Graduation Report

Time is of the Essence

Implementation of the LIMEFS Realtime Linux Filesystem

Rink Springer

Revision 1.1
2nd June 2005

STAGEVERSLAG VOOR FONTYS HOGESCHOOL INFORMATICA AFSTUDEERSTAGE

Gegevens Student:

Naam:	Springer, R.P.W. (Rink)
Studentnummer:	2016014
Opleiding:	Hogere Informatica Voltijd
Stageperiode:	07/02/2005 - 24/06/2005

Gegevens Bedrijf:

Naam bedrijf:	Philips Research
Afdeling:	Software Engineering Services
Plaats:	Eindhoven, Nederland
Naam bedrijfsbegeleider:	M. Geldermans

Gegevens Docentbegeleider:

Naam docentbegeleider:	W. Zijlmans

Gegevens verslag:

Titel stageverslag:	Time is of the Essence
Ondertitel stageverslag:	Implementation of the LIMEFS Realtime Linux Filesystem
Datum uitgifte stageverslag:	02/06/2005

Getekend voor gezien door bedrijfsbegeleider:

Datum:

De bedrijfsbegeleider,

Preface

This report will describe my graduation internship at Philips Research in Eindhoven. It is written in an informative manner, with the exception of my personal evaluation which, of course, can't be objective or purely informative. The report is written using the guidelines as outlined in [2].

The goal of the internship was to design and implement a real time filesystem for Linux. This is illustrated further in the project description on page 3.

The target audience is mostly people having an interest in the process or the execution of my internship. The technical documentation of the project is a separate document, which is included as appendix A. The project management plan, or *plan van aanpak*, is included as appendix B.

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

Rink Springer
May 2005

Contents

Summary

This report provides an in-depth view of my graduation internship at Philips Research. The direct result is a Linux filesystem which can be compiled directly into the kernel itself, utilities to create, verify and debug the filesystem and finally a technical report on the internals of the filesystem, as well as the graduation report, which you are currently reading.

During this internship, I've looked deeply in the Linux kernel, especially with regard to filesystems and disk I/O. This resulted in knowledge about Linux kernel internals, which will surely be useful in the future. The aspects of Extreme Programming were also widely applied.

Samenvatting

Dit verslag biedt een overzicht van mijn afstudeerstage bij Philips Research. Het directe resultaat is een Linux filesysteem dat direct in de Linux kernel opgenomen kan worden, programma's om het filesysteem aan te maken, te controleren en te debuggen. Tot slot is er een rapport geschreven dat de technische aspecten van het filesysteem belicht en een het uiteindelijke afstudeerverslag dat u nu aan het lezen bent.

Tijdens de stage heb ik erg diep in de Linux kernel gekeken, waarbij vooral de filesystemen en het I/O systeem centraal stond. Het resultaat hiervan is kennis over de interne Linux kernel structuren, die zeker van pas zullen komen in de toekomst. Verder heb ik ook de Extreme Programming-methode erg veel toegepast.

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]
CRE	Corporate Research Exhibition
ext3	Third Version of the Extended Filesystem, the default Linux filesystems
extent	An (offset, length) pair
FAT	File Allocation Table, the original MS-DOS filesystem
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. [3]
LIMEFS	Large-file metadata-In-Memory Extend-based File System, the filesystem created during the internship
meta data	Data about data, used to store administration about other data. Mainly used by filesystems
RTFS	Real Time File System, filesystem designed by Philips Research for realtime storage requirements
userland	Code implemented in ordinary applications
kernelspace	Code implemented within the operating system's kernel

Chapter 1

Introduction

Koninklijke Philips Electronics N.V. is Europe's biggest manufacturer of electronics, as well as one of the biggest in the world. Philips' core business is health care, lifestyle and technology.

As of 2005, the enterprise employs over 166.800 people, in over 60 countries. Out of these people, about 2.100 are employed in Philips Research, which was founded in 1914 in Eindhoven.

This report will highlight the graduation project of Rink Springer, which is the design and implementation of a Linux real time file system. The project itself will be described in the next chapter, on page 3. The actual technical overview of the report is outlined in appendix A.

The next chapters discuss an evaluation of the project, a conclusion of the project and an overview of all used bibliography.

Chapter 2

Project

During the first month of the project, I've been installing my computers and clarifying my assignment. As later outlined in section 3.4.1, there were three options possible. During this month, all these options were carefully evaluated by the end of this month in order to determine the goal of the internship. The definitive project outlined in the next paragraph.

The project is about designing and implementing a real time file system for the Linux operating system. There are already a lot of filesystems around, but this filesystems has the following distinctive features:

- Disk block size
  Instead of traditional sectors, this file system uses a user-definable block size for all disk transactions. This is designed to improve performance.
- Larger data block size
  Whereas traditional filesystems usually tend to use 64KB per data block, this filesystem uses a default of 4MB blocks. This helps avoiding overhead.
- Meta-data in memory
  Unlike traditional filesystems, this filesystem will keep all meta-data in memory. This ensures no disk access is needed to fetch this information. Most administration is built on-the-fly upon mount time.
- Extent structure
  The filesystem uses an extent structure to maintain data block allocations.

This filesystem must be implemented within the kernel of the Linux operating system. Furthermore, the following utilities must be implemented as well:

- Make FileSystem (mkfs)
  This will create new filesystem structures on a given disk.
- FileSystem ChecK (fsck)
  Used to validate the internal filesystem structures and repair the filesystem as needed.
- Dump FileSystem (dumpfs)
  Displays the complete filesystem structures as they are stored on the disk. Mainly used as a debugging tool.

Last but not least, the entire filesystem must be documented in a technical report. This report can be found as appendix A.

Chapter 3

Evaluation

3.1 Planning using PPTS

PPTS, the Project Planning & Tracking System, is the planning system in use at SES. It is suited for extreme programming projects by defining the workload into user stories. Each user story is divided into tasks, to which you can assign an estimation and the actual used working hours. All this information is nicely plotted in a graph, which is then used to predict how close we are to the iteration's result.

For the first few weeks, I've used PPTS nicely in order to keep a planning. However, as things progressed, eventually I kept forgetting to store my stories, tasks and hours within PPTS. My mentor agreed PPTS is a bit of an overkill for my project, as I am the only person working at it. Therefore, my involvement with PPTS was brief.

3.2 Code reviewing

By the end of the 5^{th} iteration, Mischa wanted to review some of my code. The result of this was the two of us having discussions of about an hour which basically came down to personal taste. Of course, there were some parts in which he totally was correct to point some issues, but most of the discussion was purely a matter of personal coding style.

After this came up a few times, we agreed that we'd no longer discuss minor details but would rather focus on the bigger picture. Since then, we have been on the same track while browsing through the code. These reviews have been very helpful to determine strong and weak points of the implementation, all of which were eventually patched up to reach the degree of quality desired.

3.3 Working with a team

During the internship, I was part of the eHub team. As required by Extreme Programming, all team members work close to each other (in our case, in the same room) so knowledge is easily available and you don't need to spend hours looking for people in case you have a question or are stuck.

It can also be very beneficial to have other people take a look at your code if you encounter a problem, as it is common to overlook simple issues. Even explaining the issue to someone else can be a great help, as it forces you to think about what you actually created. As such, I have helped coworkers with their problems and they have returned this favor to me. This approach was kind of new to me, as communication on my previous internships was not as informal or helpful.

3.4 Iterations

To Extreme Programming, an iteration is a predefined time period, 2 weeks in our case. At the beginning of each iteration, it will be determined what will be done during the iteration. Based on this, a planning is made and the work is divided between team members.

3.4.1 Iteration 1 and 2: Getting started

7 February - 4 March

Activities

The first month of the internship, this mostly came down to meeting people, setting up the work environment and having discussions about what I would be doing.

During this discussion, the following possibilities arose:

- Examine the ext3 filesystem
  This is the de facto Linux filesystem, which may be patched so it will suit our needs.
- Port RTFS to Linux
  Philips has made their own, userland realtime filesystem. It may be a good option to port this to Linux.

- Create a new Linux realtime filesystem
  Based on the RTFS ideas, it may be useful to design a new filesystem based on RTFS ideas.

In order to have some knowledge in advance, I quickly took a look on how ext3 is organized, and some minor looks at the codebase. Since the RTFS codebase was unavailable at the time, I also read some whitepapers to determine the design goals of the RTFS.

One of the issues with RTFS were its severe limitations: it was designed for limited environments, in which memory is sparse. A redesign of the filesystem could take the filesystem to the next level, in which these issues have disappeared.

Results

Based on my studies of ext3 and the RTFS, it was determined that patching up ext3 to suit our needs would be a too big risk. It would be a project on its own to dig through ext3's codebase and design, let alone suggest/implement improvements. Therefore, this option was deemed unrealistic.

At the end of this discussion, we determined that it would be much more beneficial and maintainable to design a new filesystem from scratch than to keep patching up RTFS itself. Therefore, a new filesystem would be created, LIMEFS.

3.4.2 Iteration 3: Designing the filesystem

8 - 18 March

Activities

As the goal was laid out, it was time to start designing the filesystem itself. First, this meant all different aspects of RTFS had to be examined and embedded in our filesystem.

After some careful studying, both Mischa and I developed proposals for the data structures to be used. After careful consideration, we merged the best of both designs together.

Results

By the end of this iteration, the filesystem design was finalized. Both my company mentor Mischa, who was involved in RTFS work and I were satisfied with the new design. LIMEFS still remains a very close relative of the original RTFS, but is much better geared towards current requirements.

3.4.3 Iteration 4: Starting the implementation

21 March - 1 April

Activities

During this iteration, the true implementation of the filesystem began. At first, I studied a lot of existing filesystems to determine the filesystem interface towards the kernel. From then on, a simple skeleton filesystem was created. This skeleton was loadable as a kernel module and would return failure on every operation. This simplified development by adding features one by one.

Along with coding the filesystem itself, development of the filesystem creation utility, `mkfs.limefs`, began. This utility creates a filesystem on a disk. The first version featured some debugging hacks which would allow it to create preliminary files to aid in debugging.

Results

At the end of the iteration, I was able to create a new filesystem, which could be mounted and used. Files would always be empty, since especially the file reading/writing was not yet implemented.

3.4.4 Iteration 5: Reading/writing files

4 - 15 April

Activities

As the more basic functionality was completed, file reads/writes were implemented. This proved to be challenging, as some parts are hard to debug sensibly.

This functionality needed more administrative code to be written, such as the free space administration code. In order to enforce quality on such an important subsystem, software tests were designed to maintain a correct implementation.

Results

A debugging tool, `dumpfs`, was written to dump the filesystem structures in a human-readable form for easier debugging. Also, tests were written for the extent administration code (refer to the technical report for more information).

By the end of this iteration, it was possible to make actual use of the filesystem by creating, removing, reading and writing files and directories.

3.4.5 Iteration 6: Documentation and ABISS research

18 - 29 April

Activities

Since the last iterations were all around designing and coding the filesystem, this iteration was mainly spent on running tests on the filesystem and starting writing the reports.

Along with this documentation, effort was made to look into the ABISS system. ABISS, the Active Block I/O Scheduling System, is a Linux kernel subsystem which can provide realtime disk scheduling to applications [1]. In order to do this, some filesystem information must be passed to ABISS. Some time of this iteration was spent to review ABISS, and mainly the integration between filesystems and ABISS.

Results

Most of the I/O chapter of the technical report was written, along with some minor parts of the LIMEFS chapter. The graduation report was started, but only the table of contents was written and submitted for review.

The operation of the ABISS framework was quite clear, along with the knowledge needed to integrate the filesystem with ABISS.

3.4.6 Iteration 7: More documentation, cleanups

2 - 13 May

Activities

This iteration was all about documentation and cleaning up. The technical report was written during this time, and some time invested to clean up the existing codebase.

Results

The LIMEFS chapter of the technical report was mostly finished, along with huge parts of the implementation chapter. The codebase was massively cleaned up after careful evaluation with Mischa.

3.4.7 Iteration 8: Fragmentation study, even more reports

16 - 27 May

Activities

Ad has been conducting some experiments to see how file fragmentation would affect the filesystem during normal operations. This tool would simulate an extent-based filesystem housing a lot of files. Should the filesystem be filled, a file would randomly be deleted, making space for the new data.

During this iteration, the technical report was finished and submitted to Mischa for reviewing. The remaining time was spent finishing this report.

Results

By the end of this iteration, the file fragmentation section of the technical report was written and the rest of the report was finished. It was then submitted for review.

A draft of this report itself was also finished during this iteration and submitted for review.

At the time of writing, iterations 9 and 10 were not yet started. Therefore, the descriptions given here are the planned activities

3.4.8 Iteration 9: Finishing up

30 May - 10 June

Activities

During this iteration, all reports were finished and submitted to Fontys. Work on the presentation began, as well as more and more tests to determine the bottlenecks of the filesystem as well as its performance.

3.4.9 Iteration 10: The Living End

13 - 24 June

Activities

From output gathered by the tests, the filesystem was further optimized. The graduation presentation was presented at Fontys. The remaining time was spent on finishing the project.

Chapter 4

Personal Evaluation

4.1 Learning experiences

As of any internship, the goal is twofold: for one, my knowledge should be expanded. And not solely study-related knowledge, but also personal knowledge: how to deal with problems in a project, how to talk to people... things of a more social nature. Being part of a true project at a company, so you get an impression how everything is organised and to be a part of it. Starting a project from the beginning to the end.

4.1.1 Study-related

I've learned a lot on Linux kernel internals. As I am mostly a BSD user myself, I had finally gotten the chance to get familiar with Linux.

I learned a lot about filesystems and their implementation within the Linux kernel. Not only the plain design of a filesystem, but also how this is entangled within the kernel itself.

Debugging code is always a challenge; this is especially true for kernel code. There is no sensible debugger, no breakpoints ... all came down to simple print statements and tracing kernel panics. I didn't have much experience in Linux-specific debugging when I started this assignment.

4.1.2 Personal

As for my personal learning experiences, I feel I've now really gotten an impression how a huge company is being organized. My previous internships

were at small to medium organizations, therefore this was quite an interesting experience.

Communication is really valued at Philips Research ... in fact, it is a requirement. If you cannot express yourself towards your colleagues, the environment won't be very enjoyable. At the beginning of each day, not only did I have a talk with my company mentor, but also with the entire team.

The latter took place in a so-called standup meeting around 9:00. The purpose of this meeting is to inform everyone what you are doing, what went well and what didn't. This is very useful, for people who have knowledge in the area you are struggling in will tell you, so you immediately know who you can ask for help.

People were surprised to see the discussions between Mischa and me. From the start of the project, it was made clear that the both of us had strong opinions and ideas about how things should be done. However, our disagreements were not to annoy each other, but rather to ensure we would achieve the best result on the project. I believe these discussions have really paid off and at the end, I am glad we were able to discuss with each other in such a way.

4.2 Improvements

This section will only focus on the suggested improvements regarding the process. Technical improvements are outlined in the technical report.

As for the reports, even as a lot of time has been reserved for them, it proved hard to get people to review them in time. This was also mostly to a Philips event taking place in June, the CRE, which usurps most people's time. I would suggest anyone having an internship at Philips to take this into account.

This internship has proven once more how important good communication is. I am convinced my mentor and I have had good communication, but since we were both in separate buildings, it never really motivated me to walk over for quick questions or comments. This can be seen as a defect on my side.

4.3 Working at Philips Research

While working at Philips Research, I really felt like being part of a team. People all around you are working on different aspects of the program, and

occasionally ask you to help them with bugs they keep overlooking, general comments on their code and such. And of course, the reverse is true as well.

I was really impressed by the facilities present, and even more so with the facilities which could be arranged if needed. At previous internships, I was used to working with deprecated hardware previously used by employees who got new machines. However, anything I needed (a faster computer, a harddisk for running tests on, a new screen, a good mouse etc) were mostly just a call to Ad away.

Finally, what really motivated me was the fact that people take you extremely seriously. They consider you part of the team, and will work with you to resolve any issues. At previous internships, I got the impression I was considered just a student, who shouldn't be taken too seriously.

Chapter 5

Conclusion

During this internship, I've learned a lot about the Linux kernel, especially with respect to file systems and block I/O layers.

I feel I have gotten a very good impression of Extreme Programming. It is extremely useful to see how this method is being used in a company, and how everything is managed on top of this method. Writing and using tests was very hard while doing kernel development, but several administrative parts of the filesystem were designed so they could easily be tested.

Furthermore, I was surprised at the freedom which I had while performing this project. During previous internships, my mentor there gave me a specific task which needed to be carried out. During this internship, my internship mentor actually encouraged me to make suggestions and come up with my own ideas. Working within a true team was also extremely pleasant, not to mention a new experience to me.

Finally, I am very pleased with the result of this internship. Not only have I learned a lot about Linux filesystems and I/O internals, but also about Extreme Programming and how very useful it can be to help someone if they are stuck with a problem in their code.

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] Wim Hoogland. *Rapport over Rapporteren.* Wolters-Noordhoff, 1998.

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

Appendix A

Technical Report

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:

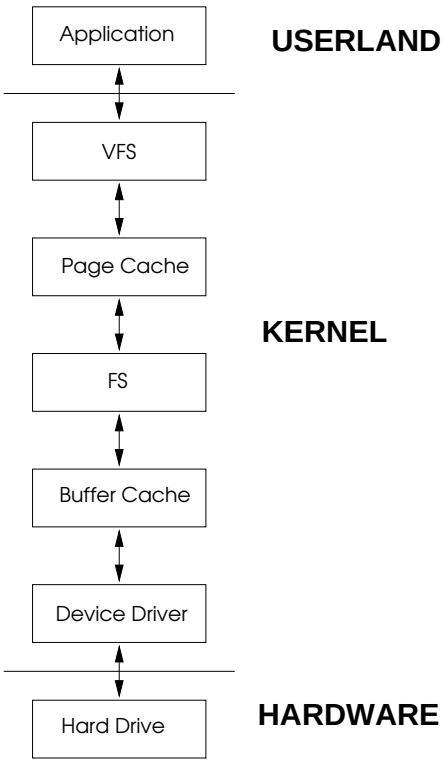


Figure 2.1: Summary of Linux I/O layers

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:

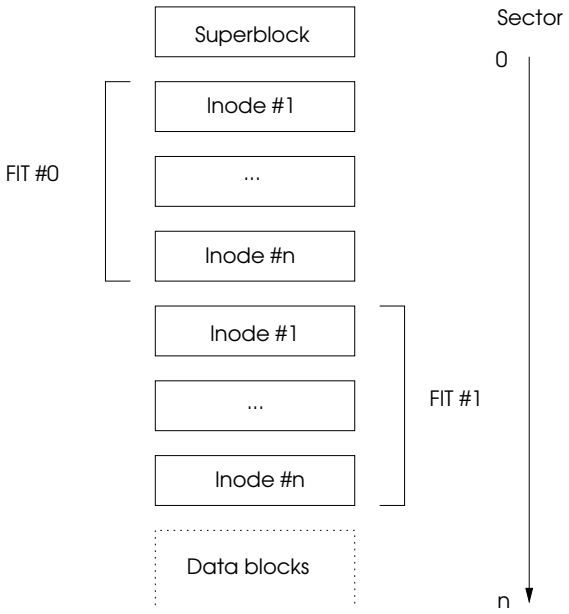


Figure 3.1: On-disk structure of the filesystem

- 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:

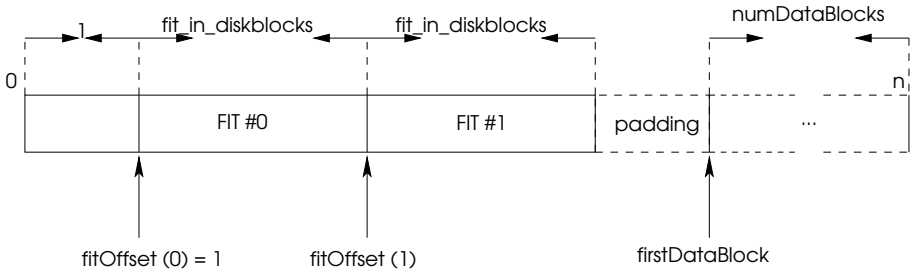


Figure 3.2: Internal offsets of on-disk structures

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.

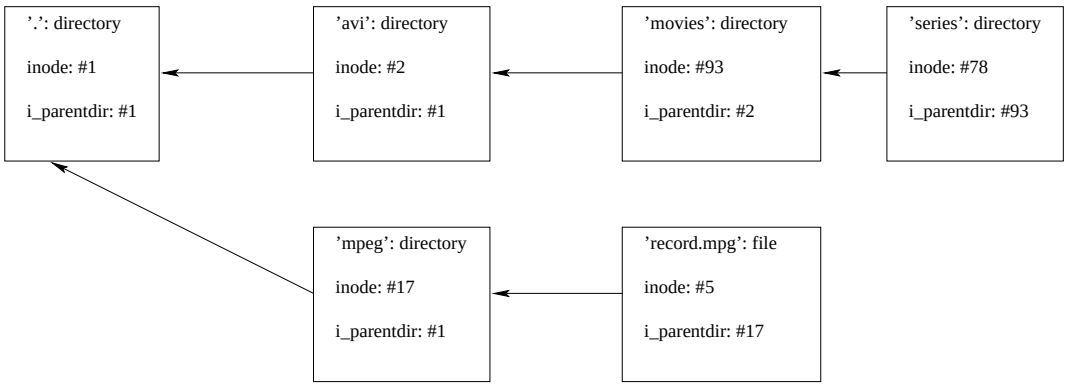


Figure 3.3: Nesting of directories

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:

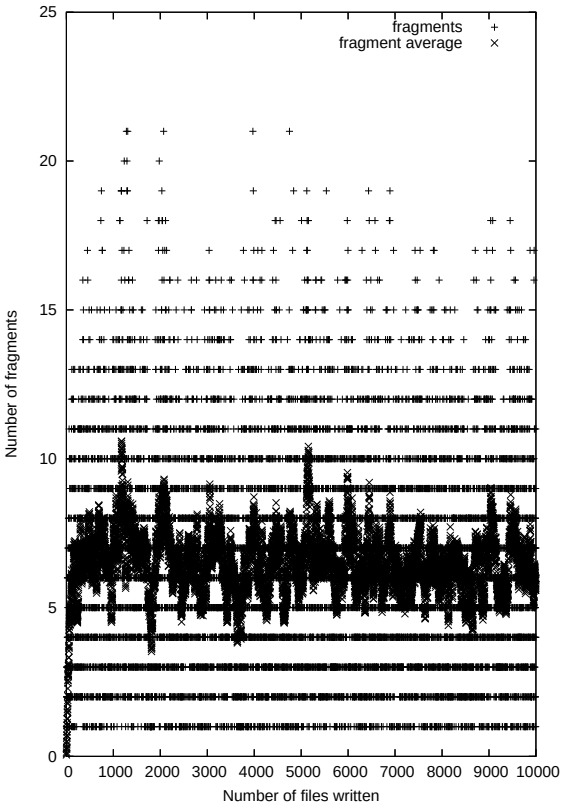


Figure 3.4: Fragments per file

[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:

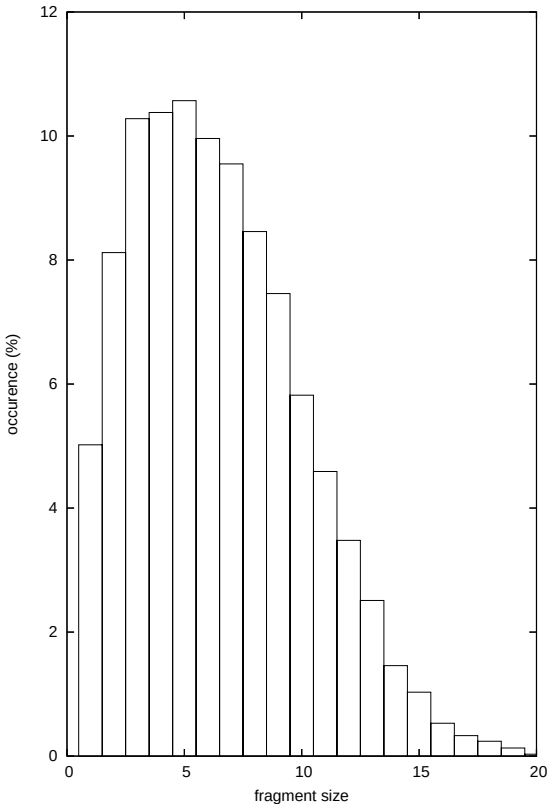


Figure 3.5: Fragment occurrence

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:

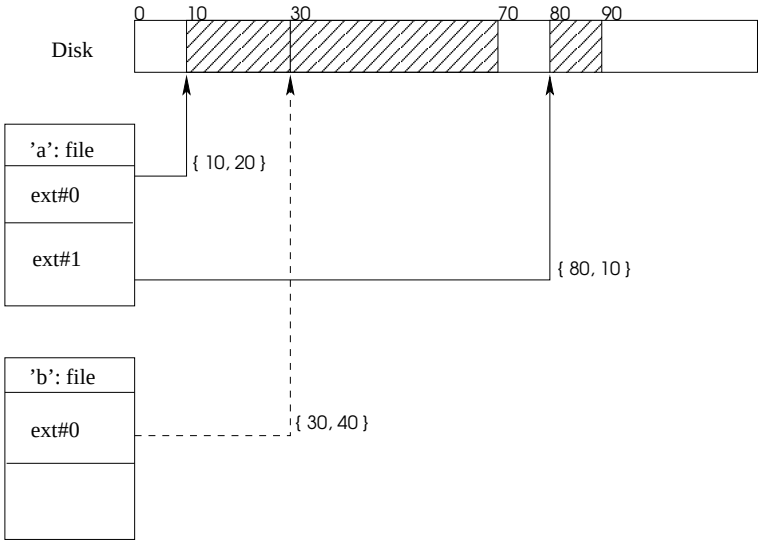


Figure 3.6: File-extent administration

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

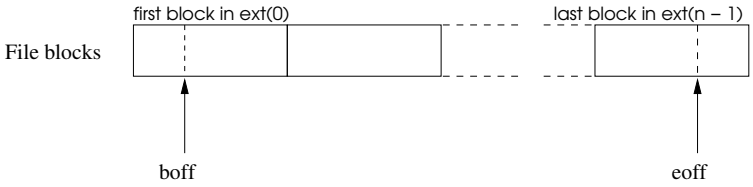


Figure 3.7: File begin- and end offsets

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.

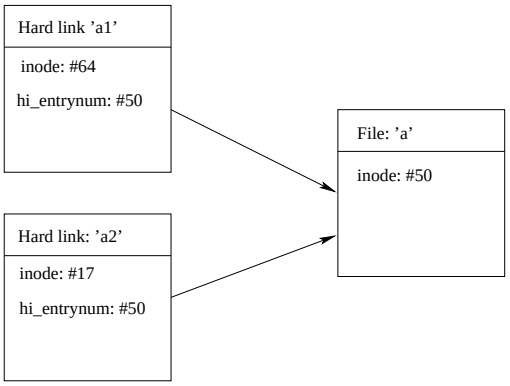


Figure 3.8: Hard link and file relationship

```
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

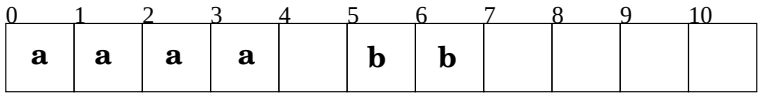
4.1 Allocation

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:

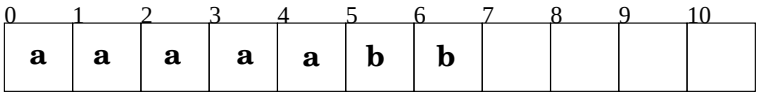


a = ex#0: { start = 0, len = 4 }
b = ex#0: {start = 5, len = 2 }

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

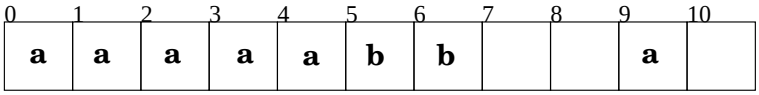
following:



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

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

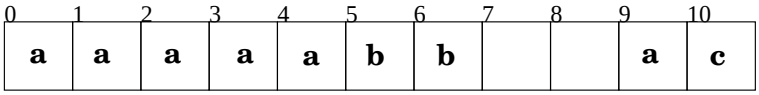
However, suppose file *a* wants another block. It may not overwrite the blocks already in use by file *b*, so we allocate a random block:



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:

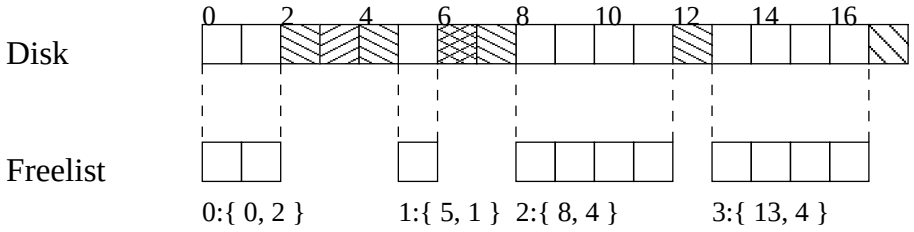


Figure 4.5: Freelist: initial status

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:

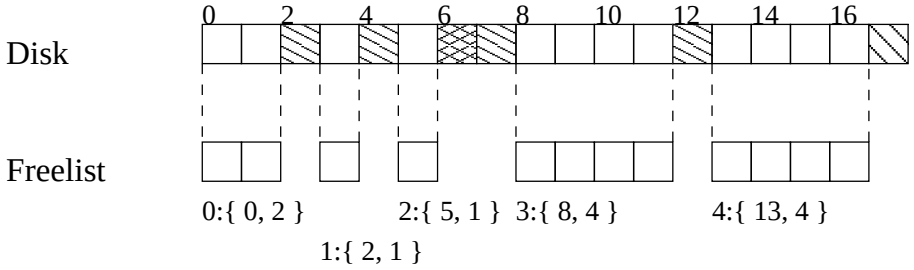


Figure 4.6: Freelist: freed block 3, gives new extent

[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:

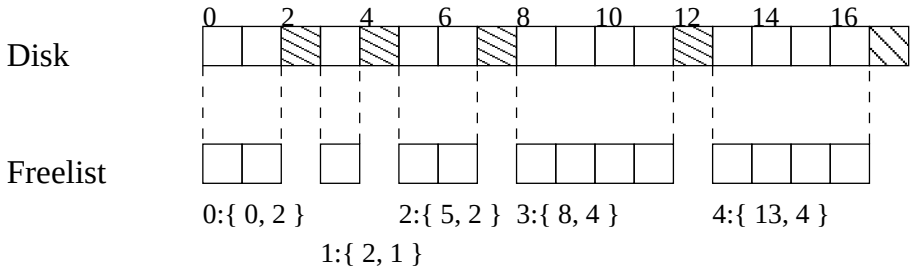


Figure 4.7: Freelist: freed block 6, expands extent

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

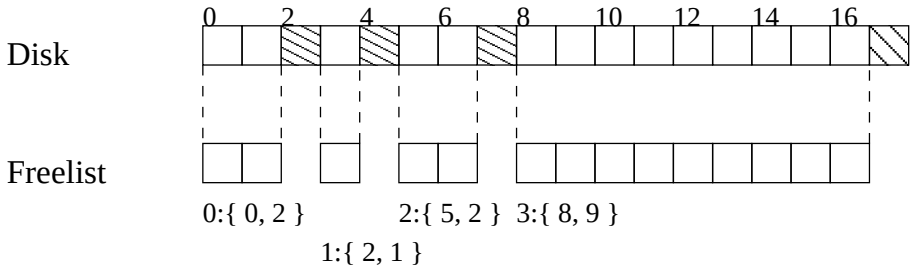


Figure 4.8: Freelist: freed block 12, merges extents

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

Appendix B

Plan van Aanpak

Plan van Aanpak

Personal Video Recording

Rink Springer

Versie 1.1
11 maart 2005

Voorwoord

Dit Plan van Aanpak is geschreven tijdens mijn afstudeerstage bij de firma Philips, afdeling Research. Het doel is de betrokkenen te informeren over mijn aanpak van het project en is in de eerste instantie bestemd voor mensen die in direct verband met de stage staan.

Inhoudsopgave

Samenvatting

Dit plan van aanpak behandelt de opzet van de Digital Video Recording afstudeer opdracht.

De volgende hoofdstukken gaan in op verscheidene zaken met betrekking tot de opdracht, zoals:

- Bedrijfsprofiel van het stagebedrijf, Philips
- Gedetailleerde beschrijving van de gebruikte ontwikkelmethode
- Gedetailleerde beschrijving van het project
- Overzicht van de planning

Verklarende woordenlijst

DVB	Digital Video Broadcasting, techniek om over een analoog frequentiedomein digitaal meerdere zenders tegelijk te versturen.
XP	Extreme Programming, software ontwikkelings methode, beschreven op pagina 5.
Linux	Open source UNIX-clone, waaraan sinds 1991 actief aan wordt ontwikkeld. Gebruik en code zijn voor iedereen vrij.
MythTV	Open source Linux project om je computer in een multimedia station om te zetten.

Hoofdstuk 1

Inleiding

Sinds de introductie van de televisie in de jaren '30 is deze voor veel mensen steeds belangrijker geworden. Daarom blijft het aantrekkelijk om nieuwe ontwikkelingen te doen op dit gebied, gezien de enorme vraag.

Eind jaren '50 kwam de uitvinding van de video recorder. Dit stelde mensen in staat om eenvoudig zelf opnamen te maken. Een limitatie hierbij is wel, dat maximaal één opname tegelijk gemaakt kan worden.

Sinds de uitvinding van DVB[1] is het mogelijk om over een frequentiebereik waar voorheen slechts één enkele zender over verzonden kon worden, 5 tot 8 zenders te verzenden. Dit maakt het mogelijk om meer zenders te ontvangen, alsmede meer zenders tegelijk op te nemen.

Het is daarom zeker de moeite waard, om te kijken of het mogelijk is om het bestaande systeem uit te breiden zodat het mogelijk is om door middel van DVB meerdere zenders tegelijk op te nemen. Hier worden nog problemen mee verwacht, die uitgezocht dienen te worden.

Hierbij komen een paar zaken om de hoek kijken. Deze zullen opgesomd worden in hoofdstuk 4 op pagina 9, waar de projectbeschrijving te vinden is.

[1] Digital Video Broadcasting

Hoofdstuk 2

Bedrijfsprofiel

2.1 Philips

Koninklijke Philips Electronics N.V. is de grootste elektronicafabrikant van Europa en één van de grootste ter wereld. Philips is actief op drie aan elkaar gerelateerde gebieden: gezondheidszorg, lifestyle en technologie.

De onderneming heeft 166.800 werknemers in dienst, verspreid over meer dan zestig landen. Het bedrijf is marktleider op het gebied van medische diagnostische beeldvormende apparatuur en patiëntenmonitoring, kleurentelevisies, elektrische scheerapparaten, verlichting en siliciumsysteemoplossingen.

2.2 Philips Research

Philips speelt een belangrijke rol bij het vormgeven van de wereld van de digitale elektronica door zinvolle technologische innovaties te bieden. Veel van deze innovaties vinden hun oorsprong in de laboratoria van Philips Research.

Philips Research, dat werd opgericht in Eindhoven in 1914, is één van de grootste particuliere onderzoeksorganisaties ter wereld. De organisatie heeft vestigingen in Nederland, België, het Verenigd Koninkrijk, Duitsland, de Verenigde Staten, China en India. Philips Research heeft in totaal circa 2.100 medewerkers in dienst.

Met de vindingen van Philips Research worden doorbraken bereikt in hoe mensen technologieën ervaren. Onderzoek is steeds meer gericht op de strategische activiteiten van Philips: gezondheidszorg, lifestyle en technologie. In de visie van ambient intelligence zullen apparaten in huis steeds meer uit

het zicht verdwijnen. Hun plaats zal ingenomen worden door een netwerk van systemen die op de achtergrond de gewenste functionaliteit bieden.

Wetenschappers afkomstig uit een breed scala van disciplines - van elektrotechniek en natuurkunde tot scheikunde, wiskunde, mechanica, informatietechnologie en software - werken nauw samen, waardoor zij invloed op elkaar uitoefenen en elkaars visie verbreden. Zo halen zij voordeel uit synergie en kruisbestuiving van ideeën.

2.3 SES

Software Enginering Services, oftewel SES, is de afdeling die software ondersteuning bied aan de andere groepen. Dit kunnen zowel groepen als mensen, binnen of buiten Philips zijn.

Qua software ondersteuning worden applicaties ontwikkeld, maar vaak worden ook drivers gemaakt voor eigen ontwikkelde IC's, apparaten en randapperatuur.

De medewerkers van SES werken veelal op een inhuurbasis. Er zijn een aantal vaste medewerkers, maar het merendeel wordt extern ingehuurd. Dit zorgt voor diversiteit onder de werknemers.

Hoofdstuk 3

Software ontwerp methode

Door de korte projectduur en snel wisselende prioriteiten van taken is het gebruik van planningmethodologie binnen research vrijwel onmogelijk.

Om toch gestructureerd te werken, word er vooral gewerkt met de Extreme Programming methode. Gezien Extreme Programming op zich geen projectmanagement met zich meebrengt, wordt de methode Extreme Programming@Scrum gebruikt. Deze methode zal in de volgende paragraaf uitgelegd worden.

3.1 Extreme Programming

Extreme Programming, oftewel XP, bestaat uit een aantal key practices die gelijktijdig in extreme vorm worden toegepast. XP beschrijft hoe ontwikkelaars te werk moeten gaan, maar niet hoe men de ontwikkelingsmethode binnen het bedrijf toepast of hoe men de manier van werken optimaliseert. Daarvoor wordt Scrum gebruikt als een aanvulling op XP. Scrum biedt mogelijkheden voor het management, waarmee ook de CMM normen gehaald kunnen worden.

XP zelf is geschikt voor softwareproducten die onder snel veranderende omstandigheden gemaakt moeten worden. Daarbij komt nog dat XP rekening houdt met een wisselende samenstelling van het ontwikkelteam en veranderende wensen van de klant.

Aan het begin van het project wordt samen met de opdrachtgever besproken welke eigenschappen er waardevol zijn, deze worden vervolgens omschreven in de User Stories. Deze user stories worden daarna in een prioriteit volgorde gezet en zullen in die volgorde afgehandeld worden. Dit gebeurt met het backlog systeem, wat bekend is uit de Scrum methodiek.

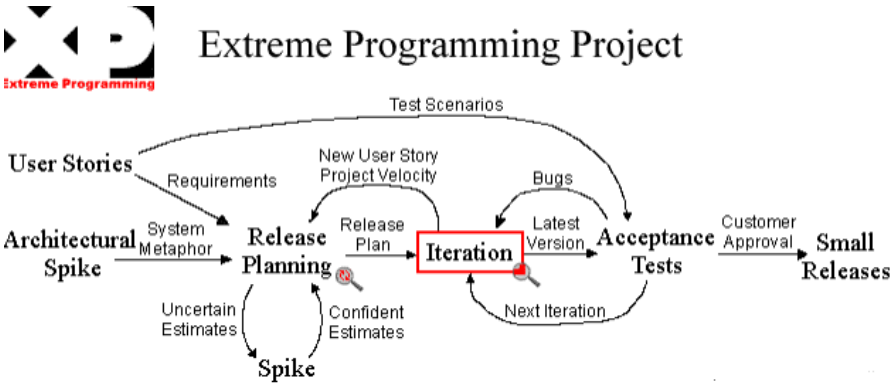
In de backlog worden alle user stories verzameld en in volgorde gezet. Er wordt dan een schatting gemaakt van de tijd die nodig is om de user stories te bouwen. Het plannen van het geheel is een dialoog tussen de opdrachtgever en de ontwikkelaars, deze sessies vinden aan het begin van elke iteratie plaats.

Een iteratie is niks anders dan een vast afgesproken tijdsinterval. User stories worden dusdanig gedefinieerd dat ze binnen een iteratie uitgevoerd kunnen worden. Hoeveel user stories er gebouwd kunnen worden in een iteratie hangt af van de team-velocity (velocity is een percentage dat aangeeft hoe goed men de tijdsschatting doet), de lengte van de iteratie en de grootte van het team.

De user stories worden verder opgesplitst in taken. De ontwikkelaars binnen het project zijn er persoonlijk verantwoordelijk voor dat hun taken aan het eind van elke iteratie af zijn zodat de user story afgesloten kan worden.

Het is beter om de complexiteit te reduceren door alleen dat te bouwen wat vandaag nodig is. Op deze manier bouwt men een zo simpel mogelijk systeem. Toekomstige wijzigingen kunnen uitgevoerd worden wanneer ze nodig zijn. Maar zou je nu iets kunnen doen om het straks makkelijker te maken dan moet je dat niet achterwege laten.

De volgende illustratie geeft een kort overzicht van de cyclus van Extreme Programming:



3.2 Belangrijke Key Practices

- Gebruik van coding standaards
- On-site customer: de klant is nauw betrokken bij het project, en men werkt op een locatie vlakbij de klant. Hierdoor is de noodzaak voor

geschreven documentatie veel minder door de korte communicatie lijnen.

- Er vinden regelmatig releases plaats zodat de klant en de gebruikers al vanaf een vroeg stadium (binnen enkele weken) een werkende versie van de software kunnen beoordelen. Er wordt dan gekeken of alle taken zijn afgerond en werken en of het product voldoet aan de wensen van de klant.
- Voordat er code geschreven wordt in XP, wordt eerst een unit test voor de code geschreven.
  - Het schrijven van unit tests is in feite het schrijven van een specificatie.
  - Test first laat je nadenken over hoe de code gebruikt gaat worden, waardoor van een hoger abstractieniveau sprake is.
  - De unit test vormt een vangnet voor het veilig aanpassen en herstructureren van de software.
  - Unit tests zijn een vorm van documentatie: ze bevatten voorbeelden van het gebruik van bepaalde code. Je weet wanneer je klaar bent, namelijk als alle testen positief zijn.
- XP past continue herstructureren van het ontwerp en de code toe als middel om de software maximaal simpel (dus minder noodzaak voor geschreven documentatie) te houden. Het herstructureren gebeurt in kleine, beheersbare stappen (genaamd refactoring), ondersteund door de reeds bestaande tests.
- Simpel wil zeggen:
  - Geen duplicate code.
  - Gestructureerd, zodat nieuwe functionaliteit makkelijk te integreren valt.
- Pair programming, het programmeren in paren maakt code kwalitatief beter. De volgende regels worden toegepast bij pair programming:
  - In principe wordt alle productie code (inclusief de tests) in paren geschreven.
  - Problemen worden altijd met een paar opgelost, in de meeste gevallen bestrijkt een probleem meer dan een module.
  - Documentatie mag door een enkeling geschreven worden, daarna moet een review plaatsvinden.

Hoofdstuk 4

Het project

4.1 Informatie vooraf

Sinds de uitvinding van DVB is het mogelijk om over het frequentiebereik waar voorheen slechts één enkele zender over te versturen was, 5 tot 8 zenders te versturen. Deze ontwikkeling maakt het mogelijk om met behulp van slechts een enkele tuner meerdere zenders tegelijk te ontvangen.

Als er de beschikking is over 4 tuners, dan is het mogelijk om 20 tot 32 zenders tegelijk te ontvangen. Gezien er maar één zender tegelijk bekeken wordt[1] lijkt dit niet zinnig. Maar het is wel zinnig om de overige 19 tot 31 zenders op te nemen, als de gebruiker dit tenminste wil.

De bestaande oplossing is gebaseerd op het MythTV project[2], een Linux-gebaseerd systeem waarmee je een PC kan omgezetten naar een huis multimedia systeem. Je kunt bijvoorbeeld door middel van een TV tuner kaart opnemen naar een harde schijf. Er is alleen weinig ervaring met betrekking tot het opnemen van meer dan 2 kanalen tegelijkertijd.

De bedoeling is, om ons zoveel mogelijk te richten op de harddiskactiviteit binnen Linux. Uiteraard zijn er veel meer factoren die meespelen, maar de bedoeling van deze stage is om ons te richten op de realtime disk I/O binnen Linux en dit zonodig te verbeteren zodat het aan de gestelde eisen voldoet.

4.2 Probleemstelling

De volgende zaken zijn onbekend en dienen onderzocht te worden:

[1] Uiteraard is meer mogelijk, maar één film volgen is vaak al een uitdaging genoeg
[2] http://www.mythtv.org

- Elke zender gebruikt een zekere bandbreedte. Kan Linux deze noodzakelijke realtime disk performance garanderen?
- Welk bestandssysteem is geschikt? Wellicht standaard FAT of ext3, of misschien het zelfgemaakte realtime filesysteem? Of kunnen er wellicht aanpassingen gedaan worden om de snelheid te verbeteren?
- Is het mogelijk om de verschillende zenders binnen de DVB stream in software van elkaar realtime te isoleren? Voorlopig is bekend dat er geen extra hardware zal zijn hiervoor.

Uiteraard kan de verdere loop van het project pas bepaald worden naarmate deze zaken bekend raken. Het bovenstaande lijstje in op prioriteit gesorteerd,

Na een studie met betrekking tot de voorspelbaarheid van de filesystemen, is er besloten om zelf het binnen Philips ontwikkelde Real Time File Systeem op te pakken, en hier een heus Linux filesysteem van te maken met de benodige tools.

4.3 Resultaat

De bedoeling is vooral antwoord te krijgen op de bovenstaande vragen. Gezien het een research omgeving is, is de noodzaak vooral een werkend geheel te kunnen laten zien wat later als basis voor een project gebruikt kan worden.

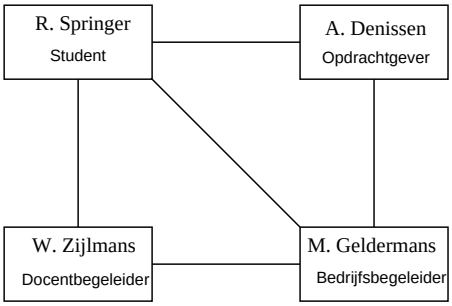
Zoals hierboven te vinden is, is het directe projectresultaat het Real Time File Systeem, maar dan als Linux filesysteem. Dit gezamelijk met alle benodigde tools.

4.4 Organisatie

De opdrachtgever van het project is de firma Philips, afdeling Research. Deze organisatie heeft als opdrachtgever de heer A. Denissen aangewezen, de stagebegeleider is de heer M. Geldermans.

De opdrachtnemer is de heer R. Springer, student van Fontys Hogescholen te Eindhoven. De begeleidende docent is de heer W. Zijlmans.

Zie de illustratie hieronder:



4.5 Randvoorwaarden

De randvoorwaarden van dit project zijn als volgt:

- Alle toegezegde middelen blijven beschikbaar tot het eind van de stageperiode.
- Er is genoeg tijd en capaciteit beschikbaar om de student te ondersteuning.
- De stagiair is vrij in de keuze van de door hem gebruikte programmatuur, mits licentievoorwaarden dit toelaten.

4.6 Risico's

Zoals bij elke stageopdracht is er het risico dat er niet genoeg kennis bij de stagiair aanwezig is, wat als resultaat heeft dat het project niet op tijd af komt. Aan de hand van de ervaring van de bedrijfsbegeleider en de kennis van de aanwezige collega's binnen het bedrijf en de stagiair zelf is de kans dat dit gebeurt klein. Verder maakt de stagiair deel uit van het projectteam, zodat hulp altijd binnen handbereik ligt.

Hoofdstuk 5

Planning

Dit hoofdstuk zal een overzicht van de planning geven. Door de enorm veranderlijke aard van het project (gezien er constant zaken uitgezocht moeten worden voordat de volgende stap bepaald kan worden), zal hierbij vooral de nadruk liggen bij de beschikbare tijd.

5.1 Overzicht

Week 1	7 - 11 februari	
Week 2	14 - 18 februari	Opstart periode, schrijven PvA
Week 3	21 - 25 februari	
Week 4	28 februari - 4 maart	Definitieve opdracht bekend
Week 5	8 - 11 maart	Filesysteem mounten en unmounten
Week 6	14 - 18 maart	
Week 7	21 - 25 maart	Bestanden/directories aanmaken en verwijderen
Week 8	28 maart - 1 april	
Week 9	4 - 8 april	Bestanden lezen
Week 10	11 - 15 april	
Week 11	18 - 22 april	Integriteit controle tool
Week 12	25 - 29 april	
Week 13	2 - 5 mei	Bestanden schrijven
Week 14	9 - 13 mei	
Week 15	16 - 20 mei	Hard/softlinks
Week 16	23 - 27 mei	
Week 17	30 mei - 3 juni	Stageverslag afronden
Week 18	6 - 10 juni	
Week 19	13 - 17 juni	Afronding stage
Week 20	20 - 24 juni	

De 21^e week, 27 juni - 1 juli, zou eventueel als uitloopweek gebruikt kunnen worden.

Hoofdstuk 6

Beheersaspecten

6.1 Geld

Philips stelt een maandelijkse stagevergoeding beschikbaar, deze bedraagt €330,00 bruto per maand. Verder zal er ook de mogelijkheid om in overleg benodige zaken te kopen.

6.2 Organisatie

Zie projectorganisatie op pagina 10.

6.3 Kwaliteit

Er zal wanneer nodig overleg zijn met de bedrijfsbegeleider. Verder zal er elke twee weken gerapporteerd worden naar de docentbegeleider.

6.4 Tijd

De stage duurt tot 24 juni 2005. Op dit tijdstip moet het project afgerond zijn en aan alle formaliteiten voldaan zijn. In geval van ziekte en andere onvoorziene zaken zou er één extra week ingepland kunnen worden.

Bijlage A

Communicatieplan

Betrokkenen

Student

R. P. W. Springer
Hendrik Druckerstraat 44
5652 RJ Eindhoven
Telefoon: 06-41681662
Email: rink@stack.nl

Bedrijf

M. Geldermans
Professor Holstlaan 4
5656 AA Eindhoven
Telefoon: 040-2744742
Email: mischa.geldermans@philips.com

Fontys Hogescholen Informatica

W. Zijlmans
Rachelsmolen 1
5600 AH Eindhoven
Telefoon: 0877-870910
Email: W.Zijlmans@fontys.nl

Algemene Informatie

- Voortgangsrapportage
  De stagiair zal de docentbegeleider elke twee weken een voortgang van zijn stageactiviteiten opsturen.
- Vragen
  Als de stagiair of de docentbegeleider een vraag heeft, is email de voorkeursmanier om contact op te nemen. Voor dringende zaken kan telefonisch contact gezocht worden.
- Veranderingen
  Bij veranderingen stuurt de stagiair het vernieuwde communicatieplan per email op naar de docentbegeleider. Deze zal binnen één werkweek laten weten of hij met het nieuwe communicatieplan akkoord gaat.

Te ontvangen documenten door docentbegeleider

Wat?	Wanneer?	Hoe?
Communicatieplan	11 februari	Per email
Brief met uitnodiging bezoek Bijlage: Plan van Aanpak Bijlage: Routebeschrijving	18 februari	Per email
Stageverslag inhoudsopgave	16 mei	Per email
Stageverslag, eerste versie	25 mei	Per email
Stageverslag, tweede versie	30 mei	Per email
Stageverslag, laatste versie	3 juni	Per post
Voortgangsrapportage	Elke twee weken	Per email

Te ontvangen documenten door stagiair

Vragen en opmerking met betrekking tot de opgestuurde documenten of andere zaken, door middel van email of eventueel telefonisch.

Index

Index