitmap block is set to zero. Here again the bitmap block is not updated in place after it was committed. Note that every such no in-place update causes the allocation of a block, and frees the block where it was just before commit.

3.4 Disk Layout

The whole file system is divided into equally large groups of blocks. One group consists of so many blocks that are addressable by one bitmap block (one bit per block). For example with block size of 4096 bytes, there are $4096 * 8$ bits. That means that in one block group there are 32768 blocks. One block being 4 kilobytes it is then 128 megabytes in one block. The first block in such a block group is the super block, although when sparse super blocks are activated it does not have to be, since in a large hard drive, this would result in thousands of super blocks. The super block is the only block that has a fixed location in LLFS. All other blocks that had fixed locations in Ext2 like block and inode bitmaps, inodes and group descriptors belong to the data area. See figure 3.2.

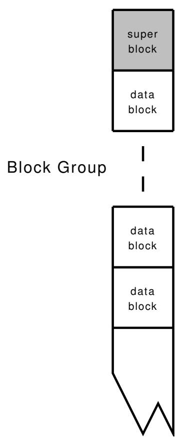

Figure 3.2: Blocks in LLFS block group

Chapter 4

Using LLFS

This chapter describes possible use of LLFS. How to get it to compile and start, how to create, remove and use clones.

4.1 Operation

LLFS is a kernel module that can be loaded on demand and used. Normally it is loaded automatically when the file system was created with mkllfs command and is mounted. I had to make some minor changes to the VFS, the Linux kernel has to be patched, compiled and installed. The patch does not influence other file systems.

4.2 Creating an LLFS File System

For creating a new LLFS file system the *mkllfs* command is used, with a block device file as an argument. For example

```
mkllfs /dev/hda1
```

creates super blocks, group descriptors, root, lost+found, .ifile, .bitmap and .config directory entries and several associated inodes on the hda1 block device.

4.3 Mounting a Clone

After the file system is created with *mkllfs* command, it can be mounted as usual with *mount* command. This mounts the master clone (clone 0) and the file system can be used like any other file system on Linux.

To mount a different clone than master clone a special option in mount command can be used[1].

```
mount /dev/hda1 /mnt/llfs -o clone=2
```

mounts clone number 2 to the /mnt/llfs mount point.

4.3.1 Creating a Clone

To create a clone for example from clone number 2 to clone number 4 is accomplished with command

```
llfs-clone /dev/hda1 2 4
```

The last consistent state of clone 2 is cloned. After clone 4 is mounted it contains the same data as clone 2. After clone 2 is written to or is even destroyed clone 4 can still be used.

4.3.2 Disposing of a Clone

Command

```
llfs-remove /dev/hda1 2
```

removes the clone number 2 and frees its blocks for further use.

[1]At this stage LLFS supports maximum 10 clones and mount option does not work. Instead different clones can be mounted on fixed directories. /llfs1, /llfs2, . . . , /llfs9. If the file system is mounted on any other directory it is mounted as a clone 0. This way LLFS can be used like any other file system, but it is also easy to access different clones.

Part II

Implementation

Chapter 5

Implementation Details

5.1 From Ext2 to LLFS

The implementation of LLFS began with the Ext2 file system. First I made a copy of Ext2, renamed it and made a configure option for my new file system. After succeeding to load it, I started to modify chunks of code, still keeping Ext2 functions in operational state so that all the time I could test the changes. Over time overwriting more and more Ext2 functions I have changed Ext2 to the LLFS file system. The reason for starting from Ext2 is that I could use its allocation methods and thus inherit the clustering already working and optimized. The data structures and layout of functions/callbacks required only minimal changes.

5.1.1 Implementing Meta-Data

In the beginning the implementation consisted of changing the inode and data block bitmap code. I have created an *.ifile* inode that contains inodes including itself. These inodes are stored sequentially in the *.ifile*, but not necessarily sequentially on the disk. To know which inode numbers are free, a free inode bitmap is used and it is stored in the same *.ifile*. See figure 5.1.

The very first block of the *.ifile* file is the free inode bitmap that covers $block\_size * 8$ inodes. It means 32768 inodes, if block size is 4096 bytes. One on-disk inode needs 256 bytes, so there are 16 inodes stored in one block

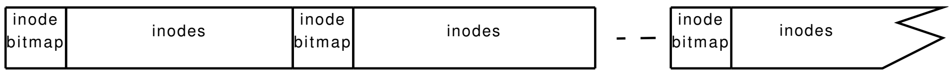

Figure 5.1: Ifile inode

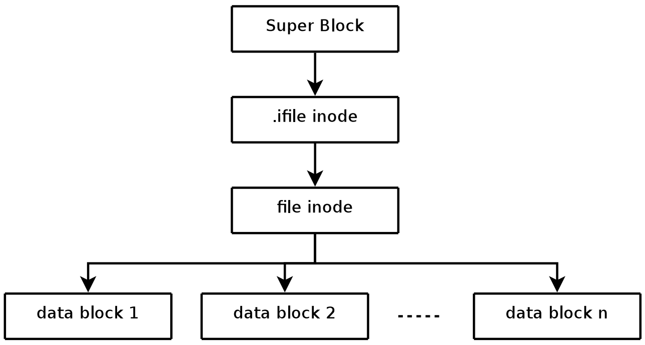

Figure 5.2: Locating data blocks from a file

and 32768 inodes are stored in 2048 blocks. The next free inode bitmap is stored in the 2049th block and covers the next 2048 blocks of inodes.

Later group descriptors structures were added to the same file. I have stored a pointer to the block where the *.ifile* inode is located in the super block, made it an array and different clones on one disk partition were possible.

In order to find data blocks of a file, the super block is read. The super block contains a pointer to an *.ifile* inode for a specified clone. The *.ifile* contains an inode of the file, which in turn contains pointers to all data blocks of the file (see Fig. 5.2).

Similarly I have created a *.bitmap* inode that contains free blocks bitmap. That solved my problem of more clones having their own bitmaps that can be removed and created instantly, with one trade-off though, that more bitmap blocks have to be scanned for free blocks, if more clones are used. I was thinking of one more in-memory bitmap that would serve as cache to represent all the clones and speed up the search for free blocks, but this I did not implement and left it for the further work.

All other meta-data that are stored in inodes and other structures did not require any change from the way it is implemented in Ext2.

5.1.2 Implementing Group Descriptors

The next task was the desc structure. In the Ext2 file system it is stored redundantly in every block group right after super blocks. I have removed pointers to the inodes and bitmaps that I did not need anymore. This made the structure smaller, but still to store it redundantly as it is in the Ext2 file system multiplied with number of clones would take too much space. I decided to part with this redundancy, which would not buy much anyway. The

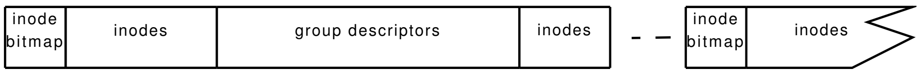

Figure 5.3: Ifile inode with group descriptors

free blocks can always be regenerated in fsck by walking through all the allocated inodes, recording which blocks they have allocated so this information is already redundant.

Group descriptors are used for finding out how many blocks are free in the described group, so that it is easier to find cluster of free blocks. It is also easier to count the free blocks of the whole file system.

Counting of free blocks is more difficult, because the group descriptor for one clone does not give information how many blocks are really free. This is because the allocated blocks can overlap between two clones and free blocks count contains only free blocks for this clone. This causes problems not only for counting of free blocks, but also for determining which block groups are available for allocation.

The group descriptors are stored in the *.ifile*. They could be stored for example in the *.desc* or *.bitmap* file, but I have decided to reuse the *.ifile* file, because it was less effort to code. See figure 5.3.

5.1.3 Implementing mkllfs

Sometime during this implementation I needed a tool, to create an empty file system with meta-data, root inode and root directory entry, so it was possible to mount it. For that I modified *mke2fs* tool that creates an Ext2 file system and named it *mkllfs*. It was much easier to do than to program *mkllfs* from scratch. I could reuse the Ext2 file system way of making root, *lost+found* and *bad-blocks* directories, respectively their inodes and added *.ifile*, *.bitmap* and *.config* files. I initialized the *.ifile* inode with this just mentioned inodes and marked bits that were taken by this procedure, in the *.bitmap* file. This part of implementation was outside of the kernel.

5.1.4 Implementing Copy-On-Write

At this point I could use LLFS with the new inode and bitmap code, but still it did not do anything more than the Ext2 file system could do, but now I could set up to work on the copy-on-write feature. Copy-on-write is to not to overwrite blocks, but allocate new position on the disk that was free and move the block over there. But we do not want to do copy-on-write all the time. When a buffer or page is in memory and it is subsequently changed as is often the case, it would make little sense to copy the buffer around by every change. A buffer or page should copied-on-write only just after it was committed and before the next commit it can be overwritten in main memory over and over. What does it mean that the buffer is committed and when the buffer is committed? Well, as for now LLFS does not solve this correctly and further work on this is required. When a buffer is synced to the disk it is considered committed. This works pretty well, when buffers are synced periodically and the super block is synced last. Unfortunately this is not always the case. For example when main memory is nearly full and buffers are freed from memory and synced to the disk in any order. But still with this approach I could test the file system, only I had to make sure not to have nearly full memory. It also gave me some incentive to fix the memory leaks.

Leaving the question of committed blocks for later, I could start to work on copy-on-write. The bitmaps, group descriptors and inodes are accessed as it is in the Ext2 file system through buffers, so every time a buffer with bitmap, group descriptors or inodes is overwritten, it is checked if it is committed and if it is, new block is allocated and it is copied to a new location. Changing the location of a block in *.ifile* or *.bitmap* file also changes the inodes of these files. A block that contains these inodes[1], if it is not committed must be reallocated again. This basically happens the first time any block is written to and then this block is overwritten only in memory, till the next commit. Meta-data are accessed at many places in the code. The free block bitmap is accessed during allocation of the new block, inodes during writing of files, or changing the directory entries, renaming of files and so on. Group description is written to during allocation of blocks as well.

Having implemented the copy-on-write for meta-data I turned to the copy-on-write for files. Before file blocks are overwritten the `prepare_write` function is called, there I can see if buffers are committed or not, and if yes, a new location is allocated for them and the buffers are copied. `Prepare_write` works with pages that contain the buffers and every time a copy-on-write happens, a new page is allocated in memory. I had to change `prepare_write` to return the new page, so that a subsequent call to `commit_write` gets the new page and not the old one. Using my redefined `prepare_write` function I could implement copy-on-write for all the directory entry functions like readlink, create, unlink, rmdir, mkdir, mknod, symlink, rename and so on defined in *namei.c* and *dir.c*.

[1]Both inodes do not have to be stored in one block, but I laid out the inodes in such a way that they are in one block, if usual block sizes are used.

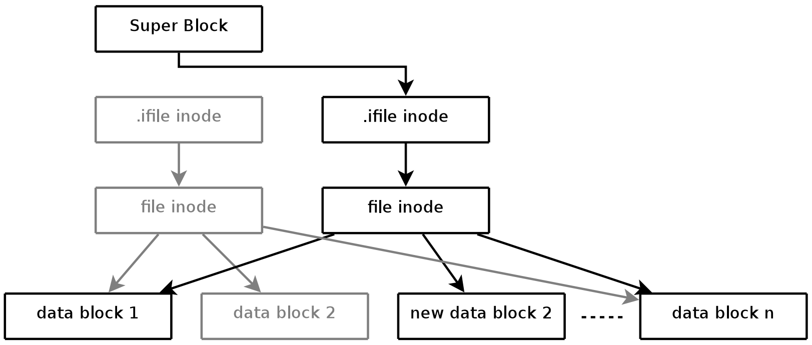

Figure 5.4: Copy-on-write

When one data block is modified for the first time after it was committed, it is copied. At the same time several other blocks are updated and depending on if they were modified for the first time after they were committed, they must be copied. In the simplest case when the data block is modified, a block that contains its inode, *.ifile* inode and super block are also modified (See Fig. 5.4).

After implementing the copy-on-write in all these cases I could finally use more clones on one block device.

5.1.5 Implementing Indirection

Up until now I neither implemented nor mentioned indirection. Every inode can store pointers to 12 blocks that store its file. Depending on block size this can be 12 or 48 kilobytes. If the file is bigger the remaining part is stored in indirect blocks. The inode contains a pointer to the block that stores pointers to the real data blocks. So for example, with an 4096 byte block size, one block contains 1024 pointers[2] with 4K blocks. Together with direct blocks and indirect blocks an inode can address 4 megabytes plus this meager 48 kilobytes. This is of course still not enough, so there are double and triple indirect blocks. With double indirect blocks already 4 Gigabytes are addressable and with triple indirect blocks 4 Terabytes. See figure 5.5.

Because my inodes and free block bitmaps are also addressed with inodes, indirection was needed for these meta-data as well. One inode takes 256 bytes, so there are 16 inodes in one block, so only 192 inodes are addressable directly. Anything over that must be stored in indirect blocks.

Let's have a look at the free block bitmap. Assuming 4K blocks, One

[2]one pointer takes 4 bytes

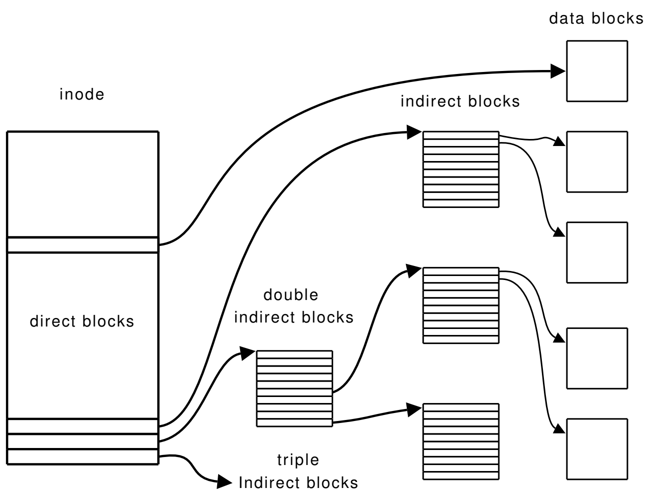

Figure 5.5: Indirect blocks

bitmap block contains free blocks of one group. One group is 128 megabytes of data. Directly a bitmap inode can address 1536 megabytes. In the second level of indirection, already over 128 Terabyte is addressable. It is reasonable scalable for me, but further work can be done here. For example not to use inode for meta-data but some other structure.

Implementing indirection in files was the relatively easy part. I added copy-on-write code for blocks with pointers to the indirect blocks and the indirect blocks. The same I could do with inodes.

5.1.6 Implementing Clones

Implementing the copy-on-write for data and meta-data, I could now start to work on the fun part, the clones. After the changes above, all the meta-data of one clone can be accessed using the *.ifile* inode of this clone. The inode is stored along several other inodes in one block with a unique block number. The next step was to store these block numbers with *.ifile* inodes somewhere. Since an Ext2 super block does not take the whole block on the disk, I was able without much effort to add an array of these block numbers for about 100 clones there.

The next issue was for the module to know which clone is used. Because the file system function are called from VFS that does not support having more clones, it is not very easy. For some functions that work with inodes and get inodes as parameters, it is possible to find out the clone number from the inode number. Some function have directory entries as parameter. From directory entry it is possible to obtain a mount point directory entry; now I need to know which mount point directory entry belongs to which clone. For that a map would be needed to map this mount point to the clone number. This is certainly doable, so I decided as proof of concept to code the clone number in the mount point directory entry. So for example /llfs1 mount point is used for clone 1, /llfs2 for clone 2 etc. This allowed for 10 clones. I initialized all of them to be clone of an empty file system during *mkllfs*. In the beginning all the clones point to the same *.ifile* inode with the same inodes and same free blocks. When one clone is mounted and it is written to, immediately its *.ifile* inode starts to be different from other clones along with other modified blocks. At any time one clone can be cloned again. That requires to just overwrite the pointer in the super block to point to the block with the *.ifile* inode.

A nice thing about newer kernels is that it is possible to mount one block device on many different mount points. With that I could mount two or more different clones at the same time.

Destroying a clone is done by setting its pointer to the block with the *.ifile* inode to a zero and invalidate all cache entries for this clone. Block group free counts would have to be recalculated, but this is still not implemented.

5.1.7 Implementing Inode, Dentry and Page Cache

The inode cache is part of Virtual File System that stores inodes, once read from underlying file system in the memory, so that subsequent reads of inodes are served from the cache.

The directory entry cache (dcache) is also part of Virtual File System that speeds up path lookup in the file system. When the path is not in the cache VFS asks underlying file system to look it up, stores it in the cache and avoids subsequent queries to the file system. This works excellently with traditional file systems, LLFS uses it as well, except that it causes all sorts of problems when more clones are used at the same time.

The virtual file system does not support mounting of more clones on one file system. Especially the inode and dentry caches get in the way. When one clone is read and after a short while another clone with the same path names or inode numbers, first the caches are checked in the VFS and cache hits from other clones are returned. I have solved this temporarily: inodes in different clones have unique inode numbers in memory, different than on the disk. Another problem is that VFS assumes that there is only one root dentry. This does not work for LLFS, so I had to do some changes in the virtual file system.

Because the same inodes have different inode numbers between clones, the inode cache is not a problem. On the other hand dentry cache is, because dentry cache checks part of a path till the mount point and can return inode numbers from different clone. I solved it, so that if something like this happens, the cache is invalidated and this information must be obtained from the disk again. Note that this happens only if they are the same files in the same directories in different clones and they are read about the same time.

Having explained this in detail, a more elegant solution should be possible: when a way to map different clones to different device files will be implemented in the future, the whole inode-and-dentry-cache problem would go away because the VFS caches would treat different clones as different file systems.

Similar problem can arise, because of the page cache. Although in the current implementation I have made an easy way out, in that the page buffers are copied in the memory before they are modified. This should not be necessary all the time though and this memory copy could be optimized away, because if only one clone uses the buffer in the memory, it would be enough to reallocate it on the disk, but leave it on the same place in the memory.

5.1.8 Implementing Block Allocation and Deallocation

Having more clones I had to implement block allocation that looks for free blocks in all the clones. Searching for free blocks proceeds as it is in the Ext2 file system with addition that all clones must be searched. Once the block group with free blocks is identified, the bitmap buffers for this block group for all clones are read. The free block is found if in all bitmaps this particular bit is set to zero. When a free block is found its bit is set to one only in the clone for which it is used. For every other clone it is zero. Other clones will not allocate this block, because they check again against all the clones. When the clone is destroyed, other clones no longer check its bitmap and the block is again available.

With indirect blocks this got more complicated. When allocating/reallocating an indirect block in the *.bitmap* file, the block with pointers to the indirect blocks that points to this indirect block needs to be allocated. It can be allocated in another indirect bitmap block, where the same events have to take place. Although this should not happen often with clustering, when file system is almost full, this allocating of indirect blocks can go on forever. This can be later improved, so that the indirect block is allocated in the same block as its parent or not at all, thus avoiding the recursion.

An additional difficulty is if a bitmap block is allocated in its own block group. Even more so, if the indirect bitmap block is allocated in its own block group and its parent.

To make it clear, with direct blocks there are two possibilities:

- the bitmap block is allocated in a different block group
- the bitmap block is allocated in the block group it manages itself.

With indirect blocks there are 4 possibilities:

- the bitmap block and its parent are allocated in different block group
- the bitmap block is allocated in its block group, but the parent is allocated in a different block group
- the bitmap block is allocated in a different block group, but its parent is allocated in its block group
- the bitmap block and its parent are allocated in the same bitmap block

Especially in the last case, if the bitmap block and its parent are newly allocated, a chicken and egg problem arises. The bitmap block cannot be allocated before the parent is and parent cannot be allocated because the bitmap block still does not exist.

With double indirect blocks the problem is similar but more complex.

Finally I made it work for simple indirect blocks, but this part of code can be and should be improved.

5.2 In-Memory and On-Disk Data Structures

5.2.1 Super Block

The super block is a central data structure for a file system. The VFS super block in-memory structure closely relates to the LLFS super block. The VFS super block data structure defined in `include/linux/fs.h` contains information about file system as a whole, on which block device it is mounted, the block size that is used, if it is dirty and needs to be written, the maximum file size, file-system type structure, callbacks for super-block operations, a magic number, the root directory entry, the pointer to all inodes, locks and other data, pointers and flags. Every file system can extend this structure with its own super block in-memory data. LLFS uses it to add information about the number of group descriptors, their sizes and pointers to the .ifiles of all the clones and their root dentries.

When a super block is written to the disk, the in-memory super block is converted to the structure that contains data from the VFS super block and extended LLFS super block in-memory data. The integer numbers are converted to the little-endian byte order, so that different architectures can read the same disk, whatever their representation of these numbers is in memory.

The super block is the only data block that is written to the same place in LLFS. The super block must be stored on predefined place, so that it can be found after a file system is mounted. I could reuse most of the Ext2 super block code and attributes. LLFS super block contains additionally an array of ifile block numbers, so that different clones can be found. The in-memory LLFS super block also contains pointers to the root dentries of all the clones. These pointers are not written to the on-disk super block.

5.2.2 Inode

Similarly to the super block, there is an in-memory VFS inode, extended with LLFS data and a converted inode structure that is written to the disk. The in-memory VFS inode contains the inode number, link count, permissions, sizes and other data associated with files, directories or special files. The LLFS in-memory inode contains pointers to the data blocks and indirect blocks. It also contains group number in which file or directory is stored.

I did not have to change the Ext2 inode structure or inode info structure. In reality I needed to make inodes with same number from different clones distinguishable for the inode and dentry cache. I could store clone number in the inode info structure and make VFS aware of clones, this would be the right approach.

For now when the inode is read from the disk its in-memory inode number is changed. This way I can find out to which clone this inode belongs and also the inode and dentry cache are happy. The formula for the in-memory inode number is $ino = real\_ino + clonenr * big\_number$ where big number denotes available inodes divided by available clones. To get a clone number to which this inode belongs is simple: $clonenr = int(ino/big\_number)$ When the inode is written, the real inode number is needed and this is computed like this: $real\_ino = ino - big\_number * clonenr$. As you can see, this reduces the number of available inode numbers, the more clones are allowed to be created. This is but a temporary solution.

Additionally every inode stores `i_block_group`, `i_next_alloc_block` and `i_next_alloc_goal` numbers.

i_next_alloc_block is the most recently allocated block in this file relative to this file. It is used to detect the continuous allocation of blocks.

i_block_group is the number of the block group where this inode is allocated. This is used to allocate directories near its parent directory. In the Ext2 file system this number is constant. In LLFS the inode location changes on the disk, so does the i_block_group number.

i_next_alloc_goal contains the physical block where the most recent block of this file was stored.

5.2.3 Group Descriptor

An LLFS group descriptor structure contains free-block and inode counts and a used directories count. All these counts apply to one group.

An Ext2 group descriptor structure contains additionally pointers to the blocks and inodes bitmap blocks and inodes table block. This pointers are not needed in LLFS, because they are stored in *.ifile* and *.bitmap* files.

Thanks to this, one LLFS group descriptor takes 8 bytes of the disk space and is more lightweight than Ext2 group descriptor which takes 24 bytes.

Free blocks counts in the descriptors are also used for determining of free space on the whole disk. Free blocks of every group are read and summed up. When a new clone is created it does not immediately consume disk space, because it shares all the blocks with the parent. In time, as new blocks are assigned to the new clone, it starts to take disk space. Let's suppose that a clone is created, after that its parent is destroyed, now the new clone should account for all the blocks that it shared with parent. This is not a problem, since the new clone contains its own copy of the free-block count from the parent. The problem is that during existence of parent and its clone, while new blocks are created and removed, it is no longer known which blocks are shared and which are not. The solution for this is to keep yet another desc structure that contains a free-block count of blocks that do not belong to any clone. This will come at a price when a clone is discarded. The free-block count would have to be calculated again.

5.3 Functions

This section describes some Ext2 and VFS functions and changes that were required in order to implement LLFS. This section can be safely skipped, if you are not a kernel programmer.

- grab_block

  `grab_block` function gets a bitmap block and goal as arguments. Goal is a preferred location in the bitmap block. If the goal bit is zero in the bitmap, it means it is free and the search is over. If that fails, the zero bit will be searched sequentially to the next 64 bit boundary. When no free bit was found, the rest of the group is searched for one zero byte. Notice it is byte not bit. If byte was found the search is continued backwards to find the first zero bit in this group of adjacent free bits. If this fails the free bit is searched bit by bit from the goal to the end of the bitmap block. If it all fails it means that all bits in the bitmap block are taken and -1 is returned. LLFS's `grab_block` has to check bit/bytes for all the clones. Although not finding any free block should not happen with the Ext2 file system, it can happen more often in LLFS, because in the current implementation if more clones are using one block group, the descriptors hold only information about free blocks for this clone. But how many blocks are really free by all clones is not known. This is because several clones can own the same blocks, but other blocks are owned only by one clone.

- reserve_blocks

  blocks are reserved in a group descriptor for a block group, before they are allocated on the disk. During this time the block group is locked. If allocator is waiting for this lock and block group gets full, the reserved blocks are released and next block group is used.

- prepare_write

  The Ext2 file system uses the `prepare_write` function from VFS. I had to define my own, in order to implement the no-in-place updates. `Prepare_write` goes through all the buffers on a page that is supplied as an argument and checks the state of the buffers and prepares them to be written. It is repeated for every page on which the file is stored. After `prepare_write` the data from user space are copied and `commit_write` is called. Only `prepare_write` and `commit_write` can be overwritten in the file system, copying from user space happens in the VFS.

  If the page is up-to-date, every buffer on the page is marked up-to-date if it was not already. Preparation is over right here if the page is up-to-date. If it is not buffers are inspected further. If the buffer is new, the new state flag is cleared. If the buffer is not mapped, meaning it is not associated with a block in memory and/or on the disk, the block is fetched or newly allocated respectively. If a new block was allocated at this point, it is either up-to-date or the buffer data that are outside of the range that file systems wishes to write, are zeroed. Again if the whole page is up-to-date, the buffer is set up-to-date, if it is not. If the buffer is not up-to-date and does not have its delay bit set and it is part of the buffer that will be written to, it is read from the disk. Then the new buffer bits are cleared for all the buffers on the page, since they are either read if they existed or zeroed.

- commit_write

  `Commit_write` goes through the buffers on the supplied page and marks them dirty and up-to-date, if they were overwritten after `prepare_write`. Additionally if all buffers on the page are up-to-date the page is marked up-to-date. The Ext2 file system uses `commit_write` from VFS. LLFS overwrites it only in order to parse the *.config* file for cloning requests.

- generic_file_buffered_write (filemap.c)

  is a VFS function that writes a file to the disk calling `prepare_write` and `commit_write`. It loops through all pages that are stored in memory for the file (or will be) and calls `prepare_write`, which can be redefined in a file system. After that page is up-to-date and in memory and data are copied from user space with `filemap_copy_from_user` and `filemap_copy_from_user_iovec`. After that `commit_write` is called. In LLFS there is a need to modify this function, unfortunately it is not possible without changing the VFS code. In VFS the `prepare_write` prepares buffers on the same page that is later committed with `commit_write`. During `prepare_write` in LLFS the buffers from the page are copied to a new page then the data from user are copied and `commit_write` is called on a new page.

- block_to_path

  returns depth of the indirection and offsets in the intermediate nodes of indirect blocks for inode block. For direct blocks it returns 1 and offset[0] is set to `i_block`. They are also indirect, double and triple indirect blocks. Boundary flag is set if the block is last before a possible next indirect block.

5.3.1 dir.c

In *dir.c* I had to make modifications to following functions:

- readdir

  While reading a directory entry a cache can return an inode from a different clone. This can be detected and the inode is reread for the current clone.

- inode_by_name

  This function returns the inode number by name from parent directory. Here LLFS incorporates clone number in the inode.

- add_link

  This function adds a directory entry to a directory. When a directory entry is added copy-on-write is implemented.

- delete_entry

  Copy-on-write has to be added in this function as well.

5.3.2 namei.c

In *namei.c* I had to make modifications to following functions:

- d_compare

  I have rewritten d_compare callback in dentry_operations structure. The llfs_compare had to be made aware of different clones and return false if dentry cache contains a directory entry from different clone.

- lookup

  This is the inode_operations callback. When an inode is looked up for the first time the inode number in the memory is changed to encode the clone number.

- create, mknod, symlink

  These callbacks create an inode for a created directory entry. At the same time the clone number is encoded in the inode by LLFS.

- mkdir

  This function creates a new inode and a directory entry. The clone number is known from the parent directory, so the new directory can be created for the right clone.

- unlink

  Unlink removes specified directory entry and decrements the inode usage count. All this must happen with copy-on-write.

- rename

  Renaming modifies two inodes and directory entries if the renaming is possible. The modification is copy-on-write too.

Part III

Testing, Debugging and Benchmarking

Chapter 6

Testing and Debugging

Running and debugging a kernel module is different from running and debugging a user space application. First of all any fault in kernel code leads to a crash or undefined behavior of the system and the whole machine should be rebooted.

To overcome this inconvenience I used User Mode Linux (UML) with gdb[1] for testing and debugging. UML is a virtual machine that runs Linux on top of Linux. That way, if a crash occurs, only the virtual machine is affected and only a restart of the virtual machine is required. It is also possible to attach gdb to the virtual Linux and get stack traces, with function names and line numbers, which I used extensively. It is also possible to set breakpoints and use commands to step through the code line by line, although this feature is decreasingly useful with increasing complexity of the code and parallel execution.

The Linux kernel also allows to turn on some checks for detecting deadlocks, memory allocations, soft lockups, mutex semantics violations among other things and print out stack traces.

Another debugging tool is simple *printk()* that prints text and just any type of variable like its user land counterpart *printf()*to the log. Amazingly *printk()* does work if it is called from any part of the code without any concurrency issues. Putting the *printk()* on right places can help with detecting code-flow problems and inspecting values of variables. Very often debugging of all sorts of problems was done by putting and removing temporary *printks* in the badly behaved code. Another kernel function *WARN_ON* can be used to dump out stack traces and the more radical *BUG_ON()* function that stops the execution of the kernel.

It is also important do debug under conditions that do not occur so often,

[1]GNU Project debugger

for example, when main memory or the file system is almost full.

During the implementation of LLFS I wrote myself small test programs that were testing various usage patterns of the file system. I could run them in endless loops to catch rarely occurring bugs or I could run all the tests one after another in order to see if the latest fix did not break something else.

Some tests looked like this:

- copying one small file, removing the file
- copying one big file, removing the file
- copying many files, removing the files
- creating many small files, removing the files
- creating a directory, removing the directory
- creating a symlink, removing the symlink
- moving a directory, moving a file
- making a clone of empty file system, writing to the both clones at the same time
- the same as above, but with ten clones
- writing to a file, syncing, appending to the file
- copying a file, making a clone, reading the file, reading the file from clone, removing the file, removing the file from clone
- and of course copying the Linux kernel, untaring the kernel, compiling the kernel, removing the kernel

Additionally these tests were executed with various combinations of syncing, cloning and wiping out of LLFS memory buffers.

Chapter 7

LLFS Performance

To test performance of LLFS I dedicated one 50GB partition of my hard disk. I compared LLFS with the Ext2 file system and two journaling file systems: Ext3 and ReiserFS. Ext3 was tested in default *ordered* mode and in *journal* mode as well. The *journal* mode makes the Ext3 file system much slower but guarantees a level of consistency similar to LLFS. Nevertheless the aim of LLFS was to match at least Ext3 even in *ordered* mode and *journaled* mode benchmarks were added but will not be commented during comparisons.

All file systems were tested on the same disk partition. Tests where copying of files were performed, the files were copied from another disk.

For time measurements I used Linux `time` command. Before every test the computer was restarted and was made sure that no unusual services are running. The file system was created with `mkfs` command of the respective file system and the created file system was mounted. All file systems used 4K blocks.

The test equipment was a PC with AMD Athlon XP 2800+ processor with 1GB RAM. The kernel version was Linux 2.6.16. The hard drive used was Seagate ST3500630A 500 GB ATA internal hard drive.

7.1 Creating and Reading Small Files

This benchmark consisted of creating and reading of small files. First a directory was created with 10 subdirectories, all these subdirectories contained 10 other subdirectories with yet another level of 10 subdirectories. The last level of subdirectories contained 10 1K files each. If you got confused by now, together there were 10,000 files.

The whole hierarchy was recursively copied to the benchmarked file system and synced twice. A time second of `cp` command was measured plus the time that `sync` command took to write all the data to the disk. This way only write performance of the tested file system was measured and not read performance from the file system where the data came from. The results can be seen in the figure 7.1.

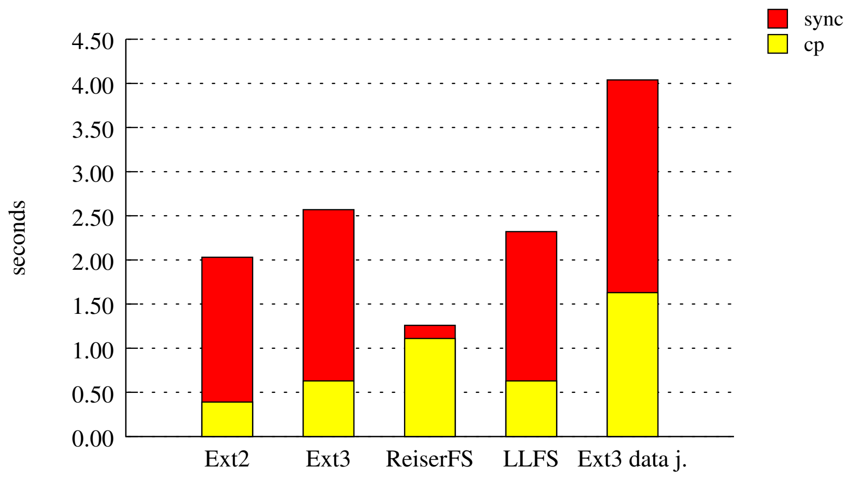

Figure 7.1: small files write performance

In this test ReiserFS was the fastest, although it took longer in the `cp` command, but then it had less to write during the `sync`. It means that user has to wait little bit longer, while the `cp` command is issued, but the writing of data in the background is much faster.

LLFS was in this test slower than Ext2, but faster than Ext3. Although LLFS needed to allocate meta-data blocks, this extra overhead was almost canceled with better spatial locality of data and meta-data.

After that the computer was rebooted to ensure that no cache interferes with results and all 10,000 files were read. See figure 7.2 for results.

In this test again ReiserFS was the fastest, since it is optimized for this sort of tests, LLFS did also good and came second followed by Ext2 and Ext3. Ext3 does not need the journal for reading, so it does not influence its read performance, but it is still slower than Ext2. LLFS fully profits in this test from less fragmentation of meta-data and data than it is in Ext2 and Ext3.

7.2 Creating and Removing of Small Files

For this benchmark the same directory hierarchy as in previous section was used. This time the directories and files were copied recursively to the bench-

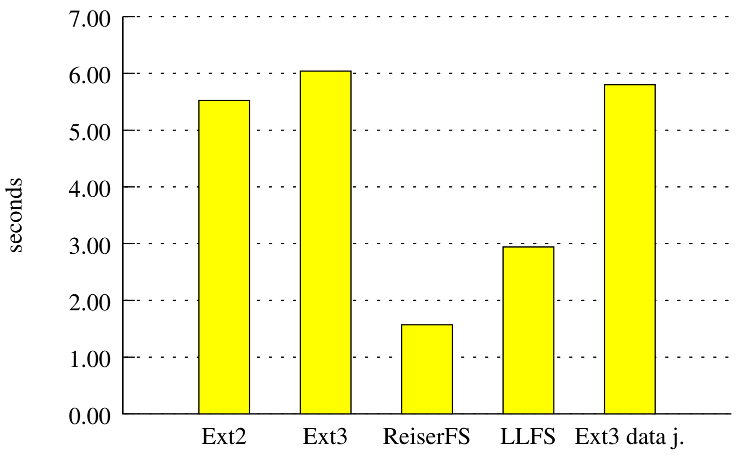

Figure 7.2: small files read performance

marked file system and immediately removed. This copying and removing of the same thing was repeated 100 times. See figure 7.3 for results.

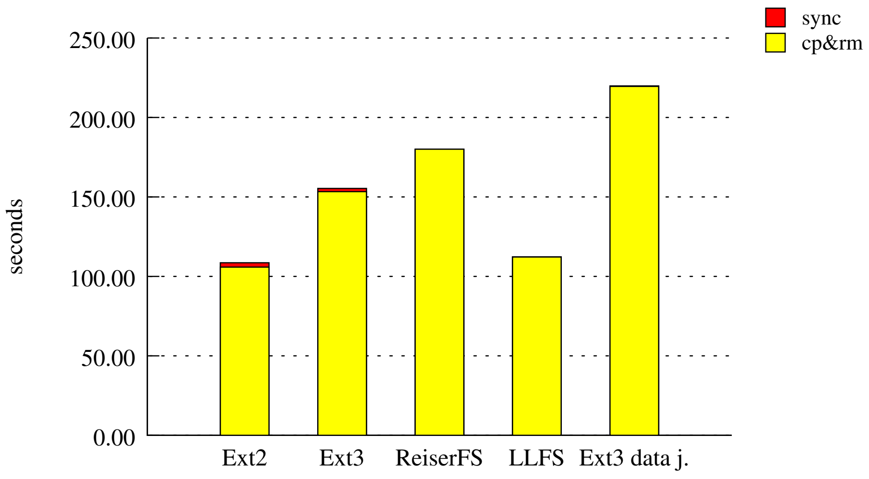

Figure 7.3: small files write and remove performance

This test is played mostly in the cache and measures most of all a file system overhead while not much is written to the disk. Here Ext2 and LLFS were the fastest, second was Ext3 that took about 37% longer and ReiserFS about 60% longer than LLFS.

7.3 Creating and Reading of Large Files

This benchmark copied 19 files each of which contained 167 Megabytes of data to the tested file system. See figure 7.4. In this benchmark all file

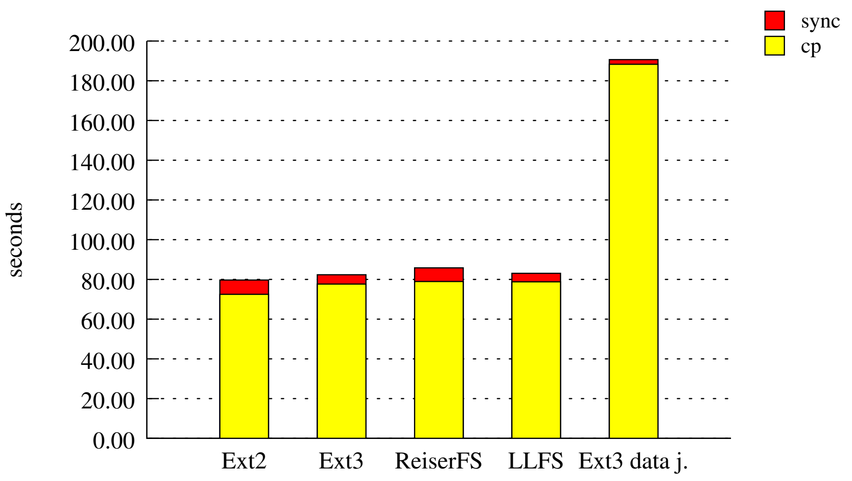

Figure 7.4: large files write performance

systems performed about the same (disregarding Ext3 in *journaled* mode).

After the reboot of the system all files were read. Again performance of all file systems was about the same, LLFS being the fastest. See figure 7.5.

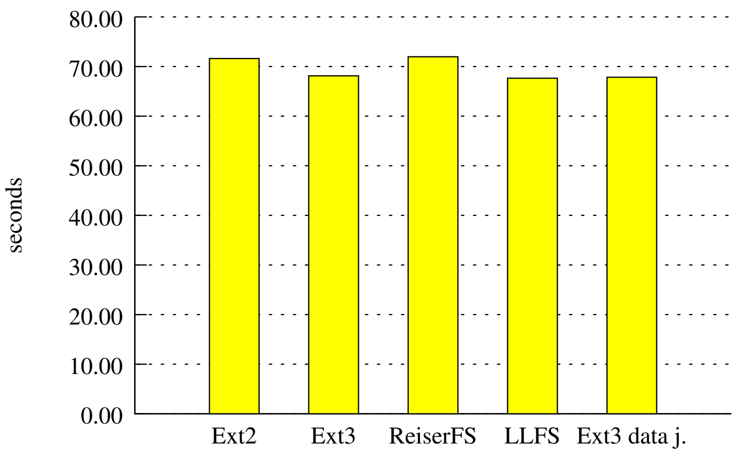

Figure 7.5: large files read performance

7.4 Writing and Reading of Log-File

This test was designed to see what additional fragmentation of copy-on-write system does to the reading performance. In this test a line with 80 characters was written to a file. The file was synced and another 80 characters were written and so on. This writing and syncing was repeated 20,000 times. In the end the file had 1.6 megabytes of data. As expected the LLFS was the slowest in this test. Ext2 performed the best. See figure 7.6. After reboot of the system, the whole file was read at once. LLFS was again the slowest, although even more so than expected and some further work is required to pin down the source of this performance problem. See figure 7.7.

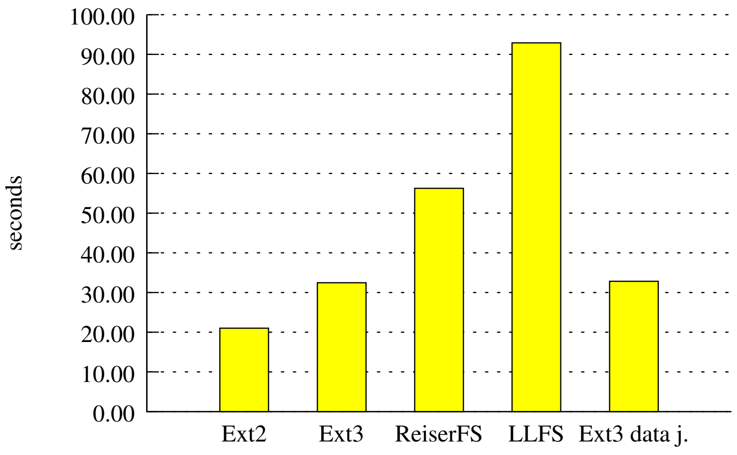

Figure 7.6: writing log-file

7.5 Unpacking, Compiling, Removing Kernel

This test was designed to compare the file systems in some real world scenario.

First a 51 megabytes tarball of Linux kernel was copied to the benchmarked file system. The tarball was unpacked to 271 megabytes, the `make` command was executed and finally the compiled kernel with 343 megabytes of data was removed. Figures 7.8, 7.9 and 7.10 show the results of this common task (for some people anyway). Figure 7.8 compares copy and figure 7.9 unpacking of the tarball. Here again the proximity of data and its meta-data helps LLFS to be the best in this benchmark. Ext3 and ReiserFS have to write their journals and are slower.

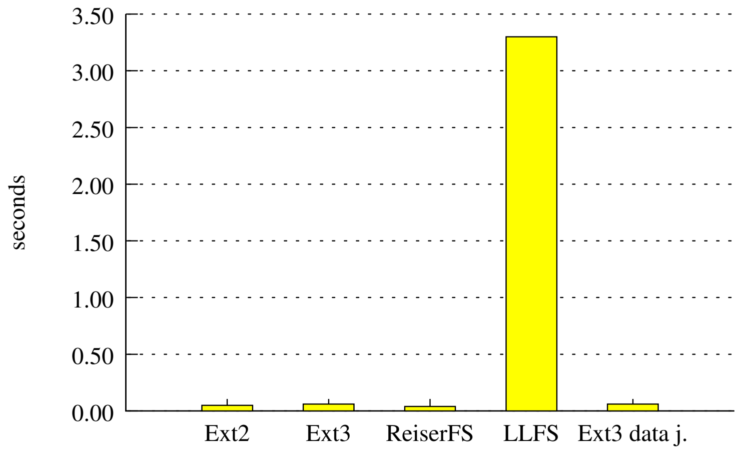

Figure 7.7: reading log-file

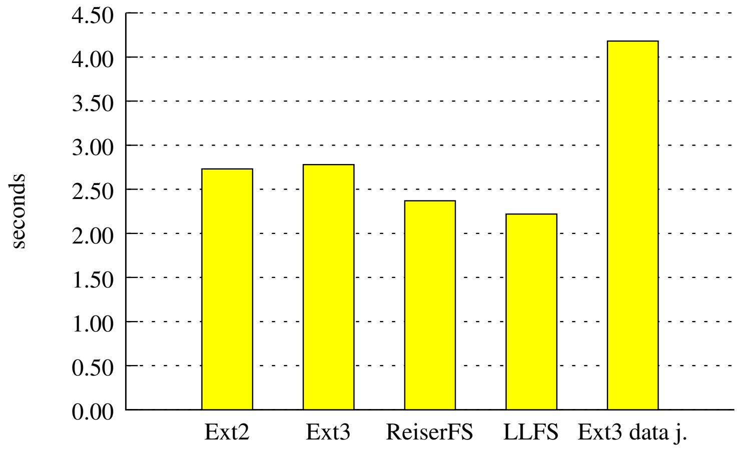

Figure 7.8: Linux kernel source copying

Figure 7.10 compares times of the `make` command. Compiling is CPU intensive and is performed mostly in the cache and not surprisingly the results of all tested file systems are almost identical.

Finally the whole kernel tree with compiled object files was removed. See Figure 7.11. In this test ReiserFS was the fastest, followed by LLFS and Ext2.

The next test was to open one by one all Linux kernel files and read all of the 7.6 million lines of code and compiled files. Again, to ensure that no caches are used, the computer was rebooted. Figure 7.12 shows the read performance results. In this test LLFS shines one more time thanks to the

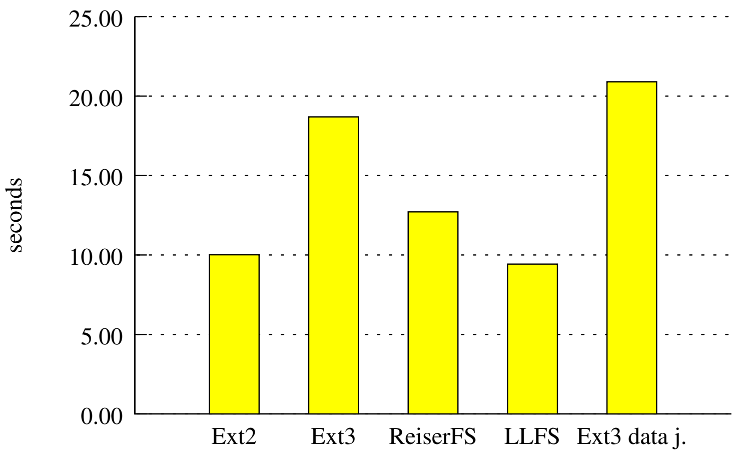

Figure 7.9: Linux kernel source unpacking

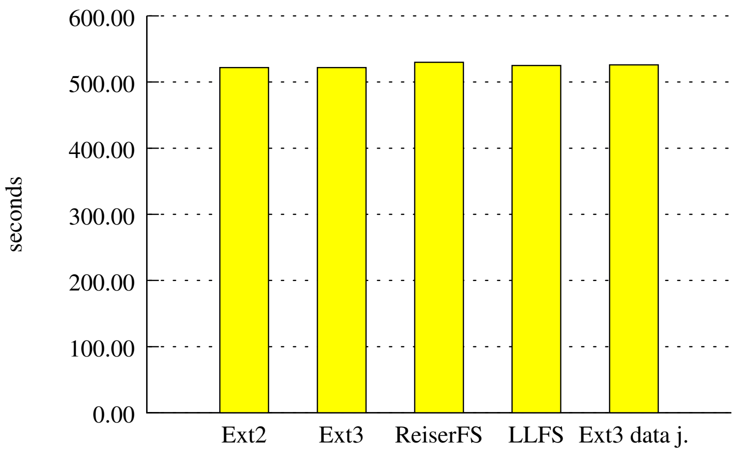

Figure 7.10: Linux kernel compiling

spatial locality of data and its meta-data.

7.6 Snapshot / Clone Performance

I made also some benchmarks with clones and without. For this test I used the files from the previous large-files test. I wanted to compare LLFS without clones, LLFS with one clone, Ext3 and Ext3 with LVM snapshot.

First LLFS with clone and without clone was tested. The 3.1 Gigabytes of large files were copied. In one test a clone was created in other was not.

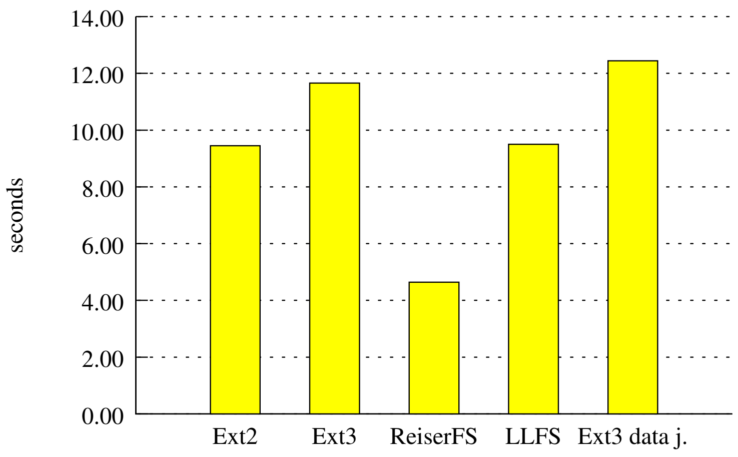

Figure 7.11: Linux kernel source tree removing

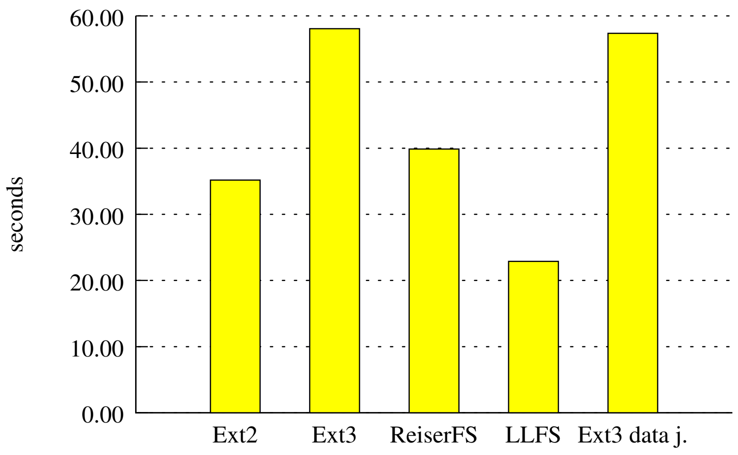

Figure 7.12: Linux kernel files reading

Then the computer was rebooted. The same 3.1 Gigabytes were copied to another directory. Only the copy command after the reboot was measured. Subsequently the `umount` command was measured as well. We will see later why.

The same tests were made with Ext3. For the snapshot test an LVM Volume was created. The same files were copied, an LVM snapshot was created and the computer was rebooted. Again the same files were copied in another directory and time was measured.

The results in figure 7.13 show that in LLFS there is negligible performance impact of a clone. Even with one clone, the LLFS is quicker than

Ext3 without LVM. On the other hand LVM snapshot makes the Ext3 file system much slower. Also the `umount` command took very long to finish and that is the reason why it was included in this comparison.

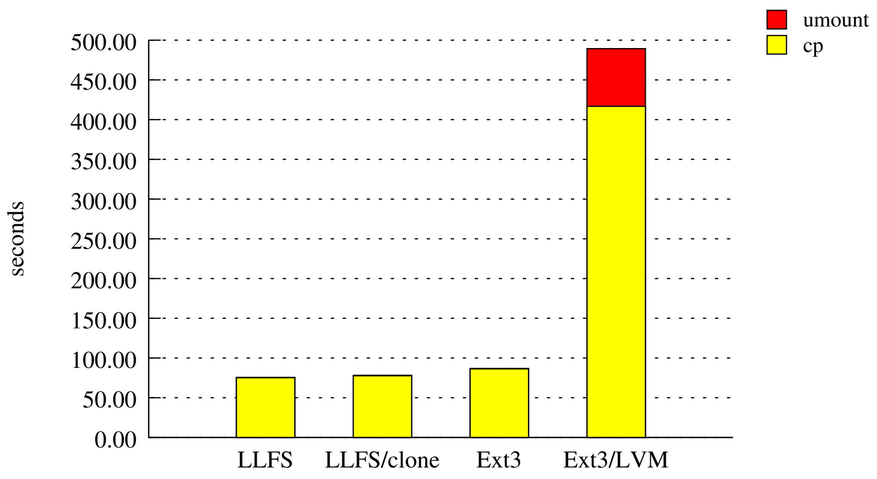

Figure 7.13: writing with snapshot

7.7 Multiple Clones Performance

To test the performance of LLFS a little bit more, I made some performance comparisons with more clones. This time I compared LLFS only with itself.

In the first test the files from the large-files test were copied 9 times to the different directories to the same clone. After that the file system with all the data was cloned 9 times. At this point the file system had 10 clones with the same 27.9 Gigabytes of data. After making sure that the caches are flushed, the directory was copied one last time to one of the clones[1]. The time of the last copy was measured.

In the second test 10 clones of an empty file system were made. Then the files from large-files test were copied to every clone but the last. Together 27.9 Gigabytes of data were copied up until now. After flushing the caches, the copy to the last clone was made and measured. This should be pretty much the worst-case scenario, because the clones share very little data and free-blocks bitmaps of all of them must be scanned.

To compare LLFS with the same amount of data on the disk but without clones, I made a test where the 3.1 Gigabytes where copied 9 times to the

[1]The copy was made to the clone 10, but it does not really matter in this case.

file system and the 10th copy was measured.

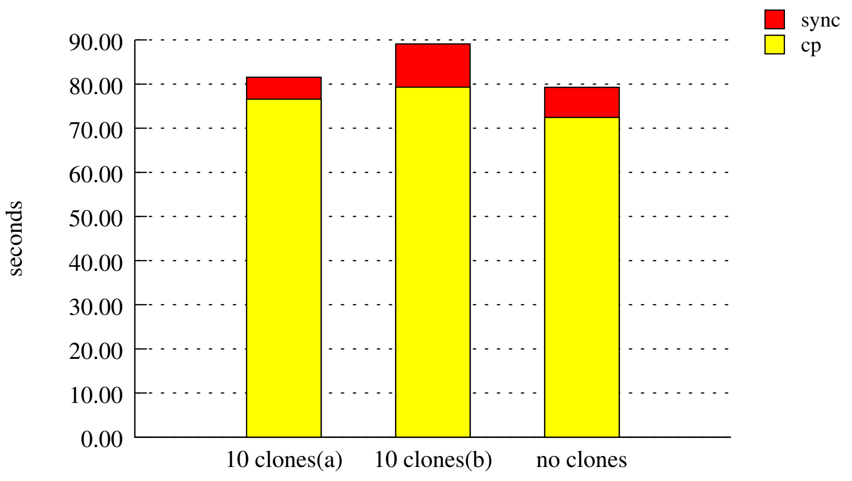

Figure 7.14: Writing with 10 clones

In figure 7.14 are the results. As expected the test without clones was the fastest, followed by the first test (a) and second test (b) was the slowest. The differences in all three tests were small, which means that even 10 clones do not pose much overhead if the system has enough resources.

7.8 Performance Test with Bonnie

In order to make some independent performance comparisons, I am including results from *Bonnie*[2] program that is part of the Debian distribution I am using. This program may not be the best hard disk benchmarking tool available or the most complete, but it is easy to use and results are easy to understand, reproduce and compare.

The first part of the test was creating, reading and removing of 1G file in various ways. See table 7.1 and table 7.2. The second test was creating, reading and removing of 102,400 1K files [3]. See table 7.3 and table 7.4. All the tests for all the file systems were run 10 times and the arithmetic average was computed.

The tables show the results. For throughput higher numbers and for CPU lower numbers are better. Like in my tests, LLFS performed about the

[2]Bonnie++, version 1.03c.

[3]All the details of the tests performed by Bonnie++ are contained in the file /usr/share/doc/bonnie++/readme.html in the Debian distribution.

same as other file systems. Only writing performance of a great number of small files[4] in one directory was very low. The same happened to the Ext2 file system because Ext2 does not use hashes or trees for directory entries like the other file systems do. LLFS has inherited the same performance problem for the same reason. However it should be easy to port the Ext3 implementation for directory entries to LLFS and fix this.

		Sequential Output					
	Size	Per Char		Block		Rewrite	
		K/sec	% CPU	K/sec	% CPU	K/sec	% CPU
LLFS	1G	38628.5	98.8	70189.4	25.5	18384.9	12.8
Ext2	1G	40669.1	96.6	68450.4	13.8	21127.6	5.0
Ext3	1G	36949.8	95.2	56069.1	23.2	20109.1	5.9
Ext3 data j.	1G	14286.8	39.3	21159.5	12.7	14006.3	7.4
ReiserFS	1G	41315.7	97.5	71711.6	25.7	22781.0	6.7

Table 7.1: Sequential output

		Sequential Input				**Random Seeks**	
	Size	Per Char		Block			
		K/sec	% CPU	K/sec	% CPU	K/sec	% CPU
LLFS	1G	30336.4	66.0	46730.5	6.8	529.9	0.9
Ext2	1G	28448.7	61.1	45676.0	6.6	745.4	0.7
Ext3	1G	26462.8	57.2	48100.4	7.2	668.4	0.5
Ext3 data j.	1G	33133.2	72.0	45415.6	6.8	697.1	0.9
ReiserFS	1G	28710.7	62.2	48816.7	8.7	765.9	0.7

Table 7.2: Sequential input and random seeks

7.9 Performance Conclusions

LLFS fulfills its promise to perform on par with other Linux file systems. Writing and reading of small or large files took about as long as in Ext2 or Ext3. ReiserFS performed in some cases much better and some cases much worse.

The reading performance especially of many small files in multiple level of subdirectories was surprisingly good. LLFS could make use of additional

[4]102,400 files with 1 kilobyte

		Sequential Create					
	Num Files	Create		Read		Delete	
		K/sec	% CPU	K/sec	% CPU	K/sec	% CPU
LLFS	102400	344.8	96.7	120374.6	99.4	34036.9	91.3
Ext2	102400	355.2	97.4	123108.4	92.1	115857.5	99.0
Ext3	102400	10359.8	81.4	69315.3	92.4	8369.5	28.5
Ext3 data j.	102400	2807.7	23.7	75161.1	96.1	17608.0	59.2
ReiserFS	102400	2459.1	26.1	486.0	1.0	410.3	3.0

Table 7.3: Sequential create

		Random Create					
	Num Files	Create		Read		Delete	
		K/sec	% CPU	K/sec	% CPU	K/sec	% CPU
LLFS	102400	347.3	97.9	113400.8	99.5	829.1	96.0
Ext2	102400	360.2	99.0	126379.3	99.2	904.0	99.0
Ext3	102400	7783.2	74.1	80792.9	97.9	7900.1	28.0
Ext3 data j.	102400	2239.9	19.7	28695.2	33.9	18064.2	62.8
ReiserFS	102400	2308.1	25.9	409.7	1.0	235.8	2.0

Table 7.4: Random create

spatial locality of the data and its meta-data and could outperform all other tested file systems.

Unsurprisingly writing files in log-file fashion did not perform very well. The resulting file was further fragmented in such an unfortunate way that reading of the whole file at once proved to be much slower than it is in any other tested file system. I believe, there it is still possible to improve the LLFS performance in this scenario a lot, but it will be never as fast as it is in traditional file systems. On the other hand this performance problem is mitigated by the fact, that the log files are usually compressed once a day which also defragments them.

Finally the performance of the file system when one clone is created is still very similar to the performance without any clones.

Chapter 8

Related Work

8.1 Beating the I/O Bottleneck

In 1988 an idea for a log-structured file system emerged. It was proposed by John Ousterhout and Fred Douglis in the paper "Beating the I/O Bottleneck: A Case for Log-Structured File Systems"[Ous88]. The concern of the authors was that exponential improvements in CPU speeds and memory sizes were not matched by similar improvement of disk speeds. They believed that making all writes to the disk in the form of an append-only log would provide order-of-magnitude improvements in write performance. Writing to the log on a disk would eliminate almost all seeks.

8.1.1 Technology Shift

In their paper the authors predict an 100 to 1000 fold increase in CPU speeds over 10 years and about 100 fold increase in memory sizes. At the same time the disks would increase their sizes, they would be smaller and cheaper but seek speeds would increase by much lower rate. These trends would force a redefinition of trade-offs needed in file systems. In fact the I/O would become a major bottleneck.

8.1.2 Solutions to the I/O Bottleneck Problem

One of the solutions to the bottleneck problem is extensive use of caches. The files as they are read are retained in a memory. Thanks to the locality in file access patterns, this cache could achieve 80-90% hit rates on then typical systems.

Although writes also make use of the cache, they must be written to the disk as quickly as possible, so the written data can be retrieved after a system crash or a power failure.

The cache improves the I/O performance of the system, but while it improves the read performance significantly, the write performance profits much less. Using the cache shifts the nature of I/O from mostly reads to mostly writes. To improve the write performance caches with battery backup are proposed. This would postpone the disk writes. The problem with this solution is that after a crash a cache recovery would have to be performed.

Finally the authors describe their then most exotic solution: a log-structured file system.

8.1.3 A Log-Structured File System

The main difference between a traditional file system and the log-structured file system is in that log-structured file system's representation on disk consists of nothing but one continuous log. The log is divided in the same-sized chunks, called segments. As files are created or modified, data and meta-data are written to an end of the log in a sequential stream.

Along with a performance improvement, the authors saw some other interesting possibilities:

Fast recovery: Fast recovery of the file system without checking the whole structural integrity of the file system.

Spatial locality: Spatial locality for files and meta-data that are written at the same time.

Versioning: Keeping old versions of modified data.

Although writes in a log-structured file system are sequential the file system needs to retrieve data randomly. If for example a file is read the file name of this file must be translated first to an inode data structure where pointers to blocks are stored that contain the whole content of the file. Sequential scanning of the whole file system would be unacceptably slow. For that a log-structured file system needs to retain data structures from traditional file systems. In log-structured file system these structures are no longer on fixed positions and one more structure “super-map” is needed to have access to them.

The log will eventually need to wrap-around. At this point there would be no more free segments, although there would be free space on the disk, created by modification and removal of data in random segments. A log-structured file system would need something like a garbage collector that reads a segment to the memory, picks up live data and moves just them to a new segment where they would take up less space, thus freeing the segment.

There are two possibilities how such garbage collector could work. When the file system is full, the file system suspends any new write requests and cleans segments in the beginning of the log by copying the live data to the end of the log. Although this approach would not create any cleaning overhead while the disk is not full, the periodical downtime, when the disk is full would be unacceptable.

An alternative is to clean the segments continuously in the background, making the file system slower, but without unacceptable downtimes.

Some performance issues are also discussed in the paper. The file map data can be written next to the file data and when they are read only one seek is required not two as it is in traditional file systems. Writing to a log file also has a negative performance impact: new entries are appended to the end of the log. Traditional file systems can keep the file's data contiguously on disk, but a log-structured file system would fragment the file. This not a problem for writing, but for reading the whole file at once.

8.2 Log-Structured File System Projects

8.2.1 Sprite-LFS

Sprite-LFS was the first prototype implementation of a log-structured file system. In Sprite-LFS the solution for ensuring that there are large extents of free space available is based on large extents called segments, where a segment cleaner process continually regenerates empty segments by compressing the live data from heavily fragmented segments[Ros91].

The authors claimed that Sprite-LFS could use 70% of the disk bandwidth for writing, whereas Unix file systems could use typically only 5-10%. All of the benchmarks in the paper are performed without the cleaning overhead and represent the best-case scenario.

Sprite-LFS was designed to use the technological shift to higher capacities of memory and disks, while the performance of hard drives would not improve that much.

The Sprite-LFS authors focused on the efficiency of small-file accesses, later they found out, that Sprite-LFS techniques work as well for large files.

The basic structures like inodes and the super block in Sprite-LFS remained identical to those used in Unix FFS, but inodes were not stored in fixed locations, unlike in Unix FFS. Sprite-LFS uses a data structure called an inode map that keeps track of the inode locations.

Cleaning

Sprite LFS read segments into memory, identified data that were not removed a wrote them to a smaller number of segments.

Sprite-LFS got away with free block bitmaps, but needed to maintain a mapping from blocks to inode numbers in the so-called segment summary block.

Following questions about cleaning policies were defined:

- When should the cleaner execute? It could either run continuously in the background at low priority, or only at night, or when the disk is almost full.
- How many segments should be cleaned at once?
- Which segments should be cleaned? The obvious choice to pick up the most fragmented ones proved not to be the best choice.
- How should the blocks be written out? Sprite-LFS tries to enhance locality with sorting the blocks by their age when they were modified.

Sprite-LFS started the cleaning when the number of clean segments dropped below a predefined threshold value. It cleaned a few tens of segments at once.

Sprite-LFS used an algorithm based on cost and benefit. It differentiates older, slowly changing data from younger rapidly-changing data, and made cleaning decisions accordingly.

Crash recovery

Sprite-LFS uses checkpoints for crash recovery and roll-forward algorithms. A checkpoint defines a file system as it was at one point in time and roll-forward tries to recover as much data as possible since the last checkpoint.

Disk layout

In Sprite-LFS the disk is divided in same length segments. After all data are written to a segment the segment gets a checkpoint that is written to checkpoint areas that are on fixed positions. The checkpoint contains pointers to all meta-data that allow to identify directories, files and their content and to determine free and allocated blocks.

There are two checkpoint areas per segment and the file system writes to them alternatively noting the time of the write. In case of a crash the last written checkpoint can be identified by comparing the timestamps.

The checkpoint is written at periodical intervals or just before the file system is unmounted. The length of the checkpoint writing interval must be considered carefully, because if it is written to often, it has a negative performance impact or if it is written not so often, the roll-forward during recovery would take longer.

8.2.2 BSD-LFS

The second try on log-structured file system was BSD-LFS. It was a redesign of the Sprite-LFS[Sel93]. Although BSD-LFS had superior performance over FFS, in the meantime an enhanced version of FFS with read and write clustering had appeared. This FFS offered better performance than BSD-LFS, however, the LFS could be extended to provide some additional functionality like versioning that traditional file system could not.

Disk Layout

BSD-LFS borrowed much of its disk layout from FFS. It used an inode data structure to map a file to its block addresses in order to allow efficient random retrieval of files. It used also direct, indirect, doubly indirect, and triply indirect blocks. The writing was different in that BSD-LFS was log-structured file system and made all writes in the end of the single continuous log. The log was divided into fixed-sized segments. When data in the file system were updated, they were gathered, reordered and written to the next available segment[1]. Modification of data in the file system, inevitably also modified associated meta-data that had to be reallocated to the next segment. The previous segments end up with all kinds of holes with free disk space that can be reclaimed later by a cleaner. The fact that data and its associated meta-data are written to a new segment and their older representation still exists on the disk is called no-overwrite policy.

Ideally a whole segment is written at once, once there are enough dirty blocks in the memory. Usually a write must be performed, even if there are not enough data, so partial segments are written. One segment can hold one or more partial segments.

Additionally BSD-LFS used a super block similar to the one used in FFS, to describe the file system as a whole.

[1]In the meantime this is also available in Sprite-LFS

Differences from Sprite-LFS

BSD-LFS was based on the logical framework of Sprite-LFS, but addressed some of the Sprite-LFS shortcomings. Among them:

- BSD-LFS used less memory than Sprite-LFS.
- Write requests were successful in BSD-LFS even if there is insufficient disk space at the moment
- Additional verification of the file system directory structure during recovery.
- Segment validation in Sprite-LFS assumes that there is no write reordering of blocks by hardware.
- Sprite-LFS had cleaner in the kernel space, which was moved to the user space in BSD-LFS
- In Sprite-LFS paper was no performance comparison with cleaner.

BSD-LFS kept the segment log structure, the inode map, segment usage table and cleaner. The cleaner was moved to user space, so that different cleaning policies could be tested and used. Sprite-LFS maintained a count of free blocks for writing. This number was decremented when blocks were actually synced to the disk. BSD-LFS used two forms of accounting. The one similar to the Sprite-LFS, but also was decremented and incremented when the change happened only in the cache. The second form of accounting kept track of how much space is available for writing. It was also decremented when a dirty block enters the cache, but it was not reclaim until it was cleaned by cleaner. This was to ensure that when a block is accepted for writing it will be eventually written to the disk.

Sprite-LFS assumed that the order in which the writing requests are placed on the disk is the actual order in which the blocks are written to the disk. Placing the segment summary block at the end of the segment was supposed to ensure that it was the last block written in the segment and thus the whole segment is in the consistent state. Since disk controllers can reorder the write requests this assumption does not hold and BSD-LFS fixes it, in that it uses checksums of partial segments to enable to identify the valid and invalid partial segments.

File System Recovery

BSD-LFS provided two phases for file system recovery. The first phase examines all the data written between last checkpoint and the failure. The second phase is a complete consistency check much like the FFS file system. To recover the BSD-LFS after a crash only the first phase is required which is very fast. The second phase can be run in the background while the file system is used. In an unlikely event of a failure in the second phase of the file system check, the file system had to be remounted read-only, the problem was fixed and the file system was remounted again read and write.

The Cleaner

The BSD-LFS cleaner was implemented in the user space, using system calls to communicate with the file system and using an ifile to get information required for cleaning. It was possible to use more than one cleaner with different cleaning policies. One cleaner that was implemented was based on the cost-benefit computation. Still some scenarios caused high performance degradation. BSD-LFS could utilize a large fraction of the disk bandwidth for writing, but the cleaner had a severe impact in certain workloads, particularly transaction processing.

8.2.3 Linlog FS

Linlog FS from Christian Czezatke was a log-structured file system designed for clones/snapshot functionality and personalities[Cze98]. The stopper was freeing of blocks, the cleaner, that did not work efficiently. Linlog FS was created for Linux 2.0, later ported to Linux 2.2, but work on it has been stopped.

LLFS is similar to Linlog FS in its goals but does not need the cleaner, because it has free-blocks-bitmap which makes allocating and freeing of blocks easy.

8.3 Linux File Systems

There are many file systems available in Linux. They were implemented specially for Linux or were ported from other operating systems. The first file system used in Linux was MinixFS, followed by Ext and Ext2. ReiserFS was the first file system on Linux to offer journaling. The most popular file systems in Linux are Ext3, XFS, and ReiserFS. They are all the journaling file systems.

8.3.1 Ext2

The Ext2 file system does not offer journaling and fast crash recovery, but is the oldest useful file system coming with the Linux kernel. It is the most portable, best tested and understood file system. It is often used to test new enhancements. It also helps that it corresponds almost one to one to the Linux Virtual File System. LLFS also started from the Ext2 file system and uses its allocation methods.

8.3.2 Ext3

Ext3 has a journaling implemented on top of the Ext2 file system. This allows for fast crash recovery. Ext3 has an advantage that it can be mounted as Ext2.

Ext3, unlike the other Linux journaling file systems, can log data blocks along with meta-data at a performance penalty because data blocks have to be written twice. This is turned off by default.

8.3.3 ReiserFS

ReiserFS uses fast balanced trees[2] to store file system meta-data. ReiserFS until version 4 provides only meta-data journaling. ReiserFS saves disk space with *Tail packing*. Small files and tails of the larger files that are smaller than a disk block, are stored together in one block unlike other file systems that leave that space unused. Generally, this allows a ReiserFS to hold around 5% more than an equivalent ext2 file system, but with performance penalty[Gal01]. This way internal fragmentation is kept low, but external fragmentation is higher, because the file *tails* can be further away from other file data.

Reiser4 uses LFS techniques (called Wandering Logs in Reiser4) to achieve very good consistency guarantees (it allows full transactions via file system plug-ins), although it is not completely clear if it gives the in-order semantics consistency guarantee. However, Reiser4 does not offer snapshots or clones. It is also a much more complex file system, and thus probably less amenable to studying variations in the design decisions.

LLFS does not provide the *Tail packing* feature, because I do not consider the performance penalty justified.

[2]B+Tree

8.3.4 XFS

XFS is a journaling file system from SGI that was ported to the Linux platform. The purpose of XFS is to optimize accessing of very large files. This is achieved with large extents and small number of descriptors required. XFS provides the fast crash recovery. It logs meta-data changes but does not log user data changes. With that XFS does not ensure full data consistency.

Chapter 9

Further Work

LLFS is proof of concept implementation and although it is possible to use it, it is still far from being a production level file system. Along with some stability issues and possible performance optimizations some further work has to be done.

LLFS supports up to 100 clones. It is possible to increase number of available clones, if one entire block or two is used for clone pointers. This could increase the number of clones by 1024 or multiple of that using 4K blocks.

The mounting of clones should be improved as well. Right now it is possible to mount different clones on predefined mount points. It is imaginable that mount command could have an option that would specify the clone that should be mounted. Another possibility is to map different clones on different device files and no new mount option would be required. It would also solve nicely the inode and dentry cache workarounds.

Searching for free blocks could be sped up, if there was a cache in memory with free block bitmap that holds information from all the clones. This would also improve the check if a block group does not have any free blocks available and the counting of free blocks. The current free block count gives only an approximation of the reality.

After a crash or power failure the file system is recovered to the state as it was after the last commit. With that a point-in-time data-consistency is ensured. Log-structured file systems implement additional roll forward to save as much data as possible. Maybe LLFS could also have something like that.

Quotas is a feature that I did not implement yet and that should be implemented as well.

Chapter 10

Conclusions

A copy-on-write file system has many advantages. It makes point-in-time recovery possible, where all the data and meta-data can be consistent after a system crash or power failure. It is also possible to create clones and snapshots in more efficient way than is currently available.

My first implementation named LLFS created for this thesis showed that it is indeed possible to implement a copy-on-write file system that supports clones, snapshots and data consistency and its performance is on par and in some cases better than journaling file systems that offer lesser consistency guarantees.

However, there is still much to do in LLFS to become accepted stable Linux file system. The code should be reviewed, improved and optimized. All the temporary solutions should be ironed out. The LLFS performance could be further improved to get closer to the Ext2 file system from which LLFS was derived.

The user tools should be programmed beyond the rudimentary level they are now.

Although there are other file systems with similar functionality appearing all over the place and are in the time of writing under hectic development, the LLFS offers some unique solutions that may prove to be the right ones in the future.

Bibliography

[Cze98] Christian Czezatke: *dtfs, A Log-Structured Filesytem For Linux,* Diplomarbeit, TU Wien, 1998

[Cze00] C. Czezatke, A. Ertl: *LinLogFS – A Log-Structured Filesystem For Linux,* Freenix Track of Usenix Annual Technical Conference, 2000, p. 77-88.

[Gal01] Ricardo Galli: *Journal File Systems in Linux,* Upgrade, The European Online Magazine for the IT Professional, Vol.II, Issue no. 6, December 2001, p. 50.

[Lov04] Robert Love: *Linux Kernel Development,* A practical guide to the design and implementation of the Linux kernel, Sams Publishing, 2004

[Ous88] J. Ousterhout, F. Douglis: *Beating the I/O Bottleneck: A Case for Log-Structured File Systems* 1988 Technical Report # UCB/CSD 88/467, Univ. of California, Berkeley.

[Ros91] M. Rosenblum, J. K. Ousterhout: *The Design and Implementation of a Log-Structured File System,* ACM Transactions on Computer Systems, volume 10, issue 1, 1992, p. 26 - 52

[Sel93] M. I. Seltzer, K. Bostic, M. K. McKusick, C. Staelin: *An Implementation of a Log-Structured File System for UNIX,* 1993 Winter USENIX Technical Conference, San Diego, CA, January 25-29, 1995.

[Sel95] M. I. Seltzer, K. A. Smith: *File System Logging Versus Clustering: A Performance Comparison,* 1995 Winter USENIX Technical Conference, New Orleans, LA, January 1995, p. 249-264.

[Sel00] M. I. Seltzer, G. R. Ganger, M. K. McKusick, K. A. Smith, C. A. N.Soules, C. A. Stein: *Journaling versus Soft Updates: Asynchronous Meta-data Protection in File Systems,* Proceedings of the

2000 USENIX Technical Conference, San Diego, CA, June 2000, p. 71-84.