Technical Report

LIMEFS Realtime File System

Rink Springer

Revision 1.1
2nd June 2005

Preface

This report is about the technical aspects of the filesystem created during my internship at Philips Research in Eindhoven. It is written in an informative manner, which should provide the reader with a clear overview of the design and implementation of the filesystem.

The target audience is mostly software engineers having an interest in the LIMEFS design/implementation. The Linux VFS implementation is also briefly outlined, more details can be found in [4]. This report assumes generic filesystem knowledge.

I'd like to to thank the following people who helped me during my internship: Bas van Tiel, Michiel van Slobbe, Ron Eringa, Ron Lievens, Ad Denissen, Mischa Geldermans, Christ Vriens, Werner Almesberger, Wim van de Goor and Wim Zijlmans.

Rink Springer
Eindhoven, May 2005

Contents

List of Figures

Summary

This report provides an in-depth view of the technical aspects of the LIMEFS filesystem. It provides an extensive overview of the LIMEFS filesystem as well as a brief overview of the Linux Virtual File System design.

The following aspects are covered by this report:

- Linux I/O Layer
  What is the basic design, what does a filesystem have to provide? Focuses mostly on the Linux Virtual File System layer.
- LIMEFS
  Overall design ideas of the LIMEFS filesystem, as well as all structures used within the LIMEFS filesystem.
- Implementation
  Linux Implementation-specific details of the filesystem.

The overall conclusion of the report is a stable filesystem which is very suitable for storing large files of a few gigabytes. There is room for improvement, especially within the administration part of the filesystem. As for most multimedia applications, the filesystem is ready to be used.

Definitions and abbreviations

ABISS	Active Block I/O Scheduling System, a scheduling system which can provide guaranteed read- and write I/O performance to applications [1]
dentry	Directory entry, data structure (include/linux/dcache.h) in the directory cache which contains information about a specific inode within a directory
extent	An (offset, length) pair
hard link	A directory entry that directly references an inode. If there are multiple hard links to a single inode and if one of the links is deleted, the remaining links still reference the inode. [6]
mmap(2), unlink(2)	These are references to C library functions. The number between parentheses defines the section number in the appropriate manual page
inode	A data structure used by the filesystem to describe a file. The contents of an inode include the file's type and size, the UID of the file's owner, the GID of the directory in which it was created, and a list of the disk blocks and and fragments that make up the file. [6]
LIMEFS	Large-file metadata-In-Memory Extend-based File System, the filesystem created during the internship
soft link	A file whose contents are interpreted as a path name when it is supplied as a component of a path name. Also called a soft link. [6]
symbolic link	See soft link

Chapter 1

Introduction

This technical report describes the LIMEFS filesystem. This filesystem was written during my graduation project of my Computer Sciences study at Fontys Hogescholen in Eindhoven and carried out at Philips Research in Eindhoven. The goal was design and implement a framework for realtime file storage, which came down to implementing a realtime filesystem which can provide guaranteed I/O performance.

The report will provide a rough description of the Linux Virtual File System structure. The next part will focus completely on the design and implementation of the XXFS filesystem, followed by concluding statements on the project.

Chapter 2

Linux I/O Layer

2.1 Introduction

This chapter will describe the Linux I/O layer. In order to develop a filesystem, you should understand how a Linux system interacts with files, devices and filesystems. This chapter will shed light on Linux I/O internals.

It is surprising to see how close Linux tries to follow the original UNIX architecture, as outlined in [2]. This book is a good reference to understand the algorithms and major design ideas (for example, of the cache code), but not the actual implementation.

Only the actual filesystem layer itself will be described in detail, because it is the most important for this project. For a complete overview of the I/O path, refer to Job Wildschut's technical graduation report: Linux IO Performance, which sheds light on the internals. [7]

2.2 Overview

Linux I/O can be summarized in the following image:

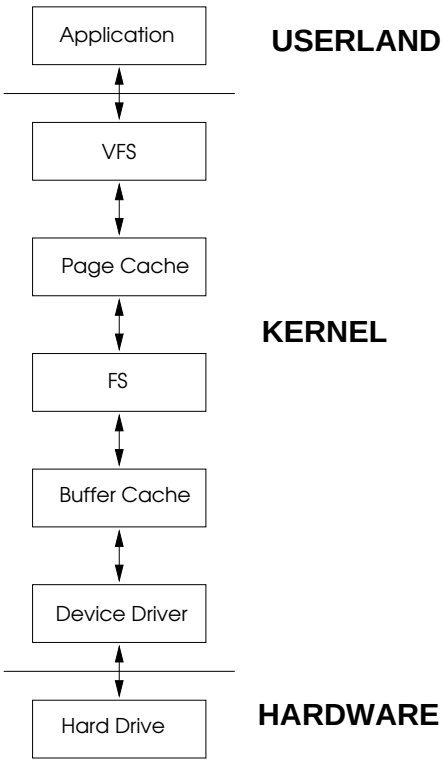


Figure 2.1: Summary of Linux I/O layers

Whenever a file is accessed, the request is passed down to the Virtual Filesystem. The VFS is actually a generic, filesystem independent interface which translates system calls, like open(2), read(2) etc to specific filesystem functions.

The page cache provides an uniform interface to file reading/writing, which handles functions like mmap(2). It provides all functionality in generic functions, relying only on the filesystem to provide buffers (using the buffer cache) for the actual data on disk. All other functionality is implemented by the filesystem code itself, described below.

The filesystem will receive VFS and page-cache requests and resolve them to I/O requests. These I/O requests are handled by the buffer cache, which will read blocks from a device and place them in a cache buffer. Modified buffers are written back to the disk as needed. The filesystem must conform to an interface as defined in the VFS layer. This layer is described in the VFS layer section later on.

The device driver is the device-dependent layer, which will handle the Linux I/O request and translate it into a device request.

2.3 VFS layer

This section will describe the VFS layer as well as the connection between the VFS and the page cache.

Before a filesystem is known by the kernel, it must be registered. This must be done using the `register_filesystem()` call, which takes a `struct file_system_type` argument.

The most important member of this struct is `get_sb` (get superblock), which points to a function which Linux will call whenever it tries to mount a filesystem of this type. This function must setup the `struct super_block` which is passed along with it.

2.3.1 Superblock operations

```
static struct super_operations lime_sops = {
 .alloc_inode   = lime_alloc_inode,
 .destroy_inode = lime_destroy_inode,
 .read_inode    = lime_read_inode,
 .write_inode   = lime_write_inode,
 .put_super     = lime_put_super,
 .write_super   = lime_write_super,
 .statfs        = lime_statfs,
};
```

The superblock operations structure contains the more low-level functions for a filesystem. These are basic operations such as inode management.

Dealing with files/directories will be done using the directory operations of the root inode (`sb->s_root` must always be set to the root inode), which will be covered in section 2.3.2.

2.3.1.1 alloc_inode

Prototype	`struct inode* alloc_inode (struct super_block* sb)`

Used to allocate an inode. The function should call the generic function `new_inode` to allocate an inode and initialize it with filesystem-specific data.

2.3.1.2 destroy_inode

Prototype	`void destroy_inode (struct inode* inode)`

The opposite of `alloc_node`, this function must clean up the extra inode information. The inode itself must *not* be destroyed, the VFS will take care of this.

2.3.1.3 read_inode

Prototype	`void read_inode (struct inode* inode)`

Called whenever an inode structure must be filled, the inode to read is specified in `inode->i_ino`. Of all inode structure members, a few deserve special attention:

`inode->i_fop` is a pointer to an operations structure, which contains pointers to functions used for actual file/directory manipulation. These file operations are outlined in section 2.3.4, the directory operations can be found in 2.3.2.

`inode->i_op` is a pointer to an inode operations structure, which contains pointers to functions only needed for directory operations. These functions can be found in section 2.3.3.

Should this operation fail, `make_bad_inode(inode)` must be used in order to invalidate the inode.

2.3.1.4 write_inode

Prototype	`void write_inode (struct inode* inode, int synchronous)`

This will write inode data to the disk. The parameter indicates whether the data must be written synchronous, but is not implemented by most filesystems.

2.3.1.5 put_super

Prototype	`void put_super (struct super_block* sb)`

This will be called when the VFS unmounts the filesystem. Normally, a filesystem should update the on-disk superblock when the filesystem is dirty.

2.3.1.6 write_super

Prototype	`void write_super (struct super_block* sb)`

Called whenever the superblock needs to be written to the disk. This is for instance used while performing a sync(1) or fsync(2).

2.3.1.7 statfs

Prototype	`int statfs (struct super_block* sb, struct kstatfs* buf)`

Used to query used/free space/inode information. `struct kstatfs` can be found in `include/linux/statfs.h`, which must be filled by this function.

2.3.2 Directory operations

```
struct file_operations lime_dir_ops = {
 .read    = generic_read_dir,
 .readdir = lime_readdir,
 .fsync   = file_fsync,
};
```

The functions outlined here are only used for directory inodes. Only `readdir` needs to be implemented (for it is very filesystem specific), the other functions call generic VFS functions which are suitable for most filesystems.

2.3.2.1 readdir

Prototype	`int readdir (struct file* filp, void* dirent, filldir_t filldir)`

Called to retrieve files in a directory (directory inode is `filp->f_dentry->d_inode`), starting from offset `filp->f_pos`. This offset must be updated after new entries are passed, so the VFS knows which offset it must query to obtain the next directory entry.

`filldir` is the function to be called for each result, a prototype can be found in `include/linux/fs.h`.

2.3.3 Inode operations

```
struct inode_operations lime_dir_inode_ops = {
 .create    = lime_create,
 .lookup    = lime_lookup,
 .link      = lime_link,
 .unlink    = lime_unlink,
 .symlink   = lime_symlink,
 .mkdir     = lime_mkdir,
 .rmdir     = lime_rmdir,
 .rename    = lime_rename,
};
```

Misleading as the name may be, inode operations are mainly used in a directory context. They are mainly invoked for creating new inodes and looking up inode information. File inodes use so-called file operations which are outlined in section 2.3.4.

2.3.3.1 create

Prototype	`int create (struct inode* dir, struct dentry* dentry, int mode, struct nameidata* nd)`

Creates a new file in `dir`, with name `dentry->d_name.name` and mode `mode`. On success, this function must call `d_instantiate()` to add the created dentry to the cache.

2.3.3.2 lookup

Prototype	`struct dentry* lookup (struct inode* dir, struct dentry* dentry, struct nameidata* nd)`

Called to retrieve the inode number from a filename (`dentry->d_name.name`) in a directory (directory inode is in `dir->d_ino`). On success, the inode found should be read (using `iget`) and added to the inode cache (using `d_add`).

2.3.3.3 link

Prototype	`int link (struct dentry* old_dentry, struct inode* dir, struct dentry* dentry)`

Create a hard link from `old_dentry` to `dentry` in directory `dir`. Inode information from `old_dentry` can be copied over to the new inode as needed.

2.3.3.4 unlink

Prototype	`int unlink (struct inode* dir, struct dentry* dentry)`

Removes an inode by name (`dentry->d_name.name`) in directory `dir`.

2.3.3.5 symlink

Prototype	`int symlink (struct inode* dir, struct dentry* dentry, const char* symname)`

Create a soft link by name (`dentry->d_name.name`) in directory `dir`. The soft link should point to `symname`.

2.3.3.6 mkdir

Prototype	`int mkdir (struct inode* dir, struct dentry* dentry, int mode)`

Create a new directory in parent directory `dir`. The directory name is passed in `dentry->d_name.name` and the mode in `mode`.

2.3.3.7 rmdir

Prototype	`int rmdir (struct inode* dir, struct dentry* dentry)`

Removes directory named `dentry->d_name.name` from parent directory `dir`. This function must check whether the supplied directory is empty, and return `-ENOTEMPTY` if not.

2.3.3.8 rename

Prototype	`int rename (struct inode* old_dir, struct dentry* old_dentry, struct inode* new_dir, struct dentry* new_dentry)`

This will rename entry `old_dentry->d_name.name` in directory `old_dir` to entry `new_dentry->d_name.name` in directory `new_dir`. This function must ensure `new_dentry->d_name` doesn't already exist before continuing.

2.3.4 File operations

```
struct file_operations lime_file_ops = {
 .llseek     = generic_file_llseek,
 .read       = generic_file_read,
 .write      = generic_file_write,
 .mmap       = generic_file_mmap,
 .fsync      = file_fsync,
 .sendfile   = generic_file_sendfile
};
```

These functions usually refer to generic VFS functions. All disk-based filesystems use the page cache for data reading/writing. The `generic_file_read` and `generic_file_write` functions use the inode's `i_mapping->a_ops` to refer to the address space operations. These are documented below:

```
struct address_space_operations lime_aops = {
 .readpage      = lime_readpage,
 .writepage     = lime_writepage,
 .prepare_write = lime_prepare_write,
 .commit_write  = generic_commit_write
};
```

The functions listed here use generic VFS calls, which ultimately only need a single function from the filesystem: the `get_block` function, which is fully explained in [2]. A more Linux-specific explanation is given in the next paragraph.

2.3.4.1 get_block

Prototype	`int get_block (struct inode* inode, sector_t iblock, struct buffer_head* bh_result, int create)`

This function is passed to generic calls (these are `block_read_full_page()` and `block_write_full_page()`), which expects it to return a mapped buffer to a requested block within a file (the size of a block must be set on initialization time, using `sb_set_blocksize`).

The filesystem should allocate a new block if the the `create` flag is non-zero.

2.4 ABISS

ABISS, or the Active Block I/O Scheduling System (as outlined in [1]) is a system designed to provide guaranteed read- and write performance to applications. The application issues a request to ABISS, in which a request is made for certain file bandwidth guarantees. ABISS will handle this by ordering writes and providing read-aheads as needed.

From a filesystem's point of view, some functions can be provided to ABISS, using the following structure:

```
struct abiss_fs_ops lime_fs_ops = {
 .get_block              = lime_do_get_block,
 .readpage               = lime_readpage,
 .alloc_unit             = lime_alloc_unit,
};
```

The first function, `get_block`, must be an entry point to the get block function as outlined in section 2.3.4.1. It is used to cache the logical block numbers, so no disk accesses are needed to look up blocks.

The second function, `readpage`, is the entry point to the file system's readpage function as outlined in section 2.3.4. This is used for the readahead functionality.

Finally, `alloc_unit` must return the basic block allocation unit of the filesystem, in bytes.

Chapter 3

LIMEFS

3.1 Introduction

The LIMEFS filesystem contains a few distinctive characteristics:

- In-memory meta data
  All meta data used is in memory and stored consecutively on disk. This ensures fast updating, with few I/O operations. Most administration is built on-the-fly during mount time.
- Few files/inodes
  This filesystem is designed to store a few very large files. This makes a larger-than-normal data block size desirable.
- Extent structure
  Extents[1] are used to maintain data block administration.
- Disk block size
  The filesystem can use an user-defined block size, which is used for all I/O requests. This allows for good performance.
- Inodes belong to directories
  A directory is not a list of inodes residing in it. Each inode knows to which directory it belongs. Since all meta data is stored in-memory, this has barely any performance constraints.

The LIMEFS filesystem is designed to provide predictable performance.

Realtime I/O scheduling is not part of the filesystem. In order to have guaranteed disk I/O performance, the ABISS framework is used. This framework

[1] An extent is an (offset, length) tuple

can provide realtime I/O performance based on the application's wishes. ABISS itself is a separate project, more information can be found at [1].

3.2 Overview

If we view a disk as an one-dimensional array of sectors, the next figure displays the on-disk structure:

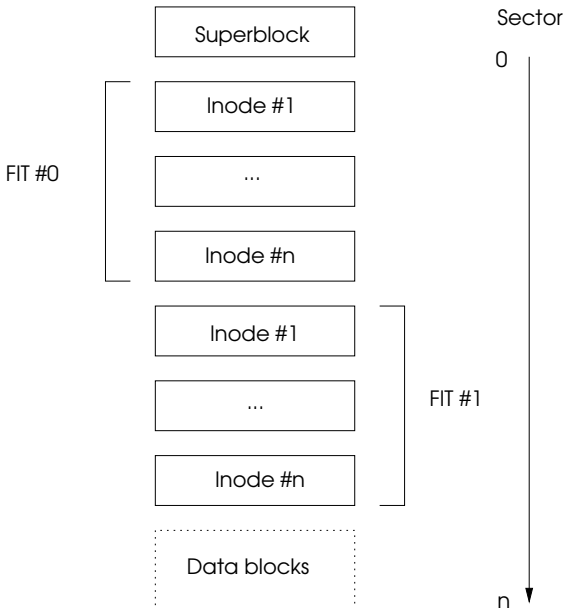


Figure 3.1: On-disk structure of the filesystem

- Superblock
  This is the very first filesystem structure. It contains read-only information needed to mount the filesystem.

- File Information Table
  The File Information Table, FIT, contains the inodes.

- Data blocks
  Space where the file data resides.

These structures will be explained in the next paragraphs.

3.3 Superblock

The superblock contains a magic value used to identify the filesystem, as well as information used for mounting. The superblock is *readonly*, and all values in it are copied to memory. This copy is used as long as the filesystem is mounted.

```
struct lime_superblock {
 uint32_t sb_magic;
 uint32_t sb_version;
 uint32_t sb_diskBlockSize;
 uint32_t sb_dataBlockSize;
 uint64_t sb_numDiskBlocks;
 uint32_t sb_inodeSize;
 uint32_t sb_numInodes;
};
```

sb_magic	Magic value used for identification `LIME_SB_MAGIC (0xFC492B42)`
sb_version	Current version number, `LIME_SB_VERSION (0x0001)`
sb_diskBlockSize	Size of a disk block, in bytes. Due to Linux constrains, this may not exceed a page size (4KB on x86)
sb_dataBlockSize	Size of a data block, in bytes. This must be a multiple of sb_diskBlockSize.
sb_numDiskBlocks	Total number of blocks on the disk (in sb_diskBlockSize byte blocks)
sb_inodeSize	Size of a single inode structure. Should be `1024` for version 1
sb_numInodes	Total number of inodes on the filesystem, including reserved[2] ones

The next page will show an overview of the calculations with their corresponding place on the disk.

[2]reserved inodes are the root and checkpoint inodes

Based on these values, calculations are made for often-used values:

$$fit\_in\_diskblocks = \frac{sbInodeSize * sbNumInodes + sbDiskBlockSize - 1}{sbDiskBlockSize}$$

$$firstDataBlock = \frac{(1 + NUM\_FITS * fit\_in\_diskblocks) + \frac{sbDataBlockSize}{sbDiskBlockSize} - 1}{\frac{sbDataBlockSize}{sbDiskBlockSize}}$$

$$numDataBlocks = \frac{sb\_numDiskBlocks}{\frac{sbDataBlockSize}{sbDiskBlockSize}} - firstDataBlock$$

$$fitOffset(n) = 1 + (n * fit\_in\_diskblocks)$$

Graphically, this gives:

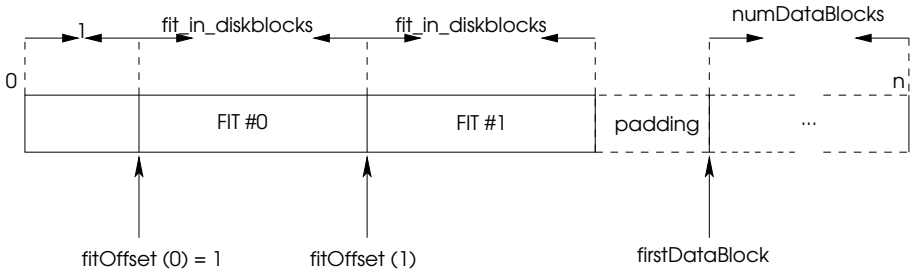


Figure 3.2: Internal offsets of on-disk structures

3.4 File Information Table

A File Information Table is an array of all inodes on the disk, and therefore spans numInodes entries (as defined in the superblock). The FIT is always stored two times; upon mount time, the filesystem reads the checkpoint inode to find the most up-to-date copy.

Notice Inode #0 does *not* exist within our filesystem. This is due to Linux having reserved this number (for example, it won't show up in directory lists). In order to work around this, we number our first inode #1, which is always the root directory inode.

```
#define LIME_MAX_NAME_LEN 256

struct lime_inode {
 uint32_t i_tag;                    /* type tag */
 uint32_t i_parentdir;              /* parent directory inode */
 char     i_name[LIME_MAX_NAME_LEN]; /* inode name */
 struct   lime_inode_data  i_idata;  /* posix data */
```

```
 union {
  struct  lime_file_inode  fi;
  struct  lime_dir_inode   di;
  struct  lime_hlink_inode hi;
  struct  lime_slink_inode si;
  struct  lime_checkpoint  ci;
  char    pad[739];
 } i_un;
};
```

In order to store POSIX attributes, extra information must be stored on a per-inode basis. This information is represented by its own structure, and described below:

```
struct lime_inode_data {
 uint16_t id_mode;     /* inode mode for protection */
 uint16_t id_uid;      /* user ID */
 uint16_t id_gid;      /* group ID */
 uint32_t id_atime;    /* last access time */
 uint32_t id_mtime;    /* last modification time */
 uint32_t id_ctime;    /* last status change time */
};
```

3.4.1 Directory Inode

This inode describes a directory, which has no special characteristics. This is because the `i_parentdir` field of each inode describes in which directory the inode resides.

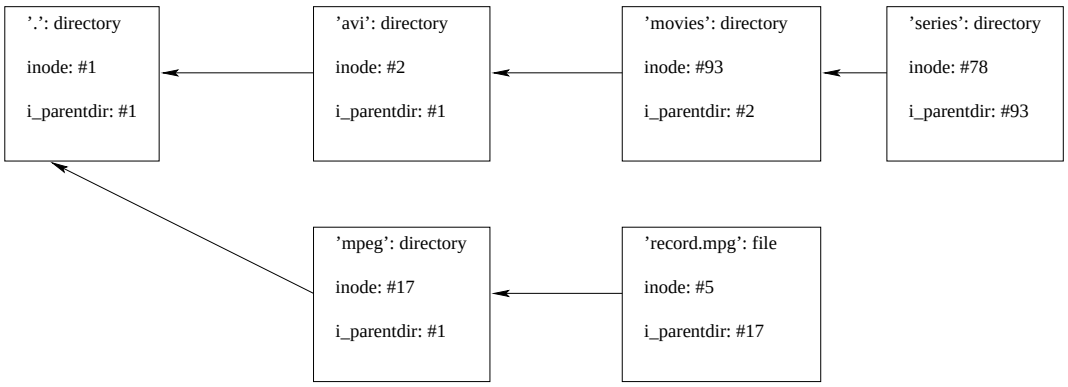


Figure 3.3: Nesting of directories

The tree for the figure is:

```
/
/avi
/avi/movies
/avi/movies/series
/mpeg
/mpeg/record.mpg
```

3.4.2 File Inode

An ordinary file, which contains data. The actual data blocks associated with the file are stored in extent structures.

```
#define LIME_EXTENTS_PER_INODE 80

struct lime_file_inode {
 uint32_t fi_boff;
 uint32_t fi_eoff;
 uint32_t fi_numextents;
 struct   lime_extent fi_extents[LIME_EXTENTS_PER_INODE];
};
```

fi_boff	Begin offset within the first extent
fi_eoff	End offset within the last extent
fi_numextents	Number of extents used
fi_extents	Structure used to store the extents

As stated in the definitions on page xi, an extent is simply a (offset, length) tuple. It contains the block offset, and the number of blocks used from this offset onwards.

Fragmentation study

As can be seen, we allow a maximum of 80 extents per file. If more extents are needed than this maximum, the file cannot be expanded beyond the current size.

In order to determine how much of an issue this is, Ad Denissen and I created a small tool which simulates an extent-based 250GB filesystem with 4MB data blocks. It simulates writing 10.000 files sized between 500 and 5000MB. If a file does not fit, it will delete an existing one at random to take fragmentation into account. Finally, we refuse to allocate the last 5% of the data blocks in order to decrease fragmentation[3].

Plotting an overview of the number of fragments per file and the average fragmentation gives an interesting result:

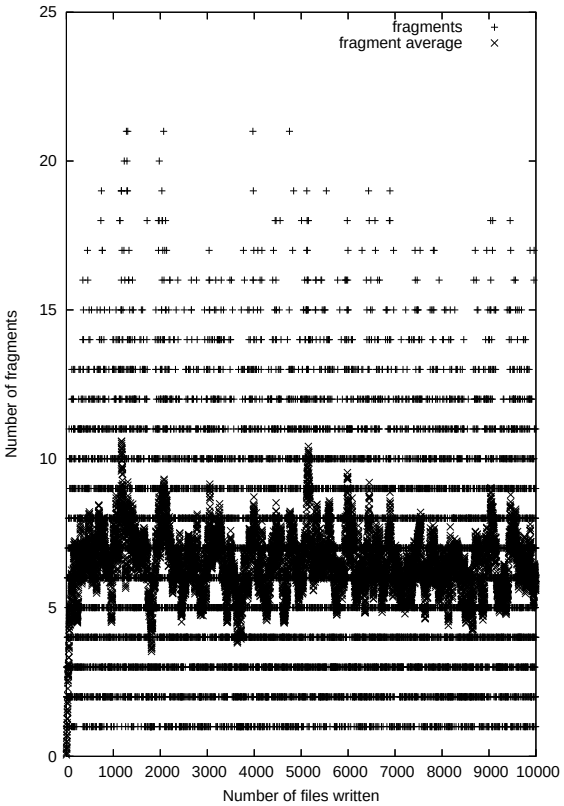


Figure 3.4: Fragments per file

[3]This value is based on some tests to determine an acceptable value

As can be seen, the number of fragments never exceeds 21. This indicates the 80 extents we have is acceptable. However, it is also interesting to see the relationship between fragment sizes and the occurrence between them.

In order to display this, we plot the fragment size as X and the occurrence in % as Y. This gives the following result:

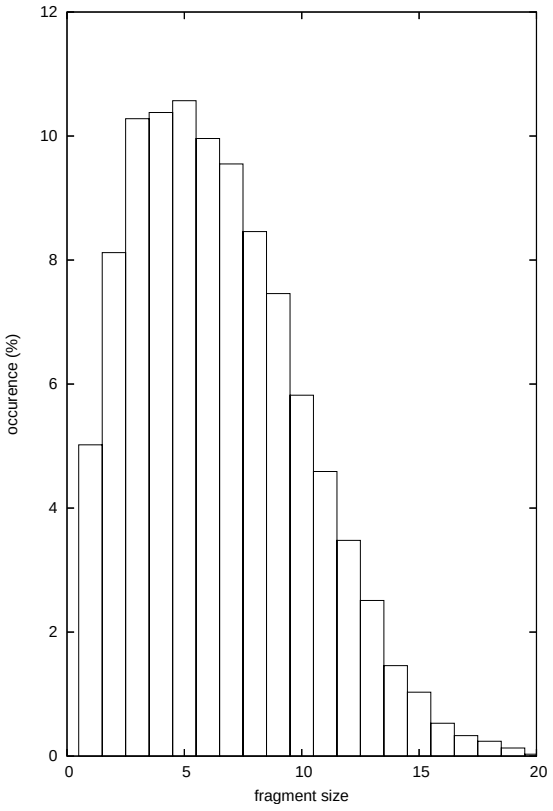


Figure 3.5: Fragment occurrence

As can be seen in these images, huge fragments are quite rare, even in everyday usage.

The extent structure is defined as:

```
struct lime_extent {
 uint32_t ex_offset;
 uint32_t ex_length;
};
```

Extents always point to the data blocks and will be expanded if more space is needed (refer to section 4.1 for more information). There is a hard limit

on how many extents a file can have (`LIME_EXTENTS_PER_INODE`), therefore a file can always consist of a minimum of
$LIME\_EXTENDS\_PER\_INODE * sbBlockSize$ bytes.

As can be expected, the filesystem will always attempt to expand the last extent. More information about the block allocation can be found in section 4.1 on page 25. The figure below shows an example of extent administration:

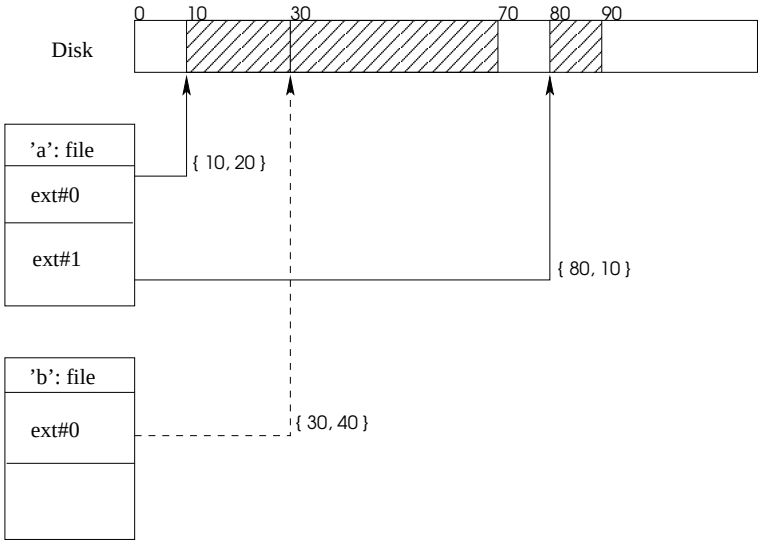


Figure 3.6: File-extent administration

As stated above, `boff` and `eoff` are offsets within the first and last block. Refer to the image below:

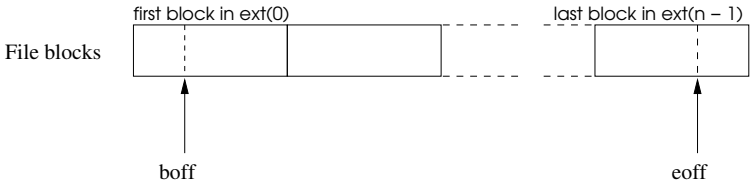


Figure 3.7: File begin- and end offsets

There are some implementation specific limitations on this approach, please refer to section 4.5 for more information.

Based on these two offsets, the file size can be calculated in bytes as:

$$filesize = dataBlockSize \cdot \left(\sum_{i=0}^{n} ext(i)_{length} \right) - boff - (dataBlockSize - eoff)$$

3.4.3 Hardlink Inode

A hard link is a reference to an inode, using a different name. Unlike traditional UNIX filesystems like FFS and ext, this filesystem allocates an extra inode with the hard link's name, but returns the destination inode. Therefore, we can implement this by simply storing the destination inode number.

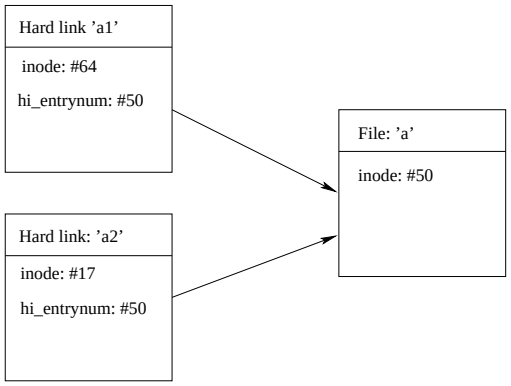


Figure 3.8: Hard link and file relationship

```
struct lime_hlink_inode {
 uint32_t hi_entrynum;
};
```

The implementation-specific notes can be found in section 4.4.

3.4.4 Softlink Inode

A soft link[4] is a reference to another location. It references by name, so therefore the destination doesn't have to be on this filesystem or even exist for that matter.

```
#define LIME_MAX_SYMLINK_LEN 256

struct lime_slink_inode {
 char si_path[LIME_MAX_SYMLINK_LEN];
};
```

The soft link implementation is simple: all that needs to be done, is to pass the `si_path` to the Linux VFS layer. This will resolve it to a new request to the accompanying filesystem.

[4] also known as a symbolic link

3.4.5 Checkpoint Inode

The checkpoint inode is used to determine the most recent copy of the FIT. It is always the final entry of the FIT; this increases the chance all entries are written before this final inode and thus are up-to-date.

```
struct lime_checkpoint {
 uint32_t ci_followup;
};
```

The followup count should only differ by 1, since they are used in a round-robin fashion. This means the FIT following the currently active FIT will be used. If the difference is not exactly one, the filesystem will not be mounted.

3.5 Data blocks

The data blocks start after the final File Allocation Table and are always aligned on a data block. They will continue until the end of the disk. The list of free blocks is constructed upon mount time, refer to section 4.2 for more information.

Chapter 4

Implementation

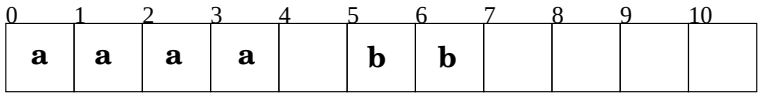
4.1 Allocation

As for any filesystem, fragmentation hurts performance and therefore is preferably kept at a minimal level. This is especially true for our filesystem, since we have a maximum number of extents which can be stored within an inode, as outlined in section 3.4.2.

In order to work around this, we use a different allocation strategy:

- Random first blocks
  For every first file block, we take a random block somewhere within the free space list of our disk.
- Extend previous blocks
  If we have a previously allocated block, we try to take the next block. If this fails, we take a random block.

Support we have the layout as below, *a* and *b* are files. The text below the image is the extent map as stored within the FIT:

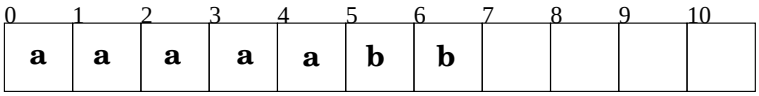


a = ex#0: { start = 0, len = 4 }
b = ex#0: {start = 5, len = 2 }

Figure 4.1: Allocation strategy: initial status

Now, let's assume we allocate an extra block for file *a*. Since there is a free block coming ahead, we can easily expand the extend. This gives the

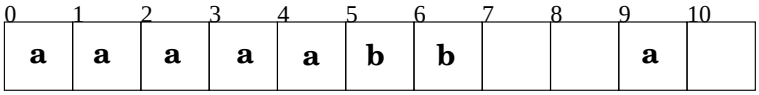
following:



a = ex#0: { start = 0, len = 5 }
b = ex#0: {start = 5, len = 2 }

Figure 4.2: Allocation strategy: after extending file 'a'

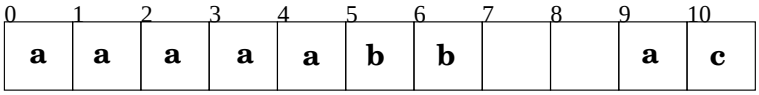
However, suppose file *a* wants another block. It may not overwrite the blocks already in use by file *b*, so we allocate a random block:



a = ex#0: { start = 0, len = 5 }, ex#1 { start = 9, len = 1 }
b = ex#0: {start = 5, len = 2 }

Figure 4.3: Allocation strategy: after extending file 'a' again

Finally, let's assume we create a new file, *c*, with a single block. This block will be randomly put somewhere on the disk:



a = ex#0: { start = 0, len = 5 }, ex#1 { start = 9, len = 1 }
b = ex#0: {start = 5, len = 3 }
c = ex#0: { start = 10, len = 1 }

Figure 4.4: Allocation strategy: new file 'c' with a data block

As can be seen, this hinders file *a* since it cannot extend to the next block and thus must create a new extent.

4.2 Free space

4.2.1 Introduction

Unlike most traditional filesystems, LIMEFS does not store an overview of free blocks on the disk. Rather, this list is constructed upon mount time and

consists of the following characteristics:

- Stored as an extent list
  In order to allow quick access, the freelist is stored as an extent-based list.
- Numerically sorted
  The list is always numerically sorted. This simplifies merging adjacent blocks.
- Limited number of entries
  The maximum list size[1] is $\frac{numDataBlocks}{2}$. Tests have shown this never happens, so we use a tenth of the maximum, or $\frac{numDataBlocks}{10}$.

These characteristics will be illustrated in the next paragraph.

4.2.2 Example

Below is an example of a free list. Shaded blocks lines are allocated, blank blocks are available:

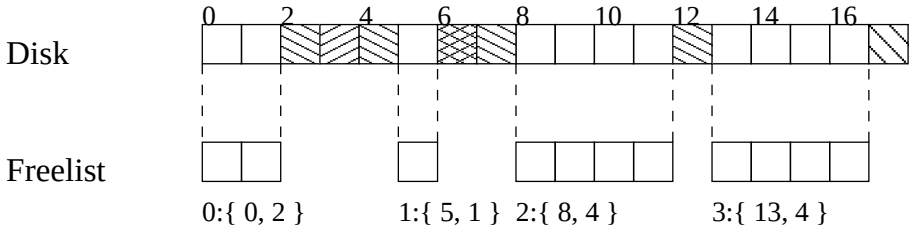


Figure 4.5: Freelist: initial status

Now, we free block 3. This means we need to add an extra entry to our freelist (since it is in the middle of used blocks. This gives the following:

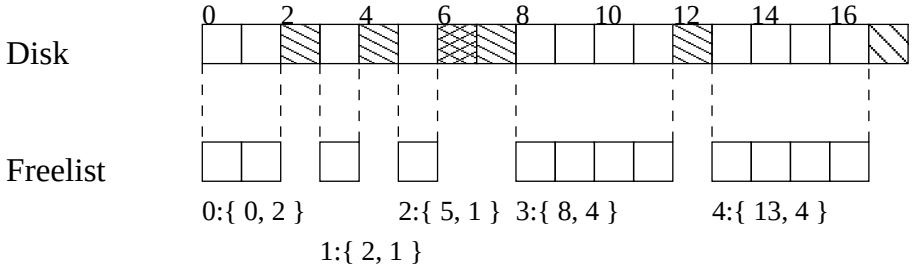


Figure 4.6: Freelist: freed block 3, gives new extent

[1] This will happen if even numbered blocks are used whereas odd numbered blocks are free

Block ranges can also be extended. For example, if we free block 6, we get:

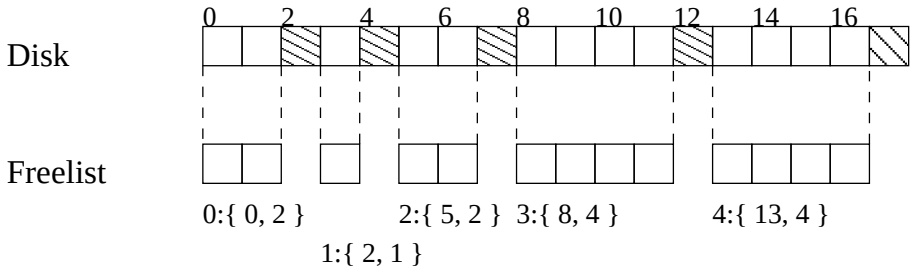


Figure 4.7: Freelist: freed block 6, expands extent

Finally, entire freelist extents can get merged. Suppose we free block 12, we get:

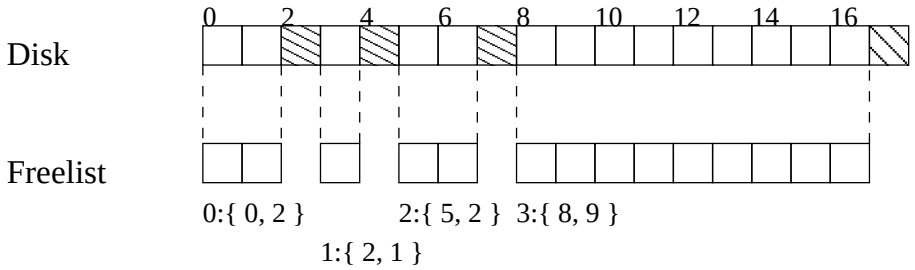


Figure 4.8: Freelist: freed block 12, merges extents

4.2.3 Operations

Only two operations need to be implemented by the freelist:

- Add a block to the freelist
  This is done by `lime_extent_mergefreeblocks`, which adds a previously-allocated block to the free list.
- Remove free block from the list
  Done by `lime_extent_removefreeblocks`, this will remove the block-number from the list.

These functions can be found in `fs/lime/freelist.c`.

4.3 Locking

From Linux 2.0 onward, there has been support for multiple processors. This used to be implemented using a single lock, but as it matured, finer-grained

locking is used to protect data structures.

Kernel-side locks in Linux

As outlined in [5], chapter 9, we could use the following locks:

- Atomic bitwise operations
  Linux 2.6 provides atomic bitwise set, clear and retrieve bit operations. These are useful for maintaining flags.
- Spinlocks
  A spin lock is an inexpensive operation, which will keep a CPU waiting (spinning) to see if a lock can be obtained. It will wait until it can. This is useful for data structures, but it may never be used if a context switch could occur.
- Reader/writer spinlocks
  This is the same as an ordinary spinlock, except that you lock for reading or writing. Multiple readers will be allowed, but there may be only one writer and never a writer if there are readers.
- Semaphores
  Semaphores are locks which can be held while blocking. This is useful for long sleeps (such as while passing data from or to userland).

Since you can not hold semaphores in an interrupt context and no spinlocks while doing a context switch, it is important to determine which lock to use. The codebase uses read/write spinlocks for the FIT and freelist[2], and atomic bitwise operations for the flags variable. The superblock does not have a lock because it is readonly, as outlined in section 3.3.

4.4 Hard links

As outlined in section 3.4.3, hardlinks are supported. However, they need special treatment on this filesystem, due to the fact that no reference counts are maintained. Since the FIT is maintained entirely within memory, this operation is relatively inexpensive.

Whenever a hardlink is created, it is like a normal inode, which contains the destination inode number. Inode reads will notice this is a hardlink, and refer to the hardlinked file.

[2]Writes are much more rare than reads, and it would be a waste of resources to wait to let readers spin if no one is writing

Deletion is different. A hardlink itself can always be deleted without any problems. However, whenever a file is deleted, it must be known whether any hard links refer to the file. If this is the case, the hardlink's filename is simply copied over the original file and the hardlink inode is removed instead of the original file to be deleted.

4.5 Begin and end offsets

As outlined in section 3.4.2, the filesystem supports begin and end offsets in order to allow truncation of a file. This truncation can occur from the begging of the file or from the end of the file, refer to section 3.4.2 for more information. Within the filesystem, the `boff` offset is simply added to the offset the user wants to read.

This has a severe limitation, namely that `boff` must be in disk blocks as only disk block-sized blocks can be read. The only way around this would be, to read 2 diskblocks and merge them. Since this cannot be done using traditional buffer cache functionality, which assumes one block is one buffer, a lot of existing code would have to be duplicated.

4.6 Commit daemon

A filesystem must always ensure reasonable integrity. Since all meta data is kept in-memory for this filesystem, a powerloss would mean the filesystem's FIT would be identical to the one on disk when we first mounted the filesystem. However, this would not be in sync with the file contents on disk.

In order to work around this, we periodically flush (commit) the FIT structure to the disk. Since we have 2 copies of the FIT (refer to section 3.4 for more information), a crash while writing the FIT means no data loss (except for data written after the last commit, of course).

This commit daemon is implemented as a kernel thread, since we need to hold separate locks. A bit in the flags register is used to mark the FIT as dirty, this bit is cleared after the new FIT is committed to disk. Upon unmounting of the filesystem, a special unmount flag is set and the commit daemon is signaled to wake up. It will do a final flush if the FIT is dirty, and exit.

The implementation of the daemon can be found in `fs/lime/fit.c`, in the function `lime_commitd`.

4.7 Tests

This is an Extreme Programming project, which means software tests are created beforehand in order to maintain a consistent level of quality. Since this is a kernel module, subsystems are hard to test using conventional tests.

To overcome this, the freelist implementation can be compiled in userland and kernelspace. Using standard testing libraries, the freelist is filled and the structures are compared to what they should be. This provides easy validation, since freelist corruption is much harder to debug when it occurs in kernelspace.

Finally, tests have been created to ensure proper operation of the filesystem by creating, deleting, writing and reading files, creating/removing directories, using links and such. Even though not all cases can possibly be tested, most tests try to provide the necessary complexity to increase the chance to trigger a failure.

All tests can be found in the `lime-linux/test` directory.

Chapter 5

Conclusion

This report provides a complete outline of the design and realization of the filesystem. The Linux Virtual File System has briefly been covered, to represent the reader an overview what a Linux filesystem needs to provide towards the kernel itself. More in-depth information can be found in [3] and [5].

The filesystem itself has also been described. This description covers the raw data structures as well as implementation-specific aspects of the current Linux implementation. This covers both implementation-specific details such as the free list, as well as limitations in the current implementation.

The following improvements could be made:

- Serializable FIT entries
  Currently, each inode has a fixed size. As we support up to 80 extents per inode, this means we usually waste quite a lot of space. By serializing inodes, we could store inodes far more efficient.
- More mindful allocation
  The space allocator will simply grab the first random free block it finds, without checking if an existing file would be better off allocating this block.

The filesystem has shown itself to be a stable and efficient filesystem with excellent transfer speeds.

Bibliography

[1] Werner Almesberger and Benno van den Brink. Active block i/o scheduling system. Website, 2004. `http://abiss.sourceforge.net/doc/abiss-lk.ps`.

[2] Maurice J. Bach. *The Design of the UNIX Operating System.* Prentice Hall, 1988.

[3] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel.* O'Reilly & Associates, 2001.

[4] Alessandro Rubini & Greg Kroah-Hartman Jonathan Corbet. *Linux Device Drivers.* O'Reilly & Associates, 2001.

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

[6] Marshall Kirk McKusick and George V. Neville-Neil. *The Design and Implementation of the FreeBSD Operating System.* Addison-Wesley, 2005.

[7] Job Wildschut. *Graduation Report: Linux IO Performance.* Philips Research, 2004.

Index