Bachelor's Thesis

File Systems and Usability — the Missing Link

Robert Freund (rfreund@uos.de)

Cognitive Science, University of Osnabrück

July 2007

Supervisors:

PD Dr. habil. Helmar Gust

Prof. Dr. Kai-Uwe Kühnberger

Abstract

Usability has never been a priority in file system design. Instead, developers focus mainly on technical aspects, such as speed, reliability, and security. But in recent decades, technological innovations have created a modern information crisis. This is characterized by an ever-growing abundance of easily accessible information. Additionally, the user is able to create and store continuously increasing amounts of digital data. This data is usually managed on the user's personal computer. Conventional file systems, however, which constitute the most important systems for document management tasks, impose a strict monohierarchy onto the user's document collection. The user is constrained by the file system's inability to represent multiple categorizations of documents without utilizing band-aid solutions such as shortcuts. As a possible approach to these issues, a file system prototype is portrayed that permits a nonhierarchical directory structure and polyhierarchical file categorizations while maintaining backward compatibility with existing applications. Furthermore, suggestions are made for carrying out a usability evaluation which could be used to measure the achieved improvement compared to conventional file systems.

Contents

1 Introduction **1**

1.1 Scope and Content of this Thesis 1

1.2 Motivation 1

1.3 What is Wrong with Monohierarchical File Systems 4

1.4 History of File Systems 8

2 Related Research and Products **11**

2.1 Approaches Using Keyword Based Search 11

2.2 Solutions for Specific Problems 12

2.3 DocPlayer 13

2.4 Lifestreams 14

2.5 File System Based Solutions 15

2.5.1 Semantic File Systems 16

2.5.2 WinFS 17

3 NHFS — a "Nonhierarchical" File System **20**

3.1 Goals 20

3.2 Structure and Semantics of File System Elements 21

3.2.1 Monohierarchical (Conventional) File Systems 21

3.2.2 NHFS 23

3.3 Extending the File System's User Interface 23

3.4 Implementation 25

3.5 Usage 30

3.6 Technical Evaluation 32

3.7 Important Features not yet Implemented 33

4 Evaluating the Usability of NHFS **36**

4.1 General Usability Considerations 36

4.2 Usability Evaluations in the File System Context 38

4.2.1 Aspects to Evaluate 38

4.2.2 The Metaphor Problem 39

4.2.3 Design Proposals for Usability Tests 39

5 Outlook and Conclusion **42**

5.1 Feature/Future Ideas 42

5.2 Conclusion 43

References **44**

1 Introduction

File systems are usually designed with three aspects in mind: speed, reliability, and security. Until recently, user-friendliness was often not a priority. In the beginning of and before the personal computer's era, this was not a big problem, because end users hardly existed. They neither had a lot of files to store, nor did they have a lot of space to store their files.

But in recent decades, availability of electronic information has greatly increased, while the amount of storage space available to the average computer user has also increased thousandfold. Such tasks as managing the photographs taken with one's own digital camera, or managing one's music collection are done using personal computers by more and more people.

From the user's perspective, this situation does not only have advantages. Hardware to store all the data that can be captured or collected on the internet is available, but a software interface that allows handling all the new possibilities is not.

This thesis intends to shed some light on why we are in this situation, while proposing a small but fundamental change to file systems to improve their usability. A prototype of an improved file system, which will be portrayed, was developed during the course of this thesis. In order to quantify the achieved improvement, suggestions for carrying out a usability study on a file system are made.

1.1 Scope and Content of this Thesis

Section 1.2 outlines the reasons for why this thesis was written. Other research and software products that address similar issues are discussed in section 2. In section 3, the goals and details of development of the NHFS software will be described. Considerations about performing a usability study to assess file systems and proposals for such a study can be found in section 4. The last portion of this thesis, section 5, contains an outlook on some topics that surfaced during the writing process, and a number of future ideas for NHFS.

1.2 Motivation

This section outlines the motivation for writing this thesis. It shows that current computer users are actually in the midst of an information crisis with the abundance of available information and relatively easy access to it, but the insufficient means by which to organize this information, e.g. when it comes to personal file management.

Information Crisis

During the last centuries, information availability has increased manifold. This can be attributed to two things: the amount of available information; and the ease of access. Channeling all the available streams of information and organizing those pieces of information needed to satisfy one's information need is not necessarily becoming easier. Shapiro et al. [1997] (ff. 55) describe an information crisis as a situation in where the available methods, means, or forms that are being used in an information activity (e.g. doing research) are insufficient to manage the continuous stream of information. As these means and methods also include tools to organize and archive pieces of information collected and created, a lack of adequate tools for these organizational and archival purposes will also result in an information crisis. One of those tools used by many is the file system itself; it is needed to organize documents and other files that one would like to keep for future reference. But the design of even the most modern file systems is based upon concepts developed in the 1960s and 1970s (see 1.4), a time when personal information management[1] was completely different from what it is today.

Amount of Available Information Barring the case of the demise of whole civilizations, information is rarely ever lost. Instead, more and more is produced all the time, and in the past few centuries, the amount of information generated has been growing. This is especially evident in the scientific field. According to UNESCO data, the number of scientists in the world has risen from about a thousand in 1800 to one million in 1950, with an approximate tenfold increase occurring every 50 years (Shapiro et al. [1997], ff. 56).

The number of scientific publications has been increasing at a similar rate. The number of books published on physics per year increased almost 45 times from 1900 to 1975. In 1980, around 50 times as many titles of scientific-technical journals appeared as in 1900 (Shapiro et al. [1997], ff. 56)[2].

Ease of Access to Information Since the appearance of the internet, World Wide Web, and other electronic means of searching for relevant data, access to information has never been easier. The widespread use of general World Wide Web search engines coupled with the emergence of resources that provide freely available information (e.g. Wikipedia, Project Gutenberg, DigBib.org) make very broad and detailed information easily accessible to everyone, requiring nothing more than an internet connection. In addition to these recent innovations, the traditional sources such as libraries, paper journals, etc. are still available and have been refined to facilitate access through the use of digital library catalogs.

[1] In this thesis, the term "personal information management" is used in a broader sense than usual. It refers more to the complete task of managing one's files and documents instead of the narrower but widely used definition of PIM meaning only the management of contacts and emails.

[2] In contrast to this increase, general population growth has been much smaller, with an increase from around 0.98 billion in 1800 to 2.52 billion in 1950 (UN [1999]).

There is also a general tendency to publish traditional offline media online. Complete works are published by projects like Project Gutenberg and DigBib.org; other services (Google Scholar[3], or Citeseer[4]), usually only make parts of offline media available (such as titles, authors, and abstracts of scientific publications). These services make access to information available at the mere push of a button, versus what used to take at least a walk to the library, if not a week waiting for the mail.

The Increasing Importance of Computers In Personal Information Management

Management of personal information, be it letters, images, music, or addresses is done on computers now more than ever before. With standard personal computers having a storage capacity in the 100 gigabyte range or more, it is not uncommon for users to have amass file collections of tens of thousands of files. Many activities formerly performed completely independently of computers (e.g. listening to, copying and sharing music, looking at pictures, reading newspapers and books), are now done via personal computers.

Conventional File Systems under Usability Aspects

From a usability standpoint, file systems on the major platforms have not changed very much since the early 1970s (see 1.4). Instead, they have and are being further developed with mainly technical aspects (speed, security, safety) in mind.

Since the only way of accessing and managing one's files is through the use of software tools that utilize the interface provided by file systems, the possibilities available through this type of access are strongly determined and limited by the characteristics of the file system.

Probably the best example for the shortcomings of today's file systems from a usability standpoint is the mixing of identifier, content, and location of files. In most monohierarchical file systems, all three mentioned concepts appear in the path to a file. "/home/bob/pictures/trip-to-paris-2005/eiffel-tower.jpg" is used simultaneously as an identifier of this file (file system operations like move, read, write on files take the path as an argument), as information describing the location of the file (the file could be stored on the partition mounted at /home/), and as a path through the taxonomy of the user's documents, describing the content of the file[5], with the file name containing even a short description of the content. This is a usability problem for several reasons. As an example, if the user decided to change this file's content, he would consequently need to change the file's name, and its path (if the position in his document taxonomy would need to be adapted to the file's new content). Thus, the identifier of the file would change, breaking

[3] `http://scholar.google.com/`

[4] `http://citeseer.csail.mit.edu/`

[5] Even more file system metadata is included here: the file type, which is specified by the file name extension on several platforms.

any references to this file.

O'Toole and Gifford [1992] describe the problem of mixing content and location considering USENET news articles as an example. They propose to use content names to access and create USENET articles. Articles containing "beatles" in the subject and "yesterday" in the message could be found (and even created) using standard file system tools[6]:

```
$ cd /sfs/news/subject:/beatles/text:/yesterday
$ ls
```

While finding items by their location its advantages as well (see Barreau [1995], Nardi et al. [1995]), having the added possibility of finding or browsing by content using normal file system utilities can prove helpful in many use cases.

1.3 What is Wrong with Monohierarchical File Systems

According to Dourish et al. [2000], there are at least four tasks that need to be performed regularly with documents: filing, managing, locating, and sharing documents.

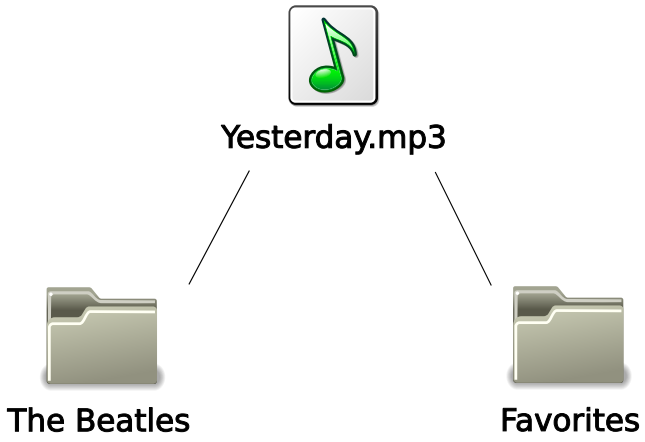

Figure 1: Question: Where do you file it?

Filing Documents A new document is received and needs to be stored in the file system. The question the user faces is illustrated in figure 1. Which folder does he file it in? Making this decision may require quite a bit of thought. Firstly, because the user may be unsure of which folder would best suit his file at that specific moment; and secondly, he may be unsure of where to file it, if he is to ever find it again for future use. This is a categorization task which is very context dependent, where the context is very likely to change over time.

[6] $ denotes the command line input.

Locating Documents When locating a document, the user can either try to remember where exactly he has placed the document he is looking for, or he can ask himself the question of where he would put this item now, if he had to file it, looking for it there — a process very similar to actually filing that same document. Due to the fact that most computer users are simply not able to remember the exact locations of all documents in their ever-growing document collections, in many cases, users have to revert to the latter variant when seeking a file, thereby performing a simulated filing (categorization) task very often.

Managing Documents Managing documents encompasses tasks like moving, deleting, and renaming, as well as some mainly system dependent tasks like changing file properties. Most of these tasks make it necessary to locate the documents and then perform an action on them. Moving a document requires locating and then filing the document into a different folder. Managing documents therefore poses as many problems to the user as the individual tasks of "locating" and "filing" do.

Furthermore, there is the aspect of dynamics: as the user's (non-file system related) tasks change, he wants to make changes in the file system. Users working with a "to do" or "important" directory to remind them of things that urgently need to be taken care of, or are relevant for a certain period of time, often need to put files that they might have already stored somewhere else on their file system in such directories again. With monohierarchical file systems, these file must be moved, and upon completing the task, the user must then return those files to the original location from which he first retrieved them, which he may not even remember anymore.

Sharing Documents On computer networks, which are becoming increasingly common even for home computer users, files and folders are often shared among users. The same holds for multi-user computer systems, where different people share any number of computers, a typical situation in academic and corporate settings. One person's method of organizing documents may not make sense to another. If Person A files a document into a shared collection, it is possible — if not likely — that Person B, who uses the same collection would like to add to this document a classification of his own.

All four of these tasks — filing, locating, managing, and sharing — are usually performed within the strict limits that monohierarchical file systems impose on the user. When adding new documents to the collection, the user is not permitted more than one location in which to file them, despite his being able to foresee different contexts in which he would like to be able to locate these documents in the future. The task of locating a file could be much easier if it were possible to file documents into more than one place from the beginning. When the need to locate the documents arises, the probability that the user's context when locating them is

matched by one of the anticipated contexts used when filing them is higher than if the documents can only be filed into a single place.

Why Symbolic Links, Hard Links, and "Shortcuts" are not a Solution

Symbolic Links Symbolic links, which are present on most Unix-like operating systems, are special files that contain a reference to another file or directory. They can be used to circumvent the shortcomings of mono hierarchical (tree) file system structures. They are a bit cumbersome to use for personal file management, because the link is not updated whenever the target pointed to by a symbolic link is moved, renamed, or deleted, the link is not updated; instead, it points to something that no longer exists. Also, some applications do not handle symbolic links as if they were the objects they point to, but rather resolve the referenced path, using that instead of the link's path. Another question is what happens if the name of a symbolic link is changed — usually the link is renamed, not its target.

Shortcuts and Aliases Microsoft Windows has a remotely similar concept to symbolic links called "shortcuts". Shortcuts are small text files with the extension .lnk that contain, among other things, the path referenced by them. When opening a shortcut, the target path is resolved at application level rather than at the operating system level, which means that any application that does not support shortcuts will just open a file pointer to the shortcut file as it is (a simple text file) using the Windows file system interface. Like symbolic links, shortcuts break once their targets disappear.

Aliases on Mac OS X resemble Windows shortcuts, but they have the advantage that they should continue to work even if the referenced object is moved. This fault tolerance is achieved by employing several search strategies in case of a broken alias, so it is likely but not guaranteed that it can be resolved.

Hard Links Hard links are a feature of all major desktop operating systems. On Windows, hard links are only possible on partitions formatted with the NTFS file system, and special utilities are necessary to create them. On Unix-type systems, any file is a hard link, which means that it is a name that points to the file's data. A file can have more than one hard link, and each hard link can have a different name. Unfortunately, hard linking directories is not supported under current operating systems; this actually was a feature of the first Unix file system created largely by Dennis Ritchie (Ritchie [1980]), but it had mainly existed to make up for one of the file system's major disadvantages, namely the lack of path names, which made it impossible to "enter" directories outside of the user's base directory. Hard links for directories were abandonned in the later course of development of the different types of Unixes.

Another reason why all these band-aid solutions (symbolic links, shortcuts, and aliases) are not commonly employed in personal file management can be found when looking at the user interface. Since working with any of them is conceptually different from working with "real" files, the solutions that programmers and interface designers have devised to handle them are also different from those used for file management. This also means that even between applications, creating a symbolic link on Linux can require totally different actions. Of the two most common file managers on Linux, Nautilus (Gnome project) and Konqueror (KDE project), the former requires a hidden keyboard shortcut to create a link when dragging and dropping a file, while the latter displays a context menu that lets the user choose "symbolic link" (among other options) whenever a file is dropped. Had the "link" functionality been more thoroughly integrated into the file system from the beginning, applications would probably show more conformity, causing less user confusion.

Differences between file management applications may not be as important on Windows and Mac OS as both have their designated file managers which are used by the majority of users on both operating systems.

A Precisely Classified Document is Harder to Locate

For any information management system, intuition implies that an accurate classification of documents leads to them being easier to locate. However, in a hierarchical system where locating means browsing levels of classifications, as is the case in a standard file system, classifying a document as precisely as possible does not necessarily make it easier or faster to find. For each label of categorization the user attaches to a document, a new level in the directory hierarchy needs to be created. A user preferring to accurately classify his documents will end up with deeply nested directory structures as his document collection grows. Especially if the user does not regularly visit parts of this hierarchy, it can quickly become more confusing than helpful.

As an example, a path like:

```
/home/rfreund/documents/work/important/taxes/tax return/2006/9872938.pdf
```

certainly represents an exact classification of the document. And of course, in reality the path necessary to navigate to it will be made much shorter because of helper links to frequently needed directories (like /home/rfreund/, /home/rfreund/documents/) that exist in all common desktop environments. Still, considering the fact that also /home/rfreund/documents/ is part of the classification hierarchy of the document (namely classifying it as a document of the user rfreund), it becomes clear that such a precise classification in a hierarchical system requires paths to objects that are so long that even the simple task of navigating to an object by repeatedly clicking with the mouse can become tedious.

1.4 History of File Systems

In order to explain why from a usability standpoint, file systems on the major platforms have not changed very much since the early 1970s, this section describes the historical roots of today's file systems.

Unix

Dennis M. Ritchie, Ken Thompson and others started working on the intended successor of the Multics operating system in 1969, which ran on the PDP-7 computer at first (Ritchie [1980] and Ritchie and Thompson [1978]). One of the stated goals for the operating system was to have a "hierarchical file system", a feature that was innovative and did not exist in many much larger operating systems (Ritchie and Thompson [1978]). The file system they created was very similar to most modern file systems in many ways. It had files and directories; directories were special files containing a list (of the inode numbers) of their children. The implemented file system operations were read, write, open, creat [sic], and close. The biggest difference between the PDP-7 file system and most file systems common today was that the former allowed hard links to directories to be created. Therefore, the first Unix file system was not a monohierarchical structure like later file systems but rather a general directed graph.

Since the development of the first Unix started more as a personal project by Ken Thompson (Ritchie [1980]) and since his employer (Bell Laboratories) did not see much of an economic opportunity in it, the first versions of Unix were licenced relatively inexpensively and with full source code access. This initial openness resulted in widespread adoption of the system, especially for educational purposes in the academic field. By the time the first commercial version of Unix appeared in the late 1970s, it was already enjoying great success. So it is likely that most people working in computer science or software development at the time had come in touch with Unix, and were possibly influenced by it.

Microsoft Windows

Due to the fact that Unix systems were one of the most widely used operating system types in the 1970s, and all Unix implementations were inspired by the ideas of the group around Ritchie and Thompson, by the late 1970s, many file systems were in use that were largely similar to the PDP-7 file system. It is likely that others, for instance developers at Apple and Microsoft, took many ideas from the Unix file systems when they developed their own operating systems in the late 1970s and early 1980s. Since Microsoft operating systems still dominate the end user computer market today, and their file system implementation had not changed much until the late 1990s — Windows 98 still carried some characteristics of early MSDOS file system wise, and even Windows XP, which was the current version of Microsoft's operating systems until

December 2006, could still run on Windows 98's file system (FAT32) — the Microsoft Windows world still functions in remembrance of early Unix file systems. For their latest version of Windows, Vista, Microsoft tried to innovate and maybe revolutionize the way the file system works with WinFS, but failed (see 2.5.2).

Apple/Macintosh

Similar observations can be made when examining the Apple/Macintosh world. After many years, Apple decided to base their operating system development on components heavily borrowing from Unix (including XNU, the kernel, and various parts from BSD Unix) in the late 1990s. Apple's operating system today conforms to the POSIX standard[7], which was created by the IEEE (Institute of Electrical and Electronics Engineers, a standards organization) in the 1980s in order to standardize the many different Unix implementations that existed at the time. So the Macintosh operating system uses a file system deeply rooted in Unix.

GNU/Linux

Another large share of file system implementations in use by end users today is very much akin with Unix file systems as well: the file systems in use under the GNU/Linux operating system. GNU/Linux was created as an open source Unix operating system beginning in the early 1980s (Linux kernel: 1990s). The emergence of GNU/Linux resulted from the growing commercialization and high license costs of Unix systems. Much of it was developed from the beginning with compatibility in mind to the commercial Unixes of the time.

As shown above, most file systems employed by end users today are inspired by or even descendents of the first Unix file systems. But at the time of the creation of the first versions of Unix, graphical user interfaces were not in widespread use at all (they did not become common until the 1980s, see Nielsen [1993], p. 57); instead, users controlled their computers using command line interfaces. Since textually oriented interfaces like the command line impose totally different aspects on how the user machine interaction takes place than graphical interfaces, the developers of the original Unix most likely came up with a file system that was optimized for being accessed by a command line interface, and not a graphical one. Had computers had mice, and drag and drop, and had advanced visual interfaces been in widespread use or even been available at the time, they might have come up with a radically different file system design. For instance, a hypothetical tag based file system (see Bloehdorn et al. [2006]) for instance seems much more appealing when imagining the user being able to visually browse a possibly multidimensional representation of tags and comfortably edit sets of tags in order to search for and select files and directories to work with them. This way of working

[7] `http://www.apple.com/macosx/leopard/technology/unix.html`

with elements on the screen had not been conceived at the time of early Unix development, and so the system handling was most likely optimized for a textual interface.

The number of files available to the original Unix creators probably also influenced the development. In the first Unix file system, creating directories was only possible at the time of file system creation. Furthermore, the system did not allow the user to switch the working directory within the fixed directory structure. Instead, the user would create a link to whatever he wished to work on at the moment in his home directory, and would thereupon work on the link. This should make clear that the developers did not need to manage a lot of files at the time, let alone directories, and might not have anticipated this situation to change so dramatically in the future.

2 Related Research and Products

Many attempts have been made to simplify management of personal information and documents in both the scientific and commercial field. This section outlines some examples of different categories of these simplifications. It does not focus on solutions based on changes of the file system, since these are only few and far between.

2.1 Approaches Using Keyword Based Search

Keyword based search[8] can be a powerful tool. For huge sources of relatively unstructured data like the web pages on the internet it may be the most useful mechanism. But in the area of personal information management, research suggests that a directory structure helps users to visualize their information space. Here, location based search is often favored (Barreau [1995], Nardi et al. [1995]) over keyword based search.

Apple Spotlight, Beagle, Google Desktop

Tightly integrated into Apple's operating system Mac OS X since version 10.4, Spotlight[9] provides instant indexing of created and modified files. All relevant file system operations are signalled to Spotlight by the operating system's kernel (figure 2). Spotlight then calls file type specific plugins that are able to extract file content and metadata (e.g. creation and modification dates, file type and size) and adds it to its search index. The search interface also finds substring matches and allows boolean expressions using the operands and, or, and negation.

Besides normal file content and metadata Spotlight also indexes keywords that the user has assigned to specific files.

This approach relies on two assumptions:

- Plugins for extracting file type specific content and metadata are available for many file types. With thousands of proprietary file formats available, the user is often at the vendor's mercy in this respect.
- Files generally carry sufficient relevant and indexable content and/or metadata. However, oftentimes there may not be sufficient such information available, as in the case of files not containing any textual information like images or audio files, which often lack metadata as well.

There are other similar ideas and products, e.g. Beagle for Linux[10] and Google Desktop [11]. Since they are fairly similar, they also carry the same deficiencies as Spotlight.

[8] keyword based search: finding matching documents by the use of keywords the user enters
[9] `http://developer.apple.com/macosx/spotlight.html`
[10] `http://beagleproject.org/`
[11] `http://desktop.google.com/`

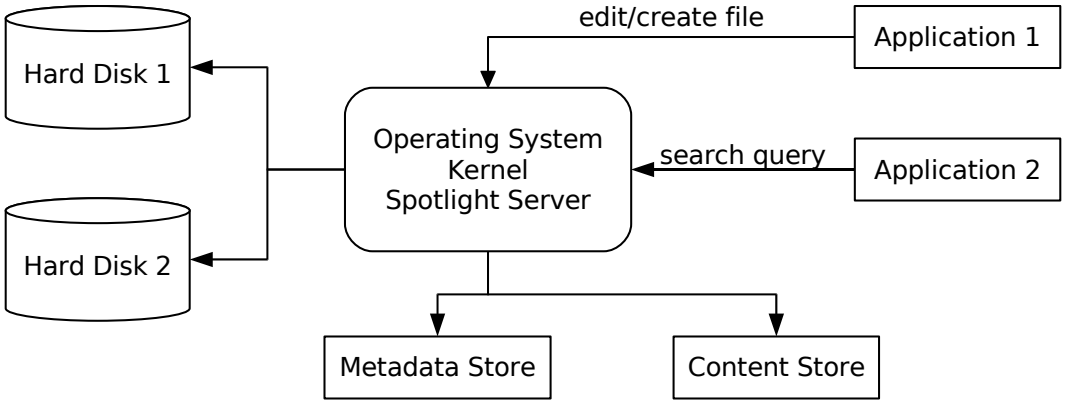

Figure 2: Spotlight overview: User space applications create and edit files and send search queries to the Spotlight server. The server indexes files on all connected hard drives and stores file content and metadata in different databases.

2.2 Solutions for Specific Problems

In many areas of personal computing, solutions for specific information management tasks have evolved. Unfortunately, these solutions help the user deal with only one specific area of the big problem of personal information management.

Media Libraries

There are many different software products that empower the user to access and manage large music collections. Some examples are Winamp on Windows, iTunes on Mac, or amaroK on Linux. These programs usually work very similarly. They allow the user to add files and folders to an internal media database. After scanning the user's complete media collection, they provide an extended interface that allows browsing and searching for file metadata. Adding a file containing the meta information "Artist: Beatles", "Album: Help", "Track: Yesterday", "Genre: 60s" will result in these pieces of information becoming available in a browsable representation of the whole collection.

In contrast to most normal file systems, where we are only able to put the file in one folder (not more than one), the file will show up at least in the views "Beatles" and "60s". This makes

- the file easier to find, since it appears in contexts where we would naturally look for it
- common tasks with sets of files much easier, e.g. "find all music from the 60s"

Email Clients

Email clients typically try to offer a very usable way to access one's email. They do so by allowing the user to sort and filter the incoming stream of information efficiently. After setting up filters manually or having

them setup automatically, new mail can be filed into different folders. Most email clients offer a permanently accessible search input field that is available to filter all views for emails with certain content or metadata only. Filtering for metadata like “rfreund@uos.de” or content like “bachelor thesis” is usually very simple.

Even though most email clients use something called “folder”, the internal representation does not have to resemble real file system folders at all. In fact, in the client included in the web browser Opera, all folders are merely views on the whole set of messages, received and sent. In this concept, emails can naturally appear in different views, or contexts. Obviously, if there are two folders called “Sent to Bob” and “Sent to Alice”, a message sent to both, Bob and Alice, is expected to appear in both folders, not in only one of them.

2.3 DocPlayer

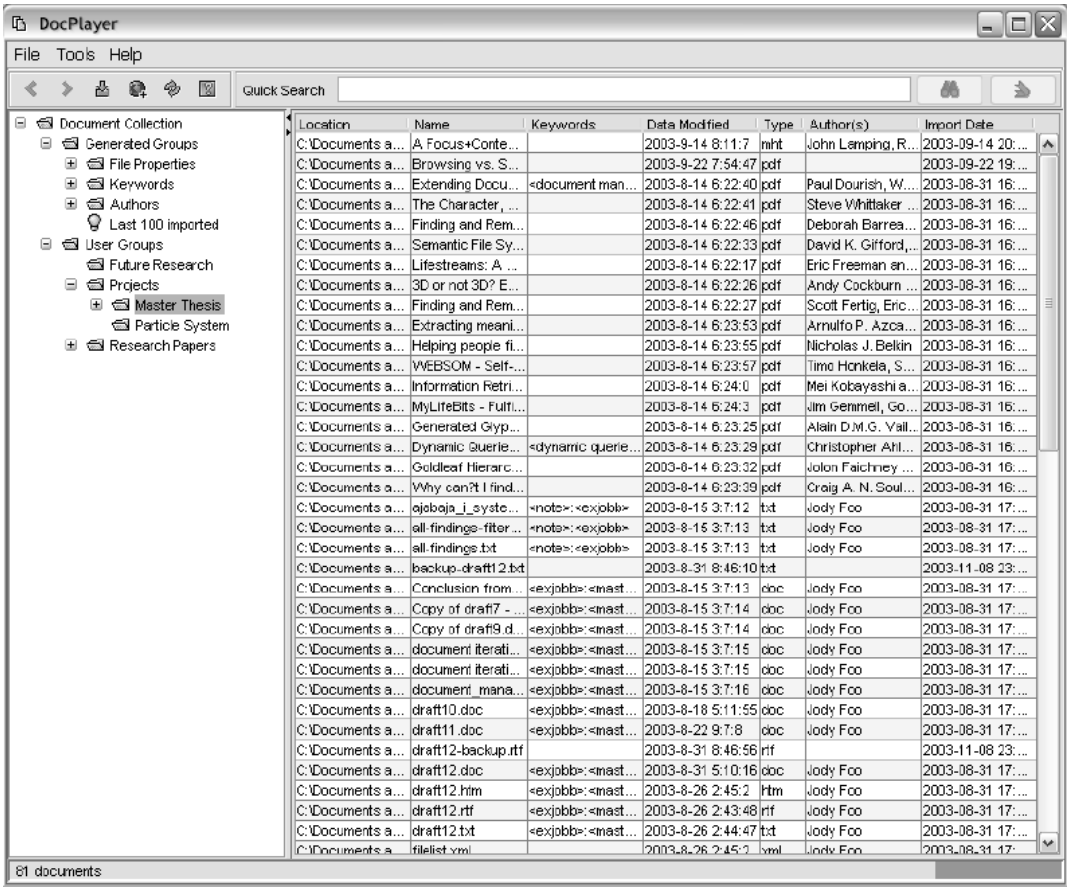

Figure 3: DocPlayer’s main screen which allows accessing the document collection using “quick search”, and the smart, automatic, and manual “groups” from the pane on the left.

DocPlayer (Foo [2003]) is an approach that applies the functionalities of media libraries to document management.

When the DocPlayer application is first run, it builds a representation of the user’s files in a MySQL

database. The database contains parts of the files' metadata (name, type, author[12], modification and creation times). After adding files, they can be accessed through the interface shown in the screenshot in figure 3. Newly imported files can be found using a search query over available file metadata, or by browsing the automatically generated "Groups" of files having the same type or author, or the "new files Group". "Groups" are a set-based implementation of the playlist model used by media libraries.

Groups can be created manually, and files can be linked statically with them. It is possible to create so called "Smart Groups" which can be defined using a search query over the collection. The "Smart Groups" are updated as files are added to and removed from the collection. Removing a group does not remove the files linked with it; the files stay in the collection.

In addition to existing metadata, DocPlayer can also store keywords about the files. These keywords are managed by the user. Search queries and "Smart Groups" can of course encompass file keywords.

The media library model offers some very interesting ideas for document management. In DocPlayer, these ideas were implemented as a Java application that allows users to have a totally different view on their collection of files. Accessing this view is completely limited to the DocPlayer application, though. This can be quite restrictive for users who do not want to use a particular program to access their files. Users prefering to use the "open file" functionality coming with most programs, or users prefering the command line to graphical interfaces will probably dislike this approach. These users need a solution that solves more of the underlying problems of file systems.

2.4 Lifestreams

The Lifestreams software architecture (Freeman [1997]) suggests to ease the task of document management as performed by the average user by using a timeline metaphor instead of the desktop metaphor. Along this timeline, documents can be created now or in the future. All files may also be cloned within the Lifestream or transfered to another Lifestream (or substream).

The main user interface of the Lifestreams system is shown in figure 4. Working with Lifestreams is comparable to working with any other file browser. After the user has found an item, the system will call external applications to display and edit it. Only the basic operations cloning, transferring, and creating are done within the Lifestreams system.

In addition to the graphical client, a command line interface to Lifestreams was developed as well, which caters to the users who prefer a text based interface to their systems. For many users this might still not be satisfactory, e.g. people using special types of graphical file browsers or those who prefer to use a certain type of command line interpreter that has certain features especially important for file management tasks,

[12]The author information is extracted by a regular expression. This only works for certain file types.

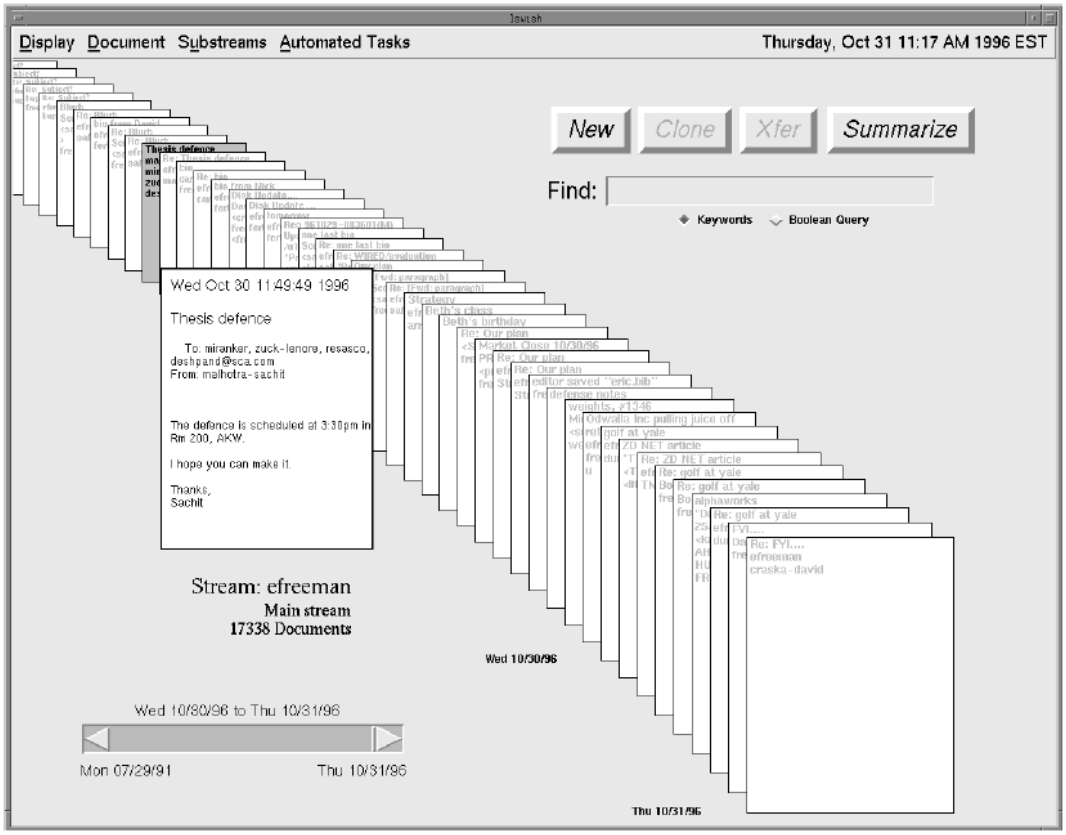

Figure 4: The Lifestreams main user interface. The chronologially ordered documents (the latest one in the front) are displayed as a stream. A slider (lower left) allows to quickly adjust the timeframe of items to display. One document has been selected and is displayed in detail. The input box (upper right) allows searching by keyword and boolean queries. Basic operations (new, clone, transfer) can be accessed directly via the buttons above the search box.

like auto-completion, or command line expansion.

Since managing documents in Lifestreams is primarily guided by the age of individual documents, other properties loose significance. If a user were to look for an item not by means of the point in time (when) he filed it into his collection but the location (where) he put it, he would not be served very well, because in Lifestreams, there is no location.

The desktop metaphor may not be the perfect solution for all file management tasks, but after all, it is one that most user are familiar with. Therefore it is questionable if radical approaches like the one taken by the Lifestreams project can have any success among a larger user base.

2.5 File System Based Solutions

There are a number of approaches that address the described usability problems on a more basic level — the file system itself. There are several reasons why this could be a better solution for the problems with current

file systems:

- The perceived problems are of a very fundamental nature, so fundamental changes are needed to resolve them.
- Fundamental changes allow to influence file system properties and functionality most.
- Compatibility considerations: changing the file system itself while preserving conformity with existing programming interfaces allows the users to continue using the graphical or command line interfaces they like best and are used to.

2.5.1 Semantic File Systems

There have been and are numerous research projects on semantic file systems. Only the best known one will be presented here as an example.

A semantic file system as developed by (O'Toole and Gifford [1992], Gifford et al. [1991]) provides access to file contents and metadata by extracting these attributes using transducers through the file system, i.e. object properties are exposed as files and directories. The user programmable transducers, which act like plugins, are file type specific.

In this specific system, file attributes are indexed on changes and updates. Users can query this index by using virtual directories. Here is an example showing how the virtual directory

```
/sfs/owner:
```

can be used to query which files belong to which user[13]:

```
$ ls /sfs/owner:
alice/ bob/
$ ls /sfs/owner:alice/
text.txt@ img.jpg@
```

The virtual directories are invisible to directory lookups, so their names need to be explicitly typed in by the user. This was done in order not to confuse standard file walking tools like "find". The files contained in a virtual directory are symbolic links to the actual files.

The biggest drawback of this particular semantic file system is the need for file type specific transducers that are necessary to extract meta information and content from the file. Typical information that could be extracted for different file types includes the author, and names of exported and imported functions for a file

[13] $ denotes the command line input.

containing programm source code; the sender, recipient, and subject for an email file. These examples already show that for each of these pieces of information, a completely different transducer is necessary. While it is feasible to create transducers for many different file types, this is all but a trivial task.

Another weakness is that virtual directories are not exposed to directory listings, so the user has to type in what kind of information he may be looking for. This requires the user to know the exact names of the attributes he is interested in — and also, which attributes he is interested in in the first place. Naturally, this works best with the command line, so for the majority of users, who exclusively use graphical interfaces, a system like this is not very well suited.

Although not directly implemented in this specific project, it would be feasible to allow user created, arbitrary field-value pairs to be attached to files. This would enable the user to create views on the file collection. The property "Beatles" could be attached to all files that should appear in the view "Beatles". A second property, "music", could be used to identify all music; all files that carry both properties can be assumed to be music by The Beatles.

2.5.2 WinFS

WinFS, a next generation storage system for Windows, developed by Microsoft, was supposed to allow database like access to the file system. Planned to debute with Windows Vista in 2006/2007, it was revoked from the product some time before Vista's launch, and the operating system was released with NTFS, which has been Microsoft's standard file system for many years now. Underlining how complex and difficult the undertaking seems to be is the fact that even the world's largest software company failed in trying to innovate this important core piece of the operating system, a part that has conceptually aged so much.

WinFS was not to be a file system itself, but a data storage service. It used NTFS as a base file system to store all of its data (Grimes [2004]). The basic data types that existed in WinFS were items, scalar, nested, and relationships:

Items Instead of files, WinFS was designed to store more general pieces of data, called items. An item could be many different things — a file, a document, a contact in the address book, an entry in the Windows registry, an image, or an entry in the Windows event log. Each type of item had to be described by a schema written in XML. New item types could be created easily. They constituted a hierarchy and inherited from one another (Wildermuth [2004]).

Scalar Types The scalar type was an atomic piece of data contained in a particular item. Scalars itself could be of many different types, mainly the ones used in programming languages, e.g. integer, floating point

number, character, or text.

Nested Types Nested types allowed to store structured sets of information. These sets could contain other nested types or scalar types. A nested type could be stored within another nested type or an item.

Relationships A relationship defined a link between two items. It was directional, and therefore established a source and a target. By creating a directory within WinFS, the user would create a relationship between the parent directory and the newly created one. For other item types special applications could be used to create the relationships between individual items. Relationships could be of three different types:

Holding Relationship The holding relationship determined the lifetime of the target. When removing the last holding relationship of a target, the target itself would be removed from the WinFS. The holding relationship resembles the hard link concept in most file systems.

Reference Relationship The reference relationship served as a mere pointer from source to target. Removing it did not have any effects on the target.

Embedded Relationship If two items had an embedded relationship, the target item was part of the source item. An item of type person (source) could have an embedded relationship with an item of type address.

One of the main ideas of WinFS was that data should be shared. For instance, any application should be able to have access to the contacts in the user's address book, so the user could use these contacts when writing a letter to a contact from his text processing software just the same as when writing an email using the email program.

Accessing the data items stored in WinFS was possible by several means. One was the WinFS API, which allowed applications to query and manipulate items. An application could easily iterate through all items of type contact; it could then filter all contacts by properties, e.g. "Firstname=='Michael'" (Grimes [2004]). Another means of accessing data items was a query language called WinFS OPath. OPath was relatively similar to standard SQL with some additions and extensions partially borrowed from C#, a programming language developed by Microsoft.

As WinFS was designed mainly for management of the user's data, it was not supposed to be in control of the whole file system. Instead, it would only be used within the "Document and Settings" directory where all the user's files and settings are stored. The rest of the file system, the data in program and system directories

in particular, would still be stored using NTFS, Microsoft's standard file system. The same was true for the relational database managed by the WinFS "relational engine", which was merely a standard SQL database stored on an NTFS partition.

The design of WinFS also ensured some backward compatibility for legacy applications. Pure data items (e.g. contacts in an address book) were only accessible by applications taking advantage of the WinFS API, but files themselves were still stored on a normal NTFS file system, while WinFS only stored information about them in a relational database, so accessing files would have been possible even from legacy applications.

The largest problem of WinFS was most likely that its usefulness depended very much on application support. Its success would have required many independent software makers to adapt their products to utilize WinFS to store and access their data in order to share it with other applications.

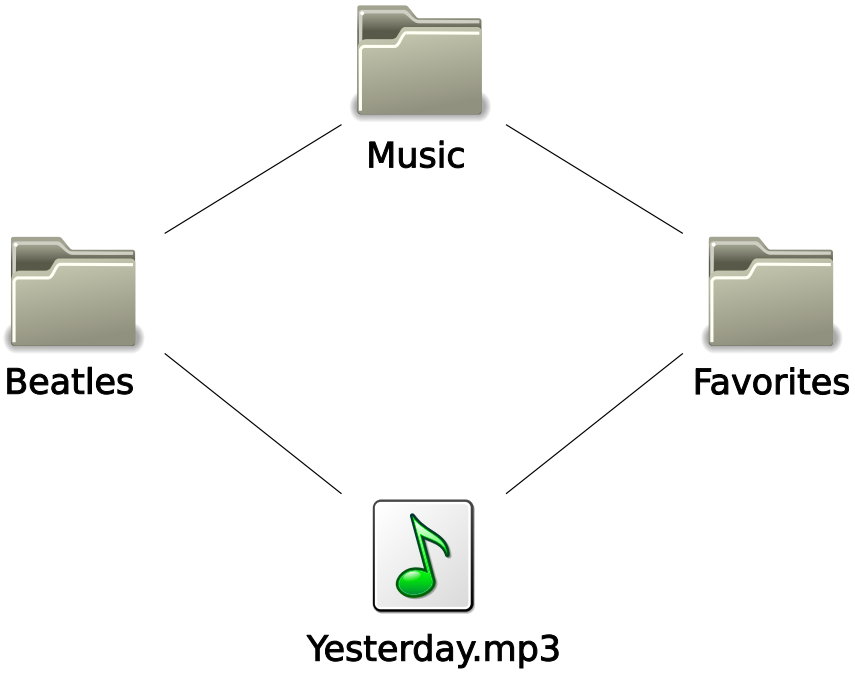

Figure 5: Freedom of filing

3 NHFS — a ”Nonhierarchical” File System

This section describes what NHFS is and what its goals are. Furthermore, some of its implementational details are outlined, and its usage is explained. The section concludes with a short technical evaluation and a list of functions or properties that are strongly needed but not implemented in the prototype yet. It should be noted that the name ”NHFS”, which stands for ”Nonhierarchical File System” — is a bit misleading, because NHFS does also contain (poly-) hierarchical elements. It was still decided to name it ”nonhierarchical” because the nonhierarchical character of its directory structure (see 3.2.2 on page 23) is expected to have a very strong incfluence on the way the file system is used.

3.1 Goals

The goal of this prototypical implementation was to create a file system which allows the user to leave behind the strict monohierarchical structure forced onto him by standard file systems. In NHFS, any file can be filed into as many directories as one wishes. Likewise, any directory can be filed into any number of other directories. As an example, the user is able to file the file “Yesterday.mp3” into the folders “Beatles” and “Favorites” (figure 5). Similarly, the directory “Beatles” can be filed into the directories “music of the 60s” and “Favorites” at the same time.

A secondary goal while developing NHFS was to make it usable in the practical sense, which means that it should:

- be backward compatible with existing file systems and programms accessing the file system as much as possible
- be easy to install, configure, and run
- not have many dependencies on other software (libraries etc.)

In order to maintain comparability with standard file systems for usability studies, it was intended that the changes compared to standard systems should be as minimal as possible.

3.2 Structure and Semantics of File System Elements

This section describes the differences in structure and semantics of conventional file systems and those with a digraphical[14] character. It also shows how these differences could be reflected in the way the file systems are used. The term "structure" refers to the way file system elements can be arranged by the user.

3.2.1 Monohierarchical (Conventional) File Systems

Structure A file system consists of the following elements[15]: Directories, files, connections between directories and directories, and connections between directories and files. In conventional file systems, these elements form a strictly monohierarchical structure; directories and files are nodes; connections are edges. Any element that is not the root node has exactly one parent (see figure 6). An element that is a directory can have any number of children.

Semantics of File System Elements The semantics of file system elements largely depend on the way these elements are used, i.e., they depend on the user.

On first sight, the directory structure that a user creates from the file system elements in order to manage his documents resembles a simple taxonomy of the user's documents. The classes of the hierarchy, which are represented by directories, are of very different nature. On the one hand, there may be very objectively accessible classes like "text documents" or "music". On the other hand, there can also be concepts like "Berlin", "60s", or "important", the meaning of which often only becomes accessible from the user's individual point of view and within a specific context. These concepts have to be interpreted totally differently in their semantics. The following examples illustrate this:

[14] digraphical: like a directed graph

[15] Additional elements that are file system type dependent, like hard links, soft links, etc. are negligible in this context.

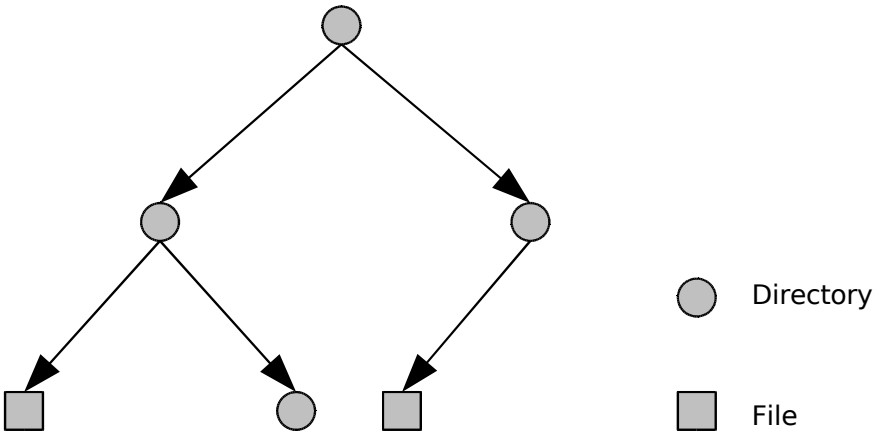

Figure 6: Structure of a monohierarchical file system

- ”Berlin”, for instance, could stand for all those documents that have a certain relation to the city of Berlin. The type of the relation can again be of very different kinds; as an example, the content of this directory could consist of pictures taken in Berlin (relation ”taken-in”); it could also consist of text documents about Berlin (relation ”content-about”). The exact type of the relation to the concept ”Berlin” often only results from the position of ”Berlin” in the directory hierarchy.

- ”important” could possibly be a directory containing objects which are generally important for the user, and which he does not want to delete in any case. Another possible way of understanding such a directory would be to assume that ”important” contains those objects which are of great importance for the user *at the moment*. Here, a strong dynamic dimension of file management manifests: whether an object is important for the user or not can change within a short period of time. This dynamism can be seen even clearer in directories like ”urgently to do”.

- ”Beatles — Help (Album)” could be a directory acting as a simple data container for the songs of this album. It could also stand for a more concept-like element, if this directory contained sub directories, and the contents of which were again related to the album of The Beatles, for instance associated film footage or images.

An item that is either a file or a directory is a pure data container when it exists as the root of a subtree (the leaves of which consist only of files), not used to make connections to other concepts or classes. In the case of a file this is also caused by the impossibility to use a file as a concept to classify other objects in the system; i.e., no connections *from* a file are possible.

3.2.2 NHFS

Structure By allowing file system nodes (directories and files) to have several parents, the resulting structure is a directed graph of directory nodes with polyhierarchically connected file nodes (see figure 7). The directory nodes can form cycles.

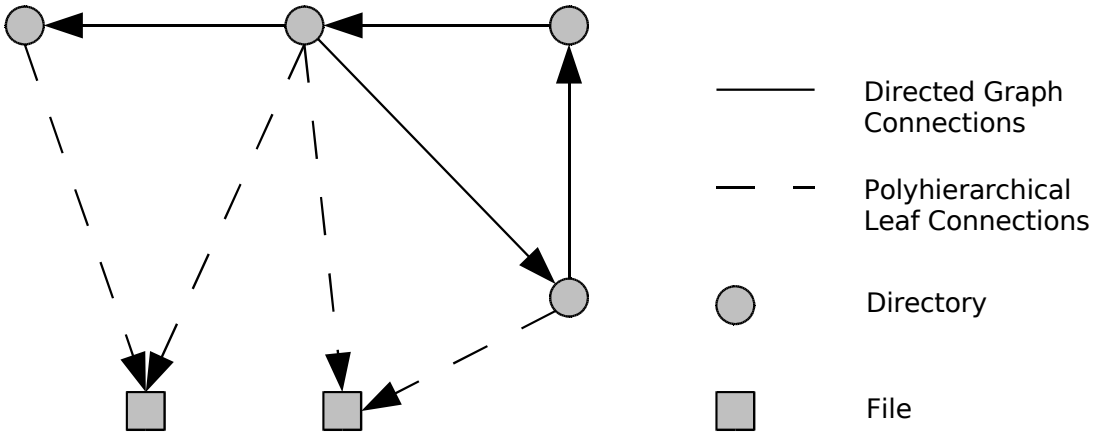

Figure 7: Structure of a digraphical file system with polyhierarchically attached leaves

Semantics of File System Elements The exclusionary categorisation, which the user is used to from monohierarchical systems, is not required anymore, so when filing an item, the user does not necessarily have to find or choose the absolutely best suitable concept (or category) to connect the item to. Instead, items can now also be connected to concepts which they may have a weaker association with than other concepts. It can be expected that this will result in file system structures that would not develop under conventional file systems. Maybe the user would now create directories that correspond more to a keyword, or a tag, which could be associated with the item. For instance, the user could create a directory "for mom", which he would connect items to, that he actually keeps somewhere else in his directory structure, but which he considers interesting for his mother, and wants to burn to a CD eventually.

3.3 Extending the File System's User Interface

Extending the file system's functionality and exposing this functionality to the user, or changing the meaning of existing file system operations is not at all easily feasible. This is caused by the fact that the file system is entangled with the virtual file system layer (the kernel), the GNU library of C functions, and user space processes. All three of them make their specific assumptions about what the file system does, and all have certain expectations of the outcome of file system operations. Because any operation takes the pathway user space application $\Longrightarrow$ glibc $\Longrightarrow$ VFS $\Longrightarrow$ file system (as described in 3.4), adding a file system operation

requires adding it to all four layers. It is not sufficient to simply extend only one of them.

Changing an existing functionality poses many problems as well. One of these problems is caching. The kernel and many user space applications keep caches of directory contents and file metadata, among other things. When calling the rename() function on a file, and this call succeeds, all caching mechanisms assume that the source file will have disappeared. If we changed rename() to actually perform something else, leaving the source file intact, the cache may not be coherent for a short period of time after the operation.

Because NHFS allows the user to "connect" a file or directory with a directory, several mechanisms were explored that could possibly be used to expose the new functionality of NHFS to the user. The following candidates from conventional file systems where considered to be overloaded or extended with the new function: copying, moving, creating symbolic link, creating hard link.

Copy If NHFS were able to determine that a file is copied to a directory, and if both target and source are on the same NHFS file system, it could theoretically decide to simply create a connection from the target directory to the source file. The process of copying a file is actually performed by the application (e.g. the file manager) used by the user. When commanded to copy a file, e.g.:

```
cp beatles/yesterday.mp3 john_lenon/
```

the application opens the source file, opens a new file at the target location, and then (repeatedly, if necessary) reads data from the source file and writes it to the target file. From the kernel or file system perspective this cannot be distinguished from two independent read and write processes[16], so NHFS is not able to determine that a copying process is taking place. This indistinguishability makes the standard "copy" functionality unsuitable for anything else.

Move Moving (or renaming — both is performed by the same function) a file is another standard way for the user to interact with the file system. As with copying, when moving a file within an NHFS, e.g.:

```
int rename(const char *from, const char *to)
```

NHFS could interpret this as a command to perform some other, special, action. It could be argued that in NHFS, the user does not need to move a file; he could simply add another connection. But since there is no connection weight or other means that could be used to clean the system of old or unused connections, the move function is still necessary. Also, as moving and renaming refers to the same function, and renaming is definitely a necessity in a file system, it is impossible to change its meaning.

[16]Not in the sense of operating system processes which are created by the kernel, e.g. whenever a program is run

Create Symbolic Link Creating a symbolic (soft) link comes very close to the functionality that was necessary to be added to the file system's user interface. A symbolic link is a special file that contains a pointer (path) to the file that it points to. Symbolic links are widely used as pointers to files that are possibly located on other file systems (partitions) and to link to directories, both situations which cannot be handled by hard links. Thus, symlink() is a necessity and cannot be substitute with other functionality.

Create Hard Link Just as with soft links, the creation of a hard link is very similar to what NHFS should provide the user with in addition to the standard file system features. Out of the four candidates considered for overloading or extending to expose the new function to the user, hard linking is probably the one least used generally. Changing it would therefore probably be least distracting to most users as well. Unfortunately, the Linux operating system kernel in its current versions does not allow hard linking directories. Since the goal for NHFS was to allow any directory to be connected with several other directories, the link function was out of the question.

As none of these four operations was suitable, a completely different solution was chosen — virtual directories. These allow to expose new functions to the user that are triggered when they are involved in standard file system operations. This has several advantages, one being that the new functionality is accessible to any standard interface the user may work with to manage his file system, graphical or command line. Also, adding other extensions to the file system is very easy this way. One could relatively simply add more virtual directories to NHFS to make additional functions available to the user. The trash can (see 3.5) is another feature that is accessed by using a special directory.

To allow the user to connect a file or directory with another directory, every directory in an NHFS contains a virtual, special directory called "link here". Any file or directory that is moved to a "link here" directory will be connected with the parent of "link here" (see also 3.5).

3.4 Implementation

NHFS was implemented in the programming language C as a user space file system for Linux, using the FUSE kernel module and library for the file system functionality running as a user process. Information about the file system structure created by the user is stored in an SQLite database. The data stored in NHFS is actually held in a directory on a conventional file system, which can be any partition where the user has write permissions. The user can choose where this data store should be located.

FUSE (File System in User Space)

NHFS was implemented using FUSE (File System in User Space, see figure 8). FUSE allows the implementation of file systems running in user space, i.e. they run as unprivileged user processes. It consists of a driver which implements the operating system's API for file systems, and a library, which acts as an interface between the kernel driver and the user space process representing the file system. FUSE has been part of the official Linux kernel since version 2.6.14. At the time of this writing, FUSE ports are also available for the FreeBSD and Mac OS X operating systems.

To understand how FUSE works, one needs to know a bit about the path any file system access takes on Linux, which is as follows: A user space program such as "ls" calls the corresponding function from the glibc (GNU C Library), for example "stat()" to gather information on a specific file. "stat()" then queries the kernel's VFS (Virtual File System), which acts as an abstraction layer between user space and a more concrete file system implementation. The VFS resolves the file system driver responsible for the file in question and passes the command on to that specific part of the kernel, e.g. the corresponding ext3 driver if the file is on an ext3 file system. If the file is on a file system mounted using FUSE, the VFS will send the "stat()" call to the FUSE driver. FUSE then hands the "stat()" call to the user space process registered to handle any access to this file.

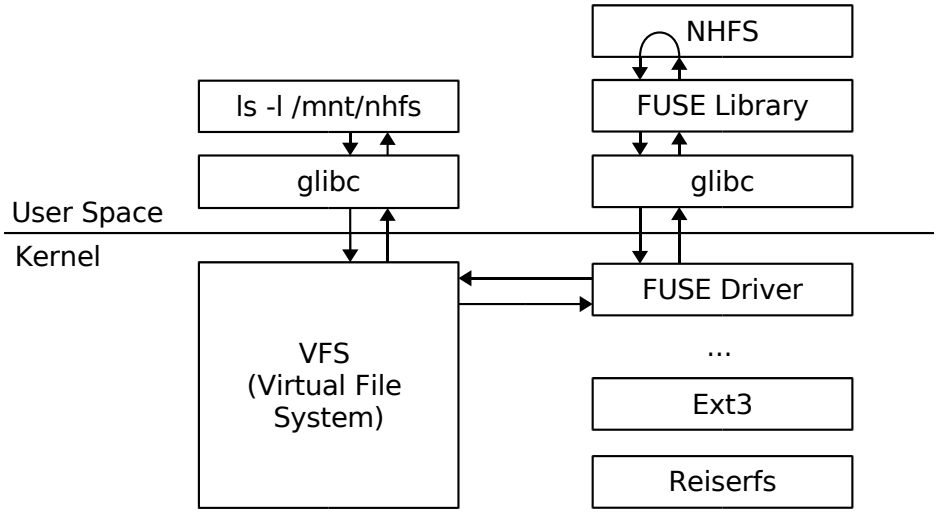

Figure 8: FUSE file system access pathway

System Architecture Overview

The system architecture of NHFS is composed of the following elements (see figure 9): The FUSE connector, the database connector, the structure store, the data store connector, the data store, and a number of

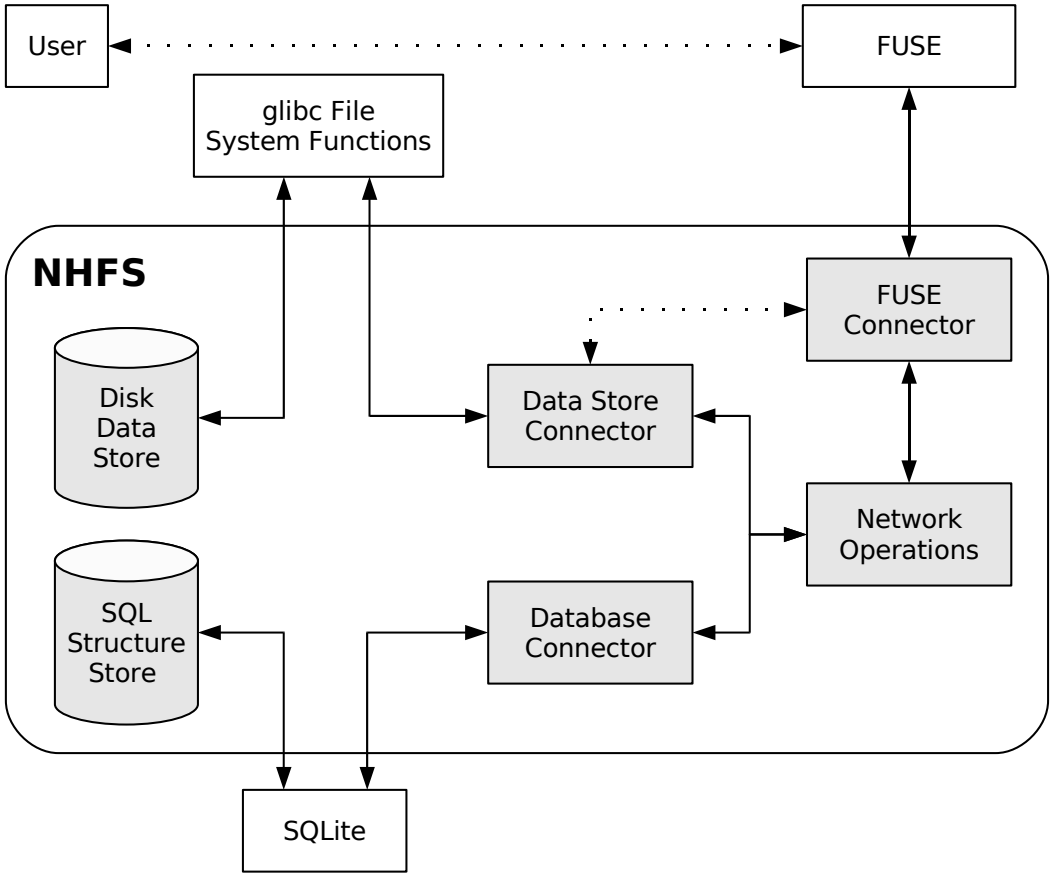

Figure 9: NHFS system architecture overview

functions (called *network operations* in figure 9) that perform operations on NHFS network[17] nodes.

Any file system access to NHFS passes through the usual layers of the operating system (see the last section for a more detailed description of the pathway). At some point, the access reaches FUSE, which relays it to NHFS. Within NHFS, the FUSE connector is responsible for receiving any such access from FUSE. Depending on the specific file system access, the FUSE connector now calls certain network operations (i.e. functions that operate on nodes in the NHFS file system network). The network operations in turn are responsible for manipulating nodes and connections between nodes in the file system; they use the database connector to manipulate the file system structure, or change node properties in the structure store. Furthermore, by use of the data store connector, they manage the nodes' data, which is stored in the data store. SQLite is utilized by the database connector to actually perform SQL queries to query or alter information about nodes or connections in the structure store (see also the next section). Likewise, the data store connector accesses the data store by means of standard glibc file system functions (see also "Storing the Data" on page 29).

For some trivial file system operations, NHFS diverges a bit from the standard process described above.

[17] in terms of: network of file system nodes

These exceptions consist of operations that do not require accessing the structure store, and can be performed almost exclusively in the data store. In these cases, the FUSE connector directly calls the data store connector to execute requested action without using the database connector.

Storing the Structure of the File System

To store information about what files and directories exist in NHFS and which of them are interconnected, NHFS uses an SQLite database. SQLite is a library that allows creating and accessing databases using standard SQL without the need for a database server. The database itself is stored in a file and any program using the SQLite library may access it. In contrast to usual database servers like MySQL, no configuration is necessary. Benchmarks show that SQLite is very fast in many usage scenarios, often even faster than standard database servers like MySQL[18]. This is especially true for single-user access.

A database is needed as a place to keep the network structure of files and directories created by the user when storing new items and making new connections between existing nodes. For each node in the network, the database contains its id, the name of the file, and all the connections that it has to other nodes.

The database structure needed for NHFS consists of 2 tables. The table "node" is used to store information concerning files and directories in the file system. For each record it contains an id (integer, primary key) and a name (text, the name of the file or directory). The second table, "edge", describes the interconnections between the nodes. It has two columns, id1 and id2, both of which are integers and specifiy that two nodes that are connected by an edge going from the node with the id "id1" to the node having the id "id2". These tables are created using the following SQL commands:

```
CREATE TABLE node (id INTEGER PRIMARY KEY AUTOINCREMENT, name TEXT NOT NULL);
CREATE TABLE edge (id1 INTEGER NOT NULL, id2 INTEGER NOT NULL);
```

Their structure can be visualized as:

```
table: node         table: edge


id   | name         id1  |  id2
 ---  ---  --- --          ---  ---  --- --
INT  | TEXT         INT  |  INT
```

Before NHFS can be used, a few basic nodes and connections need to be inserted into the database: The root node, which stands for the root of the NHFS file system; its name is "/". A node with the name "trash" needs to be created so the user can safely remove files and especially directories from the file system (see 3.5).

[18] http://sqlite.org/speed.html

```
INSERT INTO node VALUES ( NULL, "/" );
INSERT INTO node VALUES ( NULL, "trash" );
```

The root and trash nodes also need to be connected to make the trash appear in the root of the mounted NHFS file system:

```
INSERT INTO edge VALUES
(
  (SELECT id FROM node WHERE name="/"),
  (SELECT id FROM node WHERE name="trash")
);
```

When the user creates a new file or directory, NHFS adds an entry to the "nodes" table and a record to the "edge" table, describing the connection between the directory where the node was created and the new node itself.

Storing the Data

While the network structure is stored in an SQL database, the data itself (i.e., the files) is kept in a special data store directory on another file system the user has access to. For any node in NHFS representing a file or directory, the data store directory contains a directory named after the id of the node. This id directory contains the actual node data as an actual file or directory. This has several advantages:

- File system operations are just as safe as in the file system where the data store is located.
- Metadata such as permissions, owner, group, modification and access times is taken care of by a standard file system.
- Many file system calls such as stat(), chmod() can simply be redirected to the corresponding item in the data store directory.

Programming Language: C

C was chosen as the programming language for the NHFS prototype because it had the best FUSE bindings at the time the programming work for this thesis was begun. Also, unlike languages like Python or Java, C does not impose any dependency on the operating system where NHFS is to be installed.

3.5 Usage

One of the goals of NHFS was to create a file system that is fully usable by standard graphical file management software and the command line, without the need for additional tools, commands, or other software.

Running NHFS — Mounting

Just like any other file system, NHFS needs to be mounted in order to make it accessible. Mounting is the process of telling the running operating system where the new file system should become accessible (the mount point) and which type of file system driver should be used to access it (the type of file system).

For NHFS, mounting also means to set up the basic prerequisites needed for file system operations, which is the data store (see 3.4) and the NHFS structure database (see 3.4). If it does not exist yet, the data store directory needs to be created; in case of a new file system, an empty database is created in the data store as well. After making sure these prerequisites are fulfilled, the NHFS program is run, which mounts the file system.

To ease this process for the user, all of it is performed by a utility script called nhfs. This script is run by the user and takes as arguments the mount point and the path to the data store. After running the script, NHFS is accessible at the user supplied mount point.

Performing File System Operations

Once NHFS is mounted, practically all file system operations can be performed just as in any other file system. Opening (reading), storing (writing), changing file permissions and ownership, creating symbolic links and hardlinks — all standard Linux file system operations are supported. There are two operations that are new or possess different semantics now: Removing directories and creating new connections between nodes in the file system.

Removing Directories Calling “remove” on a directory from the command line or a file manager usually recurses through all sub directories of the directory being deleted, removing everything contained in them first, before moving on to the directory itself, removing it eventually. This recursion is done by the user space program, e.g. the command line “rm” program.

Removing a directory in a file system structured like a directed, cyclic graph actually refers to two different things. When the directory being removed is the root of a monohierarchical sub tree of the graph, it can be removed as usual. But what happens if somewhere, reachable from that directory, exists a connection to some other part of the file system? If the normal “remove” were to be used, the resulting recursion would result in more files being deleted than intended. This is because recursive deletion is handled within the user

space program requesting the action (i.e. the file browser), which has no knowledge of the polyhierachical character of the file system.

So what is needed to safely delete directories is a special "remove" function. Because all user space applications need to be adapted to any change to the file system, its functionality is almost impossible to extend. For NHFS, the choice was to create special directories that allow to use the new features. The "trash" directory in the NHFS root directory is one of them. Instead of removing directories directly, the user needs to move them to the "trash" using the normal move commands, e.g. "mv" on the command line. Any item being relocated to the trash is checked for connections to it from other nodes in the file system. If no other connections exist, the item will be located in the trash from then on, from where it can be safely deleted using the usual remove functions. If the other case is true, and the directory being moved to the trash is connected to other directories in our file system, then the one connection in question, i.e. the one that is subject of the current relocation to the trash, is simply dropped from the file system.

To prevent accidental deletion in directories other than the "trash", NHFS could disallow such commands and simply return a suitable error code ("permission denied"). The "trash" would be the only directory allowing active delete commands coming from the user. Unfortunately, such a behavior would be impractical since many applications delete files, e.g. temporary files, without explicit user input.

Creating New Connections Creating a new connection between nodes in the file system means to make directory "a" not only appear in "b" but also in "c" — a new connection is made between "a" and "c". To expose this functionality to the user, a special directory appears in every directory in NHFS, it is intuitively named "link here". For any file or directory moved to a "link here" directory, NHFS creates a new connection in the file system structure between the former and the parent of "link here".

An example command line session would look like this[19]:

```
$ ls
 a  b  c  link here
$ ls b/
 a  link here
$ ls c/
 link here
$ mv a c/link\ here      # move a to c/link here
$ ls
```

[19] $ denotes the command line input, # a comment. The space in "link here" needs to be escaped using a backslash in most shells.

```
 a  b  c  link here      # after moving, a is still here
$ ls c/
 a  link here            # but a is also in c now
```

It was also considered to use the symbolic link or hard link functions to create new connections within the file system. However, hard linking directories is not allowed by the Linux kernel. The functionality to create symbolic links should be preserved as they are widely used and also necessary to create pointers to files and directories across file system boundaries, which is not possible by means other than symbolic links. For these reasons, neither hard nor symbolic linking could be used as an interface to this feature of NHFS.

3.6 Technical Evaluation

Compatibility Observations

NHFS was tested using a variety of different applications, graphical and command line, and no strongly disturbing incompatibilities with existing software have been detected. Considering how fundamentally a NHFS file system structure differs from a conventional one, this does come as a surprise.

The most serious compatibility problems observed were related to caching, probably within user space applications. All graphical file system browsers, for instance, keep a cache of the current state of parts of the file system and its structures. This cache naturally becomes inconsistent with the file system when some file system operations are performed by NHFS. Specifically, moving a file or directory into a "link here" directory leads some user space applications into a state where their caching mechanism has the information that the item has been relocated, so it concludes that the item does not exist in its original location anymore. In this case, this conclusion is false, of course. In all applications that were tested that have some kind of "refresh" button, pressing this button resolves the issue.

Performance Observations

The speed of file system operations is suitable for daily use in many usage scenarios. Of course, it is by far not comparable with "real", low level file systems like "ext3". The speed of operations is often more than ten times slower, which is not surprising, considering that NHFS was not designed with performance aspects in mind and that it is a prototype. This slowdown becomes particularly apparent when reading or writing a large number of small files, especially if these are structured in rather deep directory structures. Longer paths cause a much higher load on the SQLite structure store, because these file paths need to be resolved to their corresponding node IDs. These performance observations are most certainly also related to the fact that NHFS does not support multithreading (see 3.7).

3.7 Important Features not yet Implemented

Cycle Check or Limitation of Path Length

By traversing recursively through cyclic regions of a nonhierarchical file system structure, the user is quickly able to construct an extremely long directory path (e.g. “Beatles/Lennon/Beatles/Lennon/Beatles...”). Since this confuses a number of programs used to access files, there should be a limitation on the maximum length of cyclic paths. After this limit, no directory content should be visible, instead a symbolic soft link which takes the user back to the root of his mounted NHFS filesystem should be shown.

A problem of this approach are the differences in how programs handle symbolic links. Some applications will simply insert the symbolic link into the path; others will substitute their current path with the path contained in the symbolic link (if it is an absolute link) or compute the new path based on the relative path in the link (relative symbolic link).

Another solution for the problem of cyclic paths would be to only show a directory as a symbolic link if the directory is already in the path the user has just navigated through. As an example, suppose there are two directories in a NHFS root, ”Beatles” and ”Lennon”. Each of these directories is linked within the other, so normally we would be able to browse through them and could end up with a path like ”Beatles/Lennon/Beatles/Lennon/Beatles”. As soon as we entered ”Beatles/Lennon”, ”Beatles” would not be shown as a directory anymore, but as a symbolic link to ”/Beatles”. We would not be able to create a cyclic path.

Identical Inode Numbers for all References of an Item

Just as hardlinks in the file system share the same inode number, files and directories that are in fact identical within NHFS should have the same inode number. This could, theoretically, help tools like “find” avoid cyclic directory traversals.

Inotify Support

Inotify is a file events monitoring subsystem of the Linux kernel. It can be utilized by user space programs to watch events for individual files or whole directory subtrees. In order to monitor events, a program first has to initialize an inotify instance using inotify_init(), which returns a file descriptor for the new inotify event queue. Using this file descriptor, a watch can be added on a file path (inotify_add_watch()). The call to inotify_add_watch() expects a bit mask describing the types of events to watch; these can be any type of file system event, like accessing, changing metadata, opening, closing, moving, and others.

Since inotify is handled by the kernel internally, and the kernel does not know that two paths actually refer

to the same file or directory within NHFS, file system events of one of the two objects do not automatically propagate to the other as well. Inotify support should be added to NHFS to make this propagation occur for events where it is needed and makes sense. For instance, an access event should be propagated to all references of a file or directory; a deletion event should not be propagated, since in NHFS, a deletion refers to only one reference being deleted, and usually not to the deletion of the referenced object itself.

Whenever a file system event that should be propagated within NHFS occurs, NHFS needs to find all other references of the object which is the source of the event, and then raise the same event for all references. Unfortunately, this can be a rather expensive task due to the cycles that can exist within the NHFS' file system structure. The task could be alleviated by finding out which file and directory paths the kernel keeps an inotify watch for at the moment, and only raising the events on those paths. Another approach to this problem would be to rely on cycle checks and the limitation of directory path length (both: see above) which in theory should prevent user space programs to open inotify watches for deeply recursive paths.

Quick Item Deletion

At the current state of implementation, items have to be deleted by first moving all of their references in the file system to the special trash directory. After the last reference has been moved, the item appears in the trash and can be deleted by using the conventional "remove" command. This is impractical if there is an item which has many connections to other items, and the user wants to delete it as quickly as possible. This functionality could be implemented as another virtual directory, e.g. "delete now". If an item was moved to "delete now", all references the item still has would automatically be removed, and then the item would be deleted.

Multithreading

Due to limitations in the SQLite library, NHFS cannot be run using multithreading. This is normally ensured by the "nhfs" script, which sets up the necessary environment and executes the nhfs binary. Multithreading would make it necessary to completely isolate the database access code from the rest of NHFS, in order to keep database accesses restricted to a single thread.

Hard Links with Differing Names

Hard linking is allowed in NHFS, but all hard links to a file must have the same name. Trying to create a hard link which has a different name than the file it references will result in an error message. Creating a hard link in NHFS creates a new connection between a file and a directory, so it is another method of associating files with directories.

Hard links with differing names are strongly contradictory to the idea of a file system structure where each item can be referenced any number of times, simply because they would mean that there could be many references of a file, all with a different name; this would make it very difficult for the user to identify whether a file is only a reference to another file.

Since only hard links with identical names are allowed, it is not possible to make a hard link to a file in the same directory the file is in, because there can never be two or more items with equal names in the same directory.

4 Evaluating the Usability of NHFS

The main goal of the implementation of NHFS was to improve the usability of the file system. Of course, this usability improvement should be quantifiable. Since the complete design and execution of a usability study would go beyond the scope of this thesis, this section depicts a number of ideas about how such a study could be designed and performed.

4.1 General Usability Considerations

What is Usability?

Usability can be defined in a number of ways. In "Guidance on Usability", the International Organization of Standardization defines it as follows (ISO 9241-11, from 1999):

> The extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use.

Nielsen [1993] describes usability to be a part of the usefulness of a system, and to be comprised of:

- learnability: being easy to learn
- efficiency: after having learned the system, the user should reach a high grade of productivity
- memorability: after not having used the system for a period of time, the user should still remember how to use it
- error prevention: the system should prevent errors, or at least make them easily recoverable
- satisfaction: users enjoy using the system

Nielsen [1993] makes a clear and strong distinction between usability and utility. For him, utility and usability are part of the usefulness of a system. Utility refers to the functionality, while usability can be described as the accessibility of that functionality to the user.

Goals of Usability Evaluations

Generally, the goals of the different types of usability evaluations can be described by three basic questions (see Gediga and Hamborg [2002]):

"Which is Better?" This type of evaluation compares different products or prototypical designs in order to choose the best alternative. Since NHFS was developed as an attempt to improve standard file system usability, comparing NHFS with a standard system would answer the question if this attempt was successful.

"How Good?" Answering this question aims at quantifying how strong a certain characteristic is reflected in the software. For instance, the characteristics to be evaluated can be specified as "usability goals" before the development of the system.

"Why Bad?" Finding usability weaknesses in software systems helps to generate suggestions and ideas about possible improvements. This is typically used to evaluate prototypes during the development process.

Types of Usability Evaluations

According to Gediga and Hamborg [2002], there are two major types of usability evaluations:

Descriptive Evaluation Methods Descriptive usability evaluation methods are used to objectively describe the state of a software system from the perspective of the user. Descriptive evaluations can use behavior based methods (such as recording the user's behavior while he is performing a task and "thinking out loud") and opinion based methods (e.g. questionnaires and interviews).

Predictive Evaluation Methods The goal of predictive evaluation methods is the discovery of weak points in a software system's user interface. They are also used to generate recommendations for the direction of the system's development. Examples for predictive methods are walkthrough methods, inspection methods, and heuristic evaluation methods.

Criteria of Usability Evaluations

Usability evaluation is actually only a widely accepted term that stands for "evaluation of software ergonomics". The concept of usability has been established as the basis for evaluation criteria of all evaluations of software ergonomics (e.g. Nielsen [1993]), hence the name "usability evaluation". This concept, which is of course much too fuzzy in order to be used as an evaluation criterion itself, is put in a little more concrete terms by ISO 9241-11, which states that effectiveness, efficiency, and satisfaction are the three major criteria of software usability.

Constructing criteria requires to at least keep in mind the following constraints: The characteristics of the future users of the system, the tasks the system is going to be used for, the type of study conducted (lab or field), and the type of object being evaluated (paper prototype, fully functioning system, etc.)

Collections of standard criteria possibly offer a solution for the problem of the construction of criteria. They provide proven reliability and validity and may be found in the form of standard questionnaires (Gediga and Hamborg [2002]).

4.2 Usability Evaluations in the File System Context

Considering the strict distinction of usability and utility described by Nielsen [1993] (see 4.1), which parts of the system can be judged by a usability evaluation in a file system context? What is the user interface of a file system? The programs (command line and graphical) to access the file system can certainly be considered as part of the user interface. What are the functionalities, to which the term utility refers, of a file system? If we accept that the basic functionality of a file system is to store files and directories and the relations between them, it could also be argued that the polyhierarchical structure that is possible with NHFS is basically a way to repackage the atomic elements: files, directories, and connections between them, and therefore offers the user a new way to handle files and directories.

Also, usability is certainly influenced by functionality. Functionality can enhance usability; e.g. a print dialog that would require the user to configure the printer each time it is called would most likely be much less usable than a print dialog that simply offers the user all previously configured printers. In cases like this it is hard to draw a strict line between usability and utility (functionality).

The definition of usability in ISO 9241-11 (see 4.1) is a bit different than the one given in Nielsen [1993] and will be used as a basis in this thesis. The ISO definition mentions three main goals: effectiveness, efficiency, and satisfaction. It also requires that the users of the system, the goals of the task, and the context of the use are specified.

4.2.1 Aspects to Evaluate

NHFS extends standard file systems with a function that allows the user to make a file or directory appear in any number of directories. This functionality can be accessed by using special virtual directories (see 3.5). So there are at least two different elements that can and should be evaluated.

Evaluating the NHFS' User Interface NHFS uses virtual directories to expose its special functionality to the user. Any directory created by the user will automatically contain a virtual directory called "link here". Whenever the user uses standard file system utilities (graphical or command line) to move a file or directory to another directory's "link here" directory, the object being moved will be linked (in the NHFS' meaning) to the other directory.

Due to the fact that other file systems do not have virtual directories that become active once a normal file system operation is performed on them, and that these amount to the core part of the NHFS' user interface, an evaluation of these aspects is desirable.

Evaluating the Polyhierarchical Character of NHFS With NHFS, the user can create arbitrary connections between files and directories. Using this feature will lead to the development of file system structures that are equivalent to a general directed graph of directory nodes with shallow (i.e., not more than one level), polyhierarchical subhierarchies consisting of file nodes. Two questions should be answered by an evaluation:

- Is the concept of a polyhierarchical file system easy enough to understand and does it offer enough obvious advantages for the user to employ it?
- When confronting the user with a file and directory system structure that has been prepared to contain a number of additional links between files and directories, what is his reaction to the polyhierarchical character of the system, and does he make use of it?

4.2.2 The Metaphor Problem

When a metaphor is used in the design of an element of the user interface, the designer needs a concept in reality that will be easily recognizable by the user when looking at the element. The user is then able to transfer knowledge about the concept in reality to the user interface element. For instance, one of the most widely used metaphors, the record/file metaphor, works by making the directory structure on the user's hard drive resemble a filing cabinet, e.g. in the user's office. As soon as he recognizes this resemblance, he can conclude that, just like in his real filing cabinet containing records and files, he should be able to create or change a record (i.e. file) and put it in a file (i.e. directory).

One of the weak points of NHFS is that in the reality of physical objects there can only exist a monohierarchical object organisation, simply because a record can be in *one* file only, a book can be on *one* bookshelf only, etc. Due to this lack of a counterpart of the concept of multiple allocations or relations (which are possible in NHFS) in reality, this concept might seem strange at first and may take some getting used to. No useful metaphors can be found to make the concept easier to understand for the user.

4.2.3 Design Proposals for Usability Tests

General Design Considerations

Users The NHFS' target user is someone who uses a computer regularly. Specifically, he would be someone who regularly engages in the task of file and folder management. The system would be most useful for users having many files and documents, e.g. people who use digital cameras or professional computer users.

Tasks All tasks that occur when managing files and folders, namely filing, locating, managing, and sharing documents can possibly be used in the evaluation.

Usability Criteria To measure the usability of a file system, many standard usability criteria can be used, such as the time it takes to complete specific tasks, user satisfaction, and the probability of error when performing specific tasks.

Design 1: Evaluating General Usability / Satisfaction in a Long Term Study The goal of this design would be to find out how well NHFS compares with standard file systems in terms of user satisfaction.

A very important aspect of a long term study would be the assured familiarity long term test users have with the system. This would ensure much better comparability with standard file systems, as users are usually very familiar with those already.

Such a study would require to find test users who agree to use NHFS for a longer period of time. It would take the users approximately 1-2 weeks to get familiar with the system. After that, a period of 4-6 weeks of normal usage would be necessary in order to ensure that most of the spectrum of the user's individual file management tasks has been performed using NHFS at least a few times.

After the test period, participants would be asked to fill out a questionnaire. Preferrably, this should be a standardized test, e.g. SUMI (Gediga and Hamborg [2002]) that predominantly rates how satisfied users are with the system.

A second group of test users would be necessary to rate the standard file system using the same questionnaire. The results of both questionnaires could then be compared.

A problem of this approach is that it would be difficult for users of the standard file systems to rate their file system, since most users probably do not understand the distinction of file systems and file managers, such as Windows Explorer. This problem could be circumvented by querying only users who have a deeper understanding of the subject. Therefore these test users would need to be carefully preselected. This preselection poses another problem: Since we would like to rate usability for the average user, those should consequently be queried by the study, and not only users from a small niche. As there is no strict group of target users of NHFS, this is an acceptable drawback if the study at least gave us an insight into the usability of NHFS as experienced from an expert perspective.

Design 2: Filing: Files and Directories An evaluation using this design would generate information regarding learnability and acceptance of NHFS.

Test users would be asked to come to the laboratory. After a short introduction to and training with NHFS, they would be presented with a number of files and directories with the task to file all of them into a

given (prepared) directory structure.

The files and folders to be filed would need to be easily classifiable by their names, and possibly filename extensions and icons representing the file type. Due to interindividual differences in the users' knowledge, the classification of these items must not require more than common knowledge. One approach to make items easier classifiable is to use image files which are displayed as thumbnails during the task.

The evaluation could make use of a scenario in order to make the task easier to grasp. One possible scenario could be that the user lives in a shared appartment which uses a network file server. He has been sent a number of files and folders from a flatmate with the request to store them on the network file server to make them available for the other flatmates as well. The fact that in this scenario the files and directories are shared with other people (who access them using the same directory structure) on the file server could be used to make the user consider in which directories other people would look for an item. This could lead to him trying to link the item into more directories than he normally would.

Several questions could be answered by an evaluation of this type. First, do users learn to use the NHFS's link feature quickly? Second, how strong is their acceptance for this functionality, i.e. do they use it if it is available to them?

One problematic part of this design is the above mentioned classifiability of files and directories, which needs to be assured. Also, how can the evaluator differentiate between a user who has not learned how to use NHFS quickly enough and a user who simply does not like its features (low acceptance)? An extra testing before the actual task could be used to ensure the user knows how to use NHFS. Another problem can be expected in the prepared directory structure into which the user is asked to file the items. Its structure should be as objective and easy to understand as possible. In reality, the user would create this structure himself, but as we want to avoid creating confounders, all subjects should use the same directory structure. A third problem is that users are generally not used to polyhierarchical structures, as in reality and on their computers, they use mostly monohierarchical ones. A study in the lab might not expose the user long enough to this new type of structure in order to make him feel comfortable with it.

5 Outlook and Conclusion

Over the course of this thesis, many future ideas have been conceived. These comprise ideas for an enhanced graphical user interface built on top of NHFS, and some feature ideas for the core of NHFS itself.

It is astonishing how strongly the domain of file management is lacking in innovation. Due to it being so deeply tied to operating systems and the way users interact with their computers, whoever is trying to innovate it is faced with discouraging headwind. The lack of commercial innovation leaves an empty space to be filled by the scientific field.

Just as software companies and academics have been targeting information and document management in the much larger scale corporate sector, it is possible and much desirable that the very basic form of document management, file management, will be targeted in the future as well.

5.1 Feature/Future Ideas

Exposing file system metadata through NHFS could make some common file management tasks much easier for the user. These tasks would normally require a program to perform file searches and some user interaction; with NHFS, they could be easily solvable by simply browsing to a directory.

Browsing by File Type

To alleviate tasks like browsing all jpeg or pdf files on one's computer, NHFS could be extended by a virtual "file types" directory containing a virtual directory for each file type. Each of these file type directories could list all files of that type. It would be necessary to keep file type information in the NHFS file database. Accessing "file types" would trigger NHFS to list all distinct types of available files as virtual directories.

Browsing by Date

A timeline view of the user's files and directories could be created by having a virtual directory "dates", which would contain virtual directories named after certain time ranges, e.g. "today", "yesterday", "this week", "last week", and so on. These directories would contain objects added to NHFS (or changed) within this time range.

Of course, NHFS would need to store time stamps for last modification, access, and creation time for each object in the file system structure database, a feature currently not implemented.

Recently Accessed Documents

To enable easy access to the most recently accessed files and directories, a special “recent” virtual directory would contain a number of the most recently accessed files and directories. In order to provide such functionality, NHFS would need to keep access times for all objects. Whenever a directory listing is requested on the “recently accessed” directory, NHFS would list a certain number of files and directories, starting with the most recently accessed ones, ordered by the time of last access. Should the user want to mount NHFS without the ”atime” option, this feature could be disabled.

Plugin Infrastructure

For simpler extension of NHFS (also by third parties), a plugin infrastructure could be created. A plugin would be registered for a designated virtual directory, and some or all file system operations would be redirected to the plugin responsible for this virtual directory. The plugin would use the NHFS functions to directly access the file system database.

Enhancements for the Graphical User Interface

To provide the user with a better understanding of the file system structure, a file system browser or manager that visualizes the structure graphically could be created. This browser would probably have to access the NHFS file system structure directly, without using the glibc functions and the VFS. NHFS would have to be built as a library exposing additional functions to access the file system directly.

5.2 Conclusion

In the light of recent technological developments that make easier access to globally distributed information as well as storage of ever larger amounts of data possible, this thesis makes clear that it is necessary to provide the user with more powerful tools for the organisation of files and directories. It also shows that by introducing a relatively small extension to conventional file systems it is possible to give the user completely new possibilities of file organisation, while ensuring compatibility with most existing applications and the operating system. Whether this kind of extension is the right way to fundamentally improve usability of file management tasks is a crucial question that should be answered in a usability evaluation.

References

D. Barreau. Context as a factor in personal information management systems. *Journal of the American Society for Information Science*, 46:327–339, June 1995.

S. Bloehdorn, O. Görlitz, S. Schenk, and M. Völkel. Tagfs - tag semantics for hierarchical file systems. In *Proceedings of the 6th International Conference on Knowledge Management (I-KNOW 06), Graz, Austria, September 6-8, 2006*, SEP 2006. URL `http://semfs.ontoware.org/pubs/2006-09-iknow2006-tagfs.pdf`.

P. Dourish, W. K. Edwards, A. LaMarca, J. Lamping, K. Petersen, M. Salisbury, D. B. Terry, and J. Thornton. Extending document management systems with user-specific active properties. *ACM Transactions on Information Systems*, 18(2):140–170, 2000. URL `ftp://ftp.parc.xerox.com/pub/dourish/tois-placeless.pdf`.

J. Foo. Docplayer - design insights from applying the non-hierarchical media-player model to document management. Master's thesis, Linköping University, Sweden, 2003. URL `http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-2086`.

E. T. Freeman. *The Lifestreams Software Architecture*. PhD thesis, Yale University, May 1997. URL `http://www.cs.yale.edu/homes/freeman/dissertation/etf.pdf`.

G. Gediga and K.-C. Hamborg. Evaluation in der software-ergonomie. *Zeitschrift für Psychologie*, 210 (1): 40–57, 2002.

D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. James W. O'Toole. Semantic file systems. In *SOSP '91: Proceedings of the thirteenth ACM symposium on Operating systems principles*, pages 16–25, New York, NY, USA, 1991. ACM Press. ISBN 0-89791-447-3. URL `http://web.mit.edu/6.826/archive/S97/13-Gifford-Semantic-file-systems-paper.pdf`.

R. Grimes. Revolutionary file storage system lets users search and manage files based on content. *Microsoft Developer Network Magazine*, 19(1), 2004. URL `http://msdn.microsoft.com/msdnmag/issues/04/01/WinFS/`.

B. Nardi, K. Anderson, and T. Erickson. Filing and finding computer files. In *Proceedings East-West Conference on Human Computer Interaction*, pages 162–179, Moscow, Russia, July 1995.

J. Nielsen. *Usability Engineering*. Academic Press, 1993.

J. W. O'Toole and D. K. Gifford. Names should mean what, not where. In *EW 5: Proceedings of the 5th workshop on ACM SIGOPS European workshop*, pages 1–5, New York, NY, USA, 1992. ACM Press. URL `http://www.psrg.lcs.mit.edu/history/publications/Papers/sigops.htm`.

D. M. Ritchie. The evolution of the unix time-sharing system. In *Lecture Notes in Computer Science 79: Language Design and Programming Methodology*. Springer-Verlag, 1980. URL `http://cm.bell-labs.com/cm/cs/who/dmr/hist.html`.

D. M. Ritchie and K. Thompson. The unix time-sharing system. In *The Bell System Technical Journal 57 no. 6, part 2*, July-August 1978. URL `http://cm.bell-labs.com/cm/cs/who/dmr/cacm.html`.

J. Shapiro, V. G. Voiskunskii, and V. I. Frants. *Automated Information Retrieval: theory and methods*. Academic Press, San Diego, CA, USA, 1997.

UN. The world at six billion, October 1999. URL `http://www.un.org/esa/population/publications/sixbillion/sixbillion.htm`.

S. Wildermuth. A developer's perspective on winfs: Part 1, March 2004. URL `http://msdn2.microsoft.com/en-us/library/ms996622.aspx`.