Tomáš Hrubý, Martin Jambor, Kuba Krchák, Jan Tauš

Log Structured File System for Linux 2.6

Version 1.0

Compiled on November 8, 2006.

The log structured file system described in this document is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This file system is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this file system; if not, write to the Free Software Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

Contents

1 Introduction **5**
1.1 Log Structured File System 5
1.2 About This manual 6
1.2.1 Conventions used throughout this document 6

2 Installation **7**
2.1 System Requirements 7
2.2 Installation 7

3 User's Guide **9**
3.1 Quick Start 9
3.1.1 Step-by-step 9
3.1.2 Snapshot 10
3.2 Kernel & Patches 10
3.3 Managing LFS 10
3.3.1 Creating a file-system 11
3.3.2 Mounting a file-system 11
3.3.3 Mounting a snapshot 11
3.3.4 Debugging a file-system 11
3.4 Obtaining runtime information 12
3.5 Limitations 12

4 On-disc structures **15**
4.1 Data types 15
4.2 Characteristics common with traditional designs 16
4.3 Block indices 16
4.4 Segments 17
4.5 Ifile 17
4.5.1 Segment usage table 18
4.5.2 Inode table 18
4.5.3 Size constraints 18
4.5.4 Working with ifile 19
4.5.5 Synchronization 19
4.6 Live and dead entities and inode versions 19
4.7 Data segment layout 19
4.8 Partial segment layout 21
4.8.1 Blocks 22
4.8.2 Inodes 22
4.8.3 Journals 22
4.8.4 Finfos 23

5 Kernel Module Implementation Overview **25**
5.1 In-Memory Structures 25
5.1.1 Info structures 26
5.1.2 Inodes and indirect blocks in memory 26
5.1.3 Free space accounting and preallocation 27
5.2 Reading files 28

5.3 Writing data . . . 29
5.3.1 Tasks performed by the segment building code . . . 29
5.3.2 Stages of the segment building code . . . 29
5.3.3 Plans . . . 31
5.3.4 Synchronization . . . 33
5.3.5 Writing pages . . . 33
5.3.6 Writing inodes . . . 35
5.3.7 Segment finishing . . . 36
5.3.8 BIO finish callbacks . . . 37
5.3.9 Dealing with errors . . . 38
5.3.10 Garbage collector writes . . . 38
5.4 Syncing . . . 38
5.4.1 Flushing cache . . . 38
5.4.2 Planning a consistent ifile . . . 39
5.4.3 Building the sync plan . . . 39
5.4.4 Flushing the segment plan . . . 40
5.5 Fsync . . . 41
5.6 Ihash . . . 41
5.6.1 Hash table . . . 41
5.6.2 Use cases . . . 41
5.7 Truncate . . . 42
5.8 Segment management . . . 43
5.8.1 Providing the next segment . . . 43
5.8.2 Segment writeouts . . . 44
5.8.3 Safe reclaiming . . . 44
5.8.4 Managing the currently written segment . . . 45
5.9 Syncing thread . . . 46
5.10 Garbage collector . . . 46
5.10.1 Communication protocol . . . 46
5.10.2 Sending messages to the user space . . . 47
5.10.3 Processing cleaner requests . . . 47
5.11 Directories . . . 48
5.11.1 Overview . . . 49
5.11.2 Structures . . . 49
5.11.3 Index tree . . . 50
5.11.4 Resolve operation . . . 52
5.11.5 Adding directory entries . . . 53
5.11.6 Removing directory entries . . . 57
5.11.7 Symbolik links . . . 58
5.11.8 Journalling . . . 58
5.11.9 Readdir . . . 59
5.12 Journaling subsystem . . . 62
5.12.1 Interface for directory operations . . . 62
5.12.2 Interface for the segment builder . . . 63

6 Snapshot 65
6.1 Implementation . . . 65
6.2 Free segments tracking . . . 65
6.3 Free space accounting . . . 65
6.3.1 Inodes . . . 66
6.3.2 Blocks . . . 66
6.4 Working segment . . . 66
6.5 References . . . 67

7 Utilities Implementation Overview **69**
7.1 mkfs.lfs 69
7.1.1 Initiation 69
7.1.2 Write functions design 70
7.1.3 Root directory 70
7.1.4 Ifile 71
7.1.5 Non-data segments 72
7.2 Garbage collector - user space part 72
7.2.1 Overview 72
7.2.2 Source code 73
7.2.3 Most important functions 73
7.2.4 Decision process 73
7.2.5 Logging 74

A Manual pages **75**

B The GNU General Public License **77**

Chapter 1

Introduction

1.1 Log Structured File System

Just about any basic course on operating systems that covers file systems at least mentions some kind of a log structured file system as an example of a less traditional disk layout. Log structured file systems have proved to perform very well under metadata intensive workloads and offer features unknown to ordinary file systems. On the other hand, even though Linux boasts support of a wide range of file storage systems, none of them has been log structured until this day. This document describes a brand new implementation of a log structured file system for Linux 2.6.17 (LFS).

Log structured file systems have been around for a long time. They have been proposed in 1991 by Mendel Rosenblum and John K. Ousterhout [1] who described them in the following way:

> "A log-structured file system writes all modifications to disk sequentially in a log-like structure, thereby speeding up both file writing and crash recovery. The log is the only structure on disk; it contains indexing information so that files can be read back from the log efficiently."

The proposed scheme also included division of the underlying device into segments and garbage collecting truncated and outdated data together with an optimal garbage collecting strategy based on a cost-benefit analysis.

In 1993, Margo Seltzer, Keith Bostic, Marshall Kirk McKusick, and Carl Staelin picked up on this work and implemented a log structured file system that was fully integrated into a contemporary production UNIX operating system [2] and managed to overcome many of the shortcomings of the first prototype implementation. We have based our on-disk data structures on their work even though we have amended them to better suit the current Linux environment.

Since then there exist implementations of log structured file systems for old BSD UNIXes. Google reveals several attempts to implement a log structured file system for Linux but none of them has ever been finished. Only one of them, for example, has a working garbage collector, the rest of them are often unable to reclaim free space from outdated and deleted data. The one that has a garbage collector does not support mmap, often trashes cache for no good reason and has other serious shortcomings.

Log structured file systems were proposed because they perform very well under certain workloads. Moreover, fast crash recovery is a logical extension of such a file system and they make it possible to implement so called snapshots. By a snapshot we mean a read-only file system that represents the state of the original file system at a given moment. The original file system can then be modified but the snapshot still has the same contents. This is useful particularly for backing up consistent data while the file system itself is mounted and stored information is being changed.

Even though there is still room for improvement, we have managed to put together a working and competitive file system that:

- takes full advantage of Linux 2.6 page cache,

- has working garbage collector,
- uses sophisticated data structures for large directories that considerably speed up directory operations,
- implements snapshots as described above,
- is capable to recover from a system crash.

Our file system still has a few shortcomings though. Because of the time frame allocated to us and hardware that we had at our disposal, we were only able to thoroughly test a limited number of configurations. For example, even though there is no reason why the file system should not work on an architecture different from i386 and extra care has been taken to provide for big endian architectures, no tests on other architectures were performed simply because we did not have access to the hardware. Furthermore, the number of configurations that could potentially be achieved by tuning the size of blocks and segments would require lots of months of thorough testing that we did not have. Therefore, these parameters are fixed at the moment even though internally the system is ready for a wide range of values.

1.2 About This manual

Following this introductory text, the second chapter deals with setting up the file system. It covers necessary requirements, patching the kernel, compiling the source code and installing the file system onto your computer. The third chapter describes the file system from the user point of view. It covers the utilities that come along with it and explains how you can obtain information about mounted file systems. The fourth chapter describes LFS on-disk structures, the fifth one explains the internals of the kernel module that implements the file system and the sixth presents an overview of the implementation of snapshots. The seventh chapter deals with implementation of user space utilities. Finally, we have included our manual pages and the GNU licence terms and conditions that apply to this project.

1.2.1 Conventions used throughout this document

There are not many typographical conventions in this document, nevertheless it is probably necessary to say that especially important parts of the text are printed in **bold**, technical terms are highlighted in an *italic font*, identifiers and other text appearing in the source code are typeset using a `monospace font`, just like the input and output of various utilities displayed in a terminal.

Chapter 2

Installation

2.1 System Requirements

- Vanilla kernel 2.6.17.8 with applied lfs-patch as described in next section.
  - Large block devices (LBD) are supported
  - Large files are supported
- Recent version of glibc library is required (e.g., 2.3.6 from Debian Sarge or current 2.4). This is not important for compiling file-system but for using it. Some older libraries (e.g., 2.0.6 from 1997) contained implementation of `readdir()` that can return some directory entries multiple times in case of large directories. This behavior was also observed using `ext3` file-system with indexed directories.

2.2 Installation

LFS has been developed and tested with vanilla kernel 2.6.17.

1. Apply path to configured kernel

   ```
   $ cd /usr/src/linux-2.6.17.8/
   $ patch -p1 <<path_to_lfs_instaltion/patches/lfs-2.6.17.8-1.patch
   ```

   For your connvenience, a patched kernel is also included on the CD. Compile your kernel as usually.

2. Change to the root directory of LFS distribution. Type:

   ```
   $ make
   ```

3. Install LFS by typing

   ```
   $ make install
   ```

   This will copy the garbage collector to `/sbin/lfs-gc`, `mkfs.lfs` to `/sbin/mkfs.lfs`, `fsck.lfs` to `/sbin/fsck.lfs`, `dump.lfs` to `/sbin/dump.lfs` and `liblfs.so` to `/lib/liblfs.so`. It will also install the kernel module to the appropriate directory.

4. Optionally, copy man-pages from `doc/man` to your distribution's man path[1].

5. Load LFS module by:

   ```
   $ modprobe lfs
   ```

LFS is now installed and ready for use. You can proceed to **Quick Start** in next chapter

[1] `/usr/share/man/man8` on Debian

Chapter 3

User's Guide

3.1 Quick Start

LFS is a standard Linux file-system. It is mounted as any other file-system and standard utilities like `mkfs` for managing it are provided. The following table presents a list of all support programs and their counterparts for the ext2 file-system for the convenience.

program	**ext2**	**description**
mkfs.lfs	mke2fs	Creates an empty file-system on a disk partition and fills it with an empty root directory.
lfs-gc	—	Garbage-collector that frees unused space. It is started automatically in the mount time.

3.1.1 Step-by-step

- First of all you need to create a partition on a disk. You can use the `fdisk` program. The partition should be at least 100MB large because smaller file-systems are not supported[1].
- After partitioning you have to create an empty file-system. For this purpose you have to use the `mkfs.lfs` program. In this example we use the default settings. Once a new partition has been created on `/dev/hdb1` `mkfs.lfs` output looks like the following:

```
# mkfs.lfs /dev/hdb1
mkfs.lfs: version = 1.0
mkfs.lfs: device         = /dev/hdb1
mkfs.lfs: volume name    = LFS
mkfs.lfs: device size    = 987966 kB (0.94 GB)
mkfs.lfs: block size     = 4 kB
mkfs.lfs: segment size   = 1024 kB = 256 blocks
mkfs.lfs: reserved space = 1024 kB = 256 blocks (1 segments)
writing data.....

................................
ifile has 15472 bytes
addresses per indirect block : 512
writing 0. super block at 0x0000000001000000 (checksum is
0xb29a8d3b) ... OK
writing 1. super block at 0x000000000fc00000 (checksum is
0xb29a8d3b) ... OK
```

[1] Working with `fdisk` is out of scope of this document. See `man fdisk` for full usage documentation

```
writing 2. super block at 0x000000001e800000 (checksum is
0xb29a8d3b) ... OK
writing 3. super block at 0x000000002d400000 (checksum is
0xb29a8d3b) ... OK
syncing ...
LFS successfully created on /dev/hdb1
```

- Finally, the file-system needs to be mounted (using `mount`) to some directory in your file-system. If you want to attach LFS to `/mnt/lfs` directory, the following command does the job :

  ```
  # mount -t lfs /dev/hdb1 /mnt/lfs
  ```

After successfully completing all steps, LFS can be used.

3.1.2 Snapshot

Snapshot is a special ability of LFS that has no counterpart in **ext2**-like file-systems. Sometime you want to make copy of frequently updated data or simply just to read them. The problem is that some data are changing to often and your copy would not be consistent. This problem is often encountered when one makes a backup copy. Solution is to stop all writing, so data are not change, while the copy is being made. Of course this solution has its disadvantages.

Solution provided by LFS is simple. Just mount file-system once more in a special **snapshot-mode**. Snapshot contains state of the whole file-system in the time of mount. It will never change even if the underlying LFS file-system does.

To take a snapshot you have to know an `id` of the mounted system. Because one file-system can be mounted to multiple places with different options we do not use a path (mount-point) as the identifier. Rather each LFS file-system has its own special id that can be used to reference it.

If you do not provide an `id` when mounting the file-system, an unused `id` is generated. You can check the `/sys/filesystems/lfs/` directory for all LFS file-systems in use. Each mounted LFS has its own subdirectory, which name is equal to its `id` Files in this directory export various runtime data – mainly for debugging purposes. Generated `id`s have the format `lfs%d`. Identifier `lfs0` is assigned to the first mounted LFS file-system.

Once you know the `id` of a file-system you can mount a snapshot, e.g. to `/mnt/snap`, using command:

```
#mount -t lfs-snap lfs0 /mnt/snap
```

To close snapshot just umount the snapshot file-system by:

```
#umount /mnt/snap
```

3.2 Kernel & Patches

LFS is developed for a patched 2.6.17.8 kernel. Patch is included in the distribution archive. Patch contains:

- Export of symbols of few kernel functions.
- Patch by David Howells <dhowels@redhat.com> which allows file-systems to register callback for handling page-faults. LFS can track when mmaped data are dirtied[2]

3.3 Managing lfs

All programs described in this section have their manual pages that are included in lfs distribution archive. You can read them in the Appendix A.

[2] Why LFS must know about dirtied pages is described in Section 5.1.3

3.3.1 Creating a file-system

Prior to mounting a file-system, it must be created on a device using the `mkfs.lfs` utility. Program first checks the size of the supplied device, it checks whether the device is large enough (the lower limit is 100MB) and prints out the file-system parameters (device size, segment size, block size, etc.)

`mkfs.lfs` cleans all segment summaries before any other data are written to the disk. This can take a while, depending on the device size. Program signals that the process wasn't finished yet by printing dots.

In the next stage, `.ifile` and the super blocks are written. After this, all the data must be written back from caches to the device. Since cleaning segment summaries means writing a lot of bytes for a large disks, flushing cashes takes some time as well.

Finally, message `LFS successfully created on <device>` signals that the file-system was successfully created and is ready to be used.

3.3.2 Mounting a file-system

File-system is mounted using standard **UNIX** `mount` utility. If the LFS module is not loaded you must add a type parameter `-t lfs` and module will be loaded automatically[3]. If module is already loaded you need not specify its type. In that case kernel detects the type itself. LFS supports standard mount options (e.g., `ro`, `rw`, `noatime` ...) together with LFS specific ones:

- `id=<lfs-id>` sets id of mounted file-system. This can later be used for its identification. Without this option identifier is generated automatically (`lfs0`, `lfs1` ...).
- `gc=<lfs-gc>` specifies program which will be run as a garbage collector.

If you do not specify `noatime` option it will be added. LFS cannot be mounted without this option.

3.3.3 Mounting a snapshot

Snapshot is mounted as a special read-only file-system. Its type is `lfs-snap`. Original file-system is synced during mounting the snapshot. Snapshot represents the LFS state after that sync. Snapshot file-system doesn't support any additional options.

The **device** used to mount a snapshot is always an `id` of the original file-system.

Once a snapshot is mounted all included data is still remaining on the disk. This means that if you delete a file from the *live* system, the space it was using is not freed until the snapshot is umounted. When you edit the file created before a snapshot was taken, a new version of this file will be stored on the disk along with the old one, consuming additional space. Notice that if a snapshot is mounted and the *live* file-system is being concurrently modified, all new data takes extra space even if old data is overwritten. On the other hand appending extra data to a file will not double the whole size of the file. Snapshot acts in the copy-on-write fashion.

After umount all obsolete (deleted or modified) data is freed and free space on disk increases.

3.3.4 Debugging a file-system

If you want to see on-disk data structures you can use `dump.fs` program. It shows super-blocks, inodes, segments, etc. To fully understand its output you read Chapter 4 **On-disk Structures**. `dump.lfs` has many options described in its manual page (see Appendix A). Following example shows how a super-block can be displayed :

```
$ dump.lfs /dev/hdb1 sb
fs.sb.@address                    = 0x01000000
fs.sb.s_segment_count             = 101
```

[3] Autoloading must be enabled in kernel.

```
fs.sb.s_log_blocks_per_seg          = 8 (256)
fs.sb.s_reserved_segment            = 1
fs.sb.s_free_blocks_count           = 25600
fs.sb.s_free_seg_count              = 93
fs.sb.s_first_free_inode            = 3
fs.sb.s_used_inodes                 = 3
fs.sb.s_log_block_size              = 12 (4 KiB (4096 bytes))
fs.sb.s_ifile_addr                  = 0x00406000
fs.sb.s_next_block                  = 0x408
fs.sb.s_segment_counter             = 5
fs.sb.s_mtime                       = Sun Aug 20 18:26:56 2006
fs.sb.s_wtime                       = Sun Aug 20 18:29:50 2006
fs.sb.s_mnt_count                   = 0
fs.sb.s_max_mnt_count               = 0
fs.sb.s_magic                       = 0x1234
fs.sb.s_state                       = 0x00000000
fs.sb.s_errors                      = 0x00000000
fs.sb.s_minor_rev_level             = 0
fs.sb.s_lastcheck                   = Sun Aug 20 18:26:56 2006
fs.sb.s_checkinterval               = 0
fs.sb.s_rev_level                   = 0
fs.sb.s_feature_compat              = 0x00000000
fs.sb.s_feature_incompat            = 0x00000000
fs.sb.s_feature_ro_compat           = 0x00000000
fs.sb.s_volume_name                 = "LFS"
fs.sb.s_checksum                    = 0x02d4e8f7
```

3.4 Obtaining runtime information

Mounted LFS file system provide a a lot of of information about its internal state, which can be read through **sysfs**. There is a subdirectory for each mounted instance in `/sys/filesystems/lfs`. A comprehensive overview of all the files is shown in table 3.1. Please note that values prefixed with `stat_` are not present if the LFS module is compiled without statistical data.

3.5 Limitations

- `O_DIRECT` — opening a file in the direct write mode is disabled. Direct operations does not comply with the LFS nature.
- `noatime` — this flag is forced when mounting LFS because changing access time would result in frequent writes and a great performance loss.
- `x86` — file-system was developed and tested only on **Intel 32bit** platform. Even though LFS was designed as architecture independent, it was never tested elsewher so its functionality cannot be assured.

File	**Description**
`seg_count`	file-system size in segments
`seg_size`	segment size in blocks
`block_size`	block size in bytes
`seg_counter`	actual segment counter value
`free_space_max`	maximal free space
`free_space`	actual free space
`free_space_delay`	amount of bytes, which will be added to free space after umounting the snapshot
`free_segs`	actual number of free segments
`free_segs_max`	maximal number of free segments
`inodes_alloc`	allocated inodes in the ifile
`inodes_used`	used inodes
`stat_comm_segs`	number of successfully written segments since the last mount
`stat_comm_syncs`	number of successfully written sync points since the last mount
`stat_wq_len`	length of the garbage collector work queue
`stat_wq_req`	number of request messages in the gc queue
`stat_wq_inf`	number of info messages in the gc queue
`stat_gq_len`	length of the garbage queue
`stat_gq_mapping`	number of mappings in the garbage queue
`stat_gq_inode`	number of inodes in the garbage queue
`stat_gc_send`	number of info messages send to gc
`stat_gc_recv`	number of gc requests received

Table 3.1: Information exported via `sysfs`

Chapter 4

On-disc structures

This chapter describes the on-disk structures we used in our implementation of a log-structured file system. Please note that we assume the reader is familiar with both traditional UNIX file system layout (that is explained for example in [3]) and general concepts of log structured filesystems as they are described in [1] and [2]. Please read through at least these two papers if you have not done so already. We have based our work particularly on the latter and won't go into much detail in cases when our implementation is almost the same.

All on-disk structures described in this section are defined in file `include/lfs_fs.h`. Please refer to it for exact definitions, comments and details.

4.1 Data types

Before we proceed to various structures present on the device, it essential to spend a number paragraphs with a few basic but fundamental design decisions and basic types. First and foremost, all multi-byte integers on the disk are stored in **little-endian format**. It is therefore necessary to use appropriate conversions whenever accessing any such integer that ever reaches the disk. Secondly, all structures we store on the disk are packed so that the compiler does not include any padding and thus they have the same binary representation on all platforms. This also means that the programmer is responsible for word alignment of individual items of these structures.

The current source code often requires that size of a particular structure must be a power of two. In this document we try to explicitly warn about these requirements whenever describing such a structure but it is necessary to double check before changing any on-disk structure whatsoever. Sometimes there are other requirements, for example two structures found in the ifile must have the same size. In other words, changing the on-disk structures is a delicate task that must be done thoughtfully.

Finally, this is an overview of the basic types specific to LFS that are used in on-disk structures:

`lfs_addr_t` is an address on the disk from the beginning of the device in blocks (thus it must be multiplied by the block size in order to get the address in bytes). Not surprisingly, it is used to store addresses of blocks.

`lfs_inode_addr_t` is an address on the disk from the beginning of the device in bytes. However, as the name suggests, it is only used to specify locations of inodes on the disk.

`lfs_finfo_block_t` is a logical index of a block within its file or indirect block ordering (see section 4.3) in blocks. It is used in finfo structures to identify blocks within a segment.

All types described above are internally 64-bit unsigned integers and thus impose no practical limit on the device or file size.

4.2 Characteristics common with traditional designs

We assume the reader is familiar with concepts of traditional file systems in UNIX environments (which are described for example in [3]) and therefore we will not describe those that LFS uses as well in much detail. However, let us at least briefly mention the most important ones.

Most file systems divide the underlying device into smallest addressable and indivisible units called blocks and so do we. Even though some of our metadata can share blocks (for example inodes can), files in particular are always internally accessed with block granularity. At this stage, our blocks are 4096 bytes long (see limitations on page 6).

Most importantly, we also use the traditional indexing structure of UNIX file systems, namely the inode and indirect blocks. However, indirect blocks are numbered as described in the next section. LFS also has superblocks that represent data global to the file system and are the only structures that are not written in a log-like manner. The last common concept shared with traditional file systems is that directories are represented with files with a given internal format and special access methods.

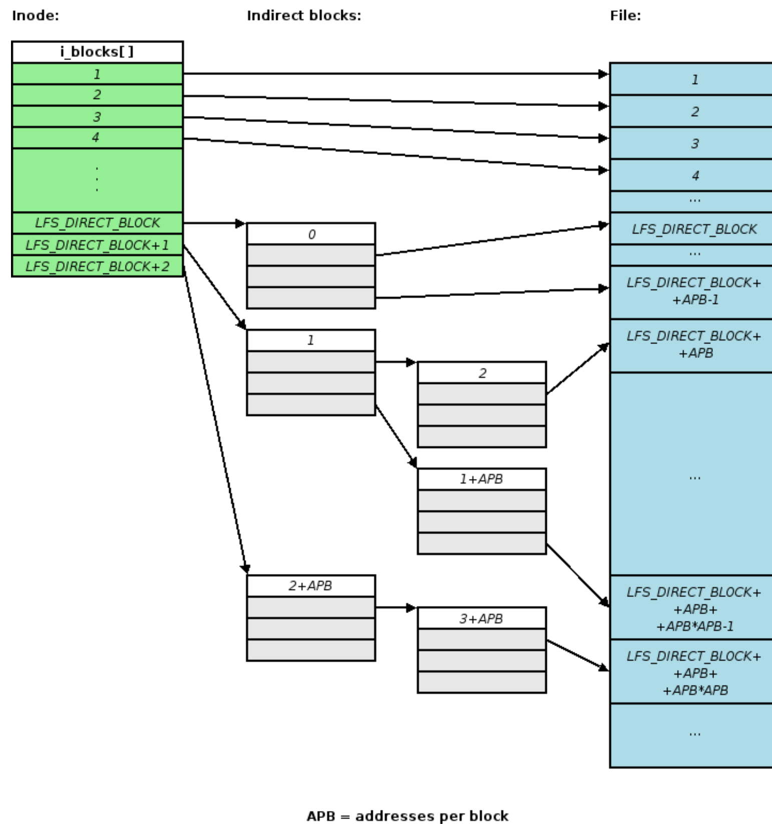

Figure 4.1: Block indices.

4.3 Block indices

Both indirect and data blocks are assigned an index that uniquely identifies them within an inode. The index of a data block is simply its offset in bytes divided by the block size. Indirect blocks are numbered in a pre-order depth first search manner. The figure 4.1 describes exactly how this is done.

4.4 Segments

All log structured file systems we know of statically partition the disk into **segments** of the same and fixed size. [4] discusses the impact of various segment sizes and recommends $4 * AccessTime * TrasferRate$ of the disk. Given today's disks, the recommended value usually varies in between one and two megabytes. In order to diminish the cleaning overhead and to limit the number of possible configurations (see limitations on page 6), the current version of LFS uses segments one megabyte long.

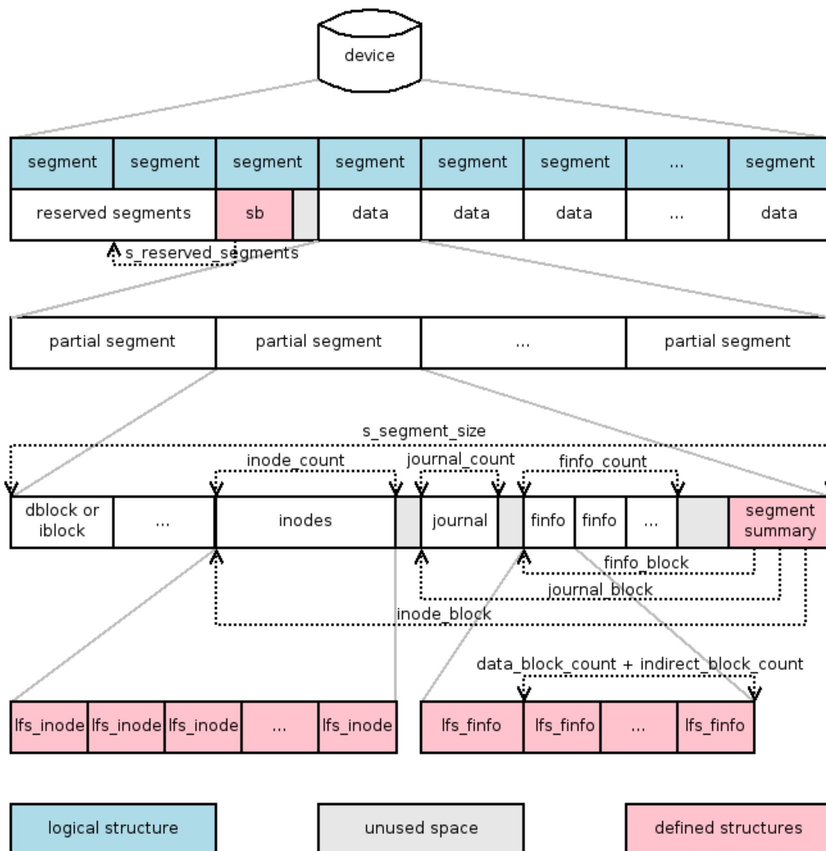

Figure 4.2: Disk structure overview

There are three main types of segments (see figure 4.2):

Reserved segment is a segment in the beginning of the device that is never touched by the file system. Usually there is one such segment on the device that can be used for whatever purpose the user chooses.

Superblock segment is a segment that contains a superblock. There are four superblocks on each LFS device and thus four superblock segments. Superblock segments do not contain any other data apart from the superblock.

Data segment is an ordinary segment that contains user data and file system metadata. We will deal with these segments in more detail in the section 4.7 on page 19.

4.5 Ifile

Every log structured file system needs a means of tracking state of individual segments and inodes. Whereas Sprite-LFS [1] used a special kernel table to store this information, BSD-LFS [2] moved both to an immutable file called **ifile** and we have adopted

the same approach. Ifile in our implementation of LFS consists of two parts and its structure is depicted in the figure 4.3.

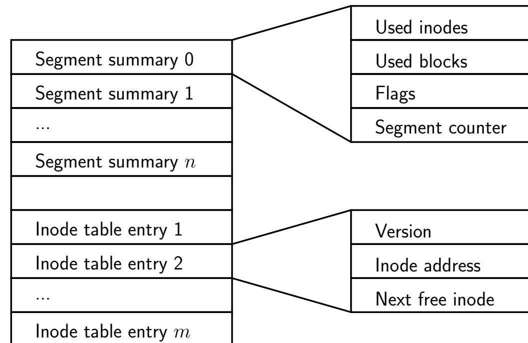

Figure 4.3: Ifile: The ifile consists of segment usage table and inode table. The size of the former does not change whereas the latter grows as the number of inodes on the file system increases.

4.5.1 Segment usage table

The first part of the ifile is called **segment usage table**. This table is created by `mkfs` and has a fixed size for the entire life of a particular file system instance. It contains an entry for each segment with information about how much live data and inodes the segment contains, what is its current logical sequential number or segment counter (see section 4.7) and flags to distinguish in between various types and states of segments. This part of ifile is especially important for the garbage collecting and segment allocating subsystems described by sections. . .

4.5.2 Inode table

The second part of the ifile is an **inode table**. This table grows as inodes are created in the file system. Every inode is described by an item in this table which enables us to:

- locate the inode on the device,
- check whether a particular inode on the disk is still alive (see section 4.6) and
- quickly delete and create new inodes.

When an inode is deleted (its nlink and reference counter both reach zero), the address in the corresponding inode table entry is set to zero and the item is prepended to a linked list of free items. When a new inode is requested, an item is taken out of the linked list and used. If there is no unused item, a new one is created at the end of the ifile and initialized. In both cases, the version of this inode number is incremented (see section 4.6 for explanation of versions). The inode table never shrinks but so far no one has considered this a problem.

4.5.3 Size constraints

Obviously, since it is likely that the number of items in both segment usage table and inode table will be very large, it is important to keep their items small. Currently, both structures are 16 bytes long. **The current code depends on the fact that both structures have the same size which is a power of two.** This requirement greatly reduces complexity and must be observed when modifying ifile internals.

4.5.4 Working with ifile

When accessing various parts of the ifile from kernel, the programmer should use the appropriate routines in ifile.c. A very brief summary of the four most important are given in table 4.1. Please refer to the source code and relevant comments for more information. As a side note, a new inode is created by `lfs_ifile_new_ino()` and they are deleted by `lfs_ifile_free_ino()`.

Function	Description
`lfs_ifile_get_segusage()`	obtains a pointer to an entry in the segment usage table. Gets, maps and locks the relevant page.
`lfs_ifile_get_inode()`	fetches a pointer to an entry in the inode table. Gets, maps and locks the relevant page.
`lfs_ifile_put_page()`	unlocks, unmaps and puts the page returned by the two functions above. Has to be called when you are done with any of the items.

Table 4.1: Ifile access functions

4.5.5 Synchronization

Of course, it is essential that concurrent access to all parts of the ifile is avoided. This is done using page locks. Each item is simply protected by the page lock of the page it resides in. Functions listed in table 4.1 lock pages properly for you.

4.6 Live and dead entities and inode versions

Since log structured file systems store all data in a single log, the data is never written to the same position from which it was read or which it was written to before. The same stands also for inodes, indirect blocks and even for the ifile (see section 4.5). Naturally, only the chronologically last copy of such data contains the currently valid version. Such inodes and blocks will be referred to as **live**. On the other hand, all other copies are outdated at any given moment. Such copies are referred to as **dead** and are subject to garbage collection.

In order to collect garbage safely, it is important to be able to determine whether a particular block or inode is alive or not. Strictly speaking, an inode on the disk is alive if and only if the corresponding item of the inode table points to it. Similarly a block is alive if and only if it is being pointed to by either the inode or an indirect block. In order to speed the process of finding out the live status of an entity, log structured file systems introduce so called **inode versions**. An inode version is a number that is associated with an *inode number* and that is incremented every time the inode is truncated to zero or the inode number is recycled and assigned to a newly created inode. When it is known that a block or an inode structure on the disk have an inode version different from the current one, they can be safely considered dead (see figure 4.4). Versions of inodes are used when collecting data and indirect blocks in the same way, the benefit of using them can actually be much bigger because loading the inode or even a few indirect blocks can be avoided.

4.7 Data segment layout

Log structured file systems need to finish a segment when they finish their part of the *sync* syscall. Both Sprite-LFS [1] and BSD-LFS [2] use so called **partial segments** in order not to waste space by leaving the rest of the segment unused. Our implementation closes partial segments at a few more occasions (fsync, pre-sync, flushing excessive amounts of journals and so on) in a very similar way.

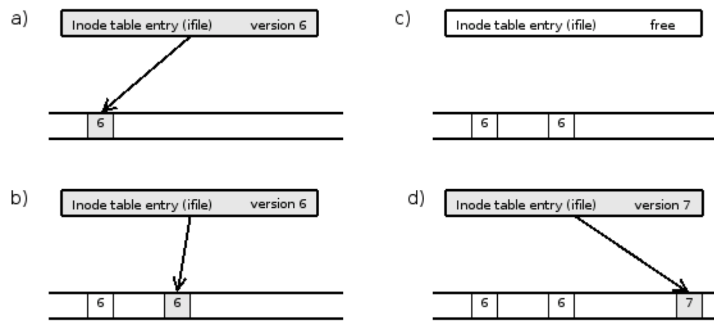

Figure 4.4: Inode versions. **a)** There is only one inode of a given number present on the disk, it has version six, is alive and pointed to by the inode table. **b)** An inode has changed and been written to the disk. A new copy with the same version is created and the corresponding pointer in the inode table is redirected to point at it. The previous inode is dead but to find this out, we must check where the live inode is. **c)** The inode has been deleted. Both inodes on the disk are dead because the inode table entry does not point anywhere. **d)** The inode number and the corresponding inode table entry have been recycled. Its version is incremented and the new inode has been written to the disk. The previous inodes are dead. This can be determined from their version because it is different from the version recorded in the inode table.

Throughout this text, unless we explicitly specify we refer to a partial segment, we mean the whole, *physical*, one megabyte long one. For example, the inode usage table in ifile has exactly one item for each physical segment, even if it contains multiple partial segments.

Every partial segment has a special structure called **segment summary** at the end. We will discuss particularities of partial segments and their summaries in the next chapter, at this point we only need to say that the summary contains the length of the partial segment. Moreover, the physical segment is always fully covered by partial segments, even if the last one is entirely empty. Such empty partial segment is called a **phantom segment** and is required to locate the previous partial segments. Thus, there is always a valid segment summary at the end of every physical segment (if it has ever been written to). Because every partial segment has its size recorded at a known place at the end and because we know the end of the last partial segment, all segment summaries are accessible and all partial segments identifiable. The figure 4.5 contains an example.

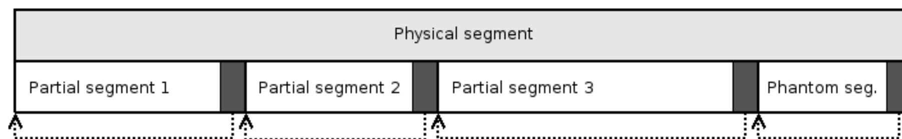

Figure 4.5: Partial Segments. The physical segment above contains four partial segments. Each of them has a segment summary represented by a dark grey box. In this case, the final partial segment does not contain any data. In other words, it is a phantom segment. Please note that the dotted lines are not actual pointers. These can be computed from the address of the segment summary (the end of the partial segment) and the size of the partial segment stored in the summary.

Even though segments form a log, this does not continuously grow from the beginning of the device towards the end. As data are overwritten and garbage collected, any free segment can be the next one in the log. Therefore, the log is actually formed

by a chronological or **logical** ordering of the segments. During a crash recovery, a roll forward utility must be able to follow the chain of segments in the order in which they were created by the file system module. Therefore, every segment summary contains the number of the segment that will be used next. Superblock contains the number of the next segment at the time of the end of a sync operation.

Needles to say, that is not enough. Only half-written segments cannot be recovered. Therefore, every partial segment summary contains a CRC of itself and of first 32 bytes of each sector that was written. Furthermore, if the next planned segment was not actually written to, an old segment could be mistaken for a new one. That is why every segment is assigned a global number that is incremented every time a new segment becomes active (i.e. the one being written to). This number is called the **segment counter**. Again, the superblock contains the counter of the next segment at the time of the last sync and every segment summary contains the counter of the next segment too. This leads to segments chained in a way similar to the figure 4.6. Moreover, the segment counter is the most important means of specifying time of various events in LFS. For example, it is necessary to know what the counter was when the last sync took place or a snapshot was mounted.

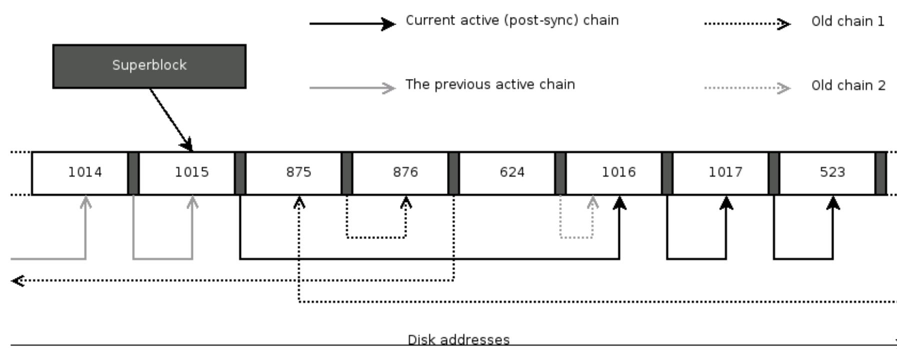

Figure 4.6: Segment chains. The figure contains a few consecutive segments on a disk which happened to form parts of four different chains. Most important is the currently active one. It begins at superblock and contains segments with counters 1015, 1016 and 1017. Even though segment 1017 points to segment 523, that is clearly an old segment because its counter does not immediately follow the previous one. The rest of the chains are not useful for roll-forward but demonstrate the behavior of the file system. The previous active chain starts outside of this picture, contains the segment 1014 and then suddenly becomes the current chain. The old chain 1 also has a beginning outside of the picture, contains segments 875 and 876 and leaves to some other part of the disk too. Finally the old chain 2 contains only segment with counter 624. Whichever segment once pointed to it has been garbage collected, reused, been assigned a new counter and became a part of a new chain. We can also see that the segment 1016 once used to be 625 but has been recycled too. Once the segment 624 is reused as well, the old chain 2 will be no more.

4.8 Partial segment layout

Partial segments contain data blocks, indirect blocks, inodes, journal lines, finfo structures and, as we already know, a segment summary. The figure 4.7 shows how these parts are ordered and located whereas the Figure 4.8 presents you the definition of segment summary. Please note that `journal_block`, `inodes_block` and `finfos_block` are addresses in blocks.

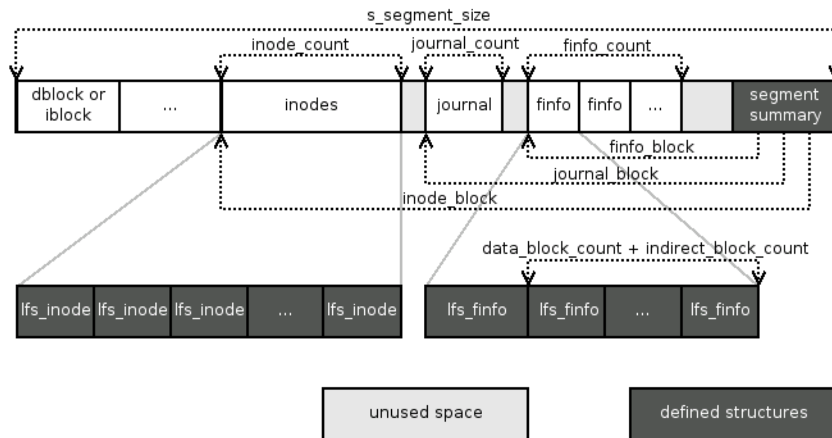

Figure 4.7: Partial segment structure. Please note that the unused space may not be present and may actually take up less than a block.

4.8.1 Blocks

If there are any indirect or data blocks present in a partial segment, they are placed one after another right from the beginning. As we have already stated in the previous section, the beginning of a partial segment can be determined by subtracting the partial segment size from the address of the end of the segment summary. Number of these blocks can be determined from *finfos* (see below).

4.8.2 Inodes

Inodes, if present, immediately follow the blocks. Currently one on-disk inode has size of 256 bytes, which means there are 16 inodes per one block. The address of the first block containing the inodes can be obtained from the `inodes_block` field of the segment summary which also contains the number of inodes stored in this segment in `inodes_count`. If this number is not divisible by 16, the rest of the last block is unused and cleared. Therefore one can traverse the inodes not by their count but until a structure filled with zeros is reached (as segment building code does).

Again, **it is important** to bear in mind **that the size of the on-disk inode is a power of two**. The current code depends on it and not observing this principle would break segment building.

4.8.3 Journals

The purpose of journal blocks is to inform the roll-forward utility about directory operations[1] These blocks contain so called **jline** structures that can span block and even segment boundaries but not across a sync. Each one describes an individual directory operation and its format differs according to the operation's type (see section 5.11.8 for details). The most important rule is that a jline describing a particular operation are saved no later than in the same partial segment that contains any part of the affected inodes. On the other hand, it can be stored to disk substantially earlier because jlines are never reordered and so they reach the disk exactly in the order in which they were carried out by the operating system.

When there are journal blocks in the partial segment, `journal_count` contains the number of blocks they span across and `journal_block` points to the first such block. The end of *jlines* is determined by reaching a cleared *jline* structure.

[1]Directory operations are file system operations that affect multiple inodes and either all or no changes must take place.

```
struct lfs_segment_summary {
        __u32   summary_checksum;
        __u32   data_checksum;
        __u32   segment_size;

        __u32   journal_count;
        __u32   inodes_count;
        __u32   finfos_count;

        __u64   journal_block;
        __u64   inodes_block;
        __u64   finfos_block;

        __u64   next_segment;
        __u64   segment_counter;

        __u32   create_time;
        __u32   flags;
} __attribute__((__packed__));
```

Figure 4.8: Segment summary

4.8.4 Finfos

When cleaning a segment, the garbage collector must be able to determine what individual blocks in a segment contain. That means to which inode they belong, whether they hold data or are indirect blocks and what is their index (as described in the section 4.3. This is accomplished using so called **finfo** structures. Take a look at its definition in the figure 4.9. It is a variable length structure consists of a header identifying the inode and a number of 64-bit unsigned integers containing indices. The field `indirect_block_count` tells how many indices of indirect blocks follow the header while `data_block_count` determines the number of offsets of direct blocks in the finfo. At the moment, only one of these numbers can be non-zero. `version` contains the version and `ino` the number of the inode the blocks belong to.

Finfos also have size constraints. **The Size of `struct lfs_finfo` must be divisible by the size of `lfs_finfo_block_t` which itself must be a power of two.**

```
typedef __u64 lfs_finfo_block_t;
struct lfs_finfo {
        __u32   indirect_block_count;
        __u32   data_block_count;
        __u32   version;
        __u32   ino;
        lfs_finfo_block_t   block[];
} __attribute__((__packed__));
```

Figure 4.9: Finfo structure definition

The finfo blocks correspond to the indirect and data blocks at the beginning of the partial segment. First index in first finfo identifies the first block in the partial segment, the second one refers to the second block and so on. After there are no indices left in the first finfo, the second one is used etc. See figure 4.10.

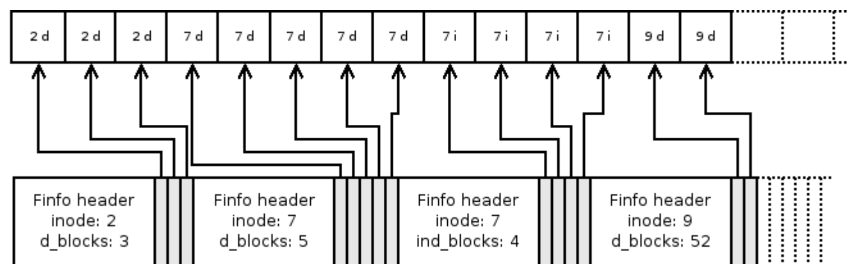

Figure 4.10: Finfos and corresponding blocks. Each grey rectangle is a logical offset of the corresponding block.

Chapter 5

Kernel Module Implementation Overview

This chapter describes the concepts behind implementation of LFS kernel module. Please keep in mind that comments in the source code are an integral and indivisible part of the documentation. This document covers the implementation from a global perspective only, it is the comments that deal with individual implementation aspects and technicalities and that really describe what a particular function or a piece of code does.

5.1 In-Memory Structures

This section briefly outlines the overall organization of in-memory structures specific to LFS. For information on general kernel structures, refer to the kernel source and kernel related literature such as [5]. Most of the in-memory structures are defined in `src/types.h` that is included from all c files. Nevertheless, there are other more specialized header files in the directory as well. All of them are described in table 5.1. Moreover, several c files define structures that are private to them. These will be dealt with in later sections where appropriate.

File	Description
`debug.h`	contains preprocessor macros used to implement and switch on and off various debug messages
`free_space.h`	contains definitions of structures internal to the garbage collector
`gc.h`	contains definitions of structures internal to the garbage collector
`indirect.h`	contains static inline functions for computations of indirect block indices
`segmap.h`	contains static inline functions for working with a segment map.
`snapshot.h`	contains definitions and declarations related to snapshot access and management.
`types.h`	is the most important header file with declarations of external functions and definitions of globally required macros and structures.

Table 5.1: Header files

From now on, the on-disk structures are also referred to as **raw** structures. For example, whereas by an *inode* we usually mean the in-memory representation of an entity

such as a file or a directory, the *raw inode* is used exclusively for a `struct lfs_inode` on the disk or its image in RAM.

5.1.1 Info structures

As customary in the world of Linux file system programming, every kernel *VFS* structure has its file system specific *info* structure and there are LFS info structures that do not have an obvious kernel counterpart. Thus, every `struct inode` of LFS is actually a part of a `struct lfs_inode_info` which is accessible using inline function `LFS_INODE()`. This info contains, among other things, an image of the raw inode on the disk.

Similarly, there is an info for the superblock as well. It is called `lfs_sb_info` and is a parameter to most of the functions in the LFS kernel module. It can be obtained from the kernel `struct superblock` by feeding it to `LFS_SBI()`. Most other LFS memory structures can be obtained from it, in one way or another. In general, the superblock info has three types of fields:

1. **General fields** such as references to the corresponding `struct superblock`, the raw superblock, buffers containing raw superblocks, the device, actual file system parameters and so on.
2. **Masks and shifts** for unit conversions and calculationg various offsets. For example, to obtain the number of blocks from a number of segments, the latter can be shifted left by `blks_per_seg_bits`. Similarly, to get a number of a block from the start of a segment from its address from the beginning of the disk, apply to it a bitwise *AND* with `blks_per_seg_mask`. There are a few such pairs and using them is preferred to computing the offsets and masks elsewhere.
3. **Various subinfos.** `struct lfs_sb_info` is a large structure and so it is partitioned into smaller info structures. Some are included directly, some are allocated separately and the superblock info contains only pointers. These can be thought of as subsystem private data and most subsystems have their global private data stored as an info accessible directly from here.

5.1.2 Inodes and indirect blocks in memory

Most other block device based Linux file systems use the device mapping to read, cache and store metadata including inodes and indirect blocks. When cache shrinks, this metadata is written to where they were read from by the device inode *address space operations*. Naturally this cannot be done in a file system that is log structured and therefore we have designed other means of keeping both entities in memory and writing them back to the disk.

Inodes

Raw inodes are stored inside each instance of the corresponding `struct lfs_inode_info`. Allocation and deallocation of inodes is done in the same way other file systems that use inode infos do it. The system asks us to allocate an inode by a call to the `alloc_inode` superblock operation. In response, we allocate whole inode info from a slab but return only the pointer to the *VFS* `struct inode`. Deallocation is done similarly and automatically by by standard means of shrinking the *dentry cache* and reference counting.

Indirect blocks

As stated above, we also need a special mechanism to cache and write indirect blocks. We therefore store them in pages of a special *address space* or *mapping* which we refer to as the **indirect mapping**. It is stored in each `struct lfs_inode_info` in field `idir_aspace`. An index of a page stored in this mapping is the index of the first block stored in it (See section 4.3, particularly figure 4.1). This mapping has *address space operations* similar to those of ordinary data mappings. The indirect blocks are thus read and written in almost the same way as data blocks.

5.1.3 Free space accounting and preallocation

LFS fully exploits the inode and page caches. Unlike traditional file systems it does not map data to specific locations on the disk at the time the user space creates a given entity but much later when the data are to be written back to the disk. Nevertheless, when LFS accepts any data from the user space, it is then commited to writing them to disk because once we have informed the user program data were accepted, there is no way to revoke that decision. That is why the file system can refuse to write a given entity only when servicing the syscall or mmap.

In particular, the file system must check there is a free spot for a new entity such as an inode or a file block whenever a user creates one. This mechanism is called **free space accounting**. The idea is simple. We track the amount of free space in the superblock info and decrease it with every new accepted file block or inode. Because log structured file systems perform poorly when their disk utilization exceeds 80% (see [1] and [4]) and because a certain proportion of the disk must be used to store metadata, we do not allow the user to use more than exactly 80% of the underlying device and react with *ENOSPC* to any attempt to do so.

It is also possible that even though there is enough of free space for a write operation to succeed, the system is in an imminent shortage of free segments. In that particular situation, the garbage collector must be informed there is an **emergency**[1] and all other processes must wait until it makes more segments available.

In order to do this we keep record of the total dirty pages and inodes in the system. When servicing a syscall that marks an inode or a page dirty we check whether we have enough free space in segments that are currently empty[2] to write the entity in question without the need to call the garbage collector while keeping at least ten free segments at collector's disposal. If this test fails the garbage collector is sent an emergency message and the process is blocked until more segments become available. In the end we secure the newly dirty object a spot in the currently free segments, this accounting is therefore called **preallocation**.

Tracking accounting states of inodes and blocks

Free space has already been allocated for a block if the **pointer** at it[3] is non-zero (i.e. the block is not a hole). Because this pointer can contain a meaningful value only after the object leaves cache and is written to disk, a special value `LFS_INODE_NEW` is stored into it after it is accounted for and before it is actually written to disk. Blocks are freed by truncate which returns a block to the current free space whenever it discovers it has removed a block with a non-zero pointer. Every existing inode has always been accounted for, otherwise it wouldn't be created.

There is only one variable holding the free space per a mounted file system and it is `free` in `free_space` in the superblock info. It contains the number of *inodes* that the user is still allowed to store on the disk. Whenever free space for a block should be allocated, the number of inodes that entirely cover a block is subtracted from this value.

On the other hand, there are separate counters of preallocated blocks and inodes. One can tell whether an inode has been preallocated by examining its `flags` for the `LFS_II_PREALLOCATED_BIT`. A block is preallocated if and only if the highest order bit (`LFS_INODE_PREALLOC_FLAG`) of its pointer is set. That means only the lower 63 bits of the pointer actually contain the physical address of the block on the device and the pointer must be masked with `LFS_INODE_ADDR_MASK` whenever used. The function `lfs_is_prealloc()` returns whether a given pointer has the preallocation bit set or not.

Storing these state information in pointers and not for example in the corresponding buffer head flags is not accidental. Truncate must correctly update both free space accounting and preallocation but does not have the direct blocks or their buffer heads at its disposal because both have already been destroyed. On the other hand, inodes and indirect blocks still must exist and information stored therein is still accessible.

[1] See message `LFS_GC_INFO_WORK` in the section 5.10.1.

[2] See section 5.8.3

[3] Either an element of the `i_blocks` array in inode or of an indirect block.

Moreover, it is essential that if a block is free space accounted for, all its superior indirect blocks are also. Similarly, a block that is preallocated has all its superior blocks preallocate too. Last but not least, the ifile blocks are never preallocated, there must always be enough space in free segments to store all dirty blocks, inodes and the whole ifile, no matter how many dirty blocks there are in the inode. Otherwise, emergency takes place. This decision ensures the segment building code which must never wait for the garbage collector can modify the ifile in any way it decides to.

Performing accounting checks

We have already stressed that the file system must check whether there is enough free space when servicing a system call that creates the given new entity so that the user process can be informed should this test fail. Similarly, preallocation test must also be performed when servicing a system calls that marks a given object dirty because at that time it is safe to put the current process to sleep. In this way we avoid blocking processes that free memory by flushing the page cache which is essential because garbage collecting can be memory demanding and so this would lead to deadlocks and out-of-memory problems.

Data blocks can be created and marked dirty by the `write` syscall or any of its variants and through *mmap*[4].In the former case, the implementation of the syscall invokes our `prepare_write()` *address space operation*. In the latter case, whenever a process is about to mark a page dirty through mmap, the kernel calls our `page_mkwrite` *vm operation*[5] which also internally calls the same `prepare_write()`.

This address space operation is then responsible for marking the given blocks dirty as well as all indirect blocks on the way to the inode and the inode itself. During this process, each newly created block is given a free space from the current free space variable and each non-preallocated block is preallocated. However, preallocation is split in two steps because a process waiting for the garbage collector must not have any pages locked or hold certain semaphores. Therefore, enough preallocation blocks are acquired right in the beginning. If there are not enough free segments, the process releases the page held, blocks until there are enough segments and then always immediately returns `AOP_TRUNCATED_PAGE`. The caller is then responsible for reacquiring the necessary page and calling the operation again. Once the necessary number of blocks is acquired, they are distributed among the current block and its superior blocks, if they haven't been already preallocated. At this point, the highest order bits in their pointers are set, as described above. Finally, all unused acquired preallocation blocks are returned to the global variable. Directory operations and truncate acquire preallocation blocks themselves because they need to do it before locking certain semaphores.

Free space is acquired for inodes when they are created in the `lfs_ifile_new_ino()`. Inodes are preallocated whenever they are marked dirty in our handler of `dirty_inode()` *super operation.* However, this handler must not sleep and so the process must be suspended later on when processing either a `prepare_write()` or `set_attr()` operations.

Free space is returned when truncate is about to free a block that was not a hole or an inode is deleted. Preallocation is returned whenever a dirty block or an inode is sent to the disk. The code that writes back dirty indirect blocks refuses to write those containing preallocated pointers. Similarly, pointers in inodes are masked before being written. Therefore, there is a per-inode read write semaphore to mutually exclude inode syncs and `prepare_write()` because otherwise some indirect blocks might not have been synced.

5.2 Reading files

LFS leaves the generic kernel functions to deliver data from the *page cache* to the user space, whether in the form of standard *read* system call, any of its variants or *mmap.* When reading data, LFS only fills in the cache.

[4]Even though it is illegal in Linux to mmap a block behind the end of file, a user process can write data to a hole inside a file and thus create a block nevertheless.

[5]This operation is provided by David Howell's patch.

First of all, however, the inode must be read, usually as an indirect response to a call to `iget()`. This means its address is retrieved from the ifile, the raw inode is read from that address (both operations are performed in `lfs_read_inode_internal()`) and finally the inode info together with kernel VFS inode are both initialized with the read data (in `lfs_read_inode()`). There is an exception to this rule which takes place when the inode in question is currently being written and which is described in section 5.6.

Pages from the data mappings are read by generic functions `mpage_readpage()` and `mpage_readpages()` and both use `lfs_get_block()` to map a given buffer to a specific spot on the disk. Pages of the indirect mapping are read by our function `lfs_indirect_readpage()` which is a modification of a generic kernel function `block_read_full_page()` that does not check whether reads exceed the file size. This function internally calls `lfs_get_block_indirect()` to map buffers to locations to disk. Both mapping functions in the end call `read_block()` which either looks up the required position from a pointer in an inode or uses the same mechanism to read a superior indirect page and locates the corresponding pointer there.

5.3 Writing data

This section describes how dirty data in various caches get written to disk. Please note that this chapter does not cover the *sync* operation which requires specific consistency measures. It will be dealt with thoroughly in the section 5.4.

5.3.1 Tasks performed by the segment building code

All data and metadata except superblocks are written into a log consisting of segments. At any given moment the currently written data are being stored into a particular segment and a particular partial segment. Moreover, data has not been written safely until at least a partial segment is finished. Therefore, the code that writes data to the disk is usually called the segment building code. It is almost entirely present in `src/log.c`. This part of the file system, complex as it is, naturally has its own info structure. It is called `struct lfs_log_info` and you can find it in `src/types.h`. This structure is actually a part of the superblock info (see section 5.1.1) called `log`. We will often refer to various fields in the log info throughout this section.

The Linux kernel, LFS garbage collector and LFS journaling subsystem can ask the segment building to perform these operations:

- **Write pages.** LFS obviously implements both `writepage()` and `writepages()` handlers of data and indirect address spaces of each LFS inode. These methods are called by *pdflush*, *kswapd*, memory management routines, *sync*, *fsync*, *msync* and so on. Last but not least, these operations may be triggered by some other part of the segment building code.
- **Write inodes.** LFS also carries out superblock operation `write_inode()`.
- **Garbage collecting.** Once garbage collector identifies which pages and inodes should be moved to the current segment it uses this subsystem to actually write them.
- **Flush journals.** Even though this is very rare and journals are usually written out because parts of relevant inodes are, the journaling subsystem can decide the journal data occupy too much memory and ask the segment building part to flush some of it to the device.

5.3.2 Stages of the segment building code

Now let us have a look at various stages of building a partial segment (see also figure 5.1):

1. **Initialization.** The decision whether to continue in the current (physical) segment or not is made when finishing a partial segment so this has already been determined at this stage. Of course, initialization of the file system is the only exception and in that case a new physical segment is set up.

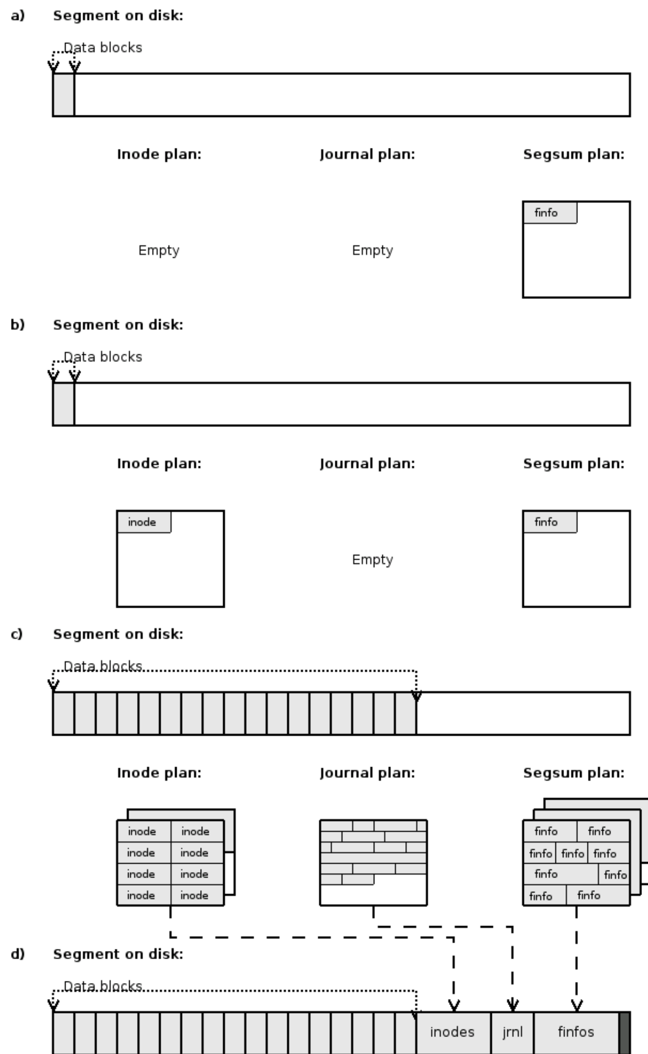

Figure 5.1: Segment building. In picture **a)** there is a newly initialized partial segment and one block written to it. At the same time, a corresponding *finfo* has been created and planned. Other plans are still empty. In **b)**, the system asked LFS to write an inode and the file system stored it in the inode plan. Picture **c)** shows the segment and plans later on, when there is a number of blocks already written to the device and two pages of inodes, a page of journal lines and three pages of finfos planned. Picture **d)** shows what happens when LFS decides to finish the segment at this time. Basically, all plans are written to the disk together with a segment summary.

Every time we write segment we must know what the next one will be (see section 4.7 and figure 4.6). Therefore, we always need to request a segment in advance a call to `usage_get_seg()` so that the segment management subsystem finds a new segment for us. All that has been described in this point so far is done by function `find_segment()`.

Finally, majority of log info structures are initialized in `init_ordinary_segment()`.

2. **Writing pages.** When the system asks us to write a page or a bunch of pages from a mapping, we do it straight away and do not queue the pages ourselves in any way[6] The blocks are written from the beginning of the current partial segment in the order they are passed to the LFS by kernel or retrieved from the given mapping.

   However, there is more to be done. At the same time, *finfo* structures are prepared in memory so that when the segment is about to be finished, they can be written to disk. Similarly, journal pages that contain *jlines* related to the processed inode (the inode to which the written pages belong) are retrieved from the journaling subsystem and queued for writing which will take place when the segment is about to be finished.

   This means that there can be a lot of data waiting in the memory to be written just before the current partial segment is closed. It is essential to guarantee there will be enough space in the current segment for all of it. This is accomplished by a remaining blocks counter in the log info called `remblock` which is decremented each time a block of data is written to the disk or sizes of planned finfos, journals or inodes (see below) grow beyond another block. When this number reaches zero, the current segment must be finished.

3. **Queueing inodes.** Whenever the system asks LFS to write an inode, it is not written straight away but the corresponding raw inode is queued instead. *Ihash* subsystem is in place to make sure no inode is written multiple times into any single partial segment. (See section 5.6 for information on how this is done and other issues *ihash* solves.)

   Writing pages and queuing inodes of course can and often does interleave.

4. **Finishing the segment.** The most obvious reason to finish a partial segment is finishing the physical one as well. Once we have determined that there is just enough space for all the queued structures, they must be written and a new segment started. Nevertheless, there may be other reasons too. Partial segments are finished before the ifile is synced during a sync, for example. Fsync or flushing excessive amounts of journals (see section 5.3.1) are another examples.

   Finishing segment basically means flushing inodes, journals and finfos to disk, calculating CRCs and creating a segment summary. Finishing partial segments also involves creation of a phantom partial segment (i.e. its summary). The whole procedure is described in detail in the section 5.3.7.

The fact that we do not queue dirty blocks is very desirable as it prevents various *out of memory (OOM)* problems and lockups. This is exactly the reason why have chosen the structure of LFS segments with a segment summary at the end rather that at the beginning (see section 4.8).

5.3.3 Plans

Storing inodes, journals and finfos as described above clearly needs some common infrastructure. We say that this data are **planned to be written** and the queues that store them are called **plans**. Normal segment building code uses an **inode plan**, a **journal plan** and a **segsum plan**, all of which are directly accessible from the log info. The segsum plan actually contains mainly *finfos* and has its name because in the early stages of the LFS project, *finfos* were considered a part of the segment summary. Nevertheless, the segsum plan also often contains the segment summary (see section 5.3.7) even though it may not necessarily be so. Plans are also extensively used when syncing the *ifile* but that is outside of the scope of this section and is covered properly in the section 5.4.

[6] Of course, the block I/O scheduler may queue our BIO requests.

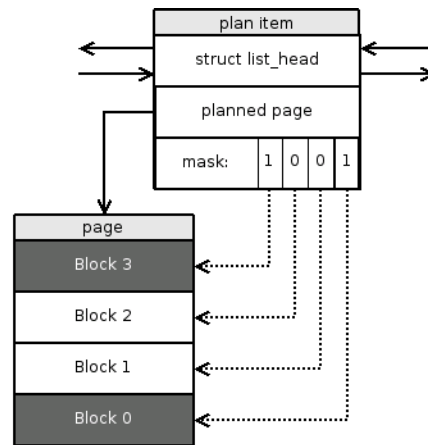

Figure 5.2: Plan item. In this example there are four blocks within a page and thus only four important bits in the mask. Because the mask is set to 1001, at this time only blocks zero and three are planned.

A plan is merely a list of `struct lfs_plan_item` instances. Each structure contains a pointer to a page with the planned data and a bit mask that tells which blocks in the page are supposed to be written and which are not[7]. The lowest-order bit of the mask corresponds to the first block within the page, the second lowest to the second block and so on. When a bit in the mask is set, the corresponding block is about to be written and the other way round. Figure 5.2 gives an example. Plan items are kept in a standard kernel linked list and together they form a **plan chain**.

All planned pages must be allocated by `alloc_pages()` so that once they are written they can be freed by `__free_pages()`. The plan items themselves are obtained from `plan_slab` slab. All plans considered in this section are empty when a new partial segment is initialized. New items are added to them and other operations are performed on them by functions listed in table 5.2. The way plan chains are written to the device will be described in section 5.3.7, their deallocation (under normal conditions) in section 5.3.8.

Function	Description
`grab_plan()`	allocates a plan item and initializes it with a given page and mask.
`release_plan()`	frees the plan info.
`error_destroy_plan()`	deallocates a whole plan chain. Intended for use when cleaning up after an error.
`append_segsum_plan()`	appends a given plan item to the current segsum plan.
`append_journal_plan()`	appends a given plan item to the current journal plan.
`append_inode_plan()`	appends a given plan item to the current inode plan.
`get_planned_page()`	allocates a page and a plan item and sets them up together with a provided mask.

Table 5.2: Plan manipulation functions

[7]At the moment there is only one block per page and so the mask always contains 1. However, it is going to be necessary to support file systems with smaller block sizes.

5.3.4 Synchronization

It is self-evident that most of the write operations must be serialized because there is only one segment being built and pages appended to its end at any moment. In order to simplify matters further, we have decided to serialize all of them by a mutex. The mutex is located in log info and is called `log_mutex`. All operations discussed above either themselves lock it or require it is already acquired.

The segment building module must also be synchronized with other events in the kernel, most notably *truncate*. That is why all inode infos contain a read-write semaphore called `truncate_rwsem` which is locked for reading whenever working with a particular inode or pages belonging to either of its mappings. Truncate itself locks it for writing because it locks the pages in the opposite order than all other operations. A great deal of synchronization is also performed through Linux page locks, most of which is not specific to LFS. On the other hand, the pointers in inodes and indirect blocks are protected by page locks of the block they correspond to.

Order of locking is the key instrument to avoid deadlocks. In this case, the `log_mutex` must be acquired first, the `truncate_rwsem` second and only afterwards a page can be locked. This requirement forces a small hack in `lfs_writepage()` (see below).

5.3.5 Writing pages

Linux kernel can ask a file system to write out a single page or to try to find a specified number of dirty pages in a mapping and flush them. The first request is communicated to LFS by calling the `writepage()` *address space operation* and is handled by the LFS segment building code for both data and indirect mappings. Both handlers do nothing but call `lfs_writepage()` with a correct value in its `indirect` parameter. On the other hand, writing out several dirty pages from a given mapping is performed by the `writepages()` *address space operation* which is also handled by LFS for both kind of mappings. Like in the previous case, they call a common routine `lfs_writepages()`. See a subset of the call graph of the page writeout handlers on the figure 5.3.

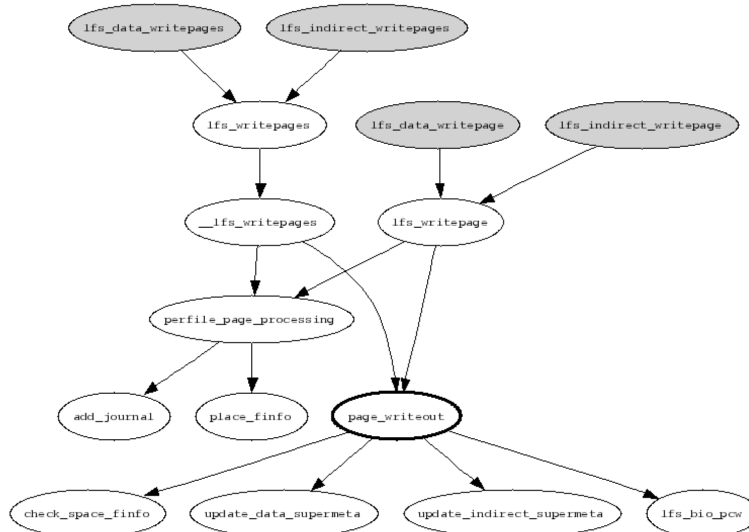

Figure 5.3: Page write call graph. Address space operation `writepage()` and `writepages()` handlers are marked by grey color. `page_writeout()` is printed in bold because this is the function which, among other things, creates and issues I/O requests.

Writepage

`lfs_writepage()` gets the page that needs to be written as a parameter. The page is obtained locked which violates the required lock ordering (see section 5.3.4 above). The page is therefore released, `log_mutex` and truncate semaphore are acquired and then the page is locked again. The code of course must check it has not been truncated and is still suitable for writing[8].

Afterwards the page cache writeout control structure (see below) is initialized and `perfile_page_processing()` called. This function is invoked each time a different inode is being written. It uses `add_journal()` to plan all *jlines* relevant to this inode and `place_finfo()` to create a new *finfo header* in the *segsum plan*. The latter function is a bit complex because the new *finfo header* can span across block and page boundaries and different actions must be taken in each case.

The most important function called at this place is `page_writeout()` which inspects the page and initiates I/O. We will deal with this function separately below.

Finally, `lfs_writepage` calls `lfs_submit_pcw()` to submit any pending *BIO* request, unlocks the locks held and calls `write_gc_stuff()` to process potential requests from the garbage collector.

Writepages

The only task of `lfs_writepages()` is to interleave the writes requested by the Linux kernel with those required by the garbage collector. The garbage collector requests are honored first and with no upper limit on the total number of pages written in one go so that there are segments ready when they are needed for new data. The garbage is written by `write_gc_stuff()` like in the previous case.

Writing dirty pages of the given mapping is the task of `__lfs_writepages()`. The function is similar to Linux kernel generic `mpage_writepages()` function. It acquires the log mutex first so that if it creates a *finfo header* it will remain the current header throughout the rest of its execution. Next, the function obtains a private array of page pointers with dirty pages of the given mapping and processes them one by one. Each page is locked and checked whether it should still be written (i.e. it has not been truncated and is still dirty).

`perfile_page_processing()` is called once if there has been a dirty page in the given mapping. The function performs the same tasks as in the case of `writepage()`. On the other hand, `page_writeout()` which is described below is called on every valid dirty page.

`__lfs_writepages()` can be limited in the number of pages to flush to the disk so that the garbage collector items are guaranteed to be processed once in a while. When this maximum number has been reached or all dirty pages have been written, the function submits any pending *BIO*, unlocks log mutex and returns. In that case it returns a special value so that `lfs_writepages()` calls it again after it has honored all pending garbage collector requests.

Page cache writeout control structure

`__lfs_writepages()`, `lfs_writepage()` and `page_writeout()` and a few other minor functions need to share a great deal of parameters. For the reasons of clarity of the code, these have been placed into a structure called `struct pcw_control` (page cache writeout control structure). Its contents includes a reference to the current page, whether the page is indirect, the inode, the obtained Linux kernel writeback control structure and other status information. Most importantly, however, there is also a reference to the current *BIO* (see [5], chapter 13) that is shared among different invocations of `page_writeout()` within a single execution of `__lfs_writepages()`. The structure also contains information about the currently processed block and the current dirty area of the page which can be added to the *BIO* by `lfs_bio_pcw()`. If the

[8] Even though this is not a very clean approach, it has been discussed in the Linux-fsdevel mailing list and has been generally regarded as tricky but viable. Our experiments seem to have confirmed this.

current *BIO* becomes full, it is submitted and a new one allocated. Finally, the last unsubmitted I/O vector is sent to the block layer by `lfs_submit_pcw()`.

Single page write

The most important function in this context that has not yet been discussed in detail is `page_writeout()`. The function begins with initializing a few structures of the page cache writeout control structure it has obtained from the caller. Next, if the page straddles the end of file, the part behind the end is cleared with zeros.

Most importantly, the function traverses dirty page buffers and performs the following actions:

1. `check_space_finfo()` adds a finfo entry to the current *finfo*. It may find out there is no room left for the *finfo* entry and the new block in the current segment. In that case it finishes the current segment, starts a new one, creates a new *finfo header* and adds the new entry into the new *finfo*.
2. The metadata pointing to the written entity is updated. The way of doing this differs for data blocks and for indirect blocks and therefore they are performed by different functions, specifically by `update_data_supermeta()` and `update_indirect_supermeta()`. The functions vary in how they locate the address that needs to be changed and then call `__update_supermeta()` to change it and initiate necessary *segment writeouts* (see section 5.8).
3. Segment *CRC* calculation is extended to cover the beginning of the current block.
4. The current dirty area of the page is extended or a new one is founded. This area is added to the current *BIO* once a clean buffer is encountered or all buffers have been processed.
5. Finally the buffer is marked as clean and several internal variables are updated. Let us at least briefly mention `curblock` of log info that contains the address of the current block and `remblock` which stores the number of blocks that are remaining in this segment.

In the end, the page is marked as under writeback and unlocked.

Please note that this is just a brief outline, refer to the source code for details.

5.3.6 Writing inodes

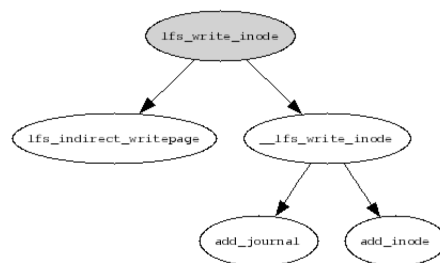

Figure 5.4: Inode planning call graph. Superblock operation `lfs_write_inode()` is marked by grey color.

Writing inodes is much simpler but it is done in two phases. First, an inode is planned. Later on, when the current segment is being finished, its metadata are updated and the raw inode is written to disk. The call graph of the first phase can be found in figure 5.4.

Linux kernel does not know about LFS *indirect mappings*. It can therefore issue destruction of an inode that has dirty pages in its indirect mapping. In order to avoid this, the dirty mapping is synced by `lfs_write_inode()` each time an inode is about to be planned to be written.

`__lfs_write_inode()` takes care of locking (it locks both the `log_mutex` and inode's truncate rwsem) and journal planning. Most importantly, it calls `add_inode()` to add the inode to the current inode plan. Both actions can cause the current segment to be finished and a new one started.

`add_inode()` attempts to lookup the inode in the *ihash* (see section 5.6) and if it succeeds, it copies the new raw inode over the old one. Otherwise it copies the raw inode into a planned allocated page. The latter case can involve page and plan allocation, updating and checking the number of remaining blocks in this segment and even segment finishing.

The final placement of the planned inodes are not known until the segment is being finished. The inode table cannot be therefore updated and relevant *segment writeouts* (see section 5.8) issued until this moment. Therefore, before the inodes are actually sent to the device, `update_inode_table()` is called to process the inode plan. It traverses the plan chain and calls `update_itable_entry()` on every raw inode. This function updates the relevant *inode table* entries and issues the *segment writeouts*. Not surprisingly, the ifile inode is handled in a special way because its *inode table* entry is not used and writeouts are issued differently during *sync* (see section 5.4).

5.3.7 Segment finishing

When a partial segment is being finished by `__finish_segment()`, it proceeds with the following steps:

1. `compute_segment_end()` calculates the expected end of the current partial segment from the current block and the number of blocks occupied by planned metadata. Based on this information, it decides whether it will continue with a new partial segment within the current physical segment or move on to a new one. The latter case may require there is a small gap in between *finfos* and the *segment summary*. This happens when a new partial segment would not fit into the remaining space and so the remaining free space is included in the current one.
2. `create_segsum()` allocates memory for the *segment summary*. The calculations carried out in the previous step decide whether the *segment summary* is created within the segsum plan or allocated separately. When LFS is about to continue writing to the same segment, a page is also allocated for the *phantom segment summary* (see section 4.7).
3. Ihash entries are marked as under writeback. See section 5.6 for details.
4. `update_inode_table()` is called. It has already been described in the section 5.3.6.
5. Segment CRC calculation is extended to cover planned inodes, journals and finfos.
6. *BIOs* are issued for planned inodes and journals.
7. The *segment summary* is created within the allocated space. At this point we know the exact locations of inodes, journals and finfos within the segment.
8. The segsum plan chain write is initiated.
9. If necessary, the *phantom segment summary* is created within the allocate space.
10. If there is a *phantom segment summary* or the ordinary *segment summary* is not placed within the segsum plan, they are also written to the disk. The two cases are mutually exclusive so the same code is used to do both.
11. Information about the new segment must be stored in the *segment usage table*. However, this must be done when no page is locked and therefore this information is stored to a buffer by `add_set_cache_item()`. The code that unlocks pages then calls `flush_usage_set_cache()` which finally asks the segment management to include this information in the table.
12. `seg_start` in the log info is set so that segment initialization knows where the next partial segment is about to begin.

5.3.8 BIO finish callbacks

The *BIOs* issued by LFS can be of the following types:

- *BIOs* writing *page cache* pages,
- *BIOs* writing plan chains and
- those writing separate and *phantom segment summaries*.

Table 5.3 gives a more exhaustive overview of all functions that generate block write requests in LFS.

Function	**Description**
`lfs_submit_pcw()`	is used to write data blocks to disk when not syncing. The *page cache* finish handlers are used and no barriers issued.
`bio_plan()`	is used to flush journals and segsum plans when finishing an ordinary (i.e. non-sync) segment. `bioend_free_all()` is used as the finish handler. This function is capable of issuing all kinds of barriers but currently it is not used to generate *sync* barriers.
`bio_inode_plan()`	is used to flush inode plans to disk. It does not issue barriers and uses `bioend_free_inodes()` to delete written inodes from the *ihash* (see section 5.6) and free the enqueued pages.
`bio_sync_data()`	is used to flush *page cache* pages during sync. It uses the same handlers as `lfs_submit_pcw()` does.
`bio_sync_inodes()`	is similar to `bio_inode_plan()` but does not increment the `curblock`. It is used to write the ifile inode during sync.
`bio_sync_meta()`	is used to write journals and *finfos* during sync. The `bioend_free_all()` is the finish handler of choice and either no barrier or the full *sync* barrier is issued.

Table 5.3: BIO issuing functions

Another and perhaps more important division between *BIOs* is into those which write *page cache* pages and those that write pages allocated by `alloc_pages()`. The first group currently uses the `bioend_pcache_simple()` to clear the pages' *writeback* flag when the write is finished. Nevertheless, there is a handler ready for filesystems with multiple blocks per page called `bioend_pcache_partial()`.

Allocated pages are deallocated by `__bioend_free_all()` after they are flushed to the disk. Pages carrying inodes are processed by `bioend_remove_inodes()` so that written inodes are removed from *ihash* before the deallocation happens. Moreover, these *BIOs* may be issued as barriers. Linux kernel *BIO barrier* is a special *BIO* flag described in [6] that prevents the block layer from reordering write requests across this particular one. LFS issues barriers when finishing segments for several reasons. The most obvious is waiting on I/O to finish before we exit from *sync* and *fsync*. However, we also issue barriers every time we finish a segment so that we know all data marked as written out in the segment usage table have actually been written out and this information is then propagated to segment management.

Because barriers are used for a variety of purposes in LFS we internally differentiate in between three types of barriers. *BIOs* issued as barriers carry along them a small structure called `lfs_barrier_info` that contain information internal to segment management, the barrier kind and so on. The three types of barriers are:

- **Simple barriers**. Segment management is informed a segment has been finished and the barrier info is deallocated by the *BIO* finish handler. No process ever waits for this barrier.

- **Sync barriers**. Segment management is informed a *sync* has been finished. The process executing the *sync syscall* waits on this barrier to finish and then deallocates the associated barrier info.
- **Fsync barriers**. These barriers are very similar to sync barriers in that they are being waited on and the initiators deallocate the corresponding barrier info. The only difference is that the segment management is not informed there was a full sync.

5.3.9 Dealing with errors

When a *BIO* finish handler detects an error, it sets the **failure flag** in log info and adds the number of the current segment into an array in the same structure. A few functions that have already been mentioned in this section call `check_failure()` to check the flag and if it is set to start a new physical segment and mark all segments in the array mentioned above as bad. The emergency flag is also set when a new segment cannot be initiated due to an I/O error.

5.3.10 Garbage collector writes

When garbage collector identifies the live data and inodes in a segment that is about to be cleaned, it queues this information and then invokes `write_gc_stuff()` to write all queued entities to a new location. In order to guarantee a certain degree of priority to the garbage collected data, ordinary segment building functions also regularly call this function regularly.

Garbage collector does not store individual pages but mappings that are written by `__lfs_writepages()` described earlier in this section. On the other hand, inodes are returned individually and they are planned by a direct call to `__lfs_write_inode()`. Garbage collection of inodes therefore does not involve syncing the indirect mapping because it is not necessary.

5.4 Syncing

Implementation of the *sync* syscall must ensure three things:

1. All dirty cached data must be written do disk.
2. All parts of directory operations (all inodes, data pages and journals) must be written either before or after the ifile is synced.
3. The ifile must be brought into a consistent state.

The first part of this section is dedicated to the first two tasks listed above because they are very interconnected. The rest of this section considers writing the ifile so that it is full consistent with itself.

5.4.1 Flushing cache

Cached data are flushed partly by the Linux kernel but unfortunately it does not guarantee all of it is synced before the *sync super block operation* is called. We therefore keep a list of dirty inodes and sync all of them in that operation ourselves. This list and all relevant functions can be found in file `src/consistency.c` and they should be fairly simple to understand. This queue never contains the ifile which is taken care of differently later on.

While flushing the cache, we must make sure either all or no parts of a directory operation (all inodes and the *jlines*) are sent to disk. This is achieved by a read-write semaphore that is tweaked a bit so that it does not starve the writer. Directory operations lock it for reading while the sync functions down it for writing. Finally, after the semaphore is acquired for writing and before dirty inodes are synced, the journaling subsystem is forced to start a new page so that a *jline* does not span across the sync. The old *jlines* are then flushed to disk when the relevant inodes or their parts are.

5.4.2 Planning a consistent ifile

Whenever a block is written to a new position on the disk, the *segment usage* table is updated because the number of live blocks of the old segment is decremented (this is also referred to as **a usage writeout**). Moreover, whenever a segment is finished, the table must also be updated with the new numbers of blocks and inodes residing in the new segment. Both operations therefore mean that pages of the ifile are marked dirty. This means that trying to obtain a consistent image of the ifile by iteratively writing all dirty pages until all of them are clean would result into an infinite loop. The solution to this problem is to plan the writes rather than immediately flush them to the device. In this way, if a page is dirty but it has already been planned, it is not planned again.

The infrastructure to plan pages has been described in section 5.3.3. When syncing, it is also used to plan the ifile page cache pages and associated *finfos*. The only planned inode during this phase of a sync is the ifile inode and it is the last planned entity during the ifile sync. There are no journals processed during this last stage.

```
struct lfs_sync_segment {
        struct list_head list;
        sector_t segnum;
        lfs_addr_t startblock;
        struct list_head data_plan;
        lfs_addr_t inode_block;
        struct list_head inode_plan;
        lfs_addr_t ss_block;
        struct list_head ss_plan;
        int phantom;
        int separate;
        lfs_addr_t end;
        lfs_addr_t last;
        struct page *page;
};
```

Figure 5.5: Sync segment plan definition

The ifile, however, can span across multiple (partial) segments. The structure `lfs_sync_segment` (see figure 5.5) represents such a partial segment. They are chained together by a linked list and represent a plan of segments (see figure 5.6). The whole chain is accessible from the log info's `segments` field, the current one can be found in `cur_sseg` in the same structure. Each structure, among other things, stores its own plan of page cache pages, inode plan and a segsum plan. If there is a phantom or separate segment summary, it is flagged in the relevant field and allocated in the given `page`. These structures are updated as pages and inodes are being planned. Once `remblock` reaches zero, the current segment is enqueued by `finish_sync_segment()` and a new one is created and initialized by `init_sync_segment()`.

5.4.3 Building the sync plan

The strategy of producing the sync plan is the following. First, we sweep through both data and indirect mapping of the inode and plan all dirty pages we find. Identifying the dirty pages is done in function `sync_ifile_mapping()`, the pages are then locked by `lock_add_ifile_page()`. These two functions are in many ways similar to `__lfs_writepages()`. Buffers of these pages are then processed by `add_ifile_page()` which may resemble `page_writeout()` but differs in a few important aspects.

As we have already said, it does not issue I/O but plans the dirty buffers instead. One of the key features is that when any other block (of the ifile) is marked dirty due to metadata updates, it is enqueued as a request to write. The request is formed

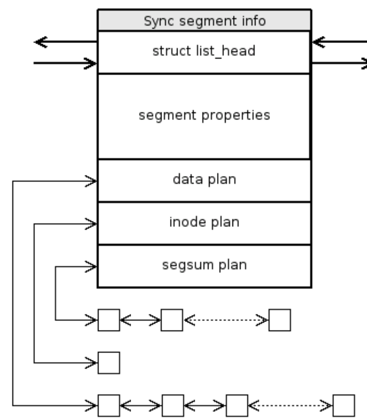

Figure 5.6: Sync segment plans

of a list of small page handlers called `struct sync_req` (see figure 5.7) which refer to a page and indicate from what mapping that page comes. Pages are added to this queue at two occasions: When pointers in indirect blocks are modified and when segment usage table entries are updated. In the latter case, the segment management code actually enqueues the relevant buffers by calling `lfs_sync_enqueue_buffer()`. After both mappings are traversed, the pages in the request queue are planned by `lock_add_ifile_page()` until the queue is empty.

```
struct sync_req {
        struct list_head list;
        struct page *page;
        int indirect;
};
```

Figure 5.7: Sync segment plan definition

To avoid planning or queuing pages that are already planned or queued, two LFS specific `buffer_head` flags have been introduced. The first one is called `queued` and is set when the page containing this buffer has been enqueued because this buffer has been marked dirty. The second one is known as `planned` and it is set when the buffer is included in the data plan of the current sync segment plan. Both flags are cleared just before their data plan is sent to disk and under certain error conditions. Understandably, a buffer is not queued if it has either of the flags set and it is not planned unless its `planned` flag is zero.

When the request queue has been emptied, the ifile inode is added to it and the current sync segment planning is finished.

5.4.4 Flushing the segment plan

The plan that has been built as described in the previous section is then flushed to disk by `bio_ifile_sync()`. This function processes the planned segments one after another in two steps. First, the buffer flags described in previous section are cleared in all queued buffers and then all planned entities are flushed to disk by `bio_sync_segment()`. This function issues *BIOs* in a way similar to `__finish_segment()` and also issues a *sync barrier* (see section 5.3.8) after flushing the last segment.

When all *BIOs* have been submitted, a new ordinary segment is initialized, the superblock is written so that its raw image on the disk contains the new ifile inode address, next block and segment counter. This is followed by waiting on the *sync barrier BIO* to complete. Finally, as locks are unlocked, the file system has been

successfully synced and the segment management subsystem and the garbage collector are informed about it (see section 5.8.3).

5.5 Fsync

Implementation of *fsync* in LFS is very simple. It consists of flushing cached data from both data and indirect mappings, planning the inode and finishing the current partial segment (which in turn writes the raw inode to disk). If a system failure occurs after this point, the roll forward utility will be able to recover the written data.

5.6 Ihash

There are two potential problems with planning and late flushing of inodes as described in the section 5.3.6:

1. The scheme as described would often store a single inode multiple times in the same partial segment.
2. When the Linux kernel is about to free a dirty inode, it asks the file system to synchronously write it to the disk. Once this operation returns, the file system assumes the inode is safely written and may be read from the device if needed again. On the other hand, LFS merely plans the raw inode to an allocated page which is written when the current partial segment is finished. However, there is a time frame in between the planning and actual writeout in which the kernel expects the inode is on the disk but it isn't. Any call to `lfs_read_inode()` would then either fail or produce incorrect results.

Both these issues are solved by a hash table called **ihash**.

5.6.1 Hash table

Ihash is a double pointer hash table consisting of 8192 `lfs_ihash_entry` structures (see figure 5.8). Each entry with a non-zero `ino` contains a pointer to a planned *raw inode* and a `writeback` flag indicating it is already being written by a *BIO*.

```
struct lfs_ihash_entry {
        ino_t ino;
        int next;
        int first;
        struct lfs_inode *raw_inode;
        int writeback;
};
```

Figure 5.8: Sync segment plan definition

As already mentioned above, the ihash is a double pointer hash table. In order to locate an inode stored in it, its inode number is hashed by hash function `hash_long()` provided by kernel to calculate its chain number. The chain number is used as an index to the hash table and the `first` field of the obtained structure is examined because it contains the index at which the particular chain begins. Chain items are connected in a single direction linked list by the `next` index. All functions using and manipulating the ihash can be founds in `src/ihash.c` and should be easy to understand.

Manipulating and searching the ihash is always protected by a spinlock in the ihash info structure that is a part of the superblock info.

5.6.2 Use cases

The ihash is used twice when an inode is planned. Before a new spot is allocated for a raw inode LFS tries to find it in the ihash. If it is found and is not undergoing

writeback, the raw inode that is already planned is overwritten with new data. This ensures a single inode is never present multiple times in a partial segment which was happening quite often before *ihash* was implemented. If the inode is not already present in the ihash, it is inserted and if it is present but marked as under writeback, the pointer in the ihash entry is updated and the writeback flag cleared.

Conversely, when an inode is requested to be read, the ihash is tried first. If it is found in the ihash, the raw data referenced there are used regardless of the `writeback` flag. On the other hand when inodes are being deleted, they are removed from the ihash because the associated ihash item no longer contains valid information.

Just before all inodes are sent to disk, all entries in the ihash are marked as under writeback. When a *BIO* carrying inodes terminates, the finish handler identifies all inodes present in the pages that were written and deletes their entries from the ihash, unless their `writeback` is not set which happens when the inode was re-inserted into the hash after the I/O has been submitted.

5.7 Truncate

From the Linux kernel's perspective, the file *truncate* operation is divided into two phases. First, the inode size is set to the new value, the data mapping is truncated to reflect the new size and the straddling part of the last page is cleared with zeroes. Afterwards, the file system is told to update its own private structures to reflect the new size by invoking the file system's `truncate` *inode operation*.

LFS must do three things. Obviously, pointers to blocks in the inode and indirect blocks must be invalidated, free space accounting and preallocation state must be updated and *segment writeouts* must be issued so that segments can be freed. All these tasks are performed by calling `lfs_trunc_iblocks()`. Dealing with direct addresses stored in the inode is easy. If such a pointer is "behind" the new file size and contains a non-zero value, it is cleared (by `__inode_set_addr()`), one block is added to the free space accounting and a block is subtracted from the corresponding segment usage table item by function `__free_block()`. This function also checks the preallocation flag of the discarded block and if it is set decrements the global counter of preallocated pages. Altogether, we say that such a block is **dropped**.

The real challenge here are the indirect blocks. Consider the example in figure 5.9. The highlighted address in indirect block j points to the new last block of the given file. It is evident that the whole indirect block k must be removed, i entirely kept and roughly a half of entries must be cleared in block j.

`lfs_trunc_iblocks()` starts by calling `__dblock_to_path()` to identify all indirect blocks through which the end of file "passes". Then at each level of indirection, the addresses with higher indices and their children are *dropped fully* while the one exactly at the particular index is *dropped partially.* However, even a partial drop may end up as a whole one when the file ends exactly at the border of the first of the block in question.

Speaking implementation-wise, we start with calling `__drop_iblock_partialy()` on the identified border block at the highest level (the block 1 in the example on the figure 5.9). If we are not at leaf level, the same function is called on the border block one level below (block j in the same example). In any case, all non-hole indices higher than the border index are dropped by `__drop_iblock_whole()`. If the block finds out it is completely empty, it signals this to the caller so that it can do the same decision and clear the relevant address. *Full drops* mean the whole subtree is traversed and any non-hole pointer is *dropped* unconditionally. Finally, the no longer necessary indirect pages are themselves truncated.

The whole operation described in this section is performed with `truncate_rwsem` of the relevant inode locked for writing so that any actions do not interfere with the segment building code or garbage collection (see section 5.3.4).

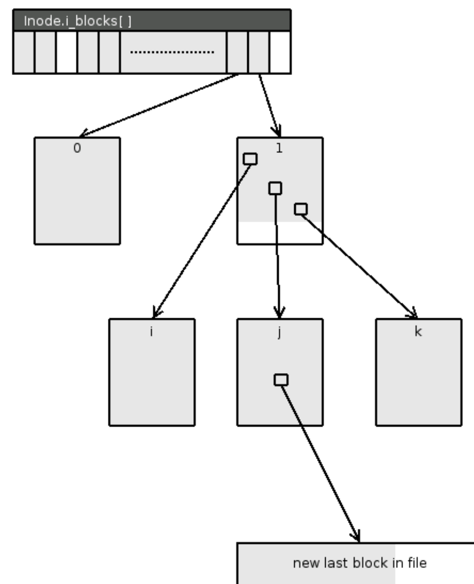

Figure 5.9: Truncating indirect blocks. The white boxes or parts of boxes are holes, the grey ones denote valid addresses on the disk.

5.8 Segment management

Segments are the basic unit of free space reclaim of all log structured file systems, a segment can be reused only once all live data have been either deleted or moved elsewhere. This basic rule requires the LFS to track the state of all segments and dictates an ability to find a new empty segment for future writebacks whenever necessary. The state of a segment consists of the number of inodes and blocks stored in it, various flags and its segment counter. The basic structure used to persistently store all this information for each segment is the *segment usage table* stored in the ifile (see section 4.5). A segment can fall into any one of the following categories:

Superblock segments contain one of the four superblocks and nothing else. They are never reclaimed.

Reserved segments are segments reserved by the user and are left untouched by the file system. LFS currently creates one such reserved segment at the beginning of the disk and actually requires it is not used.

Bad segments are segments in which a write operation has failed at some point in the past. LFS will not try to reuse such segments any more.

Empty segments contain no live data and are ready for immediate reuse.

Used segments do contain some live data. Obviously, they cannot be reclaimed and overwritten.

Possibly free segments do not contain any live data but some event must take place before the segment may be safely reused. These cases are discussed in the section 5.8.3.

Table 5.4 shows how information in segment summary determines the category of a segment. Basically, there are three fundamental operations performed by the segment management and they are covered in the three subsections below.

5.8.1 Providing the next segment

When the segment building subsystem finishes with a full physical segment, it needs to obtain the next one to write to. Any *empty* segment will do but locating it in the segment usage table would be costly. The subsystem therefore maintains a bitmap

state	flag	live data
Reserved	LFS_USAGE_RESERVED	
Superblock	LFS_USAGE_SUPERBLOCK	
Bad	LFS_USAGE_BAD_SEG	
Empty	LFS_USAGE_EMPTY_SEG	
Used	LFS_USAGE_REGULAR	> 0
Possibly free	LFS_USAGE_REGULAR	$= 0$

Table 5.4: Relationship between a segment category, flags and live data counters in a segment summary.

`free_seg_map` in the ifile info[9]. This bitmap contains exactly one bit for each segment which is set if and only if the corresponding segment can be immediately overwritten. The segment building subsystem requests segments by calling `usage_get_seg()`. This function simply finds a set bit in the bitmap, clears it and returns the corresponding segment number. The function may block in the unlikely case there is no segment available. In that case, a number of *BIOs* must be under way and their completion will make some segments available and wake up the sleeping process.

5.8.2 Segment writeouts

A segment summary must be updated whenever an entity stored within it dies. Whenever the segment building subsystem moves an entity from a segment elsewhere, a *truncate* operation frees a block or an inode is deleted, `usage_writeout_block()` or `usage_writeout_inode()` is called. Both these functions internally call a common routine `__usage_writeout()` which updates the segment usage table and sends a message to the user space garbage collector so that it can also update its internal state (see section 5.10). Because this operation usually informs the segment management some data have been written out of the segment elsewhere, it is referred to as a **segment writeout**.

5.8.3 Safe reclaiming

Reusing a segment immediately after the last *segment writeout* indicated all data in the segment are dead can lead to data loss. Consider the following situation. The segment building code has issued that last writeout and initiated a *BIO* writing the updated data to a different spot on the disk. If the system crashes before the current partial segment is completely finished, the data cannot be recovered from the new position because the roll forward utility cannot handle unfinished partial segments. If the old segment was meanwhile reused, the data in question could be lost.

This is the reason for distinguishing the *possibly free* segments from *empty* ones explained earlier in this section. When actual usage of a segment reaches zero, the segment is put into `free_seg_list` in ifile info together with a unique number `free_counter`. Whenever the segment building finishes a segment, it queries the current value of this counter by calling `usage_get_free_counter()` and stores it alongside the last *BIO* of this segment. The completion handler then passes this value back to segment management subsystem by invoking `usage_disk_sync()`. Because this function is therefore called in the interrupt context and a mutex must be acquired when manipulating the free segment queue, it only stores the passed counter. Later on, `lfs_update_free_segment()` is called at various places when it is safe to block on a mutex. This function processes `free_seg_list` and marks those segments that conform to all three following criteria as *empty*:

1. The *free counter* stored with the segment must be smaller or equal to the one passed by the last invocation of `usage_disk_sync()`. This effectively prevents the problem described above from happening because all last BIOs of segments

[9] The superblock info contains a pointer to this structure

are submitted as write barriers and so their successful completion guarantees a partial segment has been written to the disk.

2. The *segment counter* (the one described in the section 4.7) must be smaller than then segment counter of the last *sync segment*. This ensures that segments younger than the last sync are never reclaimed because doing so would break the chain of segments (see figure 4.6) the roll forward utility relies on. The figure 5.10 contains an example of what could happen if this condition was not enforced.
3. The *segment counter* must be greater than the last segment of the snapshot, if there is an active one because it contains data that are still accessible by the snapshot. Such segments are moved to the `snap_delayed` list and are marked as empty when the snapshot is umounted.

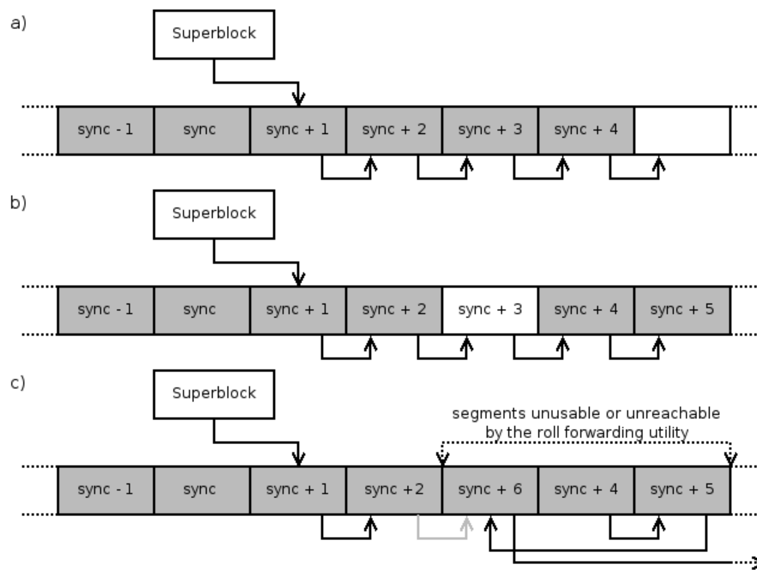

Figure 5.10: Keeping segment chain after sync. **a)** Segments written after sync are chained by pointers stored in their summaries and their segment counters follow immediately one after another. **b)** All data within the third segment die either because they are deleted or moved to the fifth one. If we allowed the third segment to be reused at this point, this could lead to situation **c)** where a number of segments are either unusable or inaccessible by the roll forward utility. The sixth segment cannot be used because its segment counter indicates it does not immediately follow the second one while the fourth and fifth segments are not even in the sync chain any more.

Points two and three above mean that only segments younger than the last snapshot segment and older than the first segment after last sync can be reclaimed. This time interval is called the **window**. Point one guarantees either the old or the new copy is on the disk at any time. On the other hand, if the machine crashes before the next sync, the roll forwarding utility is needed to recover moved data. Please note that garbage collector may decide to move virtually any data at any time so this does not apply only to the entities a user has deliberately modified.

When segments are marked as *empty* all processes that might be blocked waiting for them are awaken by calling `wake_up()` on the `free_seg_cond` queue.

5.8.4 Managing the currently written segment

Occasionally, segment writeouts are issued against the segment that is being written to at the same time. They are mostly results of truncation of data that have just been

written to the disk but sometimes even the same block is written to the same partial segment multiple times. Because the amount of live data stored in the segment usage table is set after the whole partial segment is written, this could cause the usage of the segment to become negative, confusing both segment manager and garbage collector. In order to prevent this, writeouts from the currently written segment, the so called **working area**, are not immediately reflected in the segment usage table but held aside to be processed once at least a partial segment is finished.

Moreover, when the segment usage of the active segment drops to zero after finishing a partial segment, the segment must not be marked as potentially free. This can be done only when the whole physical segment is finished and the segment building code moves on to the next one.

5.9 Syncing thread

In order to prevent situations like the one depicted on figure 5.10 we do not allow recycling and garbage collection of segments that are younger than the last sync. A periodic sync ensures that the number of affected segments is always relatively small. Periodic syncing is done by the **lfs-sync** thread that repeatedly tests how many new segments have been written by a log structured file system since the last sync. If the number has been bigger than a threshold value, a sync is issued. The threshold value is computed as maximum of ten percent of the currently free segments in the file system and 10. There is only one such thread in the system that services all currently mounted LFS instances.

5.10 Garbage collector

We have already stated in section 5.8 that LFS reclaims free space with segment granularity and that only segments that contain no live data may be reused. In order to manage disk space effectively, data in underutilized segments must be moved to new full segments. The process of doing so is often referred to as **garbage collection**. Basically, it consists of identifying live data in the underutilized segments and write it as usual to a new segment.

Because the process of deciding which segments should be garbage collected is complex and in future we intend to support multiple such algorithms at the same time, it has been moved to the user space. This chapter considers the kernel module internals only, details of the user space program can be found in section 7.2. Once this program determines which segments are to be cleaned, it is again the kernel module's job to identify the live data and write them to the current end of the file system log. The kernel part and the user program communicate through the *NETLINK* protocol.

5.10.1 Communication protocol

The *NETLINK* communication interface allows each message to be sent to either one particular process defined by its *pid* or a group of processes identified by a group id. User space garbage collector starts by sending a handshake message to the kernel which in turn registers to its array of active garbage collectors to which it unicasts the necessary messages. Unicast is necessary because delivery of each message must be guaranteed, a feature that is not provided by the multicast infrastructure.

The kernel module can send any of the following messages to the user space collectors (so called **info messages**):

`LFS_GC_INFO_INIT` is a reply to a handshake sent by the collector. The message acknowledges the user space program has been registered as a garbage collector and is going to receive the necessary updates. It also contains the superblock of the relevant file system.

`LFS_GC_INFO_SET` contains information about a new segment and its usage.

`LFS_GC_INFO_SET_PARTIAL` informs the collector a new partial segment has been added to the current segment and the exact numbers that should be added to its usage.

`LFS_GC_INFO_UPDATE` provides information about a segment writeout (see section 5.8.2) along with information how many inodes and blocks have been subtracted from the usage of the relevant segment.

`LFS_GC_INFO_WORK` is kernel's request for a collection. It includes a severity flag which means there is an imminent shortage of segments and all other processes are blocked waiting for garbage to be collected and segments freed.

`LFS_GC_INFO_WINDOW` informs the user space collector that the *window* has changed (see section 5.8.3). This essentially means that either a snapshot has been mounted or umounted or a sync took place. At any moment the garbage collector is allowed only to request collection of segment older than the last sync and younger than the snapshot, if mounted. The reasons for such requirements are also presented in section 5.8.3.

`LFS_GC_INFO_REJECT` is generated if the garbage collector requested collection of a segment outside of the collection window. This happens very rarely when the newest window update has not been yet processed by the application.

`LFS_GC_INFO_PING` is sent once a while if no other messages are issued. The garbage collector is expected to reply with `LFS_GC_REQUEST_ANSWER` and may actually also request some collecting because this message means the file system is not being modified at the moment and it seems to be a good time to defragment the disk a little.

`LFS_GC_INFO_END` is sent when the file system instance is being umounted. The user space garbage collector should terminate.

Below is an overview of all message types that the user space program can send to the kernel module (so called **requests**):

`LFS_GC_REQUEST_REGISTER` is a handshake message that initiates the communication channel. Kernel registers this garbage collector and responds with initial information and usages of all segments that contain any live data.

`LFS_GC_REQUEST_COLLECT` is a request to collect a specified segment.

`LFS_GC_REQUEST_ANSWER` is a reply to the `LFS_GC_INFO_PING` message described above.

5.10.2 Sending messages to the user space

We have already said that all messages must be reliably delivered to the user space. Reliable delivery, however, is always blocking. That is why messages are sent to the user space by a special thread called **gc-thread**. Every mounted file system has an outgoing message queue called `work_queue`[10] associated with it. New messages are usually added to it using the function `schedule_message()`.

`LFS_GC_INFO_WORK` and `LFS_GC_INFO_PING` messages may take a considerable amount of time to process because the garbage collector may issue collecting requests which are then honored in kernel but within its context. Even more importantly, processing those requests inevitably generates a fairly big amount of new info messages which can lead to an uncontrollable growth of the outgoing message queue. Furthermore, if the user space decided which segment should be cleaned before it processed all information about segment usage changes, it would be basing its actions on potentially outdated data. In order to avoid these two unpleasant situations, various info messages have different priorities and are inserted into the `work_queue` accordingly. Generally speaking, messages updating the internal state of the user space garbage collector take precedence over those that might result into more collection. All priorities are listed in table 5.5.

5.10.3 Processing cleaner requests

All requests from the user space are immediately processed by the *NETLINK* event handler `gc_request()` within the context of the cleaner. Registering a new garbage

[10] This queue used to contain collect requests as well, hence its name. However, currently it stores outgoing messages only.

Priority	Type
3	LFS_GC_INFO_REJECT
4	LFS_GC_INFO_SET
4	LFS_GC_INFO_SET_PARTIAL
4	LFS_GC_INFO_UPDATE
4	LFS_GC_INFO_WINDOW
5	LFS_GC_INFO_WORK
5	LFS_GC_INFO_PING

Table 5.5: Info message priorities. Smaller numbers take precedence over the bigger ones.

collector and processing the ping answers is fairly straightforward and therefore we will deal only with requests for a specific segment collection which are handled by `gc_request_collect()` which in turn calls the function `gc_collector()` to do most of the work.

This function proceeds in several steps. First, it checks the segment can actually be collected and is not empty. Secondly, it reads and checks the segment summary. Finally, it invokes `collect_data()` and `collect_inodes()` to collect blocks and inodes respectively. These functions determine the liveness of all blocks and inodes stored in the segment. Each live block is read, marked dirty and the associated mapping is placed into the **garbage queue** so that the segment building code can write it to a new location (see section 5.3.10). Each live inode has its `LFS_II_COLLECTED_BIT` flag set and is also added to the same queue. Note the inodes are not marked dirty because otherwise the Linux kernel might try to write it back to the disk itself which might occasionally lead to the inode being written twice. The purpose of the flag is also to prevent this from happening, it is cleared each time an inode is planned for writeback. Whenever an inode or a block is put into the queue, the reference of the inode is incremented so that it is never deallocated until the reference is decremented again by the segment building code after it has written the entity to a new location.

Finally, `gc_request_collect()` calls the function `lfs_write_gc_stuff()` (see section 5.3.10) so that the items stored in the *garbage queue* are immediately written.

5.11 Directories

The traditional layout of directories as a table of entries which grows when entries are added is deprecated. The drawback of the old approach is that the complexity of directory operations is linear. This results in a quadratic cost of an operation performed on all entries within a directory.

Modern filesystems adopted methods which allow to access an entry with a constant (or almost constant) cost. Most of the current file systems implementors decided to use **B+Tree** (JFS, NTFS, HPFS, XFS), **B*Tree** (HFS Plus) or some sort of **hash tables** (ZFS - extensible hash table).

Not surprisingly also the EXT3 developers had to deal with this trend and implemented an indexed directory structure called **HTREE** (as described in [7]) which is backward compatible with the traditional EXT2 layout, but offers the main features of an indexed directory. Moreover in case of a file system damage, there is a fall-back to the non-indexed operations.

For our **LFS** project we decided to uses a the **HTREE** with certain changes to its structure since we are not bound to the backward compatibility. **HTREE** is in fact a variation of a B+Tree, all data are in leaf blocks and all leafs are in the same depth. This chapter describes our indexed-directory implementation as well as discuss all major design decisions we took.

```
struct lfs_dir_entry {
        lfs_hash_t      hash;
        __u32           inode;
        __u32           iver;
        __u16           entry_len;
        __u8            name_len;
        __u8            file_type;
        char            name[LFS_NAME_LEN];
} __attribute__((__packed__));
```

Figure 5.11: Directory entry structure

5.11.1 Overview

As mentioned before our intention is to implement a directory layout that has an almost constant cost of file operations. Usually a file system with indexed directory structure handles very small directories in a different (unindexed) way avoiding the index overhead until the number of directory entries exceeds some threshold. *JFS* stores up to 8 entries (excluding `.` and `..`) in the inode without need of an additional data block whereas the EXT3 uses EXT2 layout for a single block directories.

In our design (inspired by the EXT3 HTREE [7]) we also begins with a single block directory without any indexing. There is a marker in the inode which tells whether the directory is indexed or not. If a new entry is being created, but there is no free space in the current data block, that block is split, data are equally copied to two new data blocks and an index block (*index root*) is created. The data blocks become leaves of the newly created index tree.

Since the index tree was introduced, each entry operation use it to resolve the leaf block in which the operation takes place. During the life of a directory, new entries are added or removed. Whenever a leaf block if full but an operation resolved it as a target for addition of a new entry, a block is split and an index which points to this block is added to an index block on an upper level of the index tree.

Initially a single (root) index block is created. Once this block is filled up with indices, it must be split and a new root is created. In such a process a single directory data block evolves into a multilevel index tree.

Last but not least it is worth to mention that in contrary to EXT3, LFS uses pages in the *page cache* when manipulating with directories instead of outdated *buffers (buffered IO)* which makes the code more readable and maintainable.

The following sections describe in detail all main features of our implementation. In Section 5.11.2 the on-disk structures are presented, in Section 5.11.4 we describe how the target data block is resolved and how an entry is looked up, Section 5.11.5 covers how an entry is added to a directory and how the index tree is created, Section 5.11.6 shows how entries are removed, Section 5.11.8 deals with journalling support for directory operations and finally Section 5.11.9 shows how a we handle directory read by an userspace application.

5.11.2 Structures

An on-disk directory is represented by so-called data blocks or leaf blocks which contain the directory entries and index blocks which (if used) keeps the structure of the directory index tree.

A directory entry (or *dentry*) is defined as `struct lfs_dir_entry` (see Figure 5.11). As you can see, the `name` is limited to `LFS_NAME_LEN` which is defined in `include/linux/lfs_fs.h` as 256. Since we include also the trailing `'\0'` the actual name length is limited to 255 characters. Actually the size of the `name` array is for userspace use (e.g. mkfs, fsck, dump_fs) and to denote the name length limit, but the size of the structure is not fixed. The length of the name is variable and is stored in the `name_len` item and

```
struct lfs_dirindex_entry {
        lfs_hash_t    hash_value;
        lfs_addr_t    block_no;
} __attribute__((__packed__));
```

Figure 5.12: Index block entry structure - index blocks contain number of such entries which maps hash values to block numbers

```
struct lfs_dirindex_block {
        __u32    entry_count;
        __u32    entries_used;
        __u64    padding;
        struct lfs_dirindex_entry  entry[];
} __attribute__((__packed__));
```

Figure 5.13: Index block structure - each index blocks has a header which determines how many indices can fit in this block and how many of them are used at the moment

is also determined by the trailing `'\0'`. The `name` array is rather used as a pointer for accessing the entry name. The actual size of a dentry is stored in the `entry_len`.

Each leaf block contains at least one directory record and it spans over the whole data block. This shows that a dentry does not contain only its data, but can include free space as well. Moreover, for alignment reasons we always round the size to **4 byte** boundary so up to 3 bytes might be wasted. Since the modern architectures have very slow access to any unaligned data, this is worth the cost.

When a new directory is created a single data block of this *empty* directory includes two dentries, for `'.'` and `'..'`. The latter is extended to cover the entire free space. As new entries are created and others are deleted, dentries are chopped into pieces of requested size or merged to avoid partitioning of the data block. This is covered in more detail by following sections.

As stated before, once there are too many dentries to fit in a single directory block, an index-tree structure is created. Each index block contains a number of indices (Figure 5.12) that point either to another index-block (Figure 5.13) or directly to a leaf (data) block. Details of the index-tree is presented in the following Section 5.11.3.

5.11.3 Index tree

As described in [7], HTREE leaves are all on the same level. Therefore the cost of resolving a block where an entry should be located, is the same for all dentries in a directory. Depth of the tree is kept in the directory's inode and helps to stop traversal from a root to the leaves. See Figure 5.14 for an index-tree example.

To be able to lookup dentries we assign a hash to each dentry. The hash is based on the entry's name. We opted for a 64-bit hash in order to minimize the amount of hash collisions and the length of hash collision chains. The hash function itself is adopted from the EXT3 filesystem.

Each index block contains a number of indices which map hash codes to a blocks which are either leaf blocks (a single block subtree) or a root index-block of a index-subtree. Only dentries with hash-code greater or equal to the one in the index ought to be found in that subtree. We keep the indices within an index block sorted in the ascending order of hash-codes. This allows us to use a **binary search** algorithm to lookup the right index. Because the hash in the index is lower bound for the hash-code within the subtree it points to, and since the same condition holds for the successive

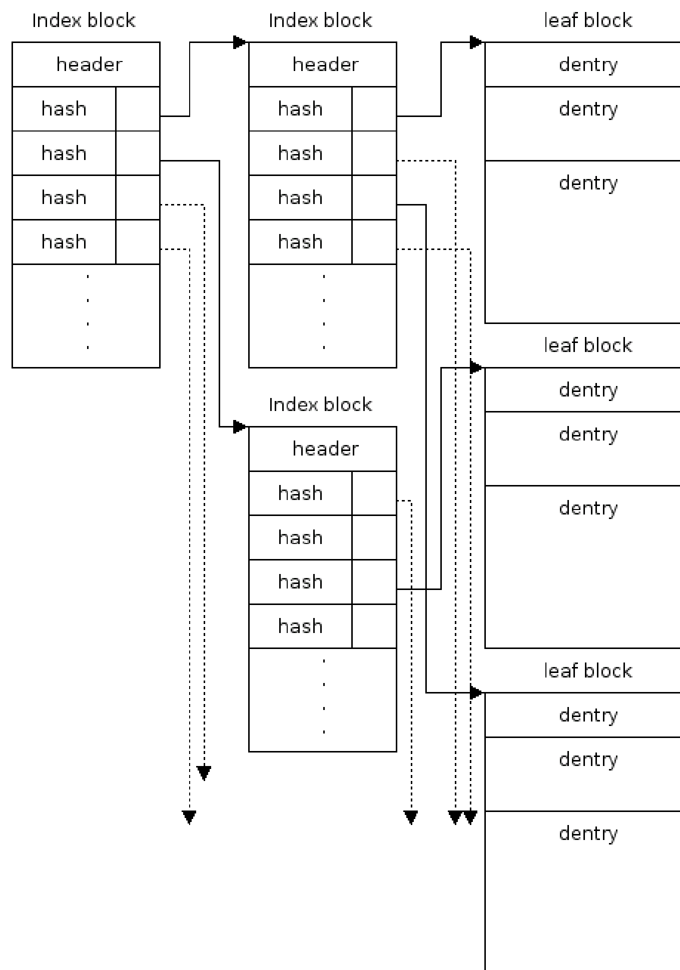

Figure 5.14: Index tree with two levels of indexing. Dotted lines point to blocks which are not show on this figure

index, the next hash can be used as an upper bound for the range of indices which can be found in an index-subtree. As a result, if the dentry is not found in this subtree, the dentry does not exist in this directory at all. There is a single exception to this rule.

Because of the hash-collisions, we cannot be sure that all indices which belong to the same hash-collision chain, fits in the same block. We use a single bit in the index hash field to denote that if a dentry is not found in this subtree, the lookup should continue in the next one (see Figure 5.15). In a case of a very long collision chain, it can span several index-subtrees. Perhaps because of the 64bit (63bit since one bit is used for marking the collisions) hash size and the good design of the hash function, we never encountered a collision-chain long enough to span more than one block, though.

Index tree limits

The tree depth is limited to 3 (2 index levels). This gives $indices_per_block^2$ leaf blocks, for default settings it is equal to $(4096/16-1)^2 = 65025$. This gives approx. 254MiB of leaf blocks. If we assume that each leaf block is 75% full and holds about 200 entries (see [7]), it results in approx. **13 million** of entries per directory. The theoretical upper bound for maximal size entries is 993814 of entries. It is unlikely that all leaf blocks are full as well as it is unlikely that all entries have the maximal name length.

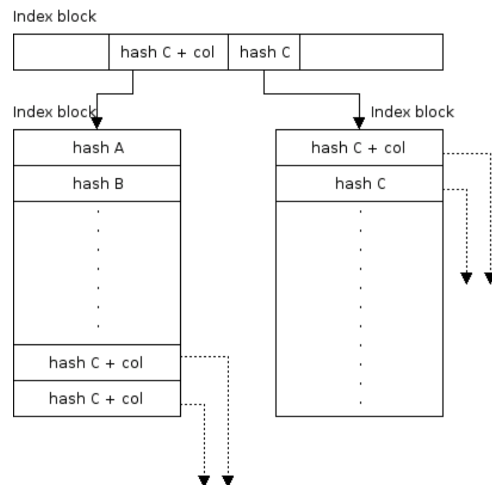

Figure 5.15: Hash collision chain spanning multiple blocks and multiple index levels. Every block but the last in the hash collision chain is marked with a collision bit. There is a collision chain on each index level. It helps to span collision chain across multiple index blocks. Dotted lines point to leaf blocks.

There is no practical objection to adding a third index level which would result in more then 3 billion of entries if anyone ever needs such a huge directories.

Compared to EXT3, we support less entries per directory because of the 64 bit hash and offsets. On the other hand this results in very short hash chains which do not occur very often at all.

5.11.4 Resolve operation

Resolving a hash-code to a block is the most frequently executed operation in the whole directory code. The core of this operations is `__resolve_leaf_blk_num()` function. In case of a single block directory, its code is trivial. The only one block is returned. The more interesting job is done in `resolve_leaf_blk_num_idx()` which searches the index tree.

`resolve_leaf_blk_num_idx()` is recursive. We are sure that the depth of the recursion is not larger than the depth of the index-tree. As mentioned before, the depth of index-tree is limited to 3 (or some other small number in the future), so the depth of the recursion is as well. The small depth of the recursion is why we use this rather than any stack-based non-recursive workaround.

This function can perform several slightly different tasks. The basic use is to lookup the leaf block with the first occurrence of a given hash. If a hash collision was detected, this function can be called in a loop with the last returned block passed in, so the function looks up the next block in a hash collision chain until the end of the collision is found. If requested, it can return the index block that points to the leaf block that was returned. This is useful, e.g. if we want to remove an empty leaf block and mark it in the index tree. This feature is not used though. Of course, this function can also return an IO error.

Lookup

Resolving a hash to a leaf block is only part of the `lfs_lookup()` inode operation since resolve always returns a blocks where the requested hash might be, unless an IO error occurred. To be sure that an dentry is present in the returned block, a traditional linear search must inspect that leaf block.

The search must be linear because dentries are not sorted within a leaf block. We decidedi so because of the high cost of keeping dentries ordered (See Section 5.11.5

on how dentries are added). Instead, we store the hash code of each dentry together with the dentry name in each dentry record. Therefore we do not have to do a string comparison if hash does not match. On the other hand, if the hash in the dentry is equal to the hash-code of the looked up record, we must compare the name string to be sure that we found the correct dentry and not just a dentry with the same hash in a collision chain. For details see the `lfs_name_to_inode()` which walks the leaf block and `lfs_name_match()` which does the string comparison.

5.11.5 Adding directory entries

Adding new directory entries is the most complex directory operation. First, the target leaf block must be found. If there is no space for a new dentry, the block must split and a new index is inserted in the upper index block. This may lead to a cascade of index block splits which can exceed the index-tree limits (as presented in Section 5.11.3 on page 51). This is reported to the user application as `-ENOMEM` We will now describe in detail all actions necessary to add a new directory entry.

Adding a dentry to a leaf block

Once a target block is resolved (as showed in Section 5.11.4) the `insert_dentry()` attempts to insert the dentry. We use the word *attempts* because there is no way to predict whether the insert operation will be successful or not.

The function starts at the beginning of the data block and linearly walks the list of all dentries present in the block until it finds a suitable location for the new entry or until the end of the block is reached.

As stated in Section 5.11.2 on page 50, the whole block is covered by `struct lfs_dir_entry` structures. Each dentry structure points to its successor with its `entry_len` field, creating a linked list within the data block. All unused space is included in dentries. Therefore suitable place for a new dentry record can be found in

- *unused dentry* - dentry which inode is set to `LFS_INODE_UNUSED` [11]. Such an unused entry must be so large enough that the new dentry can fit in.
- *used dentry* - live record which occupies more space than required. If the unused space within the dentry is large enough to fit in the new dentry, the current record is truncated to the necessary size and a new dentry is placed in the unused space. If some more unused space is left, it is included in the newly created record. See Fig 5.16

Because a directory leaf block becomes fragmented as new entries are added and removed, it might happen that there is enough unused space in the block, but no chunk is large enough. Since we want to prevent **unnecessary** block splitting, we defragment such blocks.

When walking the linear list of all dentry records, `insert_dentry()` counts the number of unused bytes. Because no suitable space was found, the end of the list is reached and all space is accounted. If the space i large enough for the new dentry, `block_defragment()` is called. Once the block has been defragmented, we try to add the new entry again. This time success is guaranteed.

`block_defragment()` just pushes unused space to successive records. It is finally gathered in a single empty dentry record at the end of the block. For details see Figure 5.17 and the code in `src/dir.c`

Building the index tree

As stated before, each newly created directory has a single leaf block. When adding a new dentry in this block (as described before), sooner or later it will happen that `insert_dentry()` returns `-ENOMEM` because there would not be enough space even after defragmentation (defragmentation is not executed because it would be useless).

[11] this macro, as defined in `include/linux/lfs_fs.h`, is equal to 0 since the **glibc** library is biased to **EXT2** and expects unused inodes to be set to 0.

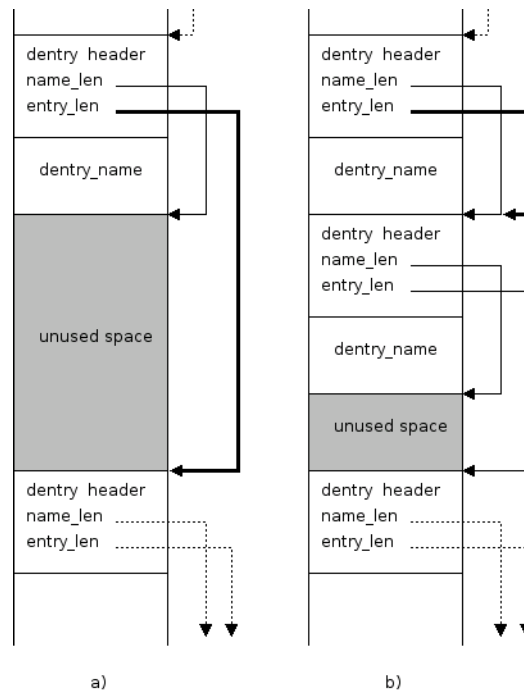

Figure 5.16: If there is unused space in a dentry record which can be reused for a new dentry (a), the record is split and the new entry is inserted (b) The thick arrow shows how the original dentry is changed to point to the newly inserted one

In this case the single leaf block is split in two and a new index root block is allocated. Approximately half of all dentries in the original block are copied to each of the new leaf blocks and indices of both blocks are inserted in the index root (See Figure 5.18). Although copying data from one block to another is expensive, we benefit at least that the block is defragmented while data are copied. `copy_half()` copies dentries from the original block to the new one and deactivates the original dentries by setting their inodes to `LFS_INODE_UNUSED_LE32`. Unfortunately this cannot be done simultaneously because of page locking. In case of a different block and page size (note that block size is always less than or equal `PAGE_CACHE_SIZE`), we might access blocks which resides in the same page. This is exactly the case when splitting a single block directory as at least one of the new blocks is allocated in the same page as the block that is being split. Therefore data are copied first and the original block must be walked once again to deactivate the copied dentries.

If an index tree already exists and a new dentry is being added there are basically three scenarios :

- *success*
  - target block is looked up, there is space enough and the new dentry is added
- *target leaf block is full*
  - in the simpler case the target block is split and there is a free slot in the parent index block to add a new index which points to a new leaf block.
  - the parent index block is full and must be split as well. This is accomplished in a similar manner to splitting of a leaf block. Inserting a new index to the parent index block may lead to a cascade effect until the root is split too.

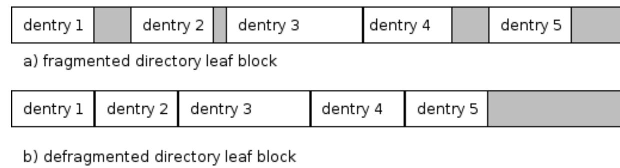

Figure 5.17: If there is enough space for a new dentry in a leaf block, but the space is fragmented in small chunks (a), the block is defragmented and all unused space is gathered to create a new large unused area (b). The gray areas stand for unused space, i.e. space to be collected

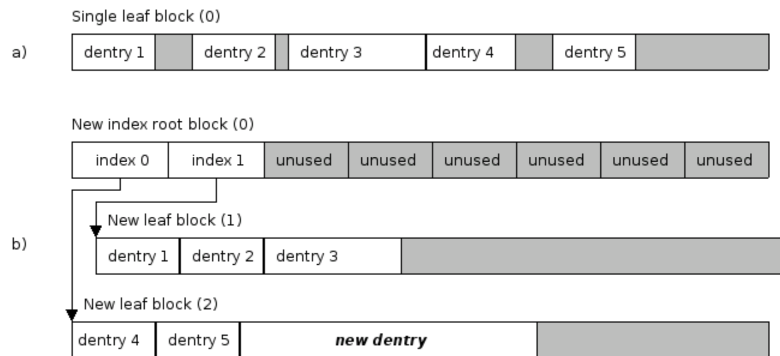

Figure 5.18: If there is not enough space in the single leaf block (a) a new index root block and a new leaf block are created, data are divide among both blocks and a new entry is added (b). The numbers in braces denote the block numbers within the directory file. Note that the original leaf is overwritten by the index root.

Splitting a leaf block

When splitting a leaf block, there is a special case of a single block directory (first split), which is a has no index block. It is just a single leaf block and must be handled accordingly. The difference is that data from the original block must be moved to another location within the file because index root must be always in the first block of the directory, so we can easily find it. In any other case the block that is being split is not moved, approximately half of its entries are copied to another block and the original block is defragmented. The core of this operation is implemented in `__split_leaf_block()` function.

The strategy for splitting a leaf block is fairly simple. Function `map_create()` reads the whole block and creates a so-called map of that block. The map contains an address of each dentry in the block together with its hash. Here we benefit from the fact that the hash is stored in each dentry so that we do not have to calculate it. After the map is created, it is sorted in `map_sort()` in ascending order of hash codes. This is necessary because of the fact that hash of records which are about to be copied to the new block must be all greater or equal than *some* value which is stored in a new index. The upper half of the hash codes is copied to its new location using `copy_half()`.

This method proved to work fine. It assures that there will be space enough for a new entry except in certain corner cases, where it is not technically possible, e.g., in a case of smallest block size of 512 bytes if there is already a dentry with maximal name length of 256 characters. Adding a new maximal size dentry is not possible. In such a case `-ENOSPC` is returned. We do not treat this as a significant problem since it is the administrator's task to create a filesystem that suites his needs.

Of course, there are some obvious improvements to this method that are considered

future work. For instance :

- Copying approximately so many dentries that half of the space in the original block is freed, instead of copying half of all dentries. This will help keep the index tree more balanced
- Trying to avoid splitting a hash collision chain would keep the cost of the resolve operation low.

Splitting an index block

After a leaf block is split, the index which points to this block must be placed in an index block. This operation, performed by `insert_index_into_block()`, called from `insert_new_index()`, is fairly simple if there is a free slot in the index block. To be able do check it quickly, each index block has a header (See Figure 5.13) that stores how many slots are in this block and how many are actually used.

As stated before, all entries in an index block are ordered, so we use a binary search to find the position where the new index should be inserted. Function `insert_index_shift()` shifts the rest of the indices and places the index at the freed position. The whole process is illustrated on Figure 5.19

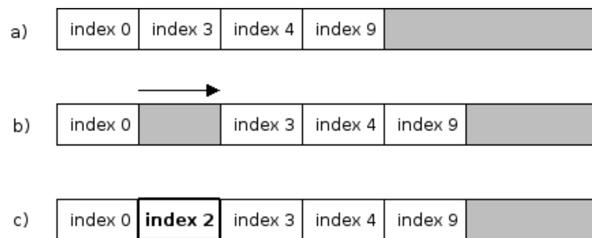

Figure 5.19: When inserting a new index (**index 2**) to an index block (a), first the new position is looked up and freed by shifting the rest of the indices (b). Then the index is placed to its position (c)

The task of `insert_new_index()` is to handle a cascaded split. As stated before, it might happen that there is no free slot left so that the index block must split. Call to `split_idx_block()` returns the index of the newly created index block and a recursive call to `insert_new_index()` inserts it to the upper index block. Perhaps this index block must also be split and so on. The recursion continues until the index is successfully stored or until the index root is split and a new index-root is created. There is always space enough and the recursion stops. Figure 5.20 shows such a situation.

In Section 5.11.4 we mentioned that we use recursion for traversing the index-tree when resolving hash-code to a leaf block. Maximal depth of the index tree limits the depth of this recursion (See limits in Section 5.11.3 on page 51). The same condition holds for the *bottom-up* recursion of the cascaded split.

Because we need new blocks in the cascaded split, we must be sure that we do not leave the directory structure in an inconsistent state if we run out of memory. Therefore `insert_new_index_probe()` tests to which extent the directory will be split and tells how many blocks to preallocate. Last but not least, it detects whether the maximal depth of the directory is going to be exceeded and if so returns error code `-ENOSPC`. This signals that there is currently no space in this directory for the entry. It does not mean that another (different) entry cannot be inserted and by doing so, the index-root must not be split. Unfortunately, this information is opaque to the userspace application. References to the preallocated blocks are passed to the splitting-part of the directory-code so that no dangerous allocation is necessary. If preallocating of necessary number of blocks fails, so does inserting of a new entry.

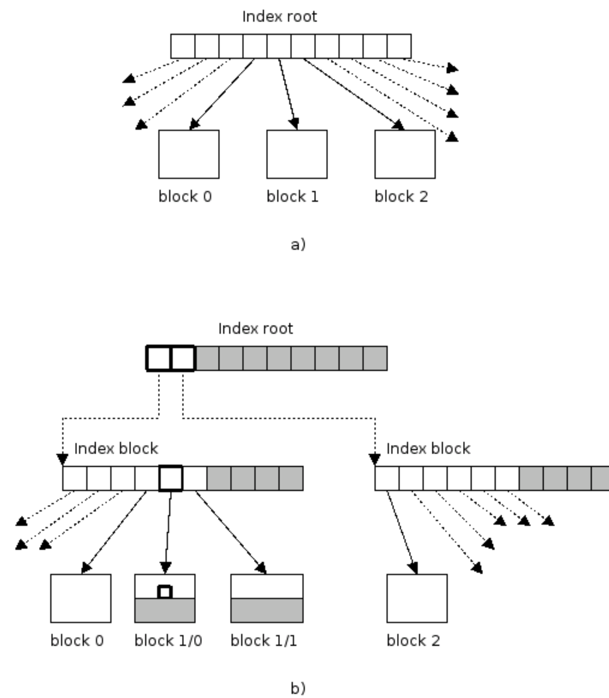

Figure 5.20: If a new dentry is to be inserted into the leaf **block 1** as presented on (a), that block must be split in two blocks **block 1/0** and **block 1/1** and a new index must be placed in the index root, which is full. Therefore a new index root is created and the original one is split in two index blocks (b). Gray zones stand for unused space, whereas newly added records are highlighted by thick borders.

5.11.6 Removing directory entries

Comparing to adding a new directory entry, removing an old one is significantly simpler. We opted for solution which is not shrinking the size of the directory since most entries are removed not from last block in the directory file (and so we could truncate the directory once the last block is empty) but rather from random blocks within the file.

To remove or delete an entry, it must be looked up as described in Section 5.11.4. Subsequently, the entry is marked as unused by setting its inode item to `LFS_INODE_UNUSED`. Doing only this would result in fragmentation of the directory leaf block. If there are *unused* neighbors, we merge them with the newly released entry to create a larger unused chunk within a block. This chunk can be more easily reused by following add/create directory operations. The core of the remove operation is implemented in `lfs_remove_entry()`. This function is reused in all directory operations that remove entries from a directory, which include

- `lfs_unlink()` - removes an entry which points to a non-directory file
- `lfs_rmdir()` - removes an entry which points to a directory
- `lfs_rename()` - removes an entry from its originally parent directory and creates a new one in another directory that points to the same inode. It also removes the destination entry if it exists and it is not a non-empty directory.

The described method of removing directory entries has drawbacks too. The most obvious is that we do not shrink the directory's size if some leaf blocks are emptied. This can result in a huge directory (in terms of used bytes on disk) that contains only several entries, which might fit in a single block directory. As well known and

extensively used filesystems do not deal with this problem either, we consider this as a minor issue. It is unlikely that applications that create a large number of entries will not remove the whole directory once its entries are not used anymore.

Another problem is that the index-tree structure is more or less static. It means that once a tree is created and indices are distributed among index-blocks the only operation that can possibly change such a distribution is adding new entries (and splitting tree nodes). Removing entries does not affect the tree structure at all. Here we highly benefit from the well designed hash function[12]. It distributes entries very well so even a different work-scheme on previously emptied directory keeps the usage of both index and leaf blocks balanced.

We considered an option of releasing leaf blocks. The problem is that the **Linux** *VFS* layer makes releasing blocks within a file difficult. Another space optimization would be merging and releasing index blocks. This would affect the overall performance of directory operations significantly. In our opinion this might be a task for a *Garbage collector* or similar process which can compact directories once the system is idle.

Both extensions remain future work.

5.11.7 Symbolik links

We support **fast** and **slow** symbolic links in the EXT2 fashion. Fast symbolic links store the destination path in the the `struct lfs_inode` area which is reserved for block addresses. Since we use 64 bit addresses, there is space for up to 192 characters. If the path is longer, it is stored as a slow symbolic link in a single data block.

5.11.8 Journalling

While our filesystem works as a log, we need journaling for directory operations. The reason is that we cannot be sure, which data are going to be written to the disk and when. Moreover the disk itself can reorder writes of the data. If the system crashes, roll-forward needs to know which operations were performed since the last *sync* to be able to remove entries if inodes are missing.

Before a directory operation starts changing data blocks, it tells the journalling subsystem (See Section 5.12) which operation will change the directory. The following operations, defined in `include/linux/lfs_dir.h` are journaled :

```
LFS_JOURNAL_OP_CREATE
LFS_JOURNAL_OP_MKDIR
LFS_JOURNAL_OP_MKNOD
LFS_JOURNAL_OP_LINK
LFS_JOURNAL_OP_UNLINK
LFS_JOURNAL_OP_RMDIR
LFS_JOURNAL_OP_RENAME
LFS_JOURNAL_OP_SYMLINK
```

Journal is divided in so-called journal lines (or jlines) of variable length. The length depends on the operation that is journaled. In general, each line has a header which describes the operation and its variable parameters. Figure 5.21 present the header of the on-disk journal line. The header is followed by one or two strings depending on how many names are used for each operation.

We can divide all operations into three groups :

- *simple* - all operations which involve only a directory and a single entry. These operations use only one string in the journal line.
  `create`, `mkdir`, `mknod`, `link`, `unlink` and `rmdir`
- `rename` - this operation does not only change the location of an entry in the directory tree, often it changes the name as well. Both the old and new name must be journaled

[12] copied from EXT3

```
struct jline {
        __u16           opcode;
        __u16           jline_len;
        __u32           inode[3];
        __u32           inode_ver[3];
        __u16           name_len[2];
} __attribute__((__packed__));
```

Figure 5.21: Journal line on disk representation

- `symlink` - this operation needs to log the name of the new symlink as well as its target

Operation `rename` needs special care if the destination already exists. The expected behavior is that unless the target is a non-empty directory, the target is removed and substituted by the moved entry. Such a situation needs to journal both `rmdir`/`unlink` and `rename`.

Core of the directory journalling is `lfs_journal_dir_writeout()` which creates the journal line and passes it to the journalling subsystem. Extra care must be taken if a journal line spans across a page border. There is a state machine inside the `lfs_journal_dir_writeout()` which fills the actual page and asks for another page.

5.11.9 Readdir

The readdir operation is used for sequential reading of all directory entries. It is not directly exported to userspace even though there is a `readdir()` function in standard libraries. Its purpose is to return a single entry per call. Asking the kernel to return a single entry each time is not efficient since crossing kernel-userspace boundary is expensive. Therefore the **Linux** kernel exports the `getdents()` syscall which gets a buffer from userspace an fills it with as many entries as possible and let the **glibc** implementation of `getdents()` deal with system specific differences of how to get data to userspace buffers. Glibc's `readdir()` returns a single entry from that buffer. We will now discuss how the directory entries are read by *LFS* and how they are returned to userspace.

Like the rest of the **Linux** *VFS*, `readdir()` expects behavior similar to **EXT2**, therefore our implementation is not straight forward but deals with similar problems as **EXT3** with indexed directories. There are certain differences.

The main problem is that *file position* in the traditional meaning does not make sense. If we were sequentially walking a directory that is being changed by another process, we might see some entries more then once. The reason is that some entries could have been copied to another block because of an index-tree split. Even if we were able to skip index blocks (which do not contain any entries) we cannot determine if some entry was already returned or not.

We compared behaviour of LFS and EXT3 on a directory with 500.000 entries. We used an old glibc's implementation of `readdir()` which performes more then 4.000 seeks. LFS returns two duplicities whereas EXT3 only one.

In our approach, we substitute the real file position by the hash value of the actual directory entry. This value expresses where such an entry can be found. If the directory is walked in the hash order, it is simple to tell whether the given entry was already sent to a user application or not by comparing its hash value with the actual one.

Another issue is to determine where to continue reading when `readdir()` is called. There are two situations :

1. As stated before, each call to `readdir()` returns just as many entries as many fit in the given buffer. To read the whole directory, a user application issues several calls. In each invocation we need to know where to continue.

2. Just as it is possible to seek in a regular file, it is also possible to seek [13] in a directory. After seeking, readdir should continue from the new position.

Hash as a file position

As described before, we use a hash instead of a real file position. This allows us to determine position in the sequence of entries sorted by hash codes. The drawback is that this is not unique since there are entries with the same hash code in case of a hash collision. So it is not possible to tell exactly which entry is referenced.

To reduce the probability of hash collisions we opted for a *64 bit* hash. It is problematic to return such a large file position in `readdir()`. Even though all related kernel functions use `loff_t` which is a 64 bit integer type, **glibc** expects 32 bits only. Therefore we return only the most significant bits. As a result seeking is not precise. But as there might be many concurrent writer [14] the application cannot count on that the directory is in the same state all the time, unless some other means of synchronization or mutual exclusion are used.

Stateful readdir

As described earlier, an application calls `readdir()` as long as there are some data to be returned. Each time, as many as possible entries are returned. Only EOF is signaled by returning no entriy. We need to remember where to continue after being called again. For earlier mentioned reasons, file position (e.g., as used in EXT2) is not enough since this information is not accurate enough. Therefore we attach a private data structure to the `struct file` structure (See Figure 5.22), where we keep all important data between successive `readdir()` invocations. It includes last file position (to be able to detect seeking), current hash to be processed, entries of the block which is being processed and a list of all leaf blocks. The last two items are described in more detail later on.

Because of the data stored in the private info structure, we need to detect if the application changed the actual file position (seek, etc.). To do so, we keep the last file position just before returning from the previous `readdir()` call. If *seek* is detected, the private data structure must be reset and refilled accordingly.

Similar measures are taken if a directory is changed while being read. We wish to return the most recent state of the directory, but this is not always possible. If a new entry with a lower hash than the current is added, this entry is not to be returned without seeking or rewinding that directory. Returning such an entry would violate the hash order. Moreover adding new entries to a directory may split some blocks and some of the entries wouldn't be returned at all. For this reason, we need to reload the private info if the directory changes. We keep an internal version of the directory. If it has changed since the directory was opened or since the private info was reloaded, it is time to reload it again.

Returning entries in a hash order

As stated before, we decided to return all entries in hash order rather than in file position order. The directory is partially ordered because of the index-tree structure. As described in Section 5.11.4 records in index blocks are ordered, contrary to leaf blocks which lack any sense of order. As a result we read the leaf blocks in the hash order and sort the content of a block when the block is being processed.

Leaf block list

When `readdir()` is called for the first time, no private info is attached to the directory yet, we create a list of all directory leaf blocks. There are several reason to do this :

[13] Seeking is limited to usage of `telldir()`, `seekdir()`, `rewinddir()` and `seek()` with `SEEK_SET` only.

[14] Access to a directory is exclusive, but `readdir()` calls can be interleaved with calls by other applications that change content of the directory.

```
struct dir_private {
        loff_t                          last_pos;
        lfs_hash_t                      min_hash;
        lfs_hash_t                      max_hash;
        lfs_hash_t                      curr_hash;
        lfs_hash_t                      next_block_hash;
        struct rb_root                  root;
        struct rb_node                  * curr_node;
        struct fname                    * curr_chain;
        struct idx_list                 * idx_list;
};
```

Figure 5.22: Readdir private info structure

- It is expected that the whole directory will be read. Our sulution walks the index-tree once instead of looking up each leaf block separately.
- It is necessary to know the minimal hash in the next block to know where to continue after the current block is finished. It is trivial to continue with the next block in the list. Without such a list we would have to read possible two index blocks at once to get this information.
- If we do not want to output the most recent content of the directory while the directory is changed, such a list is a semi-snapshot. All blocks in the list are valid all the time, the order of the blocks never changes. The only change is that some of entries could have been moved out to other blocks if a split occurred.

The list of block indices is created in `fill_idx_list()` which walks the index-tree in depth-first-search and adds an index of a leaf block to the list. The list itself consists of a linked list of pages. Each page is allocated in `alloc_idx_list_item()` Each page contains a header (See Figure 5.23) and an array of indices. When adding a index of a block to the list, we call `get_dir_block()` to get the block to memory.

This list is attached to the directory's private structure. As `readdir()` is called, the first item in the list is removed and processed. After a block is finished, `put_dir_block()` is called to release a reference to the block. Once all indices are removed from a page, the page is released in `free_idx_list()` too. `free_idx_list()` releases all items on the list. In most case just a single-item list is passed to this function, unless a reset of the whole private structure occurs

```
struct idx_list {
        int dir_block_cnt;
        int dir_block_used;
        int index;
        struct idx_list * next;
        struct idx_dir_block dir_block[];
};
```

Figure 5.23: Item of a leaf block index list

Sorting a leaf block

We need to sort all entries of a block in hash order. We adopted an EXT3-like approach. Its core is a *red-black tree* (rb-tree) where each node represents a single hash

value and its collision chain. We use the generic rb-tree implementation from the Linux kernel. The rb-tree is sorted when new nodes are inserted.

Initially the rb-tree attached to the private info is empty. Everytime the rb-tree is empty, `lfs_fill_rb_tree()` is called. It picks up a block from the index list (as described in previous subsection) and calls `block_to_rb_tree()` to fill the rb-tree with data from the leaf block. If there is a collision chain that overflows to another leaf block, the whole chain is read and inserted into the rb-tree. If in the meantime some entries were removed from the directory, it might happen that there is an empty block in the list. That block is skipped.

The rb-tree is reference through the `root` item of the private data structure. `curr_node` points to the node which is currently processed and `curr_chain` is the first item in the hash collision chain which will be processed next. For details see Figure 5.22 and the `src/dir.c` source file.

Filling the userspace buffer

The core of `readdir()` implementation is a loop which calls `filldir()`. From the point of view of `readdir()`, `filldir()` is a **blackbox** wrapped in `lfs_filldir()`.

The main loop is responsible for setting the current node of the rb-tree. If the rb-tree is emptied, it is refilled again. In contrary, `lfs_filldir()` processes a single hash chain. It calls `filldir()` for each item in the chain and if an error is returned, it sets the `curr_chain` so next the `readdir()` can continue from here on. Notice, that the before mentioned error is not fatal and its only purpose is to signal that the userspace buffer is full.

Releasing private info

Because we allocate a private data structure that we attach to the directory once the `readdir()` is called for the first time, we must be sure that those data are freed by `lfs_release_dir()` file operation once the directory is closed.

5.12 Journaling subsystem

The journaling subsystem can be found in `src/journal.c` and its purpose is to provide the directory operations with a means to store *jlines* (see section 5.11.8) and enable the segment building subsystem (see section 5.3.5) to flush to disk those of them which are relevant to the currently written entity. Internally, the subsystem is little more than a queue of pages, a current page to which new entries are being added and a counter of *jlines* to uniquely identify them.

5.12.1 Interface for directory operations

Directory operations start creating *jlines* by calling `lfs_journal_grab_entry()`. This function returns the pointer to the current position in the current page and how much space there is left in it. If there is no current page a new one is allocated. This function also locks a mutex so that no journal entries ever interleave and the subsystem state is protected from race conditions.

Conversely, when a directory operation is done with constructing a *jline*, it calls function `lfs_journal_release_entry()` so that the journaling subsystem can update its state, in particular the position in the current page and the number of the next *jline* and release the mutex. If the current page has been entirely filled by the committed *jline* it is queued. Finally, the subsystem checks whether the journal pages take too much space and if it is so, the segment building code is asked to plan a few of them and finish a partial segment so that the memory they occupy can be freed (see section 5.3.1). The function returns the number of the finished *jline* so that the directory operation can assign it to corresponding fields of relevant inode infos.

Nevertheless, the *jline* a directory operation wishes to produce may not fit into the remaining space in the page. The operation must handle this by splitting it into two parts so that the first one exactly fits into the rest of the current page and ask for a

new pointer by calling `lfs_journal_next_page()` which enqueues the current page and allocates a new one so that the rest of the *jline* can be placed into it. This function keeps the journal mutex locked.

5.12.2 Interface for the segment builder

Whenever the segment building code processes an inode, mapping or even a single page, it must acquire and plan plan all pages containing *jlines* with a number that is less than or equal to the one stored in the inode info. These are obtained by calling `lfs_flush_inodes_journals()`. In addition, if there are too many journal pages stored in the queue, this function passes some of them on to the caller too. The caller then processes the returned part of the queue and plans it with respect to plan structures, remaining blocks in the segment and so on.

Chapter 6

Snapshot

Because LFS stores new copies of data in another place on the disk the implementation of a snapshot is relatively simple. We must not allow to mark data as `FREE` that are dead and in same time used by snapshot. Of course there are problems with implementing it in Linux and how to reuse most of already written code.

6.1 Implementation

The most simple way of implementing snapshot in Linux appears to be creation of new file-system type. This file-system has no backing device and during its initialization it references a host file-system instead. So it can use host file-system's structures like `lfs_sb_info` without fear that they will be freed.

Snapshot finds its host file-system by an id that is provided in the mount time. It grabs it with code similar to kernel function `grab_super()`. From that point on the host file-system cannot be umounted. Even if it is umounted from all of its mount-points it is still mounted in kernel. After the snapshot is umounted `deactivate_super()` is called. It drops active reference to the host file-system.

When mounting a snapshot we must sync the host file-system and remember the last segment's `segment_counter` and address of the ifile inode. Snapshot creates its own ifile inode with the same address and since it is read-only, it never changes.

The last thing the snapshot does is stopping garbage collecting of any segments that were not empty before the last sync before the snapshot was taken. Message `LFS_GC_INFO_WINDOW` with last `segment_counter` is sent to the garbage collecting process. It is not an error to collect data, which were created before the snapshot. Since these segments are not freed while the snapshot is mounted, such an operation would only waste space and not free any segment.

All free space tracking data are part of `struct free_space` defined in `free-space.h`.

6.2 Free segments tracking

We test if a snapshot is mounted in `lfs_update_free_segs()` . If so, we test `segment_counter` if segment was written out before or after the snapshot was taken. If it happened after, the segment is marked as `FREE` as usually. Otherwise it is enqueued into the `struct lfs_sb_info::snap_delayed` queue and freed after the snapshot is umounted.

There is also a special atomic value `struct lfs_sb_info::snap_frozen` that locks freeing process. Freeing process needs to be locked before snapshoting the host file-system until the `segment_counter` is determined.

6.3 Free space accounting

After a snapshot is mounted we need to determine new values of `free_space::free` and `free_space::max`. We take actual value of `free_segments` and counts maximal

free space value in the same fashion as when the host file-system is mounted. A new value of `free_space::free` is equal to the new value of `free_space::max`, because there are no live data in area that is usable after snapshot.

Old values of the free space accounting are saved.

Depending on the fragmentation, the resulting value of the free space accounting can be even greater then the value before the snapshot. In such a case we take the value before the snapshot. If the space is used according to the new value, there would be problems when umounting the snapshot.

When some space is returned while a snapshot is mounted, we must test carefully whether the space was obtained by freeing data created before snapshot was taken or after. This can be determined from address of a block or an inode. There is one problem remaining. Once an inode or a block is dirtied we must acquire additional space for it. But old space cannot be freed. Instead it must be added to `free_space::delayed`, so-called free space transfer. Of course this can happen only the first time data is accessed. When they acquire new free space from the area after snapshot, they must not be transfered any more, until snapshot is umounted.

Addresses cannot be used to determine whether data was transfered or not, as data is written asynchronously to the free space accounting. It is required to use another way how to track whether data was accounted in free space before or after the snapshot was mounted. It differs for a snapshot and for direct or indirect blocks.

6.3.1 Inodes

`segment_counter` for each inode is kept in `struct lfs_ifile_info`. Once an inode is created, its segment_counter is set to 0. When an inode is written to the disk, its `segment_counter` is updated with `segment_counter` of the currently written segment. When an inode is read from the disk again, its `segment_counter` is initialized from the segment usage table in the `ifile`. Finally, when the inode is transfered we zero its `segment_counter`.

Transfer is done only if a `segment_counter` of an inode is non-zero. Because all data is synced before taking a snapshot and thus counters are updated, there is no way how to transfer data written before the snapshot was taken. Because the only operation, which may change counter to non-zero is an operation that writes inodes, we know that all data accounted from free space after snapshot have `segment_counter` set to 0 or to something greater than the segment_counter of the last snapshot.

6.3.2 Blocks

There is no reasonable way how to attach something like inode's `segment_counter` to buffers, therefore a different approach must be taken. The preallocation bit in **address** does the job. This preallocation bit is used to mark data blocks as dirty and preallocated (see Section 5.1.3). This flag can be reused for deciding if a block was subtracted from the free space after a snapshot. When a block is dirtied it is checked if it is preallocated too. If not, it must be preallocated and checked whether its `segment_counter` is smaller than the snapshot `segment_counter`. If so, its space is transfered. In any other case the block was written after a snapshot and so it must be accounted in free space after the snapshot. As for inodes, the only way to clear the preallocation flag is to write block to the disk. This operation updates address of that block and moves it to a segment with a counter greater than the snapshot. Each subsequent dirtying of this block will not issue a transfer because of a new value of the counter.

6.4 Working segment

Free space accounting, as described in the previous section, needs to be able to simply decide if a block was created before or after a snapshot. It is mandatory for correct work. It can be easily determined using a `segment_counter`. But the segment counter has smaller granularity than the sync operation that typically finishes a partial segment. The simplest workaround for this problem is not to allow sync to create partial

segment. So when the sync operation is required by a snapshot a whole segment is always written out. As a result, the *working area* is completely in area behind the snapshot.

6.5 References

For more details, see the code in files `snapshot.c`, `snapshot.h` and `free_space.h`.

Chapter 7

Utilities Implementation Overview

7.1 mkfs.lfs

The userspace utility `mkfs.lfs` is used to build a LFS file system on a block device, usually a disk partition. You can see the commandline usage below. `mkfs.lfs` supports many more options than described here, but for the reasons mentioned in the introduction, a file system with only a well tested default setting is built. Of course any setting supported by the `mkfs.lfs` creates a correct file system.

```
mkfs.lfs [-L volume_name] device
```

`-L`	user specified name for a new volume (default is *LFS*)
`device`	device where the new LFS volume should be created (e.g., /dev/hda5 /dev/sda1 /dev/hdb)

`mkfs.lfs` 's only task is to build a new empty volume and setup all on-disk structures (for details on structures see Sec. 4) so that the volume can be mounted by the Linux kernel. Creating a new LFS volume is a little bit more complicated than a static file system like the most popular *ext2* or its descendant *ext3*. `mkfs.lfs` does not only write *segment headers* and *superblocks* to the disk, but also has to write initial information to *.ifile* and the root directory. Since the file contains information about *segment-usage* and the *inode table*, writing this file to the disk changes the information which is included in the file itself. Therefore a special measures must be taken. The whole process is divided into three phases. In the first phase the root directory is written, next the *.ifile* is created and written. After its inode is know, writing the *superblocks* and empty segments (non-data segments) finalizes the whole process. Detailed description of all phases follows.

7.1.1 Initiation

The first step is to determine the size of the target device, because it determines positions of the superblocks as well as the initial size of the *.ifile* . As already mentioned, the *.ifile* consists of two parts (see Sec. 4.5 and Fig. 4.3). First part is a static table with an entry for each *segment*, whereas the other part is a dynamic inode table. As `mkfs.lfs` creates a file system with just a root directory and the *.ifile* , the inode table has a known size with just two inodes.

We are trying to keep the mkfs-code as simple as possible. Because we are about to write only a limited amount of data (.ifile, root) to the beginning of the disk, we use `mmap()` to remap first few megabytes of the device to the address space of the `mkfs.lfs` process. For the sake of simplicity we support *.ifile* with double indirect blocks only,

which limits its size to approx. 1MiB. This implies that the largest supported volume is **64 TiB**. This is enough for current hard drives as well as for the near feature. If support for larger devices is needed, it is possible to add this.

We were considering whether to use `mmap()` or `seek()` and `write()` but for the above mentioned limits and since the `mkfs.lfs` supports different *segment* and *block* size configurations, it is very much simpler not to deal with write-buffers and be able to access the written parts of disk randomly. We benefit from random access especial when writing out the *.ifile* as the finfo structures must be updated. On the other hand we use `seek()` and `write()` for writing non-data segments farther from the beginning of the device. Both approaches are described in more detail in Sec. 7.1.4 and Sec. 7.1.5

Ifile initiation

After we know how large the *.ifile* should be, we allocate its in-memory representation. Throughout the whole process of a file system creation the *.ifile* is updated and read until finally written to the disk. The most important for understanding the following text is that the *segment usage table* is also initiated. Positions of the superblocks is fully determined by the size of the device. The inode table carries information about which segments are reserved and which are used for superblocks as well as which segment where written by data. *.ifile* is described in more detail in Sec 7.1.4.

7.1.2 Write functions design

This section describes the general structure of functions used to write data to the device. Currently only root directory and *.ifile* is written, but it gives us an opportunity to extend the `mkfs.lfs` abilities to include more data and structures if needed.

Each write function takes as a parameter number of the first segment where to write. Because of the *log* nature of this file system, we write contiguously to successive segments as `mkfs.lfs` , to certain extent, simulates writing by the kernel module. Therefore each of the functions which write data to the device return the number of the last written segment. This, incremented by 1, can be passed to a next write call. It is legal to write a segment only partially, so we do not have to check if the last segment was fully written, but the new write can start in the next one. This simplifies the write code very much. Leaving gaps in segments is not an issue since data will become changed and moved to another segments and those abandoned will be reclaimed by the *garbage collector*

Another write issue is related to the *superblocks*. No data is written to segments that are reserved for superblocks (see Sec. 4.2 on page 17). Therefore each function must take care of that and skip them. It is simply achieved by reading the *segment usage table* from the in-memory *.ifile* .

7.1.3 Root directory

Function `write_root()` writes the root directory together with its inode to the first data-segment (no data segments were written yet). Actually, the first available segment must be found since there might be some reserved segments at the beginning of the device, perhaps followed by the first superblock.

In this operation only a single segment is written as the root directory has only one block and only one inode is place in this segment. As the minimal size of a segment is 256KiB, there is space enough for both.

Data written to this segment are basically static, or better always the same regardless of settings. First, the segment summary is filled, a single block root directory is written to the first data block (`.`, `..` and `.ifile` entries) and the inode of that directory is placed in the next block (inode number `LFS_INODE_ROOT` as defined in `include/linux/lfs_fs.h`) and the only *finfo* record is filled. Finally, the segment usage table is updated accordingly and root's inode address is set in the inode table.

More details can be found in the source code in `utils/mfks/mkfs.c:write_root`

7.1.4 Ifile

Ifile completition

The *.ifile* holds the up to date information about the data written to the disk. Unfortunately writing the *.ifile* to the disk changes the information inside the *.ifile* itself. Moreover updating the *.ifile* is complicated by the fact that the affected section of the *.ifile* is most likely already written to the disk and would have to be looked up and changed.

We opted for a different approach. Since we know how large the *.ifile* is that we are about to write out to the device, we can determine which segments are going to be written and how many data are going to be used in each segment. Segments that will not be written are marked as empty in the *segment usage table*. All this is accomplished in the `fill_segment_usage_table()` function in the `utils/mfks/ifile.c` file. We do not have to figure out what the address of the *.ifile* inode will be, since the inode address is not written to the inode table. Address of this inode must be written to the *superblocks*, because there must be a way how to find the *inode table*, at the moment of the file system mount. The inode table is stored in the *.ifile* .

Not only the data part of the *.ifile* must be written, but also the indirect blocks in the case of a large file. Addresses are not known prior to the actual write out.

Ifile in-memory structure

```
struct ifile {
        unsigned                            inode_cnt;
        unsigned                            seg_cnt;
        unsigned                            size;
        unsigned                            dblocks;
        unsigned                            segment_counter;
        struct lfs_segment_usage            * seg_usage;
        struct lfs_inode_table_entry        * itab;
        void                                * data_raw;
        lfs_addr_t                          * iblocks_raw;
        int                                 iblocks;
        lfs_addr_t                          * iblk_2nd;

        struct lfs_segment_usage * seg_usage_act;
};
```

Figure 7.1: in-memory .ifile structure

Fig. 7.1.4 presents the in memory structure of the *.ifile* . There are several counters which are explained in the code commentary but the data part is worth more detailed description here.

When the *.ifile* is initiated in the `ifile_init()` function, its size is already known so all memory can be allocated. `ifile_new()` computes the required space for data and sets the `raw_data` item of the ifile structure to point to the allocated unstructured memory. Because we require access to different parts of the *.ifile* as a different structured memory, we remap the other pointers of the `struct ifile` to point to various offset within the allocated memory area. `remap_pointers()` sets the `ifile.seg_usage` to point to the beginning and `ifile.itab` just after the segment usage table. Extra space is allocated for indirect blocks (`iblocks_raw`) and similarly the `iblk_2nd` is set to point to the second-level indirect block within this indirect block array.

Ifile write out

Writing the *.ifile* to the disk is performed by two nested loops. The outer loop in the `ifile.c:write_to_disk()` iterates per segment. First, the new segment is zeroed and the *segment summary* is filled. Second, the number of blocks which fits to the current segment must be figured out. We cannot use the whole unused space in the segment, because of the *finfo* records in the end of the segment. Both data blocks and relevant finfos must fit into the same segment. `get_finfos_block()` returns base pointer of the finfo block. Finfo records are filled when each block is written. Because the finfo base pointer points to a `mmap()`ed area, writing via this pointer means writing directly to the device, so unnecessary memory copying is avoided.

`write_blocks()` is responsible for writing number of data or indirect blocks as well as filling the *finfo* records. The inner loop in the `write_blocks()` function writes a single block to the device per iteration. There are several different kinds of blocks and each of needs a different way of handling. The cases are as follows :

- *data block* - all data blocks are written out prior to the indirect blocks. This enables us to update indirect blocks with data block addresses at the moment when the address is known without any need of address precomputation. A data block address is saved by `save_dblock_addr()` which means that the address is written to the *.ifile* inode or to an indirect block. The location of the address record is determined by the block number and `save_dblock_addr()` performs that simple computation.
- *second level indirect block* - identical to the data blocks, its address is saved to a parent indirect block by `save_2nd_iblock_addr()`
- *first level indirect block and second level indirection parent block* - the address is saved directly to the *.ifile* inode `blocks` array (`LFS_INODE_INDIR_BLOCK(1)` resp. `LFS_INODE_INDIR_BLOCK(2)`)
- *single indirect block* - when an *.ifile* has only a single indirect block, we have to skip the second level in the `ifile.iblocks_raw`. That is the reason why there is a special adjustment code in the *else* branch inside the inner loop (see `write_blocks()` code).

Ifile inode

For the sake of simplicity, `write_inode()` places the *.ifile* inode into a new segment. It might look like a waste of space, but *.ifile* inode will be the most frequently updated inode so after a sync this inode will be written to a new location and the segment is going to be reclaimed by the *garbage collector* as already mentioned in Sec. 7.1.2. `write_inode()` fills the segment summary and writes the inode to the first data block of that segment. Since there are no file-data-blocks, no *finfos* are included. Address of the *.ifile* is saved in the *superblock*

7.1.5 Non-data segments

The non-data segments are written last. This includes superblocks and empty segment summaries of unused segments.

Superblocks are written in the `write_sb()` function. First the in-memory temporary superblock (CPU endianness template) is converted to little endianess (LFS native) and then written to the disk. Because the disk might be very large, we prefere seeking over memory mapping. Since the device file is large, usually more than 4GiB, we have to use `lseek64()` which can deal with large files.

7.2 Garbage collector - user space part

7.2.1 Overview

The main function of the user space GC (we will call it **UrSp_GC** from now on) is a message loop. It waits for any messages from the kernel, parses them, and processes

the information. If need be, it sends requests to the kernel part.

The UrSp_GC maintains its data basically in a linklist of segments, with an index table for faster access. The linklist is always up-to-date with all the information received from kernel. It includes attributes like live blocks in next 10 segments, number of consecutive segments on either side of the segment in question, etc.

There are two types of cleaning requests. First is emergency request, it happens when system sources are very low as is disk space. For this request, the UrSp_GC mantains a special table, and it quickly picks the best candidate to clean. The second type of request is issued when the number of free segments decreases to certain value. The UrSp_GC then executes a function over the linklist to find the best candidates to collect.

Depending on the state of the system (used segments, number of received messages, etc.) the UrSp_GC can run the choosing procedure and send a clean request by itself.

7.2.2 Source code

The source code of UrSp_GC is located in the directory `/gc` (NOT `/src/gc`!). The whole code is thoroughly commented, so I will describe only the files, and what is included in them and advise you to go ahead and open the source code for further details.
Header files: `gc_const.h` - this file includes all the global constants for the UrSp_GC
The following header files are each declarations for their source (`.c`) files.
`gc.h` - this file includes declarations for the main gc.c file
`gc_er_table.h` - contains declaration for emergency table
`gc_idx_table.h` - declaration for index table pointing to linklist
`gc_linklist.h` - declarations for the linklist with segment information
`gc_netlink_comm.h` - declarations for netlink communication

Source files: `gc.c` - implementation of the main message loop and message processing
`gc_er_table.c` - source code for the emergency table
`gc_idx_table.c` - implementation of the index table (pointing to linklist)
`gc_linklist.c` - implementation of the linklist with segment information
`gc_netlink_comm.c` - netlink socket operations

7.2.3 Most important functions

`main` function. The main function incorporates the message loop (using the netlink socket operations from `gc_netlink_comm.c`). Calls `parse` and `do_task` when a message from kernel is received.
`parse` function. This function parses the data from the socket in to UrSp_GC's variables and data structures.
`do_task` function. It's basically a switch over different message types from kernel. It uses data from `parse` to update the persistent data structures (*linklist*, *idx_table*, ...). It calls functions from `gc_er_table.c`, `gc_idx_table.c` and `gc_linklist.c` or `choose_segs_to_clean` from `main.c`, which finds the most suitable candidates for collection.

There are functions for the data-modifying messages (`SET`, `SET_PARTIAL`, `UPDATE`, identUNSET, identREPAIR) in component files (`gc_er_table.c`, `gc_idx_table.c` and `gc_linklist.c`), that change underlying structures accordingly.

7.2.4 Decision process

There are two cases, where the UrSp_GC makes a decision. It is **WHETHER** to clean and **WHAT** to clean. Both are heavily parametrized, so we will try to give you the big picture here. For further information refer to `gc.c`.

We would also like to mention, that this part is definitely worth deep research and the functions for these operations tended to get more complex over the course of the project and they would most definitely get much more complex in a real product.

We base the process of choosing whether to clean on these information:
First, we base this on time and number of received messages. We will only further consider cleaning if at least `GC_RECALC_TIME` seconds passed since last recalculation/-cleaning OR if there was at least `GC_NEW_SEGS_FROM_LAST_RECALC` created OR if we recieved more than `GC_CHANGES_NEEDED` changing messages from last collection.
If one of these three time/message dependant constraints is fullfilled we look into the issue of used space. There are to values of interest here. If there is less than $max(\frac{SC}{10}, MIN_{seg} - 2)$ segment, we don't clean anything. If there is more than that, but less than $max(MIN_{seg} - 2, min(\frac{SC}{5}, 50))$ segments, we clean only lonesome segments (with empty segments on both sides from them). And finally if there is more segments than the second constant, we do the full choosing algorithm. Where MIN_{seg} is the smallest possible number of usable segments on a volume and SC is the actual number of segments of this volume. min and max are standard min, max functions returning the smaller/bigger of their operands. We use real numbers instead of further constants to make the function readable.

The segment choosing algorithm is obviously most based on live data (LD) in each segment and on the age of that segment. But it is also based on the information about the segments around the evaluated one. The algorithm checks the sum of life blocks in next few semgents (this is also important for choosing blocks of segments to be cleaned instead of "best" individuals). Then it takes in account free segments in the previous one hundred and next one hundred of segments. Yet another modifier is the number of consecutive segments in front and behind this one. It's obvious, that it is better to collect something at the edge rather than in the middle of a bunch of segments.
Schematic equation for evaluation:

$$eval(seg) = LD * age * LD_w_neighb * (100 - free_{next}) * (100 - free_{prev}) * cons_{prev} * cons_{next} \tag{7.1}$$

7.2.5 Logging

The UrSp_GC logs the important messages in the *syslog*, where they can be viewed. The lines with logs are preceded by `lfs_gc`. It logs on 2 different levels `LOG_INFO` and `LOG_ERR`. The messages of the second type are also printed on the error output.

Appendix A

Manual pages

```
MKFS.LFS(8)                                                    MKFS.LFS(8)



NAME
       mkfs.lfs - create a lfs filesystem

SYNOPSIS
       mkfs.lfs [ -L new-volume-name ] device

DESCRIPTION
       mkfs.lfs  is  used to create a lfs filesystem (usually in a disk parti-
       tion).  device is the special file corresponding  to  the  device  (e.g
       /dev/hdXX).

       -L new-volume-label
              Set the volume label for the filesystem to new-volume-label.

AUTHOR
       This  version  of  mkfs.lfs has been written by Tomas Hruby <byjac@mat-
       fyz.cz>.

AVAILABILITY
       mkfs.lfs  is  part  of  the  LFS  project   and   is   available   from
       http://nenya.ms.mff.cuni.cz/~holub/lfs/

SEE ALSO
       dump.lfs(8), fsck.lfs(8),



mkfs.lfs                        June 2006                      MKFS.LFS(8)
```

NAME

dump.lfs - dump lfs filesystem information

SYNOPSIS

dump.lfs device [all | info | sb | segs | ifile | inode=inode_number | seg=segment_number | dir=dir_ino]

DESCRIPTION

dump.lfs prints the super blocks, .ifile content and segment summaries for filesystem present on device

OPTIONS

all show all available informations

info show summary of filesystem

sb show super block of filesystem

segs show segments of filesystem

ifile show ifile content

inode=inode_number
show inode selected by inode number

seg=segment_number
show segement content of segment selected by segment number

dir=dir_ino
show directory structure from directory selected by directory inode number

AUTHOR

dump.lfs was written by Jan Taus <pan_tau@matfyz.cz> and Tomas Hruby <byjac@matfyz.cz>

AVAILABILITY

mkfs.lfs is part of the LFS project and is available from http://nenya.ms.mff.cuni.cz/~holub/lfs/

SEE ALSO

fsck.lfs(8), mkfs.lfs(8)

Appendix B

The GNU General Public License

Version 2, June 1991

Copyright © 1989, 1991 Free Software Foundation, Inc. 51 Franklin St, Fifth Floor, Boston, MA 02110-1301, USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

Terms and Conditions For Copying, Distribution and Modification

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   (a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   (b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   (c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

   Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   (a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   (b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   (c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

   The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

   If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the

conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

No Warranty

11. Because the program is licensed free of charge, there is no warranty for the program, to the extent permitted by applicable law. Except when otherwise stated in writing the copyright holders and/or other parties provide the program "as is" without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of the program is with you. Should the program prove defective, you assume the cost of all necessary servicing, repair or correction.

12. In no event unless required by applicable law or agreed to in writing will any copyright holder, or any other party who may modify and/or redistribute the program as permitted above, be liable to

YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

List of Figures

4.1 Block indices 16
4.2 Disk structure overview 17
4.3 Ifile structure 18
4.4 Inode versions 20
4.5 Partial segments 20
4.6 Segment chains 21
4.7 Partial segment structure 22
4.8 Segment summary 23
4.9 Finfo structure definition 23
4.10 Finfos and corresponding blocks 24

5.1 Segment building 30
5.2 Plan item 32
5.3 Page write call graph 33
5.4 Inode planning call graph 35
5.5 Sync segment plan definition 39
5.6 Sync segment plans 40
5.7 Sync segment plan definition 40
5.8 Sync segment plan definition 41
5.9 Truncating indirect blocks 43
5.10 Keeping segment chain after sync 45
5.11 Directory entry structure 49
5.12 Index block entry structure 50
5.13 Index block structure 50
5.14 Two level index tree 51
5.15 Hash collision chain across multiple index blocks 52
5.16 Splitting dentry 54
5.17 Directory leaf block defragmentation 55
5.18 Single block directory split 55
5.19 Inserting a new index into an index block 56
5.20 Inserting a new dentry and splitting of index root 57
5.21 Journal line on disk representation 59
5.22 Readdir private info structure 61
5.23 Item of a leaf block index list 61

7.1 in-memory .ifile structure 71

List of Tables

3.1 Information exported via `sysfs` 13

4.1 Ifile access functions 19

5.1 Header files 25
5.2 Plan manipulation functions 32
5.3 BIO issuing functions 37
5.4 Relationship between a segment category, flags and live data counters in a segment summary. 44
5.5 Info message priorities 48

Bibliography

[1] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. In *SOSP '91: Proceedings of the thirteenth ACM symposium on Operating systems principles*, pages 1–15, New York, NY, USA, 1991. ACM Press.

[2] Margo I. Seltzer, Keith Bostic, Marshall K. McKusick, and Carl Staelin. An implementation of a log-structured file system for UNIX. In *USENIX Winter*, pages 307–326, 1993.

[3] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating systems (2nd ed.): design and implementation.* Prentice-Hall, Inc., 1997.

[4] Jeanna Neefe Matthews, Drew Roselli, Adam M. Costello, Randolph Y. Wang, and Thomas E. Anderson. Improving the performance of log-structured file systems with adaptive methods. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 238–251, New York, NY, USA, 1997. ACM Press.

[5] Robert Love. *Linux Kernel Development.* Novell Press, 2005.

[6] Jens Axboe, Suparna Bhattacharya, and Nick Piggin. Notes on the generic block layer rewrite in linux 2.5. Linux kernel Documentation (block/biodoc.txt), May 2002.

[7] Daniel R. Phillips. A directory index for ext2. In *Proceedings of the Ottawa Linux Symposium*, pages 425–439, 2002.