Insight: A Semantic File System

Outsourcing Report

January 11th, 2008

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

Supervisor: Dr Peter McBrien, pjm@doc.ic.ac.uk

Abstract

For over 30 years, hierarchical file systems have enforced a certain 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 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 even deeper 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 from every program, offering a consistent method of locating data that is backwards-compatible with the existing hierarchical system.

Contents

Part I

Background

Chapter 1

History

Those who cannot remember the past are condemned to repeat it.

George Santayana

1.1 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 and clarity.

It is also worth bearing in mind that the terms *semantic file system* and *database file system* are often used interchangeably, although they are not necessarily identical. Many semantic file systems are built upon databases, however, and so the term may apply.

1.2 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* were still being written as late as 1985, including the original Mac file system and early versions of the CP/M file system.

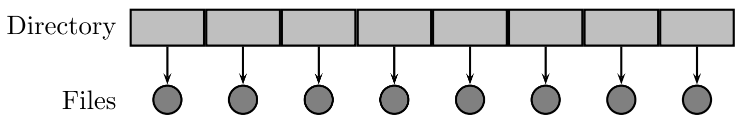

Figure 1.1: Flat file system [23]

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 [27]. This *hierarchical file system* 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.

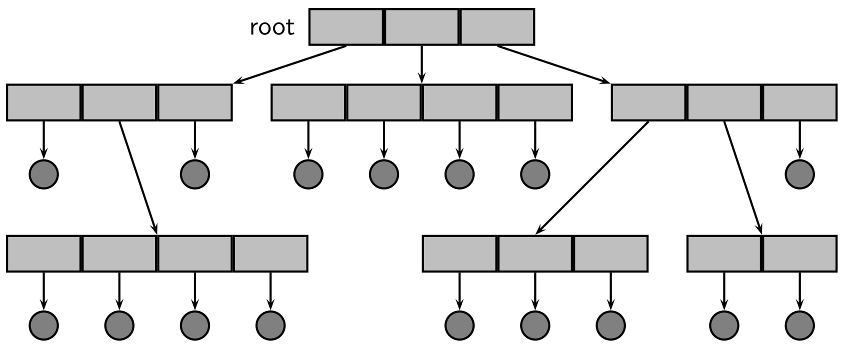

Figure 1.2: Hierarchical file system [23]

1.3 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 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 [19] 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.

1.4 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 [2, 3]. 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 to the multitude, at the potential cost of latency.

1.5 The Future

Difficult though it is to predict the future with any kind of certainty, the area of *semantic file systems* has been receiving some attention in recent years, particularly with Apple's *Spotlight* system[4], as well as the rise and fall of Microsoft's *WinFS* initiative [11, 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 2.

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 relational databases or created to search existing hierarchical file systems. As Hans Reiser points out however [20], 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[5, 12, 13, 18, 29], 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 be theoretically stored in a single directory.

Chapter 2

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.

2.1 Document Stores

A *document store* does not just provide indexing on files, but actual data storage for those files as well. This storage may not be strictly part of the store itself, but the two will be very closely linked. For example, files may not be directly stored in the database, but referenced by external object identifiers, as in MWFS[22]. 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.

2.1.1 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, 7] 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.*

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 this, it seems that some of the vision of WinFS may still be alive at Microsoft[21], although they are keeping quiet about any future offerings for the time being.

[1] `http://download.microsoft.com/download/c/e/2/ce28874c-4f44-4dbd-babb-727685e2be96/WinFS_IWish_720x486_2mbs.wmv`

2.1.2 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[16], 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[17], although the last major change was in July 2004. There were many plans for this system, 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.[15]

2.1.3 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[8].

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.

2.1.4 Tagsistant

Tagsistant is a semantic file system that was released to the public in July 2007. It was written with the File system in User Space layer (FUSE) and is compatible with Linux- and BSD-based systems. The author states[26] that:

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

The author also makes the point 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.

2.1.5 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[22], 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.

2.2 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.

2.2.1 DBFS

In 2004, a student at the University of Twente in the Netherlands worked on a database file system for his final year project[10]. 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[10].

2.2.2 Apple Spotlight

Apple's *Spotlight* desktop search tool was introduced in Mac OS X version 10.4 "Tiger", in April 2005[4]. 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.

2.2.3 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.

2.2.4 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.

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

2.3 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.

2.3.1 SFS

The first concept file system that was used to describe the idea of semantic file systems was described in a paper[9] 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.

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.

2.3.2 pStore

The *pStore* file system[30] 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) [28] 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'[30]. 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.

2.3.3 Automated Attribute Assignment

Although this is not strictly a file system, this paper[24] 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.

2.4 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.

Part II

Implementation

Chapter 3

Specification

The best way to predict the future is to invent it.

Alan Kay

3.1 Introduction

Research into the subject of semantic file systems has lead to a number of attempted solutions, but the vast majority of these rely upon integration with a relational database, such as PostgreSQL, SQLite or SQL Server. However, there are many things that a relational database cannot do well that may be desirable in a semantic file system.

Examples of these desirable qualities may include storing both arbitrary keywords describing a file as well as key–value pairs for structured information. This may be the case for images, for example, where the user may wish to store arbitrary keywords (e.g. *forest*, *mountains*, *Christmas*) as well as information about the image itself (e.g. *height* = 640, *width* = 480).

Another use case that may be difficult for relational databases to handle are key–value pairs where one key may have several values. Multiple entries for a given key may be useful if there are (for example) two or more authors for a document.

Finally, there is also the issue of data type storage. In a relational database, columns have well-defined types such as `text` or `integer`. This then leads to an implementation issue found by the developers of MWFS[22] when they attempted to store multiple data types. They had to construct a database schema that contained redundant fields in order to cater for the common data types. The alternatives would be to serialise the data to a string before storing it, or to create separate tables for each possible data type. This can lead to redundancy, inaccuracy or complexity, respectively.

It would appear that a relational database model is fundamentally wrong for this approach, and so implementing an alternative solution is proposed: a modified B^+–tree-based structure, which will be described more fully in the following sections.

3.2 Project Aims

The primary aim of this project will be to follow on from the work mentioned earlier in this report, taking the best ideas into account, adding some new ideas, and trying to avoid the pitfalls of earlier systems as much as possible. The end product should be a semantic file system that may be integrated into the Linux kernel, thereby allowing users to save, tag and organise their data.

Given the limited time frame of this project, implementation of a full file system would be infeasible as that would take many months of dedicated work. For this reason, the new file system, *Insight*, should probably be based upon existing and stable file systems such as *ext3*. This underlying file system may be used as a base for handling file allocation policies and directory storage and so on, to allow the implementation to focus upon the novel features provided by a semantic file system.

While the focus is on Linux for ease of development, the core of the *Insight* file system should be written to be platform-independent wherever practical, as one of the future aims of this project is to produce a modern 64-bit journaled semantic file system which is completely cross-platform.

It should be noted that, although *Insight* will be based upon another file system for the time being, there should be as little actual dependence on this fact as possible so that it can be made a true file system in its own right when time allows.

There should be some consideration of importing files from an existing file system. This will likely be done in bulk, and so *Insight* should be able to handle this import process with minimal user interaction. Having said this, the process may be slower than copying to a standard hierarchical file system due to the need to automatically extract metadata.

Given the complex nature of this project, it is suggested that it should be developed in separate phases that get progressively closer to the kernel. The first phase should create a userspace application that can tag files and search the indexes, based on references to existing files in a file system. This will help to verify the indexing and query code.

The second phase will move this functionality into an application closer to the kernel level, making use of the latest version of the *Filesystem in Userspace*[1] (FUSE) libraries to ease development. This will also mean that kernel recompiles are unnecessary during this stage, while *Insight* will appear to be a true file system.

Finally, it may be appropriate to move the new file system into a module which is compiled directly into the Linux kernel for maximum performance and integration.

3.3 Main Deliverables

The file system *must* fulfil the following requirements:

- Integrate with a recent version of the 2.6 branch of the Linux kernel (2.6.23 at time of writing)
- Create a file system that provides storage for file data
- Allow users to enrich stored files with semantic information or tags, which may take the following forms:
  - Simple keywords, such as *Christmas*, *project*, *report*, *university*
  - Key–value pairs, like *height* = 640, *width* = 640, *mimetype* = “`image/png`”
  - Keys with multiple values, e.g. *author* = “`David Ingram`”, *author* = “`Joe Bloggs`”
  - Structured key–value pairs: *university.year* = 4, *university.term* = 2, *photo.person* = “`John Doe`”
  - Structured keywords, for example *university.year.4* or *photo.person.“Jane Doe”*
- Fast-search indexes across these attributes, with the ability to both search and browse attribute values
- “Live queries” which update their results as soon as a change to the file system is made which would affect those results
- Backwards compatibility with existing hierarchical file systems
- At least one demonstration program that can show off some of the new features of this file system, preferably more. Suggestions include:
  - A graphical file browser that can handle user queries in a fairly intuitive way, and return results.
  - A photo organising application similar to the concept shown in the WinFS “iWish” video, with the ability to easily “tag” people in a photograph.
- Actual file system performance is not the primary focus of this project, and although benchmarks may be made for comparative purposes, it should be understood that performance will only be remarked upon if it is poor enough to be unusable

[1]`http://fuse.sourceforge.net/`

3.3.1 B$^+$ Tree

In addition, the file system should implement a modified B$^+$ tree structure[1, 14] for storing the indexes on attributes, in order to allow fast searches and attribute browsing. The main idea behind this structure is to create a multi-level tree that represents the available attributes. Each level of the tree represents the values available at that level, and each node will contain a reference to its possible sub-values and all file system inodes to which the value applies.

Note that attribute values may have a number of types, including `string`, `integer`, `float` and `pointer` (for referencing other inodes). An initial implementation may coerce all types to `string`s for simplicity.

Note: *In the following figures, the trees have been compressed for clarity. A grey triangle represents other nodes and connections in the tree. The two boxes under a node represent pointers to sub-values and files to which the attribute applies, respectively. A white box with a diagonal line through it represents a null pointer, and a grey box represents a valid pointer.*

For example, simple keywords will all be immediate children of the top-level tree, with no sub-values (see Figure 3.1). Simple key–value pairs will be structured as a two-level tree, with the top level giving the keys, and each key having a sub-tree that gives the possible values (see Figure 3.2). Note that Figure 3.2 shows the key author with the possible values Dave Ingram and Joe Bloggs, but this may also be interpreted as two structured keywords: *author."Dave Ingram"* and *author."Joe Bloggs"*.

The tree structure described above can be extended to any number of levels, so that *university.year.4* (as a structured keyword) and *university.year* = 4 (as a structured key–value pair) are completely identical and evaluate to the same part of the tree. This implementation makes searching for files with particular attributes easy, as well as displaying all possible values for an attribute in a straightforward way.

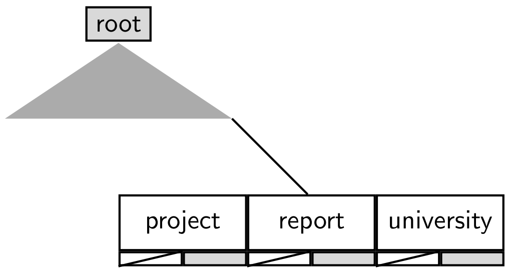

Figure 3.1: B$^+$ tree example for single keywords

3.3.2 Backwards compatability

In order to provide backwards compatability with hierarchical file systems, a method similar to that proposed by Gifford et al. [9] should be used. This also provides a natural mechanism for command-line applications to make use of the new file system.

There will be two types of directory in the initial version of *Insight*, which will only allow conjunctive combinations of attributes. These directories are *key virtual directories* and *attribute virtual directories* (by analogy to the field and value virtual directories defined in [9]. Virtual directories are created on demand by the file system; they do not have to be explicitly created by a program. In this case, unless otherwise specified, virtual directories are invisible. This means that they do not appear in directory listings, but can still be accessed directly by path lookups. This helps to contain the infinite expansion of potential directories.

An attribute virtual directory contains one entry for every file described by the attribute. These may be of the form `/insight/sunset` or they may be normalised key–value attributes such as `/insight/owner.dmi04`, or normalised structured attributes like `/insight/uni.year.4`.

Key virtual directories are also available as virtual subdirectories of the *Insight* root directory (e.g. `/insight`). However, the names of these directories all end in a colon, to distinguish them from attribute virtual directories.

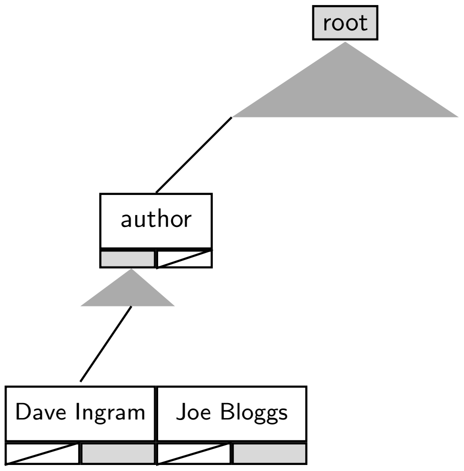

Figure 3.2: B^+ tree example for key–value pairs

A key virtual directory lists all the possible values the key can take as visible attribute virtual subdirectories. For example, a listing of `/insight/uni.year:` may give the directories `1`, `2`, `3`, and `4`, while the listing of `/insight/uni:` would return at least `year` as a subdirectory. The contents of `/insight/uni.year:/4` are identical to the contents of `/insight/uni.year.4`, as they represent the same query.

Logical conjunctions of queries may be achieved by chaining directory elements together. For example, to find all files where the `owner` is `dmi04` which relate to the fourth year of university, the user would merely need to open the directory `/insight/owner.dmi04/uni.year.4`. This is also completely equivalent to:

- `/insight/owner:/dmi04/uni:/year:/4`
- `/insight/uni.year:/4/owner.dmi04`
- `/insight/uni.year.4/owner:/dmi04`
- and any one of the other innumerable variations

Finally, it should be noted that a method exists to provide a visible directory listing of the top-level attributes that are available. This can be accessed via the special `:` directory (i.e. `/insight/:`). It is also worth noting that the choice of `.` for separating elements of structured attributes may be a poor choice, and so a little-used character (which is not a shell metacharacter) such as ¬ should perhaps be considered.

It can be quickly seen from this description that a very powerful query syntax is provided by this method, while keeping backward compatability with hierarchical systems. In this description, boolean `OR` and `NOT` operators have been omitted for simplicity, but it should be possible to add them in a similar way.

3.4 Additional Features/Future Work

With a project of this kind, there are bound to be a large number of features which could be implemented. Some of the features have been considered non-critical for this proof-of-concept implementation and are therefore candidates for future work in this area.

- Optimisations for performance, which may include a specially-designed block allocation policy for the indexes.
- Removal of the dependence on another file system implementation in order to move completely away from a hierarchical file system base.

- Access control lists (ACLs) for attributes, in to prevent some users performing searches on certain attributes.
- Platform independence – this may be brought about by removing the dependency on an underlying Linux file system, but file system drivers would still have to be written for other operating systems, such as Windows.
- A straightforward external API for automatic metadata extraction plugins, both to reduce the workload on the file system developer and to provide a way for external developers to contribute, especially for proprietary file formats.

Chapter 4

Evaluation

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

Bill Cosby

Due to the complexity involved in creating a file system, there are a number of tests that should be met during the project development as well as at the end. This chapter is therefore split into three parts, one dealing with each of the stages: B$^+$ tree prototype, FUSE prototype and kernel module.

The Linux Test Project provides a number of utilities that validate the reliability, robustness, and stability of Linux. Among their extensive test suite[1] are a number of programs for testing file systems. Although these tools are primarily aimed at testing hierarchical file systems and may not be suitable for a semantic file system, they may give some useful results and benchmark statistics.

4.1 B$^+$ tree prototype

The first prototype should demonstrate the B$^+$ tree structure described in Section 3.3 working as an index over existing files. The key features of this version of the software are:

- The software should be able to import file paths
- Files should be shown in a list, and have attributes added to them both individually and in bulk. These attributes may be simple keywords, key–value pairs, structured keywords or structured key–value pairs.
- It should be possible to construct searches using these attributes, although the search syntax may be initially restricted to simple Boolean `AND` combinations of search terms. The search results should be references to files on an existing hierarchical file system that have been imported into the program, and should allow the user to open those files.
- These queries should be "live"; that is to say, they should automatically update their results as the relevant data changes without requiring a manual refresh from the user.
- This userspace program is only required to test the concept of this indexing strategy, and as a result the interface is of little consideration apart from necessary elements for this demonstration.

4.2 FUSE prototype

This prototype should demonstrate file system operations: create, delete, open, close, read, write, etc. with the index being updated accordingly.

The second stage of the project should provide some measure of integration with the Linux kernel by means of the FUSE project, allowing *Insight* to appear as a real file system to the kernel and the other applications on the computer.

[1] `http://ltp.sourceforge.net/tooltable.php`

- The primary focus will be to transform much of the code from the first prototype to the FUSE framework for file systems.
- The code will be required to implement the basic file system operations (see Figure 4.2), many of which will pass through to the underlying file system.
- Not all file system operations must be implemented in order to create a working file system, and most of these operations will be wrappers that pass through to the underlying "repository" file system that actually stores the data.
- The default *extended attributes* mechanism may be used as an interface to the attribute indexes of this file system up to a point, although querying of the indexes would have to use a separate API.
- Querying and modifying attributes should be possible by creating directories to represent tags, and then by copying files into those directories to apply those tags to the files.
- File data should be stored in an underlying "repository" file system specified at mount time, with file names that are based on inode numbers. This naming scheme assures that there will be no file name collisions.
- It may be desirable to add plugins for automatic metadata extraction from certain file types (also known as *transducers*) at this stage.

4.3 Kernel module

The final iteration of the *Insight* project may be a kernel module. This will provide the best opportunity for full integration with Linux, allowing the direct implementation of a file system query API at the kernel level.

- The *insightfs* module should integrate and compile cleanly with a recent revision of the 2.6 Linux kernel series (e.g. 2.6.23 at time of writing).
- The code should be based upon the *ext3* source tree, using it for file allocation policies, on-disk structure, and other core file system tasks. *Insight* will control the user-facing side of the file system, however, interfacing with its internal indexes and hiding the details of the hierarchical file paths from the user.
- Much of the code should be able to be taken directly from the FUSE module with little modification, although there should be some increase in speed due to the removal of the user–kernel divide present in most calls made via the FUSE libraries.
- The file system must still implement all of the necessary file system functions.

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

* Optional function: not required for standard file system operation
† Functions only required if extended attributes are enabled

Figure 4.1: File system operations

Bibliography

[1] Anderson-Freed, Susan, ‘Notes on B+ Trees’, 1998, accessed on 2007-11-17.
`http://www-users.itlabs.umn.edu/classes/Spring-2006/csci4707/B+Trees.pdf`

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

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

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

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

[6] 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`

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

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

[9] 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.

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

[11] 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/`

[12] 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`

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

[14] Monge, Dr. Alvaro, ‘B+ Tree indexes’, 2007, accessed on 2007-11-17.
`http://www.cecs.csulb.edu/~monge/classes/share/B+TreeIndexes.html`

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

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

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

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

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

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

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

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

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

[24] 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).

[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] 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`

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

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

[30] 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).