



Diploma Thesis
Development of a Linux Overlay Filesystem
for Software Updates in Embedded Systems

Markus Klotzbücher

Constance, 31 March 2004

Abstract

Computer Systems use many different kinds of storage media such as hard disks, flash memory or CD-ROM in order to permanently preserve information. While some of these media can be modified after initial writing, for some this can only be done with much effort and some cannot be changed at all. Nevertheless, it is often desirable to do this, for instance, to be able to apply software updates to the read-only flash filesystem of an embedded system.

An overlay filesystem is a type of filesystem that serves to make read-only devices such as CD-ROM *virtually* writable. This is done by storing all changes to a writable location while allowing the user to transparently read, modify and write the files located on the read-only device.

This diploma thesis paper describes the design, implementation and evaluation of the *mini_fo* overlay filesystem, an overlay filesystem developed for the special requirements of embedded systems.

Contents

1	Introduction	6
1.1	Introduction to Thesis	6
1.1.1	Overlay Filesystems	6
1.1.2	Embedded Systems	7
1.2	Organisation	8
2	Background	9
2.1	Introduction to UNIX Filesystems	9
2.1.1	Overview	9
2.1.2	Filesystem types	12
2.1.3	The Virtual Filesystem (VFS)	12
2.2	Stackable Filesystems	14
2.2.1	Introduction	15
2.2.2	Fan out filesystems	17
2.2.3	Interposition	18
2.3	Related Work	20
2.3.1	Overlay Filesystem by A. Naseef	20
2.3.2	Translucency	21
2.3.3	FiST and Wrapfs	21
2.3.4	FreeBSD union filesystem	22
2.3.5	Inheriting File System (IFS)	22
3	Design	23
3.1	Requirements	23
3.2	System Design	24
3.2.1	Terminology	24
3.2.2	Meta Data	24
3.2.3	Whiteout entries	27
3.2.4	Overlay filesystem states	28
3.2.5	Implementation	29
3.3	Limitations	33

4	Implementation Aspects	34
4.1	Difficulties	34
4.1.1	Unique inode numbers	34
4.1.2	Directory reading	35
4.1.3	Directory renaming	39
4.2	Where to store the overlay filesystem file state?	40
4.3	“On the spot” storage	41
4.4	Testing	42
5	Performance	43
5.1	Kernel compilation	43
5.2	Time consuming operations	45
6	Conclusion	47
A	Appendix	49
A.1	Mount Syntax	49
A.2	Implementation Details	49
A.2.1	Overlay Filesystem file states	49
A.2.2	Overlay filesystem file object states	51
A.2.3	Deletion process	51
A.2.4	Creation process	52
A.3	Flowcharts	53
A.3.1	The <code>build_sto_structure</code> helper function	53
A.3.2	The <code>mini_fo_lookup</code> function	54
A.3.3	The <code>mini_fo_open</code> function	55
A.3.4	The <code>mini_fo_create</code> function	56
A.3.5	The <code>mini_fo_setattr</code> function	57
A.4	Mini Benchmark Shell Scripts	58
A.4.1	Auxiliary scripts	58
A.4.2	Append benchmark scripts	59
A.4.3	Readdir benchmark scripts	62
A.5	Acknowledgment	67

List of Figures

1.1	Overlay filesystem basic mode of operation	7
2.1	A typical UNIX root filesystem	10
2.2	A typical UNIX root filesystem after mounting a CD	11
2.3	Relation generic VFS layer - specific filesystem layer	13
2.4	Relationship between VFS, stackable- and lower filesystem	15
2.5	Multiple filesystem stacking	17
2.6	Relationship between VFS, stackable fan out- and lower filesystem	18
2.7	Stacking on top of a disk-based filesystem	19
2.8	Interposition of VFS objects	19
2.9	Chains of interposed objects for an open file	20
3.1	Storage and base filesystem	24
3.2	Unmodified overlay filesystem	25
3.3	Overlay filesystem after creating corresponding storage directory	25
3.4	Overlay filesystem after copying base file	26
3.5	Overlay filesystem after modifying penguin	26
3.6	Unmodified overlay filesystem	27
3.7	Overlay filesystem after deleting “dog”	27
3.8	Chains of interposed VFS objects for the overlay filesystem	30
3.9	Generic flowchart of a non-state changing function	31
3.10	Generic flowchart of a state changing function	32
4.1	One to one inode number mapping	35
4.2	Which inode number should be used?	35
4.3	Directory reading	36
4.4	Directory reading in the overlay filesystem	37
4.5	Directory renaming initial situation	39
4.6	Broken relationship base and storage file	40
5.1	System times for each step of the kernel compilation	44

5.2	System times of append benchmark	45
5.3	System times of readdir benchmark	46
A.1	Flowchart of the <code>build_sto_structure</code> function	53
A.2	Flowchart of the <code>mini_fo_lookup</code> function	54
A.3	Flowchart of the <code>mini_fo_open</code> function	55
A.4	Flowchart of the <code>mini_fo_create</code> function	56
A.5	Flowchart of the <code>mini_fo_settattr</code> function	57

List of Tables

5.1	Kernel compilation mini benchmark	43
A.1	Overlay filesystem file states	50
A.2	Overlay filesystem file object states	51
A.3	State transitions when deleting a file	52
A.4	State transitions when creating a file	52

Chapter 1

Introduction

1.1 Introduction to Thesis

1.1.1 Overlay Filesystems

Generally, a filesystem is a computer program that defines how data is stored and organized on a medium such as a hard disk or a CD-ROM, and that offers a set of well known operations for accessing and modifying the data. A user can access and potentially modify stored data by use of these operations, without having to know any details on how the data is stored. Yet, not all filesystems can be modified: those such as CD-ROM can only be written to once and other types might have been assigned the attribute read-only by an administrator to protect critical data.

An overlay filesystem aims to overcome this limitation, by *virtually* making a read-only filesystem writable. This is achieved by redirecting all modifying operations from the read-only filesystem to a different, writable location. When data is to be read, the set union of the read-only filesystem and the writable location is produced, while the writable location containing the newer data takes precedence in case there are duplicates. Thus a read-only filesystem can transparently be accessed and modified just like a read-write filesystem, but without modifying the original data. Possible areas of application are applying software updates to read-only filesystems (see section 1.1.2), sandboxing environments and live CDs (see Chapter 6).

Figure 1.1 illustrates the basic mode of operation of the overlay filesystem using an example with a CD-ROM (read-only device) and a hard-disk as writable location. Non modifying operations such as reading result in the information of the hard-disk and the CD-ROM being merged (a), while modifying operations such as writing are directed to the hard-disk (b).

This diploma thesis describes the *mini_fo* overlay filesystem. The origin

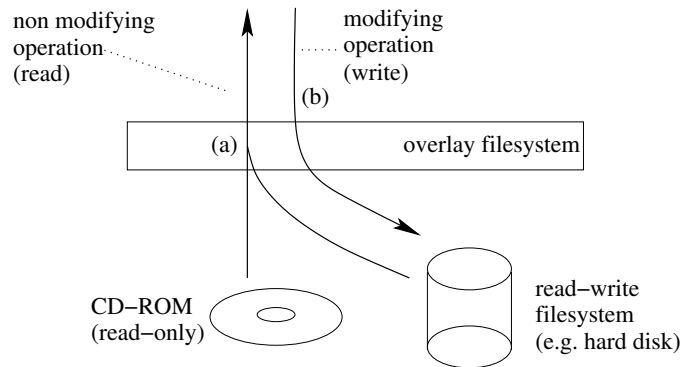


Figure 1.1: Overlay filesystem basic mode of operation

of this name is pure coincidence and explained in section 3.2.5.

1.1.2 Embedded Systems

Embedded Systems are purpose-built, mainly small computer systems that are increasingly designed using GNU/Linux [2, 3] as an operating system. Every Linux System requires a so called root filesystem that contains the most important programmes and information to start up and maintain the system.

This root filesystem is commonly accessed read-only, as changes are not required during normal operation. But in certain cases it becomes necessary to be able to modify files in the root filesystem. This occurs for example, when software updates or bug fixes need to be applied. As the root filesystem is normally located on a flash device, the use of a flash filesystem allowing read-write access such as JFFS2 (Journaling flash filesystem 2) might seem appropriate. Although this method would ease the process of writing files, the overhead caused by mechanisms of journaling and garbage collection does not justify its use in some cases. In contrast, by use of an overlay filesystem, changes to the root filesystem can be easily applied, avoiding the overhead implied by the use of flash filesystems and the negative impact of large memory footprints.

Furthermore, embedded systems have more, specific requirements that need to be considered. The amount of permanent storage such as flash memory is limited, calling for efficient handling of the storage of changes. For instance, a bad example would be to have the contents of a directory duplicated after having renamed it, as this would unnecessarily store redundant

data. Also write operations on the storage filesystem (which in most cases is likely to be a JFFS2 or similar flash partition) should be limited to the minimum necessary in order to avoid memory footprints.

Another issue is caused by the limited amount of RAM. Some existing overlay filesystem implementations hold all information on files in memory. While this approach might be reasonable for small directories or machines with a lot of RAM, an embedded system overlaying a large root filesystem could be easily rendered unusable due to the large amount of information held in RAM. Therefore special care needs to be taken on how storage of this data is managed.

1.2 Organisation

This thesis is organised as follows. Chapter 2 provides the background information required to understand the ideas and principles of the design of this overlay filesystem. First, a general introduction is given to the UNIX filesystem that applies to Linux being a UNIX clone as well. Next the concept of stackable filesystems is explained. This is important as the *mini_fo* overlay filesystem itself functions as a stackable filesystem. The last section of Chapter 2 describes and discusses other efforts that have been made to develop an overlay filesystem.

Chapter 3 describes the requirements, the design and the limitations of the current implementation. Chapter 4 goes more into detail and illustrates some interesting issues encountered during the implementation. In Chapter 5 results of several benchmarks measuring performance of different operations of the overlay filesystem are presented. Chapter 6 summarizes the thesis and concludes the work.

Finally, the Appendices contain the mount syntax, more implementation details and flowcharts for several important functions.

Chapter 2

Background

2.1 Introduction to UNIX Filesystems

This Chapter is intended for the reader with little or no knowledge of UNIX filesystems. Further information can be found in [4, 5, 6]. If you are familiar with these topics, you might like to skip this section and continue with 2.2.

Before introducing the concept of the UNIX filesystem, the terms file and filesystem need to be defined. Wikipedia[1], the Free Encyclopedia defines them as follows:

"A file in a computer system is a stream of bits stored as a single unit, typically in a file system on disk or magnetic tape. [...]"

"In computing, a file system is, informally, the mechanism by which computer files are stored and organized on a storage device, such as a hard drive. More formally, a file system is a set of abstract data types that are necessary for the storage, hierarchical organization, manipulation, navigation, access and retrieval of data. In fact, some filing systems (such as NFS) don't actually deal directly with storage devices at all. [...]"

2.1.1 Overview

Every UNIX System requires at least one filesystem called *root filesystem*, which contains programmes and information to start up and manage the system. This filesystem consists of a hierarchical tree of directories and files, whereas the root of this tree is consequentially called *root*, which is denoted by “/”. Figure 2.1 shows a typical UNIX root filesystem.

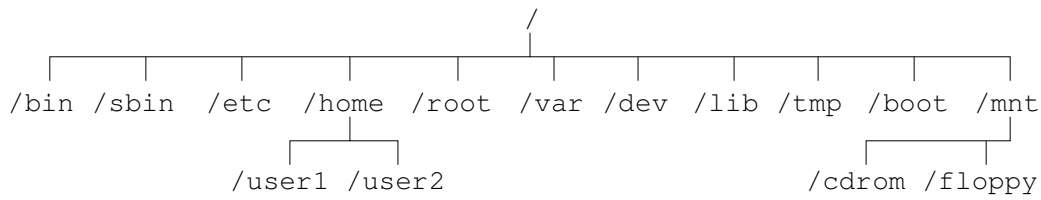


Figure 2.1: A typical UNIX root filesystem

A user may perform operations such as creating, modifying or deleting on files and directories, provided that he has sufficient *permissions* to do so. Permissions are part of the *file attributes* which, among other things, include fields such as owner of the file, file size, access times and type of file. The latter leads us to an interesting aspect of the UNIX filesystem.

The UNIX File

The term file is commonly used to describe a container of information stored in some directory, as for example a text or an executable program file. Strictly speaking this is not correct, the UNIX terminology names this type of file a *regular file*. File itself is used in a broader sense describing various types of files, which have very different uses. To avoid confusion, from now on the term file will be used for the broader meaning, for the different file types their proper names respectively. A file can be of one of the following types:

1. *Regular file*: commonly called “file”, a named container of information stored in a directory.
2. *Directory file*: a directory is a file which contains an entry called directory entry for each file in the directory. Each entry can be thought of as a name and a set of attributes that point to a file (for example its inode, see 2.1.3).
3. *Symbolic link*: a symbolic link is a file containing a pointer to a different file which can be used through the symbolic link. This file type serves to overcome the limitations of *hard links* which are discussed later.
4. *Block and character oriented device file*: these files do not store any data, but rather offer an interface to an I/O device, that can be accessed directly through the file.

5. *Special files (Pipes and Sockets)* Pipes and sockets are special types of files used for interprocess communication and networking.

The diversity of file types shows how strong the UNIX operation system is centered around its filesystem. Almost any task, from reading and writing of regular files to interaction with device drivers on the kernel side involves performing operations on files.

Mounting

In many cases, a user will require access to a filesystem located on a device different from the device containing the root filesystem. This might be a different partition on the same hard drive, a CD in a CD-ROM drive or a network filesystem (see 2.1.2). To do this, the operating system needs to be told to take the filesystem and let it appear under a directory of the root filesystem, the mount point. This process is called *mounting a filesystem*. Figure 2.2 shows the root filesystem from above after a CD-ROM has been mounted to the mount point `/mnt/cdrom`. As long as the CD-ROM is mounted, the information stored on it will be available there.

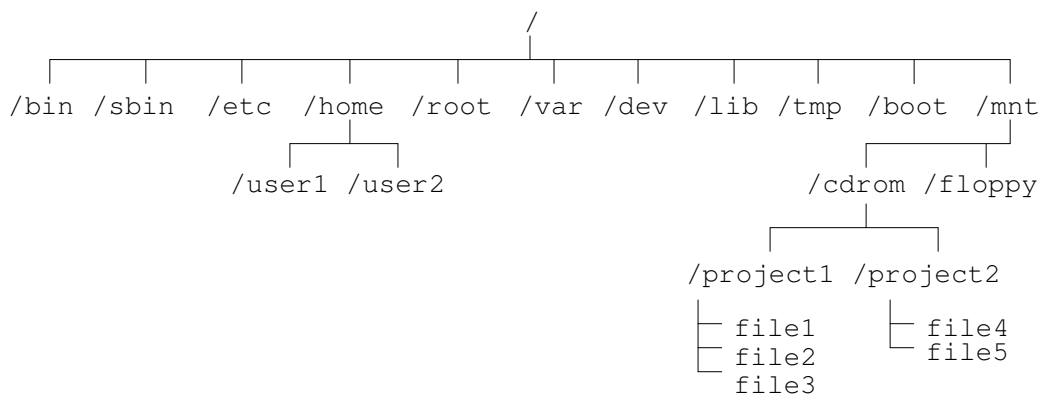


Figure 2.2: A typical UNIX root filesystem after mounting a CD

Hard links

As mentioned before directories are files that contain directory entries that point to an inode. Due to this, it become possible for one file on disk to be pointed to by different directory entries. It is said that this file has different *hard links*. In order to maintain multiple hard links, for each inode (see section 2.1.3) a link counter is maintained, which is decremented each time

a hard link is deleted and incremented when created. Only when the link counter reaches zero, the actual data blocks on disk will be released.

Hard links are subject to some restrictions. First, users are not allowed to create hard links to directories, as it could be possible to create loops that cannot be handled by some programmes. Secondly, it is not allowed to create hard links that cross filesystems. This is because in the directory entry the inode number is used as a pointer to the inode, and inode numbers are only unique within one filesystem.

2.1.2 Filesystem types

While the general Unix File Model is common to all filesystems implemented for Unix and Linux, the underlying way of how or even where the data is stored may be very different. In general, three different types of filesystem can be differentiated[4]:

Disk-based filesystems

Disk-based filesystems are probably the most common type, they manage the free memory on a hard drive partition or a similar device. Widely used examples are the Second Extended Filesystem (ext2), the Third Extended Filesystem (ext3) and the Reiser Filesystem, but also the ISO9660 CD ROM filesystem or Microsoft filesystems as the MS-DOS filesystem . Also flash filesystems such as JFFS, JFFS2 and cramfs belong to this category.

Network filesystems

These allow the use of disk-based filesystems located on different computers over a network. Examples are NFS, Coda or SMB (Server Message Block). As any other filesystem, they are mounted to mount points beneath the root filesystem.

Special filesystems

Special filesystems, also Pseudo or Virtual filesystems do not manage disk space as disk-based do, or like network filesystems do indirectly. An example is the /proc filesystem that allows direct access to kernel data structures.

2.1.3 The Virtual Filesystem (VFS)

In the last Chapter the UNIX filesystem was viewed mainly from the user's point of view, who, in most cases, does not need to know about how or even

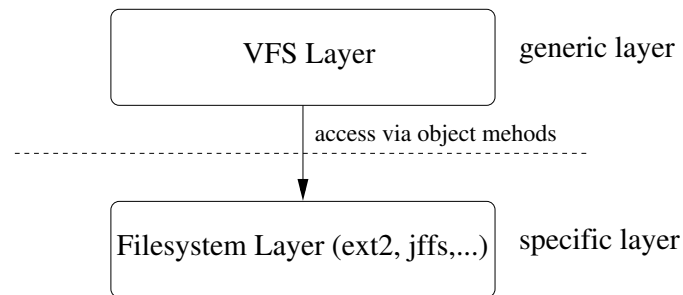


Figure 2.3: Relation generic VFS layer - specific filesystem layer

where his data is stored, as he can access it independently of the filesystem type. This requires a high level of abstraction, which is provided by the *Virtual Filesystem*, also called VFS or Virtual Filesystem Switch. The VFS can be thought of as a software layer handling the generic tasks involved in the use of a filesystem, mainly error checking, caching VFS objects, doing the required locking and communicating with user space. Once the generic tasks are completed, the VFS will pass the operation on to the specific layer, where depending on the filesystem type, it may be completed or again, passed on to the next lower layer. Figure 2.3 shows the relation between the generic VFS layer and the specific filesystem layer.

To achieve this strict separation, *the common file model* was introduced. This object oriented model defines operations and objects which are common to all filesystems. The following objects are defined in the common file model:

- *The superblock object*: holds all data relevant to a mounted filesystem. Disk-based filesystems usually store a copy of this object on disk.
- *The inode object*: this object stores information on a file located within a filesystem, such as size, permissions or access time. Most important, it holds the inode number which is used to uniquely identify a file within a filesystem. Disk-based filesystems usually store a copy of this object on disk.
- *The file object*: this object is used to store information regarding an open file. It holds information on the file and the processes using it. This object exist only in memory.
- *The dentry object*: this object represents a link from a directory entry to an inode. This is necessary because one file can be represented by

several hard links. The VFS caches the dentry objects in order to speed up lookup operations.

Each of these objects has a set of operations associated with it, which can be defined by a specific filesystem. These operations are implemented as a set of function pointers which are used by the VFS to call the appropriate function of a specific filesystem. Thus, the separation of generic and specific layer shown in figure 2.3 is achieved as the VFS can simply call the appropriate function of a specific filesystem, without having to know any specific detail on how the data is stored. Not all functions need to be defined, as the VFS provides generic functions that will be used if a function is undefined.

To illustrate this, consider a user requesting to open a regular file located on a local hard disk via the `open(2)` system call. Simplified, the following actions are performed:

1. Read the name of the file to be opened from user space.
2. Find an unused file descriptor for the new open file.
3. Perform the lookup operation, here the VFS Layer will call the `lookup` function of the specific filesystem layer that will locate the file on the hard disk and will return information on it.
4. Check if the user requesting to open the file has sufficient permission¹ to do so. Return the appropriate error code if not.
5. Allocate a new kernel file object and initialize it.
6. Call the filesystem specific `open` method passing the file object and flags if it is defined.
7. Return the new file descriptor on success or on failure the appropriate error code.

2.2 Stackable Filesystems

This section introduces the concept of stackable filesystems. This is important as the implementation of the `mini_fo` overlay filesystem is based on the existing code of stackable filesystems and is a special type of one itself.

¹If defined, the function of the disk-based filesystem will be called to perform this check, otherwise the VFS invokes a generic function.

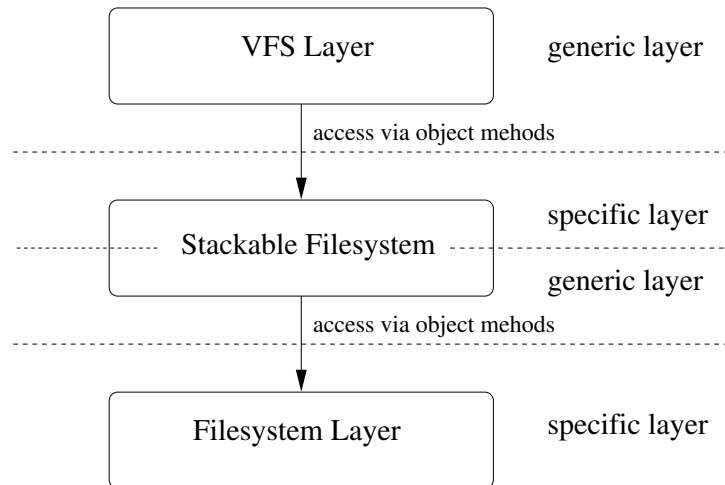


Figure 2.4: Relationship between VFS, stackable- and lower filesystem

2.2.1 Introduction

The development of filesystems is difficult and time consuming. This is because they are mainly implemented in kernel space where debugging is difficult and small bugs might crash the whole system. Also filesystems tend to consist of large amounts of code that, due to the nature of operating system kernels, is fairly complex. Therefore changing and extending existing functionality is difficult and often not welcomed by users, who rely on the stability and reliability of a certain filesystem.

Stackable filesystems provide a solution to some of these problems. Instead of modifying an existing filesystem, a new filesystem layer is stacked on top of the existing one. This new layer adds the required functionality in a modular way, and therefore can be developed, tested and maintained separately. Figure 2.4 shows the relationship between the VFS, the stackable and the lower filesystem.

The VFS Layer that handles the generic actions involved in a filesystem operation remains at the top of this hierarchy. When these actions have been completed, the function is called for the next lower level. In a traditional non stacking environment this next layer would probably be a disk-based or network filesystem. The VFS as a generic layer does not need to know of what type of filesystem the lower layer is, and will simply call the appropriate function via the VFS-object function pointers. This way it becomes possible

to introduce a new layer, the stackable filesystem layer. In practice, this is done by mounting the stackable filesystem on top of a mounted filesystem and accessing it through the new mount point.

In order to achieve this filesystem stacking the new layer has to behave differently from traditional filesystems. As lower layers do not have any knowledge about their caller, the stackable layer needs to present itself in two different manners: to the lower level as the VFS and to the higher as a traditional filesystem. When the VFS calls a function of the stackable layer, at first the generic tasks are performed, as done previously by the VFS for the stackable filesystem. Next the stackable layer invokes the same function on the layer beneath it. When the call returns, the result is handled appropriately and then returned to the VFS. That way various filesystems can be stacked on top of each other, each adding a specific functionality. For example, the scenario shown in figure 2.5 could be considered:

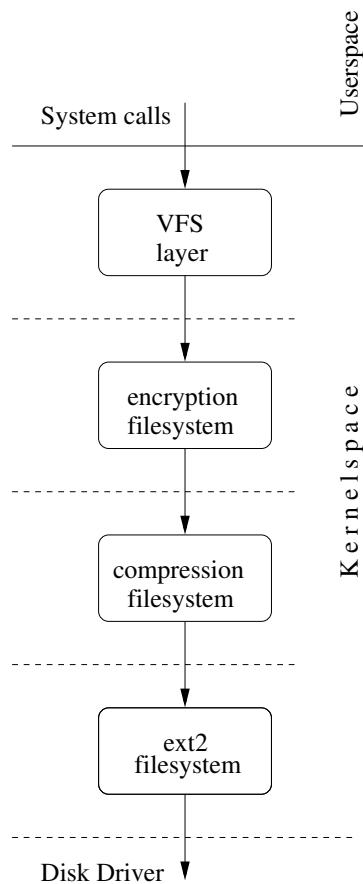


Figure 2.5: Multiple filesystem stacking

In order to extend the functionality of a standard filesystem in this example a stackable encryption and a compression filesystem are stacked on top of the disk-based filesystem ext2. None of the filesystems beneath the VFS are aware of by whom they are called, nor is there any need to know. The arrows symbolize the direction of function calls.

2.2.2 Fan out filesystems

In the previous example (figure 2.5), both the encryption as well as the compression filesystem are stacked on top of only one lower filesystem. This is normally the case, although it is possible to stack on two or more. Such a filesystem is called *fan out filesystem*. Figure 2.6 shows the relationship between the VFS, the fan out stackable layer and the lower layers.

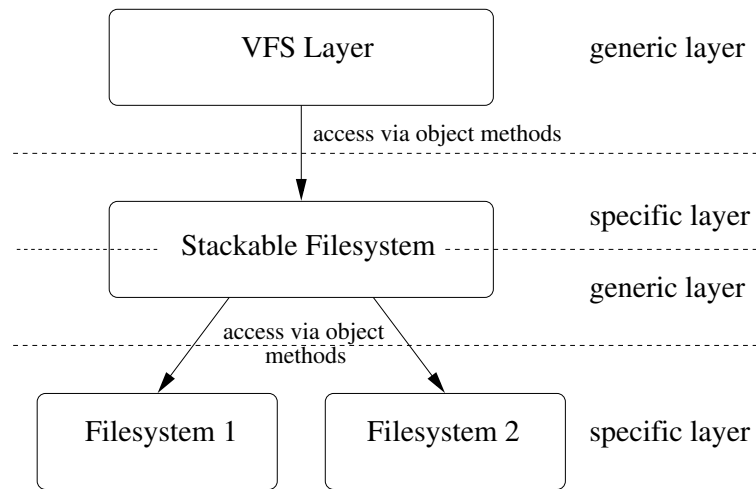


Figure 2.6: Relationship between VFS, stackable fan out- and lower filesystem

Fan out allows interesting new implementations, such as load-balancing, replicating, caching and unifying filesystems [10]. An overlay filesystem belongs to the latter category, and has a fan out of two, as it stacks on the base and storage filesystem. A detailed description of the design of the overlay filesystem can be found in Chapter 3.

2.2.3 Interposition

How is filesystem stacking achieved internally? Section 2.1.3 describes how the Common File Model defines objects and methods which are implemented by a specific filesystem and can be called by the VFS Layer. As a stackable filesystem presents itself to the VFS as a regular filesystem this is not any different, although there are some specialties related to the specific layer of stackable filesystems (see figure 2.4). Consider the example of opening a regular file from section 2.1.3, but having mounted a stackable encryption filesystem on top of a standard disk-based filesystem, as shown in figure 2.7.

At some time the VFS will invoke the stackable filesystem open method, passing the inode object of the file to be opened and the new file object. The normal behaviour of a stackable filesystem will now be to perform the generic actions and then invoke the open method of the next lower level. Please note that the VFS objects passed as parameters from the VFS are part of the stackable layer, and cannot be used to invoke the lower layer method. So to do this, first we need to obtain the disk-based filesystems VFS inode object

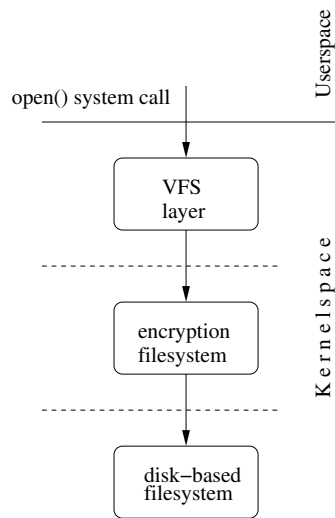


Figure 2.7: Stacking on top of a disk-based filesystem

that corresponds to the inode of the stackable layer. While in theory it would be possible to ask the lower layer to perform a lookup operation, this would be unnecessarily time consuming as this object already has been looked up before.

To avoid this, each time a new VFS object of the lower layer filesystem is created and loaded, the corresponding object in the stackable filesystem layer saves a pointer to it. This stacking of VFS objects is called *interposition*², or the stackable VFS object is *interposed* on the lower. Figure 2.8 shows this.

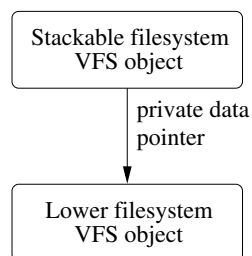


Figure 2.8: Interposition of VFS objects

In practice, interposition of VFS objects results in complex data struc-

²This term was introduced by Rosenthal, Skinner and Wong.

tures linked together. Figure 2.9 shows the VFS objects and their pointers to related objects involved in a opened file in a stackable filesystem environment. Horizontal arrows represent pointers that are common to the VFS layer, while the vertical arrows represent pointers that are required for interposition. This way the stackable layer can easily retrieve corresponding lower layer VFS objects, and invoke the appropriate operations on them.

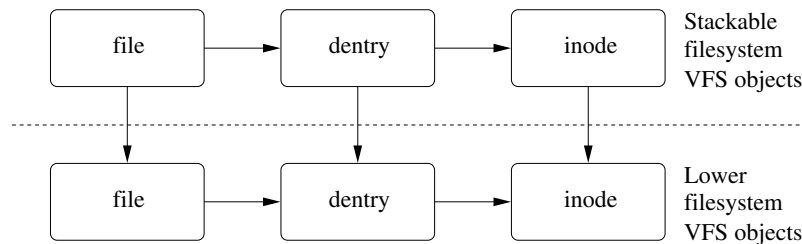


Figure 2.9: Chains of interposed objects for an open file

2.3 Related Work

The idea of an overlay filesystem is not new. Several attempts on various operation systems have been made to implement such a filesystem. The most important of these will be discussed below.

2.3.1 Overlay Filesystem by A. Naseef

This overlay filesystem [12] is implemented as a filesystem module at VFS level. This approach guarantees platform independence as well as a high level of portability across stable kernel versions. The code looks solid, well organized and well documented. This overlay filesystem is currently available in the version 2.0.1, some of the main features are:

- The code is based on 2.4.10 Linux kernel, and according to the author should run on future release with little modification. (Note: compiling on 2.4.20 failed, apparently due to a namespace collision between the `list_t` type from `<linux/list.h>` and the authors `list_t` from `kernel_list.h`)
- The author states having successfully booted from a bootable CD using the overlay filesystem as the root filesystem.

- A storage system has been added. This defines an interface between the overlay filesystem core and the physical storage used by it. Apart from making maintenance of the code easier, the idea is to allow separate development of alternative storage methods, e.g. storing overlay data on a block device instead of memory. This function, which according to the author is planned, would considerably decrease the amount of kernel memory used.

This approach is similar to my implementation, although there are some major differences. Main target of the *mini_fo* implementation are embedded systems, which have special requirements in terms of use of memory and required features. As this overlay filesystem maintains all inode information in memory, overlaying the root filesystem on a embedded system with little RAM might quickly use up all available memory. The reason why this problem does not occur in my implementation is described in section 4.3. Due to this, features such as the storage system are not necessary, and can be left out in order to reduce the code size.

2.3.2 Translucency

The translucency [13] overlay filesystem installs modified system calls which redirect access to files within the directories of the base and overlaid filesystem. Although the code is said to build on current kernels, the major drawback to this approach is that the kernel needs to be compiled for every new kernel and update. Development is more difficult, as in contrast to kernel modules, this overlay filesystem cannot be loaded and unloaded into and from the running kernel. Also many features have not been implemented yet:

- files cannot be virtually deleted (no support for whiteout entries)
- overlaid files are listed twice in a directory.
- symbolic links are followed, but not redirected properly.
- under certain circumstances files can be put in wrong directories.
- the `sys_execve` system call substitute is in x86 assembler which requires porting to each new platform.

2.3.3 FiST and Wrapfs

Wrapfs [7, 8, 11] is a template for developing stackable filesystems with the intent to ease development and increase portability. This is achieved by

interfacing directly with the kernel and providing hooks for the filesystem developer to perform operations on files, attributes and data. After defining the desired operations Wraps is simply mounted on top of existing directories and, transparently for any native filesystems, can add its functionality.

FiST [9, 10, 11] (filesystem translator) takes development of stackable filesystems a level higher by defining a high level language that allows developers to define functionality independently of the target operating system. After that, the FiST compiler *fistgen* is used to generate operating system dependant kernel filesystem modules for several supported operating systems (currently Linux, FreeBSD and Solaris).

In [9] Zadok and Nieh describe several examples, including a overlay like filesystem called unionfs, which merges two branches of a directory together. Unfortunately, the filesystem compiler does not yet support the generation of fan out (see 2.2.2) filesystems. This support would greatly facilitate development.

Nevertheless, I chose a bare *fistgen* generated filesystem as a base for the *mini_fo* overlay filesystem. More details can be found in Chapter 3.

2.3.4 FreeBSD union filesystem

The FreeBSD union filesystem[14] seems to have reached stable quality level and provides all basic features of an overlay filesystem, though less than others mentioned in this document. Unfortunately there is not a lot of documentation on design and implementation available, apart from the well documented source code. The main disadvantage is the considerable effort involved in porting from the FreeBSD VFS to the Linux VFS.

2.3.5 Inheriting File System (IFS)

This is a kernelpatch[15], that is said to be more of a proof of concept. The latest release is from Spring 94 for the ancient version 0.99 of Linux kernel. The author notes that porting to current versions will be probably a lot harder than starting from scratch. Apparently some effort has been made on porting to Linux kernel 2.0, but it does not seem that any code has been released. The project has been discontinued.

Chapter 3

Design

3.1 Requirements

The *mini_fo* overlay filesystem is intended for use in embedded systems. Therefore the requirements are oriented towards this environment, although its use is not limited to it. As mentioned before, this overlay filesystem is implemented by extending the functionality of a bare stackable filesystem[11]. This guarantees platform independency, independency of the lower filesystems and a high level of portability across stable kernel versions. For development kernel 2.4.20 was chosen, but the code should run on future versions with little or no modifications as for the 2.4. kernel series no major changes to the VFS Layer are expected.

In contrast to most desktop PCs, embedded systems have limited amounts of RAM, so one main implementation focus has to be to minimize memory usage. Furthermore the persistent storage available is normally a flash device which imposes further restrictions:

- The amount of information that may be stored on a flash device is limited, therefore storing of unnecessary or redundant data must be avoided.
- Flash devices can only be written to limited times and wear out after a certain amount of cycles. To prolong the life of the device unnecessary write/erase operations need to be avoided.

Testing is to be performed in an embedded system environment as well as on a standard desktop PC.

3.2 System Design

3.2.1 Terminology

Before discussing the design of the *mini_fo* overlay filesystem, some terms need to be defined that will be used in this and following chapters. *Base* and *Storage*¹ are the filesystems that are stacked upon (see figure 3.1). The base filesystem contains the data and is the filesystem that will be overlaid. It is never changed in any way, as all operations that change any data or file attributes are directed to the storage filesystem.

The term *meta data* refers to the data which is stored in the storage filesystem, and that represents the current state of the read-only base filesystem. Any write operation to the overlay filesystem has to result in a change to the storage filesystem, more precisely to the meta data. Obviously there are unlimited ways of storing the meta data, though the manner of storing it plays an important role in an overlay filesystem, as it has large impact on qualitative aspects such as speed, reliability and complexity of the implementation. Therefore a well thought out approach to store meta data is vital.

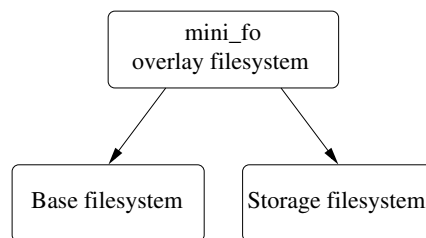


Figure 3.1: Storage and base filesystem

3.2.2 Meta Data

For the *mini_fo* overlay filesystem a hierarchical approach of storing the meta data was chosen. Generally, an empty storage filesystem represents a base filesystem without any modifications, for example after the first mounting the overlay filesystem. When modifying a file for the first time a copy of the base file is created at the corresponding position in the storage filesystem, and from then on all operations are performed on this file; the storage file takes

¹The terms base and storage filesystems have been adopted from the terminology of [12]

precedence over the base file. This is illustrated in the following example. Please note that while base and storage filesystem actually contain the data shown, overlay filesystem shows the virtually merged contents of base and storage that can be seen through its mount point.

overlay filesystem:	base filesystem:	storage filesystem:
/animals/	/animals/	/
/animals/dog	/animals/dog	
/animals/birds/penguin	/animals/birds/penguin	

Figure 3.2: Unmodified overlay filesystem

The base filesystem that has been overlaid contains some data: dog and penguin are regular files, animals and birds are directories. The storage filesystem is empty and contains no meta data, which means the base filesystem is unmodified. So a user viewing the contents of the overlaid base and storage filesystem through the mount point of the overlay filesystem sees the contents of base, as nothing has been changed so far. Now let us assume a user decides to modify the contents of the regular file “penguin”. As soon as the user requests to open “penguin” for writing, the overlay filesystem detects the modifying operation, and performs the following steps:

- Does the corresponding directory of penguin exist in the storage filesystem? -> No, then create² it. After this, figure 3.2 has changed as follows:

overlay filesystem:	base filesystem:	storage filesystem:
/animals/	/animals/	/
/animals/dog	/animals/dog	/animals/birds/
/animals/birds/penguin	/animals/birds/penguin	

Figure 3.3: Overlay filesystem after creating corresponding storage directory

- Create a copy of the file to be modified in the corresponding storage directory.

²This is done by the `build_sto_structure` function of the `mini_fo` overlay filesystem.

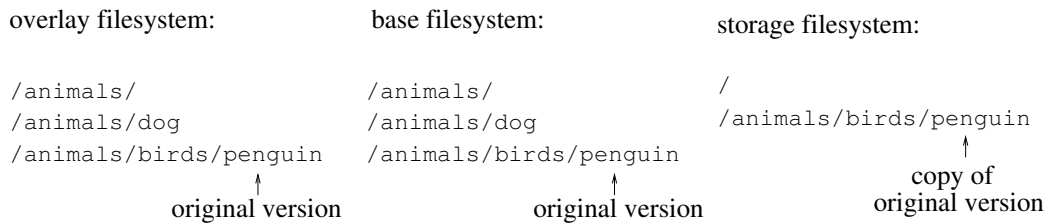


Figure 3.4: Overlay filesystem after copying base file

- Complete the open operation by opening the copy of the original file in storage. Now the user can perform the required modifications to the “penguin” file. When he finishes figure 3.3 has changed again:

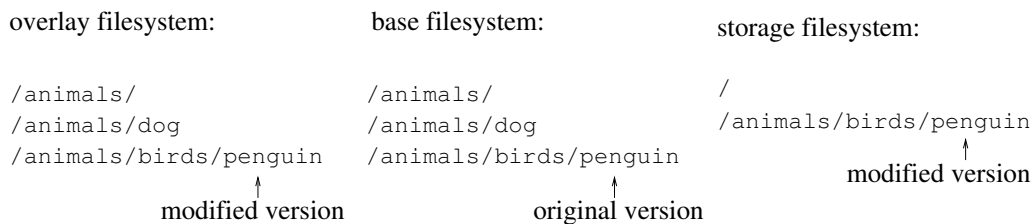


Figure 3.5: Overlay filesystem after modifying penguin

Now two different versions of the regular file “penguin” exist. The original, unmodified version in the base filesystem, and the newer version that has been modified by the user’s write operation in storage. Of course, all access to the penguin file will from now on only yield the new version located in storage, which is indicated by the “modified version” arrow in figure 3.5. Naturally the user performing such operations via the overlay filesystem mount point does not notice these changes, they occur transparently.

There is one point to be considered regarding this manner of storing the meta data. Earlier it was said that if a base file has a storage file in the corresponding directory, the storage file takes precedence over the base file. But what happens if the contents of the directory file “animals” in figure 3.5 is read? As this directory exists both in base and in storage, reading the storage “animals” directory would only yield the “birds” directory but not the “dog” file. To avoid this, the overlay filesystem needs to handle directory files in a special way. If the contents of a directory is read, the overlay filesystem produces the set union of the files in both storage and base directory, while in case of duplicates due to the precedence of storage files the attributes

of the storage file are listed. This way only modified files plus the required directories to represent their position need to be stored. The main advantage of this approach of storing the meta data is describe in section 4.3, *On the spot storage*.

3.2.3 Whiteout entries

The *mini_fo* overlay filesystem allows files to be deleted virtually through the overlay filesystem. To achieve this, so called *whiteout lists* (wol) are used. Whiteout lists exist both in memory as a list associated with a directory inode as well as permanently in the storage filesystem. Whenever a file in a base directory is requested to be removed, a whiteout entry is added to the whiteout list in the corresponding storage directory and to the memory list (see example in figure 3.6 and 3.7). When a lookup operation is performed, the overlay filesystem automatically filters out the files contained in the whiteout list, so they are not listed anymore.

overlay filesystem:	base filesystem:	storage filesystem:
/animals/	/animals/	/
/animals/dog	/animals/dog	
/animals/birds/penguin	/animals/birds/penguin	

Figure 3.6: Unmodified overlay filesystem

overlay filesystem:	base filesystem:	storage filesystem:
/animals/	/animals/	/animals/
/animals/birds/penguin	/animals/dog	/animals/wol-file
	/animals/birds/penguin	

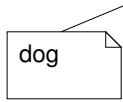


Figure 3.7: Overlay filesystem after deleting “dog”

In practice, this is a little more complicated, because files might exist in a base and/or in the corresponding storage directory. A detailed description of the procedure of deleting a file and the resulting state transitions is given in Appendix A.2.3.

This system of storing deleted files has one disadvantage compared to solutions that separate the meta data from the real data. As the file containing the whiteout list is located in the corresponding storage directory of the base file, the namespace of valid filenames in the overlay filesystem is reduced, as the whiteout list file name may not be used by any other file anymore. But as most current filesystems allow filenames to have a length of up to 255 characters, it should not be hard to choose a name for the whiteout list that will not collide with any names of used files.

In spite of this disadvantage, this way of storing the information about deleted files has several advantages. Firstly, the information can be easily found without having to search through long lists, as it would be required if all information was stored centrally. Distributing the information as such also increases the fault tolerance, as in case of corruption of a file only the information on deleted files in one directory will be lost, instead of all. Secondly, as the whiteout list file is a simple text file containing the entries separated by newline characters, files can be easily “undeleted” by removing their entries from the wol file using a regular text editor. One important advantage of this approach of storing the meta data is described in section 4.3, *On the spot storage*.

3.2.4 Overlay filesystem states

As indicated in the previous sections, a file virtually accessed through the overlay filesystem may be located in the base and / or the storage filesystem, may have been virtually deleted and contained in a whiteout list or might not even exist at all. Hence the VFS objects of the overlay filesystem file, inode and dentry might or might not be interposed (see section 2.2.3 on interposition) on a lower filesystem object. As repeatedly looking up this information when needed is not feasible because too time consuming, each overlay filesystem file is assigned a state. An overview over the different overlay filesystem states and their meanings is given in table A.1 in the appendix. If a file is encountered in a state other than the defined, this will be treated as an error and the overlay filesystem will try to recover the state. If not possible, the operation will be aborted. Generally, invalid states indicate corruption of the meta data or might be encountered if the lower filesystems base and storage are modified while the overlay filesystem is mounted.

The overlay filesystem state of a file will be determined during the lookup operation, and will reside in memory for the lifetime of the VFS object holding the state (see section 4.2). The state will change when the file is modified in a way that affects its storage location or when it is deleted. Overlay

filesystem states are never saved permanently, which allows modification of the contents of the base and storage filesystem directly, while the overlay filesystem is not mounted. This is why the state needs to be determined again each time a file is looked up.

3.2.5 Implementation

As previously mentioned, the implementation of the *mini_fo* overlay filesystem is based on a bare stackable null filesystem generated by the FiST filesystem compiler³. As a stackable null filesystem does exactly what its name says, namely nothing, the following two major tasks were required to implement the overlay filesystem behaviour:

1. Extend the stackable null-filesystem to stack on two lower filesystems, i.e. implement fan out.
2. Modify all stackable filesystem functions to behave according to the rules of an overlay filesystem (e.g. do not allow modification of the base filesystem and structure the meta data in the defined fashion).

Step 1. is achieved by extending regular non fan out interposition as shown in figure 2.9 in order to interpose one stackable layer VFS object on top of *two* lower layer VFS objects. Figure 3.8 shows the VFS objects and their pointers to related objects involved in the example of an open file in the overlay filesystem. Here again, horizontal arrows represent standard VFS pointers that exist independently of the stackable filesystem, while the vertical arrows represent pointers required for interposition. Please note that the figure is to be thought of as a three dimensional picture with the storage and the base filesystem layer being on the same level.

Step 2 is more time consuming, and requires carefully modifying the functions to perform as required. Basically all functions involved in the overlay filesystem can be divided into three groups:

1. *State changing functions* (`write`, `create`, `setxattr`, `mkdir`)
2. *Non-state changing functions* (`read`, `readdir`, `lookup`)
3. *Helper functions* (`mini_fo_cp_cont`, `build_sto_structure`)

³I named this first filesystem *mini_fo* for mini fan out filesystem. This name, deeply incorporated into the source code, has been retained since then.

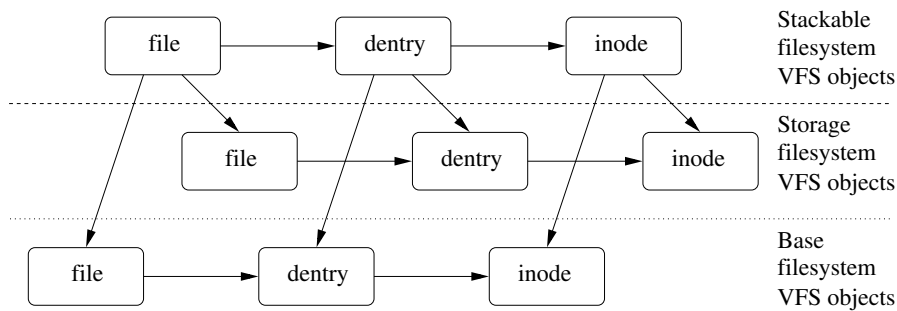


Figure 3.8: Chains of interposed VFS objects for the overlay filesystem

State changing functions are functions that result in a change of the overlay filesystem file state (see previous section 3.2.4), by performing a modifying operation on an overlay file. It is important to note that a function that modifies a file is not necessarily a state changing function. For example, consider repeatedly appending data to an unmodified base file. Only the first append operation is state changing, as it will result in creation of a copy of the base file in storage. All successive appends are no longer state changing, as they only modify the already existing storage file. Therefore they belong to the group of non state changing functions. Helper functions are a special group of functions that perform tasks related to maintaining the meta data. In contrast to the other two groups, these function cannot be called from outside the overlay filesystem.

Generally, each of the two types non- and state changing functions behave according to certain patterns. Figure 3.9 and 3.10 illustrate these.

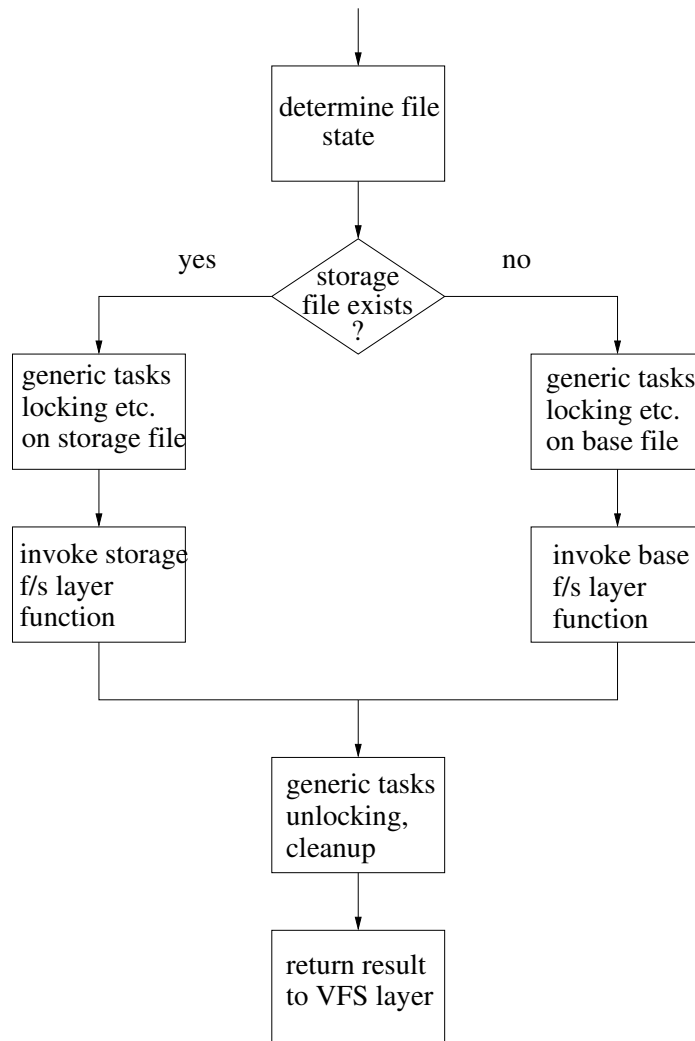


Figure 3.9: Generic flowchart of a non-state changing function

A non state changing function basically only has to determine if the operation needs to be performed on the base or the storage file, depending on which exists and which represents the current state. Then the respective function can be invoked.

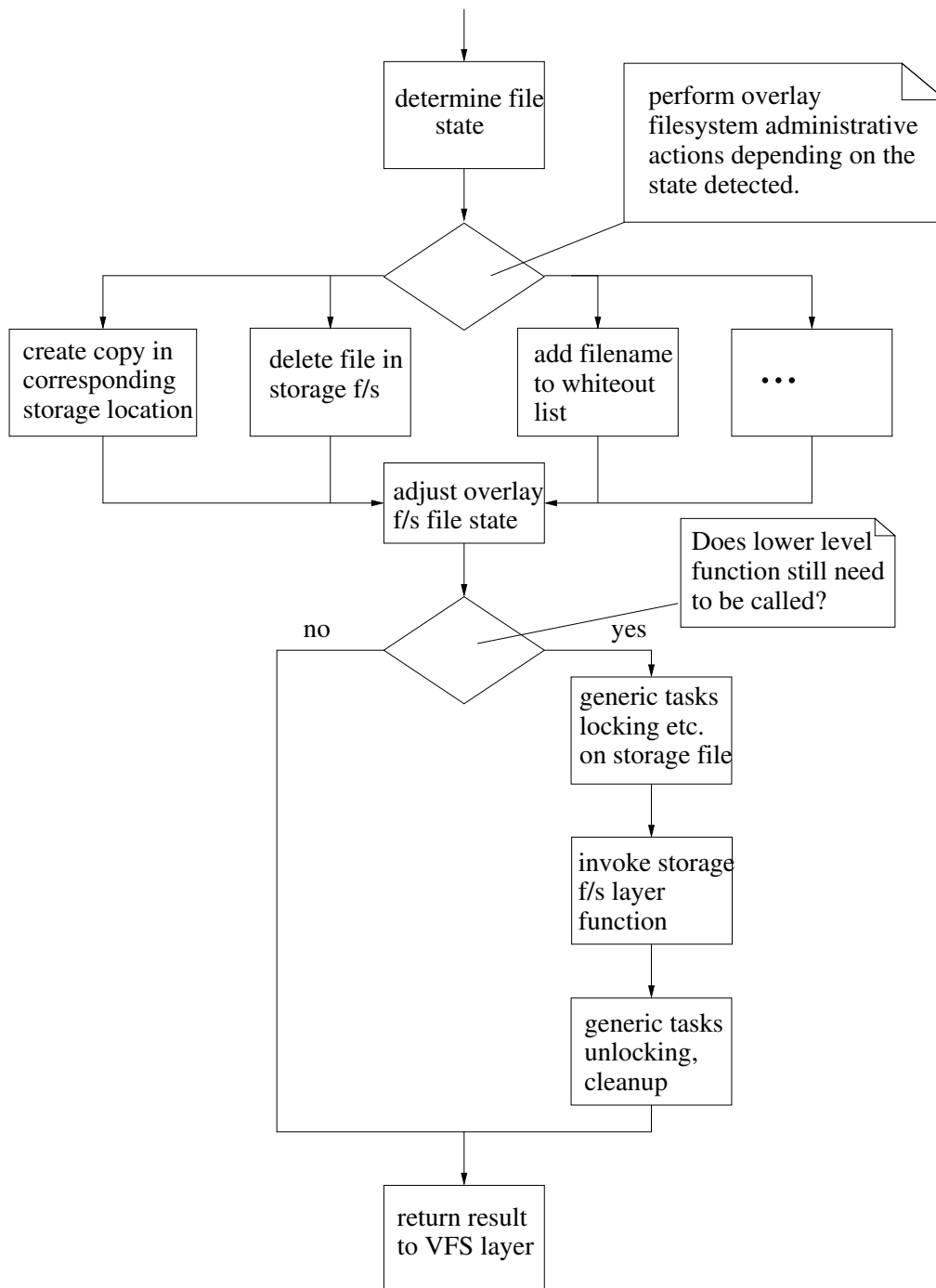


Figure 3.10: Generic flowchart of a state changing function

A state changing function is slightly more complicated, as it requires more work than just identifying the file to invoke the lower function on. Possibly the file that the operation needs to be performed on does not even exist in the right location yet, or in some cases, no lower function needs to be invoked at all. These administrative tasks are the main part of state changing functions, and are the most time consuming operations of the overlay filesystem. As usual, the real world is slightly more complicated, as a function can be state changing and non state changing at the same time, depending on the operation and current state of the file. Also, both flowcharts do not show any error handling and special actions required for directories. For detailed information on this, see appendix A.3.

3.3 Limitations

The current implementation of the *mini_fo* overlay filesystem does not yet implement all possible features, as some functions have turned out to be quite complex. The details on what difficulties were encountered are discussed in 4.1. They are merely listed here for the sake of completeness:

- Directories are not allowed to be renamed yet (see 4.1.3 for details)
- A overlay filesystem mount point cannot be exported with the NFS filesystem (see 4.1.1 for details).
- Opening a file in read-write mode (using the `O_RDWR` flag) will result in a copy of the base file made in storage, independently of it being written to afterwards or not. This will be changed to a *copy on write* strategy in the future.
- The memory whiteout lists as well as the non duplicate list are currently implemented as simple linked lists. This slows down the process of directory reading, especially for large directories containing storage and base files (see 4.1.2)
- Heavy use of hard links will result in a slowdown of state changing functions (see 4.2)

Chapter 4

Implementation Aspects

This chapter discusses various interesting issues encountered during the work on the *mini_fo* overlay filesystem. It is more detailed than the previous chapters and primarily intended for the advanced reader.

4.1 Difficulties

It can clearly be said that stackable filesystems, but especially fan out filesystems (see section 2.2) were not thought of when the VFS was designed. This becomes apparent in many situations. Some of the most interesting will be described in the following section.

4.1.1 Unique inode numbers

As explained in section 2.1.3, an inode number serves to uniquely identify a file within a filesystem. This causes no problems for non fan out stackable filesystem, as each inode of the stackable layer can be assigned the same inode number as its corresponding lower layer inode (see figure 4.1). The inode numbers are mapped one to one. Problems arise for fan out filesystems which stack on top of two or more lower layers. Figure 4.2 illustrates the problem.

The first lower level inode has the number 47, the second number 11. Now which inode number should be used for the inode of the stackable layer? Unfortunately, so far there is no real solution to this problem. For the *mini_fo* overlay filesystem the principle of one to one mapping of inode numbers was abandoned. These are now assigned randomly to the stackable layer, independently of the lower values.

This solution unfortunately has some drawbacks, as due to the random

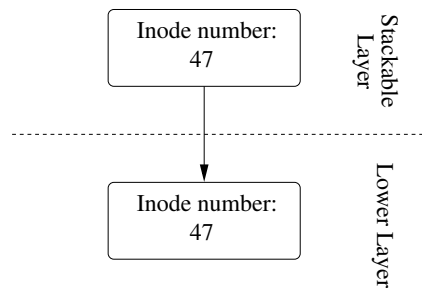


Figure 4.1: One to one inode number mapping

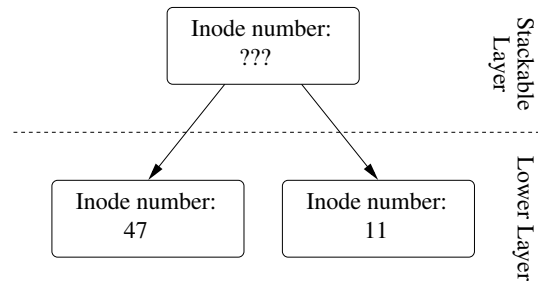


Figure 4.2: Which inode number should be used?

assignment, inode numbers are not persistent and may change over remounts. This affects some applications that rely on persistent inode numbers, as for example NFS. This is why an overlaid filesystem cannot be exported via NFS for now.

4.1.2 Directory reading

Whenever a user types the command `ls`, the contents of the current directory is read. Compared to other VFS functions, the implementation of directory reading is somewhat special. Figure 4.3 shows how this is accomplished.

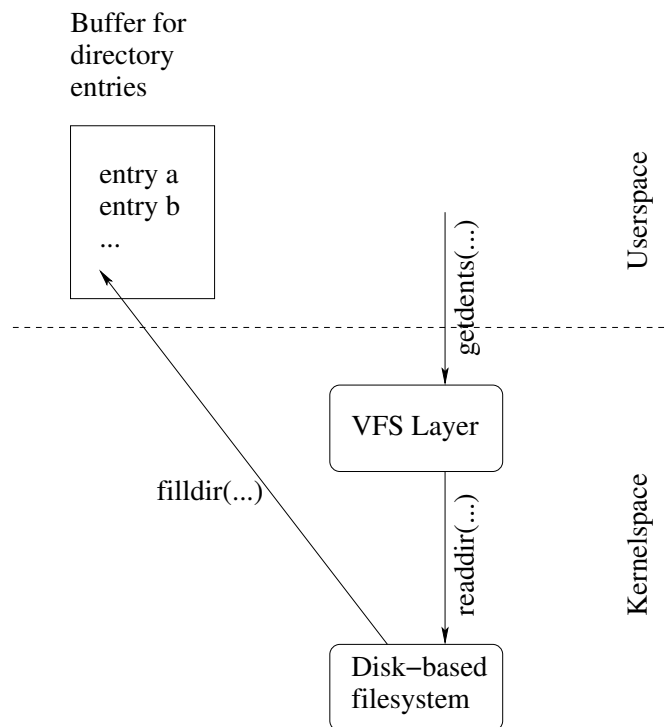


Figure 4.3: Directory reading

The request for directory reading is done via the `getdents(...)` system call, that receives as parameters a file descriptor, a pointer to a memory buffer in which the directory entries are to be copied and the size of the buffer. When the VFS receives this request, it translates the file descriptor to a VFS file object and eventually invokes the `readdir` function of the lower disk-based filesystem. Most interestingly, beside the file object and the pointer to the buffer a third parameter is passed, a function pointer to the `filldir` function. This type of function is called a callback function, as it is called from a lower layer to communicate with a higher layer. In the example the lower disk-based filesystem will start sequentially reading directory entries from disk, and call the `filldir` callback function for each entry with the pointer to the memory buffer and the entry itself as parameters. The `filldir` function will then simply copy each entry to the memory buffer, where the user space program (e.g. `ls(1)`) will expect it after the `getdents` system call returns.

While reversing the call direction in such a way is normally avoided in favor of a hierarchical layered design, this approach seems to be a trade off

between efficiency and beauty of design.

Directory reading in fan out filesystems

Unfortunately, such callback functions cause problems for fan out filesystems that lose control of the flow of information between the lower filesystem layer and the higher VFS level. As explained in section 3.2, the *mini_fo* overlay filesystem needs to create the set union of files in the storage and base filesystem, in order to avoid duplicates. Also virtually deleted files contained in the whiteout list need to be held back. Figure 4.4 shows how this is achieved.

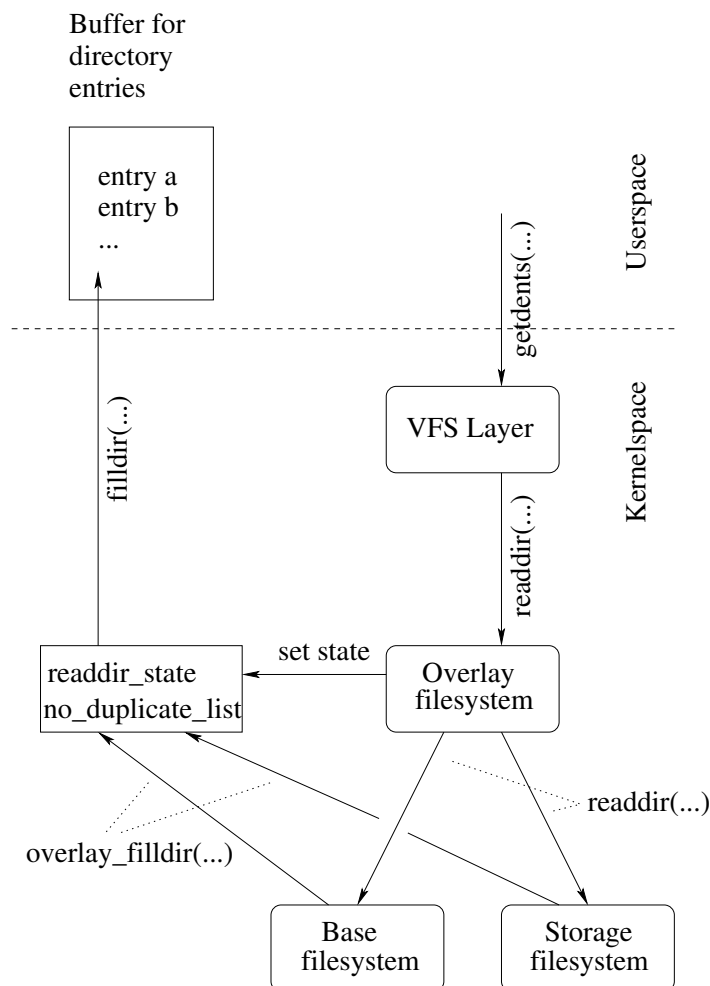


Figure 4.4: Directory reading in the overlay filesystem

As usual, the overlay filesystem has been stacked on top of the standard disk-based filesystem. When a directory needs to be read, the VFS will invoke the `readdir` function of the overlay filesystem, passing the standard `filldir` function pointer. Now, in order to regain control of the data flow between the lower storage and base filesystem and the user buffer, the overlay filesystem will pass its own `overlay_filldir` function. Thus, the lower layers will invoke the `overlay_filldir` function for each directory entry, which can then perform the required modification to the directory entry. When done, the `overlay_filldir` function will then invoke the standard `filldir`, that will finally copy the directory entry to the user space buffer.

Unfortunately, in contrast to the standard `filldir` the `overlay_filldir` function is not stateless as it requires knowledge about the previous directory entries that have been processed, the overlay filesystem state (see section 3.2.4) of a file represented by a directory entry etc. Therefore a global state variable had to be introduced, allowing the `overlay_filldir` function to determine if it is being called by the base or storage filesystem, and keep track of the entries already copied to the buffer. The whole process of overlay filesystem directory reading can be divided into the following steps:

1. *Reading storage directory entries:* the overlay filesystem will repeatedly call the storage `readdir` function until all directory entries have been read. The `overlay_filldir` function detects this phase via the `readdir_state`, and does the following:
 - (a) Check if the directory entry points to a whiteout list file. Exit function as this meta file should not be listed.
 - (b) Add the directory entry to a special non-duplicate list which will be used later to avoid listing duplicated entries.
 - (c) Invoke the standard `filldir` function to copy the data into the user space buffer.
2. *Reading base directory entries:* now that all storage directory entries have been read, the overlay filesystem will repeatedly call the base `readdir` function until all directory entries have been read from base. Again, the `overlay_filldir` function detects this phase via the `readdir_state`, and does the following:
 - (a) Check if file linked with this directory entry has been virtually deleted and therefore contained in the directories whiteout list file. Exit function if yes, as this link will not be listed.

- (b) Check if file is contained in the non duplicate list, if yes exit the function because this means the file has already been copied into the user space buffer and need not be listed again.
- (c) Invoke the standard `filldir` function to copy the data into the user space buffer.

Obviously this implementation is not particularly elegant, but necessary to achieve the required behaviour for the Linux VFS. This example clearly shows the limits of the VFS regarding the use of stackable fan out filesystems.

4.1.3 Directory renaming

Directory renaming is the only operation that turned out to be complicated by the meta data design. Consider the following example:

base filesystem:	storage filesystem:
<code>/animals/birds/penguin</code>	<code>/animals/birds/penguin</code>
<code>/animals/birds/stork</code>	
<code>/animals/birds/pheasant</code>	

Figure 4.5: Directory renaming initial situation

The base filesystem contains three regular files “penguin”, “stork” and “pheasant” whereas the “penguin” file has been modified and therefore the modified version is contained in the corresponding storage filesystem location. Both old and new version can easily be related to each other by their identical position in the directory hierarchy and their identical names.

When such a directory is renamed, this relation is broken, as shown in figure 4.6 where a user, unaware of the flying capabilities of penguins, renamed the directory “birds” to “flyingBeasts”.

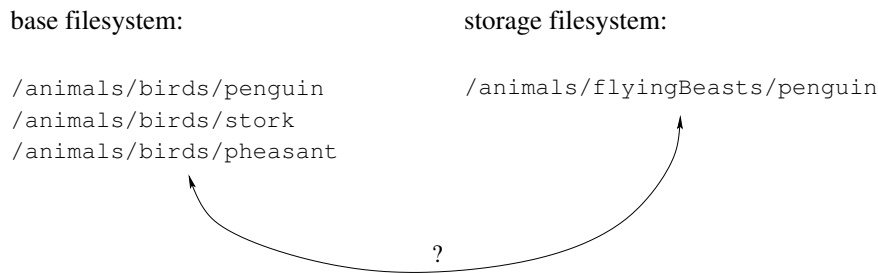


Figure 4.6: Broken relationship base and storage file

The difficulty is that the overlay filesystem now needs to remember the new relationship between the original directory name and the new renamed one. This is impossible without introducing a new overlay state indicating that a directory in storage corresponds to a directory in base with a different name. Also the old directory name needs to be added to the whiteout list, because obviously, after being renamed, it should not appear in directory listings.

While directory renaming could be implemented by introducing a new meta file similar to the whiteout list file in order to permanently store the new relationship, it is likely that a more general approach allowing the storage of multiple overlay filesystem attributes will be chosen. This will make it easier to deal with new overlay filesystem states and attributes that might be required in the future.

4.2 Where to store the overlay filesystem file state?

The overlay filesystem uses file states in order to manage the different combinations of lower files associated with an overlay filesystem file (see section 3.2.4). Basically there are two locations where this information can be stored: the dentry or the inode object. Currently, this information is held in the dentry object, which seems to be the right choice, because as can be seen in table A.1 there exist states where no overlay filesystem inodes are involved at all.

Nevertheless, using the dentry for this has one main disadvantage. Due to the principle of hard links there can be more than one dentry object associated with one inode. These all need to be kept in sync. This means, state changing functions are required to update the state and the pointers of all other dentries that point to the same inode. In systems that make heavy

use of hard links, this could be time consuming. As this is not the case for embedded systems, this approach is acceptable. A possible alternative would be to use the inode to store the state when possible, otherwise the dentry object. This will be considered for future releases.

4.3 “On the spot” storage

The VFS makes use of caches in order to speed up its operation. For filesystem operations this is mainly the dentry cache, called `dcache`. As the name says, the `dcache` caches VFS dentry objects with the intent to save the time required in reading a directory entry from disk. Each dentry is in one of four states [4]:

1. *In use*: The dentry is used by the VFS, the usage counter `d_count` field is greater than zero and the dentry is associated with an inode object. An in use dentry cannot be discarded.
2. *Unused*: The dentry object is currently not in use by the VFS, the usage counter `d_count` field is zero but the dentry is still associated with an inode and contains valid information. If necessary, it may be discarded to reclaim memory.
3. *Negative*: The dentry object is not associated with an inode, either because it resolves a pathname to a non existing file or because the corresponding disk inode has been deleted. Nevertheless, the dentry object stays in the `dcache` in order to quickly resolve further lookup operations to the same (non existing) file.
4. *Free*: The dentry object contains no valid information and is not in use by the VFS.

Caching of inode objects is done indirectly by the `dcache`, as inodes that are associated with unused dentries are not discarded and need not be read from disk in future lookup operations. By this mechanism the VFS makes optimal use of available memory to increase speed while preserving the possibility of regaining the memory not directly used if necessary.

The *mini_fo* overlay filesystem adopts itself to this manner through the way it stores the meta data in memory. As described in 3.2.2, the on disk meta data such as new files and whiteout lists are structured in a distributed manner in storage directories corresponding to base directories. In memory this is basically the same, all meta information about an overlay filesystem

VFS object is maintained together with it. For example each overlay filesystem directory inode has the whiteout list of virtually deleted files within the directory attached to it. As long as the inode object is valid in memory, the whiteout list attached to it will stay as well. If the inode is discarded, e.g. because memory needs to be regained, the whiteout list will be released as well. This way, the *mini_fo* overlay filesystem never holds any superfluous data in memory which cannot be freed if required. Because the meta data is stored where it is needed, it is said to be stored *on the spot*, respectively the storage is called *on the spot storage*.

4.4 Testing

Testing of the *mini_fo* overlay filesystem was performed in two steps. First the implemented functionality was tested with test cases crafted for this purpose. These test cases mainly consist of standard filesystem operations like creating, deleting, modifying, renaming or changing the attributes of files. While some bugs were found during this test, the second step that was the compilation of a 2.4 Linux kernel on top of the mounted overlay filesystem revealed many more issues such as new bugs and unimplemented but required functions. After fixing these issues the kernel compilation succeeded and as proof of success the resulting kernel could be booted.

Chapter 5

Performance

5.1 Kernel compilation¹

The compilation of a Linux 2.4 kernel on top of a mounted overlay filesystem not only served as a test case, but also as a benchmark. This was performed as follows. The directory containing the clean kernel sources was used as the base filesystem, an empty directory as storage. After the overlay filesystem was mounted, a configuration file was copied into the overlay filesystem. Then the standard 2.4 kernel compilation steps `make oldconfig`, `make dep` and finally `make bzImage` were executed. The time was measured using the `time(1)` command. Table 5.1 shows the results of compiling on top of the overlay filesystem opposed to a regular compilation directly in the source.

	make oldconfig	make dep	make bzImage
Kernel Com- pile without overlay f/s	real 0m5.665s user 0m0.760s sys 0m0.460s	real 0m37.090s user 0m2.810s sys 0m0.710s	real 4m10.849s user 3m8.710s sys 0m8.530s
Kernel Compi- le on top of overlay f/s	real 0m5.493s user 0m0.830s sys 0m0.320s	real 0m37.259s user 0m2.900s sys 0m0.730s	real 4m7.471s user 3m7.260s sys 0m10.080s

Table 5.1: Kernel compilation mini benchmark

The decisive factor is the system time (sys), that measures the time spent in kernel, which is mainly filesystem operations. This is illustrated in figure

¹This benchmark was performed on a laptop with an Intel 1.6 GHz Centrino Processor and 512 MB RAM.

5.1. While the commands `make oldconfig` and `make dep` take too short a time to produce reliable results, `make bzImage` shows only a small advantage for the non-overlaid environment. This may at first seem surprising, but is actually not exceptional when considering the type of operations primarily involved in kernel compilation:

- reading existing source files
- creating new object files

None of these are operations that are time consuming for the overlay filesystem, in contrast to operations such as modifying existing files (a copy of the original file is created) or reading the contents of large directories containing many modified files (filtering of base directory entries). These time consuming operations were measured with a special benchmark described in the next section.

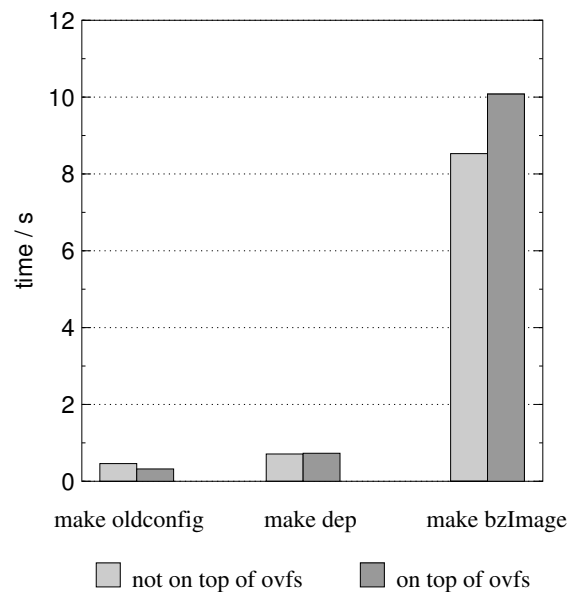


Figure 5.1: System times for each step of the kernel compilation

5.2 Time consuming operations²

This section describes two small benchmarks that measure the performance of two time consuming operations of the overlay filesystem, modifying files and reading the directory contents. The benchmark scripts can be found in appendix A.4.

The append benchmark

The append benchmark modifies existing files of different sizes by appending a small amount of data to them. While a non overlay filesystem only needs to append the data to the end of the existing data, the overlay filesystem makes a copy of the file before appending. Depending on the file size this can take some time.

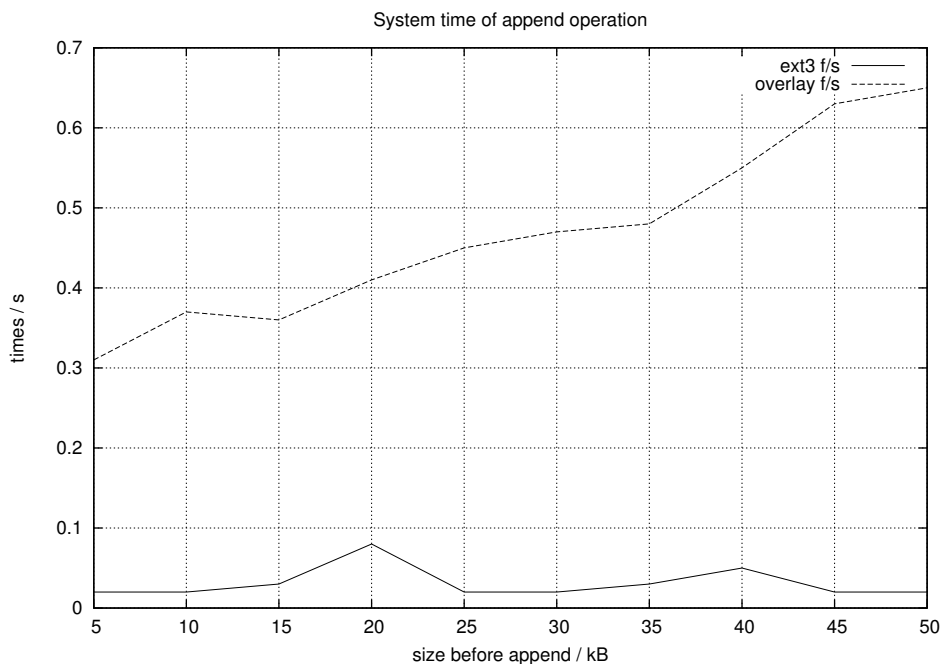


Figure 5.2: System times of append benchmark

Figure 5.2 clearly shows this effect when appending data to 1000 files of different sizes. While the time required for appending on a standard ext3 disk-based filesystem stays constant when increasing the original file size, for

²This benchmark was performed on the same machine as 5.1, but the Intel Centrino Processor was clocked at 600 MHz using the SpeedStep function.

the overlay filesystem it constantly increases due to the larger amount of data that is copied.

The readdir benchmark

Directory reading also belongs to the more time consuming operations, mainly because of the merging of the base and storage files and the filtering of virtually deleted files (see 4.1.2). This benchmark measures the time required for reading the contents of a directory containing a varying number of files. For the overlay filesystem half of the number of files was created in the storage directory, half in the corresponding base directory. In this way the overlay filesystem was stressed by having to merge a large amount of files. Figure 5.3 shows the results.

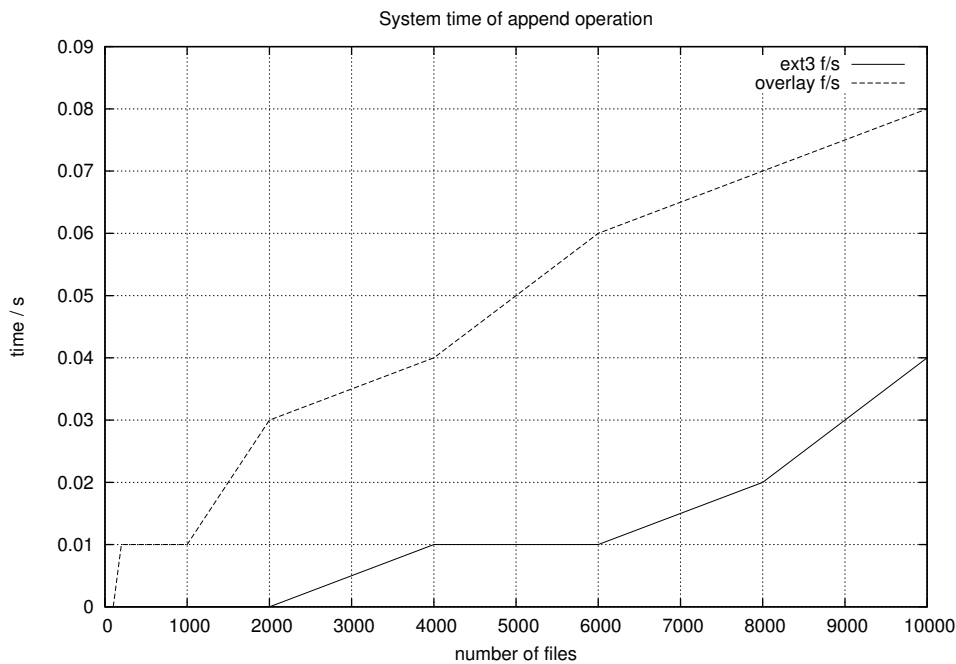


Figure 5.3: System times of readdir benchmark

It is worth noting that the implementation of the *mini_fo* readdir function is currently not implemented in a very efficient way, as linked lists are used for remembering the storage directory entries (again see 4.1.2). Doubtless the speed of directory reading could be greatly improved by doing this more efficiently, for example by using a hash table.

Chapter 6

Conclusion

The *mini_fo* overlay filesystem is a stackable fan out filesystem that stacks on top of two lower filesystems, namely base and storage. Base contains the user data, and can be a read-only filesystem, while storage contains meta data and is always required to be writable. Although being read-only, by accessing the base filesystem through the overlay filesystem base files can be virtually modified, deleted or new ones can be created. The original base files are never modified, as all changes are redirected to the storage filesystem. This is achieved by redirecting function calls, merging data and copying files between the two filesystems.

The *mini_fo* overlay filesystem has become a light-weight solution to efficiently make read-only filesystems writable. The implementation as a stackable filesystem kernel module provides a modular non intrusive way to achieve this functionality, while combining the benefits of stackable filesystems such as portability with the performance of kernel filesystems. Thus, no changes to existing kernel code, neither to the VFS layer nor to the native filesystem layer are required, as kernel modules can be loaded and unloaded into the running kernel whenever their functionality is required.

At the time of writing, the *mini_fo* overlay filesystem is already being deployed in its area of main focus, for overlaying a read-only root filesystem of an embedded system in order to easily allow software updates.

However, its use is not limited to that area. Overlay filesystems have proved to be useful in other areas as well. For instance it can be used for sandboxing by giving applications access to critical data through the overlay filesystem while not running the risk of it being corrupted. By examining the meta data it is easily possible to track which parts of data were modified. Patches to source trees can be quickly applied for testing, while allowing to keep the original state in case something goes wrong.

An other area of deployment could be so called live CD's such as Knoppix[16]

or FreeSBIE[17], that are becoming more and more popular. These bootable CD's contain complete operating systems with many features that are loaded entirely from CD and do not require making any changes to the hard disk. The disadvantage is that all changes that are made to the configuration and all files that are created in the RAM filesystem are lost each time the system is rebooted. By overlaying this non-permanent filesystem and using a directory of an existing filesystem as storage filesystem, this could be avoided and the state of the operating system could be preserved over reboots.

The *mini_fo* overlay filesystem will be continued to be developed, as not all features are implemented yet. Although the current overall performance is very acceptable, there remains room for improvement.

Appendix A

Appendix

A.1 Mount Syntax

For instructions on configuring and compiling the *mini_fo* overlay filesystem, please refer to the instructions in the README file contained in the sources.

The general mount syntax is:

```
mount -t mini_fo -o dir=<base directory>
                        dir2=<storage directory>
                        <base directory>
                        <mount directory>
```

Debugging messages can be turned on by passing the option `debug=N` after the `dir2` option, whereas `N` is the debug level ranging from 0 (no information) to 18 (a lot of information). Check `print.c` for details.

Alternatively the debug level can be set using the `fist_ioctl` programme, which is compiled together with the *mini_fo* module. To turn off all debugging: `fist_ioctl -d <mount dir> 0`, to turn on all debugging: `fist_ioctl -d <mount dir> 18`.

A.2 Implementation Details

A.2.1 Overlay Filesystem file states

Each file in the overlay filesystem has to be in one of the states listed in table A.1. Any file encountered with a different combination of existing base and storage file and WOL entry will be treated as an error.

State number (description)	which files exist?	File in WOL?	Dentry interposition	Inode interposition
1 (modified)	base AND storage	no		
2 (unmodified)	base AND NOT storage	no		
3 (created)	storage AND NOT base	no		
4 (deleted and rewritten)	storage AND base	yes		
5 (deleted)	base AND NOT storage	yes		no interposing
6 (does not exist)	-	no		no interposing

Table A.1: Overlay filesystem file states

Please note: state 4 seems to represent the same state as state 1. For non-directories this is true, but for directories it makes a difference if the base file has been deleted or not, because we need to know if we have to merge the contents of the base and storage or only the contents of the storage directory.

A.2.2 Overlay filesystem file object states

The following table describes the VFS file object interposition states corresponding to the general overlay filesystem states above. It assumes that the overlay filesystem file has been opened.

	Storage file exists	Storage file does not exist	File is marked deleted in storage filesystem.
Base file exists	<p><i>File is directory:</i> overlay filesystem file object is interposed on base and storage file objects.</p> <p><i>File is regular file:</i> overlay filesystem file is interposed only on storage file.</p>	Overlay filesystem file object is interposed only on base filesystem file object.	Overlay filesystem inode does not exist. The overlay filesystem inode of the directory holds the list of deleted files (wol) in the directory.
Base file does not exist	Overlay filesystem file object is interposed only on storage filesystem file object.	The file does not exist and therefore cannot be opened.	invalid state

Table A.2: Overlay filesystem file object states

A.2.3 Deletion process

The following table shows the state transition of an overlay filesystem file when it is deleted:

old state	new state	actions
1 (modified)	4 (deleted)	storage file is deleted, filename is added to wol
2 (unmodified)	4 (deleted)	filename is added to wol
3 (created)	4 (deleted)	storage file is deleted
4 (deleted and rewritten)	4 (deleted)	storage file is deleted
5 (deleted)	error	impossible to delete file as it does not exist in overlay f/s
6 (does not exist)	error	impossible to delete file as it does not exist in overlay f/s

Table A.3: State transitions when deleting a file

A.2.4 Creation process

The following table shows the state transition of an overlay filesystem file when it is created:

old state	new state	actions
1 (modified)	error	an existing file cannot be created
2 (unmodified)	error	an existing file cannot be created
3 (created)	error	an existing file cannot be created
4 (deleted and rewritten)	error	an existing file cannot be created
5 (deleted)	4 (deleted and rewritten)	interpose new inode and dentry
6 (does not exist)	3 (created)	interpose new inode and dentry

Table A.4: State transitions when creating a file

A.3 Flowcharts

A.3.1 The `build_sto_structure` helper function

This function serves to construct a copy of a directory structure of the base in the storage filesystem. The function is given a directory of the base filesystem as parameter, which indicates the last directory of a hierarchy to be created in storage. This is done by a recursive function that steps back through the directory tree until it encounters a directory with the overlay filesystem state *modified*, which means that both a storage and a base directory exist and that both are valid. The function then returns creating the corresponding base directory in storage. Flowchart A.1 illustrates this.

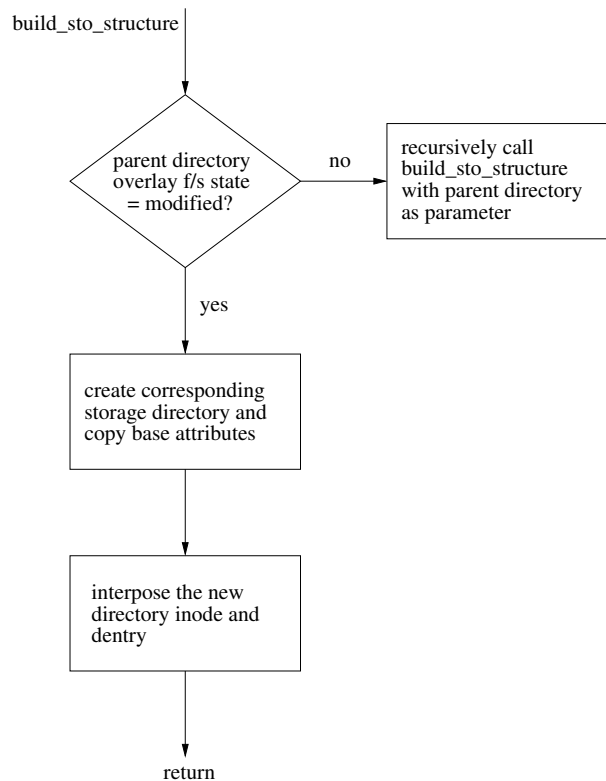
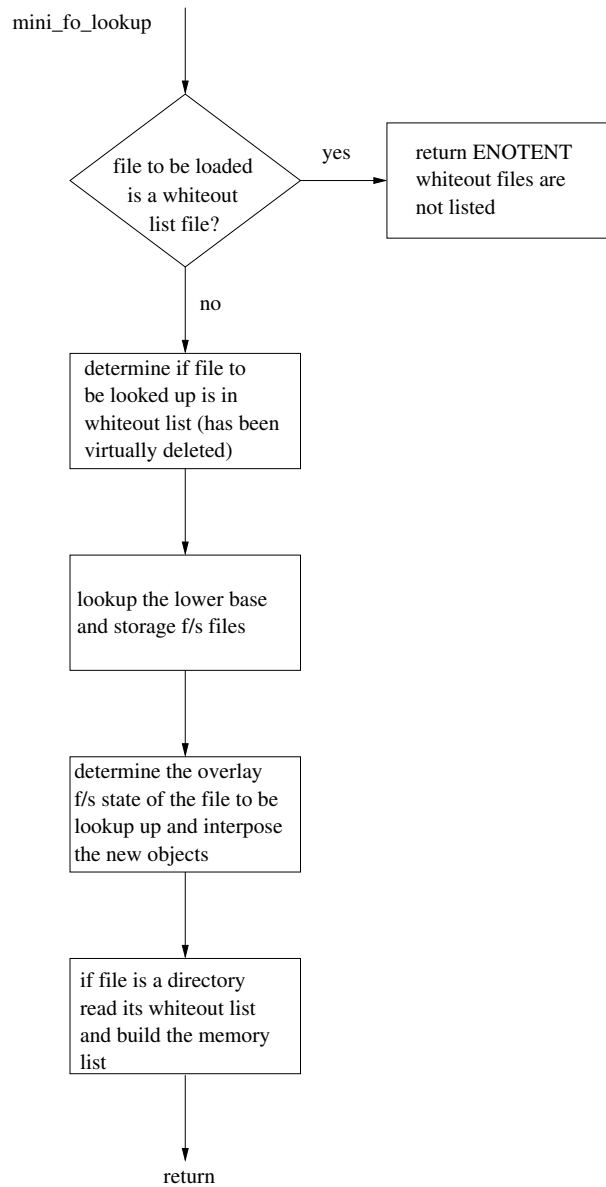


Figure A.1: Flowchart of the `build_sto_structure` function

It is worth noting that using recursion in the kernel is somewhat risky, as a deep recursion could overflow the kernel stack. Nevertheless this approach was chosen for now, as usually no deep directory hierarchies are found in embedded systems and the implementation is cleaner.

A.3.2 The mini_fo_lookup function

Figure A.2: Flowchart of the `mini_fo_lookup` function

A.3.3 The mini_fo_open function

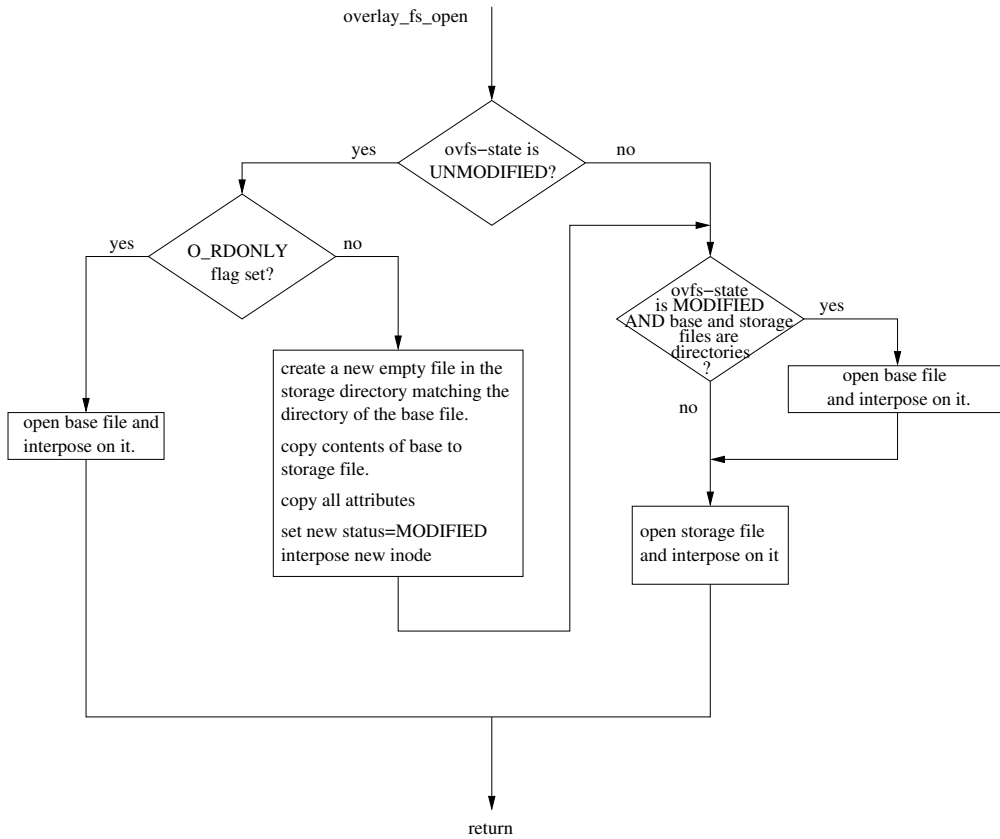
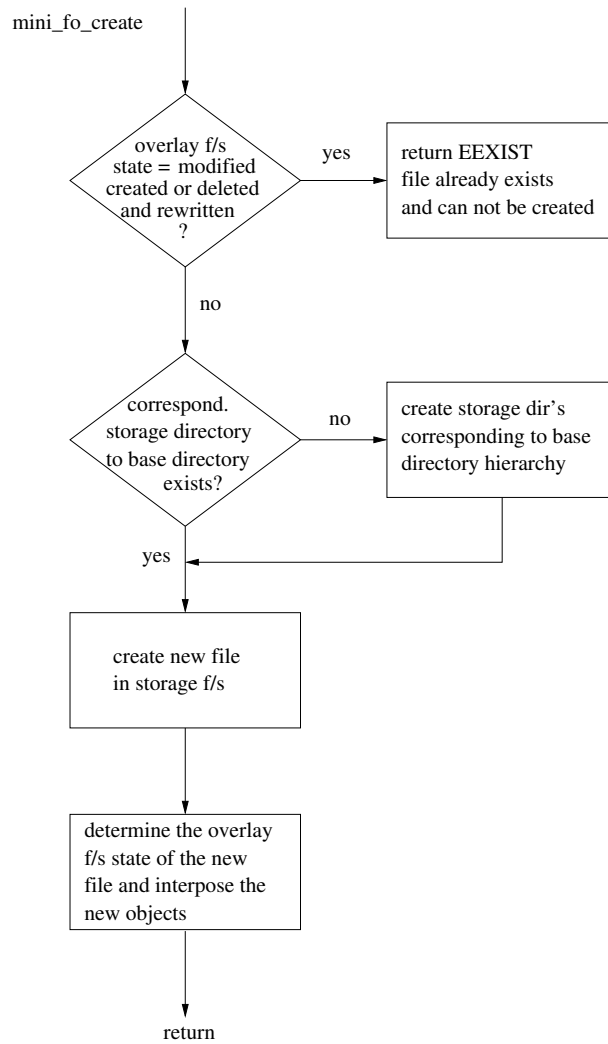


Figure A.3: Flowchart of the mini_fo_open function

A.3.4 The mini_fo_create function

Figure A.4: Flowchart of the `mini_fo_create` function

A.3.5 The mini_fo_setattr function

This function is called when attributes of a file are changed. Depending on the attributes that are asked to be changed, a copy of the original file is created and the attributes are changed on it. Changes to time attributes as access times are ignored, as this would cause many base files to be copied to storage just because they were read.

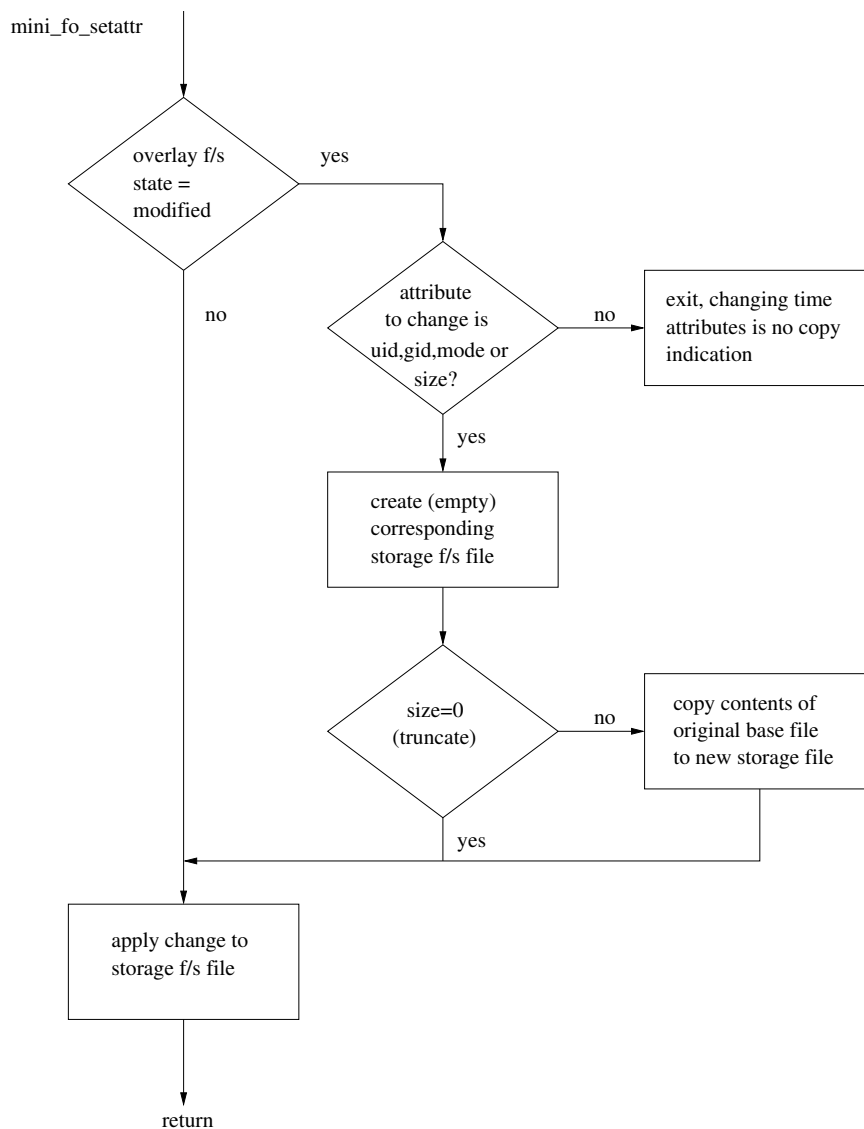


Figure A.5: Flowchart of the `mini_fo_settattr` function

A.4 Mini Benchmark Shell Scripts

A.4.1 Auxiliary scripts

These two bash scripts serve as subscripts for the mini benchmark scripts. *create_test_file* creates test files of a given size in an existing directory. *do_append* appends a small amount of data to all files in a given directory, and is invoked by the benchmark scripts to measure its time.

`create_test_file`

```
#!/bin/bash
#-----
# create_test_file <directory> <number of files> <files size>
# create the testfiles with required size in the existing
# directory passed.
#-----

if [[ $# -ne 3 ]]; then
cat<<EOF

error, wrong umber of parameters passed: $#.

usage: create_test_file <directory> <number of files> <files si\
ze (blocks)>

EOF
    exit -1;
else
    test_dir=$1
    num_files=$2
    file_size=$3
fi

for i in $(seq 1 $num_files)
do
    dd count=$file_size if=/dev/zero of="$test_dir/testfile$i" &> /\
dev/null
done

exit 0
```

do_append

```
#!/bin/bash
#-----
# do_append <directory>
#
# This script appends a small amount of data to all files
# in directory.
#-----

if [[ $# -lt 1 ]]
then
    echo "$0: wrong number of parameters: $#."
    exit -1;
else
    testdir=$1
fi

for i in $testdir/*
do
    echo "APPEND DATA" >> "$i"
done

exit 0
```

A.4.2 Append benchmark scripts

do_app_bench and *do_ovfs_app_bench* are the two scripts that perform the append benchmark. Parameters are number of files, file size and test directory. After creating the test files of the given size in the given directory, the *do_append* is called and the time is measured. *do_ovfs_app_bench* performs identically, but mounts the overlay filesystem on top of the test directory before calling *do_append* which measures the overlay filesystems append time.

do_app_bench

```
#!/bin/bash
#-----
# do_app_bench <test directory> <number of files>
#           <files size>
```

```
#
# Perform a generic append benchmarks, without the overlay f/s.
#-----
bench_dir="do_app_bench_dir-$(date +%s)"
result_file="results_do_app_bench.txt"

if [[ $# -ne 3 ]]; then
cat << EOF

do_app_bench: tests the append speed of a filesystem.

    usage: do_app_bench <directory> <number of files> <file size>
>

EOF
exit -1;

else
    test_dir=$1
    num_files=$2
    file_size=$3
fi

echo -n "Creating test files in $test_dir/$bench_dir/..."

mkdir "$test_dir/$bench_dir" || { echo "Error creating test direc\
tory"; exit -1; }
./create_test_file "$test_dir/$bench_dir" $num_files $file_size

echo "done."

cat>>$result_file<<EOF
-----
append test ('date')
Test directory:  $test_dir
Number of files: $num_files
File size:      $file_size
EOF
echo -n "Starting append test..."
{ time ./do_append "$test_dir/$bench_dir"; } 2>> $result_file
echo "done."
```

```
echo -n "Cleaning up..."
rm -rf "$test_dir/$bench_dir"
echo "done."

exit 0

do_ovfs_app_bench

#!/bin/bash
#-----
# do_ovfs_app_bench <test directory> <number of files>
#                   <files size>
#
# Perform the append benchmark on the overlay f/s.
#-----
sto_dir="sto_dir-$(date +%s)"
base_dir="base_dir-$(date +%s)"
ovfs_dir="ovfs_dir-$(date +%s)"

result_file="results_ovfs_do_app_bench.txt"

if [[ $# -ne 3 ]]; then
cat << EOF

do_ovfs_app_bench: tests the append speed of a filesystem.

    usage: do_ovfs_app_bench <directory> <number of files> <file\
size>

EOF
exit -1;

else
    test_dir=$1
    num_files=$2
    file_size=$3
fi

echo -n "Creating test files in $test_dir/$base_dir/..."
mkdir "$test_dir/$ovfs_dir" || { echo "Error creating ovfs mount \
```

```

point"; exit -1; }
mkdir "$test_dir/$base_dir" || { echo "Error creating base dir"; \
exit -1; }
mkdir "$test_dir/$sto_dir" || { echo "Error creating storage dir"\
; exit -1; }
./create_test_file "$test_dir/$base_dir" $num_files $file_size

echo "done."
echo "Mounting the overlay f/s with command:"
echo "mount -t mini_fo -o dir=\"$test_dir/$base_dir\",dir2=\"$test_d\
ir/$sto_dir\" \"$test_dir/$base_dir\" \"$test_dir/$ovfs_dir\""

mount -t mini_fo -o dir=\"$test_dir/$base_dir\",dir2=\"$test_dir/$st\
o_dir\" \"$test_dir/$base_dir\" \"$test_dir/$ovfs_dir\" || { echo "Err\
or mounting ovfs"; exit -1; }
echo "done"

cat>>$result_file<<EOF
-----
ovfs append test ('date')
Test directory:  $test_dir
Number of files: $num_files
File size:      $file_size
EOF
echo -n "Starting append test..."
{ time ./do_append "$test_dir/$ovfs_dir"; } 2>> $result_file
echo "done."

echo -n "Cleaning up..."
umount "$test_dir/$ovfs_dir"
rm -rf "$test_dir/$base_dir"
rm -rf "$test_dir/$sto_dir"
rm -rf "$test_dir/$ovfs_dir"
echo "done."

exit 0

```

A.4.3 Readdir benchmark scripts

do_rdir_bench and *do_ovfs_rdir_bench* are the two scripts that perform the readdir benchmark with and without the overlay filesystem respectively.

The only parameters are number of files and test directory. After creating the test files the time measured for reading the contents of the directory. *do_ovfs_rdir_bench*, doing the overlay filesystem benchmark creates the half of the test files in the storage filesystem, the other half in the base filesystem.

do_rdir_bench

```
#!/bin/bash
#-----
# do_rdir_bench <test directory> <number of files>
#
# Perform a generic readdir benchmarks, without use of the
# overlay f/s. Measure time required to read contents of
# a directory containing <number of files>.
#-----
bench_dir="do_rdir_bench_dir"
result_file="results_do_rdir_bench.txt"

if [[ $# -ne 2 ]]; then
cat << EOF

do_rdir_bench: tests the directory reading speed of a filesystem.

    usage:  do_rdir_bench <directory> <number of files>

EOF
exit -1;

else
    test_dir=$1
    num_files=$2
fi

echo -n "Creating test files in $test_dir/$bench_dir/..."

mkdir "$test_dir/$bench_dir" || { echo "Error creating test direc\
tory"; exit -1; }
./create_test_file "$test_dir/$bench_dir" $num_files 1

echo "done."
```

```
cat>>$result_file<<EOF
-----
readdir test ('date')
Test directory:  $test_dir
Number of files: $num_files
EOF
echo -n "Starting readdir test..."
{ time ls "$test_dir/$bench_dir"; } 2>> $result_file >/dev/null
echo "done."

echo -n "Cleaning up..."
rm -rf "$test_dir/$bench_dir"
echo "done."

exit 0

do_ovfs_rdir_bench

#!/bin/bash
#-----
# do_rdir_bench <test directory> <number of files>
#
# Perform the readdir benchmark, on top of the overlay f/s.
# Measures the time required to read a directory containing
# <number of files> files.
#-----
sto_dir="sto_dir-$(date +%s)"
base_dir="base_dir-$(date +%s)"
ovfs_dir="ovfs_dir-$(date +%s)"

result_file="results_ovfs_gen_rdir_bench.txt"

if [[ $# -ne 2 ]]; then
cat << EOF

do_rdir_bench: tests the directory reading speed of a filesystem\
.

usage: do_rdir_bench <directory> <number of files>

EOF
```

```

exit -1;

else
    test_dir=$1
    num_files=$2
    num_files_each='dc -e "$2 2 / n"'
fi

echo -n "Creating test files in $test_dir/$base_dir/..."
mkdir "$test_dir/$ovfs_dir" || { echo "Error creating ovfs mount \
point"; exit -1; }
mkdir "$test_dir/$base_dir" || { echo "Error creating base dir"; \
exit -1; }
mkdir "$test_dir/$sto_dir" || { echo "Error creating storage dir"\
; exit -1; }
./create_test_file "$test_dir/$base_dir" $num_files_each 1
./create_test_file "$test_dir/$sto_dir" $num_files_each 1
echo "done."

echo "Mounting the overlay f/s with command:"
echo "mount -t mini_fo -o dir=\"$test_dir/$base_dir\",dir2=\"$test_d\
ir/$sto_dir\" \"$test_dir/$base_dir\" \"$test_dir/$ovfs_dir\""

mount -t mini_fo -o dir=\"$test_dir/$base_dir\",dir2=\"$test_dir/$st\
o_dir\" \"$test_dir/\
$base_dir\" \"$test_dir/$ovfs_dir\" || { echo "Error mounting ovfs";\
    exit -1; }
echo "done"

cat>>$result_file<<EOF
-----
ovfs readdir test ('date')
Test directory:  $test_dir
Number of files:  $num_files
EOF

echo -n "Starting readdir test..."
{ time ls "$test_dir/$ovfs_dir"; } 2>> $result_file >/dev/null
echo "done."

echo -n "Cleaning up..."

```

```
umount "$test_dir/$ovfs_dir"  
rm -rf "$test_dir/$base_dir"  
rm -rf "$test_dir/$sto_dir"  
rm -rf "$test_dir/$ovfs_dir"  
echo "done."
```

```
exit 0
```

A.5 Acknowledgment

I would like to thank everybody who helped me in completing this diploma thesis, in particular Wolfgang Denk from Denx Software Engineering for the highly interesting and up to date subject and the constant support, Prof. Dr. Bittel for supervision on the University side, Detlev Zundel for all the Linux and Emacs tips, the people on the fist mailing list, especially Erez Zadok for answering fan out questions, my parents Angela and Kurt Klotzbücher for eliminating many comma mistakes and proof reading the thesis, and of course Shirley for waking me up at eight thirty to start working.

Many thanks thank to all friends in Constance for the good times we had during the last four years, may there be more to come.

Last but not least I would like to thank all authors of free software for their excellent work. This thesis paper has been written and designed entirely using free software.

Bibliography

- [1] Wikipedia, The Free Encyclopedia. www.wikipedia.org
- [2] The GNU Project. www.gnu.org
- [3] The Linux Kernel Archives. www.linux.org
- [4] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel (2nd ed.)*. O'Reilly & Associates, Inc., 2003.
- [5] A. Rubini, J. Corbet. *Linux Device Drivers (2nd. ed)*. O'Reilly & Associates, Inc., 2001.
- [6] W. R. Stevens. *Advanced Programming in the UNIX Environment*. Addison-Wesley, Inc., 1992.
- [7] E. Zadok and I. Badulescu. *A Stackable Filesystem Interface for Linux*. LinuxExpo Conference Proceedings, May 1999.
- [8] E. Zadok, I. Badulescu, A. Shender. *Extending File Systems using Stackable Templates*. Proceedings of the Annual USENIX Technical Conference, June 1999.
- [9] E. Zadok and J. Nieh. *Fist: A Language for Stackable File Systems*. Proceedings of the Annual USENIX Technical Conference, June 2000.
- [10] E. Zadok, *FiST: A System for Stackable-Filesystem Code Generation*, P.h.d.Thesis, 2001.
- [11] E. Zadok, FiST and Wrapfs software www.filesystems.org
- [12] A. Naseef, Overlay filesystem: <http://ovlfs.sourceforge.net/>
- [13] Translucency overlay filesystem <http://translucency.sourceforge.net>
- [14] FreeBSD union filesystem www.freebsd.org

- [15] W. Almesberger, Inheriting filesystem <http://www.almesberger.net/epfl/ifs.html>
- [16] Knoppix live CD www.knoppix.org
- [17] FreeSBIE live CD www.freesbie.org
- [18] The *mini_fo* overlay filesystem is available at [ftp.denx.de](ftp://ftp.denx.de)