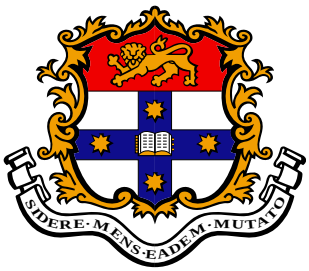
Family: Documents, files, nodes and relationships

Sam Thorogood

CHAI Group, School of IT
The University of Sydney
NSW 2006 Australia

sam.thorogood@student.usyd.edu.au



November 7, 2008

Abstract

Existing file systems fail to support users in identifying important relationships between their own documents. For example, when a user copies a file on their computer, their mental model may link the copy with the original. However, modern operating systems treat these as unrelated. Moreover, it is not straightforward to create a system that can recognise these kinds of relationships.

This thesis describes Family, a system which aims to support users in seeing the temporal sequence of files that are part of the history, or lineage, of any nominated file. We term these files the *ancestors* of this nominated file, and the inverse relationship the *descendants*. We also identify key conceptual models that constitute the cases of important family relationships. These correspond to simple operations on files (such as copy, rename) that may be implemented in various different ways at the operating system level, depending upon the application used to undertake these operations. These are operations that will typically involve multiple complex actions at an operating system level, but which are conceptually simple to end-users.

We introduce a formal algorithm for Family, as well as its specific implementation, known as Lighthouse. This algorithm is defined explicitly, describing how the broad concepts of *ancestors* and *descendants* are found through low-level analysys and observeration of low-level system activity. Finally, we evaluate Family and Lighthouse in terms of correctness versus a predefined standard, as well as performing a short performance evaluation.

Acknowledgements

Firstly, I would like to thank the creators of UNIX for popularising the wonderful modern file system that all computer users must interact with today. Secondly, I must thank Apple for creating a wonderful environment on which to develop and obtain the most relevant of information for use within my implementation.

This thesis would not have been possible without the wonderful support of my academic group here at the University of Sydney. I would like to thank Judy Kay, my supervisor, for her support and suggestions. My fellow Honours students Greg and James were instrumental in sanity-checking all my code. Coffee runs with Greg, Mat and Katie have also kept me sane. Tim Tams on Thursdays (or was it Tuesdays) kept me going with regular chocolate injections – thanks to Trent for originally organising this. Not to mention your wisdom and support throughout the first half of the year. I would also like to thank Anthony, a PhD student, for my incessant questions about LaTeX. Also worthy of mention are my fellow Honours students – again, Greg and James, along with Mat, Katie, and others – I hope to keep in touch with all of you for many years to come.

Of course, I have to mention my family – Mary, Peter, Jen and Jay. They have put up with my late nights and late mornings, making sure I still eat and keep healthy, as well as providing feedback and editing services.

Finally, many thanks to the Apple University Consortium in Australia for their support through both an Honours scholarship and an Innovation Seeding Grant.

Contents

List of Figures

Chapter 1

Introduction

Every day, users undertake a multitude of conceptual actions over their own files in the pursuit of achieving their computing goals. These actions are typically well-known to users and are well-defined by most applications that work with documents. For example: a user might *copy* a report template, before editing it and perfoming a *save*, before *moving* the file to a USB stick.

As part of this daily process, a user may build up a *mental model* of how their files relate to each other, based on these simple conceptual actions. As per our example, the end-user may hold a simple understanding that the file they have just worked on is based on the original template. However, modern operating systems are not able to capture this seemingly simple relationship, and others like it. From the point of view of Mac OS X, Windows, and Linux, these two files are independent and unrelated sequences of raw data. One of the key reasons for this is that the seemingly simple conceptual actions previously mentioned, including *copy* and *save*, are typically implemented as a multitude of lower-level system actions.

We introduce an example in Figure 1.1, and describe it below.

> John is working on a conference report. Initially, he stores his notes in a file named **Notes**. Later, he creates a new file, **Final Report**, based on his notes. At this point, he may reasonably associate the two documents within a mental model of his personal dataspace.

As noted above, modern operating systems will treat these two new documents as unrelated – as from a functional point of view, these are now two completely unrelated files. However, such relationships can be important for finding the right file for a particular task, or when an activity calls for key related files that were used to create an existing file. John may therefore find it useful to have this information available to him, rather than relying on his own personal mental model, as continued below.

> Days later, John is working on **Final Report**. He realises that an important figure is missing from his report – he may have deleted it accidentally. However, he cannot remember exactly which notes he originally used to create this document, as they have already been filed away into another directory which contains presentation notes from all conferences held in 2008.

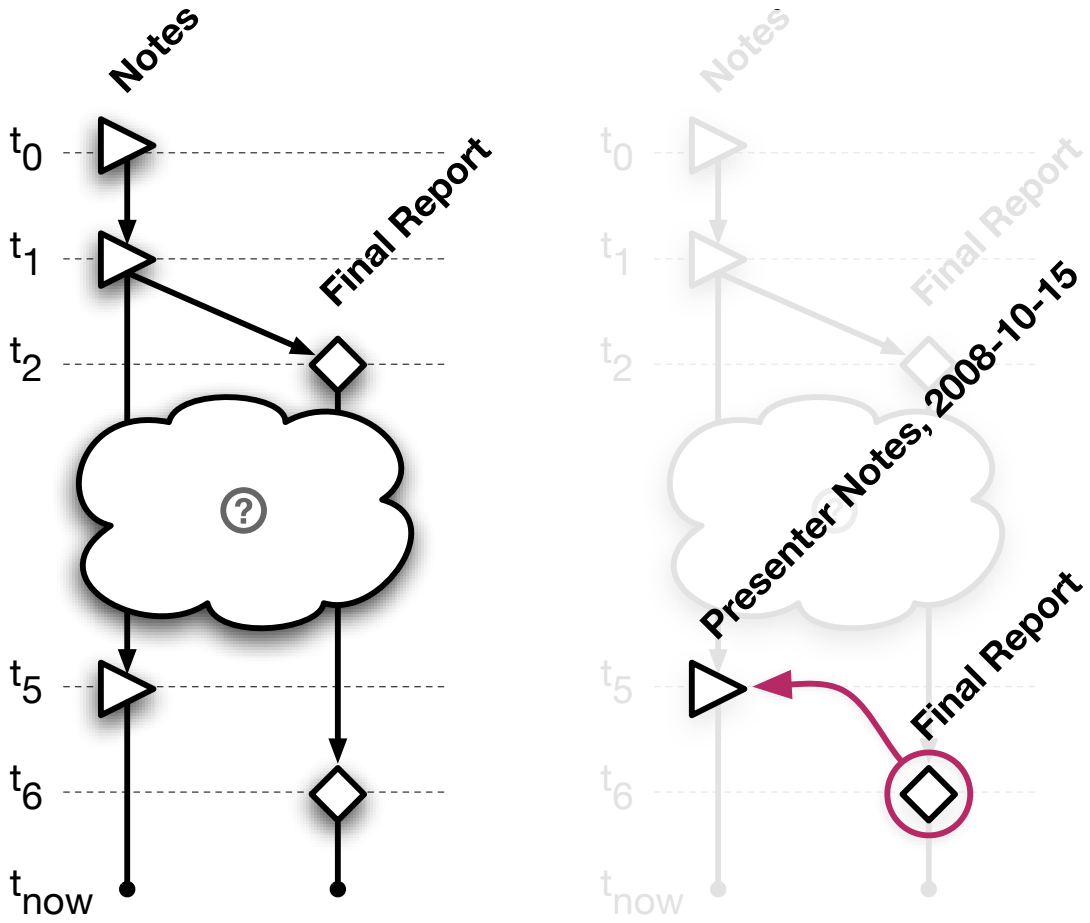


Figure 1.1: A simple relationship example, represented by the left graph. Each icon represents a unique state of a file – for example, **Notes** was created at t_0, and updated at t_1 and t_5. **Final Report** was copied from **Notes** at t_2, and updated at t_6. After some time (represented by the cloud), the two files may appear unrelated, but they retain a common ancestry. Family attempts to maintain information about this ancestry, and it could be used to simplify the relationship between the 'current' versions of these two files – as shown in the right graph – to be presented to an end-user.

John uses Family to find files that are related to **Final Report**, revealing his original set of notes. The document is now named **Presenter Notes, 2008-10-15** and is located in his presentation notes directory.

Relationship Classes

Broadly, we are concerned with two classes of relationships between files. The first, as described through the above example, relates files through conceptual actions such as *copy*, *duplicate*, or *cut and paste*. As mentioned previously, modern operating systems are not able to capture relationships of this type.

The second class of relationship links the unique states of a file over time. As files are changed, modified and *saved* over time, their content will exist in various different states. However, within most modern file systems – such as HFS+, NTFS, ext3 and XFS – files are mutable. While these file systems can provide end-users with the time each file was *last* changed, there is typically no support for retrieving information about *every* time they have been changed. In other words, there is traditionally no file system support to identify each unique state of a file over time.

There are two key motivators for being concerned about the states of a file over time. The first is a task-based goal, with the aim of relating work undertaken at a similar point in time. This information may be useful for task management or backtracking over the lifetime of one's own documents. Additionally, having a view of changes over all files – as opposed to just a single file – may help to better represent a temporal link between changes performed on many different files during a common timeframe.

We give an example of our first, task-based goal below. This is displayed in Figure 1.2.

> A user, Johan, may be working on a poster within some graphic design software. He opens the file and notices a small mistake – fixing it and instinctively saving the document. He is now interested to determine when he last worked on this document, but the 'last changed' field now reflects the time at which he fixed the typo. He was interested in finding the notes he was working on around the same time as the poster, based on their temporal usage, but will now have to approximate this information.

So, although files may have been changed at similar times as part of one activity, if some of them are later changed there is no way to detect the temporal link – or *relationship* – suggested by the work done on them at the same time.

The second motivator for interest in the states of a file over time is tied to our first class of relationship – where files are related through conceptual actions such as *copy*, *save as* or *move*. Temporal relationships are important. These states are important, as the first class of relationship may relate directly to a specific state of a file, rather than to the file as a whole. For example, one file may be *copied*, before it is updated and *saved*. The duplicate file resulting from the *copy* operation is not related to the new, updated version: it is related only to the previous state of the file. We describe this relationship as forming the history of any given file.

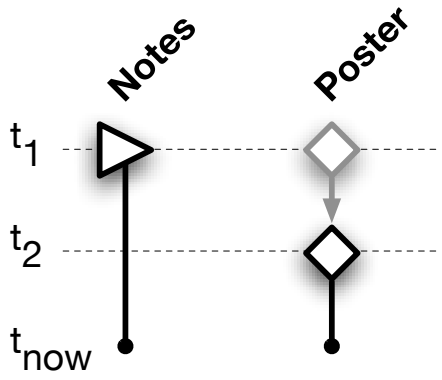


Figure 1.2: A simple example demonstrating the relevance of temporal relationships. As modern file systems only present the *last* time a file was changed, the temporal indication that these files are related (at t_1) will be lost if one of the files is changed again later (at t_2).

The combination of these two classes of relationship aims to approximate the *mental model* of how an end-user may perceive their files over time.

1.1 Thesis Structure

This thesis describes Family, an architecture which aims to support users by capturing and revealing these seemingly simple relationships, providing information about the temporal sequence of files that are part of the *lineage* of any nominated file. We also identify key conceptual actions that map directly to these important relationships which we aim to identify. We term these mappings as *conceptual models*, and a later section of this thesis aims to define them explicitly – including descriptions of how different applications may enact them in terms of lower-level system actions.

In Chapter 2, we introduce the field of "Personal Information Management", or PIM, which has been defined as "the practice and study of the activities people perform to acquire, organize, maintain, and retrieve information for everyday use." [1]. While the contributions of Family are very specific, we do have a more general objective of enriching this field as a whole, and this has motivated our research. Furthermore, this chapter reviews and introduces other relevant work that has a similar goal of supporting end-users in this field.

Chapter 3 details our approach to both the algorithm and architecture of Family. To find and capture relationships from conceptual actions between user files, we rely on a fairly complex algorithm. This algorithm has been implemented – something we describe in Chapter 4 – in Python, and while it requires some unique file system features (such as extended attributes) it is largely platform-independent. However, there is still a large proportion of platform specific-code which has the sole aim of providing all low-level system information, which is typically exposed in different ways on different platforms. In the case of our implementation, we target the Mac OS X platform, with this code implemented in C.

Our evaluation, both design and results, are covered in Chapter 5. We primarily focus this evaluation around determining the correctness of our algorithm.

Through our research into this area, and our implementation of Family, we have reached several core conclusions which have implications for future work. We discuss both of these elements in Chapter 6.

1.2 Vision & Motivation

Currently, PIM is a decentralised and disorganised environment. Indeed, it has previously been described as 'fragmented' [16]. We propose that this is fair assessment: modern computer users are regularly presented with a myriad of different applications and file formats which may all be equally as useful as each other. This leads to substantial integration difficulties.

Modern PIM is now also more decentralised than ever before. With the mainstream adoption of the internet, and the introduction of mobile computing through devices such as laptops, phones, and even the now-humble USB stick, users have been empowered to move, send, email, or even upload their data to have it accessible anywhere they are. However, this benefit came with a cost – the reality of being empowered with one's own data also made the world more complicated. Having multiple devices allowed different versions of a document to be located in two places at once. A user who emails a file to himself in order to back it up has to keep track of what version it is, and how relevant it might be if he needs to restore from it. More generally, the idea of always having the "latest version" available wherever a user wants it to be available has become both paramount and largely ignored at the same time [13].

The existing state of PIM is demonstrated in an example below.

> Zoe is a fairly non-technical computer user, working on a presentation using her desktop computer at work. She wants to take it home and continue working on it through the evening, so she transfers it to her portable USB drive. Using her personal laptop, she finishes the presentation at home. The next day, she's back at work – but has forgotten her USB drive. She is over an hour away from home, but needs that file. Her two options are to drive home to get it, or recall all the changes she made the evening prior.

The most obvious use of Family might be to help Zoe see that these files are related. Still, it should be fairly clear that she already has a good grasp of this – to her, both files represent the same *concept*, and she is only interested in having the latest version available to her. Family is broadly aimed at resolving this issue – by providing these sorts of relationships to end-users and higher-level applications. However, making such grand claims about the meaning of these relationships such as claiming that two files represent the same *concept* is out of scope of our work.

We present a more futuristic vision of PIM below, corresponding to the same scenario.

> Zoe is a non-technical computer user, and she is working on a presentation at work. She transfers it to her portable USB drive so she can work on it later at home. The next day, she's back at work – and has forgotten her USB drive. She brings up the file located on her work computer, and even though she already

understands that there is a more recent version available on her home computer, she still asks her operating system to show her any outstanding or newer instances of her presentation. This interface then allows her to automatically and seamlessly retrieve and merge her content from home.

While this example may have been 'simplified' through the use of version control tools, these may be seen as overly technical for the average end-user. Modern operating systems do not come 'out of the box' as standard with this sort of functionality. They may also require explicit steps to use their features.

It is worth noting that Zoe's initial actions in both the above examples are the same. Despite the age of the traditional file system, various research ([15; 17]) has shown that end-users are actually quite comfortable with the idioms it implies. So, while Zoe does have the benefit of resolving her problem in a new and novel way, our vision does not revolves around altering traditional user behaviour.

Existing Applications

Our introduction also defines a relationship class which describes files in unique states over time. Our motivation for this class is to reveal not only when a file was *last* changed, but also *every* time it was changed. While this information is important to create correct, time-relevant conceptual relationships – such as when a file was *copied* – it also has a simplier application to systems that already exist today.

Apple's 'Time Machine' [4] is a backup tool integrated with Mac OS X Leopard that allows users to see the state of their computer's files at regular historical intervals. However, Time Machine is limited in that it will only back up documents on a regular basis, rather than in response to changes undertaken by users. In a futuristic vision, Time Machine would back up relevant parts of a user's hierarchy in response to when, and how often, these areas were used – a document which is saved and updated every five minutes might be more important to archive than a file which is changed every three or four days. This information is, as previously mentioned, not available within a traditional file system.

Time Machine can also not identify files with respect to their location over time. If I move **Report** to my Documents folder from my Desktop, Time Machine will simply create a new backup of **Report** in Documents and it will fail to correctly link this to the previous location that it was stored. A user then seeking an old version of **Report** on Desktop will have to remember that they moved the document at some point in the past. Ideally, Time Machine would be able to use relationship information to automatically direct users to the correct file.

1.3 Challenges

None of the aforementioned relationship classes are currently revealed through traditional file systems. Some research into PIM aims to 'rewrite' the way end-users perform PIM [19]. However, a core aim of Family is to capture relationships classes *without* any explicit interaction from end-users. The motivation for this aim is driven by the fact that – as introduced previously in our vision – various research suggests that users are actually comfortable with the way

the traditional file system works, and the 'filing' idiom it provides. And, other research ([16]) at least accepts that end-users will use the traditional idioms provided by hierarchical file systems in order to achieve their goals – even if they do not go as far as proposing that they actually like them.

Thus, we are presented with a core challenge, which reduces to determining what is occuring within a low-level file system that does not reveal this information in the first place. The file system at any given point represents the outcome of how applications have interacted with it. Thus, a natural approach to solving this problem is to analyse the actions these applications undertake while *using* the file system, and attempt to infer meaning from this.

However, this is more complicated than may be originally perceived. As mentioned previously, while users may undertake *conceptual actions* using applications – such as *save*, *rename*, *copy and paste*, etc – these same applications will actually apply the user's conceptual meaning to the file system in terms of a multitude of lower-level system calls. Furthermore, different applications may deconstruct these conceptual actions in different ways, resulting in a situation where a single fixed approach to analysing application behaviour is insufficient.

There is also a second core challenge in supporting Family, which revolves around the idea of *unobserved changes*. While it seems difficult – but plausible – to interpret how applications deconstruct a user's conceptual actions, and convert these to relationships, the idea of deconstructing actions from applications we cannot even formally observe appears almost impossible. The most simple example of this is if a document is sent to another user who edits it and then returns it, we would still like to identify the changes that it may have undergone. In this case, we are actually forced to observe the outcomes of behaviour rather than being able to observe the behaviour itself. However, this is both an incredibly difficult task as well as being out of the scope of this thesis. Thus, we reduce our goals: we only aim to correctly observe that a file was previously related to some file on a local computer, rather than explicitly defining *how* it is now in its current state. We can visualise this much like an ambiguous 'cloud' (as seen in Figure 1.1) – still providing end-users with some useful information, but of course not able to determine every point in time at which it was changed or updated, and therefore, every one of its unique states.

We discuss our algorithm and implementation – which aims to overcome these challenges – in Chapter 3.

1.4 Contributions

- The *idea* of Family

  We introduce Family, a new and novel way to support users through capturing and revealing seemingly simple relationships that may already exist within a user's *mental model*. This support is implicit, as end-users are also not required to 'rewrite' their PIM strategies in order for relationships between their files to be captured. Furthermore, we discuss the most relevant classes of relationships in this field, and how they map to a user view of their files through this mental model.

- Algorithm & Architecture

  This thesis details how we overcome several core challenges through a concrete algorithm designed to capture information about relationships between files. This algorithm, as mentioned, is designed to observe applications in their activities over traditional file systems, rather than seeking to examine their output.

- Research Outcomes

  Research and development of Family has led to various interesting conclusions about our approach to determining Family relationships.

- Implementation

  We implement Family on the Mac OS X platform, using a combination of low-level code in C and high-level code in Python. This tool will also be publicly released under a GPL license, with the title *Lighthouse*. It is made available for download at `http://www.it.usyd.edu.au/~stho2688/`.

Chapter 2

Related Work

Most, if not all, modern operating systems are based around the use of traditional file systems to help users achieve their computing goals. This can be more broadly described as personal information management, or PIM. The international conference CHI defines the study of PIM as "the practice and study of the activities people perform to acquire, organize, maintain, and retrieve information for everyday use." [1].

The traditional hierarchical file system was originally popularised through the development of UNIX [21], and to a lesser extent, Multics [10]. Daley and Neumann [12], in their work on Multics, originally laid out a specification for a general purpose file system in 1965. They discussed the idea of using symbolic addresses (i.e. filenames) to map to absolute, physical locations on random access devices such as hard disks. This paper also formalised the definition of a directory, a special file maintained by the file system which could contain a list of links to other files. This definition allowed for directories to contain other directories; Daley and Neumann simplified their definition of a hierarchical file system by describing it as "a tree of files" 2.1. Today, nearly all form of personal information management completed on any modern operating system still takes place within the scope of these ideas – originally brought into the mainstream nearly forty years ago.

Since their inception, hierarchical file systems have been both criticised [19; 7] and praised [15; 17; 8] for their limitations and benefits, respectively. Regardless, they are a core element of any modern operating systems, and form a basis for all forms of PIM.

Blunschi et al. [7] introduce a vision consisting of a 'personal dataspace' consisting of automatically replicated emails, documents, pictures, etc. Elements of this dataspace may be explicitly shared to other users. However, all interaction, even by users external to the system, is envisioned as a seamless operation over this data which has been automatically replicated across all relevant personal devices. Blunschi et al., in a large way, introduce a de-facto standard for the goals of ultimate PIM integration.

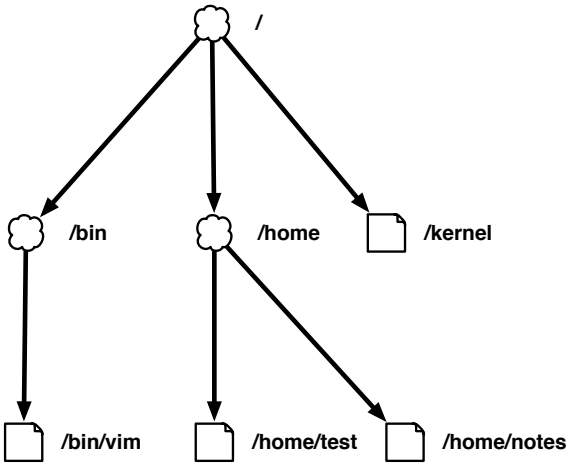


Figure 2.1: A simple file system tree, similar to the one originally outlined by Daley and Neumann [12].

2.1 Files

The traditional file system aims to provide files, ordered sequences of elements which – to an underlying file system – are essentially devoid of format [12]. No meaningful action on any of these files may be performed without, for example, the involvement of a higher-level tool such as a text editor. Various authors describe both the benefits and the difficulties with this simplistic approach. Blunschi et al. [7] coin the term "Personal Information Jungle", reflecting the so-called jungle of currently available file formats and data processing solutions. They propose that computer users endure *physical data dependence*, relating to the fact that users need to know about devices and formats that are used to store their data in order to effectively achieve any goals related to PIM. However, it may also be argued that this unstructured approach has led to the innovation of unique and interesting file formats and storage practices, not inhibited by having to provide any higher-level meaning.

Applications will typically use files and file systems to their own advantage; storing information that they process in specific locations or in forced hierarchies. The formatless nature of files on a traditional file system provides a perfect environment for this kind of [ab]use. Furthermore, Karger and Jones [16] describe PIM as 'fragmented'. They introduce various unification strategies, and discuss their limitations – considering that the lowest common denominator of any application is a file, and that *any* need for higher-level meaning is compromised by attempting to provide a system that supports all possible usage of this base structure.

Mahalingam, Tang and Xu [19] observe that human memory operates in terms of document content or features, rather than filenames and position in a file hierarchy (in this context, features may also represent relationships between files). They argue for new file access mechanisms, beyond hierarchies containing files of arbitrary data. This notion is similar to the approach of WinFS [20], which was a feature originally planned for Microsoft's most recent operating system, Windows Vista, but removed from the final release. It aimed to provide

a "relational filesystem" which could provide lifelong document identifiers, as well as create relationships between these identifiers. In the initial implementation, it captured a small selection of relationships, such as 'author' field of an email entity being related to a unique contact card entity.

2.2 Layout

While this thesis does not focus explicitly on use patterns or behaviour of users within their own folders, they are an interesting starting point. For example, the placement of multiple files within the same folder may explicitly define – inside a user view – that they have some relationship, such as being components of a bigger task.

So, existing operating systems provide very good support for hierarchical relationship with folders. This is a valuable relationship: James et al [15] established that it serves the 'average user' well and Karger et al [17] show how users benefit from organising documents into a personal hierarchy. Notably, this only captures relationships if the user places related files into the same position in the hierarchy. However, a hierarchy has various limitations. Different applications may restrict or control placement of their documents within the hierarchy, or enforce their own. For instance, Mac OS X's Mail application represents each mail message as a single file on-disk, but enforces their location and creates its own virtual hierarchy within the application.

2.3 Temporal Usage & Relationships

Current implementations of hierarchical file systems provide only limited support for users to identify temporal relationships over files. While these systems can provide end-users with the time each file was *last* changed, there is no support for retrieving information about *every* time they were changed. This information may be useful for task management or backtracking over the lifetime of one's own documents. Additionally, having a view of changes over all files – as opposed to just a single file – may help to better represent a temporal link between changes performed on many different files during a common timeframe.

> A user, Johan, may be working on a poster in some graphic design software. He opens the file and notices a typo – fixing it and instinctively saving the document. He is now interested to determine when he last worked on this document, but the 'last changed' field now reflects the time at which he fixed the typo. He was interested in finding the notes he was working on around the same time as the poster, based on their temporal usage, but will now have to find this file manually.

So, although files may have been changed at similar times as part of one activity, if some of them are later changed there is no way to detect the temporal link – or *relationship* – suggested by the work done on them at the same time. In this case hierarchical file systems do not support a user's mental model of file *ancestors* as discussed initially.

2.3.1 Explicit

Cox and Josephson [11] discuss the issue of temporal usage in terms of file synchronisation. For background, Pierce and Balasubramaniam [6] present an inferred framework for describing file synchronisers – these are tools are intended to manage the replication of files over multiple (potentially disconnected) traditional file systems. A conclusion that may be drawn from successful operation of these tools is the implicit relationship that the files it manages are identical to their synchronised counterparts. Cox and Josephson propose that while traditionally, synchronisation times are considered to encompass a file modification history, this is actually only a limited view of managing distinct versions of files. For example, some file synchronisation systems may perceive many changes in a single file, and a submission of this file, as a single action. However, by maintaining a distinct vector of every time a file was changed, we can better understand how all changes have been propagated to different replications prior to this point in time.

Regardless, a conclusion that may be drawn from successful operation of these tools is the implicit relationship that the files it manages are identical to their synchronised counterparts. For example, if a document titled 'Report' has been synchronised into two distinct hierarchical locations, these two distinct file system entities may be implicitly viewed as being an identical *version*, and exist as a single concept in a user mental model. However, even after these relationships have been identified implicitly within some software, they can not be exposed in a consistent way. No file system allows for these relationships to be defined on files directly, so these relationships only exist because higher-level tools like file synchronisers define them.

Similarly, version control systems [9; 18], widely used by programmers, can assist in identifying relationships between existing files and their previous versions. As well as this, these forms of tools can provide temporal information about 'changesets', or blocks of work – potentially completed on multiple files – completed at specific points in time. In a similar vein to file synchronisers, these changes must be explicitly *checked in* to the system, and will treat all work completed since the last check in as a single block, ignoring specific temporal changes.

However, users must interact explicitly with both traditional version control tools and file synchronisers. For example, a user must explicitly stop and *check in* their changes. Mainstream version control tools are not able to uniquely identify user work and implicitly track this over time.

2.3.2 Implicit

Conversely, there has been some work towards our vision of implicitly capturing and presenting relationships. Soules and Ganger [22] created a system capable of making use of information about file system access to identify temporal relationships between files. Essentially, they aimed to group sets of files based on the time at which those files were used. This grouping was then used to enhance search results for end-users. They reported that this approach improved search precision from 17% to 28%. They also made use of the 'relationship' idiom, creating a relation graph based on file system usage. However, their relationships did not include the concept of *versions*, or our notion of *ancestor*. For them, every file was treated as a single, most current version.

Chapter 3

Approach

This chapter describes the the algorithm behind Family. The primary contribution of this section is the description of the algorithm, and architecture, used within Family to capture conceptual relationships. However, we will first introduce the relevant terminology, as these terms are unfortunately fairly overloaded.

We introduce three key terms: *document*, *file* and *node*. These are important as they bridge the gap between a user's *mental model* and the extremely raw file system view describing the unique state of every file.

- *Mental model*

  Broadly, a user's *mental model* describes the way in which a user thinks about each file and its relationships with other files. For example, a user might consider several documents in various folders to be related to each other. One of these files may have originally come from an email attachment, and a user's mental model might include this relationship. Another file, **Notes new**, may be a copy of an older file, **Notes**, which has been changed. However, modern operating systems do not typically attempt to support this model. They do not understand or attempt to support conceptual meaning or relationships such as 'new versions'; to them, every document is just a *file* – an independent sequence of bytes with a meaningless name.

- *Document*

  The notion of *document* is part of a user's mental model. It will have a known name, such as **Report 10-02**; a known location, such as a user's desktop or on a USB stick; and typically also contain content that may be viewed or changed by an application able to open documents of its type. And, regardless of how a document is actually technically contained on a disk, it will typically be seen by a usert as a single entity that remains constant between its creation and deletion. If the document is copied, duplicated or created by saving another document with a new name, a user is able to update their own internal mental model to capture the way in which these new files are related.

- *File*

  The term *file* tends to be overused, but from the view of the operating system, a file can be simply defined as a filename and a sequence of bytes. From the higher-level view of an application, a *file* may be interpreted any way they like – a PDF viewer will interpret a PDF in order to render it, while a tool to count its pages will ignore much of the content and focus on a few specific facets.

  From the view of an end-user, the terms *file* and *document* tend to represent the same idea. That being said, we have throughout this thesis used both terms fairly interchangeably in regards to the way users view their file system. However, while a *document* can be understood and arranged by a user into relevant mental models or user views, a *file* does not enjoy the same privilege – it only exists in a technical context. Additionally, a *document* may be comprised of a package folder which actually encompasses multiple files.

  Modern operating systems and file systems also support the notion of extended attributes, which may be applied to any specific *file*. These are arbitrary key/value pairs that may enhance a file or provide out of band data. Additionally, one of the fundamental properties of files is that they are mutable, and only have a current state. They are created and updated over their lifetime backed by a single *catalog identifier*, a unique identifier that has explicit relevance to each specific mounted file system. Users may save new versions of files regularly, causing the overall file concept to be updated, and information about the previous state to be lost.

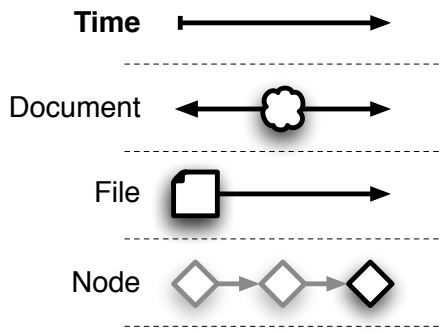


Figure 3.1: Demonstrating the leap between our three core concepts.

- *Node*

  Family embraces the term *node* to describe an immutable state of a file, where *state* describes a file's name and content for a particular period in time. A file must always map to at least one node, as its content always represents one state. However, a file may also map to multiple nodes, if it has previously existed in other states. This is the case if, for instance, a file undergoes changes – rather than being created and deleted but never modified, in which case the file will continue only to map to a single node. This distinction is demonstrated explicitly in Figure 3.1.

  Technically, each *node* has a small set of properties which allow it to be mapped to

a real-world *file*. These include properties such as file size, local *catalog identifier*, along with the file's creation and last modified times. Each *node* also contains a set of evidence, beginning empty, which may grow throughout its lifetime. These facets of evidence either provide information about the node itself (such as when it was last opened), or they provide information about how this node relates to others (such as if it is a *descendant* of a previous node). Both these important features of nodes – properties and evidence – are heavily used within our algorithm, introduced below.

We show the jump between our three core definitions – of *document*, *file* and *node* – in Figure 3.1.

3.1 High-level Formalism

Firstly, we describe various formal approaches for the form of data we aim to capture. Our evaluation, to be completed in Chapter 5, is assessed on the basis of our implementation's correctness versus these approaches.

- $Anc(X)$ – or ancestors – *nodes* the user acted on to create X
- $Dec(X)$ – or descendants – *nodes* of which X is an ancestor
- $Family(X)$ – a combination of both $Anc(X)$ and $Dec(Anc(X))$.

These formalisms are key to Family, as they describe in specific terms our ultimate goals. While we have originally described relationships in terms of classes, these ultimately reduce to either providing information about *ancestors*, or information about *descendants*. It is also worth noting that the two base cases, $Anc(X)$ and $Dec(X)$, are simply the inverse of each other.

We can list a short example of *ancestors* of an arbitrary node, A. Content may have been copied from B into A – therefore, B fits our definition of ancestor. However, A may have also been originally created by copying C. Additionally, C may have been originally created by simply making changes from D! Thus all these nodes – B, C, and D – would fit the definition of $Anc(A)$.

Conversely, we can show the *descendants* of node B as much the inverse. As content was copied from B to A, it fits the definition of a descendant. Additionally, if B was at one point copied to F, it also gains the label of descendant. So A and F both fit $Dec(B)$.

3.2 Conceptual Models

The above formalisms are constructed through *node* relationships that fit various definitions within a user's traditional mental model. The below conceptual models are the most important set of these relationships, and they explicitly define how the nodes they involve should be defined in terms of ancestors and descendants.

We have identified four important cases described below.

- Document copy or 'save as'

  For example, a user might use a command-line tool to copy one file (e.g. **Notes**) to another, brand new file (e.g. **Abstract**). They may alternatively open **Notes** with a text editor and use the standard 'save as' functionality to create the new file, **Abstract**. By mapping this user functionality to abstract nodes, relationships may be drawn that would allow users to backtrack from the newer file, in this case **Abstract**, to its original source – **Notes**. Formally, the *ancestors* of the node behind **Abstract** now include the node behind **Notes**. Similarly, the *descendants* of **Notes** now include **Abstract**.

- Document changes (edits) over time

  As a user saves his or her work in-place over time, each save is identified by the creation of a brand new *node* representing the current state. This provides support for identifying revisions and historical relationships. For instance, if a document is copied at 10:15 to create a new version located elsewhere, it is important to appreciate that this copy was made from a specific immutable state of a file (i.e. a *node*), rather than from its entirety as a mutable concept over time.

  Again, the way this conceptual model fits within our definitions of *ancestors* and *descendants* is fairly clear-cut. The older state is now a direct *ancestor* of the current state, and the current state is now a direct *descendant* of the older state.

- Related work on documents, including copy and paste from one document to another

  It is important to identify more general relationships between two documents. Documents can be related on a task level – for instance, two documents may be open at the same time as they are both related to achieving the same user goal. Text and content may even be copied back and forth from both documents to each other. By identifying that these documents were in use at the same time, we support the notion that users can backtrack from their current state and determine documents which are not strictly related in the sense of *family*, but are related in other ways and may still be useful to support the mental model.

  More specifically, this relationship is again very back-and-forth. If content is copied from one document to another, then the original document must now be perceived as an *ancestor*, while the target document must be considered as a *descendant*.

- Documents sent and returned outside the machine

  The previous three examples all refer to specific cases that may be observed on a controlled, local system. However, Family extends to work performed on multiple machines or by multiple users. Put simply, a user may send a file to a colleague, who may make changes to the file and return it. A user's mental model might support the idea that this file has a similar name, and has a later modification date, so it is likely that this new file is somewhat related to the original. We would like to identify this relationship semi-formally, with an understanding that while the file is 'out of control', its state is generally ambiguous (represented by a cloud in Figure **??**).

  While this conceptual model is less specific, it still fits into our definitions of *descendants* and *ancestors*. This is largely because despite the uncertainty of the actions a file may have undergone outside the system, there is still a temporal connection between the

original file and the newer, observed file. In a broad sense, we see the change as one 'leap', as opposed to many, but we can still see the returned file as an *ancestor* of the last state it was observed in. Furthermore, like our previous examples, the opposite point of view considering *descendants* is also true.

3.3 Low-level analysis

The algorithm of Family, in an overall sense, involves observing system calls in order to determine relationships between *nodes*. These relationships aim to map to the conceptual actions users have undertaken, such as *save*, *copy* and *duplicate*, which then map to the base cases of *ancestors* and *descendants*.

System calls

We perform this algorithm for every single system call we observe. These system calls may be slightly different depending on the operating system focused on, but they fit into several simple broad categories. Many, for instance, are merely useful to suggest that a file should be analysed for changes: even though they do not explicitly define that a change has actually occurred. They fall into the following three categories.

- Requests; e.g. `open()`

  These include events such as the low-level action of opening a file, including authorisations for specific actions. For instance, a PDF reader may open a file requesting only to read it. However, a text editor may open a file with both read and write authorisation. These authorisations do not imply that an application will modify a file, just that they have *permission* to do so.

- Changes; e.g. `unlink()`, `creat()`, `write()`, and varieties of `open()`

  Single-file changes come in a variety of forms. In most cases, these will be unique and will be recognised by Family in various different ways. Broadly, they fall into several sub-categories.

  - File creation

    The creation of files is a key action that may be undertaken by applications. In some cases, this event may be split into the creation of a file versus the creation of a folder or directory. There may also be a special case if a *link* is being created to a previous file, rather than dealing with a brand new file.

  - File deletion

    Deleting files is another key action that may be undertaken by applications. Again, special considerations must be made where the file is linked to from multiple locations. Files may also be deleted as the result of rename operations, discussed below.

- Changes in content

  Depending on how a file system provides these events, they may be provided in a single notification after-the-fact or as intermediate notifications on specific reads. Regardless, these events are typically coalesced as an application may perform hundreds or thousands of individual changes in order to perform an overall update.

- Changes to extended attributes

  A file may have its owner changed, or extended attributes changed. These changes are less relevant to Family, as most applications do not perform substantial activity using these operations. In fact, Family actually uses extended attributes to achieve its goals, therefore creating an issue with loopback if these changes (which Family performs itself) are observed by itself again.

- Movement or structural change; e.g. `rename()`, `exchange()`

  As well as changes to files in terms of their creation, deletion and content, we are also concerned with their movement within a system. If a file is renamed to a new, unique name, this is perceived as a new unique state. There is also a special case where an operating system attempts to rename *over* an existing file. This is interesting – as regardless of its underlying meaning, it implies that the existing file is to be removed.

  Mac OS X also provides for a unique system call: `exchange()`. It allows for the content of two files to be atomically 'flipped', therefore invaliding the current state of both. We support this system call within the architecture of Family, but again, it is only provided on Mac OS.

Since our algorithm (defined in 3.4) analyses every single system call, these system relevant as they allow the algorithm to perform differently depending on what category they fall into. The quirks of each event are also vital to understand in order to correctly process them.

Node identification

From a functional point of view, Family attempts to identify and 'tag' all files currently in use by a user. Tagging a file allows it to be uniquely identified over time, and in fact, each tag maps directly to a node. As files are mutable over time, Family seeks to observe any change in its properties which may imply an overall change in state. It performs this by attempting to assert that the newly changed set of properties still matches the properties tied to the given 'tag' and its mapped node. If the assertion fails, a new node is created – marking the old node as destroyed – and evidence is added to give reason to the way the new node is related to the old node. In the simplest case, this evidence simply lists that the new node is directly derived from the previous node. The properties that may be examined include a unique identifier for the local disk, the file's unique identifier (or 'catalog ID') on disk, a unique identifier for the file's containing folder, its size, its creation date, and its last modification date. This set also includes the file's name.

For example, a pre-existing node may be copied (using a file manager like Mac OS X's Finder) to a new node, creating a brand new file. In this case, the pre-existing node is never changed, but Family will detect that the new file was created from copying the pre-existing node, and

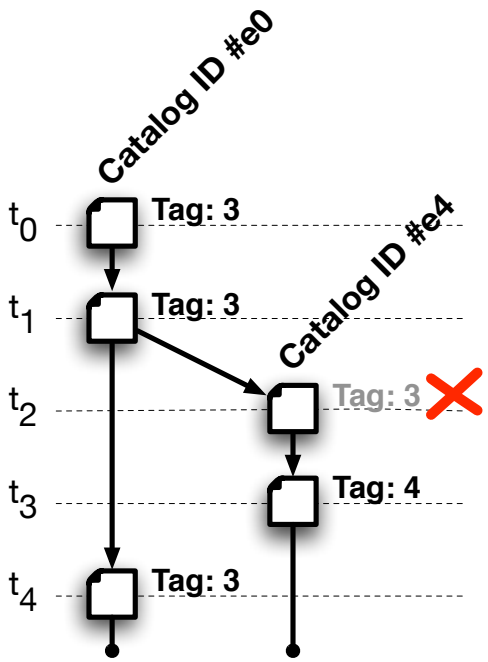


Figure 3.2: Identifying a copied file

create a new node with evidence to suggest as such. Figure 3.2 demonstrates this example making reference to the 'tag' idea described above. A file is first created at t_0, located within the local disk with catalog ID $\#e0$. Family then assigns it the next available tag, 3. It is edited at t_1, retaining the same tag. At t_2, the original file is copied along with its tag to a new location, catalog ID $\#e4$. As the tag 3 does not correspond to its known catalog ID, Family assigns the file a brand new tag, 4, to this new file. This change is undertaken at t_3. As well as updating this tag, Family may now (at a higher level) create evidence linking these two unique states of both files. The original file remains unchanged.

3.4 Algorithm

Our algorithm is listed below. It interprets each file system event (each falling into one of the categories and/or sub-categories above) individually and attempts to correctly construct nodes and derive relationships through an ongoing supply of these events.

For each observed event, we:

- Attempt to map the *focus* of this event (a *file*) to a *node* that is already known
    i. Look up the node that this focus file maps to, through the node identifier stored as an extended attribute on this focus file itself
    ii. If no identifier was found, or its corresponding node was not found, we have not mapped this file

iii. Otherwise, the 'correctness' of this mapping is determined by whether our known properties (e.g. file size, modified time) match the properties found on the focus file

- Use this mapping to determine the most correct *node* to work on
  - If the focus file already maps to a node, and this mapping is 'correct'
    i. Do not update or change existing nodes
  - If the mapping is 'incorrect':
    i. Create new node – mark the focus file as corresponding to this node through an extended attribute
    ii. Link this new node to the previously mapped node through new evidence

    This evidence will either imply that the file has been *copied* (a relationship between files), or simply *updated* (a temporal relationship representing unique states over time).
  - If the focus file is not mapped at all:
    i. Create a new, unrelated node with no evidence – again, mark the focus file as this node through an attribute
- Apply evidence relevant to the event type

  While many events are essentially advisory, and do not imply any specific evidence we should create, there are several key types of events that the algorithm behind Family will look for.

  i. The deletion event, whether explicit or caused by a rename

  Evidence will be submitted to the target node marking it as removed. In the case of being overwritten, evidence will be submitted to the file being renamed indicating that at this point in time, it was responsible for the demise of the now deleted node.

Chapter 4

Implementation

Family is primarily implemented as a user-space Python tool, named Lighthouse, which runs on Mac OS X Leopard [5]. It sources various low-level information from its current environment, such as system calls, and applies the algorithm described in Chapter 3 in order to identify and represent relationships between unique file states.

4.1 Kernel Extension

Lighthouse relies heavily on a Mac OS X kernel extension, named `deveye` and written in C, developed by the author of this thesis specifically for this task. Further information about the public interface to this extension is available in Appendix A. It interfaces with fsevents [2] to provide real-time information about all file changes on a Mac OS X system, along with a custom-built interface to KAuth [3] which provides real-time information about current file usage (including notification of file reads).

An additional feature of `deveye` is that it allows user-relevant access to this information, as normally fsevents and KAuth are system-level devices only accessible by the superuser. This information is revealed through a public API, implemented through `ioctl()`, that may be called by all users on the system to find events relevant to that specific user only – without any special privileges. So, while the kernel driver itself must still be installed with superuser privileges, once it has been installed any application run by a unprivileged user may use its output – including the program written for this thesis which runs completely in user-space.

Again, the installation and usage of this extension is briefly detailed in Appendix A. It is a new and useful software tool which has considerable value for exploring new ways to support user information. For example, it has served as a foundation for another Honours thesis project in 2008 which aims to support users in taking an activity-based view of their file system.

4.2 Lighthouse

Firstly, the Python implementation of Lighthouse uses a small C to Python bridge known as `eyedropper`. This simply provides information from the kernel extension, accessible through `ioctl()`, in a Python-like form for rapid development. Lighthouse is almost a direct implementation of the algorithm described in Chapter 3. It functions by parsing each system call, determining the relevant focus *file* and mapping this to a relevant *node* representing that file's unique state. It also embraces the three categories of events to determine the most appropriate relationships to create. These relationships are then stored using SQLObject [14] inside a small file-based database, available for any further tools to examine and parse.

Most notably, this implementation does not yet correctly support the conceptual model of 'copy and paste', or the operation of 'save as'. This is due to our limited time during our research into the area, and the vast many ways applications attempt to achieve these simple goals. Much research into enhancing PIM focuses on specific applications, such as popular modern office suites. However, we have been hesitant to analyse specific applications as it – in the long term – severely limits the focus of any research.

4.3 Testing Framework

We have also developed a Python tool which uses the data produced by Lighthouse in a resolution process designed to provide broad information about the *ancestors* and *descendants* of any specified file. This tool is a simple, publicly available user interface to what Lighthouse generates, and it may also be used as the basis for our testing harness which will be described in the next chapter.

In terms of our formalisms, this tool attempts to map a given *file* to a current, active target *node*. This *node* represents the current state of the given *file*. It then resolves the output of $Anc(X)$, $Dec(X)$ and $Family(X)$ by inspecting the available evidence attached this *node*, and the further *nodes* this evidence leads to, etc. The *nodes* found through this process are then filtered to include only those which are the most active, and which currently exist – that is, *nodes* which map to *files* that a user can currently access.

This mapping example is visualised in Figure 4.1. The *files* on the left of the figure indicate items that the user has current access to. These are the user's files: **Notes**, **Report** and **Abstract**, shown as clipped squares. For the example, the file **Notes** has three *nodes* over time, the first version at t_0, the next at t_1, and the most recently changed version of the node was created at t_5. This latest version corresponds to the user-accessible file, the last time in the figure.

Similarly, the file **Report** has three *nodes* over time – at t_2, t_3 and t_6. **Abstract** also has unique versions at t_4, t_6 and t_8.

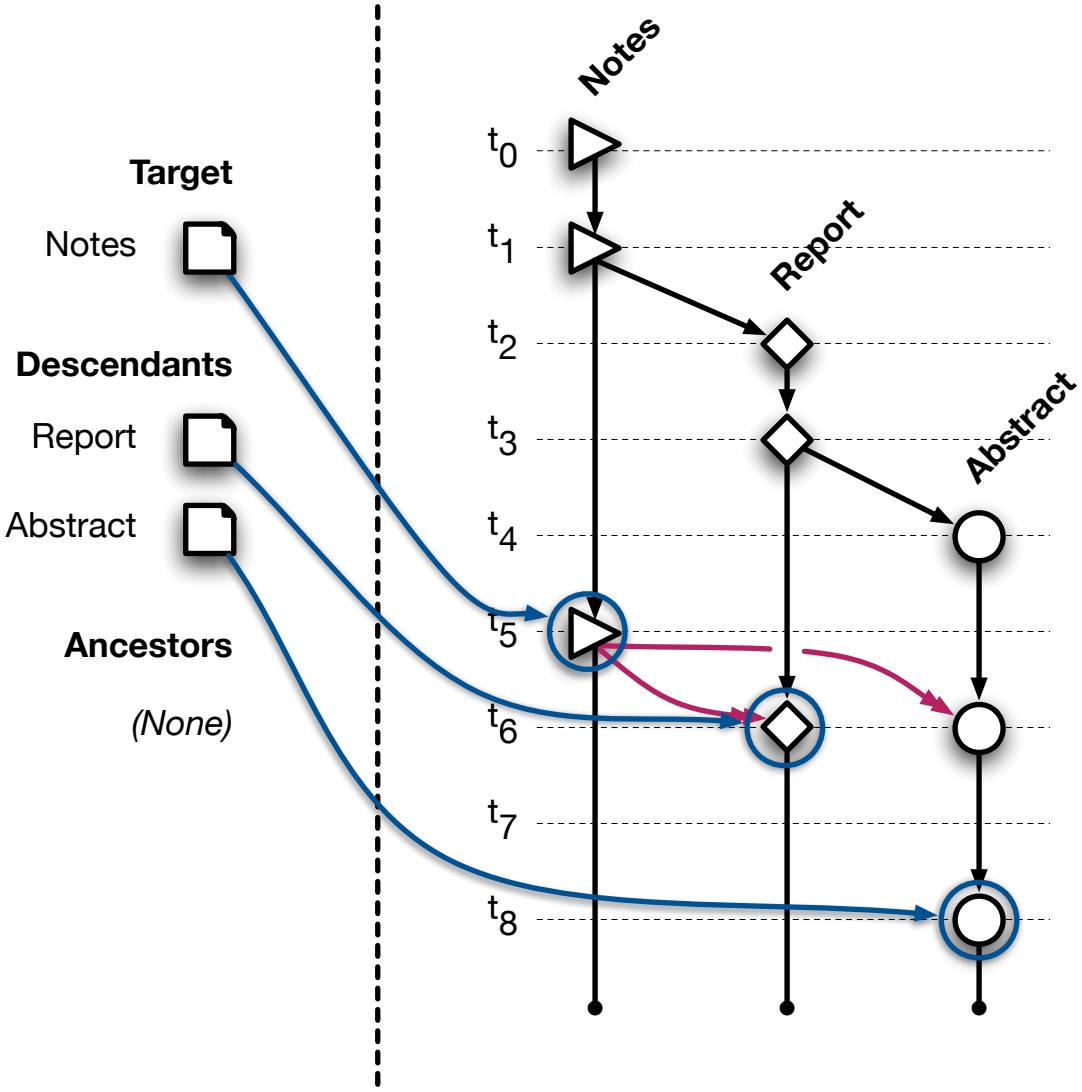


Figure 4.1: Mapping files to nodes, as well as finding their ancestors and descendants.

4.4 Walkthrough

In Figure 4.2, we demonstrate a simple example of how the testing framework is able to reveal a simple file copy, through a small walkthrough in the Mac OS X [5] Terminal. In this example, **TestFile** is created (creating a brand new node) and then a small amount of content is appended to it (creating an updated state of **TestFile**, and therefore another node). The file is then copied to **TestFile-New**. We then run our testing framework on one of the files within the scenario, revealing every node, including the original state of **TestFile** – which is now marked as destroyed, as it has been replaced with a more current state.

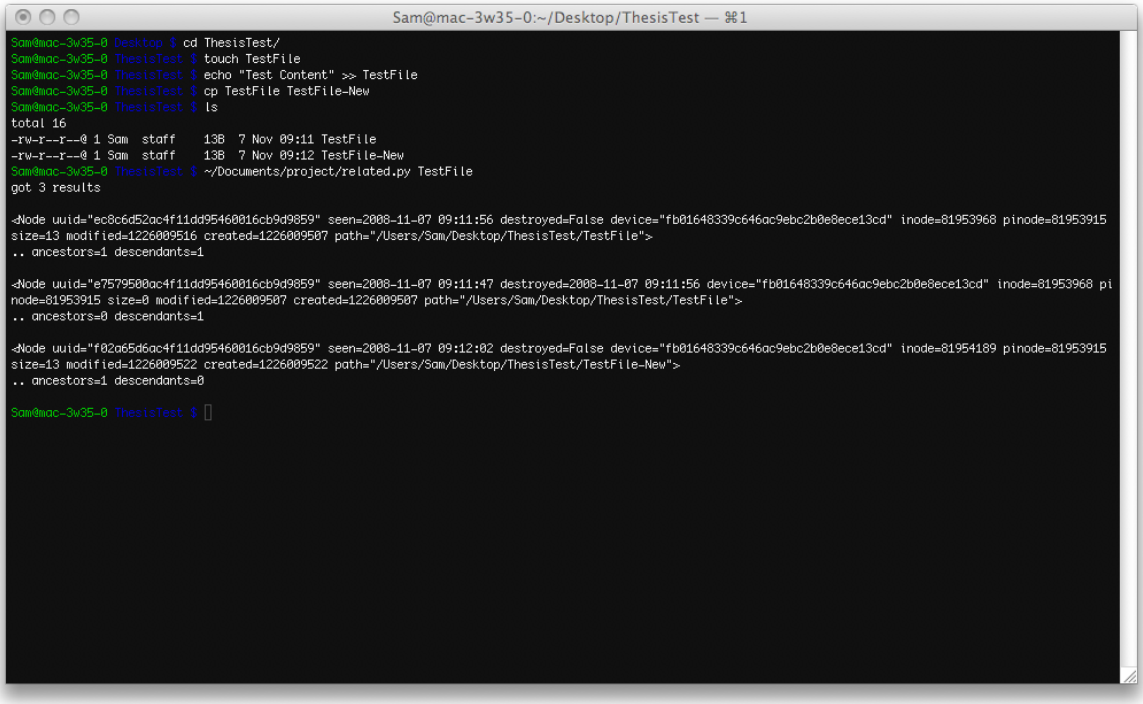


Figure 4.2: Walkthrough of the testing framework

Chapter 5

Evaluation

We evaluate Family by performing a series of synthetic tasks and comparing their output to the correct output, in terms of *ancestors* and *descendants*, and hence the overall notion of Family. By this process, we evaluate the correctness of our algorithm and implementation.

We also evaluate our implementation of Family in terms of system runtime performance, in two key areas. Firstly, we determine the ongoing baseload cost of capturing and processing events in real-time. Secondly, we analyse the time it takes from when a user performs a conceptual action to when this action is correctly mapped in a way that it can be later retrieved. These need to be fast since they constitute an ongoing background cost for this approach.

We do not assess the cost in 'looking up' family relationships – while lookups may be relatively costly, they are provided purely as a benefit to end-users. Thus, users only incur cost when they are interactively attempting to find more information about their files, a cost they may well expect. However, manual testing as shown this cost to be on average less than one second for the vast majority of look ups.

5.1 Correctness

To determine the correctness of Family, we designed and implemented a testing harness. We designed a series of tasks, utilising different application and operating circumstances, and define each as a concrete test. These tests were then be performed in terms of the previously defined formalisms, $Anc(X)$, $Dec(X)$ and $Family(X)$. More explicitly, we aimed to assess whether for each node, z, of our test set, that $Anc(z)$, $Dec(z)$ and $Family(z)$ return nodes that fit a set of proposed, expected output.

Single case tests

We designed a series of tests that are pure tests of the key elements of our conceptual model, each based on a minimal set of files undergoing a single unique action one or many times.

While some of these tests may operate on similar file names, we conduct the tests with a 'clean slate' at the beginning of each.

1. Copy using Mac OS X Finder [5]; **Notes** to **Report**

   This is a fairly simple case of duplicating a file. Much of the motivation and development behind Family has come from the understanding of this core relationship. As in our 'copy' conceptual model, this action defines the newer file as a descendant of the older, and the older file as an ancestor of the newer – the older file was used to create the newer file.

2. Copy folder using Mac OS X Finder; **NETS.key** to **Dup.key**

   In this case, all files within one folder should correspond correctly to their counterparts in the other. **Dup.key** files should match their counterparts in **NETS Presentation.key** as their ancestors, and vice versa. This is an extended version of the first test, designed to evaluate whether location proximity and temporal proximity (as all files will be copied at a similar time) will not affect the correct outcome.

3. Copy and paste using TextEdit [5]; **Notes** to **Report**

   This test should result in identifying **Report** as now descendant from **Notes**, while **Notes** is now a new ancestor of **Report**. In many ways, this is similar to our base case of copying files, except in terms of specific content rather than a file concept as a whole. This is important as it represents one important and open way that files are created and manipulated within applications. For instance, users must be able to create a document either by copying it in an operating system interface (as per our first test), but they may be equally as happy to copy an entire file's content into a new window and save that. From an end-user's point of view, they may see no conceptual difference in what they are aiming to achieve, even though this test may be much more complex than a simple file copy operation.

4. Save as using SubEthaEdit [23]; **core.c** to **core-v2.c**

   This tests another conceptual action which – while implemented in a more complex way – is potentially no different from the case where an end user than performs a simple file copy. Regardless, it has slightly more breadth than a simple copy. A user may make changes before saving a new copy of the previous file.

5. Save using SubEthaEdit; **Report**

   We will expect to see many states of **Report** over time, corresponding to every time the file is saved to disk. The ability to correctly identify this relationship forms the basis for all temporal information between nodes in Family. It is therefore extremely useful to confirm that this relationship is created successfully. This matches the second conceptual model, listing changes over time.

6. Save using `vim`; **core.c**

   As the command-line tool `vim` interacts with most file systems in a very unique way, it is a worthy candidate to test. However, for all other purposes, `vim` will function in the same way as the above test on SubEthaEdit.

7. Backup using `cp`; **Archive.tar** to multiple locations

   Using the command-line utility `cp`, we will determine that all backups from one source file correctly include the source file as one of their ancestors. While this test is simply an expanded version of the first copy test, this is important as it ensures that the broad notion of Family is maintained. In this case, our original file must have a number of descendants all directly attached to its node.

8. External changes; **Birdie.jpeg**

   We describe this simple external example as an expanded form of saving files over time. This test involves a file – located locally on a USB stick – being taken from the system and having changes made to it. In the case of **Birdie.jpeg**, a photo, we might expect the image to be touched up, or cropped etc. We want to be able to determine that the file has undergone changes, and represent this in a temporal fashion, but we are not concerned with every specific change.

We summarise these single case test results in the first three columns of Table 5.1. This table shows each test, along with the application that was used to perform it. It then describes every file that should, or otherwise is, related to this task – including the file that could be either described as *source* or *destination*. More technically, we also show each unique state of each file, with the 'current' and available version indicated in bold. For instance, in the case of saving an existing file, we represent the previous unique states of both **Report** and **core.c** as unbolded listings of the same file name.

Most importantly, Table 5.1 shows the both the expected and actual count of *ancestors* and *descendants* that are attached to each node. While these, of course, link to other nodes, these are correct within the bounds of each test – that is to say, the descendants and ancestors listed for each node only reference other nodes within the same test.

Results

In terms of actual results, all but the 3rd and 4th single case tests have been successful. Our expected set of ancestors and descendants are correct for all cases except that of copy and paste, and save as. While this relationship would be important for the complete success of the Family algorithm, as discussed within Chapter 4, the implementation does not currently extend to supporting this kind of conceptual model.

5.2 Performance

In terms of performance, Family runs as a subtle and ongoing background process, only undertaking activity as a result of other user actions. In this sense, it is difficult to perform a performance analysis - Family involves several processes and the costs are incurred as an effect of each user action, as well as other system background actions. We have allowed Family, as well as the low-level kernel extension, *deveye*, to run in parallel with our normal activity of an iMac 2.16ghz Core 2 Duo, with 2gb RAM and there has been no noticable degradation in the responsiveness of the system under a diverse range of desktop activities.

Test	Application	File	Expected		Results		Tot.
			Anc.	*Dec.*	*Anc.*	*Dec.*	
Copy	Finder	**Notes**	0	1	0	1	2/2
		Report	1	0	1	0	
Copy Folder	Finder	**NETS.key**	0	1	0	1	6/6
		NETS.key/PkgInfo	0	1	0	1	
		NETS.key/Index.gz	0	1	0	1	
		Dup.key	1	0	1	0	
		Dup.key/PkgInfo	1	0	1	0	
		Dup.key/Index.gz	1	0	1	0	
Copy & Paste	TextEdit	**Notes**	0	1	0	0	0/2
		Report	1	0	0	0	
Save As	SubEthaEdit	**core.c**	0	1	0	0	0/2
		core-v2.c	1	0	0	0	
Save Existing File	SubEthaEdit	Report	0	1	0	1	4/4
		Report	1	1	1	1	
		Report	1	0	1	0	
Save Existing File	`vim`	core.c	0	1	0	1	4/4
		core.c	1	1	1	1	
		core.c	1	0	1	0	
Multiple Copies	`cp`	**Archive.tar**	0	5	0	5	10/10
		Others (x5)	1	0	1	0	
External Changes	*n/a*	Birdie.jpeg	0	1	0	1	2/2
		Birdie.jpeg	1	0	1	0	

Figure 5.1: Single case test results

Baseload cost of Family is shown in Figure 5.2. Broadly, this shows no substantial difference in three different environments. The first environment represents the base load environment, without any component of Family. The second environment includes the kernel extension, `deveye`, along with its support utility. The last environment also includes the user-space tool Lighthouse. The only notable feature of any environment is the tendency for the complete Lighthouse tool to present a greater standard deviation in its results.

This test suite was contributed by Greg Darke.

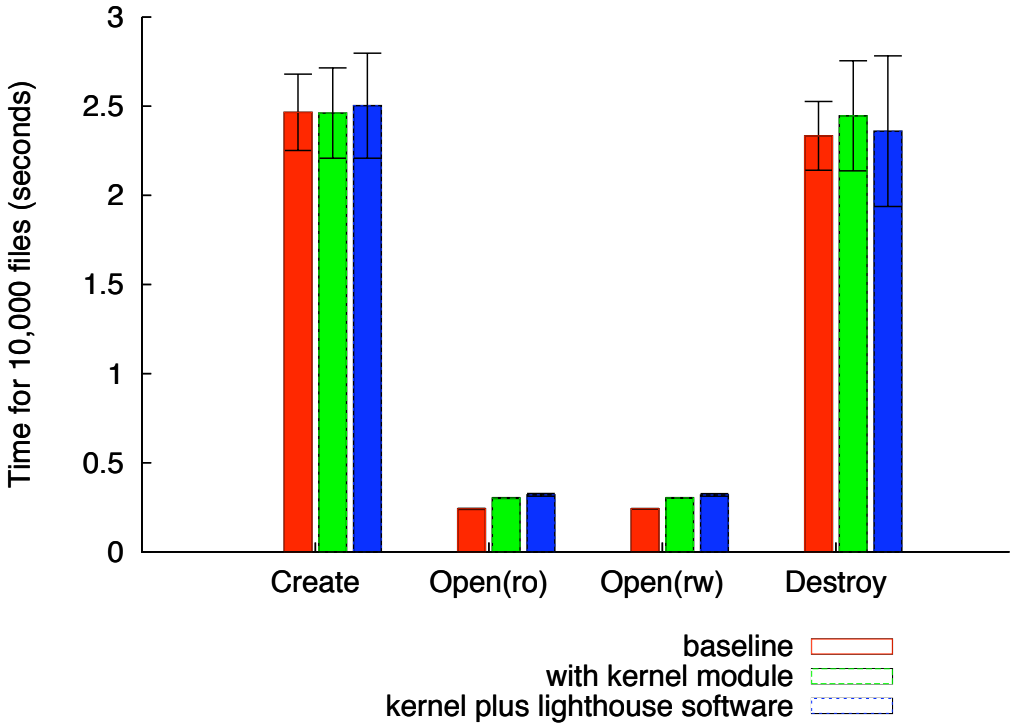


Figure 5.2: Timing for traditional file system actions

We also performed short analysis of the time taken, reported by Lighthouse, to process sets of various actions. We show this in Figure 5.3. This is the total time it takes Lighthouse to correctly parse each system call correctly – this does not count any time that it sits idle waiting for further events. As these tests were performed on a live system, the total number of system calls parsed during the time the formal test was being conducted may be much higher than that of the actions taken, due to other background processes. Despite the relative high speed of processing single events, approximately 60% of all work involved in Lighthouse is taken up by database usage, something that could very easily be optimised in the future.

Action Type	Action Count	Internal Events	All Overhead		Database Only	
			Time	Per Action	Time	Per Action
Copy	500	1284	8.5318s	0.0171s	5.3114s	0.0106s

Figure 5.3: Time taken for correct mapping

Chapter 6

Conclusion

6.1 Contributions

This thesis has explored and contributed a new and novel way to help users find files that are related to each other because of their common ancestry or usage patterns. This is valuable as these relationships may be important to help users find the right file for a particular task, or when an activity calls for key related files that were used to create an existing file.

The key contributions of the thesis include the introduction of a new notion, which aims to capture relationships that exist in a user's mental model but are totally ignored in current operating systems. We have established a formal description of this user notion of a Family, in terms of the concepts, *ancestors*, *descendants* and unique states – or *nodes* – of all files. We have also demonstrated that it is practical to correctly capture many forms of conceptual models, and classify these relationships in terms of *ancestors* and *descendants*. We have also contributed an implementation and test cases for this implementation. Lighthouse is a functional and usable tool that provides a local database, able to be queried. And it is implemented – as originally intended – as an implicit and transparent background tool. The performance of the system behaviour while Lighthouse is running is also negligible versus a base case. Thus it captures data with no ongoing cost to end users, providing information when users or upstream applications request it.

6.2 Future Work

This work provides foundations for a whole new field of research into file access interfaces. This will include user studies, determining the usefulness of having this *ancestor* and *descendant* information accessible in a easy-to-use and transparent fashion.

One broad conclusion that we have reached is that operating systems in general need to have a better conceptual understanding of user *intent*. This can be described as a 'bottom-up' approach to problem solving. Many typical actions on a machine are so implicit, so everyday, that it seems extreme to say that the internals of an operating system should be able to

have a handle on how or *why* they are performing a certain action. This has implications not only for attempting to observe a modern system, but also for security and other areas.

For example, the distinction between how a user loads a program could be extremely useful in providing relevant warnings or alerts to end-users. A certain level of security rating could be applied to a program if a user physically uses a mouse click to start it, or explicitly types it from the command line. However, the same level of security might not be guaranteed if the same program was started by a potentially malicious background process.

A similar analogy can be applied to our work. We aim to analyse events 'after-the-fact', or from a 'top-down' approach – which, after this thesis, should be understood as a substantial effort. If applications could inform the operating system about the simple, conceptual actions they intend to take, then this higher-level conceptual information could be mined with ease. Instead, these same applications may now embrace quirks and intricacies in order to achieve their goals, such as 'guaranteed' atomic file saves. However, the real issue is that nearly every application embraces the *same* quirks. If operating systems were updated to provide 'quirk-free' interfaces (even if these interfaces were eventually implemented in a quirky way), then information about calls to these interfaces could be analysed – a more 'top-down' approach.

6.3 Conclusions

Existing file systems fail to support users in identifying important relationships between their own documents that they may hold within a personal mental model. It is not straightforward to create a system that can recognise these relationships. We have explored the algorithm and architecture of Family, as well as a real-world implementation in Lighthouse, which aims to work towards this goal.

Bibliography

[1] Personal Information Management: PIM 2008 Workshop. http://pim2008.ethz.ch/, 2008.

[2] Apple, Inc. File System Events Programming Guide: Introduction. http://developer.apple.com/documentation/Darwin/Conceptual/FSEvents_ProgGuide/Introduction/chapter_2_section_1.html.

[3] Apple, Inc. Technical Note TN2127: Kernel Authorization. http://developer.apple.com/technotes/tn2005/tn2127.html.

[4] Apple, Inc. Mac OS X Leopard - Features - Time Machine. http://www.apple.com/macosx/features/timemachine.html, 2007.

[5] Apple, Inc. Apple – Mac OS X Leopard. http://www.apple.com/macosx, 2008.

[6] S. Balasubramaniam and B.C. Pierce. What is a file synchronizer? *Proceedings of the 4th annual ACM/IEEE international conference on Mobile computing and networking*, pages 98–108, 1998.

[7] L. Blunschi, J.P. Dittrich, O.R. Girard, S.K. Karakashian and M.A.V. Salles. A dataspace odyssey: The iMeMex personal dataspace management system. *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2007.

[8] R. Boardman and M.A. Sasse. "Stuff goes into the computer and doesn't come out": a cross-tool study of personal information management. *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 583–590, 2004.

[9] CollabNet. SCM Category Home. http://scm.tigris.org/, visited 2008/09/28.

[10] Fernando J. Corbató and V. A. Vyssotsky. Introduction and Overview of the Multics System. *IEEE Ann. Hist. Comput.*, Volume 14, Number 2, pages 12–13, 1992.

[11] R. Cox and W. Josephson. File Synchronization with Vector Time Pairs. 2005.

[12] Robert C. Daley and Peter G. Neumann. A general-purpose file system for secondary storage. pages 138–166, 2000.

[13] David Dearman and Jeffery S. Pierce. "It's on my other computer!": computing with multiple devices. In Mary Czerwinski, Arnold M. Lund and Desney S. Tan (editors), *CHI*, pages 767–776, Florence, Italy, April 5-10 2008. ACM.

[14] Ian Bicking. SQLObject. http://www.sqlobject.org/, 2008.

[15] William Jones, Ammy Jiranida Phuwanartnurak, Rajdeep Gill and Harry Bruce. "Don't take my folders away!": organizing personal information to get things done. In Gerrit C. van der Veer and Carolyn Gale (editors), *CHI Extended Abstracts*, pages 1505–1508. ACM, 2005.

[16] David R. Karger and William Jones. Data unification in personal information management. *Commun. ACM*, Volume 49, Number 1, pages 77–82, 2006.

[17] DR Karger, W. Jones, O. Bergman, W. Pratt and M. Franklin. Towards a unification & integration of PIM support. *Jones, W., Bruce, H.: A Report on the NSF-Sponsored Workshop on Personal Information Management, Seattle, WA*, Volume 2005, pages 30–33, 2005.

[18] Matt Mackall. Mercurial. http://www.selenic.com/mercurial/wiki/.

[19] M. Mahalingam, C. Tang and Z. Xu. Towards a semantic, deep archival file system. *Distributed Computing Systems, 2003. FTDCS 2003. Proceedings. The Ninth IEEE Workshop on Future Trends of*, pages 115–121, 2003.

[20] Brent Rector. Introducing Longhorn for Developers, Chapter 4: Storage. http://msdn.microsoft.com/en-us/library/aa479870.aspx.

[21] Dennis M. Ritchie and Ken Thompson. The UNIX time-sharing system. *Commun. ACM*, Volume 17, Number 7, pages 365–375, 1974.

[22] Craig A. N. Soules and Gregory R. Ganger. Connections: using context to enhance file search. *SIGOPS Oper. Syst. Rev.*, Volume 39, Number 5, pages 119–132, 2005.

[23] The Coding Monkeys. SubEthaEdit. http://www.codingmonkeys.de/subethaedit, 2008.

Appendix A

Kernel Extension

The kernel extension developed for this project is known as `deveye`, and aims to coalesce various sources of raw data from within Mac OS X into a common interface accessible by any process running in userspace. This raw data represents the reporting of system calls as well as some particular interpretations of system calls. For instance, `deveye` is able to report a `close()` that occurs on any particular file, but only if that file has been modified during its time open (`SFE_MODIFIED`).

API

Installation

`deveye` is distributed as a single Mac OS X kernel extension, `deveye.kext`, along with a configuration file for the Mac OS X system manager, LaunchD. The kernel extension should be placed in `/System/Library/Extensions`, and the owner of the extension and its contents should be changed to 'root:wheel'. Lastly, the configuration file, `com.moofco.deveye.plist` should be placed in `/Library/LaunchDaemons`. It should be loaded into the system by invoking `sudo launchctl load /Library/LaunchDaemons/com.moofco.deveye.plist`.

This will immediately start `deveye`, and reveal its public interface at `/dev/eyefs`. This API may now be used by user-space applications.

Usage

This section describes the interface for `deveye`. First, it is worth noting that `event.h` and `deveye.h`, the public header files for `deveye`, are available following. However, a short usage example is included here regardless.

Generally, an application must first `open()` the `deveye` device node at `/dev/eyefs`. A standard request structure should then be created, and allocated a static buffer as an output

target, before the entire request passed to `ioctl()`. A full example, `eyerender`, which simply displays all events generated within the kernel extension, is included on the following page.

com.moofco.deveye.plist

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE plist PUBLIC -//Apple Computer//DTD PLIST 1.0//EN
http://www.apple.com/DTDs/PropertyList-1.0.dtd >
<plist version='1.0'>
<dict>
<key>Label</key>
<string>com.moofco.deveye</string>
<key>Debug</key>
<false />
<key>ProgramArguments</key>
<array>
    <string>/System/Library/Extensions/deveye.kext/eyemgr</string>
</array>
<key>KeepAlive</key>
<true />
<key>RunAtLoad</key>
<true />
</dict>
</plist>
```

eyerender.c

```

/**
 * eyerender.c - a quick-and-dirty user-space tool to print output from the device node
 */

#include <errno.h>
#include <fcntl.h>
#include <grp.h>
#include <libproc.h>
#include <pthread.h>
#include <pwd.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ioccom.h>
#include <sys/ioctl.h>
#include <sys/param.h>
#include <sys/stat.h>
#include <sys/sysctl.h>
#include <sys/types.h>
#include <sys/uio.h>
#include <unistd.h>
#include "event.h"
#include "deveye.h"

int get_eyefs();

/**
 * Core eyemgr code.
 */
int main() {

    uint8_t output[SFE_MAXLEN];
    int eyefs, status;
    struct eyefs request;
    struct sfe *outputp = ( struct sfe * ) output;
    struct argsfe *arg;

    memset( ( void * ) &request, 0, sizeof( struct eyefs ) );

    // grab eyefs fd
    if( ( eyefs = get_eyefs() ) < 0 ) {
        fprintf( stderr, "couldn't open eyefs (%d) - is the kext loaded?\n", eyefs );
    }

    // poll for events from eyefs
    for( ;; ) {

        request.size = SFE_MAXLEN;
        request.version = EYEFS_VERSION;
        request.store.ptr = ( void * ) output;

        if( status = ioctl( eyefs, EYEFS_GET, &request ) )  {
            break;
        }

        printf( "event (last=%lld):\n", request.last );
        printf( " -- uid=%d eid=%lld args=%d", outputp->uid, outputp->eid,
            SFE_NUMARGS( outputp->event ) );
        if( outputp->event & SFE_MOD ) {
            printf( " MOD" );
        }
        if( outputp->event & SFE_FSEVENTS ) {
            printf( " FSEVENTS" );
        }
        printf( "\n" );

        if( arg = SFE_ARG0( outputp ) ) {
            printf( " -- arg0 dev=%d uuid=%llu.%llu ino=%d\n",
                arg->dev, *( ( uint64_t * ) &arg->uuid ),
                    *( ( uint64_t * ) ( ( ( char * ) &arg->uuid ) + 8 ) ),
                        arg->ino );
            printf( "    %s\n", arg->path );
        }

        if( arg = SFE_ARG1( outputp ) ) {
            printf( " -- arg1 dev=%d uuid=%llu.%llu ino=%d\n",
                arg->dev, *( ( uint64_t * ) &arg->uuid ),
                    *( ( uint64_t * ) ( ( ( char * ) &arg->uuid ) + 8 ) ),
                        arg->ino );
            printf( "    %s\n", arg->path );
        }

    }

    // fall out
    fprintf( stderr, "broke out: status=%d, errno=%d\n", status, errno );
    return status;

}

/**
 * Connect to the eyemgr device node.
 */
int get_eyefs() {

```

```
    // is it really not more complicated than this?
    return open( "/dev/" EYEFS_DEV, 0 );

}
```

event.h

```

/**
 * event.h - provides the public and private (kernel) definitions for the sfe kext
 */
#ifndef __EVENT_H
#define __EVENT_H

#include <sys/ioccom.h>
#include <sys/param.h>
#include <sys/vnode.h>
#include <sys/kernel.h>
#include <stdint.h>

/**
 * UUID typedef - typically included out of kernel, but not available in kernel.
 */
#ifdef KERNEL
typedef unsigned char uuid_t[16];
#endif

/**
 * Header for a single dynamic-length event (24 bytes).
 */
struct sfe {
    uint64_t eid;
    uid_t uid;
    gid_t gid;
    pid_t pid;
    uint32_t event;
};

/**
 * Arguments for the event (24 bytes). The actual size of this struct will be
 * inclusive of the path length, which should be 8-byte aligned.
 */
struct argsfe {
    uuid_t uuid;
    dev_t dev;
    ino_t ino;
    uid_t uid;
    gid_t gid;
    int32_t mode, length;
    char path[0];
};

/**
 * Internal helper methods for the usage of events.
 */
#ifdef KERNEL
int eventsetup();
int eventshutdown();
int uuid_submit( struct argsfe * );
int uuid_fetch( struct argsfe * );
int queue_push( struct sfe * );
int queue_push_user( user_addr_t, int );
int event_vnode( kauth_cred_t, kauth_action_t, vfs_context_t, vnode_t, vnode_t );
int event_fileop( kauth_cred_t, kauth_action_t, vfs_context_t, vnode_t );
int event_obtain( uid_t, uint64_t, uint8_t, struct sfe * );
int event_obtain_user( uid_t, uint64_t *, uint8_t, user_addr_t, int * );
#endif

/**
 * The maximum length of any single event.
 */
#define SFE_MAXLEN ( sizeof( struct sfe ) + 2 * ( sizeof( struct argsfe ) + MAXPATHLEN ) )

/**
 * Definitions to return the total size of a whole event along with the total
 * size of a single argument.
 */
#define SFE_ARGSIZE(x) ( x ? (sizeof(struct argsfe)+((struct argsfe*)x)->length) : 0 )
#define SFE_ARG0(v) ( \
    ( (v)->event & SFE_ARG  ) == SFE_ARG  ? \
    ((struct argsfe*)(((void*)(v))+sizeof(struct sfe))) : 0 \
)
#define SFE_ARG1(v) ( \
    ( (v)->event & SFE_ARGS ) == SFE_ARGS ? \
    ((struct argsfe*)(((void*)SFE_ARG0(v))+sizeof(struct argsfe)+SFE_ARG0(v)->length)) : 0 \
)
#define SFE_SIZE(v) ( \
    ( ( (v)->event & SFE_ARG  ) == SFE_ARG  ? SFE_ARGSIZE(SFE_ARG0(v)) : 0 ) + \
    ( ( (v)->event & SFE_ARGS ) == SFE_ARGS ? SFE_ARGSIZE(SFE_ARG1(v)) : 0 ) + \
    ( sizeof( struct sfe ) ) \
)

/**
 * Define high-byte properties (part of events).
 */
#define SFE_MOD         0x0100
#define SFE_ARG         0x0200
#define SFE_ARGS        0x0600
#define SFE_MERGED      0x1000
#define SFE_FSEVENTS    0x2000
#define SFE_PRIVATE     0x4000

/**
 * Generates the number of arguments from an event field.
 */
#define SFE_NUMARGS(v)  ((v & SFE_ARGS) == SFE_ARGS ? 2 : ((v & SFE_ARG) ? 1 : 0))

/**
 * Event type definitions. The event field may be masked with 0xff to find
 * specific event types.
 */
#define SFE_OPEN        ( 0x01 | SFE_ARG ) // open notification
#define SFE_READ        ( 0x02 | SFE_ARG ) // authorisation to read
#define SFE_EXEC        ( 0x03 | SFE_ARG ) // authorisation to execute/browse
#define SFE_WRITE       ( 0x04 | SFE_ARG ) // authorisation to write (i.e. no changes have been made yet)
#define SFE_CREATE      ( 0x05 | SFE_MOD | SFE_ARG )
#define SFE_DELETE      ( 0x06 | SFE_MOD | SFE_ARG )
#define SFE_STAT        ( 0x07 | SFE_MOD | SFE_ARG ) // file stats changed
#define SFE_RENAME      ( 0x08 | SFE_MOD | SFE_ARGS ) // if the second argument has a uid/gid, it has been replaced
#define SFE_MODIFIED    ( 0x09 | SFE_MOD | SFE_ARG ) // dirty close
#define SFE_EXCHANGE    ( 0x0a | SFE_MOD | SFE_ARGS )
#define SFE_FINDER      ( 0x0b | SFE_MOD | SFE_ARG ) // finder info changed
```

```
#define SFE_MKDIR       ( 0x0c | SFE_MOD | SFE_ARG )
#define SFE_CHOWN       ( 0x0d | SFE_MOD | SFE_ARG )
#define SFE_MOUNT       ( 0x0e | SFE_PRIVATE | SFE_ARG )

#endif/*__EVENT_H*/
```

deveye.h

```

/**
 * deveye.h - provides simple definitions for user-space to communicate with us
 */
#ifndef __DEVEYE_H

#include <sys/ioccom.h>
#include <sys/types.h>
#include <stdint.h>

/**
 * Global version, device name created in /dev, along with xattr definitions.
 */
#define EYEFS_VERSION    2
#define EYEFS_DEV       "eyefs"
#define EYEFS_TLD       "com.moofco.deveye"
#define EYEFS_UUID      "uuid"


/**
 * Request class for access to our standard device (EYEFS_DEV). Includes an
 * internal union for 32-64 transition between the real world and kernel land.
 */
struct eyefs {
    int16_t version, count;
    int32_t size;
    uint64_t last;
    union {
        void *ptr;
        user_addr_t addr;
        int32_t old_addr;
    } store;
};

/**
 * Device ioctl() request actions.
 *  - GET: waits and returns events
 *  - POLL: same as GET, but will return immediately if there are no pending events
 *  - PUSH: used internally by `eyemgr` to submit fsevents
 */
#define EYEFS_GET       _IOWR( 's', 1, struct eyefs )
#define EYEFS_POLL      _IOWR( 's', 2, struct eyefs )
#define EYEFS_PUSH      _IOW( 's', 3, struct eyefs )

#endif/*__DEVEYE_H*/
```

Internals

We have also included the internal source code for deveye. The vast majority of the init/-configuration code is within `deveye.c`, with much of the conceptual storage and list code offloaded into `event.c`.

deveye.c

```
1
2   /**
3    * deveye.c - provides the core outwards controls/functions for the kext.
4    */
5
6   #include "deveye.h"
7   #include "event.h"
8
9   #include <libkern/OSAtomic.h>
10  #include <libkern/libkern.h>
11  #include <miscfs/devfs/devfs.h>
12  #include <sys/conf.h>
13  #include <sys/kauth.h>
14  #include <sys/kernel.h>
15  #include <sys/systm.h>
16  #include <sys/unistd.h>
17  #include <sys/vnode.h>
18  #include <sys/errno.h>
19
20  static kauth_listener_t gListener = NULL;
21  static SInt32 gActivationCount = 0;
22  static SInt32 gDevInstalled = -1;
23  static void *gDevNode = NULL;
24
25  /**
26   * Standard device switch open/close implementation.
27   */
28  static int dev_opcl(
29      __unused dev_t dev,
30      __unused int flag,
31      __unused int mode,
32      __unused struct proc *p ) {
33      return 0;
34  }
35
36  /**
37   * Handles all user-space calls to ioctl, providing events either as a
38   * 'get' or 'poll' operation.
39   */
40  static int dev_ioctl(
41      __unused dev_t dev,
42      u_long cmd,
43      caddr_t data,
44      __unused int flag,
45      struct proc *p ) {
46
47      int status = EIO;
48      kauth_cred_t cred;
49      uid_t uid;
50      struct eyefs *request = ( struct eyefs * ) data;
51      user_addr_t target;
52
53      // discard invalid requests
54      if( request->version != EYEFS_VERSION ) {
55          return ENOPROTOOPT; // protocol not available
56      }
57
58      // increment usage counter, grab cred/uid for validation
59      OSIncrementAtomic( &gActivationCount );
60      cred = kauth_cred_proc_ref( p );
61      uid = kauth_cred_getuid( cred );
62
63      // all a bit strange, but seems to work . . .
64      if( !proc_is64bit( p ) ) {
65          target = request->store.addr;
66      }
67      else {
68  //      target = CAST_USER_ADDR_T( request->store.old_addr );
69          target = request->store.addr;
70      }
71
72      switch( cmd ) {
73      case EYEFS_GET:
74      case EYEFS_POLL:
75          // submit request internally
76          status = event_obtain_user( uid, &( request->last ), cmd == EYEFS_POLL, target, &( request->size ) );
77          if( status ) {
78              request->count = 0;
79          }
80          else {
81              request->count = 1; // for now, we only have a single result
82          }
83          break;
84
85      case EYEFS_PUSH:
86          // this may only be invoked in push
87          if( uid ) {
88              status = EACCES;
89              goto fallout;
90          }
91
92          // FIXME: not yet supported
93          if( request->count != 1 ) {
94              status = E2BIG;
95              goto fallout;
96          }
97
98          // submit request internally
99          status = queue_push_user( target, request->size );
100         break;
101
102     }
103
104 fallout:
105     // release resources, return determined status
106     kauth_cred_unref( &cred );
107     OSDecrementAtomic( &gActivationCount );
108     return status;
109
110 }
111
112 /**
113  * Device switch structure, points only to default open/close constructs, and our major ioctl implementation.
114  */
115 static struct cdevsw gDevSwitch = {
116     dev_opcl, dev_opcl,
117     eno_rdwrt, eno_rdwrt, dev_ioctl, eno_stop, eno_reset,
118     NULL, eno_select, eno_mmap, eno_strat, eno_getc, eno_putc, 0
119 };
120
121 /**
122  * The core kauth handler for vnode operations.
```

```
 */
static int kauth_fileop( kauth_cred_t cred, __unused void *idata,
    kauth_action_t action,
    uintptr_t arg0, uintptr_t arg1, uintptr_t arg2, uintptr_t arg3 ) {

    vnode_t vp = ( vnode_t ) arg0;
    vfs_context_t ctx;
    int uid;

    OSIncrementAtomic( &gActivationCount );
    ctx = vfs_context_create( ( vfs_context_t ) 0 );

    if( action == KAUTH_FILEOP_OPEN && ( uid = kauth_cred_getuid( cred ) ) ) {
        if( event_fileop( cred, action, ctx, vp ) ) {
            // do we care?
        }
    } else if( action == KAUTH_FILEOP_EXEC && ( uid = kauth_cred_getuid( cred ) ) ) {
        if( event_fileop( cred, action, ctx, vp ) ) {
            // do we care?
        }
    }

    vfs_context_rele( ctx );
    OSDecrementAtomic( &gActivationCount );

    return KAUTH_RESULT_DEFER;

}
/**
 * Core kernel extension init function.
 */
kern_return_t deveye_start( kmod_info_t *ki, void *d ) {

    // setup internal event storage
    if( eventsetup() ) {
        printf( "deveye_start: could not eventsetup()" );
        return KERN_FAILURE;
    }

    // create the kauth listener.
    gListener = kauth_listen_scope( KAUTH_SCOPE_FILEOP, kauth_fileop, NULL );
    if( gListener == NULL ) {
        printf( "deveye_start: could not create gListener for vnodes.\n" );
        return KERN_FAILURE;
    }

    // create a new device node
    gDevInstalled = cdevsw_add( -1, &gDevSwitch );
    if( gDevInstalled < 0 ) {
        printf( "deveye_start: could not insert cdevsw\n" );
        return KERN_FAILURE;
    }

    // install the device node in `/dev`
    gDevNode = devfs_make_node( makedev( gDevInstalled, 0 ),
        DEVFS_CHAR, UID_ROOT, GID_WHEEL, 0644, EYEFS_DEV, 0 );
    if( NULL == gDevNode ) {
        printf( "deveye_start: could not make device node\n" );
        return KERN_FAILURE;
    }

    // success!
    printf( "deveye_start: ok\n" );
    return KERN_SUCCESS;

}

/**
 * Core shutdown function.
 */
kern_return_t deveye_stop( kmod_info_t *ki, void *d ) {

    // revoke event setup (has to happen first - kicks the habit of listeners)
    eventshutdown();

    // wait for our activation count to drop
    do {
        struct timespec oneSecond;
        oneSecond.tv_sec = 1;
        oneSecond.tv_nsec = 0;
        msleep( &gActivationCount, NULL, PUSER, "com_moofco_kext_deveye.stop", &oneSecond );
    } while ( gActivationCount > 0 );

    // release the kauth listener
    if( gListener != NULL ) {
        kauth_unlisten_scope( gListener );
        gListener = NULL;
    }

    // release the device node
    if( gDevInstalled > -1 ) {
        cdevsw_remove( gDevInstalled, &gDevSwitch );
        gDevInstalled = -1;
    }

    // uninstall the device node from `/dev`
    if( gDevNode != NULL ) {
        devfs_remove( gDevNode );
        gDevNode = NULL;
    }

    // success
    printf( "deveye_stop: ok\n" );
    return KERN_SUCCESS;

}
```

event.c

```

/**
 * event.c - provides tighter-knit functionality for managing events
 */

#include <libkern/OSBase.h>
#include <libkern/OSAtomic.h>
#include <libkern/libkern.h>
#include <miscfs/devfs/devfs.h>
#include <sys/conf.h>
#include <sys/kauth.h>
#include <sys/kernel.h>
#include <sys/systm.h>
```

```
#include <sys/unistd.h>
#include <sys/vnode.h>
#include <sys/errno.h>
#include "fastlock.h"
#include "event.h"

#ifdef DEBUG
#define dprintf(format, args...) printf(format, ## args)
#else
#define dprintf(format, args...) 0
#endif

#define QUEUEBITS 13
#define QUEUELEN (1<<QUEUEBITS)
#define QUEUEMASK (QUEUELEN-1)

static SInt64 gUniqueEvent = 0;
static SInt32 gQueuePos = 0;
static SInt32 gQueueActive = 0;
static struct {
    fastlock_t lck;
    union {
        struct sfe sfe;
        uint8_t __internal[SFE_MAXLEN];
    } ev;
} gQueue[QUEUELEN];

#define MAPBITS 13
#define MAPLEN (1<<MAPBITS)

static struct {
    uuid_t uuid;
    dev_t dev;
} gDevUUID[MAPLEN];

/**
 * Configure the event queue structure.
 */
int eventsetup() {

    memset( gQueue, 0, sizeof( gQueue ) );
    memset( gDevUUID, 0, sizeof( gDevUUID ) );

    gUniqueEvent = 0;
    gQueuePos = 0;
    gQueueActive = 1;

    printf( "deveye: eventsetup using %d bytes for queue,"
        " %d bytes for dev/uuid (now open for business)\n",
            sizeof( gQueue ), sizeof( gDevUUID ) );

    return 0;

}

/**
 * Notify our listeners that the queue structure is shutting down.
 */
int eventshutdown() {

    gQueueActive = 0;
    wakeup( ( caddr_t ) &gQueue );

    printf( "deveye: eventshutdown ok\n" );

    return 0;

}


/**
 * Hash a device node.
 */
int uuid_devhash( dev_t dev ) {

    int a = ( int ) dev;

    a = a ^ ( a >> 4 );
    a = ( a ^ 0xdeadbeef ) + ( a << 5 );
    a = a ^ ( a >> 11 );
    a = a % MAPLEN;

    return a;

}

/**
 * Device vs. UUID push.
 */
int uuid_submit( struct argsfe *arg ) {

    int i, target = uuid_devhash( arg->dev );

#ifdef DEBUG
    // ensure that this method has been called properly
    if( !arg ) {
        printf( "deveye: uuid_submit without arg\n" );
        return -2;
    }
    dprintf( "deveye: trying to place dev %d into bucket %d\n", arg->dev, target );
#endif

    // find a bucket that this uuid/dev can live in
    for( i = target + 1; ( i % MAPLEN ) != target; ++i ) {
        if( gDevUUID[i % MAPLEN].dev == 0 || gDevUUID[i % MAPLEN].dev == arg->dev ) {
            break;
        }
    }
    i %= MAPLEN;

    // check rollover conditions
    if( i == target ) {
        printf( "deveye: couldn't fit device uuid for dev %d\n", arg->dev );
        return ENOMEM;
    }

    // place the uuid where it should be
    gDevUUID[i].dev = arg->dev;
    memcpy( &( gDevUUID[i].uuid ), &( arg->uuid ), 16 );

    // success!
    printf( "deveye: dev %d[%d] given uuid %llu.%llu\n", arg->dev, i, *( ( uint64_t * ) &arg->uuid ), *( ( uint64_t * ) ( ( ( char *
    return 0;

```

```
}

/**
 * Copy out a UUID if its found.
 */
int uuid_fetch( struct argsfe *arg ) {

    int i, target = uuid_devhash( arg->dev ), status = 0;

#ifdef DEBUG
    // ensure that this method has been called properly
    if( !arg ) {
        printf( "deveye: uuid_fetch without arg\n" );
        return -2;
    }
    dprintf( "deveye: trying to look for dev %d in bucket %d\n", arg->dev, target );
#endif

    // find the bucket containing this dev
    for( i = target + 1; ( i % MAPLEN ) != target; ++i ) {
        if( gDevUUID[i % MAPLEN].dev == arg->dev ) {
            break;
        }
        if( gDevUUID[i % MAPLEN].dev == 0 ) {
            status = EIO;
            goto fallout;
        }
    }
    i %= MAPLEN;

    // check rollover conditions (unlikely, but eh)
    if( i == target ) {
        status = ENOMEM;
        goto fallout;
    }

    // copy the uuid into the output struct
    memcpy( &( arg->uuid ), &( gDevUUID[i].uuid ), 16 );

    // success!
    dprintf( "deveye: found dev %d in bucket %d\n", arg->dev, i );
    return 0;

fallout:
    // fallout condition, reset the uuid to blank
    memset( &( arg->uuid ), 0, 16 );
    return status;

}

/**
 * Internal (single) event push, from kernel memory.
 */
int queue_push( struct sfe *ev ) {

    int32_t pos;

    // handle private events
    if( ev->event & SFE_PRIVATE ) {
        int status = ENOTSUP;

        switch( ev->event ) {
            case SFE_MOUNT:
                status = uuid_submit( SFE_ARG0( ev ) );
                break;
            }

            return status;
        }

        // update event, eid and uuids for args
        ev->eid = OSAddAtomic64( 1, &gUniqueEvent );
        if( ( ev->event & SFE_ARG ) == SFE_ARG ) {
            uuid_fetch( SFE_ARG0( ev ) );
            if( ( ev->event & SFE_ARGS ) == SFE_ARGS ) {
                uuid_fetch( SFE_ARG1( ev ) );
            }
        }

        // grab next position, copy, free lock
        for( ;; ) {
            pos = OSIncrementAtomic( ( SInt32 * ) &gQueuePos ) & QUEUEMASK;
            if( FastSIntLockEx( &gQueue[pos].lck ) ) {
                break;
            }
            printf( "deveye: could not lock at %d to insert, trying next.\n", pos );
        }
        memcpy( ( void * ) &gQueue[pos].ev, ( void * ) ev, SFE_SIZE( ev ) );
        FastSIntUnlockEx( &gQueue[pos].lck );

        // wakeup listeners
        wakeup( ( caddr_t ) &gQueue );

        // success
        return 0;

}

/**
 * Inserts a single SFE into the kernel.
 */
int queue_push_user( user_addr_t ev, int length ) {

    uint8_t buffer[SFE_MAXLEN];

    // confirm we've been given enough data
    if( length < sizeof( struct sfe ) ) {
        printf( "deveye: copyin not valid - len=%d, len(sfe)=%d\n", length, sizeof( struct sfe ) );
        return EINVAL;
    }

    // copy in from user-space
    if( copyin( ev, ( void * ) buffer, length ) ) {
        printf( "deveye: copyin failed - len=%d\n", length );
        return EFAULT;
    }

    // push the event
    return queue_push( ( struct sfe * ) buffer );

}

/**
 * Does the internal work regarding the creation of a formal event from KAuth-vnode data.
```

```
 */
int event_fileop( kauth_cred_t cred, kauth_action_t action, vfs_context_t ctx, vnode_t vp ) {

    uint8_t buffer[SFE_MAXLEN];
    struct sfe *ev = ( struct sfe * ) buffer;
    struct argsfe *arg = ( struct argsfe * ) ( buffer + sizeof( struct sfe ) );
    struct vnode_attr va;

    // grab standard vars
    ev->uid = kauth_cred_getuid( cred );
    ev->gid = kauth_cred_getgid( cred );
    ev->pid = proc_selfpid();

    // set up vnode request
    VATTR_INIT( &va );
    VATTR_WANTED( &va, va_fsid );
    VATTR_WANTED( &va, va_fileid );
    VATTR_WANTED( &va, va_mode );
    VATTR_WANTED( &va, va_uid );
    VATTR_WANTED( &va, va_gid );

    // throw out if this vnode can't be found
    if( vnode_getattr( vp, &va, ctx ) ) {
        return -1;
    }

    // grab details from vnode
    memset( arg->uuid, 0, 16 );
    arg->dev = ( dev_t ) va.va_fsid;
    arg->ino = ( ino_t ) va.va_fileid;
    arg->mode = ( int32_t ) VTTOIF( vnode_vtype( vp ) ) | va.va_mode;
    arg->uid = ( uid_t ) va.va_uid;
    arg->gid = ( gid_t ) va.va_gid;

    // grab path from vnode, and store it aligned to a 8-byte word
    arg->length = MAXPATHLEN;
    if( vn_getpath( vp, arg->path, &( arg->length ) ) ) {
        arg->path[0] = '\0';
    }
    arg->length = ( ( strlen( arg->path ) >> 3 ) + 1 ) << 3;

    // submit fileop open
    if( action == KAUTH_FILEOP_OPEN ) {
        ev->event = SFE_OPEN;
        queue_push( ev );
    }
/*
    // submit vnode auth read
    if( action & KAUTH_VNODE_READ_DATA ) {
        ev->event = SFE_READ;
        queue_push( ev );
    }
*/

    // submit vnode auth execute
    if( action == KAUTH_FILEOP_EXEC ) {
        ev->event = SFE_EXEC;
        queue_push( ev );
    }

/*
    // submit vnode auth write
    if( action & KAUTH_VNODE_WRITE_DATA ) {
        ev->event = SFE_WRITE;
        queue_push( ev );
    }
*/
    // done and done
    return 0;

}

/**
 * Finds the event with id most closest (greater) to 'last' (or any, if zero).
 */
int event_obtain( uid_t uid, uint64_t last, uint8_t poll, struct sfe *ev ) {

    uint64_t event_start;
    int32_t pos, closest, count;

retry:
    count = 0;
    closest = -1;
    pos = OSAddAtomic( 0, &gQueuePos );
    event_start = OSAddAtomic64( 0, &gUniqueEvent );

    // check shutdown condition
    if( !gQueueActive ) {
        return EBUSY;
    }

    // work backwards
    for( ;; ) {
        pos = ( pos - 1 ) & QUEUEMASK;

        // check raw wrap
        if( ++count == QUEUELEN ) {
            printf( "deveye: had a simple wrap\n" );
            pos = -1;
            break;
        }

        // try to get a shared lock on this object - if we can't, we've wrapped (or, uh, just started)
        if( FastSIntLock( &gQueue[pos].lck ) < 0 ) {
            dprintf( "deveye: pos %d was being written to\n", pos );
            continue;
        }

        // check other wrap conditions (blank entry, id wrap)
        if( !gQueue[pos].ev.sfe.eid || gQueue[pos].ev.sfe.eid > event_start ) {
            FastSIntUnlock( &gQueue[pos].lck );
            printf( "deveye: pos %d lock ok, but was blank or a wrap\n", pos );
            pos = -1;
            break;
        }

        // check uid condition
        if( !uid || gQueue[pos].ev.sfe.uid == uid ) {

            // if we don't care what event we find
            if( !last ) {
                dprintf( "deveye: got pos %d because we didn't care\n", pos );
                break;
            }
```

```
            // if we're not quite there yet, store closest
            if( gQueue[pos].ev.sfe.eid > last ) {
                if( closest != -1 ) {
                    FastSIntUnlock( &gQueue[closest].lck );
                }
                closest = pos;
                continue;
            }

            // if we've hit our limit - note we don't actually want /this/ event
            if( gQueue[pos].ev.sfe.eid <= last ) {
                FastSIntUnlock( &gQueue[pos].lck );
                dprintf( "deveye: hit our eid limit on pos %d\n", pos );
                pos = -1;
                break;
            }

        }

        // unlock before loop
        FastSIntUnlock( &gQueue[pos].lck );

    }

    // if we had focus on a closest ok element, use that
    if( closest != -1 ) {
        pos = closest;
        dprintf( "deveye: got pos %d - was closest to eid limit\n", pos );
    }

    // check outcome of search
    if( pos == -1 ) {

        // ignore if we're polling
        if( poll ) {
            return EAGAIN;
        }

        // check if we should wrap without waiting for a signal
        if( event_start != OSAddAtomic64( 0, &gUniqueEvent ) ) {
            dprintf( "deveye: we have new events,"
                " we don't need to bother sleeping\n" );
            goto retry;
        }

        // sleep for a new event
        dprintf( "deveye: sleeping for relevant event\n" );
        if( msleep( ( caddr_t ) &gQueue, NULL, PCATCH, "event_obtain", NULL ) ) {
            return EINTR;
        }
        goto retry;

    }

    // return this event
    dprintf( "deveye: returning event at %d\n", pos );
    memcpy( ( void * ) ev,
        ( void * ) &( gQueue[pos].ev.sfe ),
            SFE_SIZE( &( gQueue[pos].ev.sfe ) ) );
    FastSIntUnlock( &gQueue[pos].lck );

    // success
    return 0;

}

/**
 * Quick hack to insert a single SFE into the kernel.
 */
int event_obtain_user(
    uid_t uid, uint64_t *last, uint8_t poll, user_addr_t ev, int *length ) {

    uint8_t buffer[SFE_MAXLEN];
    int status, size;

    // confirm we've been given enough space to work with
    if( *length < SFE_MAXLEN ) {
        printf( "deveye: copyout might fail - len=%d, len(sfe)=%d\n",
            *length, sizeof( struct sfe ) );
        return EINVAL;
    }

    // find a relevant event
    if( status = event_obtain( uid, *last, poll, ( struct sfe * ) buffer ) ) {
        return status;
    }

    // copyout a single event
    size = SFE_SIZE( ( struct sfe * ) buffer );
    if( copyout( ( void * ) buffer, ev, size ) ) {
        printf( "deveye: copyout failed - len=%d\n", size );
        return EFAULT;
    }

    // update last event pointer
    *last = ( ( struct sfe * ) buffer )->eid;
    return 0;

}
```

eyemgr.c

This tool is compiled and run as root, supporting the kernel extension directly with information sourced from the private FSEvents [2] API.

```
/**
 * eyemgr.c - handles module load, and fsevents hook
 * in the not-too distant future, it would be great to have this all inside the kernel, but for now - here it is
 */

#include <CoreServices/CoreServices.h>
#include <CoreFoundation/CoreFoundation.h>
#include <errno.h>
#include <fcntl.h>
#include <grp.h>
#include <libproc.h>
#include <pthread.h>
#include <pwd.h>
#include <stdint.h>
#include <stdio.h>
```

```
#include <stdlib.h>
#include <string.h>
#include <sys/ioccom.h>
#include <sys/ioctl.h>
#include <sys/param.h>
#include <sys/mount.h>
#include <sys/stat.h>
#include <sys/sysctl.h>
#include <sys/types.h>
#include <sys/ucred.h>
#include <sys/uio.h>
#include <sys/xattr.h>
#include <unistd.h>
#include <pthread.h>

#include "fsevents.h"
#include "event.h"
#include "deveye.h"

#define BUFLEN 1024*16

int get_fsevents();
int get_eyefs();
int parse_fsevent( void *, void * );
uint32_t sfe_type( int32_t );
void register_volume_path( const char * );
void fsevents_callback(
    ConstFSEventStreamRef,
    void *, size_t, void *,
    const FSEventStreamEventFlags eventFlags[],
    const FSEventStreamEventId eventIds[] );
void *mount_mgr( void * );

static int eyefs;

/**
 * Core eyemgr code.
 */
int main() {

    uint8_t buffer[BUFLEN], output[SFE_MAXLEN];
    int fsevents, length, parsed, status = EIO;
    struct eyefs request;
    struct sfe *outputp = ( struct sfe * ) output;
#ifdef DEBUG
    struct argsfe *temparg = ( struct argsfe * ) ( output + sizeof( struct sfe ) );
    struct argsfe *temparg2;
#endif
    pthread_t th;

    sranddev();

    // set up eyefs request struct (won't change)
    request.count = 1;
    request.version = EYEFS_VERSION;
    request.store.ptr = ( void * ) output;

    // check that we are root
    if( getuid() != 0 ) {
        fprintf( stderr, "eyemgr; should be invoked as root"
            " (quit fucking around :D)\n" );
    }

    // grab fsevents fd
    if( ( fsevents = get_fsevents() ) < 0 ) {
        fprintf( stderr, "eyemgr; couldn't open fsevents (%d)"
            " - is this tiger or higher?\n", fsevents );
    }

    // load the deveye kext
#ifdef DEBUG
    fprintf( stderr, "eyemgr; DEBUG - not attempting to load deveye kext\n" );
#else
    if( system( "/sbin/kextload /System/Library/Extensions/deveye.kext" ) ) {
        fprintf( stderr, "eyemgr; couldn't load/find deveye kext\n" );
    }
#endif

    // grab eyefs fd
    if( ( eyefs = get_eyefs() ) < 0 ) {
        fprintf( stderr, "eyemgr; couldn't open eyefs (%d)"
            " - was the kext loaded ok?\n", eyefs );
    }

    // create mount/umount thread
    pthread_create( &th, NULL, mount_mgr, NULL );

    // core loop, read from fsevents
    while( 0 < ( length = read( fsevents, buffer, sizeof( buffer ) ) ) ) {

        // we might be given multiple events; parse them all
        for( parsed = 0; parsed != length; ) {

            // clear output struct (probably not required)
            memset( ( void * ) output, 0, SFE_MAXLEN );

            // set up request
            parsed += parse_fsevent( buffer + parsed, output );
            request.size = SFE_SIZE( outputp );

            // FIXME: remove debug output
#ifdef DEBUG
            printf( "got event (size=%d): ev=%d, arg0=(%d) %s",
                request.size, outputp->event, temparg->length, temparg->path );
            if( ( outputp->event & SFE_ARGS ) == SFE_ARGS ) {
                temparg2 = ( struct argsfe * ) ( ( ( void * ) temparg )
                    + sizeof( struct argsfe ) + temparg->length );
                printf( ", arg1=(%d) %s", temparg2->length, temparg2->path );
            }
            printf( "\n" );
#endif

            // submit to eyefs
            if( ioctl( eyefs, EYEFS_PUSH, &request ) ) {
                status = errno;
                goto fallout;
            }

        }

    }

fallout:
    // fsevents is shutting down; machine is probably going off
```

```
    if( length == 0 ) {
        status = 0;
    }
    fprintf( stderr, "falling out: length=%d, status=%d\n", length, status );

    // fall out
    return status;

}

/**
 * Connect to the fsevents device.
 */
int get_fsevents() {

    int fd, fsfd;
    fsevent_clone_args retrieve_ioctl;
    signed char event_list[FSE_MAX_EVENTS];

    memset( event_list, 0, sizeof( event_list ) );

    // set up event list
    event_list[FSE_CREATE_FILE]         = FSE_REPORT;
    event_list[FSE_DELETE]              = FSE_REPORT;
    event_list[FSE_STAT_CHANGED]        = FSE_REPORT;
    event_list[FSE_RENAME]              = FSE_REPORT;
    event_list[FSE_CONTENT_MODIFIED]    = FSE_REPORT;
    event_list[FSE_EXCHANGE]            = FSE_REPORT;
    event_list[FSE_FINDER_INFO_CHANGED] = FSE_REPORT;
    event_list[FSE_CREATE_DIR]          = FSE_REPORT;
    event_list[FSE_CHOWN]               = FSE_REPORT;

    // set up ioctl request
    retrieve_ioctl.event_list = event_list;
    retrieve_ioctl.num_events = sizeof( event_list );
    retrieve_ioctl.event_queue_depth = -1;
    retrieve_ioctl.fd = &fd;

    // open fsevents device, get inner fd
    if( 0 > ( fsfd = open( "/dev/fsevents", 0 ) ) ) {
        return -2;
    }
    if( 0 > ioctl( fsfd, FSEVENTS_CLONE, &retrieve_ioctl ) ) {
        return -3;
    }
    close( fsfd );

    // success
    return fd;

}

/**
 * Connect to the eyemgr device node.
 */
int get_eyefs() {

    // is it really not more complicated than this?
    return open( "/dev/" EYEFS_DEV, 0 );

}
```

```
/**
 * Creates a single sfe (in stream format) from the given fsevents input buffer.
 */
int parse_fsevent( void *buffer, void *output ) {

    struct sfe *ev = ( struct sfe * ) output;
    struct argsfe *arg = NULL;
    struct {
        int32_t type;
        pid_t pid;
    } __attribute__((__packed__)) *header = ( void * ) buffer;
    struct {
        uint16_t type;
        uint16_t length;
        union {
            char str[0];
            dev_t dev;
            ino_t inode;
            int32_t mode;
            uid_t uid;
            gid_t gid;
            int32_t int32;
            int64_t int64;
        } v;
    } __attribute__((__packed__)) *fsarg;
    struct proc_bsdinfo pinfo;
    int file;

    ev->event = sfe_type( header->type ) | SFE_FSEVENTS;
        // identify this event as from fsevents
    ev->pid = header->pid;
    ev->uid = -1;
    ev->gid = -1;
    file = -1;

    // attempt to grab the uid/gid of this event via system call
    if( 0 < proc_pidinfo( header->pid, PROC_PIDTBSDINFO, 0, ( void * ) &pinfo, sizeof( struct proc_bsdinfo ) ) ) {
        ev->uid = pinfo.pbi_uid;
        ev->gid = pinfo.pbi_gid;
    }

    // parse every argument to this fsevent
    for( fsarg = ( void * ) header + sizeof( *header );
        fsarg->type != FSE_ARG_DONE;
        fsarg = ( void * ) fsarg + ( sizeof( uint16_t ) * 2 + fsarg->length ) ) {

        // handle header & non-file arguments
        switch( fsarg->type ) {
        case FSE_ARG_INT32:
            ev->eid = ( int64_t ) fsarg->v.int32;
            break;
        case FSE_ARG_INT64:
            ev->eid = fsarg->v.int64;
            break;
        case FSE_ARG_STRING:
            // move up to the next file arg
            if( file == -1 && ( ev->event & SFE_ARG ) ) {
                file = 0;
                arg = ( struct argsfe * ) ( output + sizeof( struct sfe ) );
            }
            else if( file == 0 && ( ev->event & SFE_ARGS ) == SFE_ARGS ) {
                file = 1;
```

```
                arg = ( struct argsfe * )
                    ( output + sizeof( struct sfe ) +
                        sizeof( struct argsfe ) + arg->length );
            }
            else {
                file = -2;
                arg = NULL;
                break;
            }

            // copy string, pad to 8-byte boundary
            arg->length = stpcpy( arg->path, fsarg->v.str ) - arg->path;
            arg->length = ( ( arg->length >> 3 ) + 1 ) << 3;
            break;
        default:
            // if we don't have a file, ignore specific file args
            if( file < 0 ) {
                continue;
            }
        }

        // handle specific file args
        switch( fsarg->type ) {
        case FSE_ARG_DEV:
            arg->dev = fsarg->v.dev;
            break;
        case FSE_ARG_INO:
            arg->ino = fsarg->v.inode;
            break;
        case FSE_ARG_MODE:
            arg->mode = fsarg->v.mode;
            break;
        case FSE_ARG_UID:
            arg->uid = fsarg->v.uid;
            if( ev->uid == -1 ) {
                ev->uid = arg->uid;
            }
            break;
        case FSE_ARG_GID:
            arg->gid = fsarg->v.gid;
            if( ev->gid == -1 ) {
                ev->gid = arg->gid;
            }
            break;
        }

    }

    // fix uid/gid if we couldn't use procinfo,
    // or didn't use one from an argument (rare)
    if( ev->uid == -1 ) {
        ev->uid = 0;
    }
    if( ev->gid == -1 ) {
        ev->gid = 0;
    }

    // return total consumption
    return ( void * ) fsarg - ( void * ) header + sizeof( uint16_t );

}


/**
 * Translates a given fsevent type to the corresponding sfe type.
 */
uint32_t sfe_type( int32_t fsevent ) {
    switch( fsevent ) {
    case FSE_CREATE_FILE:           return SFE_CREATE;
    case FSE_DELETE:                return SFE_DELETE;
    case FSE_STAT_CHANGED:          return SFE_STAT;
    case FSE_RENAME:                return SFE_RENAME;
    case FSE_CONTENT_MODIFIED:      return SFE_MODIFIED;
    case FSE_EXCHANGE:              return SFE_EXCHANGE;
    case FSE_FINDER_INFO_CHANGED:   return SFE_FINDER;
    case FSE_CREATE_DIR:            return SFE_MKDIR;
    case FSE_CHOWN:                 return SFE_CHOWN;
    }
    return -1;
}

/**
 * Register volume information for the volume mounted at the given path.
 */
void register_volume_path( const char *path ) {

    int fd = -1;
    struct stat info;
    CFUUIDRef uuid;
    CFUUIDBytes bytes;
    uint8_t output[SFE_MAXLEN];
    struct eyefs request;
    struct sfe *outputp = ( struct sfe * ) output;
    struct argsfe *arg = ( struct argsfe * ) ( output + sizeof( struct sfe ) );

    // try to open the device as a fd (lock it from being unmounted)
    if( -1 == ( fd = open( path, O_RDONLY ) ) ) {
        fprintf( stderr, "eyemgr: couldn't open device at path `%s`\n", path );
        return;
    }

    // try to grab the device node
    if( fstat( fd, &info ) ) {
        fprintf( stderr, "eyemgr: couldn't stat %s\n", path );
        goto fallout;
    }
    fprintf( stderr, "eyemgr: register_volume_path got `%s` (dev %d)\n", path, info.st_dev );

    // set up request
    request.count = 1;
    request.version = EYEFS_VERSION;
    request.store.ptr = ( void * ) output;

    // clear output struct (probably not required)
    memset( ( void * ) output, 0, SFE_MAXLEN );
    outputp->event = SFE_MOUNT;
    arg->dev = info.st_dev;
    request.size = SFE_SIZE( outputp );

    // try to grab the xattr
    if( -1 == fgetxattr( fd, EYEFS_TLD "." EYEFS_UUID, ( void * ) &arg->uuid, 16, 0, 0 ) ) {
        // assume ENOATTR

        if( errno != ENOATTR ) {
            fprintf( stderr, "eyemgr: non-ENOATTR errno (%d) on getting uuid, path = %s\n",
```

```
                errno, EYEFS_TLD "." EYEFS_UUID );
        }

        // copy a ref to the device's FSEvents uuid
        if( NULL == ( uuid = FSEventsCopyUUIDForDevice( info.st_dev ) ) ) {
            uuid = CFUUIDCreate( NULL );
        }

        bytes = CFUUIDGetUUIDBytes( uuid );
        memcpy( arg->uuid, ( char * ) &bytes, 16 );
        printf( "eyemgr: got id, %llu.%llu\n",
            *( ( ( uint64_t * ) &arg->uuid ) + 1 ), *( ( uint64_t * ) &arg->uuid ) );
        CFRelease( uuid );

        // alternatively, generate a random uuid
/*
        else {
            int i;
            for( i = 0; i < sizeof( arg->uuid ); i += sizeof( int ) ) {
                *( ( int * ) ( ( ( char * ) arg->uuid ) + i ) ) = rand();
            }
            printf( "eyemgr: generated random id, %llu.%llu\n",
                *( ( ( uint64_t * ) &arg->uuid ) + 1 ), *( ( uint64_t * ) &arg->uuid ) );
        }
*/

        // set xattr
        if( -1 == fsetxattr( fd,
            EYEFS_TLD "." EYEFS_UUID,
                ( void * ) &arg->uuid, 16, 0, XATTR_CREATE ) ) {
            fprintf( stderr, "eyemgr: couldn't set uuid on new volume,"
                " errno=%d\n", errno );
            goto fallout;
        }

    }
    else {
        printf( "eyemgr: found id on disk, %llu.%llu\n",
            *( ( ( uint64_t * ) &arg->uuid ) + 1 ), *( ( uint64_t * ) &arg->uuid ) );
    }

    // try to push mount event
    if( ioctl( eyefs, EYEFS_PUSH, &request ) ) {
        fprintf( stderr, "eyemgr: register_volume_path could not ioctl() with mount info, eyefs=%d, errno=%d\n", eyefs, errno );
    }

fallout:
    if( fd > -1 ) {
        close( fd );
    }

}

/**
 * Handle FSEvents callbacks, and filter out just the mounts and unmounts.
 */
void fsevents_callback(
    ConstFSEventStreamRef streamRef,
    void *clientCallBackInfo,
    size_t numEvents,
    void *eventPaths,
    const FSEventStreamEventFlags eventFlags[],
    const FSEventStreamEventId eventIds[] ) {

    int i;
    char **paths = eventPaths;

    for( i = 0; i < numEvents; ++i ) {

        // handle a mount event
        if( eventFlags[i] & kFSEventStreamEventFlagMount ) {
            register_volume_path( paths[i] );
        }

        // we don't do anything with this at the moment/
        //  - can we, since we can't get the device node now?
        else if( eventFlags[i] & kFSEventStreamEventFlagUnmount ) {
//            printf( "UNMOUNT: %s\n", paths[i] );
        }

    }

}

/**
 * Handles all mounts/unmounts.
 */
void *mount_mgr( __unused void *x ) {

    CFStringRef root = CFSTR( "/" );
    CFArrayRef pathsToWatch = CFArrayCreate( NULL, ( const void ** ) &root, 1, NULL );
    FSEventStreamRef stream;
    CFAbsoluteTime latency = 0.2;
    Boolean ret;
    int nDevices, index;
    struct statfs *fsList;

    // set up FSEvents stream
    stream = FSEventStreamCreate( NULL,
        &fsevents_callback, NULL, pathsToWatch,
        kFSEventStreamEventIdSinceNow, latency,
        kFSEventStreamCreateFlagNoDefer );
    FSEventStreamScheduleWithRunLoop(
        stream, CFRunLoopGetCurrent(), kCFRunLoopDefaultMode );
    ret = FSEventStreamStart( stream );
    // handle file systems that already existed
    if( ( nDevices = getfsstat( NULL, 0, MNT_NOWAIT ) ) < 0 ) {
         fprintf( stderr, "fatal; getfsstat failed\n" );
         exit( 1 );
    }

    // allocate space for the fslist
    if( NULL == ( fsList = ( struct statfs * )
            malloc( nDevices * sizeof( struct statfs ) ) ) ) {
        perror( "malloc" );
        exit( 2 );
    }

    // obtain information about existing file systems
    if( ( nDevices = getfsstat( fsList, nDevices * sizeof( struct statfs ), MNT_NOWAIT ) ) < 0 ) {
        fprintf( stderr, "fatal; getfsstat failed with full request\n" );
        exit( 3 );
    }
```

```
    for( index = 0; index < nDevices; ++index ) {
        register_volume_path( fsList[index].f_mntonname );
    }
    free( fsList );

    // run loop
    printf( "eyemgr: mount_mgr setup ok (%d)\n", ret );
    CFRunLoopRun();

    // shutdown!
    CFRelease( pathsToWatch );
    CFRelease( stream );

    return 0;

}
```

fastlock.c

```

#include "fastlock.h"

/**
 * Obtain a shared lock on the integer at the given location. Returns the
 * positive lock count, or -1 if locked exclusively.
 */
int FastSIntLock( fastlock_t *v ) {
    int32_t value;
    while( ( value = *v ) >= 0 ) {
        if( OSCompareAndSwap( value, value + 1, ( UInt32 * ) v ) ) {
            return value + 1;
        }
    }
    return -1;
}

/**
 * Releases a shared lock on the integer at the given location. Returns the
 * positive (or zero) lock count, or -1 if locked exclusively (or unlocked).
 */
int FastSIntUnlock( fastlock_t *v ) {
    int32_t value;
    while( ( value = *v ) > 0 ) {
        if( OSCompareAndSwap( value, value - 1, ( UInt32 * ) v ) ) {
            return value - 1;
        }
    }
    return -1;
}

/**
 * Obtains an exclusive lock with the given integer location. Returns true if
 * successful, false otherwise.
 */
Boolean FastSIntLockEx( fastlock_t *v ) {
    return OSCompareAndSwap( 0, -1, ( UInt32 * ) v );
}

/**
 * Releases an exclusive lock with the given integer location. returns true if
 * successful, false otherwise.
 */
Boolean FastSIntUnlockEx( fastlock_t *v ) {
    return OSCompareAndSwap( -1, 0, ( UInt32 * ) v );
}
```

fastlock.h

```

#include "fastlock.h"

/**
 * Obtain a shared lock on the integer at the given location. Returns the
 * positive lock count, or -1 if locked exclusively.
 */
int FastSIntLock( fastlock_t *v ) {
    int32_t value;
    while( ( value = *v ) >= 0 ) {
        if( OSCompareAndSwap( value, value + 1, ( UInt32 * ) v ) ) {
            return value + 1;
        }
    }
    return -1;
}

/**
 * Releases a shared lock on the integer at the given location. Returns the
 * positive (or zero) lock count, or -1 if locked exclusively (or unlocked).
 */
int FastSIntUnlock( fastlock_t *v ) {
    int32_t value;
    while( ( value = *v ) > 0 ) {
        if( OSCompareAndSwap( value, value - 1, ( UInt32 * ) v ) ) {
            return value - 1;
        }
    }
    return -1;
}

/**
 * Obtains an exclusive lock with the given integer location. Returns true if
 * successful, false otherwise.
 */
Boolean FastSIntLockEx( fastlock_t *v ) {
    return OSCompareAndSwap( 0, -1, ( UInt32 * ) v );
}

/**
 * Releases an exclusive lock with the given integer location. returns true if
 * successful, false otherwise.
 */
Boolean FastSIntUnlockEx( fastlock_t *v ) {
    return OSCompareAndSwap( -1, 0, ( UInt32 * ) v );
}
```

Appendix B

Lighthouse

The code below is the complete implementation of Lighthouse in Python.

core.py

This describes the core of Lighthouse, and mirrors the defined algorithm most explicitly. It does not provide interactive functionality, however.

```
#!/usr/bin/env python

from __future__ import with_statement
import eyedropper, schema, re, os, time, uuid, stat, threading, sys, signal
import logging
from util import *

logging.basicConfig()

class Lighthouse( object ):
    """ Lighthouse watches your files. To a big extent, it is the middle-man
    between direct disk access and behaviour (lighthouse/inodes), and the
    node-based database structure which conceptualises nodes and their
    relationships. """

    def __init__( self ):
        """ Configures a new Lighthouse object. """
        self.devices = DeviceSet()

    def watch( self ):
        """ A helper method to watch for events until a KeyboardInterrupt.
        This method doesn't maintain a state over the last seen event, so
        calling it multiple times will mean events are dropped. """
        try:
            for event in eyedropper.listen():
                self.act( event )
        except KeyboardInterrupt:
            pass

    def threaded( self ):
        """ Waits for events to be pushed into our queue, then acts on them. """

        # set up shared objects
        condition = threading.Condition()

        # threaded function
        def func():
            while func.active:
                with condition:
                    while func.active and not len( func.queue ):
                        condition.wait()
                    mine = func.queue
                    func.queue = []
                for i in mine:
                    self.act( i )
        func.queue = []
        func.active = True

        # run function, pass events through
        th = threading.Thread( target = func )
        th.start()
        try:
            for event in eyedropper.listen():
                assert event is not None
                with condition:
                    func.queue.append( event )
                    condition.notify()

        # fallout conditions
        except KeyboardInterrupt:
            print "## Ctrl-C."
        except AssertionError:
            pass

        # shutdown
        func.active = False
        with condition:
            condition.notify()
        th.join()
        print "## Shutdown OK."

    def act( self, event ):
        """ Act on a single event. """

        # ignore our own pid.
        if event.pid == os.getpid():
            pass

        # standard unix rename - if target has an inode, it 'exists'
        elif event.type == eyedropper.RENAME:

            # this file exists, and will be baleeted!
            if event.target.inode:
                self.release( event.target )

            # seen the original, but with a new name
            node = self.seen( event.focus, path = event.target.path )

        # content flip
        elif event.type == eyedropper.EXCHANGE:
            # TODO: remember how this works
            pass

        # otherwise, just observe (or release) the file
        else:
            if event.type == eyedropper.DELETE:
                self.release( event.focus )
            else:
                node = self.seen( event.focus )

    def release( self, file ):
        """ Handles notifications that files have been released. Does not
        attempt to interact with the file on-disk, as it has been released,
        and we might interact with a new file in the same place. """

        if file.uid != os.getuid():
            return

        # grab device structure
        device = self.devices.get( file.uuid, file.device, file.path )
        if not device:
            return None
        assert device.uuid is not None

        # try to find a matching node

        nodes = schema.Node.selectBy( inode = file.inode, device = device.uuid )
        try:
            node = max( nodes, key = lambda f: f.seen )
```

```
        except ValueError:
            return None

        node.apply( schema.Destroyed() )
        print "-- Releasing node (last seen at) `%s` (%s)"
            % ( os.path.join( node.dirname, node.basename ), node.uuid )

        # return node itself
        return node

    def props( self, path ):
        """ Returns a small tuple of properties of the file at the given
        path. Format is ( created, modified, size ). """

        # standard args
        result = os.stat( path )
        modified = result.st_mtime
        size = result.st_size
        created = Created( path )

        # modify directory results /slightly/ so we don't have so many changes
        if stat.S_ISDIR( result.st_mode ):
            modified = created
            size = -1

        # return tuple
        return( created, modified, size )

    def brand( self, node, force = False ):
        """ Brand a given LocalDevice node with an ID, or grab the current
        branding. """

        try:
            if force:
                raise AttributeError

            id = node.id

            # prevent sqlobject from breaking
            if len( filter( lambda f: ord( f ) > 127, id ) ):
                id = None
                raise AttributeError

            # other weird stuff
            if id is None:
                raise AttributeError

        except AttributeError:
            try:
#                id = uuid.uuid1().bytes
                id = uuid.uuid1().hex
                node.id = id
            except AttributeError:
                raise OSError

        return id

    def seen( self, file, path = None ):
        """ Handles notifications that files have been observed. From a
        high-level point of view, this method doesn't care about how this
        has occurred. Returns the found (current) node. """

        stime = time.time()

        if file.uid != os.getuid():
            return

        # fix presented path, grab relevant device struct
        if path is None:
            path = file.path
        device = self.devices.get( file.uuid, file.device, path )
        if not device:
            return None
        assert device.uuid is not None

        # how can we access this file?
        if device.volfs:
            apath = "/.vol/%d/%d" % ( file.device, file.inode )
        else:
            apath = path

        # grab parent data
        try:
            parent = os.stat( os.path.dirname( path ) )
            if parent.st_dev == file.device and parent.st_ino != file.inode:
                parent = parent.st_ino
            else:
                parent = 0 # we are the top level node
        except( OSError, IOError ):
            parent = None # parent is undefined

        # get information about the file in question (if we OSError, the file has gone)
        try:
            created, modified, size = self.props( apath )
            dirname = unicode( os.path.dirname( path ), 'utf-8' )
            basename = unicode( os.path.basename( path ), 'utf-8' )
            id = self.brand( device[file.inode] )
        except( OSError, IOError ):
            print "-- Node disappeared `%s` (?)" % ( path, )
            return

        # create and compare all immutable evidence on this node
        try:
            node = schema.Node.find( id )
            assert node.inode is not None
            assert node.inode == file.inode
            assert node.pinode == parent
            assert node.device == device.uuid
            assert node.size == size
            assert node.modified == modified
            assert node.created == created
#            assert node.dirname == dirname
            assert node.basename == basename
            print "-- Touched node `%s` (%s)" % ( path, id )

        # update our node structure
        except AssertionError:

            logging.exception( "Node Changed (%r, %r)", node.basename, basename )

            # something changed - create a new derived node
            dbtime = time.time()
            if node.inode:
                id = self.brand( device[file.inode], force = True )
```

```
                replacement = schema.Node( uuid = id )

                # apply relevant evidence
                if node.inode != file.inode or node.device != device.uuid:
                    replacement.apply(
                        schema.CopyCreation( parent = node )
                    )
                    print "-- Creating copied node `%s` (%s)"
                        % ( path, id )
                else:
                    replacement.apply(
                        schema.DerivedFrom( parent = node )
                    )
                    node.apply( schema.Destroyed() )
                    print "-- Creating derived/updated node `%s` (%s)"
                        % ( path, id )

                # focus on our new node
                node = replacement

            else:
                print "-- Creating completely new node `%s` (%s)"
                    % ( path, id )

            # add same location hints
            nodes = schema.Node.selectBy(
                pinode = parent, device = device.uuid, basename = basename
            )
            try:
                target = max( nodes, key = lambda f: f.seen )
                node.apply( schema.SameLocation( previous = target ) )
                print "-- Added SameLocation evidence directed at previous "
                    + "node (%s)" % ( target.uuid, )
            except ValueError:
                pass

            # configure the node
            node.inode = file.inode
            node.pinode = parent
            node.device = device.uuid
            node.size = size
            node.modified = int( modified )
            node.created = int( created )
            node.dirname = dirname
            node.basename = basename
            print "dbtime took %f" % ( time.time() - dbtime )

        print "took %f" % ( time.time() - stime )

        # return the node
        return node
```

schema.py

The SQLObject ?? database schema, providing *nodes* and *evidence*.

```
#!/usr/bin/env python

""" This file defines the basic set of evidence available to nodes within
the Lighthouse/Family system (aka Sam's thesis). Evidence may be attached
to any particular Node instance within the system, and is based around the
core idea that it defines why this node exists. """

from sqlobject import *
from sqlobject.inheritance import *
import uuid, os

class Node( SQLObject ):
    """ Each node represents an immutable file within our graph. A single
    node may be dead or alive; its reasoning for being created or destroyed
    is however attached to its micro-reasons. Since they are immutable, a
    node may exist for current and prior device/inode combinations. """

    # hex uuid, timestamp, evidence link
    uuid = StringCol( length = 32, alternateID = True )
    seen = DateTimeCol( default = DateTimeCol.now, notNone = True )
    evidence = MultipleJoin( 'BaseEvidence' )

    # device/fs attributes and timestamps
    inode = IntCol( default = None )
    device = StringCol( length = 32, default = None )
    pinode = IntCol( default = None )
    size = IntCol( default = None )
    modified = IntCol( default = None )
    created = IntCol( default = None )

    # path attributes
    dirname = UnicodeCol( default = None )
    basename = UnicodeCol( default = None )

    # indexes
    inodeIndex = DatabaseIndex( 'inode' )
    inodeDeviceIndex = DatabaseIndex( 'inode', 'device' )
    parentDeviceName = DatabaseIndex( 'pinode', 'device', 'basename' )

    # get a node for this uuid
    @staticmethod
    def find( id ):
        """ Get a node for the passed UUID, no matter what the cost. """
        try:
            node = Node.byUuid( id )
        except SQLObjectNotFound:
            node = Node( uuid = id )
        return node

    # apply given evidence to this object
    def apply( self, evidence, *args ):
        for v in [ evidence ] + list( args ):
            assert isinstance( v, BaseEvidence )
            v.node = self

    def __str__( self ):
        return self.uuid

    def __repr__( self ):
        return( "<Node uuid=\"%s\" seen=%s destroyed=%s device=\"%s\" inode=%s pinode=%s "
            + "size=%s modified=%s created=%s path=\"%s\">" ) % (
            self, self.seen, self.destroyed, self.device, self.inode, self.pinode,
            self.size, self.modified, self.created, os.path.join( self.dirname, self.basename )
        )

    @property
```

```
    def ancestors( self ):
        """ Return all direct ancestors of this node. """

        output = set()

        temp = list( DerivedFrom.select( BaseEvidence.q.node == self ) )
        temp = map( lambda f: f.parent, temp )
        output.update( temp )

        temp = list( SameLocation.select( BaseEvidence.q.node == self ) )
        temp = map( lambda f: f.previous, temp )
        output.update( temp )

        # success!
        return output

    @property
    def descendants( self ):
        """ Return all direct descendants of this node. """
        # generate list of direct children (evidence)
        output = set()
        output.update(
            list( DerivedFrom.select( DerivedFrom.q.parent == self ) )
        )
        output.update(
            list( SameLocation.select( SameLocation.q.previous == self ) )
        )

        # map this to actual nodes
        output = set( map( lambda f: f.node, output ) )

        # success!
        return output

    @property
    def destroyed( self ):
        """ Return the time at which this node was destroyed (or False). """
        evidence = list( Destroyed.select( BaseEvidence.q.node == self ) )
        if len( evidence ):
            return evidence[0].timestamp
        return False

class BaseEvidence( InheritableSQLObject ):
    """ The root evidence object. Can be instantiated on its own, but would
    mean extremely little. """

    # node target, timestamp
    node = ForeignKey( 'Node', default = None )
    timestamp = DateTimeCol( default = DateTimeCol.now, notNone = True )

    # indexes
    nodeIndex = DatabaseIndex( 'node' )

class SameLocation( BaseEvidence ):
    """ This file is at the same location as a previous file. """
    previous = ForeignKey( 'Node', notNone = True  )

class Replaced( SameLocation ):
    """ This file replaced a previous node, causing it to be destroyed. """
    pass

class Destroyed( BaseEvidence ):
    """ This node has been observed as destroyed. """
    pass

class DerivedFrom( BaseEvidence ):
    """ This node was directly derived (modified) from a previous node. """
    parent = ForeignKey( 'Node', notNone = True )

class CopyCreation( DerivedFrom ):
    """ This node is a direct copy of a previous node. """
    pass
```

interactive.py

Sets up the local Lighthouse database using SQLObject. Included by other, interactive tools.

```
#!/usr/bin/env python

import sqlobject, core, schema, os

# define connection
path = os.path.expanduser( "~/.lighthouse.db" )
sqlobject.sqlhub.processConnection = sqlobject.connectionForURI( "sqlite://" + path )
#sqlobject.sqlhub.processConnection.debug = True

# try to create tables
tables = [ 'Node', 'BaseEvidence', 'SameLocation', 'Replaced', 'Destroyed', 'DerivedFrom', 'CopyCreation' ]
for i in tables:
    try:
        schema.__dict__[i].createTable()
        print "Created table `%s`: ok" % i
    except:
        if __name__ == "__main__":
            print "Created table `%s`: already exists" % i

```

test.py

This sets up and then runs the core of Lighthouse; invoke this interactive tool to begin monitoring events.

```
#!/usr/bin/env python

import sqlobject, core, schema, os
import interactive

if __name__ == "__main__":

    # watch for events
    house = core.Lighthouse()
    try:
        house.threaded()
    except KeyboardInterrupt:
        pass

```

related.py

Provides the main testing framework as used within our evaluation, and described in our implementation. An interactive tool.

```
#!/usr/bin/env python

import sys, os, interactive, schema, util

# setups
assert __name__ == "__main__"
assert len( sys.argv ) == 2
target = sys.argv[1]

# grab inode/device
s = os.stat( target )
inode, device = s.st_ino, s.st_dev

# grab database node
node = schema.Node.byUuid( util.Node( device, inode ).id )

# grab all 'family' nodes
nodes = [ node ]
i = 0
while i < len( nodes ):
    node = nodes[i]

    for e in node.ancestors:
        if e not in nodes:
            nodes.append( e )

    for e in node.descendants:
        if e not in nodes:
            nodes.append( e )

    i += 1

# display all
print "got %d results" % len( nodes )
print
for i in nodes:
    print repr( i )
    print "ancestors=%d descendants=%d" % ( len( i.ancestors ), len( i.descendants ) )
    print
```

util.py

Mostly provides code used to help identify files over time, as discussed in our implementation. Used internally by other parts of the code.

```
#!/usr/bin/env python

import warnings
warnings.simplefilter( "ignore", DeprecationWarning )

import os, pickle, re
import uuid
from uuid import UUID
from xattr import xattr

import MacOS, macfs

class DeviceSet( object ):
    """ A wrapper object to help present all devices on a system. """
    # FIXME: obviously this doesn't handle inserts/removal of devices at the moment.

    def __init__( self ):
        self.devices = {}

    def get( self, uuid, device, path ):
        """ Get a device, directly based on uuid, device no, and reference path. """

        # hackery laem
        real = long()
        for i in uuid:
            real <<= 8
            real += ord( i )
        uuid = hex( long( real ) )[2:-1]
        del real

        if not uuid:
            return None

        if device not in self.devices or self.devices[device].uuid != uuid:
            try:
                self.devices[device] = LocalDevice( uuid, device, path )
                print "new device created: dev %d, uuid %s, ref path = %s" % ( device, uuid, path )
            except OSError:
                return None

        return self.devices[device]

    def __contains__( self, device ):
        """ Does this set contain the given device node? """
        return device in self.devices

class LocalDevice( object ):
    """ A device object which represents a local, accessible device. Allows
    users to interact with paths or inodes. """

    def __init__( self, uuid, device, path ):
        """ Create a new LocalDevice with reference arguments """

        if not uuid:
            raise OSError( "Blank UUID passed." )

        self.uuid = uuid
        self.device = os.stat( path ).st_dev

        if self.device != device:
            raise OSError( "Path/device mismatch." )

        try:
            self.prop = self.props( "/.vol/%d/@" % ( self.device, ) )
            self.volfs = True
        except OSError:
            self.volfs = False
            if not path:
                raise Exception( "This device does not exist or support volfs,"
                                 + " so a path must be specified." )
            self.prop = self.props( path )
        self.mount = self.prop['Mount Point']
```

```
72
73      def __contains__( self, inode ):
74          """ Does this device currently contain the passed inode? """
75
76          # throw out if volfs is not supported
77          if not self.volfs:
78              raise Exception( "volfs is not supported on this device!" )
79
80          # throw out on badly typed submissions
81          if not isinstance( inode, ( int, long ) ) or inode < 0:
82              raise TypeError( "A LocalDevice only supports positive "
83                               + "integer inodes." )
84
85          # fix inode request
86          if not inode:
87              inode = "@"
88
89          # return the status of this request
90          return os.path.exists( "/.vol/%d/%s" % ( self.device, inode ) )
91
92      def __getitem__( self, inode ):
93          """ Returns an inode structure from the LocalDevice. """
94
95          # inode does not exist
96          if inode not in self:
97              return None
98
99          # fix inode request
100         if not inode:
101             inode = "@"
102
103         # return a relevant node structure
104         return Node( self.device, inode )
105
106     @staticmethod
107     def props( path ):
108         """ Returns properties of the device mounted at the given path. """
109         output = {}
110         source = os.popen( "diskutil info %s" % path ).read().split( "\n" )
111         for i in source:
112             try:
113                 k, v = i.split( ":", 1 )
114                 output[k.strip()] = v.strip()
115             except:
116                 pass
117         if "Usage" in output:
118             raise OSError( "No such file or directory: '%s'" % ( path, ) )
119         return output
120
121 class Node( object ):
122     """ A simple xattr wrapper that should only be used by LocalDevice. """
123     _FORMAT = "com.moofco.lighthouse.%s"
124     _xattr = None
125
126     def __init__( self, device, inode ):
127         self._xattr = xattr( "/.vol/%d/%s" % ( device, inode ) )
128
129     def __getattr__( self, k ):
130         if k.startswith( "_" ):
131             return super( Node, self ).__getattr__( k )
132         path = Node._FORMAT % k
133         if path not in self._xattr:
134             raise AttributeError, k
135         try:
136             return pickle.loads( self._xattr[path] )
137         except:
138             raise AttributeError, k
139
140     def __setattr__( self, k, v ):
141         if k.startswith( "_" ):
142             super( Node, self ).__setattr__( k, v )
143         else:
144             path = Node._FORMAT % k
145             if v is None:
146                 del self._xattr[path]
147             else:
148                 self._xattr[path] = pickle.dumps( v )
149
150     def __delattr__( self, k ):
151         if k.startswith( "_" ):
152             super( Node, self ).__delattr__( k, v )
153         else:
154             self.__setattr__( k, None )
155
156 def Created( path ):
157     """ Return the creation time of a specified path. Mac specific. """
158     # weird base case that tends to fail out
159     if re.search( "^/\.vol/(\d+)/@$", path ):
160         cmd = "diskutil info %s | grep '^   Mount Point'" % path
161         output = os.popen( cmd ).read()
162         try:
163             path = re.search( "^   Mount Point:\s+(\S.*)$", output ).group( 1 )
164         except:
165             raise Exception( "Can't find mount point of root inode: '%s'" % path )
166
167     # normal case (easy)
168     try:
169         return float( macfs.FSSpec( path ).GetDates()[0] )
170     except MacOS.Error:
171         raise OSError( "No such file or directory: '%s'" % path )
172
173 # export list
174 __all__ = [ "DeviceSet", "LocalDevice", "Created" ]
```