Imperial College
London

Insight: A Semantic File System

Final Report

June 18th, 2008

David Ingram
Department of Computing
Imperial College London
david.ingram04@imperial.ac.uk

Supervisor: Dr Peter McBrien, pjm@doc.ic.ac.uk
Second Marker: Dr Chris Hogger, cjh@doc.ic.ac.uk

Abstract

For over 30 years, hierarchical file systems have enforced a limiting mode of thinking upon users, requiring them to organise their files into specific paths. Despite this, humans naturally tend to associate objects in the physical world with a set of loosely-defined attributes, rather than a well-defined position.

Many users will say that they cannot locate or organise the files they have created, either because they can no longer remember the names they gave the files, or because they can only recall information about the subject of the files or data contained therein. Users therefore revert to searches, which may be time-consuming and frustrating, as they may not be able to search for the data they can recall.

In order to provide a closer match to the way the mind organises data, file structure should not be solely based upon one unique hierarchical location but custom semantic attributes assigned to the data. These attributes may take the form of keywords (e.g. *amusing*), structured keywords (*document.report*), key–value pairs (*filetype* = '*mp3*') or more complex structures.

The aim of this project, therefore, is to build a proof-of-concept semantic file system for Linux, providing fast keyword- and key–value-based indexes that will allow users to find and structure their files as they require, without needing to resort to awkward searches. This system should be available to every program that deals with files, offering a consistent method of locating data that is backwards-compatible with the existing hierarchical system.

Acknowledgements

A project such as this can never be the work of just one person. Many people have provided input, advice and assistance, and without them it would not have been possible. I would therefore like to thank the following people for their involvement with this project:

- Dr. Peter McBrien, for kindly agreeing to supervise my project proposal and particularly for his assistance in the late stages of the project.
- Dr. Chris Hogger, for his suggestions and enthusiasm.

And to those who have contributed in other ways:

- Nick Pope, for keeping my visions realistic and for some stylistic assistance with LaTeX.
- Katie Stevens, for not begrudging me the time spent on this project, for her superior mastery of the English language, and for providing the viewpoint of a normal user.
- David Durant, for his advice, suggestions, and not least his criticisms that helped shape my reports.
- My housemates, for their advice, suggestions, and cooking skills.

This project comes at the end of four years' study at Imperial, and it would be impossible to mention by name everybody who has shared it with me. I would therefore like to thank my friends and family for their support, and for sharing both the good and bad experiences throughout my degree. I could not have done this without you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

> Therefore, since brevity is the soul of wit,
> And tediousness the limbs and outward flourishes,
> I will be brief
>
> *Hamlet Act 2, scene 2*

File systems are an integral part of day-to-day computer use that are often taken for granted and therefore often overlooked. They have remained largely unchanged semantically for thirty years, and the main innovations have been related to clustering, network file systems, and reliability. There have been no real changes to the way in which users interact with the system.

Organisation is a difficult problem which users face every time they deal with files. Most users have evolved personal systems of categorising their files. However good these may be, they cannot be perfect. Just as electronic files are intended to be a superior analogue of paper files (and very often are), electronic folders should be more powerful and more flexible versions of their physical counterparts.

In order to improve upon the concept of folder-based storage in the majority of existing file systems, the major contributions of this project are:

- Design of a semantic file system (Chapter 4), including a syntax for legacy application access (Sections 4.1 & 4.2) and query engine (Section 4.5)
- Implementation of a proof-of-concept semantic file system for Linux (Chapter 5)

The remainder of this chapter deals with the project as a whole, while Chapters 2 and 3 cover background material referred to elsewhere in this report. As this project has very broad scope, a number of suggestions for future work have also been included (Section 7.2)

1.1 Motivation

The main motivation behind this project is frustration with the current methods for organising files based on arbitrarily-chosen classification.

> **The key problem:** *Files can only (generally) exist in one location in a file system hierarchy, but there may be multiple appropriate categorisations.*

Deciding how to organise one's files can be difficult. Files can only be placed in one folder, which can be considered a category. Unless those categories are well-chosen, it may be difficult to locate files easily.

Unfortunately, there are a large number of possible categorisations for files. This can be illustrated with the following examples, each listed with potential difficulties:

- Organising files by file extension
  - Different folders for `.txt`, `.doc`, and `.pdf` files.
  - However, this destroys any relationship between files, e.g. a timesheet stored as a spreadsheet and an invoice stored as a document.
  - Then again, not all files ending with `.pdf` may be documents – they might be brochures or posters.
- Organising files by similarity
  - Keeping files closely related to one subject together, e.g. all files related to coursework 1 of course 217 of year two of university could be stored in `Uni/year2/217/cw1`.
  - Likewise, pictures for a trip could be stored based on the date they were copied from the camera: "`Pictures/20070911: Paris`".
- Organising files based on internal information
  - Collecting pictures of sunsets, rivers, forests, trees together no matter when or where they were taken.
  - Organising music into folders based on artist, album, track number and title, e.g. "`/music/Queen/Greatest Hits II/17 - One Vision.mp3`".
  - Organising music by genre, or other attributes such as whether it is an instrumental track.

All of these methods have their advantages and disadvantages, but only one can be used at a time. If you had initially organised your pictures based on the date they were copied from the camera, and then you wanted to find a picture of a sunset over the mountains, it might take a significant amount of time to find. Content-based organisational schemes for the same data tend to be mutually exclusive, such as the two music examples given above.

This state of affairs has been accepted by users as being a natural limitation of the computer systems they use, but this does not have to be the case. With ever-increasing numbers of files, even the best organisational models can break down.

In corporate settings, web-based document repositories may be implemented that store additional metadata which is subsequently used to index documents in a more consistent way depending on the user's status. If a large number of reports about a multi-national corporation are stored, then different people will want those reports broken down in different ways: by geographical area, by company, by fiscal entity, by date, or by report type. The different orders of these classifications would make the design of a single hierarchical system an organisational nightmare.

1.2 Aims

There are many possible solutions to this problem of organising files. Organisations may implement web-based document repository systems, but these can be inflexible and introduce more steps into the task of opening and saving files.

There are a number of existing systems that use some metadata about files to make them easier to find. Many music players have a *multimedia library* feature which will index the artist, album and title fields along with the file name and other metadata to help users find the music they want to hear.

Specialised metadata-indexing systems such as these perform their purposes well, although their indexes are locked inside the programs, so an instant-messaging client could not make use of the multimedia library of a music player without specific interoperability code.

The primary aims of this project were:

1. Create a semantic file system for Linux that allows users to organise files in a more intuitive manner.

2. Allow users to create a dynamic categorisation for their files, related to their meaning (semantics) rather than just placing them in a single category/sub-category.

3. Provide an interface that is backwards-compatible with existing programs that rely on path-based file systems.

There were also a number of secondary aims, which would serve to highlight the use of a semantic file system in real applications:

1. Create a file import program that would automatically assign some attributes to the file.

2. Create a demonstration program or programs to illustrate the flexibility of a semantic file system.

3. Develop a plugin for an existing program that would enhance its usability by using this file system.

1.3 Tradeoffs

As with any new developments, there are bound to be some tradeoffs. The most obvious is that of speed vs. functionality. As more features are added, the system becomes more useful, but the speed of operation will very probably decrease. However, as this project provides only a proof-of-concept implementation, speed should not be an issue unless it is unusably slow.

One other performance tradeoff that should be considered is memory usage against speed of operation. Accessing blocks on a hard disk is a relatively slow operation, as there are moving physical parts involved. Keeping data in memory is much faster, at the cost of denying that memory to other applications. The focus of this project is not on optimum real-world behaviour, and so concerns about cache size are not relevant.

1.4 Solution

This project presents a partial solution to the problem of organising files. It provides a virtual file system layer that refers to files that currently exist on a user's computer, elsewhere in the directory hierarchy. These files can then be transparently accessed, read and written to via Insight.

Directories translate to this system in a natural way and can be thought of as categories for the files they contain. These may contain a number of subcategories (e.g. `type` would have subcategories `music` and `video`), which may contain still further subcategories.

Files can be members of multiple categories at one time, and will show up in directory listings for parent categories as well. This allows users to find files quickly without having to navigate down to the exact subdirectory.

Another feature of the file system is that standard directory traversal can be thought of as adding progressively more specific filters to the list of files you want to see. This means that the path `foo/bar` would only return files tagged with both `foo` and `bar`.

1.5 Note on terminology

Please note that the plural form of *index* used throughout this document is *indexes*, for consistency. In addition, the word *attribute* may be used in a general sense to cover simple keywords, key–value pairs, structured keywords and structured key–value pairs, for brevity.

It is also worth bearing in mind that the terms *semantic file system* and *database file system* are often used interchangeably, although this is not technically correct. A semantic file system implies some measure of reasoning about the stored data, whereas a database file system does not. Many semantic file systems are built upon databases in one form or another, and so the terms may be legitimately interchanged.

Chapter 2

History

> Those who cannot remember the past are condemned to repeat it.
>
> *George Santayana*

2.1 Early file systems

File systems as we know them have been around since at least 1964, with the advent of DECtape. This provided a very durable and reliable storage medium for the operating systems of the time running on Digital Equipment Corporation computers, starting with the PDP-6. This provided a basic way to name data, but little more than that. Other file systems around the same time also had no directory hierarchy, and these *flat file systems* (see Figure 2.1) were still being written as late as 1985, including the original Mac file system and early versions of the CP/M file system.

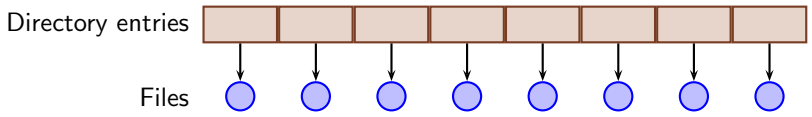


Figure 2.1: Flat file system [22]

However, by 1972, Version 6 UNIX had introduced the V6FS file system, which had grown from Ken Thompson's paper–tape-based file system for Multics, written in 1969 [28]. This *hierarchical file system* (see Figure 2.2) was one of the first to introduce the idea of directories, thereby providing users with a way to keep their files separate. The hierarchical structure was kept over the next three decades, being adopted by Microsoft in their FAT file system series, as well as Apple in HFS.

The hierarchical file system provided many of the features we now take for granted, such as directory structures, file owner and permissions, timestamps and theoretically unlimited path depth. This also paved the way for the "devices-as-files" paradigm adopted by Linux and BSD.

2.2 Later technology

The next major development in file system technology was the introduction of *journaling*, which was first included in OpenVMS in 1979, with the ODS-2 file system. Although this provided

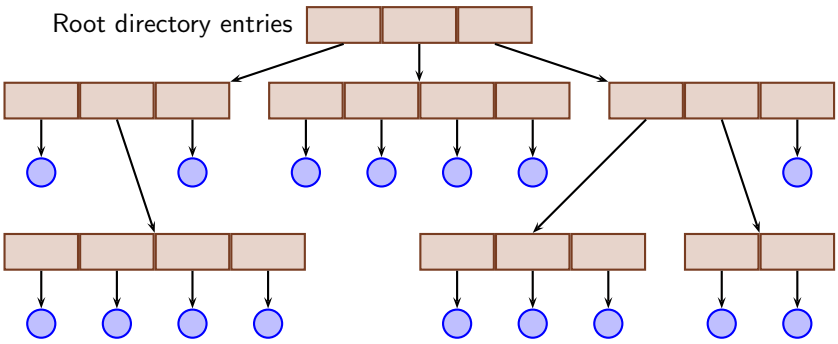


Figure 2.2: Hierarchical file system [22]

metadata-only journaling, it was still a big step forward. Journaling is the process by which a file system records changes (particularly to metadata) in a special log area before performing writes. Performing this step helps ensure metadata consistency in the event of a crash, although it does not necessarily guarantee user data integrity. The file system is then protected from structural damage or inconsistency, saving a large amount of time-intensive file system consistency checks at mount time may be skipped simply by replaying the journal.

This idea has been taken from the database world, in which data integrity is paramount. Full journaling is also available for some file systems, allowing user data to be recovered in the event of a crash, but it has a large performance penalty as every item of data must be written twice. Technology such as the *wandering logs* used in ReiserFS4 [17] may help reduce this penalty.

VxFS by Veritas was the first commercial *journaling file system* available, but NTFS was the first journaling file system to be widely used by both commercial and home users. Another feature introduced by these two file systems was *alternate data streams*, although the idea had existed in the Mac world for some time.

Alternate data streams attach more than one set of binary data to a given file name. On Macs, this is implemented as a *data fork* (the actual file data) and a *resource fork* (data that could be translated for different locales, or metadata containing the program used to edit a given file). Many file systems now allow a theoretically unlimited number of alternative data streams to be attached to a file, sometimes also known as *extended attributes*.

Other relatively recent developments in mainstream file systems are *access control lists* (ACLs), which provide fine-grained permissions for file and directory access. Although these were available in an early form in 1969 (providing permissions for the owner, owning group, and everyone else), they became more flexible and more popular with NetWare's NWFS in 1985 by allowing permissions to be set for multiple individual users and groups, and have been a more or less standard file system feature from VxFS in 1991.

2.3 Recent file systems

The Be File System (BFS) was written in 1996/1997, with the aim of producing a modern, 64-bit capable journaling file system as well as an indexing and querying system similar to a database. This idea of building indexes and query facilities into the file system has rather surprisingly remained largely unique to BFS.

One of the most common file systems in use under Linux is the *ext3* file system, which grew from the Second Extended File System (*ext2*) with the addition of journaling. The fast and very

stable *ext2* file system has been used since early 1993, and has become the standard for benchmark comparisons. It was a rewrite of the Minix file system, which gave the designers the opportunity to include a number of ideas from Berkley's Fast File System (FFS). *ext3* provides all of the features expected from a modern Linux file system, including journaling, access control lists, hard and soft links, and extended attributes.

Another popular file system for Linux is *reiserfs* by Namesys. This file system has all of the features of *ext3*, along with greatly increased scalability and speed (particularly with small files) and larger limits on file and partition sizes. The next version of this file system (*reiser4*) is currently being written, and has shown itself to be stable for the majority of users. Despite this, it has not yet been accepted into the Linux kernel [1, 2]. One other fairly popular file system is XFS, which was designed by SGI to be a high-performance journalled file system for IRIX.

Most recent file systems tend to focus on clustering or multi-user performance rather than adding new functionality (such as GFS, GPFS, OCFS, and Lustre). For example, Google's file system (GFS) is highly distributed, with access controlled by a master server which holds the file system metadata. The slaves store an enormous amount of data, but they have a relatively high failure rate because there are a large number of them, and so the file system must take this into account with some redundancy. Most of these decisions were made in order to provide high data throughput, at the potential cost of latency.

2.4 The future

Difficult though it is to predict the future with any kind of certainty, the area of *semantic file storage* has been receiving some attention in recent years, particularly with Apple's *Spotlight* system[3], as well as the rise and fall of Microsoft's *WinFS* initiative [10, 25]. There have also been efforts for Linux-based computers, such as GNOME's *Storage* project, *Tagsistant*, and also *DBFS* for KDE. More details about these are given in Chapter 3.

It would seem, however, that each of these advanced semantic file systems will stay closely tied to one operating system, although there would appear to be no need for this restriction other than the potential technical difficulty. Part of this may be due to the fact that these systems have been based upon local relational databases or created to search existing hierarchical file systems. As Hans Reiser points out however [18], a relational model is not necessarily the best way to represent such unstructured or associatively-structured data, and multiple layers above the file system may indicate underlying inadequacies that should be addressed directly.

File systems tend to be written for specific purposes such as performance, security, or flexibility. With these differing needs, it is no wonder that there are well over 70 file systems available, some of which are extensions of other file systems, and some of which are actually built on top of existing systems. A new approach may bring a number of benefits, including reduced overheads in terms of both time and space, as well as the ability to easily create structures with complexities beyond file systems built upon relational databases. This also opens up the possibility of using visualisation tools to discover information about the user's data, for example by using a "tag cloud" style of visualisation based on popular keywords attached to files.

It may be argued that, with the increasing interest in semantic file systems[4, 11, 12, 16, 30], this is just a return to the days of flat file systems with advanced indexing. With a semantic file system, the actual location of files is unimportant and irrelevant, and so they can all theoretically be stored in what is effectively a single directory.

Chapter 3

Current State of the Art

> The trouble with having an open mind, of course, is that people will insist on coming along and trying to put things in it.
>
> *Terry Pratchett*

There have been many attempts to create semantic file systems, which have taken many different forms. A few of the more prominent projects are outlined in this chapter. Note that although not all of these projects are at the file system level, they do share some similar goals.

3.1 Document stores

A *document store* does not just provide indexing on files, but actual data storage for those files as well. For example, files may not be directly stored in a database, but referenced by external object identifiers, as in MWFS[20]. The distinguishing feature is that it is possible to put files into the store natively, rather than merely recording references to existing files elsewhere in the file system hierarchy.

3.1.1 Microsoft WinFS

Windows Future Storage (*WinFS*) was a code name given to a relational data storage/management system developed by Microsoft. The aim was to provide a SQL Server layer above the existing NTFS file system that would allow users to find files on their own terms, using a structured query language called *OPath*. The project was first demonstrated in 2003 as a storage system for an upcoming version of Windows, which has since been released as Windows Vista. It was designed to provide management of data, whether structured, semi-structured or completely unstructured.

A preview video called IWish [1] was circulated at the 2003 Professional Developers Conference, showing many of the concepts for WinFS. Unfortunately, despite the release of a promising beta 1 version[25], the project was terminated in June 2006[6] with some clarification from one of the developers, Quentin Clark:

> *[. . .] we are not pursuing a separate delivery of WinFS, including the previously planned Beta 2 release. With most of our effort now working towards productizing [sic] mature aspects of the WinFS project into SQL and ADO.NET, we do not need to deliver a separate WinFS offering.*[5]

[1] `http://is.gd/qAH`

Some of the concepts from the iWish video do not fit directly into the model of a file system, but of particular interest is the calendar application, which shows a novel way of organising photographs.

Despite the lack of completion, it seems that some of the vision of WinFS may still be alive at Microsoft[19], although they are keeping quiet about any future offerings for the time being. Although there has been recent work on Windows 7, it appears that WinFS as a file system has been quietly distributed between different projects[27] rather than as a file system, although there have been unsubstantiated rumours that it will be included in Windows 7.

3.1.2 GNOME Storage

GNOME Storage was a project with some similar goals to WinFS, namely giving users the ability to quickly find the files they were after based on file metadata. It provided natural language query parsing[14], but it was not a file system as such – rather, it provided a layer in the desktop workspace that would allow people to search for relevant files.

The Storage project, despite promising beginnings, has not been developed for some years now. The last-modified date on the project's revision control system is late October 2004[15], although the last major change was in July 2004. There were plans to make this system central to the GNOME desktop, but at the end of the day it was primarily a proof-of-concept system for Linux, developed around the time of WinFS to keep pace with technological advances.

GNOME Storage provided a number of automatic filters for extracting data from files for a limited number of file types, and the author attempted to ensure that users did not have to manually classify data.[13]

3.1.3 BFS

The BeOS operating system, created in 1991 by Be Inc., was written to run specifically on the BeBox computer. It was optimised for digital media work, and therefore had a number of novel features for the time. One of these features was a file system optimised for high throughput, with a unique indexing system[7].

BFS allowed users to create a number of custom-defined attributes to identify files, although this was also extended to other items, like emails. Users could then create live queries that update their results as soon as the related indexes change. These queries acted like folders, so that users could locate files based upon metadata they had specified.

3.1.4 Tagsistant

Tagsistant is a semantic file system that was released to the public in July 2007. It was written with SQLite and the File system in User Space layer (FUSE) and is compatible with Linux-, BSD- and OSX-based systems. The author makes the interesting point that:

> *For me, a semantic filesystem is basically a tool that allows one to catalogue files and to extract subsets using logical queries.*[26]

The author also states that a file system is the most universal interface available for all programs to use. This makes it as easy as possible to integrate with all other applications on the system.

With the release of version 0.2 in February 2008, some semantic reasoning capabilities were added. This allows Tagsistant to know that one tag is a subset of another, or that two tags are equivalent.

Unfortunately, this offering suffers from some speed issues, due in part to the query system and lack of SQL indexes. The author hopes to improve this in the future, however.

3.1.5 MWFS

Yet another approach to a semantic file system has been taken at Imperial College in the past, in the form of a third-year Computing group project in 2004/2005[20], supervised by Professor Susan Eisenbach. The aim was to solve the problem posed by hierarchies: data does not always fit neatly into one location in the hierarchy, and working out the best place to store a file can be awkward.

This implementation was again a layer above the traditional file system, supported by a combination of Java and PostgreSQL. It relied upon a client-server model, with applications logging into the MWFS server in order to access the files stored within. Again, although this was not a true file system, it provided file storage through the database, and could automatically infer some attribute values from the metadata stored in certain types of file.

No further work has been done on this project since it was finished in January 2005, however.

3.2 Metadata indexes

In contrast to a document store, a *metadata index* merely creates and maintains indexes of various elements of metadata related to files. The indexes may be updated automatically by means of file system notification hooks, or a *crawler* program may be run on a regular basis to update the indexes. Use of a crawler does however mean that the metadata held and searched upon may be outdated, and this is something that users should keep in mind.

3.2.1 DBFS

In 2004, Onne Gorter, a student at the University of Twente in the Netherlands worked on a database file system for his final year project[9]. This project was mainly written in O'Caml, and was not a true file system as such. Instead, it provided a layer above a hierarchical file system which interprets graphical searches as SQL queries against an SQLite database, and returns the results. In addition to this, it incorporates a "crawler" program that aims to keep the database in sync with any changes to the underlying file system.

The primary focus of DBFS was from a human-computer interface (HCI) perspective rather than a systems perspective. This made it less important that the underlying file system was still hierarchical, and instead tackled the issue of how users should interact with the file system. This was achieved by replacing the default open/save dialog boxes in KDE with a complete re-implementation that instead queried the DBFS backend.

The project is no longer actively developed, as the author does not have the time. He did state, however, that he was interested in developing it further, particularly for the GNOME environment[9].

3.2.2 Apple Spotlight

Apple's *Spotlight* desktop search tool was introduced in Mac OS X version 10.4 "Tiger", in April 2005[3]. It provides a way for users to quickly locate many different items in their computer, including items that are not files, such as system preferences. It also performs full-text search of documents, as well as the ability to constrain searches using created/modified dates, file size and file type.

The index is maintained by a crawler daemon program that is constantly running in the background, updating the index when files are created or modified. This crawler has a number of plugins for different file types, to allow it to extract and index more information about certain files, such as building full-text search indexes for text documents. Apple have released an API that allows

application developers to write their own Spotlight plugins to allow easier searching with their proprietary file types.

Spotlight also includes the facility (since version 10.5 "Leopard") to show a preview of some documents, so that the user may not have to open the application in order to verify their search result. Also added recently was the ability to make use of the indexes stored on other Macs over a network.

3.2.3 Google Desktop

In October 2004, Google released the first beta version of Google Desktop Search, an application which provides Google-style searches of a user's email, files, music, photos, chat logs and web history. Users may also install "Google Gadgets", which are mini-applications that provide various functionality, such as displaying weather conditions, personalised news or new email notification, to name but a few. These Gadgets may also be developed by third parties, and allow access to the search facilities built into the main application.

Google Desktop Search versions for Mac and Linux became available in April and June 2007, respectively. Although these may not share exactly the same feature set as the Windows application, it still works well across multiple platforms, even integrating with the Apple Spotlight tool.

3.2.4 Beagle

Beagle is a desktop search tool that started in April 2004, based upon the Apache Lucene data indexing engine[2]. It enables the user to search their "personal information space" in order to find what they are looking for. It provides a back-end service which runs a real-time crawler on a user's personal data, adding files as they are created and updated, indexing emails, instant messenger conversations and web pages as they arrive at the user's computer.

Beagle supports a large number of file formats and data sources, including various types of instant messenger program, email client, personal note programs, RSS news feeds and address books. It can also create full-text indexes and extract other metadata from office documents, text documents, help documents, images, audio, video, archives, and many more.

3.3 Semantic file system papers

A number of actual file systems have been created or used as thought experiments in research papers. A very brief overview of some of the more well-known file systems is given in this section.

3.3.1 SFS

The first concept file system that was used to describe the idea of semantic file systems was described in a paper[8] written by David Gifford, Pierre Jouvelot, Mark Sheldon and James O'Toole, Jr. and published in 1991.

This paper outlined the ideas and motivation behind semantic organisation of files, as well as some of the benefits it might bring. They created a proof-of-concept file system that created symbolic links to files elsewhere in the hierarchy based on attribute–value pairs defined by an automatic indexing system. This file system crawled all publicly-readable files on the host computer, storing and indexing them by passing them through a *transducer* module that understood the file type.

[2] `http://lucene.apache.org/`

The search facility was based upon the idea of *virtual directories*, or directories which do not exist on disk but are created on an as-needed basis. The contents of these virtual directories were the results of a query generated by passing through the hierarchy. An example of such a query was given in the paper:

> For example, in the following session with a semantic file system we first locate within a library all of the files that export the procedure `lookup_fault`, and then further restrict this set of files to those that have the extension `c`:
>
> ```
> % cd /sfs/exports:/lookup_fault
> % ls -F
> virtdir_query.c@        virtdir_query.o@
> % cd ext:/c
> % ls -F
> virtdir_query.c@
> %
> ```

This demonstrates the power of the semantic file system, and also shows that the virtual directories (e.g. `ext:`) are invisible. That is to say, they do not exist in directory listings, but can still be accessed directly. Note that queries are implemented by specifying the attribute as a directory, followed by its value. Should one wish to view all the values for an attribute, a listing in the attribute directory (e.g. `exports:` or `ext:`) would return the available values.

This paper was very important in the later development of semantic file systems, and has lead to a great deal of derivative work, including the vast majority of semantic file systems or desktop search tools in existence today.

3.3.2 pStore

The *pStore* file system[31] was proposed by Zhichen Xu, Magnus Karlsson, Chunqiang Tang and Christos Karamanolis in 2003. It built upon the earlier work by Gifford et al., and considered a generic and flexible data model for semantic file systems.

Their solution was to create a *semantic-aware* store named *pStore*, an extension to existing file systems that supports a wide variety of semantic metadata. The data model associated with this file system was based up the Resource Description Framework (RDF) [29] already created for the Semantic Web, and can be used to track arbitrary connections between objects in the store as well as providing standard attributes.

Many features introduced by this paper surpass the abilities of database-backed systems, including the ability for dynamic schema evolution, in order to 'capture new or evolving types of semantic information'[31]. It is also simple and lightweight, because many applications do not require the full ACID properties provided by database systems. In fact, some Unix file systems do not guarantee ACID properties if a file system should fail.

One of the other interesting features that Xu et al. considered was the idea of file versioning as a part of the file system. This would change the requirement for version control systems, and could open the possibility of recording past versions of operating system configuration files, for example.

The authors of the paper acknowledged that there will be many challenges in implementing such a system, but that the benefits would be very valuable, especially with regard to increasing productivity.

3.3.3 Automated Attribute Assignment

Although this is not strictly a file system, this paper[23] by Craig Soules and Gregory Ganger describes a method for accurately automating attribute assignment by context analysis. This allows the indexing engine to infer useful attributes for many files, making queries more effective without requiring more effort from the user.

The proposed system makes use of the fact that most people only use computers for a limited number of tasks performed by a small set of applications. These few applications are then responsible for creating the majority of the user's files. Adding some measure of intelligence to these applications that provide hints about the context of their content could greatly enhance the user's searches.

For example, if a user initiates a web search for a celebrity and then downloads a number of pictures, these are very probably pictures of that celebrity, and may be automatically tagged. Similarly, information may be gleaned from other sources, such as email, and applied to related data like attachments.

It is not just the user's actions in the program at file creation time that have a bearing upon its meaning. Associations between files may reach between programs. If a user accesses a number of text files at the same time, they may well be related. Accessing one file and creating another may indicate a dependency relationship.

The authors of the paper acknowledge that this requires further study, but may prove to be a useful tool in the automatic assignment of useful attributes to files. Users will be more inclined to use a system if it seems to know what they are thinking, and can act accordingly and provide reasonable defaults.

3.4 Other solutions

Alternative solutions to the problem of locating specific files do exist, such as Picasa for images, or iTunes, Amarok and Winamp for music. These solutions are user-space programs that effectively organise a small number of different file types, but that organisation only persists within the program itself. If the user wishes to locate a file by virtue of its metadata then they must use the facilities inside that program, which may then allow them to trace back to a particular file in the file system.

However, this leads to fragmentation of the file system organisational space. Users must open individual applications in order to locate files, and may therefore need to open multiple applications if they are looking for files of different types.

3.5 Summary

With the large number of systems available, it is possible to analyse them and note the successes or areas for improvement for each. This can help to build up some initial requirements or suggestions for the direction this project should take.

Project	Pros	Cons
WinFS	• Semantic storage • Reasoning about data • Atomic unit is a record in a file	• SQL-based layer above NTFS • Development abandoned

Project	Pros	Cons
GNOME Storage	• Integrated into desktop • Provides GUI and live preview	• SQL-based layer above VFS • No longer actively developed • Uses crawler program
BFS	• Automatically-updated indexes • Built into file system • Atomic unit is a record in a file	• Restricted to BeOS • No longer actively developed
Tagsistant	• Virtual file system • Requires manual tagging	• Based on SQLite • Speed issues
MWFS	• Very flexible	• Complex interface • Client-server model • Uses Java/PostgreSQL • No longer developed
DBFS	• Insight into GUI component	• Manual tagging required • GUI-only • Client–server • SQL-backed
Apple Spotlight	• Extensive GUI support • Not limited to files • Atomic unit is a record in a file	• Restricted to Mac only
Google Desktop	• Fast searches	• Limited indexing • No custom attributes • Relies on crawler program
Beagle	• Fast searches • Indexes IM conversations • Many auto-import filters	• Can be memory-hungry • Relies on crawler program • No custom attributes
SFS	• Many useful ideas for syntax	• No available implementation • Uses symbolic links
pStore	• Aware of semantics • Able to reason about data	• Not implemented
Automated Attribute Assignment	• Methods for assigning attributes	• No implementation • Requires co-operation from apps

From all of this information, some desirable and undesirable features of a semantic file system can be picked out. The desirable features in a semantic file system are therefore:

- Automatic attribute assignment
- **Manual attribute assignment**
- Automatic attribute updates without requiring a crawler program
- Good **operating system integration**
- **Structured attributes** for further organisation (i.e. sub-attributes)

- Indexing not limited to files but parts of files (e.g. individual emails in a mail spool file)

As this is an idealised list, it will not be possible to implement all of these features given the limited timeframe of this project. Therefore the main focus has been placed on those items in bold.

Chapter 4

Design

> The future, according to some scientists, will be exactly like the past, only far more expensive.
>
> *John Sladek*

This chapter provides the design behind the proposed semantic file system, called *Insight*[1]. In addition to the internal design details, some additional information is provided on how the concepts translate to existing path-based systems, to provide backwards compatability with existing programs.

4.1 Files and directories

Changing the way in which a file system operates at the conceptual level means that the traditional semantics of directories have to be revised. Instead of providing a single (sub-)categorisation for a file (Figure 4.1(a)), directories act more like filters in a query, which could be interpreted as intersecting sets (Figure 4.1(b)).

A file can be thought of as an element that belongs to one or more sets, rather than to just one categorisation. For example, pictures that naturally fall into multiple categories can be easily represented in this system, as they contain a wealth of (possibly unrelated) information.

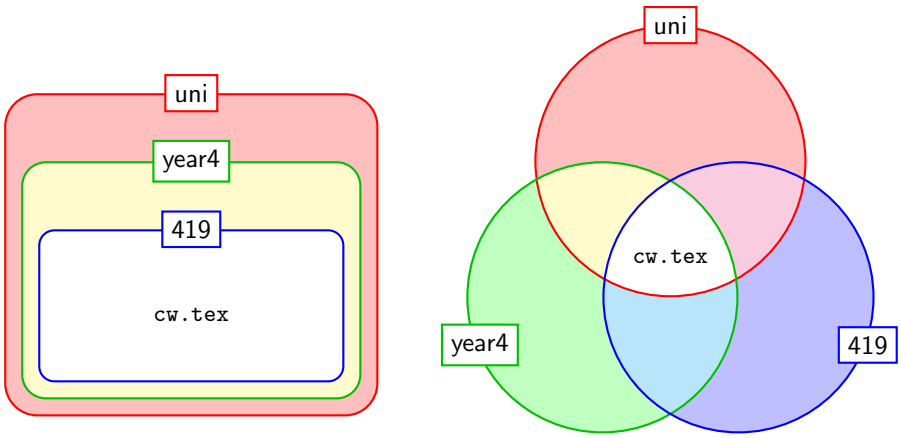
This method of arranging files takes some ideas from the paper by Hans Reiser about file systems and naming conventions[18] mentioned in Section 2.4. He argues that people naturally categorise data using a combination of ordered and unordered sets.

As files and directories change their meanings in this system, symbolic links to directories also have a different effect. There may be attributes that are synonymous, for example picture and image and photo. One of these attributes can then be chosen as the "canonical" attribute name, and the other synonyms can then appear as symbolic links to it.

4.2 Path syntax

Due to the complex nature of Insight, the paths it accepts must follow some syntax rules. These rules can also provide shortcuts for accessing folders.

[1] **I**nsight: **N**ew **S**emantics via **I**ntuitive **G**rouping and **H**uman-friendly **T**echnology



(a) Category view (b) Set view

Figure 4.1: Two views of the path `/uni/year4/419/cw.tex`

Two characters are set aside as not being valid in a path because they have special meaning to the file system. The colon character (`:`) is used as a *subtag indicator*, and the backtick character (`` ` ``) is used as a *subtag separator*. These were chosen because they are standard ASCII characters but they do not often occur in path names.

When browsing, the user is presented with a folder called '`:`' if the current directory has any sub-attributes. This is one example of the *subtag indicator* character in use. Entering this directory will then list all of the immediate sub-attributes of the parent.

Insight also makes use of "invisible directories", as defined by Gifford et al.:

> A directory is *invisible* when it is not returned by directory enumeration requests, but can be accessed via explicit lookup. If [...] virtual directories were visible, the set of trees [...] would be infinite.[8]

For example, the paths `/foo/:/` and `/foo:/` are identical, apart from the fact that the latter will not show up in directory listings in the root. By taking this a step further, the *subtag separator* comes into play. The directory `/foo/:/bar/` represents everything in the `bar` sub-category of `foo`. A shorter and equivalent form is the path `/foo`bar/`.

When reading paths, it helps to bear in mind the idea of *complete* and *incomplete attributes*. A *complete attribute* contains no *subtag indicators* (`:`) after all of the `/:/` sequences have been resolved to *subtag separators* (`` ` ``). An *incomplete attribute* will end with a *subtag indicator*.

A path can be thought of as a conjunction of *complete attributes*, optionally followed by an *incomplete attribute*. An illustration of this is given in Figure 4.2, with a set visualisation of the complete parts of this path in Figure 4.3.

As many different forms of path can have the same meaning, it must be translated into one single *canonical path*, in order to simplify later processing. The canonical form has been chosen to replace sequences involving the subtag indicator character (`/:/` and `:/`) with the subtag separator wherever possible. This means that the paths `/year:/4/` and `/year/:/4/` will both be canonicalised to `/year`4`. Attributes can then be easily extracted by splitting the path into `/`-delimited segments. In addition, if and only if a canonical path contains the *subkey indicator* character, it contains an incomplete tag. This process is illustrated in Figure 4.4.

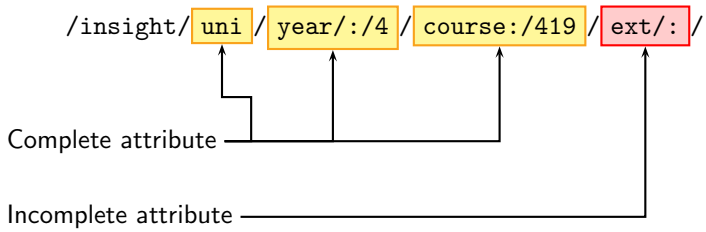
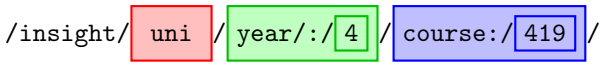
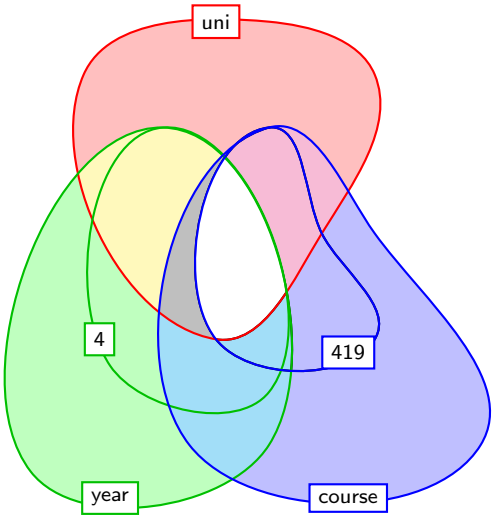


Figure 4.2: Path syntax





Note: The white area in the centre is the result for the path above. The grey area to the left is part of course but not course`419 and so is not included.

Figure 4.3: Set visualisation of the path `/uni/year/:/4/course:/419/`

4.3 Data storage

A number of existing projects use or intend to use SQL as their back-end data storage system. I do not feel this is appropriate, as the types of data I am storing do not translate well to relational schemas. In order to store sub-attributes in a relational database, a table definition such as that given in Figure 4.5 might be used to store tags.

Unfortunately, this leads to queries being inefficient, as the underlying storage is not optimised for the type of searches we are doing. A number of self-joins would be required for even simple queries. This indicates that an SQL database is therefore not a good choice for the back-end storage, as the workarounds to force the data to fit a schema could cause large overheads.

In order to reference files in this file system, each is given an identifier called an *inode* that is unique inside a file system. Due to the short timescales for this project, it is infeasible to create a full on-disk file system with proper block allocation and file storage, and so *Insight* must be a layer above the existing VFS (Virtual File System) in Linux.

To this end, *inodes* are essentially references to files elsewhere on the system rather than references to a collection of data blocks. In *Insight*, these inode numbers should be generated, stored and

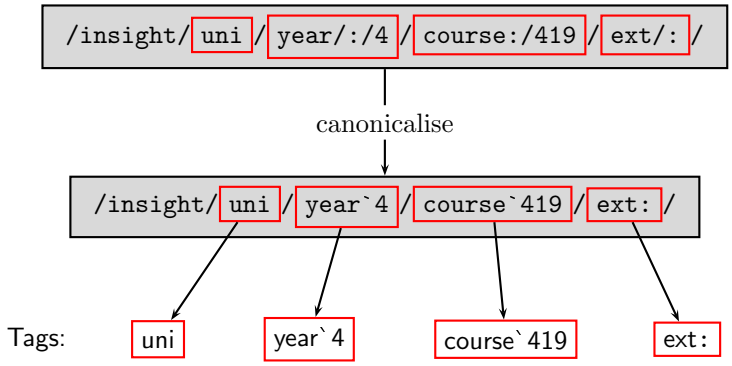


Figure 4.4: Path canonicalisation and tag extraction

Column	Type	Description
id	autoint	Unique ID for a tag
synonym	int/NULL	ID of the canonical tag this is a synonym for, if set
parent	int/NULL	ID of the parent of this tag, if set
name	string	The name of the tag

Figure 4.5: Sample 'tags' table layout

translated to real path names when required.

Data should be stored in a format that allows for fast searches. This data will be primarily stored on-disk, but disk accesses are relatively slow. This can be alleviated to some degree by implementing caching, but initial disk accesses are still necessary. The optimum structure for storing the indexes should minimise the number of times the disk is read. One data structure that is very good at minimising block reads while providing fast searches is a B^+ *tree*.

4.4 B$^+$ tree

A B$^+$ tree is a tree-like data structure that provides a high degree of fanout and stores data at the leaves. All leaves in the tree are at the same distance from the root. This means that very few block reads are required in order to access one of a large number of records.

For example, if the degree of fanout is 125 (i.e. each node can have 125 children) and the average fill factor is 66%, then $(66\% \times 125)^n$ records can be indexed with n levels on average, and a maximum of 125^n records can be indexed. If the tree has four levels, say, then that gives average of 83^4 or 47,458,321 records in the average case and 125^4 or 244,140,625 records in the best case, with only at most five disk reads required (including one for the location of the tree root).

For *Insight*, I will need to use multiple nested trees in order to support structured attributes (e.g. `TV`series` or `uni`year`4` requiring one and two sub-trees, respectively). The general structure of one of these trees is shown in Figure 4.6. The depicted tree contains the keys `A`, `B`, `D`, `E`, and `E` has a pointer to a sub-tree with keys `X`, `Y` and `Z`. Note that the leaves of the tree form a linked list for fast traversal, making range queries easy to program.

This structure provides fast lookup, insertion and deletion. If also gives flexibility with attributes, making them easy to parse and translate to the set of inodes in a data block (which can be seen at the bottom of Figure 4.6).

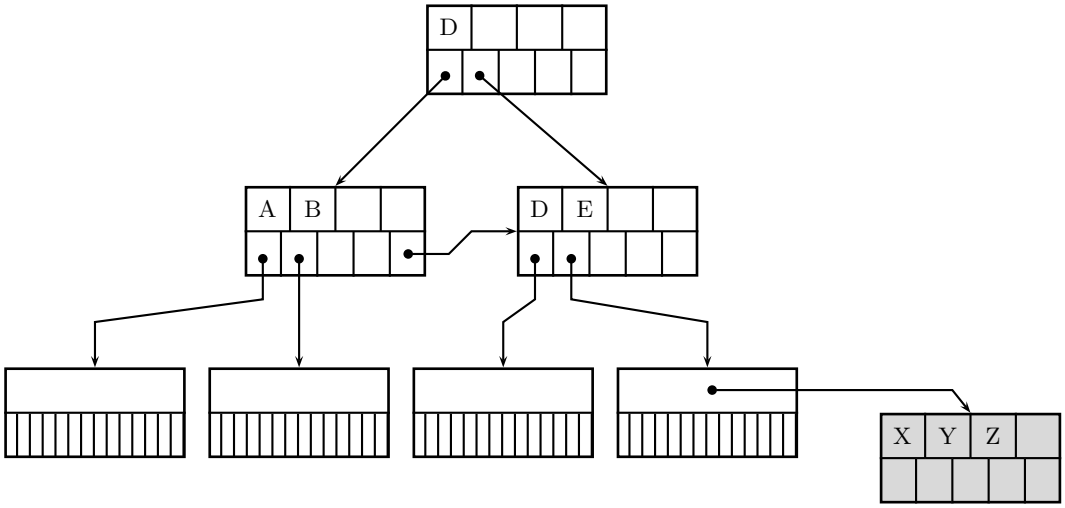


Figure 4.6: Overview of a generic *Insight* B^+ tree

4.4.1 Data storage requirements

The data that needs to be stored in each tree node is minimal:

- A boolean to indicate whether the node is a leaf or not
- The number of keys stored in the node
- n keys and $n + 1$ pointers

Similarly, the requirements for a data block include:

- The number of inodes associated with the key
- Flags and other details about the key
- A pointer to the sub-tree for this key, if applicable
- The set of inodes associated with this key

Finally, some meta-information must be stored:

- A file format version identifier
- A pointer to the root of the main tree
- A list of inodes that have no attributes assigned to them

4.5 Query engine

The final major part of the design for the *Insight* file system involves the query engine. Paths have been reduced to set intersection, so building a query is a simple matter of breaking the provided path into tags as shown in Figure 4.4. The contents of listings can then be determined using the following rules:

1. If the path contains an incomplete attribute:
   - There are no files listed.
   - If the incomplete tag (without indicator character) is represented as T, the list of directories will be $\{x \mid x \in subtags(T)\}$, discarding the other parts of the query.
2. If there are no incomplete attributes in the path:
   - Files listed can be expressed in set notation as the files satisfying $\bigcap_n P_n$ where P_n is the nth path element. Note that $\forall x \in subtags(P_n) \quad P_n \subseteq x$, i.e. an attribute also contains all of the files in its subattributes.
   - Directories listed are all of the top-level attributes, along with the special `:` directory if and only if the path has at least one element and the last element of the path has one or more subattributes.

These two simple rules give some usable directory listings, but provide some awkwardness when browsing. For example, tags could be redundantly repeated in a path, (e.g. `/insight/TV/TV`).

In addition to this, if we list the contents of a subattribute directory then we would get the result in Figure 4.7(a). It may be more desirable to have the directory listing in Figure 4.7(b) instead, as the `TV` attribute would provide no useful filtering. Its subattributes may still be desirable, however, so they should be listed.

```
$ cd /insight/TV/:/episode/
$ ls -F
:/  Film/  mime/  TV/  type/
$
```

(a)

```
$ cd /insight/TV/:/episode/
$ ls -F
:/     mime/       TV`series/  type/
Film/  TV`season/  TV`title/
$
```

(b)

Figure 4.7: Two possible directory listings for `/insight/TV/:/episode/`

The query code supports disjunction and negation as well as just conjunction, even if this cannot be expressed neatly through the legacy path syntax. The query system is also designed to be flexible and extensible, to allow addition of range queries (for example).

4.6 Limitations

As a result of the time constraints on this project, it has not been possible to provide a full range of functionality through the file system interface. It was decided that the most natural subset of the query operations to use is conjunction.

Adding further directories to a path is a natural way of narrowing down the search for a file. For example, some people may use the partial hierarchy in Figure 4.8 to organise their data in an existing file system. It can be seen that `419` is a subcategory of `year4` which is itself a subcategory of `uni`.

Adding negation and disjunction to the syntax is also possible, but then paths become awkward as the introduction of grouping and more symbols become necessary. This also results in more characters being reserved for special use. For example: `/insight/(/tag1/|/tag2/)/!/tag3/` could represent $(tag1 \cup tag2) \setminus tag3$ but it is inelegant and would require the special `(`, `)`, `!` and `|` directories to be shown in every listing, in order to make them easily browseable with existing tools.

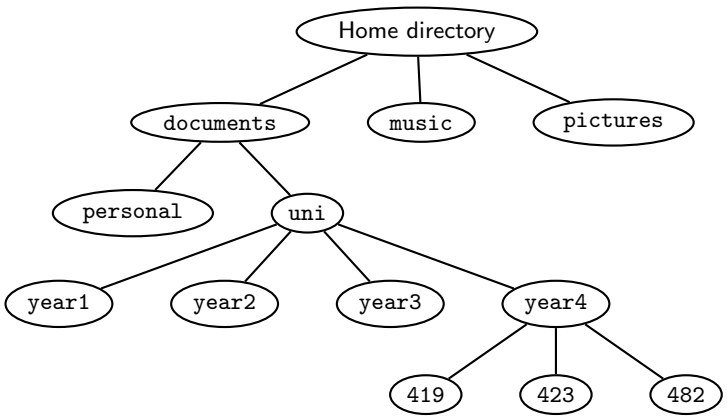


Figure 4.8: A sample (partial) directory tree

Chapter 5

Implementation

> If you limit your choices only to what seems possible or reasonable, you disconnect yourself from what you truly want, and all that is left is a compromise.
>
> *Robert Fritz*

The next stage of the project (once the design had been sketched out) was the choice of language. The most natural language for file system creation to maximise speed is C, in my opinion. It is very well-known and well-used, and the vast majority of the Linux kernel is C-based.

The remainder of this chapter details the implementation of the *Insight* file system.

5.1 B^+ tree prototype

In order to test the concept of the nested B^+ tree system that I had planned to use in *Insight*, it was necessary to build the B^+ tree code separately with a dedicated test harness. This approach allowed the tree to be tested independently of the file system, and any problems to be isolated and fixed quickly.

A full range of tests were run on the tree code, to test normal node insertion/deletion/lookup, subtree creation/deletion, and subtree node insertion/deletion/lookup. This covered the main functions that would later be used in the full file system, and ensured that they would work under a variety of conditions.

5.1.1 On-disk format

This prototype exposed some interesting issues with regard to the storage format. One such issue was verifying that internal addresses are consistent; i.e. if we are expecting the address of a data block, we should be able to verify that the block we actually read matches that format. This was accomplished by the use of *magic numbers*, or special values at a known location in a block that serve to identify its type. These values were chosen to approximate to words when viewed as hexadecimal, for easy identification. For example, `0xDA7AB10C` could be read as "data block".

These magic numbers can be seen in the first part of each block in the on-disk format represented in Figure 5.1. Note that the diagram is only partially to scale, and refers to some constants that are dynamically determined based on the chosen block size. Given a block size of 512 bytes, they will have the values shown in Table 5.1.

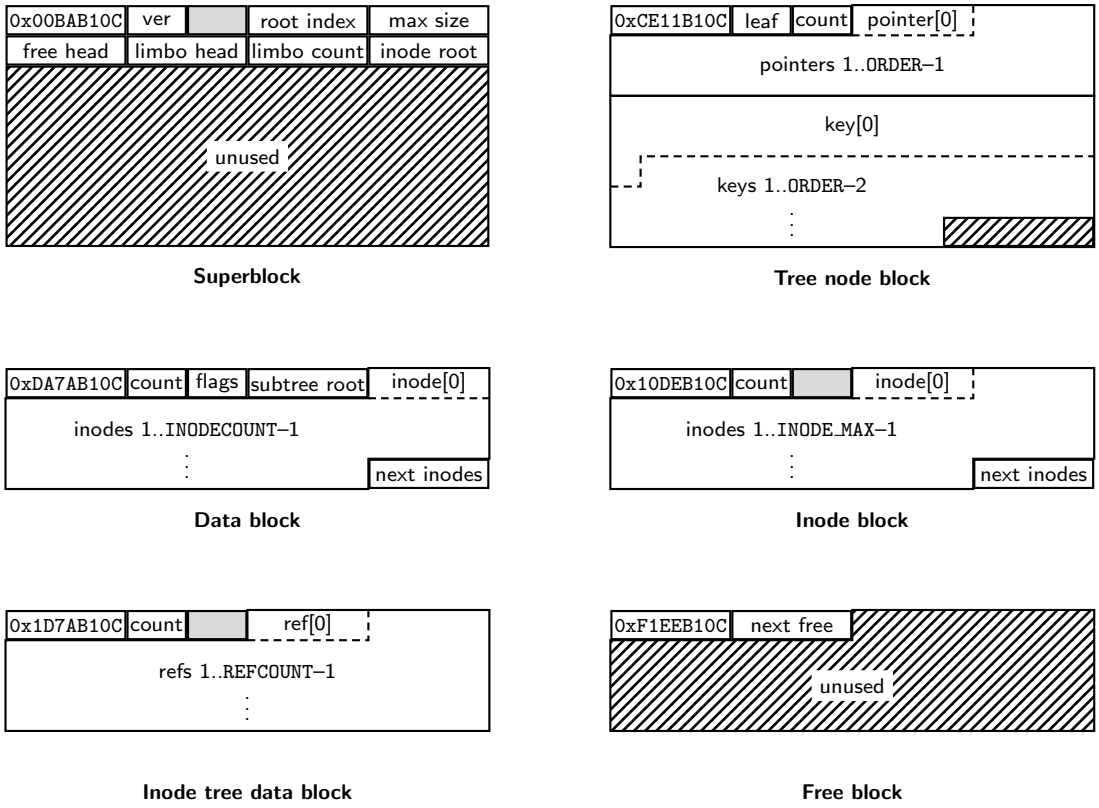


Figure 5.1: On-disk format

Constant	Value	Description
`ORDER`	14	The number of pointers/keys in a tree node
`KEYSIZE`	33	The maximum length of a key/attribute name, including the terminating null character
`INODECOUNT`	124	The number of inodes that appear in a data block
`INODE_MAX`	125	The number of inodes that appear in an inode block
`REFCOUNT`	125	The number of reference pointers that can appear in an inode tree data block

Table 5.1: Values of constants assuming block size of 512

5.1.2 Multiple trees

As can be seen from the on-disk format in Figure 5.1, data blocks can contain a pointer to the root of a sub-tree which contains further attributes. The superblock also contains a pointer to the inode tree, or reverse tree.

The reverse tree maps inodes to attributes (rather than the other way around). This is useful for two reasons: finding which attributes have been applied to a file, and (more importantly) detecting when all attributes have been removed from a file so it can be placed into *limbo*. Once a file is removed from limbo, it is removed from the file system completely. This is to prevent accidental deletion.

5.2 Query trees

Query trees are essentially simplified versions of Abstract Syntax Trees (ASTs) that are often used by parsers. They represent the syntax of a query in an easily-traversable format.

5.2.1 Query tree nodes

A query tree is made of one or more nodes, arranged hierarchically from a root node. Each node has a type code that identifies its meaning and which data fields in the node are valid. These types are listed in Table 5.2, along with their meanings and valid data fields. Note: the next fields are pointers to other query tree nodes, whereas tag is a string tag name, and inode is an inode number.

	Valid data fields				
Type code	**tag**	**inode**	**next[0]**	**next[1]**	**Description**
`QUERY_IS_ANY`	N	N	N	N	Matches any inodes in limbo
`QUERY_IS`	Y	N	N	N	Matches any inodes in the given attribute or its sub-attributes
`QUERY_IS_INODE`	N	Y	N	N	Matches only the specified inode
`QUERY_NOT`	N	N	Y	N	Negates the subtree results
`QUERY_AND`	N	N	Y	Y	Performs set intersection on the results of the two subtrees
`QUERY_OR`	N	N	Y	Y	Performs set union on the results of the two subtrees

Table 5.2: Query node types

5.2.2 Query extraction

A query tree t is generated from an input path p using the following algorithm:

1. If p is an empty path, then the query tree is trivially a single `QUERY_IS_ANY` node and we can return success.
2. Split p into substrings $p_0 \cdots p_n$ by using the directory separator character (`/`).
3. For each substring p_i:
   (a) If p_i is a valid attribute, then wrap the attribute name in a `QUERY_IS` query tree node.

(b) If $hash(p_i)$ is a valid inode, then wrap the inode number in a `QUERY_IS_INODE` query tree node and go to step 4.

(c) Otherwise, the component of the path is invalid and we can abort processing and return an error code.

(d) If a partial query tree does not exist, make the new query node the root of the partial query tree.

(e) Otherwise, wrap the new node and existing tree root in a `QUERY_AND` query tree node and make it the tree root.

4. After the substrings have been processed and a query tree is available, do a quick sanity check:

   - If the tree does not contain an `QUERY_IS_INODE` tree node, then there is nothing to check.
   - Otherwise, if the query returns zero results then the path does not exist. This is the case where we have a valid file name but it is filtered out by one or more of the attributes in the path.

A full query tree is obtained by this process, which can then be used to retrieve a list of inodes for further processing. Details of the tree construction for the path `/uni/year`4/course`419` are given in Figure 5.2, and further examples of query trees for various paths are shown in Figure 5.3.

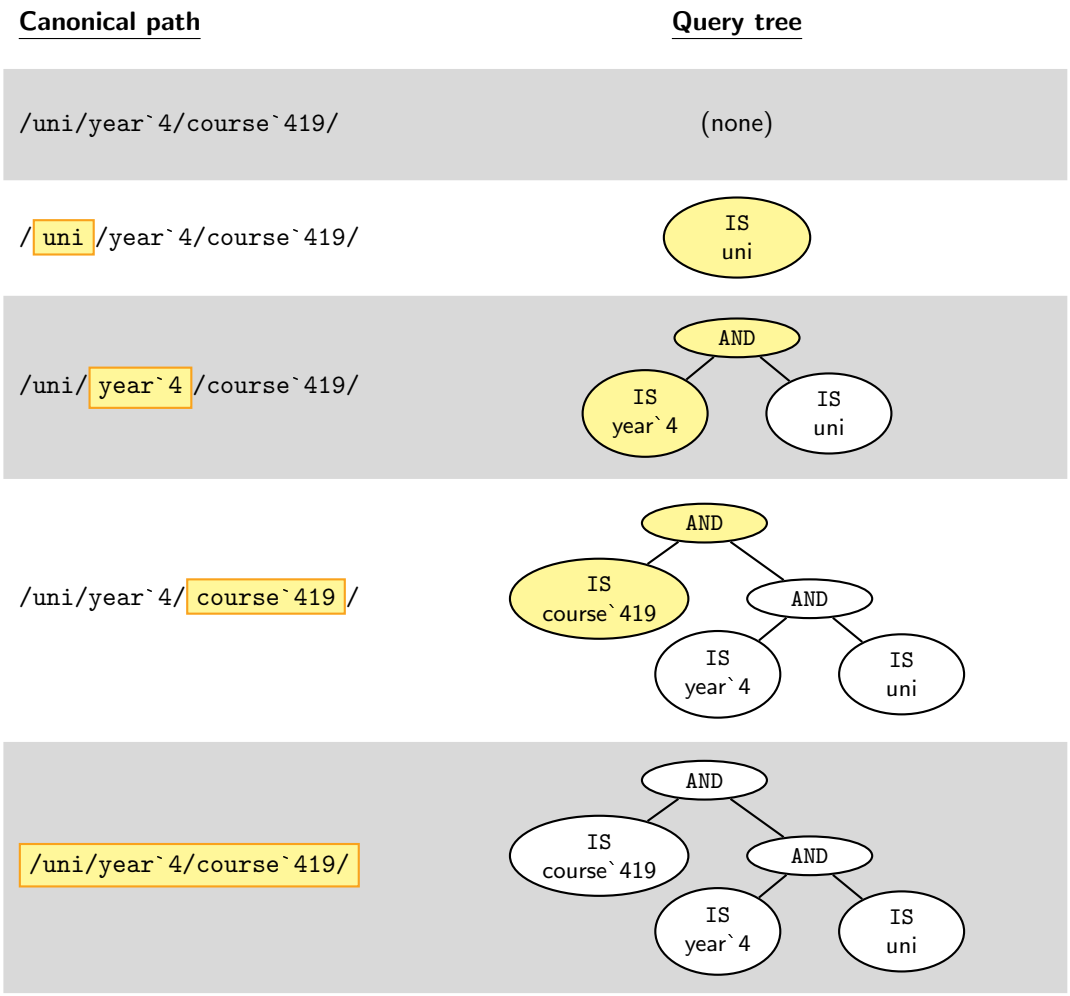
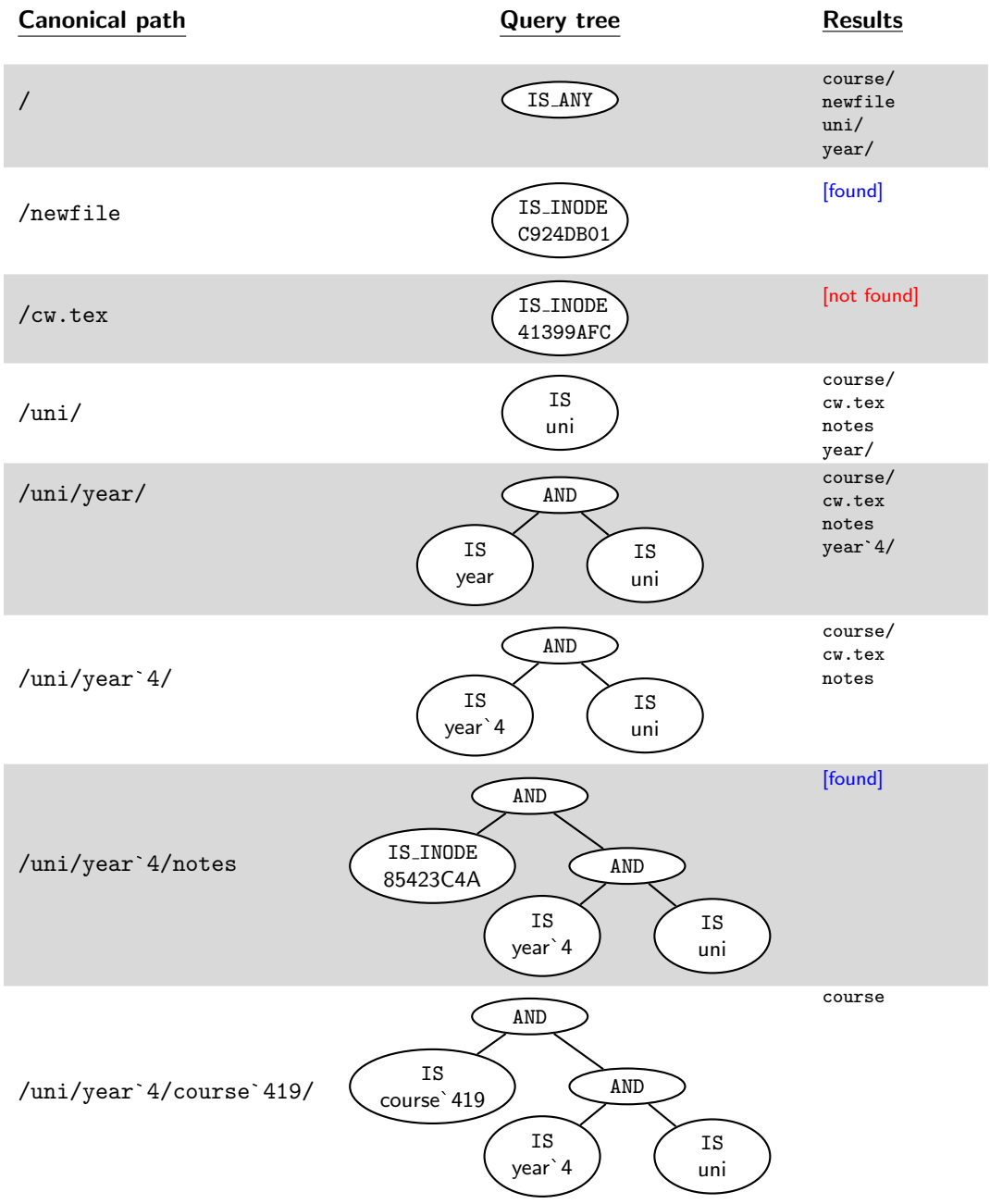


Figure 5.2: Query tree construction for `/uni/year`4/course`419`



Assume that `uni`, `year`, `year`4`, `course`, `course`419` are valid attributes, that `cw.tex` is a file tagged with `uni`, `year`4` and `course`419`, `notes` is tagged with `uni` and `year`4`, and that `newfile` is untagged. The hashes of `cw.tex`, `newfile` and `notes` are `41399AFC`, `C924DB01` and `85423C4A` respectively.

Figure 5.3: Examples of query trees built from paths

5.3 FUSE implementation

5.3.1 Introduction to FUSE

FUSE (File system in user space) is an abstraction layer that allows a fully functional file system to be written in user space, and allows regular users to mount file systems. It does not require recompilation of the kernel, and provides a simple and efficient API for rapid development.

The user space program communicates with the kernel API via a special device node (`/dev/fuse`) that is opened by each user space file system. The communication between programs (e.g. `ls`), the kernel, and a FUSE file system is shown in Figure 5.4.

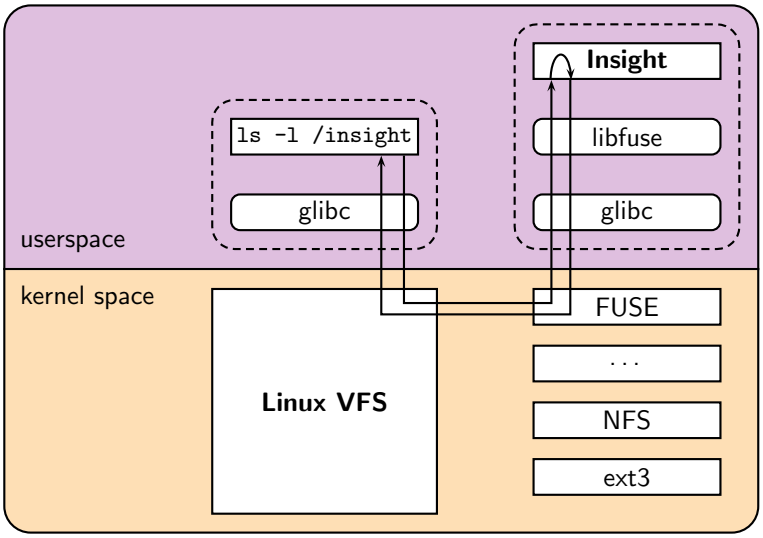


Figure 5.4: The path of a file system request via FUSE[24]

FUSE provides such a simple and straightforward API that a "hello world" example file system can be written in under 100 lines, yet it is powerful enough to cater to all sorts of needs and file system experiments[1].

5.3.2 Inode assignment

Inodes are numeric identifiers that should be unique throughout a file system (although no guarantees are made between file systems). If *Insight* was able to allocate blocks on disk in the same manner as a traditional file system, the inode would identify the range of blocks belonging to a file. As this is not possible, inodes must be subverted for another use.

Insight stores files by creating a symbolic link (symlink) to the target file when importing it, naming the link according to the calculated inode value for the basename of the path, i.e. the part of the path following the last '`/`'.

This inode value is calculated using a hashing function. Because there was no time to provide adequate research and implementation of a suitable hashing function, an existing implementation was used. The hashing algorithm is a keyed 32-bit hash function using TEA in a Davis-Meyer function[21]. The implementation used was written by Jeremy Fitzhardinge as part of the *reiserfs* project and released under a free software licence.

[1] `http://fuse.sourceforge.net/wiki/index.php/FileSystems`

5.3.3 File system operations

All of the file system operations supported by FUSE are listed in Table 5.3, along with some brief descriptions. A number of the file operations can be passed through to the 'real' underlying file system, such as `chmod`, `chown`, `utimens`, `open`, `read` and `write`. These functions each receive the path to the file the operations should affect. In order to translate this to a real file name, they must:

1. Obtain the canonical form of the input path
2. Pass the path to the query tree generator to check its validity
3. Ensure that exactly one inode is returned by the query
4. Translate the inode into a path to a symbolic link in the link repository
5. Read the target of the symbolic link to get the destination file name

The other two major file system operations are `getattr` and `readdir`. The first of these may be called a large number of times during normal operation, including at least once per file in a directory listing and also when testing file existence. Its purpose is to return useful information about a file or directory, often in response to a `stat()` system call, and the speed of the `getattr` implementation can be a major contributing factor to file system performance.

The `readdir` function is responsible for calculating and populating directory listings. It may also return information about the entities the directory contains such as their inode numbers. More information about how directory listings are calculated is given in the following section.

5.3.4 Directory listing generation

Creating a directory listing is a relatively complex process, and follows this algorithm:

1. The query tree should be computed for the path, and any appropriate errors returned if the query tree creation fails.
2. The two special entries '`.`' and '`..`' are added to the directory listing to represent the current directory and parent directory, respectively.
3. Following this, the query tree is scanned for incomplete attributes.
   (a) If an incomplete attribute is found, then any inode/file processing is skipped.
   (b) Otherwise, the set of inodes returned by the query is calculated, and each inode is turned back into a filename using a similar process to that described in Section 5.3.3. Each file name is then added to the directory listing.
4. The next step is to determine which directories should be listed.
   - If the current path is not the root path, there are no incomplete attributes and the last path component has one or more sub-attributes, the special sub-attribute directory '`:`' is added to the listing.
   - If there is an incomplete attribute in the path, all of the sub-attributes of the incomplete attribute are added to the listing, provided they do not already appear in the path.
   - Otherwise, all of the top-level attributes are added to the listing, provided they do not already appear in the path. If a sub-attribute of a top-level attribute appears in the path, then its sibling sub-attributes are listed using canonical syntax as long as they would produce useful results (see lines 19–34 of Figure 5.5 for an example).

Examples of directory listings are shown in Figure 5.5.

Function name	Description
`access`*	Check file access permissions
`chmod`‡	Change the permission bits of a file
`chown`‡	Change the owner and group of a file
`create`*	Create and open a file
`destroy`*	Clean up file system
`fgetattr`*	Get attributes from an open file
`flush`*	Flush cached data
`fsync`*	Synchronise file contents
`fsyncdir`*	Synchronise directory contents
`ftruncate`*	Change size of an open file
`getattr`	Get file attributes
`getxattr`†	Get extended attributes
`init`*	Initialise file system
`link`	Create a hard link to a file
`listxattr`†	List extended attributes
`lock`*	Perform POSIX file lock operation
`mkdir`	Create a directory
`mknod`	Create a non-directory, non-symlink node
`open`‡	Open a file
`opendir`*	Open directory
`read`‡	Read data from an open file
`readdir`	Read directory contents
`readlink`	Read symbolic link target
`release`	Release an open file
`releasedir`*	Release directory
`removexattr`†	Remove extended attributes
`rename`	Rename a file
`rmdir`	Remove a directory
`setxattr`†	Set extended attributes
`statfs`‡	Get file system statistics
`symlink`	Create a symbolic link
`truncate`‡	Change the size of a file
`unlink`	Remove a file
`utimens`*	Change file access/modification times
`write`‡	Write data to an open file

* Optional function: not required for standard file system operation
† Functions only required if extended attributes are enabled
‡ Functions implemented as direct pass-through

Table 5.3: FUSE file system operations

```
$ cd /insight/
$ ls -F
Film/  mime/  TV/  type/  untagged-file.pdf

$ cd TV/ ; pwd
/insight/TV

$ ls -F
:/     TV Show - 1x01.avi  TV Show - 1x04.avi
Film/  TV Show - 1x02.avi  type/
mime/  TV Show - 1x03.avi

$ ls -F :/
episode/  season/  series/  title/

$ cd :/episode/ ; pwd
/insight/TV/:/episode

$ ls -F
:/          TV`series/          TV Show - 1x03.avi
Film/       TV`title/           TV Show - 1x04.avi
mime/       TV Show - 1x01.avi  type/
TV`season/  TV Show - 1x02.avi

$ ls -F :/
1/  2/  3/  4/

$ cd :/1/ ; pwd
/insight/TV/:/episode/:/1

$ ls -F
Film/       TV`series/          type/
mime/       TV`title/
TV`season/  TV Show - 1x01.avi
```

Figure 5.5: Examples of directory listings

Chapter 6

Evaluation

> I don't know the key to success, but the key to failure is trying to please everybody.
>
> *Bill Cosby*

The project started with the following primary aims:

1. Create a semantic file system for Linux that allows users to organise files in a more intuitive manner.
2. Allow users to create a dynamic categorisation for their files, related to their meaning (semantics) rather than just placing them in a single category/sub-category.
3. Provide an interface that is backwards-compatible with existing programs that rely on path-based file systems.

There were also a number of secondary aims:

1. Create a file import program that would automatically assign some attributes to the file.
2. Create a demonstration program or programs to illustrate the flexibility of a semantic file system.
3. Develop a plugin for an existing program that would enhance its usability by using this file system.

This chapter considers the degree to which the aims were achieved (Section 6.1) and also notes some limitations of the software (Section 6.2).

6.1 Meeting the aims

This project worked towards two types of target, with the secondary aims relying on the achievement of the primary aims.

6.1.1 Primary aims

The first and most important aim of the work was the creation of a semantic file system that integrates with the standard Linux file system interfaces, and which can be used in the same manner as existing file systems. This target has been met, and *Insight* is the proof-of-concept implementation.

The second primary aim was to provide the ability for users to create dynamic attributes that they can apply to files. Again, *Insight* provides this functionality, with a theoretically unlimited organisational depth. In addition, its structured attributes can be represented as either simple keywords, structured keywords, key/value pairs, or structured key/value pairs.

The third aim is closely linked to the first; *Insight* creates an interface to tagged files that can be used by all existing programs with no modifications required. Creation, deletion, assignment and removal of (sub-)attributes are all represented by simple file system operations (directory creation/deletion, copying files into a directory and deleting a file from a directory). Queries are handled by simply browsing paths with a well-defined structure.

6.1.2 Secondary aims

As the primary aims have been achieved, some thought can be given to the secondary aims.

One of these secondary targets was the creation of an import program that would assign some attributes to the file automatically based on information about it. A rudimentary import program has been created that can initialise the following attributes on file import, as applicable:

- All files
  - File extension (`_ext`)
  - File owner (`_owner`)
  - File owner group (`_group`)
  - Symbolic file permissions (`_perm`)
  - File modified date (`_date`year`, `_date`month`, `_date`day`)
  - MIME type (`mime`major`minor`, e.g. `mime`text`plain` for text/plain documents)
- MP3 files
  - Artist (`music`artist`)
  - Track (`music`track`)
  - Title (`music`title`)
  - Album (`music`album`)
  - Genre (`music`genre`)

Note: a leading underscore in a top-level attribute name indicates that the attribute is hardcoded into *Insight* and will be automatically updated when the file metadata changes. Attribute names starting with an underscore cannot be created by the user at the top level due to this restriction.

Another secondary aim was the creation of a demonstration program to illustrate some of the use of a semantic file system. The chosen subject for a demo program was tagging people in photographs (and applying general tags) and storing that information within *Insight* so it can be used elsewhere. This will be demonstrated during the presentation.

The final secondary aim was to create a plugin for an existing program that would enhance its functionality by using *Insight*. Unfortunately this could not completed due to lack of time, but I feel that this task was the most open-ended and least well-defined.

6.2 Limitations of this solution

There are a number of limitations with this implementation, as it stands, but these can be fixed given sufficient time and effort. These limits generally arose due to design choices that were made or through lack of time for full implementation.

- Only one file with a given file name can exist in the file system at any one time. This is due to the restrictions imposed by the inode system, as well as the potential for confusion. A full file system that controlled block allocation and proper inode assignment would not have this issue, although it may choose to ensure that file names are not repeated for clarity.
- There may also be issues with hash collisions, i.e. when two different file names hash to the same inode value. In this case, the user will be informed that the file already exists.
- The path-based query interface is currently limited to solely conjunctive queries, although the internal code can handle negation and disjunction as well. This limits the power of the user to perform searches. This can be helped by introducing a syntax for the additional query operators, or by introducing an alternative interface to the query engine, perhaps via a GUI.
- It is perhaps unintuitive to have to browse into another subdirectory in order to view the sub-attributes of a given attribute. In hindsight, perhaps it would have been best to organise directories the other way round.
- Despite the use of the FUSE library, this code is not truly portable between operating systems as it has been written for and solely tested on Linux. It makes a number of assumptions which may not be true on other platforms, and would require some work to be platform-agnostic. The core should be independent enough that only minimal changes would be required.
- The automatic assignment and update of file attributes is hardcoded to just a few types and attributes. A plugin system to automate attribute assignment would be a very flexible solution.
- Although the FUSE library supports multi-threaded and re-entrant execution, there is no support in *Insight* for the issues that multi-threading can cause. It currently forces FUSE to use a single-threaded mode instead.

Chapter 7

Conclusion

> I may not have gone where I intended to go, but I think I have ended up where I needed to be.
>
> *Douglas Adams*

7.1 Project conclusion

Designing and creating a file system is a complex process. It may start from a simple idea, but it can rapidly branch out and may easily grow beyond the bounds of feasibility. Keeping such a project bounded in size and complexity is difficult, particularly when it is a project in which one has a deep personal interest.

Abstraction layers like FUSE make it easy to build file systems, but building a robust file system well is another matter entirely. Many factors have to be taken into account, and a file system is meant to be extremely stable. Time restrictions mean that rigorous robustness tests could not be carried out, but this project is only intended as a proof-of-concept file system which may be used as a base for further development.

As a proof of concept, I believe that this project has shown the potential of a semantic file system. It is certainly something that warrants further study and work, and I fully intend to continue to develop this prototype into a full and stable file system that may be used on different operating systems. There is a lot of scope for future work, and a few selected topics have been briefly covered in the following section.

7.2 Future work

7.2.1 Extending the syntax

The query syntax should be extended to match the capabilities of the internal query engine. A syntax to represent disjunction (OR), negation (NOT) and bracketing should be created, in such a way as to be as compatible with existing tools as possible.

Possibilities for the syntax include using the raw '`|`', '`!`', '`(`' and '`)`' characters inside a path:

```
/(tag1|tag2)/!tag3
```

Alternatively, creating additional special directories has the advantage that queries can be created by using existing file browser utilities, but the disadvantage of more verbose syntax:

```
/(/tag1/|/tag2/)/!/tag3
```

7.2.2 GUI integration

Another interesting area of future work is the development of replacement open/save dialogs for GUI environments which are better-suited to semantic file systems[9]. These could provide the user with a list of frequently-used attributes, or perhaps a list of automatically-selected attributes based on known information about the file being saved. This would make it much easier for users to manually tag files when they are saved, as well as providing a more powerful query interface than the legacy path-based system.

7.2.3 Extended attribute integration

At the moment, the extended attribute functions provided by *Insight* are just wrappers for the underlying file system functions. Extended attributes as they currently exist in Linux provide name/value pairs in one of four namespaces: `security`, `system`, `trusted`, and `user`, for SElinux, system-specific purposes (e.g. ACLs), privileged applications and user-defined purposes respectively.

Providing an additional namespace (`insight`) could allow applications to discover all *Insight* attributes given to a file, and then to add/modify/remove these attributes in a very easy manner, without resorting to standard file system operations.

7.2.4 Additional query operators

In the current incarnation of *Insight*, the query operators are restricted to simple equality. In order to make queries more powerful, range operators ($<, >, \leq, \geq, between$) could also be introduced. This will allow for a much larger spectrum of expressions. In addition, a regular expression matching operator could be extremely useful, although it could slow queries down dramatically.

7.2.5 Dynamic query expressions

In order to make queries even more powerful, it might be interesting to extend the query syntax still further to allow for dynamic expressions and function evaluation. This could pave the way for directory contents changing based on the time of day, for example, or for showing all files modified in the last ten minutes.

This would again have the tradeoff of speed against functionality, so perhaps query engines with differing abilities could be dynamically selected at file system mount time.

7.2.6 File system visualisation tools

A popular method of visualising tag usage in so-called "Web 2.0" sites is via a *tag cloud*, or a collection of differently-sized tag names. The font size of the tag is then proportional to the number of items tagged with it. It may prove interesting to produce similar data visualisation tools, so that users can see their most-used tags at a glance.

7.2.7 Data types

Assigning a data type to the subtree of an attribute would change the way its subtree is ordered, and could be advantageous. At the moment, all attribute and value types are coerced to strings, but in many cases integers or even floating-point numbers would be desirable.

As things stand, numeric types will be incorrectly ordered when rendered as strings, as they would be sorted lexicographically. Consequently, range queries on numeric types would either be highly inefficient or not work at all.

There is also the possibility of `enum`-style types, e.g. days of the week or months of the year.

7.2.8 Directory schemas

Directory schemas are a way of transforming the attributes held in *Insight* into easily-browseable paths. For example, if TV programmes are organised using the following attributes:

TV` series: The TV series name (e.g. Dexter, Chuck, "Trick or Treat")

TV` season: The season number (e.g. 1, 2, 3, 4, ...)

TV` episode: The episode number (e.g. 1, 2, 3, 4, ...)

TV` title: The episode title (e.g. "Treat – Confidence", "Chuck versus the Intersect")

then it can be very awkward to actually find the files. It would be much easier with known structures like this for the directory hierarchy to follow the tags in a well-defined way. This can be accomplished by using a *directory schema*.

Directory schemas would be stored with the other parts required for the file system, namely the tree store and repository directory, probably in an optional file named `schemas`. The following text is a suggestion as to how it might be implemented.

The `schemas` file will be written in plain text, and should begin with a file format version identifier. It will also use the convention that `#` indicates a comment until the end of the line.

Each schema specification will consist of a `DIRECTORY` specifier that gives the name of the schema. For example, `DIRECTORY /TV` would make the schema accessible under the `/_schemas/TV` directory inside the *Insight* mount point.

Following the directory specifier is a `FILTER` line that provides a query string to use to select files that will appear in the schema directory. The syntax for this should be similar to standard C-like logical operators, e.g. `FILTER (${music})` or `FILTER (${music`waltz} && ${music`piano})`, which would match files tagged with music or files tagged with music`waltz and music`piano, respectively.

Finally, a `SCHEMA` directive specifies the output format, e.g.:

```
SCHEMA /${music`artist}/${music`album}/$[${music`track} - ]${music`title}.${ext}
```

This produces paths that look like (for example):

```
/Casa Musica/Ballroom Classics 4/04 - Sprint.mp3
```

And these paths are automatically produced for every file that matches the filter, making the files much easier to access. There are some syntactic rules to work out, such as what to do when a tag isn't specified for a file that matches the filter, but this should provide a good starting point.

Bibliography

[1] Andrews, Jeremy, 'Linux: Why Reiser4 is not in the kernel', Jul 2006, accessed on 2008-01-04.
`http://kerneltrap.org/node/6844`

[2] Andrews, Jeremy, 'Linux: Reiser4's future', Apr 2007, accessed on 2008-01-04.
`http://kerneltrap.org/node/8102`

[3] Apple, 'Apple Spotlight', 2004, accessed on 2008-01-09.
`http://www.apple.com/macosx/features/spotlight/`

[4] Braun, Matthias, 'Call for a metadata-enabled filesystem', Accessed on 2007-12-18.
`http://www.stud.uni-karlsruhe.de/~uxsm/MetaData-Filesystem.html`

[5] Clark, Quentin, 'Update to the [WinFS] update', Jun 2006, accessed on 2007-12-22.
`http://blogs.msdn.com/winfs/archive/2006/06/26/648075.aspx`

[6] Clark, Quentin, 'WinFS update', Jun 2006, accessed on 2007-12-22.
`http://blogs.msdn.com/winfs/archive/2006/06/23/644706.aspx`

[7] Giampaolo, Dominic, *Practical File System Design with the Be File System* (San Fransisco, California: Morgan Kaufmann Publishers, 1999).

[8] Gifford, David K., Jouvelot, Pierre, Sheldon, Mark A., and James W. O'Toole, Jr., 'Semantic file systems', *SIGOPS Oper. Syst. Rev.*, vol. 25, no. 5: (1991) pp. 16–25.

[9] Gorter, Onne, *Database File System*, Master's project, University of Twente, Aug 2004, accessed on 2007-11-22.
`http://tech.inhelsinki.nl/dbfs/`

[10] Hunter, David, 'Say goodbye to WinFS', Jun 2006, accessed on 2007-12-22.
`http://www.hunterstrat.com/news/2006/06/24/say-goodbye-to-winfs/`

[11] Kiselyov, Oleg, 'A dream of an ultimate OS', in *MacHack'95* (MacHack'95, 1995), accessed on 2007-12-18.
`http://okmij.org/ftp/DreamOS.html`

[12] Miller, Charles, 'Filesystem sacrilege', Jan 2003, accessed on 2007-12-18.
`http://fishbowl.pastiche.org/2003/01/19/filesystem_sacrilege`

[13] Nickell, Seth, 'Design Fu blog archive', Accessed on 2007-12-22.
`http://www.gnome.org/~seth/blog`

[14] Nickell, Seth, 'GNOME Storage', Accessed on 2007-12-22.
`http://www.gnome.org/~seth/storage/index.html`

[15] Nickell, Seth, 'GNOME Storage Subversion repository', Accessed on 2007-12-22.
`http://svn-archive.gnome.org/viewvc/storage?view=revision`

[16] Nielsen, Jakob, 'The death of file systems', Feb 1996, accessed on 2007-11-12.
http://www.useit.com/papers/filedeath.html

[17] Reiser, Hans, 'ReiserFS version 4', 2004, accessed on 2007-06-11.
http://www.namesys.com/v4/v4.html

[18] Reiser, Hans T., 'Future vision whitepaper', 1984, revised 1993, accessed on 2007-06-11.
http://www.namesys.com/whitepaper.html

[19] Rooney, Paula, 'WinFS still in the works despite missing Vista', *Channelweb Network*, accessed on 2007-12-22.
http://www.crn.com/software/196600671

[20] Sackman, Matthew, Russell, Francis, Richards, Sam, and Osborne, Will, *MWFS*, Third year group project, Imperial College London, Jan 2005.

[21] Schneier, Bruce, *Applied Cryptography*, 2nd ed (New York: John Wiley & Sons, Inc., 1996).

[22] Silberschatz, Abraham, Galvin, Peter Baer, and Gagne, Greg, *Operating System Concepts*, 6th ed (New York: John Wiley & Sons, Inc., 2003).

[23] Soules, C. A. N. and Ganger, G. R., 'Why can't I find my files?', in *Workshop on Hot Topics in Operating Systems (HotOS)* (2003).

[24] Szeredi, Miklos, 'FUSE: File system in USEr space', Accessed on 2008-06-16.
http://fuse.sourceforge.net/

[25] Thurrott, Paul, 'Windows Storage Foundation (WinFS) preview', Aug 2005, accessed on 2007-12-22.
http://www.winsupersite.com/showcase/winfs_preview.asp

[26] Tx0, 'Tagsistant – what is a semantic file system?', Jul 2007, accessed on 2007-12-18.
http://home.gna.org/tagfs/index.shtml

[27] Udell, Jon, 'Where is WinFS now?', May 2008, accessed on 2008-06-16.
http://perspectives.on10.net/blogs/jonudell/Where-is-WinFS-now/

[28] Virtual Exhibitions in Informatics, 'Beginnings of the UNIX file system', Jul 2007, accessed on 2007-11-26.
http://cs-exhibitions.uni-klu.ac.at/index.php?id=216

[29] W3C, 'Resource description framework (RDF) model and syntax specification', Feb 1999.
http://www.w3.org/TR/REC-rdf-syntax/

[30] White, Daniel, 'Towards a single folder filesystem', Accessed on 2007-12-18.
http://www.skytopia.com/project/articles/filesystem.html

[31] Xu, Z., Karlsson, M., Tang, C., and Karamanolis, C., 'Towards a semantic-aware file store', in *Workshop on Hot Topics in Operating Systems (HotOS)*, pp. 145–150 (2003).

Last updated: June 18, 2008.