A Review of File System Design Methods and the Design and Implementation of a Smartcard File System

by
Eelco van der Werff

Supervised by
Dr. ir. J.A.G. Nijhuis

Dept. of Computer Science
University of Groningen
Groningen, the Netherlands
January 2004

RuG

Abstract

Like almost any other computing device, smartcards are becoming more powerful. This increase in computing power and storage capacity has caused a trend towards multi-application smartcards. Because of the increasingly large amounts of data stored on a smartcard, the need for a file system arises. The limitations of current smartcard technology (such as limited stable storage capacity, very slow writes, very little main memory, lack of autonomy, etc), make it infeasible to use existing file system implementations. In this thesis, we present a file system specifically designed for smartcards.

In order to arrive at a solid design of such a smartcard file system, we first present the results of a survey of existing file system design methods, together with an analysis of which of those methods are most suitable for use in a smartcard file system.

Based on this analysis, we propose the design of a smartcard file system, SCFS. SCFS combines a number of techniques, such as shadow paging, and access control lists, to provide high reliability and high security. We also present a quantitative evaluation, which shows that, despite providing high levels of reliability and security, SCFS is able to attain adequate performance.

Contents

Survey

1 Introduction

This thesis discusses the design and implementation of a smartcard file system. This chapter begins with a description of smartcards and file systems. Based on these descriptions, we will establish a number of requirements for a smartcard file system.

1.1 Smartcards

A smartcard is a credit card sized plastic card, with a microprocessor and memory chip embedded in it. Many characteristics of smartcards, like their size, the interface to the outside world, communication protocols, and much more, have been standardized, and are defined in the various sections of the ISO 7816 standard [17].

Figure 1 – A smartcard[1]

Smartcards are designed to be a safe place for storing valuable information, and for performing trusted processing on that information[2]. This makes smartcards suitable for a variety of applications, like storing private information, e-wallets, identification, and so on. In chapter 1.2.3, we describe some of these smartcard applications in more detail.

In order to make a smartcard a safe harbour for information, it is crucial that information can only leave the device via channels controlled by the smartcard. Smartcard manufacturers go to great lengths to try to make their devices tamper-resistant, i.e. resistant to attacks that try to access information stored on the smartcard

[1] Image copyright 2003, ActiveCard Corporation,
< http://www.activcard.com/newsroom/image_gallery.html>

[2] Early smartcards have a long history of cases where their security measures were successfully evaded. Famous examples include the consistent compromising of smartcards used for protecting Canal+'s Pay-TV systems.

directly. Kömmerling and Kuhn [19] describe a variety of techniques for extracting protected software and data from smartcards, and their countermeasures.

1.2 Smartcard Technology

1.2.1 Processor

Early smartcards were based on relatively slow 8-bit microcontrollers. This limited to amount of processing that could be done on the smartcard. More modern smartcards are equipped with 32-bit RISC processors running at 25 to 66 MHz. They often come with coprocessors designed to accelerate encryption operations. This enables the smartcard to perform complex operations, like data-0encryption.

1.2.2 Memory

Smartcards usually contain three types of memory chips: ROM, RAM, and non-volatile RAM.

ROM

Read-Only Memory (ROM) is used to store information that does not change during the lifetime of a smartcard. Typically, it contains code and static data. The fact that ROM memory contents cannot be changed ensures that the programs stored in ROM, e.g. the operating system, cannot be tampered with. In addition, ROM is cheap and efficient in terms of power consumption and die area. These characteristics make that smartcards are typically equipped with more ROM than other types of memory. Current smartcard designs have between 100 KB and 1 MB or ROM.

RAM

Unlike ROM, Random-Access Memory (RAM) can be written to. However, RAM memory is volatile, i.e. it loses its contents when it is powered off. This makes is unsuitable for storing information that must be retained when a smartcard is not powered on. It is usually used as a 'scratch pad' for temporary information storage and calculations.

RAM requires more transistors per bit than ROM and hence uses more power and takes up more die area. Because of this, smartcards have low amounts of RAM, between 1 KB and 10 KB.

EEPROM

Electrically Erasable Programmable ROM (EEPROM) combines some features of RAM, and ROM. Like ROM, it is non-volatile, i.e. its contents are not lost when it is powered off. In addition, like RAM, it can be written to. These features make it a suitable storage medium. Its main drawback is that writing to EEPROM is a very slow operation.

Like RAM, EEPROM is expensive in terms of power usage and die area. However, since it is the only way for a smartcard to store information, smartcards normally have more EEPROM than RAM. Because of the high cost of EEPROM, the amount of EEPROM on modern smartcards varies more than that of ROM and RAM. Depending on their intended usage (and cost), smartcards have between 10 KB and 1 MB of EEPROM.

1.2.3 Smartcard Reader

A smartcard is a passive device; it has no built-in power source. In order for it to do anything useful, it has to be connected to a smartcard reader. This reader supplies the smartcard with power, and communicates with the smartcard. Smartcards come in two guises: contact card, and contactless cards.

Contact cards need a physical connection with the reader. Such cards have a number of metallic contact pads on their surface. There are eight contacts, defined in ISO 7816-2. Two for power supply voltage (VCC) and ground (Gnd), one for reset (RST), one for the clock signal (CLK), one for input and output (I/O), one for programming the IC (VPP), and two reserved for future use (RFU).

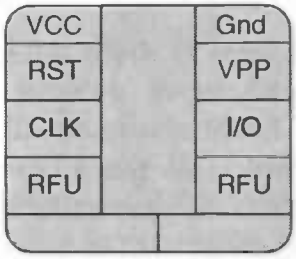

Figure 2 – Smartcard contacts as defined in ISO 7816-2

Contactless smartcards communicate with the reader by radio waves. The cards have an embedded antenna that receives the power and data transmitted by the reader.

1.3 Smartcard Applications

The functionality offered by smartcards can be divided in three categories:

- Information storage
- Identification
- Payment

Many smartcard applications use a combination of these functions. Below we will describe some scenarios where smartcards are used.

1.3.1 Telecommunication

The most widespread use of smartcards is without doubt the Subscriber Identification Module (or SIM card) found in GSM mobile phones. Although perhaps not immediately recognized as such, the little SIM card that stores personal subscriber information and preferences is in fact a smartcard. However, due to the demand for ever-smaller mobile phones, the SIM card has been reduced in size from its traditional credit card size, to its current dimensions.

A SIM card provides both information storage and identification functionality. Users can store information like telephone numbers, SMS messages, and so forth on the card. Additionally the card contains information used to identify the subscriber to the network. Before the phone can be used on the network, the user has to enter a PIN code, which is compared with the PIN code stored on the SIM card.

1.3.2 Payment

Smartcards are used to make small payments, as a replacement for coins. More and more devices like pay phones and vending machines that traditionally used coins are being equipped with smartcard readers. This has advantages for both the users of these devices – who no longer have to worry about having the right amount of change - as well as for the proprietor – who no longer has to collect the coins from his devices.

Smartcards used for payments come in two variants: rechargeable cards that can be reloaded with money, for example at ATMs, and prepaid cards that can be bought with a specific amount of money preloaded and are disposed of when they are empty. Prepaid cards are typically single-purpose cards; they can only be used in e.g. pay phones. Rechargeable cards are sometimes also used as a general

replacement for coins. Examples of such 'electronic purses' are Singapore's CashCard, and the Dutch Chipper and Chipknip projects.

Some typical examples of situations where smartcards are used for making payments are:

- Pay phones
- Public transport
- Vending machines
- Parking lots

1.3.3 Loyalty

Super markets and other retailers use smartcards for loyalty plans; a well-known example is AirMiles. In exchange for information about their purchases, shops offer their customers discounts or gifs. This is usually implemented by issuing smartcards to the customers, which they have to present to receive their discount.

1.3.4 Access Control

Smartcards are also used to control access, for example to a company's premises or to a computer system. The smartcard will typically be used for authentication, i.e. to ensure that someone is who she claims to be. This is usually done by storing some information on the card that is known only to the legitimate owner of the card, like a PIN code or a password. Alternatively, information unique to the holder can be stored. This can be a fingerprint, information about the retina, or other uniquely identifying information.

1.3.5 Multi Application Cards

Like any other computational device, the processing power and storage capacity of smartcards increases at a high rate. The first smartcards had 4-bit processors, and a memory capacity of less than a kilobyte. The current generation is equipped with 32-bit processors and EEPROM memories of 1MB or more. This increased capacity has given rise to the multi application card, where one card is used in many different roles.

For example, Florida State University has issued 40,000 smartcards to its students. These cards are used as students' personal identification, for access control at dormitories, and to pay for a wide range of

services like food, payphones, photocopying, transportation, and vending machines.

Using one smartcard in many different roles means that much more information will be stored on the card. This information has to be stored in a reliable and secure way. Traditionally, this was handled individually by each application. Although appropriate for resource limited, single application smartcards, this approach loses its appeal when multiple applications are on the some card. If each application has to manage its own data, a lot of functionality will be duplicated in each application. Another problem arises when applications want to share data. This is much easier when there is a central repository for all data.

These considerations indicate the need for a file system. The availability of a file system removes the need for each application to implement its own methods for making sure that its data is stored safe and secure.

1.4 File Systems

In the previous section, we concluded that a file system would be valuable for multi-application smartcards. In this section, we will give a more extensive description of what the goals of a file system are, and how these goals are typically achieved.

A file system has two main purposes:

- Applications can store information in a file system
- Applications can retrieve stored information from a file system

To make this possible, file systems typically provide two abstractions: files and directories. Files are named containers in which information can be stored. Directories provide a way to organize those files by providing a hierarchical namespace.

1.4.1 Files

A file is an abstraction mechanism. They provide a way to store information on a storage medium, and to read it back later. A file consists of several components:

Data

The most important component of a file is the information that is stored in it, its data. The data associated with a file is conceptually a sequence of bytes. Bytes can be written to a file, and read from it.

Metadata

In addition to the data associated with a file, a file also contains some metadata – data describing the actual data. For example, the size of the data associated with the file is part of a file's metadata. Most file systems store file data physically separated from metadata. To be able to locate the data, the metadata also includes information about where on the storage medium the actual data is located.

Typically, information about when the file was created, last modified, and last accessed, is also part of the metadata. Additional metadata can also be associated with files, for example information about access rights, the type of the file (like text file, application, etc.)

Naming

Another important metadata item that is part of a file's metadata is the name of the file. A name provides a means of referring to a file. When a file is created, it is given a name. This name is subsequently used to access the file.

1.4.2 Directories

Most file systems use directories to create a hierarchical namespace. A directory contains a number of entries. Each entry contains a file name, and that file's metadata (or a pointer to the metadata). Usually, directories are files themselves, and hence a directory can also contain other directories. This allows the creation of a directory hierarchy. At the top of this hierarchy is the *root* directory.

Such a hierarchical namespace allows large numbers of file to be stored in an organized manner, making it easier to retrieve files at a later time.

1.5 A Smartcard File System

In the previous sections, we described what a smartcard is, what a file system is, and how the trend towards multi-application smartcards drives the need for a smartcard file system.

In this section, we will present a set of requirements for such a smartcard file system.

1.5.1 Reliable

The most important requirement for a smartcard file system is that it is reliable. In the context of a file system, reliability means that the file system is resilient to system failures.

A system failure is a failure in which the entire contents of volatile memory are lost. Examples of system failures are, interruption of the power supply, a reset occurring because of a bug in system software, and so forth.

System failures can lead to data loss, and inconsistencies. Data loss can occur when data is 'in transit' to the storage medium at the time when the system failure occurs. This happens for example if data is cached in volatile memory and has not yet been written to the storage medium when the system failure takes place. Data inconsistencies can occur when related pieces of data are being written when a system failure occurs. When a system failure interrupts an update to related pieces of data, some parts of the data will already contain the updated version, while other parts still hold the original version.

System failures, particularly power failures, are a regular event for a smartcard. A smartcard does not have its own power supply, but instead obtains its power from a so-called reader. The design of most smartcard readers is such that the smartcard can be removed from the reader at any time[3]. This causes the power supply of the smartcard to be interrupted, and subsequent loss of the contents of volatile memory.

Because of this, a smartcard file system should be able to recover from system failures, and provide guarantees about when data is safe and when it might be lost due to system failures.

1.5.2 Secure

Smartcards are typically used to store important and private information, e.g. money (e-wallets), medical or insurance information, biometric profiles, etc.

It is imperative this information is not accessed or modified by unauthorized applications. On the other hand, it is desirable that information can be shared between applications when appropriate.

[3] An exception to this rule are readers such as those typically found in ATMs, which completely 'swallow' the smartcard, making it impossible to remove the card until the ATM releases it.

This calls for a file system that strictly enforces security policies, and allows for a high-granularity specification of access rights.

1.5.3 Efficient

Smartcards typically use EEPROM as their storage medium. Although reading from EEPROM is comparable to reading from RAM in terms of speed, writing to EEPROM is a very slow process. Typical write speeds are between 5 and 10 KB/s [16].

This means that to be able to achieve acceptable performance, the file system should try to minimize the amount of write operations it performs.

1.5.4 Compatible

Several competing platforms exist for smartcards, including MultOS [23], JavaCard [7], and Windows for Smart Cards [27]. Each of these platforms defines their own file system interface. It is not clear yet which platform(s) will prevail, therefore, the file system should be able to support multiple application programmer interfaces (APIs).

1.5.5 Requirements

Summing up, we have established the following requirements for a smartcard file system:

- **Reliable**: the file system should be resilient to system failures.
- **Secure**: the file system should support detailed security policies, and it should be able to enforce those policies.
- **Efficient**: the file system should try to minimize the number of EEPROM write operations.
- **Compatible**: the file system should be able to support multiple APIs.

1.6 A New Design

In this section, we will explain why it is neither feasible nor desirable to use an existing file system implementation on a smartcard.

The foremost reason is that most existing file systems were designed to meet different requirements, or at least have a different emphasis on the various requirements. The biggest problem is that existing file systems do not meet the reliability requirements of a smartcard file

systems. Usually, this is because existing file systems often sacrifice reliability for higher performance. In situations that do require high reliability, hardware-based approaches are typically used to offer resilience against system failures, e.g. redundant components or UPS.

Another reason is that modern file systems were designed for hardware with vastly different capabilities than what is available on a smartcard. While a modern smartcard's processing power is comparable to that of a recent desktop PC, the amount of main memory and storage capacity on a smartcard is very small.

For example, any desktop PC produced in the last ten years has had a least a thousand times as much main memory as a state of the art smartcard. Nowadays, even embedded systems that use a file system, for example the Symbian operating system for mobile phones, are equipped with many times more main memory (several megabytes) than available on any smartcard.

These observations have led us to believe that the best approach for creating a smartcard file system is to make a new design. In the remainder of this thesis, we will describe the design and implementation of a smartcard file system that aims to meet the requirements we described previously.

The design and implementation of this smartcard file system was carried out by the author as part of an internship at Infineon Technologies AG in Augsburg, Germany. Infineon is a leading designer and manufacturer of smartcards. In addition to the smartcard hardware, Infineon also supplies a software library that provides hardware abstraction, and can serve as the basis for a complete smartcard operating system. The smartcard file system we present in this thesis is meant to become a part of this core library. Therefore, the file system will initially be targeted to run on a smartcard currently under development at Infineon, the SLE 88CX720P [16]. However, we will keep the design as general as possible, so the file system can also be used on similar smartcards.

1.7 Outline

The remainder of this thesis is organized as follows:

In chapters 2-4, we present a survey of existing file system designs and implementations. The aim of this survey is to answer the following questions:

- What methods exist for providing resilience again system failures in file system?
- What methods exist for securing data in file systems?
- What methods exist for providing high performance for file system (write) operations?

In chapters 5 and 6, we present an analysis of the methods described in preceding chapters, and will select methods that are most appropriate for a smartcard file system.

In chapters 7-10, we present the design and implementation of our smartcard file system.

In chapters 11 and 12, we present a qualitative and quantitative evaluation of how our implementation meets the requirements.

Chapter 13 and 14 summarize the conclusions of this work, and offer suggestions for future work.

Survey

In the following chapters, we give an overview of the various methods that have been developed by researchers and commercial file system designers for making file systems reliable, secure, and efficient. In the next part, chapters 5 and 6, we will analyze these methods to determine which of these methods are most appropriate for the design of our smartcard file system.

2 Reliability

As we pointed out in section 1.5.1, an important requirement for a file system is that it should be resilient to system failures. This applies not only to smartcard file systems, but also to practically every file system. Therefore, a lot of research has been done to develop methods to provide that resilience. In this chapter, we will give an overview of those methods.

2.1 Resilience to System Failures

As we saw in section 1.5.1, system failures can cause data inconsistencies. A file system that is resilient to system failures has to be able make guarantees about what data inconsistencies can, and cannot, occur because of a system failure. In practice, this depends largely on which data inconsistencies a file system can detect and recover, after a system failure.

Many existing file system only provide guarantees about the consistency of metadata, and leave the task of keeping the regular data consistent to the applications.

One of the reasons for this decision is that it is easier to keep metadata consistent. A file system knows about the structure of its metadata. Furthermore, there is usually some redundancy in metadata. These properties can be used to detect when metadata has inconsistencies, and often to recover such inconsistencies.

Another reason for the fact that many file system designers choose not to worry about the consistency of regular data is that ensuring data consistency results in a loss of performance. Metadata usually constitutes only a small part of the total amount of data, making the performance penalty for keeping metadata consistent relative small. When it comes to guaranteeing the consistency of regular data, many file system designer prefer performance to reliability.

In the remainder of this chapter, we will first examine some methods for guaranteeing metadata consistency.

2.2 Metadata Consistency

Many file system operations consist of several related updates to separate metadata items. If a system failure occurs in the middle of such updates, inconsistencies can occur in the metadata.

For example, when creating a new file on a typical UNIX file system, the file system allocates an inode[4], initializes it, and constructs a directory entry that points to it. If a system failure occurs when the directory entry has reached the disk, but the initialized inode has not, the metadata is inconsistent, since the directory entry now points to an inode with undefined content. Similar situations can occur when deleting files, appending data to files, etc.

2.3 Update Sequencing

One important group of techniques that guard metadata rely on *update sequencing*: by carefully choosing the order in which metadata updates are written to disk, the damage caused by system failures can be limited to recoverable inconsistencies, i.e. inconsistencies that can be detected and repaired.

For instance, in the example given above: if the directory entry reaches the disk, but the inode does not, the directory entry references an inode with undefined content. In general, it is not possible to distinguish between a valid inode, and one with undefined content.

On the other hand, if the inode is always initialized before the directory is written, all that can happen is that an inode is lost because it is no longer referenced anywhere. This is easily detectable by checking if each inode that is marked as being used, is actually referenced somewhere. This is usually done by a special *scavenger* program that is run after a system failure. Examples of such programs include UNIX's *fsck* [25] and MS DOS's chkdsk.

Recoverability of metadata after a system failure can be ensured by always ordering metadata updates according to three rules: [12]

- Never point to a structure before it has been initialized (e.g., an inode must be initialized before a directory entry references it).

[4] In UNIX file systems, all the metadata belonging to a file is stored in a so-called *inode*.

- Never reuse a resource before nullifying all previous pointers to it (e.g., an inode's pointer to a data block must be nullified before that disk block may be reallocated for a new inode).
- Never reset the last pointer to a live resource before a new pointer has been set (e.g., when renaming a file, do not remove the old name for an inode until after the new name has been written).

2.3.1 Synchronous Writes

Most file systems use some form of caching to improve performance. Copies of disk blocks being modified are kept in main memory, and updates are made to these cached copies. Cached blocks are written back to disk when the cache is full, after a certain period of time, or when an application requests it. Because of these *delayed* or *asynchronous* writes, the file system cannot control in which order the disk blocks are written, which means that it cannot enforce the above-mentioned ordering rules.

Therefore, many file systems use *synchronous writes* to keep metadata consistent. Instead of being cached, disk blocks containing metadata are simply written to disk in the order determined by the ordering rules. Since hard disks typically guarantee that writing a single disk block is an atomic operation, sequencing metadata updates like this is enough to limit the damage in case of a system failure to safe metadata inconsistencies.

Since it is such a straightforward technique, synchronous writes are used in many (older) file systems, including the VMS file system [24], MS DOS's FAT file system [9], and many traditional UNIX file system such as the original UNIX file system [33], BSD's FFS [26], and Linux's ext2 [2].

The main drawback of using synchronous writes is that metadata updates are not cached. This means that disk speeds becomes the limiting factor for metadata updates, rather than processor and memory speeds.

The resulting performance degradation is in fact so severe that many file systems implementations choose to ignore the ordering rules in certain cases, thereby trading integrity and security for performance.

2.3.2 Dependency Tracking

An alternative to immediately writing all metadata updates to disk is to make the cache aware of the order in which blocks have to be written to disk. Using this technique means that metadata updates

can be cached, while still maintaining metadata consistency. This is called *dependency tracking* because the cache now has to track on which other parts of metadata each metadata item depends, i.e. which metadata items have to be written to disk first, before a particular metadata item can safely be written to disk.

Inter-buffer Dependencies

Dependencies can be tracked at the level of disk (or cache) blocks. This method is straightforward, but provides only limited opportunities for delaying (caching) writes. This is because the system must avoid creating circular dependencies. A circular dependency can occur because one cache block can contain multiple metadata items. The system must prevent such circular dependencies by writing appropriate cached blocks to disk. Unfortunately, situations causing circular dependencies occur frequently in typical file system usage patterns. This renders this method of tracking dependencies useless in practice.

Soft Updates

Circular dependencies can be avoided by tracking dependencies at the level of individual metadata items, as opposed to tracking them at the more coarse-grained buffer level. This allows a cached block containing updated metadata to be written back to disk at any time by undoing the updates that are still in progress. When the block is written, the updates are repeated, and the metadata update is allowed to complete. This guarantees that the on-disk state of the file system is always consistent.

This approach is called *soft updates* and is described in more detail in [13]. Soft updates are used in newer versions [12] of the Fast File System [26], which is used in several variants BSD operating systems.

2.4 Full Data Consistency

The methods described above can only be applied to metadata. They require knowledge of the structure of the data being stored and a certain amount of redundancy in that data, to be able to detect inconsistencies, and to recover the data after a system failure. Such information is generally not available about 'regular' file system data.

In the following sections, we will describe some methods that can guarantee the consistency of all data, both metadata and regular data.

2.5 Transactions

As we mentioned above, these methods cannot assume anything about the data that has to be kept consistent, i.e. they make no assumptions about the structure of the data. Instead, these methods make sure that always at least one version of the data is available. In other words, when data is updated, either a backup is made of the old data, or the new data is first written to another location, before the original data is overwritten. By ensuring that either the old or the new version of the data is available at any time, the system can always recover a consistent version of the data after a system failure.

In other words, these methods implement transactions. A transaction is an atomic unit of read- and write-operations; it is completed either in its entirety, or not at all.

Although transactions have long been used in file systems to protect metadata, the idea of using them to protect all data has never really caught on. One reason for this is that file system API's generally do not provide any mechanisms for applications to control what data should be part of a transaction. Without explicit support for transactions in the API, all a file system can guarantee is that single file system operations are atomic. This is often not the most desirable behaviour. Regular data, like metadata, tends to be related, and therefore updates to several structures in a file – or even updates spread over multiple files – are needed to keep the data consistent.

A file system that can only guarantee the atomicity of single file system operations is not of much use in such scenarios. The application still needs to implement its own ways of keeping data consistent, which leads to two layers in the system trying to do the same work.

This observation has held back the implementation of full data consistency in file systems. A possible solution for this, namely providing transaction support in the file system API, is used in the – still under development – Reiser4 file system [29].

2.6 Logging

The most common technique used to implement transactions is by using a *log* or *journal*. While executing a transaction, the system writes information about that transaction to a log. In the event of a system failure, the information in the log is used to undo or redo the transaction, thus providing atomicity.

Many popular file systems use logging to ensure metadata consistency. Logging only metadata is typically a low overhead approach, because metadata is typically only a small fraction of the total data. Additionally, writing to a log is cheaper than writing to a random location, further reducing the overhead caused by logging. We will discuss the performance aspects of logging in more detail in chapter 4.

2.6.1 Redo Logging

Redo logging, also called *new value logging* or *write-ahead logging*, writes all the updated data to the log before it overwrites the original data. A system using new value logging can always redo a transaction after a system failure by rewriting the updated data found in the log.

Redo logging cannot be used to abort an already started transaction. However, for a file system this is not a problem, because they do not usually provide mechanisms for aborting transactions.

A drawback of redo logging is that all updated data has to be written to the log, before any old values can be overwritten.

Many commercial file systems use redo logging to guarantee metadata consistencies, including XFS [37], JFS [5], and NTFS [36]. The Ext3 file system [38] uses redo logging for both metadata and regular data.

2.6.2 Undo Logging

Undo logging, or *old value logging*, means that before data is overwritten with updated values, the old value of that data is written to a log. After a system failure, the old values in the log can then be used to undo the effects of any transactions that were in progress when the failure occurred.

Unlike redo logging, undo logging does not require that all data is written to the log before old values are overwritten. Instead it requires that all data has reached the disk when the transaction commits (finished), since a commit indicates that the changes made by a transaction are now permanent, and hence are not allowed to be undone in the case of a system failure.

Undo logging is not used by any file system known to the author. The reason for this is probably that file systems typically have no transaction abort. In a database, it is quite common to abort a running transaction, for example, when a transaction conflicts with

another running transaction. File systems have no need for this because they have full control over the transaction, and hence know in advance if a transaction might cause a conflict.

2.6.3 Combining Redo and Undo Logging

Both redo and undo logging impose restrictions on which data can be cached. Redo logging requires all updates are written to the log before any old data is overwritten, and undo logging requires that all updates are written to disk before the transaction commits.

These restrictions limit the freedom of the system to schedule disk writes. For example, writes can be delayed in anticipation of further updates to the same data, or the can be done at more convenient time.

To overcome these restrictions, a file system can write both undo and redo information to the log. This approach is taken by the Episode file system [8], amongst others.

2.6.4 Log-structured File Systems

A variation on the logging techniques described previously is to use the entire file system as a log, appending all data and metadata to the log. This approach, called a log-structured file system, is proposed by Ousterhout in [31], and described in more detail in [34]. It has the potential to improve performance since it eliminates the needs to write data twice (once to the log, once to the original location).

2.7 Shadow Paging

A system can also provide transactions without writing undo or redo information to a log. This can is achieved by writing all updates to a *shadow copy* of the original data. This approach, called shadow paging, was first presented in [21].

When all updates have been made and the transaction commits, the system makes the shadow copy the current copy. This is usually achieved by having a directory, which records the location of each data item. By updating the directory to point to the shadow copies, the shadow copies become the current data.

Mime [6] and WAFL [15] are some examples of file systems that use shadow paging. Reiser4 [29] use a combination of logging and shadow paging. It uses heuristics to decide at runtime which of the two methods will be most efficient.

2.8 Conclusions

A wide variety of techniques exists to provide various levels of resilience to system failures. Techniques based on update sequencing, such as synchronous writes or soft updates, can only protect metadata. These methods do not provide enough resilience to meet the requirements for a smartcard file system. However, techniques that provide a transaction mechanism, like the various forms of logging and shadow paging, are more general, and can be used to protect all data in a file system. In the next part, chapters 5 and 6, we will take a closer look at these techniques to determine which are most appropriate for use in a smartcard file system.

3 Security

In this chapter, we present several approaches to providing security in a file system. We first cover various techniques for providing access control. Further on, we will look at protection against offline attacks.

3.1 Access Control

Most file systems support some form of *access control*. This means that they offer some way of specifying which objects a subject can access. In this context, objects are things like files, directories, and the like. A subject is the entity that is allowed or denied access to an object. Usually this is a user, or a process.

3.1.1 Access Matrix

Lampson access matrices [20] provide a way to describe the protection state of a system. Each subject in the system has a row in the table, and each object has a column. The entries describe the access rights subjects have on the objects.

	Object1	Object2	Object3
Subject1	r, w	r	r, x
Subject2			r, x

Table 1 – Example of a Lampson access matrix

The table above shows an example of such an access matrix. Each entry contains some letters denoting the access rights, e.g. Subject2

has read and execute rights for Object3, and is allowed no access to Object1 and Object2.

There are two ways of representing an access matrix in a file system.

- Access control lists: store a list of subjects and their access rights with each object
- Capabilities: store a list of objects and associated access rights with each subject

3.1.2 Access Control Lists

Access control lists (ACLs) are the more common way of storing access control information in a file system. One of the simplest forms of ACLs is found in traditional UNIX file systems:

Associated with each file (and directory) are 9 bits that specify the access rights to that file. Those 9 bits are divided into three groups of three bits each. The three groups indicate the access rights for the user who created the file (the owner), a group of users associated with the file, and everybody else. The tree bits in each group represent read, write, and execute permissions respectively.

Other file systems use ACL implementations that are more flexible. It is usually possible to specify access rights for any number of users or groups of users. In addition, more fine-grained access rights can be specified. An example of this can be seen in Table 2, which shows the access rights available in NTFS.

Directory specific	**Files specific**	**Shared**
Traverse directory	Execute file	Delete
List directory contents	Read data	Read attributes
Create file	Write data	Write attributes
Create directory	Append data	Read access rights
Delete file		Modify access rights
Delete directory		

Table 2 – Access rights in NTFS

Other examples of file systems offering similar ACL implementations are XFS, and JFS. A draft POSIX specification (1003.6) for ACLs was created, but this specification was abandoned.

3.1.3 Capabilities

Our description of capabilities in section 3.1.1 was a slight simplification. ACLs and capabilities are not equivalent in the sense that they are simply two different ways to represent a Lampson access matrix. The difference is that where ACLs specify the operations that *users* are allowed to perform on objects, capabilities specify operations that *processes* are allowed to perform.

This subtle distinction has some important implications. Tying access rights to users instead of processes means that each process a user runs has the same access rights. This makes selectively restricting access harder. In a capability system, a user can chose to deny an untrusted process certain permissions. When ACLs are used, a process always has the same permissions as the user who owns the process.

3.2 Offline Attacks

In addition to providing ways to limit access during normal system operation, some file systems also try to prevent attempts to circumvent access control mechanisms by simply accessing the storage medium directly. Such attacks are called offline attacks.

The most common type of offline attack is to steal the storage medium, and to place it in a system that is under control of the attacker. This allows the attacker to bypass any access right checks in the file system software.

3.2.1 Encryption

One way of preventing offline access is by encrypting the data in the file system. Microsoft's NTFS supports this on a file-by-file basis.

Encryption can also be done in layers below the file system, for example by the storage hardware itself, or by a device driver for the storage medium.

3.3 Conclusions

In this chapter, we have described a number of approaches for providing security in file systems. We have covered techniques for protecting data against malicious applications that try access data not intended for that application, as well as techniques that provide protection against attacks that try to circumvent such measures.

4 Performance

In this chapter, we will describe means of improving performance in file systems.

The limiting factor for file system performance is usually not the maximum data transfer rate of the hard disk, but the access times. In order for a hard disk to be able to read or write data, the heads of the hard disk have to be in the right position. Hard disks are typically set up such that the heads are already in the correct position if data is accessed sequentially. If data is accessed in a random pattern, each data access has to be preceded by a seek operation to move the heads to the proper position. This seeking is a mechanical process that takes a relatively large amount of time.

For example, a typical current hard disk has a maximum data transfer rate of 40MB/s, and an average access (or seek) time of 4ms. Let us say the file system stores data in blocks of 4KB. If those blocks are randomly distributed over the disk, each access to a block is preceded by a seek. The actual transfer of the data in a block takes $\frac{4KB}{40MB/s} = 1ms$, but the seek takes 4ms. Therefore, the total time needed to access 4KB of data is 5ms, resulting in a throughput of $\frac{4KB}{5ms} = 8MB/s$, or only 20% of the theoretical maximum.

If, on the other hand, the blocks were placed sequentially on the hard disk, the heads would already be in the right position to access each block, obviating the need for a seek, and thus allowing the maximum data transfer rate to be achieved.

4.1 Read-Optimized vs. Write-Optimized

File system performance optimizations can be divided into two groups: read-optimized, and write-optimized. Read-optimized file systems try to maximize read performance, based on the observation that reads are typically much more common than writes. Such file systems therefore try to optimize their on-disk data structures, and file allocation policies to minimize the time a hard disk spends seeking.

Write-optimized file systems on the other hand focus on maximizing write performance, assuming that files are cached in main memory and that increasing memory sizes will make the caches more and more effective at satisfying read requests [30], [34].

4.2 Sequential Access

Various methods have been devised to minimize the number of seeks needed during file system operations. Those methods are usually based on the observation that files are often accessed sequentially. By laying out the blocks belonging to a file in a sequential way, applications that access files sequentially will cause the disk to be accessed sequentially too.

4.2.1 Contiguous Files

The first file systems forced the user to specify the size of a file when the file was created. The file system then allocated the required amount of consecutive blocks for that file.

Besides being simple to implement, this approach has the benefit that the blocks belonging to a file are always laid out sequentially, thereby minimizing the number of seek operations when the file is accessed sequentially.

The obvious drawback is that the file size has to be known when the file is created. Without this information, the file system does not know how many blocks it should allocate to the file. In most practical situations however, it is not feasible to specify the file size in advance.

4.2.2 Contiguous Allocation

Forcing files to be stored contiguously is obviously not very flexible. However, in order still to gain the benefits of contiguously allocated files, many file systems try to allocate blocks for a file in a contiguous way.

A naïve implementation of this contiguous allocation policy, which simply allocates blocks to files as the files grow, will often find that it cannot allocate blocks contiguously, because adjacent blocks have already been allocated to another file.

Therefore, some file systems preallocate blocks. When data is written to a file, the file system preallocates multiple adjacent blocks at once. Preallocation leads to more contiguously allocated blocks. Obviously, the file system has to take care to deallocate those blocks if it turns out that those blocks are not actually written to. Preallocation was introduced in the DEMOS file system [32].

4.3 Clustering

Another characteristic of hard disks is that the seek time depends on the distance the heads have to move to reach their desired position. In other words, access two disk blocks that are close together takes less time than accessing two blocks that are far from each other.

File systems exploit this characteristic by placing logically related blocks near each other on the disk.

4.3.1 Allocation Groups

A commonly observed pattern in file system usage is that files in the same directory tend to be related. The BSD Fast File System (FFS) [26] exploits this property by dividing the disk in allocation groups. An allocation group is a group of disk blocks that are near each other. FFS tries to allocate blocks for files in the same directory from the same allocation group to minimize seek time when they are accessed together.

Many file systems have begun to use similar techniques.

4.3.2 Embedded Inodes

Another technique that tries to group related disk blocks is called *embedded inodes*. A file in a conventional UNIX system consists of two parts, an inode that describes which data blocks belong to the file, and a directory entry that stores the name of the file. The directory entry is part of the data associated with the file's parent directory, and the inode is usually in a special inode section of the disk. This means that the two parts that constitute a file are located in separate blocks on the disk.

Opening a file requires that both blocks are accessed. In [11], Granger and Kaashoek propose to move the information in the inode into the directory entry. This technique halves the number of blocks accessed when open or creating a file.

4.4 Logging

As we saw chapter 2, logging metadata is a way to keep metadata consistent without needing synchronous writes. Synchronous writes are bad for file system performance for two reasons. First, they increase latency, because file system operations have to wait for the synchronous write to complete before the operation can continue. Second, they causes more write operations since frequently accessed

data that otherwise would have cached, has to be written to disk repeatedly.

Logging also improves recovery time drastically. Whereas file systems that use synchronous writes, or other types of dependency tracking, e.g. soft updates, have to scan the entire file structure, logging file systems only have to replay the log to restore the file system to a consistent state.

4.4.1 Log-structured File Systems

Log-structured file systems provide two additional benefits over file systems that use a separate log. First, all writes are to the log, which means that all writes can be sequential. Second, data only has to be written once, not twice as is the case with other file systems that use logging. A drawback however, is that log-structured file systems need a background process that cleans the log, i.e. frees up log space by removing data that is obsolete because a newer version of the data has been written.

4.5 Shadow Paging

Like a log-structured file system, a system that uses shadow paging only needs to write data once. A drawback is that shadow paging destroys locality and sequentiality of data, because updated data is always written to a different location. This write-anywhere behaviour makes it difficult to implement techniques like contiguous allocation, or clustering.

4.6 Conclusions

In this chapter, we have described several techniques that can be used to achieve higher performance in typical file system operations.

We will use these descriptions, together with those in the previous chapters, in the next part, chapters 5 and 6, to determine which of those techniques are most applicable to a smartcard file system.

Analysis

In this part, we analyze the methods we described in the previous chapters. We will determine which methods are most suitable for implementing a file system that meets the requirements we defined in section 1.5. We will use the results of this analysis for the design of our smartcard file system, which was presented in preceding chapters.

5 Smartcard File System

Before we analyze the methods described previously, we will first list the major differences between a smartcard file system, and a more conventional file system.

As we mentioned in section 1.6, our file system will initially be used on an Infineon SLE 88CX720P smartcard [16]. This does not mean that our design will be usable solely on that particular smartcard. Instead, we use the capabilities of the Infineon smartcard as a reference point: an indication of the strengths and weaknesses of the environment in which our file system will be used.

A smartcard file system operates in an environment vastly different from the situation in which other file systems typically operate. This will obviously influence our analysis and design decisions. What might be an appropriate trade-off for a file system designed to run on a system with multiple terabytes of storage capacity and multiple gigabytes of memory, is not necessarily to best option for a smartcard file system intended to store maybe one megabyte of data and a mere eight kilobytes of memory at its disposal.

5.1 Hardware

Below we give a short overview of the specifications of the SLE 88CX720P smartcard. Some of the details we give now will not be directly relevant in these chapters, but will be referred to when we describe the design and implementation of the smartcard file system, in chapters 7-10.

5.1.1 Processor and Memory

The SLE 88CX720P smartcard has a 32-bit micro-controller, operating at 55 MHz. It is equipped with 240 KB of ROM, 80 KB of EEPROM, and 8 KB of RAM.

This means that the processing power of this smartcard is roughly comparable to that of a PC from about ten years ago. However, the amount of RAM is almost a thousand times less than what was common ten years ago.

5.1.2 Storage Medium

The storage medium, 80 KB of EEPROM, is also significantly different from the hard disk for which most file systems are designed. The size is the most obvious difference, several orders of magnitude less than any hard disk ever built.

Another difference is that the performance characteristics of EEPROM differ greatly from those of a typical hard disk. The table below summarizes the main performance indicators for the EEPROM in the Infineon smartcard [16], and a modern hard disk [22].

	EEPROM	Hard disk
Read speed	10 MB/s	54.2 MB/s
Write speed	3.5 KB/s	54.2 MB/s
Access time	6 ns	0.8 ms (min) 8.5 ms (average) 17.8 ms (max)

Table 3 – EEPROM vs. hard disk performance.

As can be seen in the table, the read speeds of EEPROM and a typical hard disk are comparable. However, the write speed and access times are very different. A hard disk has no significant difference between read and write speeds. On the other hand, writing to EEPROM is very slow. The roles are reversed however, when we look at the access times. Like other memory chips, EEPROM can be accessed virtually immediately; there are no heads or other mechanical parts to be moved.

5.2 Requirements

As we already mentioned in section 1.6, a smartcard file system has different priorities than most other file systems. Traditionally, file systems have tried to achieve maximum performance, often at the expense of some reliability. In our smartcard file system, reliability is an absolute requirement. So while high performance is definitely very desirable, it should not be achieved by sacrificing reliability.

5.3 Conclusions

In this chapter, we have highlighted the most important differences between a smartcard file system and other file systems.

From a hardware perspective, the most important difference is that the EEPROM used on smartcards does not have high access times as hard disks have. This is an important difference because many performance optimizations found in file systems are based on reducing access times.

As far as requirements are concerned, the biggest difference is the high reliability required from a smartcard file system as opposed to more traditional file systems.

6 Requirements

In chapters 2-4, we described various methods that are used in file systems to meet the requirements we set for our smartcard file system. In this chapter, we will analyze these methods in order to determine which of those methods are most applicable to a smartcard file system.

6.1 Reliability

The need to be able to guarantee the consistency of all data in the file system means that the metadata approaches described in chapter 2.2 provide insufficient guarantees.

This narrows our choices down to logging versus shadow paging. From a reliability point of view, there is nothing to differentiate between the two: both can preserve data consistency in case of a system failure. There are important differences when it comes to performance. We will discuss these differences further in section 6.3.

6.2 Security

6.2.1 Access Control

As we saw in chapter 3.1, capabilities have a number of advantages over ACLs. Unfortunately, there are also some problems when using a capability-based approach. Most importantly, capabilities require system-wide support.

As we are designing a file system, and not an entire operating system, this more or less rules out capabilities.

6.2.2 Offline Attacks

There is no need for our file system to provide data encryption. The Infineon smartcard automatically encrypts the entire memory contents. Additionally, the smartcard is provided with a range of countermeasures against attacks. Many of such attacks work by exposing the smartcard to extreme conditions and exploiting quirks caused by those extremes. The countermeasures include low and high voltage sensors, low and high frequency sensors, reset filter, temperature sensor, glitch sensor, and light sensor.

6.3 Performance

Traditional file system performance optimizations – like allocating data contiguously, clustering related data, and so forth – all aim to minimize the movement of the hard disk's heads, thereby reducing the access (or seek) time. Obviously, such techniques are of no use when the storage medium is not a hard disk, but EEPROM, which – as we saw in section 5.1.2 above – has very low access times.

6.3.1 Logging vs. Shadow Paging

Virtually all systems that implement a transaction mechanism, like file systems, databases, persistent stores, and so forth, have chosen logging over shadow paging. In fact, the designers of one of the first commercial database systems – IBM's System R – mention shadow paging as the biggest mistake in their design [4].

The reason for this general aversion of shadow paging is that is was found to have inferior performance [41]. Whenever data is updated in a shadow paging system, it is written to a different location. This tends to destroy any ordering the data originally might have had, and thwarts any efforts made by disk allocation policies to keep data stored contiguously. This behaviour is the exact opposite of what is necessary to attain maximum performance on hard disks.

Logging on the other hand, is much more suited to performance characteristics of hard disk. Writes to a log are inherently sequential, which is the optimal way of accessing a hard disk.

In our case however, access times are not an issue. Therefore, logging is not automatically the best approach for our file system. Therefore,

we will take a closer look at the performance aspects of logging, log-structured approaches, and shadow paging in the following sections.

As we saw in section 2.6, there are two approaches to logging: 'normal' logging, using a separate log, and the log-structured approach. When a separate log is used, the actual data is stored in a conventional manner and the log is only used for recovery purposes. With a log-structured system on the other hand, data is only stored in the log.

The main advantage of a separate log is that it does not affect how the actual data is laid out on disk, i.e. the system is free to use the most appropriate allocation policies and clustering techniques. The drawback of a separate log is of course that data has to be written twice, once to the log and once to its actual location. However, this is not as bad as it might seem. First, the writes to the log are sequential and therefore relatively cheap. Second, the use of a log allows the system to delay writes to the actual data, i.e. such writes can be cached.

A log-structured system does not have to write data twice because the log is also the final destination of the data. Furthermore, all writes are to the log, and therefore sequential. The drawbacks of a log-structured approach are that the log dictates the layout of data, so while writes are always sequential, reads tends to require a lot of disk head movement. However, given enough memory, reads can be cached quite easily. Another drawback is that by always appending new data to the end of the log, the log will eventually reach the end of the disk. By this time, much of the data in the log is outdated because newer versions have been written to the log. In other words, the free space in the log becomes fragmented into small pieces corresponding to files that were deleted or overwritten.

There are two ways of dealing with this fragmentation ([34]): threading and copying. Threading leaves the live data in place and threads the log through the free space. However, threading causes the free space to become fragmented, so the log can no longer be written sequentially, making the log-structured system no faster than a traditional system. The alternative is to copy the live data out of the log in order to free up large extents of free space. The drawbacks of this are that it requires copying otherwise unmodified files, and that it uses a background process (or daemon) to clean the log periodically. Additionally, a log-structured system typically needs large in-memory data structures to be able find data efficiently in the log (e.g. the inode map used in [34] and [35]).

Like log-structured systems, shadow paging only has to write data once. Additionally, unlike log-structured system, shadow paging does not require a cleaning process running in the background. The main performance drawback of shadow paging is that it needs a directory for mapping logical to physical block addresses. The overhead of such a directory depends largely on the implementation.

Concluding, when we look at the performance aspects relevant for EEPROM-based systems, logging with a separate log is the clear loser because it requires two writes for each update. Log-structured systems and shadow paging do not have this disadvantage.

In our scenario, log-structured has major disadvantages. First, our file system has to be implemented in an environment without an operating system, which makes it impossible to use a background process for cleaning the log. Second, the large memory footprint of log-structured systems is a big problem in environments as memory-constrained as our smartcard.

Shadow paging requires an efficient directory implementation. In chapter 8, we will describe how the characteristic properties of EEPROM can be used to achieve this efficiently.

Shadow paging has another significant advantage over logging. Logging requires some form of recovery after a system failure. Depending on the type of logging used (see section 2.6), the system has to undo or redo the effects of transactions that were in progress at the time the system failure occurred. Depending on the number of active transactions, this recovery process can take a significant amount of time. Time, during which the file system is offline, i.e. it cannot be used.

Shadow paging, on the other hand, does not need any recovery. No information is ever overwritten before a transaction commits. Therefore, after a system failure, the system is in the same state as it would be, when the transactions that were in progress at the time the system failure occurred, never happened.

6.4 Compatibility

Providing an API compatible with other interfaces like Windows for Smart Cards or JavaCard is not so difficult from a software engineering point of view. This can be achieved by adding a compatibility layer that converts calls to one of those interfaces to calls to the native SCFS interface.

6.5 Conclusions

Our high reliability requirements make a transaction-based approach most suitable to provide resilience to system failures. Of the various transaction-based approaches, shadow paging appears to be best suited to the characteristic hardware capabilities of a smartcard.

Access control lists can be used to define fine-grained security policies. Protection against offline attacks is not necessary for a smartcard file system, since smartcard are already equipped with ample hardware counter-measures to such attacks.

Backward compatibility with existing file system APIs can be achieved by implementing a separate interface layer.

We will use the results of this analysis in chapters 7-10, when we present the design and implementation of our smartcard file system.

Design and Implementation

In the preceding chapters, we arrived at a number of techniques that will be used in the design of our smartcard file system. Based on these finding we have designed a smartcard file system. In the following chapters, we will describe the design and implementation of this smartcard file system. For brevity, we will refer to this smartcard file system as SCFS.

7 Architecture

In this chapter, we provide a description of the architecture of SCFS. In the next chapters, we will describe each SCFS layer in more detail.

The SCFS design consists of several layers, as shown in Figure 3 below. The lowest layer, the *storage layer*, is responsible for reliably storing blocks of data. The next layer, the file system layer, uses those blocks to implement the components of the file system itself, like file, directories, and access control lists. The highest layer, the *interface layer* allows the implementation of multiple application programming interfaces (APIs), like a native interface and backwards-compatible interfaces.

This separation into layers leads to a flexible system, which, for example, makes it possible to implement multiple APIs on top of the core file system. Furthermore, it makes the implementation easier and cleaner. For example, the file system layer does not have to worry about how to implement data consistency, since this task is completely handled by the storage layer.

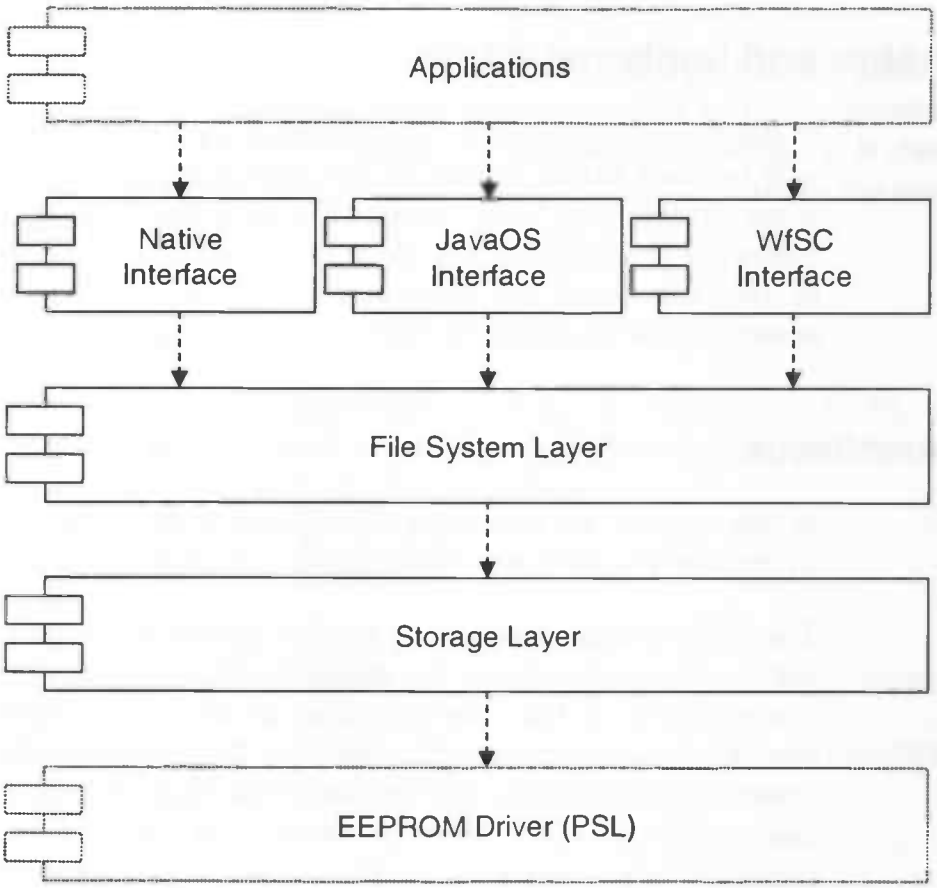

Figure 3 – SCFS layered architecture

7.1 Storage Layer

The storage layer provides the building blocks on which the actual file system is implemented. It provides a block device interface and a transaction mechanism.

7.1.1 Block Device

The storage layer abstracts the details of the underlying storage medium. It does this by presenting a block device interface. This interface consists of functions to allocate and deallocate blocks, and to read and write data stored in the blocks.

7.1.2 Transactions

All the operations that operate on the blocks are carried out within the context of a transaction. The transaction interface provides functions for starting transactions, for committing transaction (i.e.

make all changes made by a transaction permanent), and for aborting transaction (i.e. undoing all changes made by a transaction).

The transaction mechanism implemented in the storage layer provides a number of guarantees about its transactions:

- **Atomic**: A transaction is an atomic unit of processing; it is either performed in its entirety, or not performed at all.
- **Serializable**: Transaction should appear to be executed one after the other. Transactions should not interfere with each other, i.e. a transaction should not make its updates visible to other transactions until it is committed.
- **Durable**: The updates applied by a committed transaction must persist. These updates must never be lost because of any subsequent failure.

Atomicity means that the storage layer has to undo any changes made by transactions that were unable to complete, e.g. because a system failure occurred. Durability implies that all changes made by transactions that completed successfully, i.e. committed, must be permanent. In practice, this means that all updates made by a transaction must be written to the storage medium when the transaction commits. Serializability ensures that concurrently executing transaction do not interfere with each other.

The serializability and durability properties of the transaction mechanism allow the storage layer some flexibility about when it actually writes updates to the storage medium. More specifically, updates only have to be written when a transaction commits. This means that the storage layer can cache writes while a transaction is active.

7.2 File System Layer

The file system layer implements what can be viewed as the file system's *personality*, i.e. it defines the behaviour of the file system. The file system layer provides the following abstractions: files, directories, and access control lists (ACLs).

The file system layer provides a fairly low-level interface. This allows the interface layer (see below) maximum flexibility when implementing an existing API, like the JavaCard API [7], or the Windows for Smart Cards API [27].

7.3 Interface Layer

The interface layer uses the low-level interface provided by the file system layer to present a more convenient API to the applications. Additionally, the interface layer is responsible for security. It checks the validity of the arguments passed to it, and it enforces the access rights defined by the ACLs.

7.4 Conclusions

In this chapter, we have described the architecture of SCFS, and stated the tasks of each layer. In the following chapters, we will describe how each layer implements those tasks.

8 Storage Layer

In this chapter, we describe how the storage layer was implemented.

The storage layer implementation consists of three components: a space manager, a block cache, and a transaction manager.

8.1 Space Manager

The space manager component manages the free space and the allocation of blocks. The allocation and deallocation routines use a simple bitmap to keep track of free blocks. This bitmap is not stored in EEPROM but is reconstructed from the shadow paging directory (see section 8.3 below) when the file system is mounted.

Finding free blocks is done by a simple linear bitmap scan. Other, possibly more efficient, techniques were considered, including binary buddy allocation [18], and B-trees [37], but the small capacity of the EEPROM meant that allocating blocks was not a bottleneck.

8.2 Block Cache

The block cache component caches frequently accessed data. It is used by transactions to access the data blocks. In the current implementation, blocks are 16 bytes. This size corresponds with the block size of the EEPROM on the Infineon smartcard. The EEPROM block size dictates the minimum block size used by the storage layer. Storage layer blocks cannot be smaller than the native block size.

Reading and writing blocks in not implemented using read and write functions, but by a *pinning* interface: when a transaction wants to

access a block, is has to pin (lock) that block in memory. This tells the storage layer to keep that block in the cache. The transaction then receives a pointer to a cached version of that block, which it can use to access the block. When it has finished reading and writing the block, the transaction unpins the block. This pinning interface obviates the need to copy data between the cache and buffers used by the transaction.

The current implementation uses an LRU replacement algorithm to decide which blocks to remove from the cache when the cache is full.

8.3 Transaction Manager

The transaction manager component takes care of the atomicity, serializability, and durability of transactions. Atomicity and durability are ensured by shadow paging. Serializability is achieved by a locking mechanism.

8.3.1 Shadow Paging

The basic idea behind shadow paging is that modifications to a block are never written to the original block, but always to a copy of the original block, the *shadow copy*). Since the original data is never overwritten, there is no need for a rollback recovery when a transaction aborts, or when a failure occurs while a transaction is in progress.

The downside however is that the physical location of a block changes each time the block is updated. To hide this side effect from the upper layers, the storage layer maintains a directory containing a *logical-to-physical mapping* (LP-map). This directory maps logical block addresses to actual physical addresses. The LP-map is updated whenever the physical location of a block changes. Changing the physical address is transparent to the upper layers, since they always refer to a block by its logical address.

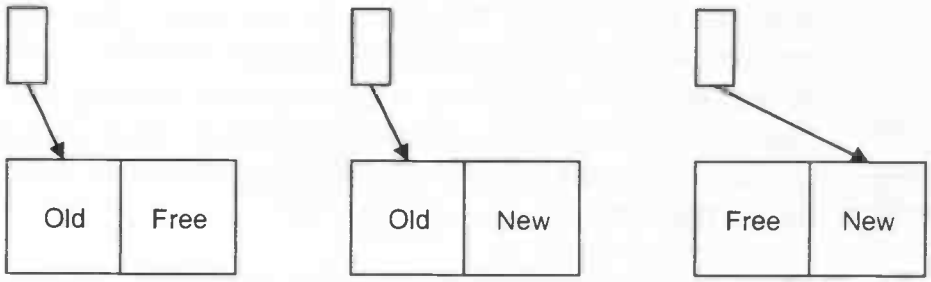

Figure 4 – Shadow Paging

Figure 4 visualizes this process. It shows one entry in the LP-map, and two blocks. Initially the LP-map entry points to the original block ("Old"). Then, a transaction wants to update that block, and writes new data to a shadow copy ("New") of the original block. Finally, when the transaction commits, the LP-map entry is updated to point to the new data and the old version of the block is available again.

LP-Map Implementation

An important issue that comes up when implementing shadow paging is how to update the LP-map. When a transaction commits, the LP-map is updated. However, this update has to be atomic because a system failure while updating the LP-map will result in an inconsistent system. This poses a problem, because doing atomic updates is exactly the problem we tried to solve by using shadow paging.

The original shadow paging implementation [21] kept two LP-maps, and a bit to indicate which LP-map is current. Various other approaches for implementing atomic LP-map updates are described in chapter 6 of [41].

SCFS currently uses a type of undo logging: before the updates to the LP-map are made, backups are made of the blocks that will be modified. If a system failure occurs during the update, the backups are used to undo any modifications made to the LP-map. The main motivation for the approach is that this mechanism is already implemented in the EEPROM driver of the Infineon smartcard. Additionally, this approach uses less space than keeping two entire copies of the LP-map. A drawback is that it requires twice as many writes because both the backups and the updates have to be written.

Currently, an entry in the LP-map is 16 bits wide. This means that the storage layer can hold a maximum of $2^{16} = 65536$ blocks. Given a

block size of 16 bytes, the maximum amount of data that can be stored by the storage layer is 1024 KB.

Remap

Transactions are serializable, i.e. updates made by a transaction, should only be visible to other transactions after the transaction commits. This means that each running transaction has a different view of the LP-map. The storage layer implements this by associating a private version of the LP-map to each transaction. Each time a transaction updates a block, the new logical to physical mapping is added to this private mapping, the *remap*.

When a transaction commits, the remap is merged with the global LP-map. Once this merge has completed, the commit operation has succeeded, and the changes made by the transaction are final. If a transaction is aborted, its remap is simply discarded, and any changes made by the transaction are thereby undone.

Example

The images below demonstrate how a block is modified using shadow paging. Elements in bold indicate changes between subsequent images.

Figure 5 shows the initial state of the file system. The LP-map indicates that logical block L1 maps to physical block P3, L2 maps to P1, and L3 and L4 have not been allocated. Physical block P1 contains 'DEF', P3 contains 'ABC', and P2 and P4 have not been allocated. The cache and remap are currently empty.

The data left of the dotted line is the *persistent* data; this data is stored in EEPROM. To the right of the dotted line is the *volatile* data, i.e. the data that is only used at run-time. This data is kept in RAM.

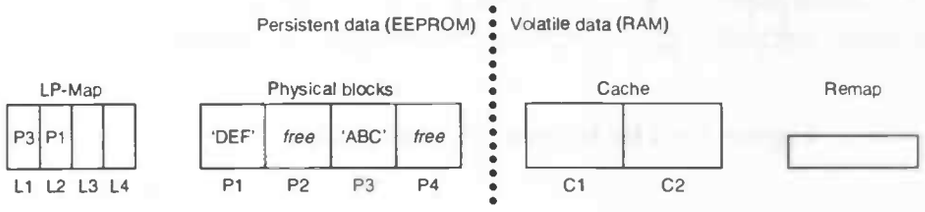

Figure 5 – Initial state

A transaction now wants to write 'XYY' to logical block L1, this involves the following steps:

1. Allocate new physical block (P2), as a shadow copy of P3.

2. Add mapping <L1, P2> to the remap.

3. Write <P2, 'XYY'> to the cache.

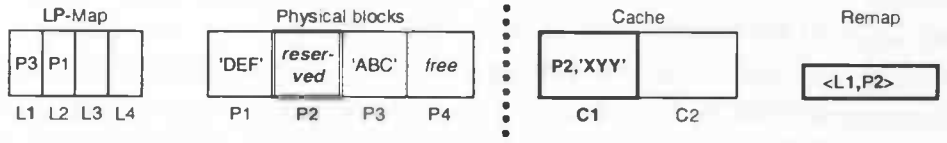

Figure 6 – The transaction writes 'XYY' to logical block L1.

The transaction changes its mind, and decides that it wants to write 'XYZ' instead to block L1. Since the physical block corresponding to logical block L1 is still in the cache, all that has to be done is to write the new value to the cache, i.e. no expensive NVM writes are necessary.

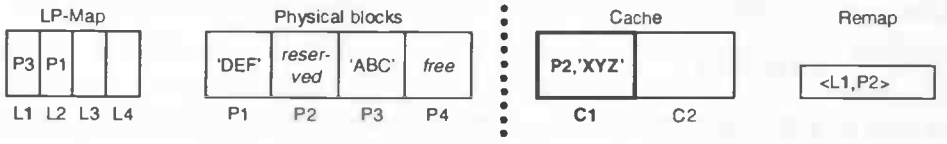

Figure 7 – The transaction writes 'XYZ' to logical block L1.

The transaction now commits. This involves the following steps:

1. Write the block in the cache to its corresponding physical block in NVM.

2. Update the LP-map to show the new mapping from the remap.

3. Free the old physical block P3.

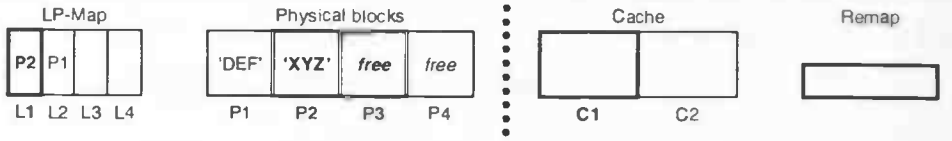

Figure 8 – The transaction commits.

8.3.2 Locking

In order to make transaction serializable, the storage layer uses locking. Each block has a lock associated with it. When a transaction T_1 wants to read or write a block, it first has to request the lock associated with that block. If no transaction currently holds the lock, then T_1 is granted the lock. If a transaction T_2 already has the lock, then T_1 has to wait until T_2 finishes and releases the lock.

The description above is a bit simplistic; in reality, the storage layer uses two types of blocks:

- read-lock (or shared lock)
- write-lock (or exclusive lock)

The advantage of this is that when transactions only want to read from a block, they can request a read-lock, which is a shared lock. Multiple transactions can acquire a shared lock on the same block simultaneously. This increases concurrency, and thereby efficiency.

When transactions release their locks is determined by the locking protocol. Our current implementation uses strict two-phase locking, which is a variation on two-phase locking (2PL). 2PL is a locking protocol that guarantees serializability. When 2PL is used, a transaction is not granted any more locks, once it has released a lock.

In practice, the transaction does not always know what operations it will perform next. In particular, when transactions are under the control of a higher layer, it is not known if the transaction has acquired all the locks it will need. As a result, it is also not known if any locks can be released yet.

Therefore, a variation of 2PL, strict 2PL, is more appropriate in our case. With strict 2PL, a transaction *T* does not release any of its write-locks until after it commits or aborts. Hence, no other transaction can read or write a block that is written by *T*, unless *T* has committed.

8.4 Conclusions

The storage layer provides the core functionality on which the actual file system is implemented. By combining shadow paging with a two-phase locking protocol, the storage layer can guarantee the atomicity, serializability, and durability of transactions. The transaction mechanism allows write caching, which has the potential to speed up the file system significantly.

9 File System Layer

In this chapter, we describe how the file system layer, which sits on top of the storage layer, is implemented.

The file system layer provides a UNIX-like file system. It shares many of its characteristics with UNIX and POSIX file systems:

- Case-sensitive naming: file names are case-sensitive, i.e. it is possible for distinct files with names 'file', 'File', and 'FILE' to coexist in the same directory.
- Hierarchical directory structure: the namespace of the file system is organized as directed acyclic graph (DAG). Although files can be referenced from multiple directories, directories themselves cannot. This avoids the creation of cycles in the namespace, simplifying enumerating (parts of) directory trees.
- Separation between storage and naming: files consist of two parts, the data contained in the file, and a name. The data associated with a file is described by an *inode*. The name is stored in a directory entry. This entry contains a reference to an inode, thereby linking file names with their associated data.
- Indexed storage allocation: the blocks containing the data associated with a file are described by an inode. The first few block addresses are stored in the inode itself. For larger files, with more blocks, the inode contains addresses of so-called indirect blocks. These blocks contain additional block addresses. To support even larger files, multiple levels of indirect blocks can be used.

9.1 Inodes

An SCFS inode is the same size as that block size provide by the storage layer, currently 16 bytes. The inode contains the size (in bytes) of the data associated with it. The size field is 16 bit, limiting the file size to $2^{16} = 65536$ bytes. There is an attributes field, also 16 bits wide. Twelve of these bits can be set by applications, the other four are reserved by SCFS, one to indicate the inode is a directory, one to indicate the inode is an ACL, and two reserved for future use. An inode also contains an ACL reference, indicating the ACL file associated with this inode. One byte is reserved for future use.

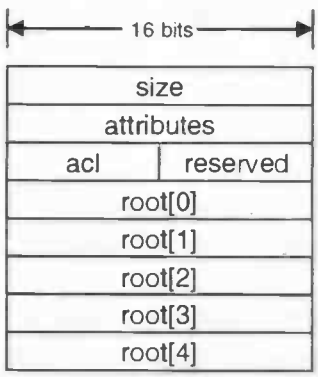

Figure 9 – Inode layout

An inode also contains an allocation forest with five pointers to allocation trees. Allocation forests are described in more detail in the next section.

9.1.1 Allocation Forests

Most UNIX file systems support a limited number of indirect blocks, triple indirection being the usual maximum. This puts a limit on the maximum file size that those file systems can support. In practice, this is not a restriction. Such file systems tend to use block sizes of at least 512 bytes. If a file system uses 4 bytes for each block address, then one indirect block can point to 128 blocks. Therefore, a triple indirect block can address files of up to $128^3 \cdot 512 = 1GB$ in size. When a block size of 2048 bytes is used, this number becomes 256GB. As a result, using no more than three levels of indirection imposes no practical limits on file sizes in such system.

In our case, the metrics differ significantly. A typical block size for the EEPROM memory found in smartcards is between 16 and 64 bytes. Even if fewer bytes are used for each block address, for example 2 bytes, an indirect block of 16 bytes can only point to eight blocks of storage. This means that a triple indirect block can only address files of up to $8^3 \cdot 16 = 8KB$. This is obviously not enough. Even adding another level of indirection only allows for files up to 64KB.

Placing artificial limits on the maximum level of indirection is clearly not acceptable in our case. This observation has led us to use a more flexible inode design.

An inode in an SCFS file system contains multiple entries for storing block addresses. Each entry can point to a data block, or to an indirect block of any number of indirections. The exact configuration of the inode can be chosen by the user when the file system is created.

Each inode contains a list of the blocks storing the data that is associated with the inode. Because an inode has a limited size, only a limited number of block addresses can be stored in an inode. To avoid limiting the file system to unrealistically small file sizes, the block list uses several levels of indirection.

In other words, each block address in an inode is the root of an allocation tree. The combination of these trees is called an allocation forest. Each tree has a specified level, i.e. the distance from the root of the allocation tree to the leaves (data blocks).

Level 0 means a direct block pointer, i.e. the pointer points directly to a data block of the file. Level 1 is an indirect block pointer, the pointer does not point to a data block, but to a block containing pointers to data blocks. Level 2 pointers point to blocks containing pointers to blocks containing pointers to data blocks, etc. Figure 10 shows an example inode with three allocation trees, of levels 0, 1, and 2 respectively.

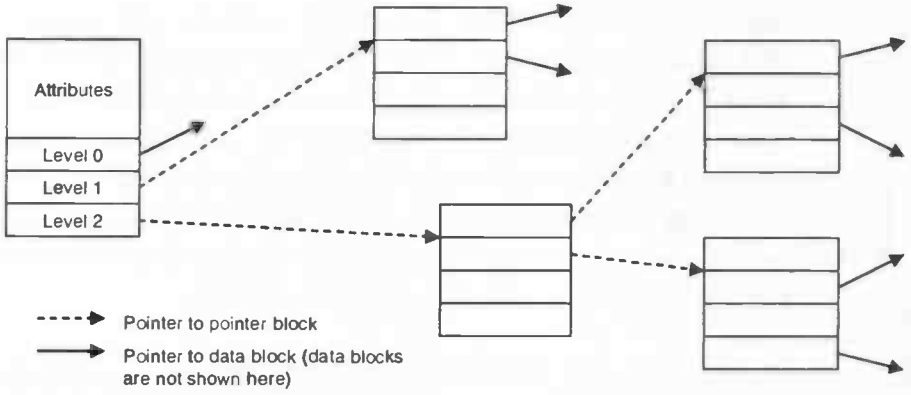

Figure 10 – An inode with an allocation forest of height 3

9.2 Directories

A directory is a special file – indicated by a flag in the attributes field of the inode. A directory has a number of entries, each containing a file name and a pointer to an inode.

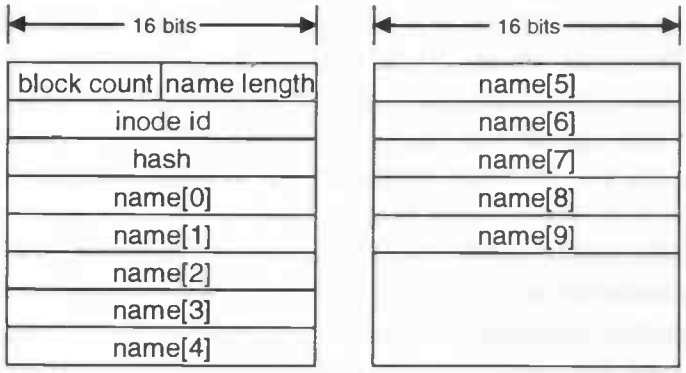

Figure 11 – Directory entry layout for an entry with a file name of length 10

A directory entry consists of one or more blocks, depending on the length of the file name. The first block contains information about the number of blocks occupied by this entry, the length (in characters) of the file name. It also contains the logical block number of the inode

associated with this directory entry. Additionally, it contains a hash (16 bits) of the file name. This hash is used to speed up directory lookup operations: if the hash of the desired file name differs from the hash in the directory entry, then this entry can be skipped immediately. The name of the file is stored as a string of Unicode characters [39]. The characters are encoded using UTF-16. The first block can contain five characters. If the name is longer, additional blocks are used.

9.3 ACLs

As we saw in section 9.1, each inode contains a reference to an ACL. The ACL contains a list of rules that specify the permissions for that inode. The file system layer does not check the permissions defined in ACLs when it performs file system operations. It only provides mechanisms for modifying and querying permissions. It is the responsibility of a higher layer to check access permissions when it requests the file system layer to perform operations. This approach allows a higher layer to decide what access permissions it wants to check.

As we mentioned above, an ACL contains a list of rules. Each rule consists of operation, and a Boolean expression. The Boolean expression consists of user IDs and the Boolean operators AND and OR. If the Boolean expression evaluates to true, the operation is permitted.

To determine if a certain operation is allowed on an inode, the file system evaluates the appropriate expression, based on the currently authenticated users[5].

Figure 12 below shows the layout of an ACL rule. It contains the operation to which the rule applies, the type and length of the Boolean expression, and the expression itself. An expression (see Figure 13) consists of one or more sub-expressions. Each sub-expression is a list of user IDs.

The type of the expression determines how the expression is evaluated. SCFS supports two types of expression: disjunctive and conjunctive expression. For disjunctive expressions, the AND operator is used to evaluate each sub-expression, and the result of

[5] The file system keeps a list of currently authenticated users. However, the file system itself does not authenticate users. This is handled by another part of the operating system. The file system merely provides functions that the operating system can use to tell the file system which users are currently authenticated.

the expression is the OR of the results of each sub-expression. For conjunctive expressions, it is the other way around.

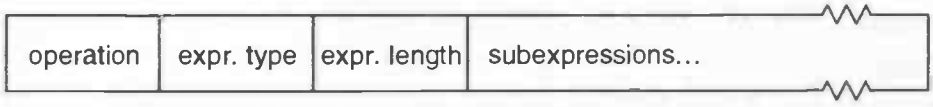

Figure 12 – ACL rule layout

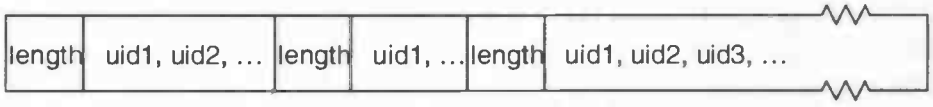

Figure 13 – ACL expression layout

9.4 Conclusions

In this chapter, we have described the design and implementation of the file system layer. The file system layer provides the basic operations for reading and writing files, creating and deleting files and directories, and storing and querying access rights in ACL files.

10 Interface Layer

In this chapter, we describe the uppermost layer of SCFS: the interface layer.

The interface layer defines the interface between the file system and the applications that use it. It is to only part of SCFS that is directly accessed by applications. Therefore, the main task of the interface layer is to act as a security barrier. The interface layer checks access right (using ACLs), and validates the parameters passed to its functions.

The other goal of the interface layer is to allow the implementation of multiple application programming interfaces (APIs). There are currently two implementations of the interface layer. One implementation, the native interface, presents an API compatible with that of the other drivers in Infineon's Platform Support Library,

or PSL[6]. The other, the WfSC interface, implements the file system related functions of Microsoft's Windows for Smart Cards API.

The biggest difference between the two implementations is how they handle transaction. The WfSC API does not specify a transaction mechanism. Therefore, the WfSC layer uses a separate transaction for each function call made. Another option would have been to start a transaction when a file is opened, and to commit the transaction when the file is closed. As we shall see in the evaluation, larger transactions mean that more updates can be cached, which improves performance. Unfortunately, this approach means that when a system failure occurs before an application closes a file; any updates made to that file are lost. This behaviour does not match the WfSC specifications, which state that the results of each successfully completed function call should be permanent.

The native interface on the other hand exposes the transaction mechanism. Hence, applications can define what constitutes a transaction. This allows applications maximum flexibility and the greatest opportunities for maximizing performance.

10.1 Conclusions

The interface layer defines the behaviour of the file system. We have successfully implemented two different APIs, thereby demonstrating the flexibility of the file system layer.

In the preceding chapters, we have described the design and implementation of SCFS. By separating the system into a number of layers, each with its own responsibilities, we have achieved a design that can be extended or modified easily. The storage layer provides a transaction mechanism for modifying blocks of data. The file system layer uses this to implement file system primitives. On top of the file system layer sits the interface layer, which provides multiple high-level file system interfaces.

In the next part, chapters 11 and 12, we will evaluate our implementation.

[6] The PSL is a library of device drivers supplied by Infineon. The PSL is intended as a hardware abstraction layer below the operating system. It includes, amongst others, drivers for the EEPROM, a random number generator, and encryption routines.

Evaluation

In these chapters, we evaluate to what extent our implementation meets the requirements we specified in section 1.5. We present a quantitative analysis of SCFS performance, based on a number of micro-benchmarks. Furthermore, we describe how SCFS meets the other requirements.

11 Performance

It is difficult to compare SCFS's performance to other file systems directly. Besides the fact that SCFS provides higher reliability than other file systems, which gives it a performance disadvantage, there are simply no other file system implementations available on the Infineon smartcard on which SCFS currently runs. Because a comparison to other file systems was not possible, we have benchmarked a number of performance indicators. We present the results of those benchmarks in this chapter.

As we noted in section 5.1, writing to EEPROM is many orders of magnitude slower than reading from it. Furthermore, access times are negligible. This means that the performance of SCFS will be determined largely by amount of data that is written. Other factors that are relevant in 'regular' file systems, such as the amount of data that is being read, or the physical layout of the data are of minor importance.

Two factors determine the amount of data that needs to be written to perform a file system operation: metadata overhead and caching.

11.1 Metadata Overhead

Metadata comes in two varieties, user metadata, and system metadata. Examples of user metadata include file names (directories) and ACLs. System metadata consists of elements like inodes and indirect blocks.

The user can largely determine the size of the user metadata. It can be reduced e.g. by using shorter file names, or by letting multiple files share one ACL file. The size of the system metadata is determined by the number and size of the files. The two figures below give some indications of the metadata overhead in SCFS.

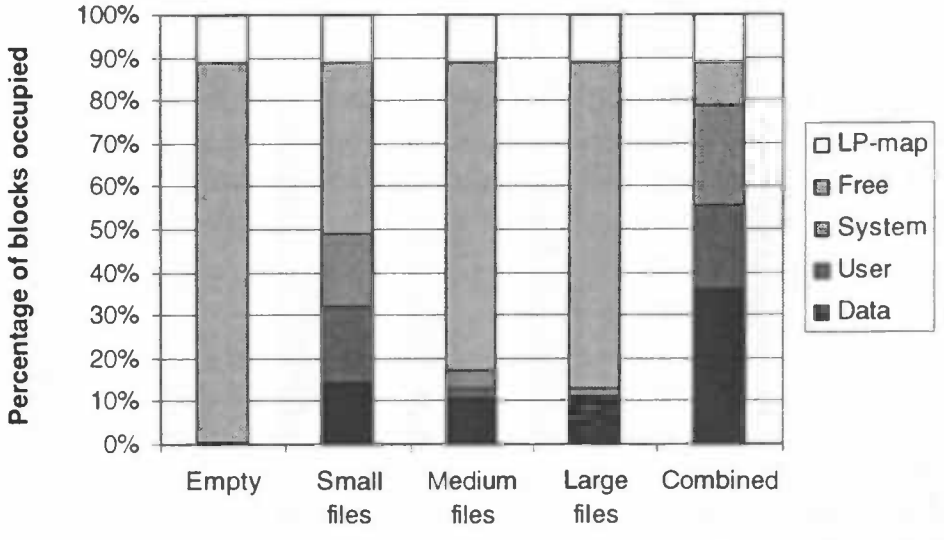

Figure 14 – Percentage of blocks occupied by each metadata category

Figure 14 shows the relative sizes of the different metadata types in some typical file system configurations. Each file system consists of 2048 blocks of 16 bytes each. The LP-map takes up an additional 256 blocks (2048 entries of 2 bytes each).

The 'Empty' file system configuration only contains a root directory and one ACL file. The 'Small files' configuration contains one subdirectory with 200 files, each between 10 and 30 bytes. The 'Medium files' configuration contains a subdirectory with 20 files, each between 150 and 250 bytes. The 'Large files' configuration contains a subdirectory with two files, each between 1900 and 2100 bytes. All three configurations store approximately 3800 bytes. All files have a file name of length five, and share a single ACL file. The 'Combined' configuration contains all the files in the 'Small files', 'Medium files' and 'Large files' configurations.

The first thing that is obvious from Figure 14 is that large files have a smaller total metadata overhead (as indicated be the larger bar representing the free space). Furthermore, user and system metadata have comparable sizes.

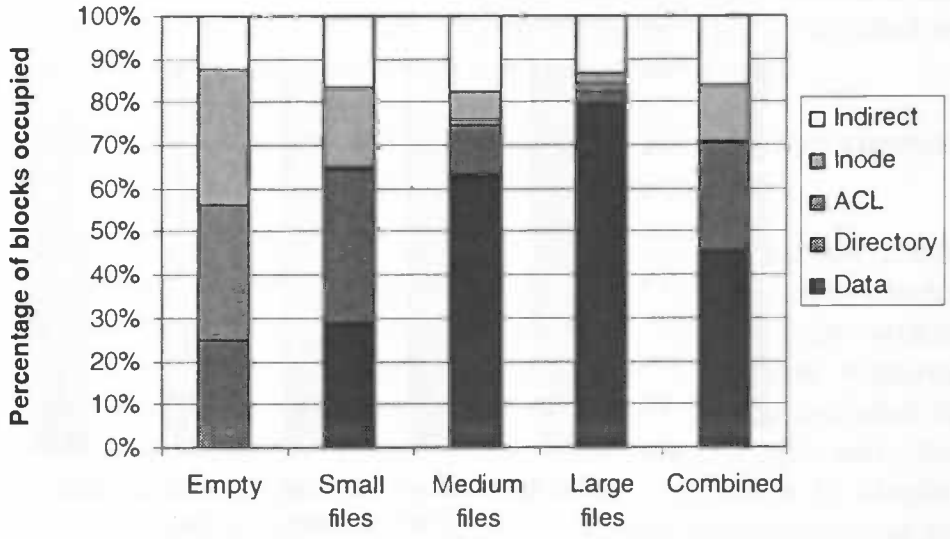

Figure 15 – Percentage of blocks occupied by each metadata category

Figure 15 shows a more detailed view of some of the relevant aspects from Figure 14. The overhead of the LP-map and the free space have been removed from the graph, and the user and system metadata categories have been split into ACLs and directories, and inodes and indirect blocks respectively.

This graph shows that, for small files, inodes and indirect blocks contribute evenly to the amount of system metadata overhead. Large files obviously use less inodes.

Obviously, the amount of user metadata overhead depends heavily on factors beyond the control of the file system, like the length of file names, and the number of different ACL files used. In general, we can say that the amount of space occupied by ACL files is small, while the space occupied by directory entries depends on both the number of files, and the length of their names.

Concluding, system metadata consumes between 10% and 35%. In total, metadata amounts to between 15% and 70% of the total amount of data, but this figure depends heavily on factors determined by the users, not the file system itself. Additionally, the LP-map consumes 11% of the blocks allocated to the file system. This does not depend on the actual amount of data stored in the file system.

11.2 Caching

Another crucial factor in file system performance is how much of the writes can be avoided by caching. Caching works by keeping

updated data in RAM for a while, anticipating that that data will be written more times. If that happens, the updates can take place in the (much faster) RAM, and only have to be written to EEPROM once. Multiple updates to the same block of data can occur in many situations. The most obvious example is when an application does this explicitly. Another example occurs when data is appended to a file. Each time this happens, the file size in the inode has to be updated.

11.2.1 Effects of Transaction Granularity

The reliability requirements mean that data cannot be cached indefinitely; caching is restricted to caching writes done within a transaction, because all updates made by a transaction have to be written to EEPROM when the transaction commits. Therefore, caching is only worthwhile when a block of data is updated multiple times within a transaction. Since the chances of a block of data being updated multiple times increase when transactions are larger, the granularity of transactions is important for file system performance.

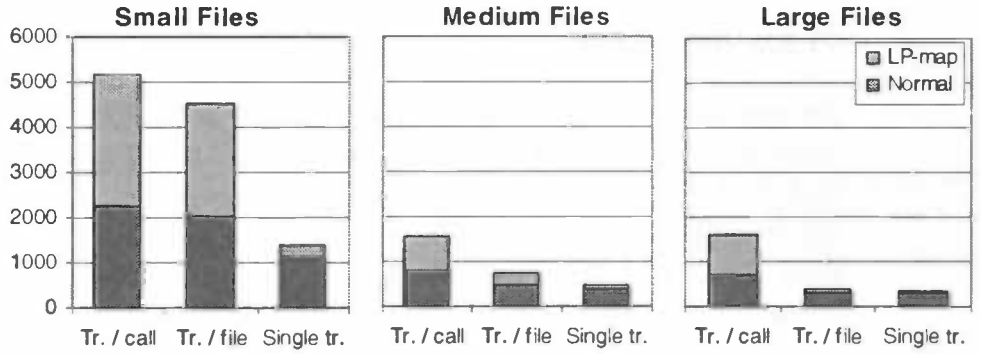

Figure 16 – Effects of transaction granularity on performance

Figure 16 shows how the transaction granularity affects efficiency of the cache. The three graphs show the results of three benchmarks. The benchmarks consisted of writing the file system configurations 'Small files', 'Medium files' and 'Large files' respectively. The files in each configuration were written one by one, in steps of 32 bytes.

Each benchmark was run with three different transaction granularities:

- Transaction/call: a separate transaction for each function call
- Transaction/file: a transaction for each file written
- Single transaction: a single transaction for the entire benchmark

The bars in each graph show how many blocks of data were written to EEPROM during a particular benchmark. Each bar is divided in updates to normal data blocks and updates to the LP-map.

The benchmarks confirm our observations in the previous chapter, namely that large files are more efficient than small files.

The benchmarks also illustrate that large transactions are more efficient than small transactions. One reason is that less data blocks are written when large transactions are used, because more writes can be cached. Another reason is that each transaction requires updating the LP-map. The amount of data that has to be updated in the LP-map is only one eighth of the normal data (a 2 byte entry for each 16 byte block), but writes to EEPROM are always to blocks. Therefore, updating a single 2-byte LP-map entry requires writing an entire 16-byte block. Large transactions on the other hand tend to have a certain amount of locality in their updates. Therefore, LP-map entries tend to be concentrated in a few EEPROM blocks. This drastically reduces the amount of LP-map blocks that need to be written.

11.2.2 Write Cost

As we stated above, the deciding factor in SCFS's performance is amount of data that needs to be written to EEPROM. A way to quantify this factor is by calculating the *write cost*. The write cost is the average number of bytes written to EEPROM for each byte of user data. The term user data here includes both the actual data stored in files, and user metadata such as directories and ACL files.

Table 4 below shows the average write cost for each of the benchmarks described in the previous section. This gives an idea of the overhead caused by the system metadata and shadow paging.

	Tr. / call	Tr. / file	Single tr.
Small files	6,99	6,10	1,90
Medium files	5,28	2,47	1,60
Large files	6,29	1,48	1,42

Table 4 – Average write cost

To gain more insight in what parts of the file system incur the highest overhead, the two tables below show what part of the overhead is caused by the file system layer, and what part is caused by the storage layer.

These two tables show that for small transactions, the storage layer (i.e. shadow paging) is the main cause of the overhead. For larger transactions, the storage layer and file system layer contribute evenly to the total overhead.

	Tr. / call	Tr. / file	Single tr.
Small files	1,54	1,54	1,54
Medium files	1,32	1,32	1,32
Large files	1,19	1,19	1,19

Table 5 – Average write cost of the file system layer

	Tr. / call	Tr. / file	Single tr.
Small files	4,55	3,97	1,24
Medium files	4,01	1,88	1,21
Large files	5,29	1,25	1,20

Table 6 – Average write cost of the storage layer

11.3 Conclusions

The metadata overhead in SCFS can be significant when file sizes are small. For larger files, the overhead is considerably smaller. In our recommendations for future work, we will suggest a number of ways to reduce metadata overhead.

When transactions are larger, more blocks can be cached, which improves performance considerably. This is largely caused by the overhead of the shadow paging mechanism. We have added transaction extensions to our implementation of the WfSC API, to enable applications to use larger transactions.

12 Other Requirements

In this chapter, we provide some comment on how SCFS meets the requirements that could not be quantified.

12.1 Reliability

The first and foremost requirement for the smartcard file system was reliability, i.e. resilience to system failures. This goal has been achieved by using a transaction mechanism based on shadow

paging. By exposing the transaction mechanism in the API, applications can define the scope of the transactions. For example, this allows an application to ensure consistency, even when the related data is spread out over multiple files. Furthermore, it allows larger amounts of data to be cached, which is beneficial for performance.

12.2 Security

The use of ACL files to define permissions allows for a high level of security. ACL files allow a fine-grained specification of permissions. This means that only permissions that are absolutely necessary have to be granted.

12.3 Compatibility

By implementing multiple interface layers, SCFS can provide the API's of other file systems or smartcard platforms. In addition to a native interface, we have implemented the Windows for Smart Cards file system API. As we saw in chapter 10, WfSC does not provide a transaction mechanism. This limits the scope of transactions to a single function call, which, as we saw above, is detrimental to performance. To solve this, we have implemented an extension to WfSC that exposes the transaction mechanism. This allows applications to take full advantage of the transaction capabilities of SCFS.

12.4 Conclusions

By combining a number of methods that we described and analyzed in chapters 2-6, SCFS is able to meet the reliability, security, and compatibility requirements we set for a smartcard file system.

Conclusion

As smartcards become increasingly powerful, multi-application cards are used more and more. This trend gives rise to the need for a smartcard file system. However, smartcard file systems have requirements distinctly different from traditional file systems. Furthermore, the hardware limitations of current-generation smartcards, combined with the unusual characteristics of EEPROM, mean that conventional approaches to file system design are not always the best approaches for a smartcard file system.

In the introduction, we defined a set of requirements that a smartcard file system should meet.

In chapters 2-4, we presented a survey of existing file systems designs, and described a variety of methods and techniques from those designs that can be used to meet the requirements we defined.

In chapters 5 and 6, we analyzed those methods and techniques, in order to determine which of those are most applicable to the unique circumstances of a smartcard file system.

Based on this analysis we designed and implemented a smartcard file system, which we presented in chapters 7-10.

In chapters 11 and 12, we evaluated the performance of our smartcard file system, and we commented on how this file system meets the requirements we set for it.

13 Contributions

The main contributions of this thesis are:

- A comprehensive review of methods and techniques to provide reliability, security, and high performance in file systems in general, and smartcard file system in particular. Our main finding was that shadow paging, which was previously written off because of its bad performance, turned out to provide better performance on EEPROM storage media than comparable methods, such as logging.

- The design and implementation of a file system, SCFS, tailored to the unique characteristics of smartcards. The main features of SCFS are, a transaction mechanism to provide the reliability needed by a smartcard file system, caching facilities for improved performance, flexible ACL-based security, and support for multiple file system APIs.

14 Future Work

In this section, we list a number of suggestions to improve the SCFS smartcard file system.

14.1 Reliability

One aspect that we did not yet consider is how to provide resilience against media failures. As the name implies, media failures are failures of – a part of – the storage medium.

EEPROM blocks have a limited lifetime. Each block can only be written a limited number of times, in case of the Infineon smartcard circa 500,000 times. Although this number is probably enough to last the entire lifetime of a typical smartcard, it is worth to consider the impact of this.

One technique to cope with this is to store redundant information that can be used to recover the data stored in a dead EEPROM block. Another approach is to use wear-levelling, i.e. the file system tries to spread the writes evenly over the entire file system area, thereby avoiding that certain data hotspots reach the end of the lifespan long before other parts.

14.2 Performance

The main opportunity for improving SCFS performance is by reducing the overhead introduction by the file system. A number of techniques could be implemented to reduce the number of blocks used by file system data:

- **Extents.** The current implementation of the file system layer used index blocks to describe the blocks containing the data that belongs to a file. Another way to represent this information is by using extent pointers. The basic idea is to allocate blocks in large contiguous extents. Instead of storing a list of block numbers, the data can now be described by tuples of <block number, length>. As long as extents have a length of at least two blocks, this representation reduces the space needed to describe the blocks containing file data.
- **Embedded inodes.** Another technique is by using embedded inodes [11]. Instead of storing file information in inode blocks, this information is stored directly in the directory entry. This provides a slight reduction in the per-file overhead.

Bibliography

[1] Baker, Mary, Satoshi Asami, Etienne Deprit, John K. Ousterhout, and Margo Seltzer. "Non-volatile memory for fast, reliable file systems." *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)* 27.9 (1992): 10–22.

[2] Card, Rémy, Theodore Ts'o, and Stephen Tweedie. "Design and Implementation of the Second Extended Filesystem." *Proceedings of the First Dutch International Symposium on Linux*, 1994.

[3] Challis, Michael F. "Database Consistency and Integrity in a Multi-User Environment." *Databases: Improving Usability and Responsiveness*. Ed. Ben Shneiderman. Haifa, Israel: Academic Press, 1978: 245-270.

[4] Chamberlin, Donald D. et al. "A History and Evaluation of System R." *Communications of the ACM* 24.10 (1981): 632-646.

[5] Chang, Albert, Mark F. Mergen, Robert K. Rader, Jeffrey A. Roberts, and Scott L. Porter. "Evolution of Storage Facilities in AIX Version 3 for RISC System-6000 Processors." *IBM Journal of Research and Development* 34.1 (1990): 105-110.

[6] Chao, Chia, Robert English, David Jacobson, Alexander Stepanov, and John Wilkes. "Mime: A High Performance Parallel Storage Device with Strong Recovery Guarantees." *Technical Report HPL-CSP-92-9rev1*. Hewlett-Packard Laboratories, 1992.

[7] Chen, Zhiqun. "JavaCard™ Technology for Smart Cards: Architecture and Programmer's Guide." Reading, MA: Addison-Wesley, 2000.

[8] Chutani, Sailesh, Owen T. Anderson, Micheal L. Kazer, Bruce W. Leverett, W. Anthony Mason, and Robert N. Sidebotham. "The Episode File System." *USENIX Annual Technical Conference*, 1992: 43-60

[9] Duncan, Ray. "Advanced MSDOS Programming." Redmond, WA: Microsoft Press, 1986.

[10] Elmasri, Ramez, and Shamkant B. Navathe. "Fundamentals of Database Systems," Third Edition. Reading, MA: Addison-Wesley, 2000.

[11] Ganger, Gregory R., and M. Frans Kaashoek. "Embedded Inodes and Explicit Grouping: Exploiting Disk Bandwidth for Small Files." *USENIX Annual Technical Conference*, 1997: 1-17.

[12] Ganger, Gregory R., Marshall Kirk McKusick, Craig A. N. Soules, and Yale N. Patt. "Soft Updates: A Solution to the Metadata Update Problem in File Systems." *ACM Transactions on Computer Systems* 18.2 (2000): 127-153.

[13] Ganger, Gregory R., and Yale N. Patt. "Metadata update performance in file systems." *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation*, 1994: 49-60.

[14] Giampaolo, Dominique. "Inside the BeOS; Modern File System Design." San Mateo, CA: Morgan Kaufmann Publishers, 1998.

[15] Hitz, Dave, James Lau, and Micheal Malcom. "File System Design for an NFS File Server Appliance." *USENIX Winter Technical Conference*, 1994: 17-21.

[16] Infineon Technologies AG. "SLE 88CX720P Chip Card IC Datasheet." 3 December 2003 < http://www.infineon.com/cgi/ecrm.dll/ecrm/scripts/public_download.jsp?oid=28834&parent_oid=29212>.

[17] International Standardization Organization (ISO). "Integrated Circuit(s) Cards with Contacts." ISO/IEC 7816, 1995.

[18] Knuth, Donald E. *The Art of Computer Programming, Vol. 1, Fundamental Algorithms*. Reading, MA: Addison-Wesley, 1969.

[19] Kömmerling, Oliver, and Markus G. Kuhn. "Design Principles for Tamper-Resistant Smartcard Processors." *USENIX Workshop on Smartcard Technology*, 1999: 9-20.

[20] Lampson, Butler W. "Protection." *Operating System Review*, 8.1 (1974): 18-34.

[21] Lorie, Raymond A. "Physical Integrity in a Large Segmented Database." *ACM Transactions on Database Systems*, 2.1 (1977):91-104.

[22] Maxtor Corporation. DiamondMax D740X Datasheet. October 13, 2003 <http://www.maxtor.com/en/documentation/data_sheets/diamondmax_D740X_datasheet.pdf>

[23] Maosco Ltd. "MULTOS." April 17, 2003 <http://www.multos.com>.

[24] McCoy, Kirby. *VMS File System Internals*. Newton, MA: Digital Press, 1990.

[25] McKusick, Marshall K., and T. Kowalski. "Fsck – The UNIX file system check program." *4.4 BSD System Manager's Manual*. Sebastopol, CA: O'Reilly & Associates, 1994: 3-21.

[26] McKusick, Marshall K., William N. Joy, Samuel J. Leffler, and Robert S. Fabry. "A Fast File System for UNIX." *ACM Transactions on Computer Systems*, 2.3 (1984): 181-197.

[27] Microsoft Corporation. "Microsoft Windows for Smart Cards." *The Microsoft Windows Smart Card Toolkit*, 1999.

[28] Miller, Ethan L., Scott A. Brandt, and Darrell D. E. Long. "HeRMES: High-Performance Reliable MRAM-Enabled Storage." *The 8th IEEE Workshop on Hot Topics in Operating Systems*, 2001.

[29] Namesys. "Reiser4." 12 March 2003 <http://www.namesys.com/v4/v4.html>.

[30] Ousterhout, John K., Herve Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. "A Trace-Driven Analysis of the Unix 4.2 BSD File System." *Proceedings of the 10th Symposium on Operating Systems Principles*, 1985: 15-24.

[31] Ousterhout, John K., and J. Douglis. "Beating the I/O Bottleneck: A Case for Log-Structured File Systems." *Operating Systems Review* 23.1 (1989): 11-27.

[32] Powell, Micheal L. "The DEMOS File System." *Proceedings of the Sixth Symposium on Operating Systems Principles*, 1977: 33-42.

[33] Ritchie, Dennis M., and Ken Thompson. "The UNIX Time-sharing System." *The Bell System Technical Journal*, 57.6 (1978): 1905-1930.

[34] Rosenblum, Mendel, and John K. Ousterhout. "The Design and Implementation of a Log-Structured File System." *ACM Transactions on Computer Systems*, 10.1 (1992): 26–52.

[35] Seltzer, Margo, Keith Bostic, Marshall K. McKusick, and Carl Stealin. "An Implementation of a Log-Structured File System for UNIX." *Proceedings of the USENIX Winter Technical Conference*, 1993: 201-220.

[36] Solomon, David A., and Mark E. Russinovich. "Inside Microsoft Windows 2000," Third Edition. Redmond WA: Microsoft Press, 2000.

[37] Sweeney, Adam, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geo Peck. "Scalability in the XFS File System."

Proceedings of the USENIX 1996 Annual Technical Conference, 1996: 22-26.

[38] Tweedie, Stephen C. "Journaling the Linux ext2fs Filesystem." *LinuxExpo '98*.

[39] The Unicode Consortium. The Unicode Standard Version 2.0. Reading, MA: Addison-Wesley, 1996.

[40] Wu, Michael, and Willy Zwaenepoel. "eNVy: A Non-Volatile, Main Memory Storage System." *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1994: 86-97.

[41] Tatu Ylönen. "Shadow Paging is Feasible." PhD Thesis, Helsinki University of Technology, 1994.

Glossary

ACL Access Control List, a list of (principal, permission) pairs that define the permissions for an object. A list of (user, permission) pairs that defines which access rights each user has to a particular object, such as a file, or directory.

API Application Programming Interface. The interface (calling conventions) by which an application program accesses operating system and other services.

EEPROM Electrically Erasable Programmable Read-Only Memory. A memory chip that retains data content after power has been removed. EEPROM can be erased and reprogrammed within the computer or externally.

ext2 Second Extended Filesystem, a Linux file system with a traditional file system layout, optimized for speed. See also [2]

ext3 Third Extended Filesystem, a Linux file system that improves the reliability of ext2 by using journalling. See also [38].

FAT File Allocation Table. A table used to keep track of disk storage and file allocation. Used by the FAT8, FAT12, FAT16, and FAT32 file systems used in MS-DOS and earlier versions of Windows. See also [9].

FFS Fast File System. A file system used in various BSD operating systems. FFS is based on traditional UNIX file systems like UFS, and uses clustering to improve performance. See also [26].

GSM Global System for Mobile Communications. A European standard for digital cellular radio communication (mobile phones).

ISO International Organization for Standardization. An international body responsible for creating international standards in many areas, including computers and communications

JFS Journaled File System. A file system developed by IBM, which uses journaling to protect metadata. Originally used in AIX and later ported to Linux. See also [5].

Mime A disk array device developed by Hewlett-Packard that uses shadow paging to provide strong recovery guarantees. See also [6].

NTFS New Technology File System. A file system developed by Microsoft for its Windows NT operating system. See also [36].

NVRAM Non-Volatile Random Access Memory. Any type of memory that is made non-volatile by connecting it to a constant power source, such as a battery. Therefore, non-volatile memory does not lose its contents when the main power is turned off.

POSIX Portable Operating System Interface. The name for a series of standards developed by the IEEE that specify a Portable Operating System interface.

PSL Platform Support Library. A library of support functions and hardware abstraction, supplied by Infineon with its 88 series smartcards.

RAM Random Access Memory. The working memory of a computer where data and programs are temporarily stored. RAM only holds information when the computer is on.

ROM Read-only Memory. A memory system that stores information permanently and does not lose its contents when power is switched off.

SIM Subscriber Identity Module. A smart card commonly used in a GSM phone. The card holds a chip that stores information and encrypts voice and data transmissions, making it close to impossible to listen in on calls. The SIM card also stores data that identifies the caller to the network service provider.

UNIX An operating system that supports multi-user and multitasking operations, developed by Kenneth Thompson and Dennis Ritchie of Bell Labs in 1969.

VMS Short for Virtual Memory System, a multi-user, multitasking, virtual memory operating system that runs on DEC's VAX and Alpha lines of minicomputers and workstations. VMS was introduced in 1979 along with the first VAX minicomputer.

WAFL Write Anywhere File Layout. A file system used in NetApp's NFS server appliance. Uses shadow paging for reliability and snapshots. See also [15].

WfSC Windows for Smart Cards. A scaled down version of Windows for use on smart cards.

XFS eXtended File System. A file system designed by SGI for its IRIX operating system, designed to be scalable and efficient. Uses journaling for metadata protection.