Katedra počítačů
ČVUT v Praze, Fakulta Elektrotechnická
Technická 2
CZ-160 00 Praha 6

Jádro verzovacího souborového systému

Versioning file system core

Diploma thesis

Václav JŮZA

2005-12-01

Abstrakt (česky)

Tato diplomová práce obsahuje implementaci verzovacího souborového systému a rozpravu o ní. Verzovací souborový systém ukládá samočinně staré verze souborů, když jsou změněny a případně může odstraňovat nejstarší verze. Nejznámější verzovací souborový systém je použit v operačním systému OpenVMS. Můj souborový systém ukládá verze do podadresáře a jejich název je plně nastavitelný. Hlavními přínosy jsou plně přístupná metadata formou souborů v čistém textu, a co nejvyšší nastavitelnost, ne co největší výkon.

Abstract (english)

This diploma thesis contains my implementation of versioning file system and discuss it. Versioning file system stores automatically old versions of files, when files are changed and possibly removes the oldest versions. The best known versioning file system is used by OpenVMS operating system. My file system stores versions in a subdirectory and their name is fully configurable. Main features are fully accessible metadata in form of plain text files and maximum configurability, not maximum performance.

Goal

Suggest and after agreement with the project leader implement a versioning file system core, with an easy usability, management and compatibility with existing applications keeping in mind. Use an Unix-like operating system as a host environment.

Thanks

I would like to thank to Ing. Milan Juřík, my diploma thesis leader, for his comments, proposals and help on this theme.

I thank also to all the teachers, who helped me to get many useful knowledges.

I thank to my family for supporting my studying on the university as well.

Declaration

I declare, that I have done this work independently with an contribution of my diploma thesis leader and have listed all information sources.

I agree with use of the results by ČVUT FEL, the implementation under the terms of the license of its files (GNU GPL and LGPL), and this text in any way.

Contents

1 Introduction **1**

2 Theory **3**
2.1 Possible approaches 3
2.1.1 How to present the versions to the user 3
2.1.2 Where to implement version system 4
2.1.3 Implementation 5
2.2 Existing versioning solutions 5
2.2.1 OpenVMS 5
2.2.2 FreeVMS 7
2.2.3 CopyFS 8
2.2.4 Version control systems 11
2.2.5 XDelta 14
2.3 Virtual file system frameworks 17
2.3.1 LUFS 17
2.3.2 FUSE 17
2.3.3 PlasticFS 19

3 Analysis **21**
3.1 Discussion of the design 21
3.1.1 How to implement the versioning system 21
3.2 The Fsfipi framework 22
3.2.1 Standard fsfipi filters 22
3.3 Vaves, a fsfipi module for making versioned filesystem 23
3.3.1 Types of changes of files 23
3.3.2 Information to be stored 23
3.3.3 The directory structure 26
3.3.4 Removing version, files, and directories 26

4 Implementation **27**
4.1 Fuse 27
4.1.1 User's point of view 27
4.1.2 Programmer's point of view 29

4.2 The Fsfipi framework 32
4.2.1 How does it work 33
4.2.2 The interface 35
4.3 Standard fsfipi modules 38
4.4 Vaves 38
4.4.1 Issues 39
4.4.2 Source files 40
4.4.3 Data structures 41
4.4.4 Internal functions 41
4.4.5 How does the filesystem functions work 42

5 Tests **47**
5.1 The seek and write test 47
5.2 The filesystem test 48
5.3 The test of vaves specifics 48
5.4 The Bonnie++ benchmark 49
5.5 The FS_INOD test from the LTS 49

6 Benchmarks **50**
6.1 What and how it was measured 50
6.2 Results 52
6.3 Interpretation 56

7 Conclusion **58**

Bibliography **59**

Appendixes **60**

Used software **60**

List of Figures

2.1 Command flow on FUSE filesystem 18

3.1 Data model of the Vaves 24

4.1 Command flow on an Fsfipi filesystem with the Vaves filter . 34
4.2 Files used in Vaves with their default names 45
4.3 Removing a version 46

6.1 Troughput of file i/o operations on ext3 with dirindex 56
6.2 Troughput of directory operations on ext3 with dirindex . . . 56

List of Tables

6.1 Throughput of operations on default ext3 53
6.2 Throughput of operations on ext3 with dirindex 53
6.3 Throughput of operations on reiserfs 54
6.4 Throughput of operations on non-empty reiserfs 54
6.5 Throughput of operations directly on filesystem 55
6.6 Throughput of operations with vaves 55

Chapter 1

Introduction

The goal of this diploma thesis is to suggest and implement an versioning file system, which could be used in the user's home directory on an Unix-like operating system.

The main inspiration comes from OpenVMS operating system, which operates very well with file versioning, but has some compatibility issues with many applications, which had to be accommodated. Number of file version is a part of the full file name, and all the file versions are listed when reading a directory. In addition some unaccommodated applications could be puzzled, because the version number in the listing is separated by the semicolon character. Due to this, it is much work to port some applications to this system, because some of them don't expect semicolons in the name, and many of them determine file types from the part of the name after the last dot. In addition, this system is not widely accessible for everyone.

The more popular way of versioning, than a versioning file system, is to use some version control systems, such as CVS, Subversion, git, etc. Use of them usually requires special software client, and so called repository and working directory.

In this diploma thesis, I implemented an versioning file system (called vaves), which should be compatible with existing applications and be as transparent as possible and uses such a framework, that can be mounted even by a common user. So as to make (not only) this system less system dependent, I have also made a framework (called fsfipi) for file system filters (i. e. set of file system manipulating functions, that implement file system calls, which perform some action and call file system calls of the next filter), an input filter for fuse (linux module for file systems in userspace) and an output filter for connection to the real local file system.

In the theoretical part, I will mention possible approaches to the implementation of the versioning system, some existing versioning solutions and frameworks, which are candidates to be used for the implementation.

In the analytical part, I will discuss the design issues of my file sys-

tem. I will also specify the demands on the fsfipi framework and the vaves versioning system.

A chapter about the implementation follows. It contains a description of use and interface of the used frameworks and the vaves versioning system itself. In addition, there is some description about how some issues were solved or that some issues are not solved and which way they may be solved in the future.

The next chapter talks about how the system was being tested for quality (i. e. does what it should), and the following chapter about testing the throughput.

Then follow an conclusion with a summary what was managed and what can be done better in the future, and an obligatory list of bibliography and used software.

Chapter 2

Theory

2.1 Possible approaches

I will mention some base design principles of a versioning file system.

2.1.1 How to present the versions to the user

It is desirable to design the versioning file system in such a way, so that existing applications will not be broken and can access even the old versions.

The other issue concerned is usability. Users should be able to do all the things they used to do on an traditional file system as easy as possible and to return to an old version of their document or to delete it or to change its permissions if they know how to do that for regular files.

Compatibility issues include:

- An application may require, that if it create some file, it will find the file with the same name in the directory listing (what they get using `readdir()`. (This is not the case of OpenVMS).
- Applications may guess file type from the file name extension (the part of the file name after the last dot). A version number after that may break them.
- The file system should not produce exotic characters in file names, which may break some applications, that are not strictly fool-proof.

Human readability and usability issues include:

- An user should recognise the original file name in the name of a version.
- Versioning should not break alphabet order of file names.
- The names should not get too long and should not include bunch of numbers.

- The directory listing should not be flooded with many entries.

These issues should be taken in account when suggesting the versioning file system.

2.1.2 Where to implement version system

The versioning system could be one of the following:

- It could be a file system implementation inside operating system kernel or some kernel module, and implement complete file system (either fork of existing file system, or completely new file system). For instance, in the OpenVMS ([1],[2]) versioning is supported by its native file system.

  This file system could be very fast, but users may miss some features of other systems. If it were a fork of an existing file system, there would be a problem of synchronisation with new releases of the original file system.

- Add a module to some file system, which support this (i. e. reiser4)

  This could be quite fast, but there would be a dependency on changing API, and on the particular file system.

- It could be in the operating system (kernel or kernel module), but implement only the versioning and use actual operations of another file system as storage.

  This would be slower, but does not have disadvantages of above.

- An userspace file system, mounted using an operating system kernel module for mounting userspace or network file systems (i. e. Coda, FUSE, LUFS.). It can use either system calls or library routines or external programs, such as some version control system for the actual data storing.

  This would be slower, it would be less dependent on the OS kernel and in some cases, it can run without root privileges (which are always a security risk for such a complicated work). Once implemented in the userspace, the file system will be able to serve not only as a storage for the file system managed by the kernel, but as a storage for an ftp server or some other userspace application as well.

- Using a system which substitutes library functions and system calls by preloading a library (i. e. PlasticFS)

  This would be less user friendly, and it is hard to ensure, that all applications accessing same files are using it, and this is not always possible.

- An userspace system with its own interface, like most version control systems do it.

  Not compatible with general applications, but there are more possible features than in the classical file system.

2.1.3 Implementation

The system can either copy the old version of the file on write operations or it can store some information about the changes (delta) and some complete versions. It can also combine both approaches.

The variant with storing the deltas can store either the newest version completely and reverse delta of some older versions, or delta for the newest version against the previous. Because the newest version is most probable to be accessed, it is more convenient to store it completely.

Generating the deltas can be done either by doing comparison between the two complete versions, like userspace tools do (diff, xdelta, ...) and possibly use them,

Copy on write approach is more convenient when using small files, because of not much more space and fast operations. For small changes on big files, delta recording is more convenient, especially when constructing in the time of write using information, which file system calls were applied.

2.2 Existing versioning solutions

2.2.1 OpenVMS

Non-free, commercial operating system. It was developed by the *Digital Equipment Corporation (DEC)* in the 70's, for the VAX (this means Virtual Address Extensions) family of processors (an extension of PDP-11). This project was formerly called starlet (the VAX-11/780 processor was called star), than VAX-11/VMS, then VAX/VMS. In 1991 it was renamed to OpenVMS and was ported to Alpha processors and introduced partial support of POSIX standard and UNIX compatibility. In 2001 it was ported to Intel Itanium.

Its native file system called ODS (Files–11 On-Disk Structure) has many features. For example ACL (Access Control Lists), record oriented access (files can be accessed by records, these can be indexed, etc.; this is called Record Management System), file system and files can be split on multiple disks, ..., and supports versioning.

The name contains the number 11 not because it has 11 so called system files (which contain system information like bitmap of used blocks), since they are 11 since ODS-2, but because it was used on PDP-11 and then VAX-11. In a simpler form it was also used in the RXS-11 real-time operating system.

During the time, there have been made 3 different versions of this file system: ODS-1, ODS-2 and ODS-5. The summary of differences there is in [1]. ODS-3 and ODS-4 are Files–11 support for CD-ROMS.

ODS-1 is old and therefore only supported on VAX. It supported file names at most 9 characters long plus 3 characters long file-type extension. Name could contain only uppercase (but the access is case-insensitive) alphanumeric characters (only English alphabet was supported, of course). ODS-1 can't be shared across a cluster, does not support quotas and journalling.

ODS-2 has most the ODS features, but there are still restrictions on file names. Those can be 39 characters long and their file-type extension can be 39 characters long. A name can contain uppercase alphanumerics and the dollar sign, hyphen and the underscore. So that a file name such as xy.tar.gz was still not allowed. It supported both the original DEC platforms, Alpha and VAX.

The most visible difference between ODS-2 and ODS-5 is that ODS-5 has less restrictions on the file names. All ISO Latin-1 characters or all Unicode characters can be used. A name can consist up to 238 bytes (including the dot). This is either 238 ISO Latin-1 characters or 119 Unicode characters (probably only characters less than U+10000 can be used). ODS-5 specific features cannot be used in the VAX version of the OpenVMS. ODS-5 also supports Itanium.

The full file path in VMS looks like:

```
NODE"user password"::device:[directory.subdir]filename.type;ver
```

The file version can also be separated by the dot instead of the semicolon. Of course, not all the parts are mandatory. Only the file name and type are mandatory. When no version is specified, the newest is used. Versions are numbered from 1 to 32767. Version 0 is a reference to the newest version of document, version -1 to the previous, etc. An old version can be either removed manually, or there can be a version limit, and then only that number of the newest versions is kept. This version limit can be set to a particular file with command

```
SET FILE /VERSION\_LIMIT=n
```

or can be set to a directory with command

```
SET DIRECTORY /VERSION\_LIMIT=n
```

which means that all the new files in that directory will have set this version limit, unless explicitly changed. All versions seem to be stored completely. When there is an concurrent write (for example, subprocess A opens a file for write or appending, is writing, subprocess B opens the file for writing, is writing, B closes, A closes), then the newest version is that created by the process having opened it as last (in this case B). Two processes cannot append to the same file concurrently (file is locked).

From the practical point of view, disadvantages for common users (not server administrators) are: high cost, doesn't run on cheap hardware, not many end-user applications. The default listing of files is unfamiliar with non-VMS applications, especially when accessing such system through ssh or ftp from other systems: those applications are not able to guess file type from the file extension and when an user searches wanted file, sees garbage of the old versions. If an application running on VMS wants to list the filenames without versions, it can use DCL command procedure like this (an equivalent to the unix `ls` command):

```
$ start:
$   a=f$search("*.*")
$   if a.eqs. "" then exit
$   b=f$parse(a,,,"name")+f$parse(a,,,"type")
$   write sys\$output b
$   goto start
```

An application can also use some library functions for listing filenames without version numbers or have their own. The Mozilla web browser, for example, displays only the filenames in its file dialogs, but the user can't select non-recent version of some file there.

Advantages

- automatic removal of the old versions,
- consistent concurrent access,
- both the file name access and the access to a particular version are possible.

Disadvantages

- OpenVMS itself is not free, and the license fee is expensive,
- usable by all platform applications, but adapting a new one is hard,
- directory listing is flooded,
- few people are familiar with OpenVMS.

Information about OpenVMS can be found on [1], [2].

2.2.2 FreeVMS

In the time of writing this section, actual version was 0.1.1. This project is trying, among others, to implement an operating system under the GNU GPL licence according to the specification of VMS, and to implement various features to other operating systems. ODS-2 code, however, is not under the

GPL. It is derived from OpenVMS code and includes comment which have to be kept. This includes:

```
This is part of ODS2 written by Paul Nankervis,
email address:  Paulnank@au1.ibm.com

ODS2 is distributed freely for all members of the
VMS community to use. However all derived works
must maintain comments in their source to acknowledge
the contibution of the original author.
```

Their implementation of ODS-2 is read-write but the write still may be buggy, their implementation of ODS-5 file system is unfortunately still read-only. FreeVMS currently supports intel x86 compatible hardware.

It would be hard work to use some ODS-2 code in a system other than VMS, and it is a legal issue combining it with a code under the terms of GNU GPL inside one product. In addition to that, the code is not easy to understand.

(Dis)advantages summary

- most properties like OpenVMS,
- freeVMS is not so much usable yet,
- software is quite free, but distributed works must keep comments in their source code to acknoledge the contribution of the oroginal author, which may cause license issues combining this code with a code under the GNU GPL or other licenses.

You can find FreeVMS and information about it on

- `http://www.free-vms.org/` or
- `ftp://ftp.nvg.ntnu.no/pub/vms/freevms`

2.2.3 CopyFS

This is a virtual file system on top of the FUSE, probably under the GNU GPL, but there is no text of the license. The licensing is only mentioned in the file *interface.c*:

```
/*
 * cpyfs - copy on write filesystem
 * Copyright (C) 2004 Nicolas Vigier <boklm@mars-attacks.org>
 *                    Thomas Joubert <widan@net-42.eu.org>
 * This program can be distributed under the terms of the GNU GPL.
 * See the file COPYING.
 */
```

But there is no file named COPYING in the tarball. This comment is probably kept from the example file system of FUSE, which is copyfs very strong modification of, and therefore distributing under another licensing terms would be probably illegal anyway.

Authors of CopyFS are Nicolas Vigier and Thomas Joubert. Current version of CopyFS is 1.0 and it does not seem being developed any more.

Using copyfs

You can mount it with a command

```
fmount /version/directory /mount/directory
```

or by directly launching fs daemon by

```
RCS_VERSION_PATH=/version/directory/ fcopyfs-daemon /mount/directory -d
```

where */version/directory* is a directory, where the data of all versions and metadata are actually stored, while */mount/directory* is a directory where the user is to work and the contents of which is versioned. The version directory does not have any special format, only a file called *metadata* should be created, which is done by the `fmount` script.

In the mount directory, only actual versions are normally visible. One can list versions (besides looking to the version directory), with the `fversion` perl script. It either lists all versions of the file and marks an active version, prints the active version number, *locks* some version (changes which is active) or release a lock (active is the latest). It is also able to *tag* files. This means, that command

```
fversion -t tagfile directory
```

saves active version numbers for all files in the directory to the tagfile, one file per line, the version number is separated by the pipe character. Such state can be restored using that tagfile with the `fversion` script.

Version numbers are in the form *major.minor*, where the *major* increases when the file data is modified and the *minor* is increased when the ownership or permissions changes. Extended attributes are not specially solved, so that they are reset in each new version. And particular attribute can be read, but attribute listing does not work. Major number starts at 1, minor at 0.

Different versions can be files of different types, one can be a regular file, the other a symbolic link, another a directory, a block special device, a named pipe, and so on, when one file is removed and then another created. Only regular files make new version when they are written to. When a file is removed, all old versions are kept, but a line containing zeroes is appended to the stored metadata. This line is overwritten, when a new file with that name is created, so that it is not possible to find out, that there was a period when no such file existed. When the file is removed, the metadata is not accessible from the mounted directory.

There is a security issue, that when some file is removed, and then a new with the same name is created, its permissions are set not using umask, but they are taken from the previous version (for instance if you have standard umask for example 022, make a directory, remove it and create a file with the same name, it will have execute permissions).

Implementation

The fversion script does not look to the version directory, but it reads and writes extended attributes `rcs.locked_version` and `rcs.metadata_dump`, so that it is a clean solution not relying on the implementation. The format of `rcs.metadata_dump` is

```
majorversion:minorversion:mode:owneruid:ownergid:filesize:mtime
```

where `mode` is in decimal and includes bits describing a file type, and `mtime` is an integer.

All versions are stored completely, and currently there is no way to remove the old ones, neither manually nor automatically from the mounted file system (they can be, of course, removed from the version directory).

The contents of versions of files is stored on the version directory in files named

```
12345678.filename
```

where `12345678` is the major version number (here always aligned to 8 digits).

The metadata of the files is stored in files named

```
metadata.filename
```

the metadata of the root file system directory is in a file

```
metadata
```

The version locks are stored in a file called

```
dfl-meta.filename
```

For files in subdirectories, these files are in appropriate

```
12345678.dirname
```

directories. The metadata file contains one version per line in the format:

```
majorversion:minorversion:octalmode:owneruid:ownergid:versionfilename
```

where `versionfilename` contains the name like *12345678.filename*, `octalmode` is permission mode in standard chmod format (unlike the extended attribute *rcs.metadata_dump*).

The `dfl-meta.filename` contains only the version number in the form `major.minor`

Limitations

- Creating of hard links is not permitted.
- Because copyfs uses the old fuse interface (access to files through a name and not a file descriptor), concurrent access to the files is inconsistent (when the process A opens and writes to a file, then the process B opens, writes, closes, the A writes and closes, the contents is not what the B produced - like standard Unix does, but a mix of both outputs).
- The bug, that the permissions of a removed and recreated file are taken not from umask, but from the previous version.

Advantages summary

- human readable data structures,
- archiving directories and special files,
- daemon in userspace,
- directory listing not flooded with version files.

Disadvantages summary

- special command for access to the old versions (fversion) needed,
- does not remove old versions,
- inconsistent concurrent access.

The authors of CopyFS are Nicolas Vigier and Thomas Joubert. CopyFS can be found on `http://invaders.mars-attacks.org/~boklm/copyfs/`

2.2.4 Version control systems

Userspace programs, primarily used by program developers, their main function is making and managing differences between versions (mostly of source code and documentation files consisting of plain text) and synchronization among developers. User *copies* (*checks out*) global *repository* to the local *working copy* (version control system downloads the files and saves their checksums or only their date of latest modification). The user makes his modifications and when finished, *merges* (*checks in*) the changed files back to the repository. This is called copy-and-merge.

Files changed in the repository by another user after the *checkout* can be downloaded using an *update* operation.

These systems are able to view differences between any two versions of the file and store them all.

There are two kinds of versioning systems, centralized (like CVS or Subversion) or distributed. Some developers of large projects do not like cvs and even subversion because it is not possible there, that some users have the permissions to only some parts of the project, so it limits distributed development. And all the traffic goes to one server.

Thus distributed versioning systems like the git are developed, where the project can have more branches, each of which has its own maintainer and most developers have write access to only some parts of the project. The branch can have subbranches. Each can branch have its own repository where the developers can play with the project and when done then can merge it to another repository. Detailed description of such systems is however outside of the scope of this document, because I want to implement a file system rather than a development tool.

Concurrent Versioning system (CVS)

The most known and spread version control system under the GNU General Public License. But it is not the oldest. Its most common predecessors are RCS (Revision Control System, does not have client/server architecture) and SCCS (Source Code Control System, was part of Unix and it is not free).

In the directory with the working copy and in each subdirectories there is a subdirectory called `CVS`, which contain three files: `Entries`, `Repository` and `Root`. In the `Entries` file there is a list of subdirectories and files belonging to the repository. Entries of the files contain even an information about the version (last synchronized with the repository) and the date of the last modification (on the local disk, so that it can be the date of the check out). There may even be local only files in the directory with the working copy (the files not having an entry in the Entries file). New file are added to the repository using the cvs *add* command. When checking in, modified files are recognised because their modification date on disk is newer that in the Entries file.

- no known way to use as a file system,
- can not well operate with file metadata,
- non-atomic operations,

More information about cvs can be found on cvs project homepage: `http://www.nongnu.org/cvs/`

Subversion

Version control system with some advantages over CVS, for example it can remember versions of a file moved or renamed, can store some metadata and so on.

It can be used with the Apache http server to provide access via webdav protocol. Webdav extends the http protocol by adding methods for file manipulation (besides the `PUT` method), as `MOVE` and `COPY`, and for directory (there called collection) manipulation (`MKCOL` creates directory, ...), and metadata (properties) assigned to files (like `PROPFIND` and `PROPPATCH`), and for locking.

Then, there are deltaV extensions to the protocol, defining versioning. There are several methods there:

- per-resource versioning (client places a resource under version control, and every time being modified, its version (state) increases,
- server-side copy model (user creates working copy on the server, performs checkout, upload(put) new version and checkin),
- client-side copy model (user has local working copy and does checkin and checkout,
- autoversioning (user has common dav client having no idea about versions, and when calls put, the server silently performs checkout, put and checkin in server-side model.

Apache module for subversion (mod_dav_svn) implements subset of the deltaV extensions and the most usable use of it is through autoversioning and common webdav client, and it behaves as common network file system. The manipulation with old versions, however, needs regular subversion client.

Mounting in Linux using the Davfs (which uses the Coda file system module) as client has the problem, that mod_dav_svn does not implement locks, while davfs Linux file system uses them. We can avoid this problem through use mod_dav_lock apache module, which implements locks using its own database. But when use concurrently with regular subversion client, it would not use that database.

The licence allows redistribution and use in source and binary forms, with or without modification, but with an so called advertisement clause, so it is not compatible with the GNU general public license. Apache has its own license, Apache Licence, which grants rights to reproduce in source or binary form or make derivate works and distribute them in source or binary form, when keeping all copyright, trademark, ... stuff. Some Apache modules are under some another licences.

Dis/Advantages summary

- Very complicated to get it work as a file system (getting locks to function),
- many programs involved (subversion, apache, coda kernel module, davfs),
- access to the old versions through subversion clients (not through the file system),

More information about subversion can be found on subversion homepage: `http://subversion.tigris.org/`

2.2.5 XDelta

Formerly, this was something like the diff tool for binary files, it takes two versions of file, and produces so called delta, or take an old file and the delta and create a new version. It can only patch the unmodified old file; it recognises a file by a checksum. The intention was to use it in version control systems, in particular in the PRCS, which the developer of xdelta contributes to. The Xdelta will be used in version 2 of the PRCS. The author of Xdelta is Josh MacDonald.

There are 3 incompatible versions of xdelta. These versions do not have much common in the code itself.

The Xdelta 1.x was just like the *diff* utility working even with binary files with the ability to uncompress gzipped source files and compress the output also with zlib. It was released under the GNU GPL licence.

The Xdelta2 was made as an application-level file system based on the Berkeley database, which uses transactions. The access is network transparent, the storage uses the same functionality as the version 1. There is no special command specialized on extracting the delta between two files, like in the version 1. The commands are analogical to version control systems. This version was released under the BSD-style license, a few files remained under the GPL.

The xdelta3 returned to make an interface for creating deltas. The format was changed to a VCDIFF standard (RFC3284) [3]. The delta is represented as a sequence of commands ADD and COPY. There can be COPYes from different sources, in this case the source (the old version) and the target (the part of the new version already built). This is also used to make a delta of a single file, that means, that besides computing differences between two files, it has a command to compress a single file (only ADD and COPY from target).

The author claims it has about 20% worse compression ratio than gzip. Thanks to this kind of compression, it drops support for zlib. There is an

intent to support more then one source in future versions of xdelta and the PRCS. Xdelta3 is under the GPL again.

Delta file format

Common property of xdelta1 and xdelta3 delta formats is, that both have headers containing format version, file names, lengths of all parts of delta, etc., and delta, where commands and inserted data are stored in separate streams. xdelta1 can print information from the header of delta in human readable form, xdelta3 can print even all ADD and COPY commands with their sizes and addresses. Xdelta1 also stores complete md5 checksums of the source and the target, Xdelta3 only adler32 checksum of the target.

Xdelta3 delta is divided into 3 streams: data, instruction and address. Each instruction in the instruction stream consists of 1 byte instruction code and one (if the size is less than 128) or more bytes containing the number of bytes to be copied or inserted. In the data stream there is the data to insert, the sections are not separated (the pointer is always incremented by the size of operation). The control stream contains offsets from where to copy.

The delta information printed in human readable form with command

```
xdelta3 printdelta p1
```

where the file *p1* was made by command

```
xdelta3 -s a c p1
```

looks like this:

```
VCDIFF version:               0
VCDIFF header size:           11
VCDIFF header indicator:      VCD_APPHEADER
VCDIFF secondary compressor:  none
VCDIFF application header:    c//a/
XDELTA filename (output):     c
XDELTA filename (source):     a
VCDIFF window number:         0
VCDIFF window indicator:      VCD_SOURCE VCD_ADLER32
VCDIFF adler32 checksum:      C2FC4A0E
VCDIFF copy window length:    160
VCDIFF copy window offset:    0
VCDIFF delta encoding length: 44
VCDIFF target window length:  265
VCDIFF data section length:   25
VCDIFF inst section length:   7
VCDIFF addr section length:   2
  Offset Code Type1 Size1 @Addr1 + Type2 Size2 @Addr2
  000000 019  CPY_0 160 @0
  000160 001  ADD    25
```

```
  000185 035  CPY_1  80 @329
SIZE=265  TGTLEN=265
```

where CPY_0 means copy from source and CPY_1 from target.

Implementation

The algorithm has linear time complexity in average. It calculates some checksums for blocks of the sources (source file and target), and puts these checksums into a hash table. If some part is same as something already processed, it has the same checksum and finds the appropriate record in the hash table, where the source and the position are stored, and if it is really the same, COPY instruction will be generated. Algorithm details are beyond the scope of this documents. For details see xdelta3.c file in the xdelta3 source code.

Summary

- differences for binary files,
- linear time complexity for making the differences,
- very small and lightweight.

More information about xdelta is available on:

- Xdelta homepage
  `http://xdelta.org/`,
- An old author's page about xdelta containing some useful informaton
  `http://www.xcf.berkeley.edu/~jmacd/xdelta.html`,
- PRCS project site
  `http://prcs.sourceforge.net/`,
- page of the author
  `http://www.xcf.berkeley.edu/~jmacd/`,
- Presentation of copyfs:
  `http://www.cs.berkeley.edu/~jmacd/stanford/sld001.htm`

2.3 Virtual file system frameworks

Making the filesystem in userspace has the advantage, that the code can use library functions and call other programs, the code can be used not only as a file system of the operating system but in various servers like an ftp server, webdav server, etc, can be theoretically reused on more operating systems, and it does not need to be changed when the operating system kernel changes its internal interface or policy, and its development does not need to be synchronised with the development cycle of the kernel. In addition, any complex and unnecessary code in the operating system kernel is always a security risk because of possible bugs.

So there is a small overview of some frameworks those can be used to implement an userspace file system:

2.3.1 LUFS

LUFS is a virtual file system framework for Linux, consisting of a kernel module and an userspace daemon. The kernel module delegates the filesystem calls to the daemon. The communication is done through UNIX domain sockets. LUFS file system can be mounted by regular users with help of a suid binary. Read and write calls use file name as a parameter and open files more often than necessary.

LUFS is no more developed since 2003, and most file systems using it have switched to FUSE.

More information about LUFS there is on:
`http://lufs.sourceforge.net/`

2.3.2 FUSE

FUSE means Filesystem in userspace. It is a virtual file system framework for Linux, consisting of kernel module, userspace library and `fusermount` utility. A filesystem is a daemon implementing fuse filesystem calls and passing them to a function from fuse userspace library, which mounts the filesystem and runs the main event loop.

The daemon can run as a user (only has to be in the group fuse), but in that case other users cannot access that file system due to security reasons (for instance so as the user could break some scripts by making a directory structure with cycles)

The called fuse library forks the process and the child executes the `fusermount` suid binary. Fusermount mounts the filesystem and opens a special device `fuse`, through which the communication with the kernel is done. The handle of opened device is transferred to the parent through a UNIX domain socket (created before the fork). Then the parent process dispatches commands sent through the device in an event loop.

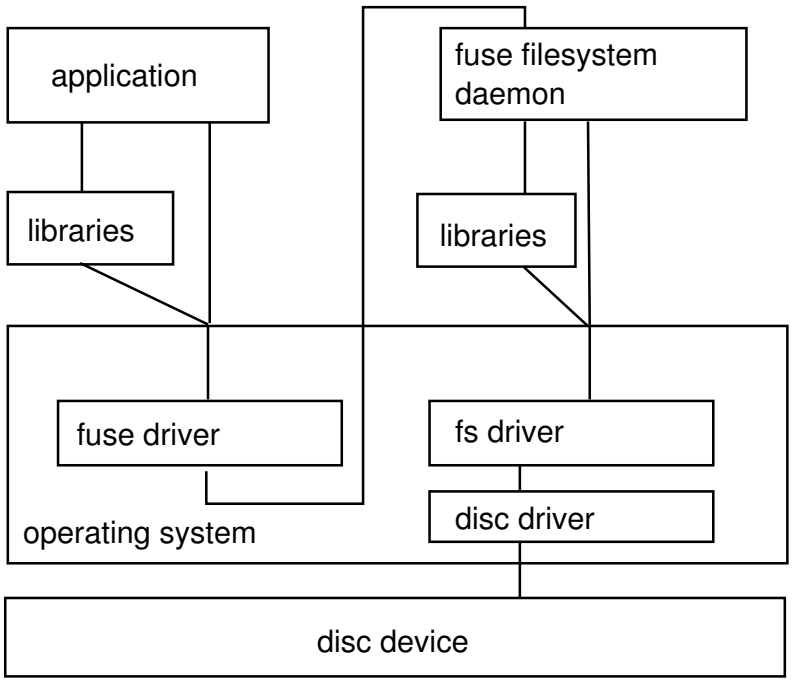

Figure 2.1: Command flow on FUSE filesystem

Read and write calls take file name as an argument and since the version 2.0 also a structure which can contain a file handle and which has to be maintained by the filesystem daemon.

There are many file systems under FUSE. Many of them provide a function, that they modify the behavior of some existing filesystem directory subtree (for example turn filenames to lowercase). If one wants to use more such modifications, (s)he must mount the first fuse filesystem and let it use the original path, then the second fuse filesystem and let it use the path, where the first filesystem is mounted and so on if there are more such filesystems. Then for every filesystem operation, every system call is called at three nested levels. It would be a good idea to support pipe of filters without the need for mounting more file systems.

Usage of a fuse filesystem

Users usually (if the daemon passes the arguments to the fuse library) mount a fuse filesystem using the following syntax:

```
path/to/fsdaemon mount/point [ <fuse switches> ] [ -o <fs_options> ]
```

where *fuse switches* and fuse specific filesystem options set things such as allowing multithreading, handling of inode numbers, handling of removing

files (rename if open), debugging, etc.

Unmounting is done with the command:

```
fusermount -u mount/point
```

The superuser can do both using mount:

```
mount -t fuse /path/to/daemon mount/point [options]
```

or

```
mount -t fuse none mount/point -o fs=/path/to/daemon[,opt=val...]
```

Fuse and the Linux kernel

About November 15th 2004, there were discussion, whether to merge FUSE into the Linux kernel. Linus Torvalds and other major kernel developers disliked this idea, saying, that it is too complicated and messy, it should be simple and do only general page cache reading. He also criticized the presence of exotic features said not to work fine. Linus also said userspace filesystems not to be the right way.

After some time and fixing some problems (for example building on 64 bit architectures) fuse was merged into Andrew Morton's testing kernel tree 2.6.11-rc1-mm in January 2005.

Because of FUSE popularity, and because no one suggested anything better, FUSE was included into the Linux kernel 2.6.14 in September 2005.

Fuse summary

- easy to use interface,
- a filesystem 'LUFS bridge', binary compatible with existing LUFS file systems,
- the 'in' and 'out' interface are not the same - building a pipe of filters is not easy,
- is not a standard part of unix system.

More information about fuse is in implementation part and on [4].

2.3.3 PlasticFS

The plastic file system changes, how the filesystem looks like for applications. It implements replacements for filesystem calls and library functions in a library. The program is forced to use this library using the LD_PRELOAD environment variable. It needs no support in the kernel, but is dependent on the implementation of the C library.

The advantage of PlasticFS is that these file systems are implemented as filters and can be piped from one to the next.

The author of PlasticFS is Peter Miller. The project does not seem to be developed more since 2004. More information about PlasticFS there is on PlasticFS homepage: `http://plasticfs.sourceforge.net/`

Chapter 3

Analysis

3.1 Discussion of the design

3.1.1 How to implement the versioning system

The main decision criteria are speed, configurability and dependency on any particular technology and interface.

I preferred the last two over the first one, because most time critical applications are not to use versioning system.

Making the versioning filesystem as a filesystem as a part (module) of some unix like operating system or even a part of some existing filesystem would have two disadvantages: First, only users of such system would be able to use it, and second, the most spread ones often change their program interface. Another reason is that too complex things do not belong into an operating system kernel, because of risking its stability, increasing its size and the need for coordination. If the filesystem core is in userspace and well done, then it will be possible to also make interfaces to other userspace applications, for example ftp server or any other server which mediates access to a directory tree with files. Also some operating systems (for example the GNU/Hurd) implement all filesystems in userspace.

The configurability is important because of compatibility with different applications, which may have different requests and because different users may prefer different behavior. For example the names of the file versions may be required to have the same alphabetical order as the original file names, to have the same extension (the part of the name after the last dot) as their original file name, or to be as short as possible. Another useful configuration option is whether to keep versions of a file when the file is removed. When all of the operations are made by humans, it is good to save the old versions for the case, that the user removed the file by accident (for example klicked on the neighbouring icon). If there is some script operating over that directory, in addition to the user, this script may create a directory with files and remove them and will fail when trying to remove that temporary directory,

so may fail it there are some old versions inside.

There is already the FUSE framework in the GNU/Linux, described in the previous chapter. Because it does not support pipelines I decided to make my own userspace file system framework, which will run primarily under fuse, but there would even be the possibility to implement some other interfaces, eiher to some other frameworks or even operating systems or a userspace application could use it instead of the file system of the operating system.

Versioning system will be written in the C language, because it can be compiled on high number of systems, compiled code is very quick, and it will cooperate mostly with tools and systems with C interface.

3.2 The Fsfipi framework

So as to allow more interfaces to the version filesystem, I decided to make a simple filesystem framework, called FSFIPI (Filesystem filter pipeline). Its purpose is to change the behavior of a filesystem by a pipeline of filters, each doing one simple modification.

Filters have the same interface from the higher level and to the lower level (by lower I mean closer to the physical data storage). This interface is made by a structure of pointers to functions manipulating the filesystem and data structures, and a register function, which constructs filter instance and initialize it. Only the filter in the lowest level uses the real filesystem functions instead of calling methods of lower level, and the filter in the highest level has an interface of the system which uses it (I have a fuse interface, there would be possible ftp-server interface, nfs-server, ...).

The fsfipi itself should only load these modules with their parameters from command line, and call destructors on them. So as to be as modularized as possible, the modules are shared libraries, loaded at run-time.

3.2.1 Standard fsfipi filters

Fsfipi should be made as modularized as possible. All the basic functionality, that might be used be most filesystems, should be implemented by standalone filters.

There will be one highest level filter, which will run FUSE filesystem daemon. It will be called *fuse_interface*, Its filesystem function (passed to the FUSE library, and called by FUSE), should only call corresponding functions of the lower fsfipi filter, and transform its parameters as appropriate.

The standard lowest level filter will be called *localfs* (this name is taken over from plasticfs). It shoud perform requested filesystem operations on the local filesystem.

Because there may be some filters modifying file names, there should be a base for these filters, called *rename*. It should call a function to transforming

the file names, and pass the request to the lower level. The actual renaming function must be defined in a separate file and be linked with, in order to reuse the code. In fact it will be deriving a subclass from an abstract class.

Usually, an userspace filesystem doesn't modify behavior of the whole local filesystem, but some directory subtree, for example the home directory. For this purpose, there will be a *subtree* filter, which adds some directory (or some other) prefix at the beginning of the file paths used in the operations. This filter will be made using the rename module.

3.3 Vaves, a fsfipi module for making versioned filesystem

The versioning system core itself is called Vaves. It is made as a module for the Fsfipi framework. Since Fsfipi allows more front-ends, vaves will be able to run under more systems as well.

3.3.1 Types of changes of files

Generally, there are three types of file changes. First, any file (even the special one) can be removed, and a new one with the same name can be created. Second, the regular file (or directory) can be written to or truncated, so that the contents changes. Third, the file metadata can be changed (owner, group, permissions, extended attributes, etc.), without modifying the data.

I follow here the idea from copyfs, major and minor versions. When the contents changes (or the file is removed and created again), the major version is increased, when the change concerns only the metadata, the minor version is increased. The data is stored for each major version, the metadata for each minor version.

There is a special case, when the file is renamed. Then the system can track the contents, like for instance subversion does, or it can track the name, like for instance copyfs. In the first case, it is a metadata change, in the second case, it is file removal and creation.

Vaves system tracks the file names, so it can (if remembering removed versions is set on) remember, that, for instance, a regular file X, was removed and replaced with a symbolic link to a file Y.

3.3.2 Information to be stored

I choose the method of storing whole file versions and copy their data on change. Its advantages are easy implementation, quick access to the versions, no need to delay write operations by computing differences with the cost of higher usage of disk space.

Another design decision, which influences the stored information is that the names of the versions will be fully configurable, so as to allow them to be compatible with different possible requests.

The third factor is that the permissions should be stored separately for each version, because when the file is removed and another with the same name created, the two may have another owner and permissions. In addition, for archival purposes it will store even the changes of permissions of the file with unmodified data, as the minor versions.

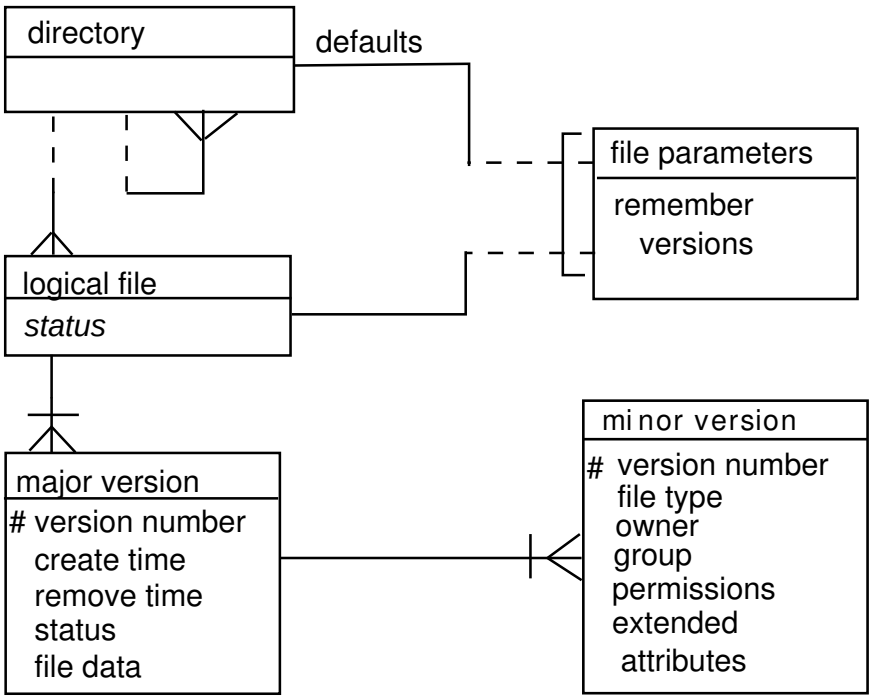

Figure 3.1: Data model of the Vaves

So, the system should store:

1. the data of each major version of a file,
2. the metadata for each minor version,
3. the range (the maximum) of used major version numbers of each user file,
4. the range (the maximum) of used minor version numbers of each major version,
5. possibly the date of creation and 'removal' (since when the version is not actual); this could be used for example by userspace scripts which clean up old and unused file versions,

6. some global parameters (name of the subdirectories, in which the user can find file versions and some accessible metadata), a pattern for creating the names of file versions, etc. (no need not to be stored, it can be passed to the filesystem daemon),

7. if the name of a file version is fully configurable, either one reverse pattern for extracting the file name from the version name (generally not possible), or the logical file name (i. e. the file which the version belongs to) for each user visible version.

8. some information about individual file policy, in this case, for instance, how many old version to keep. This is stored once for each user file, and in each directory there are defaults for the newly created files.

The information stored for each unit will be stored in a separate file, the data of more units (for instance the data of more minor versions) will not be merged.

All this data is visible to the user with appropriate permissions, and can be manipulated by userspace scripts.

Because the filesystem can contain not only regular files, but directories, symbolic links, sockets, etc, as well, the metadata of files is stored as an empty file with the same filesystem metadata, as the user file (permissions, owner, group, extended attributes, etc.). If it was stored in a regular file, the attributes of that file would have to be read too, before opening the file, and parsing the information would be necessary, but it would support even the features not supported by underlying filesystem on the other side. I call these files 'fake inodes'.

The minor versions are not visible directly, but the user or scripts may see fake inodes, so that they can revert metadata changes.

The pattern for creating the name of the file versions is configurable, and there is no general algorithm to determine the name of the user (logical) file the version belongs to, from the name of the version (file). So as to determine that, vaves stores the user (logical) file name and the version number in a regular file with the same name, as the name under which user accesses the version. Because only the major versions are visible to the user this way, other vaves metadata stored for the major versions are stored there as well, in particular the range of minor versions (of that major version), the creation and 'removal' time. The same metadata can be accessed by the vaves user through the metadata directory. The file data of the version is stored in a separate file or even a special file or directory, depending of the type of the logical file.

For each logical file there also exist an contents file, which contains the range of major versions, the status of the file (if it is 'removed' or exists), and the per-file settings (actually only how many old versions to keep).

In each regular directory (not the special one for the metadata), there exists one file of default settings (the number of old versions to keep) for new files in the directory.

3.3.3 The directory structure

The versions and the metadata files there are in subdirectories, so as not to flood the listing of a directory containing regular files. The name of both the directory for metadata files (which are supposed to be used only by experienced users or scripts, by default .VAVES) and the directory with versions (where users are supposed to find old versions, by default VAVES) is configurable via a command-line parameter.

But Vaves gives even the possibility to store all these files in the same directory. For this case, both types of files can be recognised by prefixes, which are configurable as well (by default '.vaves_' for the metadata and ' ' for the versions.

So as not to break alphabetical sorting and applications using the file 'extension' to determine file type, the default format of version names contains the name of the file, the version number and the 'extension'. The extension is taken as the part of the name from the last dot, so that if there is no dot in the name, the extension is the same as the whole file name.

3.3.4 Removing version, files, and directories

Even though all the metadata is accessible to the user via files, the user need not to be forced to remove all the metadata files manually.

When the user removes a version file, all the metadata files for that major version are deleted as well. Then the range of stored major versions is updated, and when empty, remaining files concerning that file are removed.

Removing the files is more complicated. When using by humans, sometimes a file is removed accidentally (for instance by clicking on an neighbouring icon). In this case it is convenient, when the versioning system keeps storing the removed file, so that it can be easily restored even by relatively inexperienced user.

However, if there are also some scripts, which create temporary directory and some files and then remove it all, they will fail removing the directory, because of not being empty. For this purpose, there is a remove-all-versions mode, in which deleting a file removes, not only the reference, but all the versions.

When the user removes the directory, the subdirectory for version and the subdirectory for the metadata are silently removed as well.

Chapter 4

Implementation

My versioning file system vaves uses the fsfipi framework, which is also my work, that is why I will discuss it as well. Since it uses FUSE as the primary interface, I will describe FUSE as well.

4.1 Fuse

4.1.1 User's point of view

Particular file system is implemented in some filesystem daemon. Any user in the fuse group can mount a filesystem by running this executable with mandatory parameter specifying mount point, optional fuse parameters (debugging, disabling multithreading, running in foreground, etc.). Parameters for particular filesystem are mostly passed through the environment.

Due to security reasons, there are some limitations for common users:

- The user must have write permissions to mount-point,
- if the mount-point is sticky directory, it must be owned by the user,
- in the default configuration, no other user (including root) can access the filesystem. (Can be overridden by /etc/fuse.conf and an explicit option must be used).

The superuser can use the `mount` command as well, and is not limited by the conditions above.

Settings and options

Options regarding the policy can be set in the file /etc/fuse.conf. Currently there are two:

mount_max = NNN Set the maximum number of FUSE mounts allowed to non-root users. The default is 1000.

user_allow_other Allow non-root users to specify the 'allow_other' or 'allow_root' mount options (see below)

Library options (passed to the daemon, I don't list all the options):

-f run in foreground; useful when running from console,

-s single thread; necessary when the file system does not support threads,

-o mount options separated by commas.

There are several mount options:

allow_other, allow_root Allow access to other users, or to the root respectively. By default, these options are allowed to the root only, but they can be allowed to all user in /etc/fuse.conf (see above). With this option, it is useful to use an option default_permission.

default_permission By default, FUSE doesn't check file access permissions, the filesystem is free to implement it's access policy or leave it to the underlying file access mechanism (e.g. in case of network filesystems). This option enables permission checking, restricting access based on file mode. This option is usually useful together with the 'allow_other' mount option.

kernel_cache This option disables flushing the cache of the file contents on every open(). This should only be enabled on filesystems, where the file data is never changed externally (not through the mounted FUSE filesystem). Thus it is not suitable for network filesystems and other 'intermediate' filesystems. NOTE: if this option is not specified (and neither 'direct_io') data is still cached after the `open()`, so a `read()` system call will not always initiate a read operation.

direct_io disables the kernel page cache.

hard_remove disable the default behavior, where a removed file is only renamed and the actual removal takes place after the `release()` call.

use_ino, readdir_ino Use inode numbers reported by the filesystem always, or when reading directory entries respectively. By default fuse does not use these numbers.

nonempty Allow mounting over non-empty directory. By default, fuse since version 2.3.1 does not allow it. This option is not in older versions of fusermount.

umask=M, uid=N, gid=N Override filesystem permissions (clears bits set in umask), set owner user or group respectively.

4.1.2 Programmer's point of view

The main task of the programmer is to implement some or all filesystem functions. Pointers to these function are filled to a structure of the `fuse_operations` type. Then he only has to pass this structure and fuse arguments to a function `fuse_main`, which is in the fuse library and which does all the job (processing arguments, mounting, communication with the kernel, the main event loop, handling of the `TERM` signal, and so on). Programmers can also do some job themselves and use a more detailed API instead of the `fuse_main` function.

The `fuse_main` function has the following prototype:

```
int fuse_main
  (
    int argc,
    char *argv[],
    const struct fuse_operations *op
  );
```

where `argc`, `argv` are the same as those given to `main()`, and `op` is structure containing the pointers to the functions implementing each operations, defined as a following structure:

```
struct fuse_operations
{
    int (*getattr) (const char *, struct stat *);
    int (*readlink) (const char *, char *, size_t);
    int (*getdir) (const char *, fuse_dirh_t, fuse_dirfil_t);
    int (*mknod) (const char *, mode_t, dev_t);
    int (*mkdir) (const char *, mode_t);
    int (*unlink) (const char *);
    int (*rmdir) (const char *);
    int (*symlink) (const char *, const char *);
    int (*rename) (const char *, const char *);
    int (*link) (const char *, const char *);
    int (*chmod) (const char *, mode_t);
    int (*chown) (const char *, uid_t, gid_t);
    int (*truncate) (const char *, off_t);
    int (*utime) (const char *, struct utimbuf *);
    int (*open) (const char *, struct fuse_file_info *);
    int (*read) (const char *, char *, size_t, off_t,
                 struct fuse_file_info *);
    int (*write) (const char *, const char *, size_t,
                  off_t, struct fuse_file_info *);
    int (*statfs) (const char *, struct statfs *);
    int (*flush) (const char *, struct fuse_file_info *);
    int (*release) (const char *, struct fuse_file_info *);
    int (*fsync) (const char *, int, struct fuse_file_info *);
    int (*setxattr) (const char *, const char *, const char *,
                     size_t, int);
```

```
    int (*getxattr) (const char *, const char *, char *, size_t);
    int (*listxattr) (const char *, char *, size_t);
    int (*removexattr) (const char *, const char *);
};
```

where `fuse_dirfil_t` is defined as:

```
typedef int (*fuse_dirfil_t)
(
  fuse_dirh_t h,
  const char *name,
  int type,
  ino_t ino
);
```

and *fuse_file_info* is defined as:

```
struct fuse_file_info
{
  int flags;/*Open flags*/
  unsigned long fh;/*file handle*/
  int writepage;/**/
};
```

Most of these functions are similar to UNIX system calls, with following differences (stolen from *fuse.h*):

- All methods are optional
- All operations should return the negated error value (−`errno`) on error.
- There is no `create()` operation, `mknod()` will be called for the creation of all non-directory, non-symlink nodes.
- There is a change in FUSE 2.2 in `read()` and `write()`. Up to the version 2.1, they are not passed a file handle, but rather a pathname. Since 2.2, there is a parameter of type pointer to struct `fuse_file_info`, containing flags for `open()` (in 1.x flags were passed directly), file handle `fh` (may be filled in by `open()`, used by `read()`, `write()` and `release()`), and `writepage` field (indicating, if the write was caused by `writepage()`).
- The offset and length of the `read()` and `write()` are passed as the arguments, like the `pread()` and `pwrite()` system calls. (NOTE: `read()` should always return the number of bytes requested, except at end of file)

- `open()` should not return a file handle, but 0 on success. No creation, or truncation flags (`O_CREAT`, `O_EXCL`, `O_TRUNC`) will be passed to `open()`. `open()` should only check if the operation is permitted for the given flags.

- `getattr()` is similar to `stat()` and doesn't need to fill in the following fields: `st_ino`, `st_dev`, `st_blksize`

- `readlink()` should fill the buffer with a null terminated string. The buffer size argument includes the space for the terminating null character. If the link-name is too long to fit in the buffer, it should be truncated. The return value should be 0 for success.

- `getdir()` is the `opendir()`, `readdir()`, ..., `closedir()` sequence in one call. For each directory entry the `filldir` parameter should be called.

- `release()` is called when an open file has: 1) all file descriptors closed 2) all memory mappings unmapped For every `open()` call there will be exactly one `release()` call with the same flags. It is possible to have a file opened more than once, in which case only the last release will mean, that no more reads/writes will happen on the file. The return value of `release()` is ignored.

- `flush()` is called when close() has been called on an open file. NOTE: this does not mean that the file is released (e.g. after fork() an open file will have two references both of which must be closed before the file is released). The `flush()` method may be called more than once for each `open()`. The return value of `flush()` is passed to the `close()` system call.

- `fsync()` has a boolean `datasync` parameter. If this parameter is TRUE then it is an `fdatasync()` operation.

The programmer can use more detailed api instead of `fuse_main`, which consists of several functions, most of which are used in `fuse_main`:

- `fuse_mount()` and `fuse_unmount()`

- `fuse_new()` (initializes data structures)

- `fuse_loop()` and `fuse_loop_mt()` (the main event loop single/multi threaded)

- `fuse_exit()` (exits main event loop)

- `fuse_get_context()` (gets context of current operation – UID, GID and PID of the calling process)

- `fuse_invalidate()` (invalidates cache data of a file)
- `fuse_is_lib_option()` (indicates, if specified mount option is processed by the library or the kernel)

There is also an advanced API, for use, if a programmer would like to change the implementation of fuse commands and/or for debugging purposes.

It is recommended to define `FUSE_USE_VERSION` macro to a value like 22 (version 2.2), 11 (version 1.1) to specify the API version to use.

Implementation

User mode filesystem program calls *fuse_main()*, which parses the arguments and calls *fuse_mount()*. It creates UNIX domain socket and *fork*s the process and the child *exec*s setuid root utility `fusermount` (calling user must be either root or a member of the fuse group). `fusermount` opens the fuse device, with the name */dev/fuse*. (character special device, major 10, minor 229), or */proc/fs/fuse/dev* (in case of an old fuse kernel module), or tries to create such device in temporary directory. Then adds the file descriptor to the fuse device to the mount options and calls the *mount()* system call.

Then the kernel module stores pointer to *struct file* corresponding to that descriptor to the filesystem superblock (memory structure). The struct file is a handle to an open file 'from the kernel side'.

UNIX domain sockets can transfer not only some data, but also an open file descriptor, i.e. not only the integer, but the permission itself. After the `mount()` call, `fusermount` sends the descriptor of the open fuse device to its parent, through the UNIX domain socket, and exits.

If someone tries to access the filesystem and has necessary permissions, the fuse module sends a request through the file handle (`struct file *`), associated with the filesystem superblock.

On the other side, `fuse_main()` enters an event loop, in which it reads commands from the descriptor to the fuse device, and executes them by calling functions in the `fuse_operations` structure. The loop is also checking the `exited` variable, which is set, when a signal like *SIGABRT*, *SIGINTR* is received or when `fuse_exit()` is called, and in such case, exits the loop.

4.2 The Fsfipi framework

So as to allow more interfaces to the version filesystem, I decided to make a simple file system framework, called FSFIPI (Filesystem filter pipeline). Its purpose is to change the behavior of a filesystem by a pipeline of filters, each doing one simple modification.

Since the filesystem daemon is most probably run by scripts, each parameter for individual filter is one command-line parameter of the fsfipi.

Parameters for one filter and for the next filter are separated by '–' parameter. The name of the filter to use is determined by the first parameter after this separator.

4.2.1 How does it work

Filters have the same interface from the higher level and to the lower level (by the lower I mean closer to the physical data storage). This interface is made by a register function, which constructs filter instance and initialize it, and by a structure of pointers to functions manipulating the files and directories and pointers to data structures. The register function is the constructor of data structures of the filter – it allocates all the data structures, initializes them and return the pointer to the main structure with the functions.

Only the filter in the lowest level uses real filesystem functions instead of calling methods of lower level, and the filter in the highest level has an interface of the system which uses it. I have a Fuse interface, there would be also possible to make the highest level with another interface or as an server of some protocol for manipulating files, such as an ftp-server, nfs-server, etc.

The fsfipi itself does only loading these modules with their parameters from command line, and call contructors (the register function) and destructors on them. The first is done with a `fsfipi_load_module` function taking the same arguments as the filter constructor (see below). It determines the file name of the filter binary (shared library) from the first argument, loads this binary, determines the name of the constructor, loads it, runs it and returns its result.

Imagine the following command:

```
fsfipi \
  -- fuse\_interface /path/to/mount "$@" -f -s\
  -- locase\
  -- subtree /path/where/stored \
  -- localfs
```

fsfipi will first load module `localfs` from the library `libfsfipi_localfs.so` (in default configuration on unix-like system), then executes the register function with the name `fsfipi_localfs_register` passing '`localfs`' as the first argument and passes a NULL pointer as the lower level. Then tries to load module `subtree` in the library `libfsfipi_subtree.so` and calls its register function `fsfipi_subtree_register` passing arguments '`subtree`' and '`/path/where/stored`' and passing the structure returned by the previous register function (from `localfs`). Then it tries to load the `locase` module (Such a module does not exist yet, but probably will. Such filters are those Fsfipi was designed for) and `fuse_interface` with its parameters. The register function of `fuse_interface` does not return immediately, but runs a fuse filesystem.

When the user does an operation under the `/path/to/mount` directory, the linux kernel finds that it is mounted on the fuse filesystem, and tells the Fuse kernel module to handle it. The Fuse module sends a command to the Fuse fs daemon (run by `fuse_interface`), and it is handled in the Fuse library. It calls appropriate function defined in `fuse_interface`. This function maps the call to the corresponding fsfipi method, which was passed from the `locase` module. This method in the locase module changes the letter in the path parameter and calls the same function from the `subtree` module, which adds prefix to the path and calls the method from the `localfs` module. The function from the `localfs` module use a standard filesystem call or a library function, which operates on the real local file system. Because the prefix added in the `subtree` module, it accesses only a part of the filesystem.

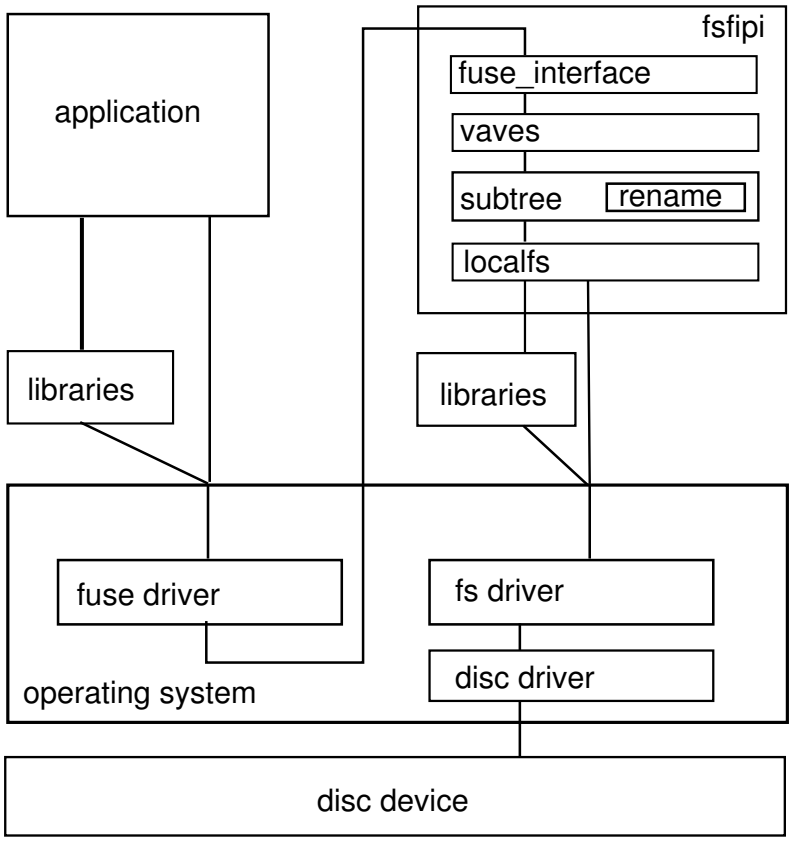

Figure 4.1: Command flow on an Fsfipi filesystem with the Vaves filter

4.2.2 The interface

Every filter contains a constructor function of the following type:

```
typedef
    struct fsfipi_operations *fsfipi_register_func
    (
      struct fsfipi_operations *lower_level,
      struct fsfipi_global *global_data,
      int argc, char *argv[]
    );
```

The parameter `lower_level` is a pointer to the structure of the filter in the lower level, `global_data` is a pointer to the data common to all filters, `argc` and `argv` are parameter count and the filter parameters. The function returns a pointer to a `fsfipi_operations` structure of the filter instance.

The `fsfipi_operations` structure contains the pointers to file system functions. All the member functions take `context` as their first argument, which is a pointer to the `fsfipi_operations` structure. The set of this functions is mostly taken over from fuse. The declaration of this structure is:

```
struct fsfipi_operations {
  void *private_data;
  struct fsfipi_global *global;
  struct fsfipi_operations *next;
  void (*init)
    (struct fsfipi_operations *context, void *data);
  void (*unregister)
    (struct fsfipi_operations *context);
  int (*getattr)
    (struct fsfipi_operations *context,
     const char *path, struct stat *stbuf);
  int (*readlink)
    (struct fsfipi_operations *context,
     const char *path, char *buf, size_t size);
  int (*mknod)
    (struct fsfipi_operations *context,
     const char *path, mode_t mode, dev_t dev);
  int (*mkdir)
    (struct fsfipi_operations *context,
     const char *path, mode_t mode);
  int (*symlink)
    (struct fsfipi_operations *context,
     const char *path, const char *to);
  int (*link)
    (struct fsfipi_operations *context,
     const char *path, const char *to);
  int (*rename)
```

```
  (struct fsfipi_operations *context,
   const char *path, const char *to);
int (*unlink)
  (struct fsfipi_operations *context, const char *path);
int (*rmdir)
  (struct fsfipi_operations *context, const char *path);
int (*chmod)
  (struct fsfipi_operations *context,
   const char *path, mode_t mode);
int (*chown)
  (struct fsfipi_operations *context,
   const char *path, uid_t uid, gid_t gid);
int (*utime)
  (struct fsfipi_operations *context,
   const char *path, struct utimbuf *buf);
int (*open)
  (struct fsfipi_operations *context,
   const char *path, struct fsfipi_file_info *handle);
int (*truncate)
  (struct fsfipi_operations *context,
   struct fsfipi_file_info *handle, off_t offset);
int (*ntruncate)
  (struct fsfipi_operations *context,
   const char *path, off_t offset);
ssize_t (*read)
  (struct fsfipi_operations *context,
   struct fsfipi_file_info *handle,
   char *buf, size_t size, off_t offset);
ssize_t (*write)
  (struct fsfipi_operations *context,
   struct fsfipi_file_info *handle,
   const char *buf, size_t size, off_t offset);
int (*fsync)
  (struct fsfipi_operations *context,
   struct fsfipi_file_info *handle, int datasync);
int (*release)
  (struct fsfipi_operations *context,
   struct fsfipi_file_info *handle);
int (*opendir)
  (struct fsfipi_operations *context,
   const char *path, struct fsfipi_file_info *handle);
int (*readdir)
  (struct fsfipi_operations *context,
   struct fsfipi_file_info *handle,
   void *buf, fsfipi_fill_dir_t fillfunc, off_t offset);
int (*fsyncdir)
  (struct fsfipi_operations *context,
   struct fsfipi_file_info *handle, int datasync);
int (*releasedir)
```

```
    (struct fsfipi_operations *context,
     struct fsfipi_file_info *handle);
  int (*setxattr)
    (struct fsfipi_operations *context,
     const char *path, const char *name,
     const char *value, size_t size, int flags);
  ssize_t (*getxattr)
    (struct fsfipi_operations *context,
     const char *path, const char *name,
     char *value, size_t size);
  ssize_t (*listxattr)
    (struct fsfipi_operations *context,
     const char *path, char *buf, size_t size);
  int (*removexattr)
    (struct fsfipi_operations *context,
     const char *path, const char *name);
  int (*statfs)
    (struct fsfipi_operations *context,
     const char *path, struct statfs *stfbuf);
};
```

The purpose of the members is:

- The `private_data` member points to the private data of the filter object (settings, parameters, state),
- the `global` member points to the data common to all filters and about the whole filesystem. Currently there is information whether to debug and whether the whole pipeline is thread safe,
- the member called `next` points to the `fsfipi_operations` structure of the filter in the lower layer,
- optional function `init` is called when the cascade of filters is constructed. Usually, initialization is done in the register function,
- destructor function `unregister` should release all data structures,
- file system methods taking the file path as the second argument: `getattr`, `readlink`, `mknod`, `mkdir`, `symlink`, `link`, `rename`, `unlink`, `rmdir`, `chmod`, `chown`, `utime`, `setxattr`, `getxattr`, `listxattr`, `removexattr`, `ntruncate`, `statfs`; the names are self-explanatory and correspond to system calls with the same name, only `getattr` corresponds to `stat()` and `ntruncate` corresponds to `truncate()` system call (file is determined by the name, not by a descriptor),
- regular file data manipulation functions: `open`, `truncate` (truncates file by handle), `read`, `write`, `fsync` (corresponds to `fsync()` or `fdatasync()`, depending on a parameter), `release` (corresponds to close);

`open` gets file name and returns a handle (pointer to `fsfipi_file_info` structure), the others take that handle as the second argument,

- directory manipulation functions: `opendir`, `readdir`, `fsyncdir`, `releasedir`; these methods are directory equivalents for the above, except for `readdir`, which calls a callback function (the pointer of which it has got as an argument) for each directory entry instead of filling the buffer itself.

4.3 Standard fsfipi modules

I created some basic fsfipi modules according to the analythical part, which made it possible to use it above a linux file system, and to only add specialized filters. These standard modules are:

- `localfs` performs fsfipi operations on the real filesystem. Used as the lowest level filter,
- `fuse_interface` runs a fuse file system and the fuse operations are transformed to fsfipi operations sent to the lower level. Used as the highest level filter,
- `rename` is an abstract 'class' for a filter, which changes file names. All operations taking file name as an argument call function `transform_name`, which is not defined in this module,
- `subtree` is based upon the `rename` module, and its name transforming function adds a prefix to each path, usually a directory path; it makes possible, that localfs operates on specified directory subtree rather than the full filesystem,
- `debug` prints which calls are passed through this filter. This module takes a *prefix* parameter, which is used to prefix debug messages (so that if this filter is used more than once in one pipeline, the messages can be distinguished. If this parameter is not used, filter does not pass calls to the lower level (so that the test can be read only).

4.4 Vaves

Main properties of the versioning module are determined by the analytical part of this document. There are some issues which had to be solved some way.

4.4.1 Issues

Version number overflow

The question is if the version number has some limit and if so, what to do when this limit is reached for some file. For practical reasons, the version number is internally represented as an integer of fixed length, in particular the type `long` is used, which is usually 32 bit wide and with a sign. So that positive numbers up to 2,147,483,647 can be used. Because the maximum representable number is so high, that a file changed every second will reach the limit after 68 years, reaching this limit means that most probably something goes wrong on a machine (and very often writing daemons are not the target group for the versioning system), vaves simply return an error (out of range) and does not perform the requested operation.

Multiple threads

I was initially thinking it should support multiple thread operation, but there are some problems. Because there are more physical files for one logical (user) file, and some are related to the other, it would be reasonable, if locks were related to the logical files. But for instance when accessing the version, file must be first read to determine, which logical file does it belong to. But when reading that file, no other thread should write to the file.

There are some possible solutions:

- To have two kinds of locks one for versions and one for logical files and using an optimistic deadlock policy. When accessing a version, the version would be locked and then the logical file, and when accessing a logical file or a metadata file, the logical file would be locked first and then the version. If a deadlock occurred, it would do some rollback. Implementing that rollback mechanism would be a complicated task, and writing to a journal would slow down the operations.

- To have two kinds of locks one for versions and one for logical files and avoid deadlocks. Deadlock cannot occur, if the locks are always taken in a defined order. We would always lock logical file before the version. When accessing a version, the version file would be locked, read, and unlocked. After that unlocking the version, the logical file would be locked and the version again. But then in the meantime, when nothing was locked, some other thread could change the version file, remove the version or even all the version with the metadata files, so that there would be more file reads and complicated tests.

- One big critical section for all the operations or locking all the file system for read or write. There would be almost no speed advantage over the single thread implementation.

- To resign on the fully configurable version names, and to determine the logical file name from the name of the version. Then the logical files could be easily locked.

Another significant difference between single and multi-threaded implementation would be the fact, that many big data structures would be used by each thread. So, for now, vaves can safely run only in one thread. Multi-thread support may be in some future version.

Hard links

If there are two hard links to the same file, if one is changed, the same changed file should be under the other link. In standard Unix file systems, this is implemented using inodes, which point to the data. But in the versioning system, both inodes of the old versions need to have the same inode and both inodes of the new versions need to have the same inode. Then either the versioning information should be per inode (but we are tracking the name history), or there should be some pointer to the other names (for example, the hard links can have two-way linked list), so that we can update all the other names. This would be time consuming. Because this correct behaviour is seldom necessary and the most common use of hard links is to safe disk space when storing (large) files, which are not written to, vaves only links the newest versions, and if one file is modified, the other will remain unchanged.

Directories

Because of it were too complicated to version directories, and because it is not necessary, directories are not versioned. When a directory is created, all the old versions are removed.

4.4.2 Source files

`fsfipi_vaves.c`

Definition of the file system functions with the fsfipi interface.

`fsfipi_operations.c`

Definition of the functions manipulating the files, which are used by function from `fsfipi_vaves.c`. For example routines creating new versions, removing versions, etc.

`fsfipi_operations.h`

Function prototypes of functions from the fsfipi_operations.c file.

`fsfipi_utils.c`

Simple routines for parsing strings and constructing the names of different kinds of files, which do not access files.

`fsfipi_utils.h`

Function prototypes of functions from the `fsfipi_utils.c` file.

`vaves.h`

Structures used in above files, constants, macros.

`vaves-static.c`

Because used debugging tools can not debug code in dynamically loaded shared libraries (the default use of fsfipi), this is a source for statically linked executable using vaves. Used modules are not determined by program arguments, but they are hard-wired in the source.

4.4.3 Data structures

There are three essential data structures in Vaves:

`data_vaves` contains the parameters of the vaves filter Pointer to it is stored in the `private_data` field of the `fsfipi_operations` structure.

`vaves_buffers` contains buffers for constructing file names and the operations with the file data (copying) or extended attributes. The reason, why they are grouped in the structure is that in the single thread implementation they are static, but if there would be a multi-thread implementation, they will have to be on the stack.

`lfile_data` contains the information related to the logical file, such as the complete path, name, extension, maximal and minimal major version numbers, current state (removed or existing), how many versions should be kept, and if the structure represents a logical file, a version file, or a metadata file.

`vfile_data` contains information about the used version, such as version numbers, state, and creation and removal times.

4.4.4 Internal functions

Important functions used by more file system calls or by other internal functions. I list only the most important ones.

`get_new_min_ver()` creates a new minor version of a document and fills the logical file name and version numbers (of the new version) into the data structures (`vfile_data, lfile_data`).

`get_new_maj_ver()` creates new major version of a document and fills the name and version into the data structures. If the file does not exist yet, it creates all the necessary metadata files. If the file does exist it copies the contents to the new version and removes the oldest version according to the `remember_versions=N` parameter.

`remove_item()` removes a logical file (not the versions if `remove_all_versions` is not set). Used by `unlink()`, `rmdir()` and `rename()` operations.

`remove_version()` removes a specified file version and its metadata files.

`update_version_extent()` updates the minimum and maximum major version numbers after version removal.

`get_version_info()` fills the logical file name and version numbers into the data structures (and reads the files necessary to find it out).

`get_version_info_r()` as the previous, but for read-only operations, so that in case of version files the contents file needs not to be read, and when the file does not exist, an error is returned. Uses the function above.

`read_metadata()` reads parses a file with metadata, and calls passed callback function, which fills appropriate data structures.

`get_*_name` uses filled data structures to construct the name of a metadata file of appropriate type.

`split_name()` parses given path to find a path of the logical file and an information, if the path represents a logical file, a version or a metadata file, so that the names of the metadata files can be constructed.

4.4.5 How does the filesystem functions work

Most file system operations need to determine the name of a logical file and its version to use (if not using a version file, the latest version is used). Determining the name and the version of a file (in the function `get_version_info` is done in the following way:

First, it is determined, if the name represents a a file version (there it is in the version subdirectory and begins with the version prefix), some metadata (there it is in the metadata subdirectory or there it is in version subdirectory and does not have the version prefix – for the case both directories are the same) or a logical file (the file which is versioned). Then it determines the name of the logical file and its version to use.

If the path belongs to a 'version file', the file with that name has to be read, that contains the information, which logical file it belongs to. Then,

only in the case of write operation, it reads the 'contents file' to determine the newest versions of the file and make the new minor/major version.

If the path belongs to a logical file, the 'contents file' in the metadata directory must be read. It contains the highest version numbers (major and minor).

If the path belongs to a metadata file, this is handled in a special way - such a file represents itself.

Once the logical file name and the version number are known, paths to all the metadata files and version files can be constructed and new versions created.

The operations which read some file metadata (`getattr()`, `getxattr()`, `readxattr()` call `get_version_info()` so as to get the information about the name and version and then use `get_inode_name()` so as to obtain the name of the 'fake inode' and perform the operation on the 'fake inode'. The `getattr()` operation obtains some information (length) from the data file.

The functions manipulating file metadata (chown, chattr, etc.) call `get_new_min_ver()` (which copies the fake inode as well), then obtain the name of the 'fake inode' and perform the operation on that.

The `open()` operation behaves different if a file is to be open read only and if the file is to be opened for writing. If for reading, then the name and version is determined and the date file is opened and its descriptor will be used for `read()` and `release()` operations. If the file is to be opened for writing, `get_new_maj_ver()` is called (so that the data file and fake inode are copied to the new version) and then the data file is opened in that mode.

If a new logical file is to be created using `mknod()` or `mkdir()`, a new version is created (increased numbers, updating metadata) but no contents copying occurs. Vaves recognises, that a file is removed, even if there are some old versions, from the field *remove_status* in 'contents file' or from the fact, that there is no 'contents file' yet. It is done using `get_new_maj_ver()`. Only logical files may be created.

If a logical file is to be removed using `unlink` or `rmdir`, then depending on the parameter `remove_all_versions` either all versions (and its files) are removed or only the *remove_status* is changed and the link with the name of the logical file is removed.

If a (major) version is to be removed, all its files are removed and the range of versions of the logical file is updated. If no versions leave then all files belonging to that logical file are removed.

One of the most complicated functions is `rename`. It first creates a new version of the destination and determines the version of the source. So as to avoid copying, it first tries to make the data of the new file as a hard link to the data of the old file. If this does not succeed (either the file is a directory or it is not supported), in case of a directory it is moved (directories are not versioned) and in case of a file it is copied. Finally, the old file is marked as removed (or all the versions are removed).

Functions `read()` and `write()` work with the handle returned by the `open` method, and only pass parameters to the read or write of the lower level module. That handle was returned by the lower level `open()` method when opening the 'data file'.

When reading the contents of directories, the `readdir()` function only returns the result of the lower layer, beacuase all the metadata files are visible to the user, the version files have only different behavior, and the logical files are implemented as symbolic links to the data file of the current version, so they are listed as well. The last fact has the advantage, that the version file system can be used for reading even on the physical file system, without running the file system daemon.

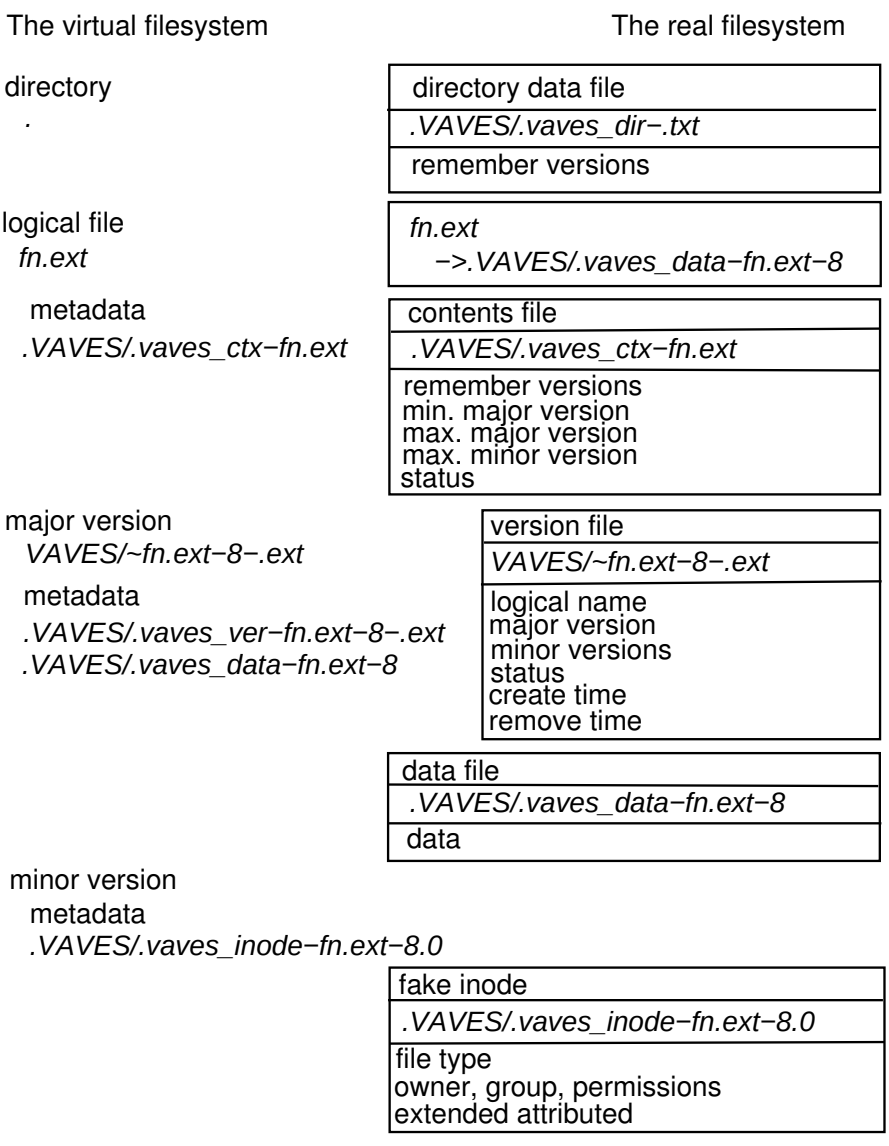

Figure 4.2: Files used in Vaves with their default names

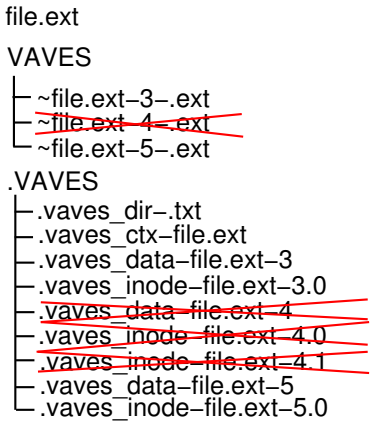

Figure 4.3: Files removed, when the user deletes ˜file.txt-4-.txt, the fourth version of a file file.ext

Chapter 5

Tests

There are many methods of testing correct function of a new file system. Their purpose is to test if all the function work correctly and if they work correctly not only at the first attempt but even under heavy load. The latest is well done using some benchmark. For first two, I have used some self-made tests.

The first test is comparative, and the success can be determined by comparing the two produced files. The two next tests are semi-automatic tests. This means, that I made scripts, which perform some filesystem operations and after some of them display the recursive directory listing or the file contents, which has to be checked by human. This method was chosen because of productivity; automatic checks takes more time to implement than a look on the listing. The next test is a by-product of the use of the benchmarking tool. The next test used comes from the Linux Test Project.

All the tests were successful.

5.1 The seek and write test

This test performs large writes and seeks to two files and then it compares them. One of these files is stored on a well-tested filesystem and the other on the being tested filesystem.

The program performing writes and seeks is written in C and stored in the file `test-write.c`. It opens one file, seeks in that file to positions between 0 and the value of the fifth parameter (by default 10MB), then writes sequences of random bytes of random length between 1 and the value of the sixth parameter (by default 1MB). The seek and write is repeated the value of the fourth parameter times (by default 200 times).

5.2 The filesystem test

This semiautomaic test is run by `test-fs.sh`. It checks, if the filesystem behaves correctly, i. e. if created files exist, contain what was written to them, have attributes which they should have, don't produce errors, etc. To run this test, the filesystem has to be run in mode `remove_all_versions=1`.

The test does in particular:

- Writes to a file, rewrites it and appends to it, reads the contents,
- changes its permissions,
- removes the file and creates a directory with the same name,
- creates and changes a file in that subdirectory,
- deletes the file in the subdirectory and the subdirectory,
- writes to a file, renames it and reads it,
- creates a long file and lists its contents in a pager,
- moves the file in a subdirectory and back,
- makes a symbolic link,
- set, lists and removes an extended attribute of a file,
- removes the used files and recursively lists directory contents.

After most operations, test lists the result (the contents of a file or a directory) and waits for user input to continue.

5.3 The test of vaves specifics

This test can be run by `test-vaves.sh`. Its purpose is to check, whether the versioning works as it should. Tests for instance combining of changing permission of the version and of the logical file, removing individual versions, creating subdirectory and file in them and removing them, etc. To run this test, the filesystem has to be run in mode `remove_all_versions=1`.

The test does the following:

- Writes to a file and rewrites it,
- changes permissions of the new and the old version,
- rewrite the file, so that there is a new version,
- changes permissions of the file,

- removes the second, the third and the first version,
- creates a directory with the same name as the file above,
- writes to and rewrite a file in that directory,
- removes the file in the directory and the directory,
- writes to a file with the same name,
- removes that file.

5.4 The Bonnie++ benchmark

Even the used benchmarking tool checked some qualities of the file system. It read froem and wrote to a file of length of 464MB without errors, created 16384 files, accessed them and removed them in both sequential and random order without errors as well, and then succeeded removing the temporary directory, in which this all was done. This means that what was read (and had been written before), existed. The details about this tool there are below.

5.5 The FS_INOD test from the LTS

This test is a part of the Linux Test Project. It has three parameters: `numdir`, `numfiles`, `numloops`. It creates two directories `dir1` and `dir2`. Then creates `numdir` subdirectories in each of them. Then it creates `numfiles` files in each of these subdirectories in the `dir1` directory in sequential order. Then it removes these files and creates them in the `dir2` directory and removes those as well. This file creating and removing is repeated `numloops` times. This test was successful with the parameters of 100, 100 and 2 and the vaves run with the `remove_all_versions=1` parameter.

Chapter 6

Benchmarks

6.1 What and how it was measured

I used bonnie++, a utility to test hard drive performance. The utility consists of a few test, measuring different types of file operations. Bonnie++ and the information about it can be found on

- `http://www.coker.com.au/bonnie++/` or
- `http://sourceforge.net/projects/bonnie/`.

Bonnie++ contains following tests:

I/O tests: Used file of length 464MB

Sequential per character write
: writes the data using the putc() stdio macro/function,

sequential block write
: writes the data using the write() syscall,

per block read and rewrite
: each block is read (using read()), dirtied and written using write(), requiring lseek() to the original location,

sequential per character read
: reads file using getc() in a loop,

sequential block read
: reads the file using read(),

random seeks
: three parallel processes doing a total of 8000 lseek()s to random locations. In each case the block is read (using read()), 10% of them are also dirtied and written back using write().

File manipulation tests (manipulates 16384 files):

It uses file names containing 7 digits numbers and an string of 0 to 12 random alpha-numeric characters. For the sequential tests the random string follows the number, for random tests the random string is the first.

Sequential file create
: creates files in the number order,

sequential file stat
: performs stat() on all files in the order, in which they are returned from readdir(),

sequential file delete
: deletes the files in the same order,

random file create
: creates the files,

random file stat
: perform stat() on random files. Not all files must be stated(),

random file delete
: Delete files in random order.

So as to measure, how much the Vaves is slower than a standard unix filesystems, I compared the following tests:

1. The benchmark was running directly on the standard filesystem. (labeled 'disk')

2. The benchmark was running on a directory on the fuse filesystem fusexmp, which mirrors the whole virtual filesystem. The mirrored directory is the same as the one used in the first test. Comparing with the first test gets us the overhead caused by using fuse and the fuse example filesystem. (labeled 'fusexmp')

3. The benchmark was running on a directory on a fsfipi filesystem with only the necessary filters, i.e. `fuse_interface`, `subtree`, `localfs`. The directory mirrored is that used in the first test. Comparing with the second test gets us the overhead of fsfipi. (labeled 'fsfipi')

4. The benchmark was running on a fsfipi filesystem using even the vaves filter with parameter `remove_all_versions=1` (so as not to fail when removing a temporary directory) Comparing with the third test gets us the overhead of the Vaves. (labeled 'vaves')

So as to compare the influence of the underlying filesystem and device, I applied the above tests on the following configurations:

- ext3 with *filetype* option on and *dir_index* option off (the default)
- ext3 with *filetype* option off and *dir_index* option on
- reiserfs
- reiserfs (partition used from 59%, there may be some fragmentation)

All the tests were done on a machine with CPU AMD Sempron 2300+ with 128kB L1 cache and 256kB L2 cache, 256 MB RAM (16MB shared with integrated GPU), VIA VT8378 [KM400] chipset, IDE hard disk Maxtor 6Y080L0.

6.2 Results

Here there are the tables of results of the tests. If bonnie++ returns +++++ instead of a value somewhere, it is because the test lasted less than 0.5 second, which means that the throughput was at least 32768 files per second, so I write '>32768' there.

	disk	fusexmp	fsfipi	vaves
putc write [KB/sec]	27085	21717	22585	22380
write [KB/sec]	43894	41824	38744	39567
rewrite [KB/sec]	12735	7189	7357	7531
getc read [KB/sec]	19832	16352	16624	15441
read [KB/sec]	40594	31597	33717	27782
seek [/sec]	130.4	96.9	102.7	49.6
seq. create [/sec]	1514	1143	792	30
seq. stat [/sec]	>32768	>32768	31190	2415
seq. delete [/sec]	>32768	17897	17321	1632
rand. create [/sec]	1287	1010	920	30
rand. stat [/sec]	>32768	>32768	>32768	104
rand. delete [/sec]	3345	2499	2367	206

Table 6.1: Throughput of operations on default ext3

	disk	fusexmp	fsfipi	vaves
putc write [KB/sec]	28386	20425	22525	22027
write [KB/sec]	42296	35844	41943	43551
rewrite [KB/sec]	12060	7806	7329	7660
getc read [KB/sec]	19370	8549	16437	15883
read [KB/sec]	38206	11294	33913	31543
seek [/sec]	113.3	103.0	103.0	53.0
seq. create [/sec]	32314	7639	7898	1492
seq. stat [/sec]	>32768	>32768	>32768	157
seq. delete [/sec]	>32768	13059	12747	330
rand. create [/sec]	>32768	7761	7656	1379
rand. stat [/sec]	>32768	>32768	>32768	165
rand. delete [/sec]	>32768	12617	11639	328

Table 6.2: Throughput of operations on ext3 with dirindex

	disk	fusexmp	fsfipi	vaves
putc write [KB/sec]	27765	19685	22501	22844
write [KB/sec]	42472	40690	44443	42823
rewrite [KB/sec]	12096	7665	7796	7832
getc read [KB/sec]	17025	15497	15555	16061
read [KB/sec]	36931	31525	33567	30676
seek [/sec]	151.6	118.4	115.7	61.2
seq. create [/sec]	20887	6738	6744	395
seq. stat [/sec]	>32768	>32768	>32768	615
seq. delete [/sec]	17333	8440	8234	274
rand. create [/sec]	19778	6661	6374	404
rand. stat [/sec]	>32768	>32768	>32768	558
rand. delete [/sec]	15127	8054	7755	209

Table 6.3: Throughput of operations on reiserfs

	disk	fusexmp	fsfipi	vaves
putc write [KB/sec]	25758	17534	15272	15632
write [KB/sec]	39769	33353	39449	31201
rewrite [KB/sec]	11654	8480	7100	6894
getc read [KB/sec]	11369	7568	11264	11876
read [KB/sec]	15813	9490	21752	22846
seek [/sec]	132.6	93.7	112.5	48.7
seq. create [/sec]	17987	6136	4120	303
seq. stat [/sec]	>32768	>32768	24055	1029
seq. delete [/sec]	13386	5535	4804	164
rand. create [/sec]	17290	6046	4351	311
rand. stat [/sec]	>32768	>32768	30360	1022
rand. delete [/sec]	11993	6642	4681	116

Table 6.4: Throughput of operations on non-empty reiserfs

And for the comparison, how the filesystem influence this performance.

	ext3 diri-	ext3 diri+	reiserfs	used reiserfs
putc write [KB/sec]	27085	28386	27765	25758
write [KB/sec]	43894	42296	42472	39769
rewrite [KB/sec]	12735	12060	12096	11654
getc read [KB/sec]	19832	19370	17025	11369
read [KB/sec]	40594	38206	36931	15813
seek [/sec]	130.4	113.3	151.6	132.6
seq. create [/sec]	1514	32314	20887	17987
seq. delete [/sec]	>32768	>32768	17333	13386
rand. create [/sec]	1287	>32768	19778	17290
rand. delete [/sec]	3345	>32768	15127	11993

Table 6.5: Throughput of operations directly on filesystem

	ext3 diri-	ext3 diri+	reiserfs	used reiserfs
putc write [KB/sec]	22380	22027	22844	15632
write [KB/sec]	39567	43551	42823	31201
rewrite [KB/sec]	7531	7660	7832	6894
getc read [KB/sec]	15441	15883	16061	11876
read [KB/sec]	27782	31543	30676	22846
seek [/sec]	49.6	53.0	61.2	48.7
seq. create [/sec]	30	1492	395	303
seq. stat [/sec]	2415	157	615	1029
seq. delete [/sec]	1632	330	274	164
rand. create [/sec]	30	1379	404	311
rand. stat [/sec]	104	165	558	1022
rand. delete [/sec]	206	328	209	116

Table 6.6: Throughput of operations with vaves

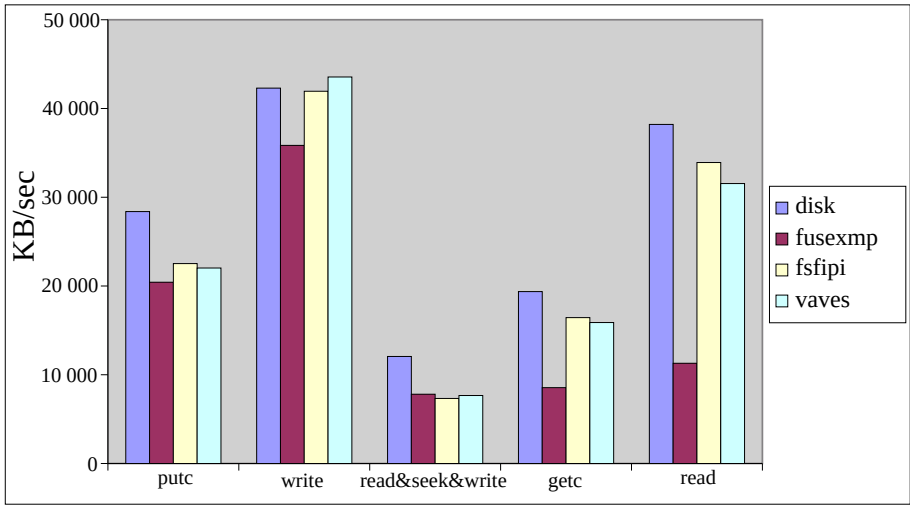

Figure 6.1: Troughput of file i/o operations on ext3 with dirindex

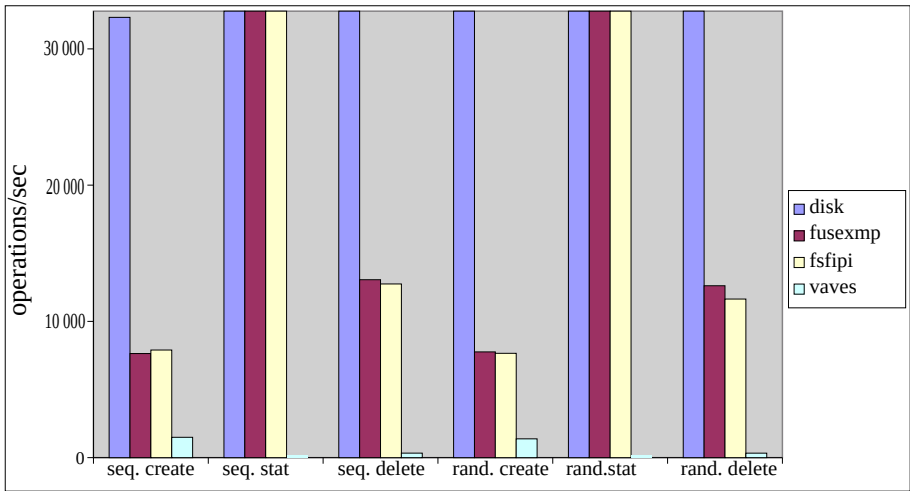

Figure 6.2: Troughput of directory operations on ext3 with dirindex

6.3 Interpretation

The throughput of fusexmp and fsfipi is almost the same (Mostly, fsfipi is a little better, sometimes fusexmp is better). Comparing them both with the operations directly on filesystem, the file input and ouput does need almost the same amount of disk operations, but requires more CPU time, that is why when the CPU is busy doing putc() operations the throughput of them is that slower than on the filesystem itself. Even the block read operations and seeks are much slower, but generally not less than of a half. Directory

operations of fuse virtual filesystems are about three to five times slower than directly on a filesystem with non-sequential search in directory (in the case of the filesystem with sequential search, the difference is naturally not so big).

Vaves file i/o operations are comparable to those of the fuse filesystems alone (this is because the test opens the file only once, and then only reads/writes), mostly only a bit slower. The directory operations however are much slower, than directly, in particular from 5 times to 50 times in case of create and unlink operations, even several hundereds times in case of calling stat. The reason is simple: Normally only one file is searched and many directory data is cached, but in vaves first the contents file is searched, then its data block is read (which is not in the cache), then the metadata of major version must be read, and then the 'fake inode' of the minor version (in the case of stat). In short, many files are searched, read, created or removed respectively. Even in the case of only getting the attributes of a file, which is normally very quick operation.

The benchmark naturally did not measure the speed of operations with versions because it would require a special benchmarking tool and one could not compare the results with anything anyway. I can say that it would not be much slower, because the number of files to be read is the same or sometimes even less (contents file does not have to be read for version reading operations). Accessing versions is supposed not to be used so often as accessing regular files anyway.

Chapter 7

Conclusion

I succeeded to implement the versioning system core in userspace. It is very well configurable, but its main drawbacks are the low speed of file open operations, and the inability of using multiple threads and for some people also the lack of atomic operations and the fact, that all the versions are stored completely.

These remain challenges to the future. The system could be speeded up with low cost, if we do not use minor versions, there may be a setting in the future for disabling minor versions. Another increase of the speed could be reached if we merge the storage of the version metadata and the data to one real file or if the metadata of all versions were stored in one file, so that much less files would have to be opened, but this would be at the cost of losing the transparency. Another way of increasing the speed would be if the version names were not so configurable so the name of the logical file would be determined without reading the metadata file and if the metadata for versions were not necessary (but we would lose the information, when the version was deleted). Then a hard link to the newest version of a file would be possible, which could speed up access to the newest version.

In my humble opinion, the fsfipi framework, which I made as well, will have better future, than the versioning system vaves itself.

Bibliography

[1] Hewlett Packard.
Guide to OpenVMS File Applications,
OpenVMS systems documentation edition, 2002.
Electronic version at
`http://h71000.www7.hp.com/doc/os82_index.html`
or a copy at
`http://cs.felk.cvut.cz/vms82/`.

[2] Brian Schenkenberger.
OpenVMS File System Internals.
Digital Press, Dec 2003.
ISBN: 1555582699.

[3] RFC 3284 - The VCDIFF generic differencing and compression data format.
`http://www.faqs.org/rfcs/rfc3284.html`.

[4] Miklos Szeredi.
Fuse homepage.
`http://fuse.sourceforge.net/`.

Used software

I used gcc compiler, kdevelop development environment (which uses gdb debugger), bonnie++ benchmarking tool, different versions of fuse, as it was developed (from some 1.x to 2.4; current code will run under 2.4), bash (running the scripts) and posh (testing POSIX compliance of scripts), GNU core utilities. And many more software not directly influencing the work like Konsole (running scripts, testing filesystem, etc.), Krusader, KDE, Linux kernel and the whole Debian GNU/Linux operating system.

I used LaTeXand its front-end Kile for writing this text, gnumeric for plotting the graphs and inkscape for creating the figures.