AlamoFS: A versioning file system with support for non-linear history

by

Eric Stratmann

A Thesis submitted to the Faculty
in partial fulfillment
of the requirements for the
BACHELOR OF ARTS

Accepted

_____
Thesis and Major Advisor, Paul Shields

_____
Second Reader, Allen Altman

_____
Third Reader, Robert Snyder

_____
Mary B. Marcy, Provost and Vice President

Simon's Rock College of Bard
Great Barrington, Massachusetts
2008

Abstract

AlamoFS: A versioning file system with support for non-linear history

by

Eric Stratmann

This thesis presents the design and implementation of a versioning file system with support for non-linear history. AlamoFS is a userland file system that uses the FUSE kernel module and runs on the Linux operating system. As versioning features become more common in modern operating systems, AlamoFS explores several concepts that have not been included in any other file systems. In particular, the file system allows users to create their own branches.

Contents

Chapter 1

Introduction

AlamoFS is a versioning file system with support for non-linear history developed for the Linux operating system released under the BSD license. The aim of AlamoFS is to explore the concept of a branching file system and to make such a file system intuitive and easy to use.

Although various versioning file systems have been designed in the past[7, 10, 4, 3], none have gained much popularity. The majority of file systems in use today have minimum versioning features at most, usually limited to the ability to take snapshots of the file system[1, 8, 6]. Nonetheless, versioning is a feature that can be of great help to users, and AlamoFS attempts to bring closer the day in which all file systems have extensive versioning capabilities.

1.1 Background

Use of AlamoFS does not require much expertise on the part of the user. A basic understanding of the Linux command line should be sufficient, as usability was one of the goals of AlamoFS. In future versions of AlamoFS, even the requirement of having to use the command line might be removed.

For a reader of the thesis, a bit more background knowledge would be helpful, and so a brief introduction to several fundamental concepts relating to AlamoFS is presented.

1.1.1 Role of the file system

Any modern operating system (OS) uses some sort of non-volatile storage device to maintain user data when the computer is turned off. The most common devices used are hard drives and solid state drives. To avoid the clunky terminology of "non-volatile storage

device", the expression "hard drive" will be used throughout the text. Unless mentioned otherwise, the expression will refer to all such devices.

It is not an easy task for an OS to store files on the hard drive. An OS typically deals with information on the file level, while a hard drive works at the block level. Because of how slow it is to read from a hard drive compared to memory or cache, hard drives divide themselves up into blocks. When the OS reads from the hard drive, the OS must read the entire block (or blocks), even if the data required is smaller. A typical block size is 4KB. Blocks themselves store data as a sequence of binary numbers.

A hard drive is often divided into different partitions. Unlike blocks, which are a hardware distinction, partitions are implemented in software. Partitions are used, for example, to keep multiple operating systems on a hard drive or to use different file systems for different directories. When we say that a file system interacts in some way with the hard drive, we mean it interacts with the partition the file system is on. Of course, the size of a partition must be an integer multiple of the hard drive's block size.

The role of a file system is to create the abstraction of files. Since the OS deals with files and the hard drive deals with blocks, the file system acts as an intermediary between the two. Because of this abstraction, the OS and any program which uses kernel system calls to work with files does not need to know about the layout of the disk. As far as a program is concerned, the OS provides a namespace of files, usually organized as a hierarchy of directories and there are no such things as blocks. The partition that it works on only consists of directories and files inside these directories.

The kernel provides another level of abstraction known as the virtual file system (VFS) layer. Not only should a program not care about how files are stored on disk, it should also not matter which file system it is using. Instead of making file system specific function calls, such as *alamofs_open(args)*, it just calls the kernel's *open* system call. Linux's VFS provides a set of standards for functions and data structures that every file system must implement. These functions are the normal functions that a user expects a file system to provide, such as opening a file, writing to a file, reading from a file, moving a file, and finding all the files in a directory. The file system must also make available different statistics, such as the amount of free space.

Provided that it meets the requirements of the VFS, a file system is free to do whatever it wants. The VFS requires a file system to meet these specifications, but it does not enforce any sort of semantic meaning. For instance, a file system could wipe the entire partition whenever the *open* system call is called from a program, although this would not be a useful file system. The VFS also does not care how the file system works. It can store

its data however it wants on the hard drive, using whatever algorithms it wants. Instead of giving these functions behavior which is unexpected and not useful, the file system can be used to provide supplemental behavior which gives the user more flexibility and power in using the computer.

1.1.2 Versioning

A versioning file system keeps track of all file modifications and stores this information on disk. This involves changing functions such as *write* and *delete* to keep versioning data of files on the partition. Most file systems in use have what is known as "mutable state", which is the intuitive notion of what a file system should do. When a user edits a file, it is reasonable to expect that the file system has changed this file and that the next time it is opened, these changes are still there.

A versioning file system does not attempt to change a user's normal experience with his file system, but it adds a few more features. Instead of editing the file on disk, the file system create a new version of the file. [1] When the file is modified, the file system copies the data, and any updated information is only reflected in the new version of the file, not the old copy. The user then can still use his file system as a normal one, but the features a versioning system provides are still available if the user wishes to use them.

In essence, this means that old data is never irretrievably lost. If a user makes a modification or deletes a file he later wishes he had not, that action can be undone.

Branching is a feature that may be implemented on top of versioning. It allows a user to keep the history of a file system as a set of parent-child relationships. Instead of just viewing an older version of a file as a version at some point in time, files also carry the information of who its parent (or in some cases, parents) are and who its children (if any) are. The uses of branches are explored further in the second chapter.

1.1.3 Revision control software

To fully understand the concept of versioning, the reader should be familiar with the idea of revision control (also referred to as version control and source control), from which versioning originated. The features that users would like when using a versioning file system can all be found in revision control software. This software has been around for longer than

[1] To say that it creates a new copy of the file is not entirely true. This is an inefficient way of storing versions of a file. Modern versioning file systems creates a diff between the two versions. This is, however, an implementation detail, and to the user it seems as if there are two copies of the file. When discussing different versions of files, we will pretend there are really two copies of the file, since the two notations are functionally equivalent.

versioning file systems and provides many more features, due to its smaller scope and long development history.

Revision control is a practice which has its roots in software development. A traditional approach to revision control is to have a central repository (or server) and clients which are able to connect to the server. If they have the appropriate permissions, clients can "check-out" files from the repository or "check-in" files after updating them.

The major advantage of revision control software is that it keeps all previous versions of every file. Suppose user A is using revision control to write his paper. He has a paragraph which is included in the paper. He later decides the paragraph is wrong and deletes it. Much later, he decides that it really was a good paragraph, and would like it back. If he had been making consistent check-ins, he would go to his revision control software and ask to look at an earlier paragraph. He could then copy and paste it back into his most recent copy of the document.

The problem that arises from revision control is conflict. Since repositories are often multi-user, more than one user may try to make conflicting changes to a file. A typical conflict might go as follows. User A checks out document S and edits it. Around the same time, user B checks out the same document and edits it. After user A is finished with his editing, he sends his updated version back to the server. Later, user B finishes his editing, and also sends his updated version to the server. Unfortunately, the two users have made conflicting edits. The two of them have, perhaps, changed the same paragraph. Some of their other changes were to completely different parts of the file, but two of their changes in particular conflict with each other. At this point, instead of accepting user B's edit, the server tells user B that there has been a conflict and he must change it before resubmitting it. The server sends back the document after marking the areas where there was a conflict. If user B still wants to have his changes submitted, he must manually fix the conflict and resubmit it.

What has been described is the traditional form of revision control. This is how the original revision control systems (such as *RCS*[13]) work and many of the popular ones today (*CVS*[2], *SVN*[11]) use this model. Lately, distributed revision control systems (such as *git*[15] and *Darcs*[9]) have started to be used more frequently. They are similar to the traditional ones, but they do not have the concept of a central server. Instead, every local client is a server. Users are able to commit to their own repository, as well as pull changes from other servers. This has several advantages, such as being able to make commits when you don't have access to the main server (either due to lack of permission or lack of Internet connection) and allows users to keep their local repository however they

want. Creating a new branch does not require talking to a server and any user can create one. This is the approach which AlamoFS takes for branches. There is no need for some sort of central server. Users may create a branch at any time without much work. Branches are easy to make.

Looking at revision control software and versioning file systems, the similarities are immediately apparent. Despite this similarity, the functionality found in revision control far surpasses that in versioning file systems. Revision control is an essential part of software development, and software developers tend to be good at making software useful for themselves, and by extension for other software developers. Versioning file systems do not share this usage, and have as a result been mostly stagnant while huge advances have been made in revision control.

Because revision control's domain is smaller, it is much easier to work with. Problems that are easily solved for revision control are not as easy to implement in a file system. It is relatively easy to think of new features for revision control, but to take these changes and apply them to a file system often requires a lot of thought and modification.

1.2 Motivation

By storing old versions of files, a file system uses up more disk space, which is undesirable. This loss must be weighed against the functionality gained to determine if it is an acceptable trade-off. The file system developed in this thesis aims to minimize the amount of disk space used and maximize the functionality gained to make this an acceptable compromise. Not all users will agree, of course, so this file system is not intended to be a general purpose file system for all users to have on their computer.

With hard drive capacity growing every year, many users have more disk space than they know what to do with. With the exception of media files (such as videos, music, and pictures), most files users care about tend to be pretty small. Text files or papers might be on the order of a few kilobytes. By making a trade-off for space, an operating system is able to keep track of previous versions of a file.

For instance, a user might accidentally delete a file and later want it back. Without a versioning file system, the file may no longer be retrievable. Or instead of deleting the entire file, the user might have only deleted a paragraph that he cared about. In this case, the user can view previous versions of the file and find the deleted text.

Of course, the usefulness of these features depends on who is using them and the resources he has. For instance, users with older and smaller hard drives would find less use

for a versioning file system. The extra 10% disk space used in a large hard drive may not even be noticed by most users, but a user with a small device who is struggling to fit all the data he has on it would not like the added storage requirements. Similarly, a user who does not value the data on his computer much and instead uses it for temporary data or data which he does not care to track would find the extra requirements unacceptable.

Simply keeping around old versions of files is a basic feature that can be found in even the most basic of versioning file systems. This is the main purpose of these file systems, and does not require much programming to implement. The aim of AlamoFS is to increase the functionality possible with these features. For instance, even if a file system allows the user to look at previous versions, if it has a poor interface it is less useful. If every single version of the file must be scrutinized to locate the one sought, much time would be wasted. A good versioning file system must, therefore, make it easy to search previous versions. There are many commands a user might want to run, and an advanced versioning file system should make as many of them possible, without making the system difficult to use.

1.3 Priorities

Given the time frame of this project, it was impossible to implement everything an ideal versioning file system would have. The project had to prioritize different aspects of the file system.

The part that received the least priority was performance. There are all sorts of workloads that a file system might have depending on how it is used. It might be used by a user who does not do much besides surf the web and send e-mails, or maybe it is used by a power user who has his file system working to its maximum capability. Still, since file system use is often pretty minimal, giving performance a low priority was decided to be an acceptable trade-off.

Instead, the most attention was given to usability. It is possible to increase the performance of this file system while maintaining the same functionality, but that is not what this thesis explores. The thesis concerns itself with finding ways to enhance the user experience and allowing the user to do things he was unable to do with previously existing file systems.

1.4 Implementation

There were several options on how to implement versioning in a file system. The approach chosen was to implement AlamoFS using FUSE (Filesystem in USErspace)[12]. This it the approach that the Wayback file system takes[3].

The most obvious alternative would have been to implement the functionality directly into the file system. This would have provided the best performance and would perhaps be the "cleanest" approach. This choice was initially considered, but ultimately rejected for several reasons.

There were two possibilities for building the features into the file system. Either the file system could be designed from scratch or another file system could have been modified to add versioning support. Creating a new file system would have provided the most flexibility for choosing how to implement the new features, but this approach was not feasible for a two semester senior thesis. There have been many file systems designed in the past, and the time and energy that goes into creating them is large. It is of course possible to create one in this time frame, but its functionality would have suffered. Although it might not have been necessary to implement a fully functional file system to demonstrate the capabilities of a versioning file system, the pay-offs were not worth it.

Modifying another file system would have been easier than creating a new one, but this adds the problem of backwards compatibility. It would have been difficult to modify the file system in such a way that it could still work with non-modified versions of the file system, and so that non-modified versions of the file system could read the new one. And if the file system was still being changed by the maintainers, the modified file system might soon become too far behind. The reiser4 file system allows users to develop their own plugins to the file system. This approach might have worked, but since reiser4 has not yet been included in any major operating system and the file system's future is uncertain, this option was not pursued.

Another option considered was to use extended attributes (xattrs) available on different file systems. Many file systems allow the user to add meta-data to files. This is implemented by using key-value pairs. By calling the appropriate system call, the user can add meta-data to any file on the file system. For example, one could add a new key called "parent" and a value that pointed to the parent of the file. This was not chosen for two reasons. The first is performance. By keeping all the versioning information in one file instead of in many, the number of reads required for many operations (such as listing a directory) is cut down significantly. The other reason is usability. Many operating systems do not come with extended attributes enabled by default, so enabling them would require at the very

minimum a kernel recompilation, which is not an easy task for many users.

The approach taken provides the most usability. The user is not required to recompile the kernel, add any new kernel modules, have a specific file system, or even reboot their system. If the user decides he no longer wishes to have versioning functionality, it can easily be removed and the user can continue using the regular file system.

FUSE is a kernel module that is currently included in the stock Linux kernel. It allows developers to create a file system that runs entirely in userspace, instead of kernelspace. This allows AlamoFS to use all the libraries available in userspace, while kernelspace file systems are restricted to the kernel libraries. The approach is closer to the microkernel approach, which is much more modular.

By being stacked on top of a regular file system, users have a lot more flexibility. Since AlamoFS is not tied to any one file system, users can pick their file system of choice and run AlamoFS on top of it. This way, the user gets the advantages of a well-tested and developed file system as well as the advantages of versioning.

The disadvantages to this approach are mostly in performance. Since all file system calls must go through the FUSE interface, the approach adds a performance overhead.

1.5 Technical challenges

Adding advanced versioning features to a file system is not as simple as porting the code from the revision control software to the file system. Since the two are so fundamentally different, there are a lot of issues that come up. Some revision control features need to be adapted to be feasible in a file system, and some are probably impossible to adapt.

The main difference is that the two work on completely different scales. A project under version control might contain anywhere between one and many thousands of files (the Linux kernel, for example, has on the order of 20,000 files in the 2.6.24 version). A file system, on the other hand, could contain millions of files. Algorithms and techniques that work in revision control software might be too slow to use in a file system. Even worse, revision control software can take longer for a single operation than a file system can. If it takes two seconds to make a commit, that is acceptable because it does not happen often. Any operation that takes a few seconds in a file system would be unacceptable, since the user expects instantaneous response times for operations. If the file system took several seconds to list the files inside a directory, a user might soon become annoyed.

In revision control software, a commit is clearly defined. A commit only happens when the user requests one. The user can work on a project for as long as he likes, and then

only commit when he's ready. In a file system, it is less clearly defined. If the user is asked when to make commits, he will not make them very often and may forget to do so. In order to make sure the commits are made often and reliably, it is better to have the operating system make them itself.

Besides borrowing features from revision control, a versioning file system should also take advantage of features which would only be feasible for a file system. There are concepts which only work in revision control, and similarly ones which are only applicable to file systems.

Even after the features have been chosen, there are many issues to consider in the implementation (both visible and invisible to the user). There might be several approaches one could take to various problems, and each might have their own advantages. Some of these are best dealt with by allowing the user to choose what behavior he would like, perhaps through a configuration file. Others, however, require a design decision on the part of the file system implementer. These choices can profoundly affect the usability and usefulness of the file system, so they must be made carefully.

When the operating system stores previous versions of a file, how often does it make a new version? Is it better to make commits upon every change to the file, or once every few seconds or minutes? If a program makes dozens of edits in one second, should all of them be recorded, or only a subset?

How are previous versions of a file accessible to the user? Are they available in the same way as normal files are (e.g., will a directory have files like *foo.txt*, *foo.txtVERSION1*, etc.?) or will they be hidden behind an interface? If so, how? Should the user be able to edit them? How can they be searched? Is it possible to delete them?

Which files should be kept under version control? A user might watch a movie on his hard drive, and then want to remove it permanently. If it stays under version control, it would take up a lot of disk space. It is also unlikely that the user will want files he cannot modify (such as executables) to be kept under version control. How does the file system determine which files should be kept and which should not?

What should the file system do when the user runs out of disk space? Should it refuse to write any more data? When not using a versioning file system, the user can delete some unwanted files to make room. If this is not allowed, the user might be required to purchase a new hard drive.

We will show how AlamoFS handles these and similar problems in our discussion of design issues in the following chapters.

Chapter 2

Usage

The interface to AlamoFS currently consists of a command line program, which handles all the operations a user might want to perform. After mounting the file system, usage of the file system is not limited to the command line, of course. The user can use an editing program or a GUI (graphical user interface) file browser to access his files. Access to the versioning specific information is, however, limited to the command line. Searching for old files, reverting [1] old files, or branching can only be done from there.

It would be possible to construct a graphical interface to do all of this, but this was not done due to time constraints. Such an interface could make it easier for the user to visualize the structure of the older files on his file system and in general be simpler to use.

To illustrate the use of AlamoFS, this chapter will present an example command line session that shows many of its features. Sections will each show a part of the session which is relevant to the topic being discussed. For the most part, the sections should be self-contained.

2.1 Mounting

To begin using AlamoFS, the user must first mount the file system. This is done by calling the program with the name of the directory where the actual data should be stored and the directory to be mounted. That is, the user creates two directories, say *datadir* and *mountdir*. The user would then mount with the following arguments.

```
$ alamofs datadir mountdir
$
```

[1] "Revert" is a verb commonly used in the thesis. It means to replace the current file with an older version of that file.

To begin using the file system, the user would change directories to *mountdir*. The file system keeps all the data in *datadir*. When the directory is unmounted, *mountdir* will be empty, but the data will still be contained in *datadir*. The use of two different directories is unfortunate, but necessary due to technical limitations in FUSE. [2]

Assuming there were no errors during mounting (which would happen, for example, if the versioning data were corrupt), the user can then change directories into the *mountdir* directory, or access its files otherwise, and use the directory like any other.

2.2 Accessing as a regular file system

Until the user runs any of the versioning specific commands, the file system behaves under the same semantics as any other file system. All the regular user tools continue to work properly.

```
$ cd mountdir/
$ ls
$ touch file1
$ ls
file1
$ echo "Hello" >> file1
$ cat file1
Hello
$
```

2.3 Working with versioning

When the user is ready to use the versioning functionality, he may use the versioning commands available in AlamoFS. The first one a user is likely to use is the command to search for older files.

[2]To see why this is an issue, we must know what is really happening. Say the user asks to read the contents of one of the files. This call goes to the VFS, where it is intercepted by FUSE, which tells AlamoFS of the user's request. When the two directories are separate, AlamoFS then makes another call to the VFS, but asking to read the file in the data directory. Since this directory is not mounted as a AlamoFS file system, the call is handled by the native file system and it works as expected. If, on the other hand, the two are in the same directory, when AlamoFS make a request to read the file, the VFS will notice that the directory it is in is mounted as a AlamoFS file system, so it sends the new request to AlamoFS, leading to an infinite loop which results in the file system hanging. It might be possible to alter FUSE so that mounting from a single directory is possible, but having to use two directories is merely a small inconvenience.

2.3.1 Searching

AlamoFS provides the "–search" (or more simply "-s") option to find versions of a file which meet a specific criteria. When a user wants to find a previous version of a file, it is not feasible to look at every single version, since there could be hundreds of them. AlamoFS allows users to help it find the version they want. Users can specify a time range, text it should contain, text it should not contain, etc. For instance, a user might know that the version he wants was made after January 5, but before January 10.

Let us continue with our example session to demonstrate some of these features. Suppose the user makes a later modification to the file.

```
$ echo "World" >> file1
$ cat file1
Hello
World
$ ls
file1
$
```

There are now two versions of the file, though only one is displayed. AlamoFS creates a new version of a file on every modification with up to a granularity of one second. The original file contains the text "Hello", and the second file "Hello `\n` World". In normal usage, however, the user can only see the latest version of the file.

If we want to see the different versions of the file, we can try a search.

```
$ alamofs --search file1
Filename: /file1@1205568925; Time: 2008-03-15 04:15:25
Filename: /file1@1205568947; Time: 2008-03-15 04:15:47 (current)
$
```

We can see that there are two versions of the file, corresponding to each of the edits made. Note that even though the most recent file appears only as *file1* in our file system, it shows up as *file1@1205568947* in the search. While this may be a bit confusing, it is consistent with the fact that each file is immutable. So from the moment that the file is modified, it is given a permanent name which can never change. In regular usage, the current version of the file is always displayed with its non-versioned name.

The number that is after the "@" symbol is the Unix timestamp associated with the time at which the edit was made. A Unix timestamp is a measurement of how many

seconds have passed since midnight of January 1, 1970. While the starting point was mostly arbitrary, Unix timestamps are a commonly used manner of storing time values on Unix-like systems. Given that AlamoFS only keeps one version for edits within one second of each other, this timestamp uniquely identifies any version of the file.

Each file has an associated "versioning set", which is the set of all files related to it, either through being a child of the file, being in a branch off the file, etc. Performing a search on a file looks through the entire versioning set.

Instead of listing all versions of a file, the user may want to restrict the files listed. Some example commands are listed below.

```
$ alamofs --search file1 time <2008-03-15 04:15:30
Filename: /file1@1205568925; Time: 2008-03-15 04:15:25
$ alamofs --search file1 time =2008
Filename: /file1@1205568925; Time: 2008-03-15 04:15:25
Filename: /file1@1205568947; Time: 2008-03-15 04:15:47 (current)
$ alamofs --search file1 hastext World
Filename: /file1@1205568947; Time: 2008-03-15 04:15:47 (current)
$ alamofs --search file1 nothastext World
Filename: /file1@1205568925; Time: 2008-03-15 04:15:25
$ alamofs --search file1 time <2008
$
```

These examples are slightly contrived since there are only two versions of the file, but the search functionality is much more useful when there are many different versions.

2.3.2 Comparing versions

Cat

Once we have limited the number of versions we wish to look at, we can compare them. To look at the contents of a file, we use the –cat parameter with the versioned name of the file.

```
$ alamofs --cat file1@1205568925
Hello
$
```

The example shows one drawback of using the command line to do this. The names given to older files are a bit cumbersome. It is annoying to type and easy to mistype. A GUI

interface where the user can simply click on a file would be more convenient. AlamoFS allows users to tag files with a user supplied name (see section 2.3.4), but there several improvements that could be made to the command line interface. Refer to the last chapter for a discussion of the possibilities.

Diff

Using the cat command to print the contents of a file is usually not a good way to identify which version we want. It is often more convenient to see the difference between two files. To see the difference, the "diff" option can be used. This option outputs a unified diff of the two files.

```
$ alamofs --diff file1@1205568925 file1@1205568947
--- file1@1205568925    2008-03-14 04:15:25.000000000 +0000
+++ file1@1205568947    2008-03-14 04:15:47.000000000 +0000
@@ -1 +1,2 @@
 Hello
+World
$
```

Any lines which have been added start with a +, and any which have been removed start with - (there are none in this example). Here we can see that the only change was that a line with the word "World" was added to the second file.

A diff format is not the most readable way of presenting changes. Here again it might be better to have a GUI interface, or at least a more colorful interface. The *Vim* editor, for example, although it is a command line editor, is able to display diffs in a much more accessible manner.

2.3.3 Reverting changes

In some cases, using the diff command and the cat command on the file the user was looking for is sufficient. If the user has deleted a paragraph he wanted to get back, he could copy and paste it back to his document. In other cases, it is simpler to get back the entire document. This is possible with the "–revert" option.

```
$ alamofs --revert file1
$ cat file1
Hello
```

```
$ alamofs --revert file1@1205568947
$ cat file1
Hello
World
$
```

When only given the non-versioned name of a file, "–revert" brings back the parent of the file currently active. If a full name is given, then that version is brought back.

2.3.4 Tags

If a user knows that a particular version of a file will be needed later, he can tag the version with a name. The file will then also be accessible by the name filename@tag. Instead of writing down what the timestamp is or having to use a search to find it later, tags allow for an easy to remember way to refer to files.

```
$ alamofs --search file1
Filename: /file1@1205568925; Time: 2008-03-15 04:15:25
Filename: /file1@1205568947; Time: 2008-03-15 04:15:47 (current)
$ alamofs --tag file1@1205568925 firstcopy
$ alamofs --cat file1@firstcopy
Hello
$
```

2.4 Storage Policies

It is not always desirable to version all files in a file system. Some examples of files that might not be kept in versioning:

- Files which are read-only
- Files in a certain directory (e.g., */tmp/*)
- Files larger than a certain size
- Files that end in a certain extension

AlamoFS has three different storage policies which may be used: keep all, keep none, and keep one.

Keep all is the standard policy that is used. All changes to a file are versioned.

Keep none is also self-explanatory. No versioning information is kept at all. If the file is modified or deleted, the old data is gone permanently.

Keep one saves a copy of the file in case of deletions, but does not keep older versions when the file is modified.

To assign these policies to specific files, the user may edit */match.vdata* at the root of the file system. This is a configuration file which the file system reads each time it is mounted. The file system comes with a standard *match.vdata* which looks like:

```
size >3M : one
size >10M : none
read-only : none
indir /tmp/ : none
```

Any file which does not meet any of the criteria in the file is automatically set to "Keep All".

2.5 Branching

Branching allows users to keep a non-linear history of their files, and is one of the main features that separates AlamoFS from other versioning file systems. Usually, when versions of previous files are kept, they are thought of as being linear. At some time X, the file looked like this. At some later time Y, it looked like that. At time Z, etc. Each file has a parent, but each file has only one child as well.

When branches are introduced, each file has can have more than one child. So although you can trace any one file back to the beginning in a linear way, all the versions are ordered non-linearly.

In revision control software, branching is used often. If a user would like to work on an experimental feature without touching the main branch, he just creates a new branch. If he does not like the changes, he can discard them. If he does like them, he can merge them back into the main branch.

If a user would like to do this in a file system without support for branches, he must resort to ugly workarounds. He might copy an entire directory and then work on both directories separately. If he likes the changes he made in the copy, he can delete the original and keep the copy.

This approach has several drawbacks. What if the user makes changes to both directories and then wants to merge them? There is not an easy way of doing this. This approach

also clutters up the user's file system with several directories, even if he is not working on all of them.

Branching solves these problems.

The concept of branching is slightly different in a file system than in revision control. A branch in revision control is a copy of the entire contents of the repository, and the user is only in one branch at a time. A file system must support concurrent branches that do not necessarily contain all of the file system. A user might only want to make a branch of one file or one directory.

There is always a default branch in AlamoFS, namely the *master* branch. When the user makes a modification to a file not contained in another branch, it is considered to be part of the *master* branch.

The "–branch –list" command lists which branch the file is currently in.

```
$ ls
file1
$ alamofs --branch --list file1
master
$
```

Since no other branches have been made yet, the file is listed as being in the *master* branch. To actually change to a new branch, the "–branch branchname filename" command can be used.

```
$ alamofs --branch experimental file1
$ cat file1
Hello
World
$ alamofs --branch --list file1
experimental
$
```

The above example shows how *file1* is originally in branch *master*, and afterward is in branch *experimental*. The contents of the file do not change after the new branch is made. Let us make some modifications to a file, and then switch back to the master branch.

```
$ echo "I am in the new branch" >> file1
$ cat file1
Hello
```

```
World
I am in the new branch
$ alamofs --branch master file1
$ cat file1
Hello
World
$
```

After switching back to the main branch, the changes made in the branch are gone. Next we will make some changes to the master branch, and then merge the two.

```
$ echo "I am part of the master branch" >> file1
$ cat file1
Hello
World
I am part of the master branch
$ alamofs --branch --diff experimental
--- file1@1205747192    2008-03-17 05:46:32.000000000 +0000
+++ file1@1205747170    2008-03-17 05:46:10.000000000 +0000
@@ -1,3 +1,3 @@
 Hello
 World
-I am part of the master branch
+I am in the new branch
$ alamofs --merge experimental
There was at least one conflict in the merge. Please edit file1.rej and
then copy it to file1.
$ cat file1.rej
Hello
World
<<
I am part of the master branch
<<
>>
I am in the new branch
>>
$
```

If there had been no conflict (for instance, if the line "I am part of the master branch" had been added to the start of the file, rather than the end), there would have been no *.rej* file and *file1* would have contained the contents of the merge. But since the two attempted to modify the same line in different ways, there was no way for the file system to merge them successfully.

Branching a directory works similarly.

```
$ mkdir foo
$ cd foo
$ echo "I am file a" > fileA
$ echo "I am file b" > fileB
$ alamofs --branch dirbranch foo/
$ alamofs --branch --list foo/
dirbranch
$ echo "Some extra text" >> fileA
$ cat fileA
I am file a
Some extra text
$ alamofs --branch master foo/
$ cat fileA
I am file a
$
```

Merging a directory is done recursively.

One form of branching that AlamoFS does not allow is nested branches. You can not make a branch in one directory, then another one in a subdirectory or parent directory. Although such a feature might be useful, it usually just leads to complications which would confuse a user. The one exception is the main branch. Branches in the *master* branch do not count as nested branches.

To see why recursive branches are not permitted, let us look at an example.

```
$ mkdir dir
$ cd dir
$ echo "Hello" >> file1
$ alamofs --branch --list dir/file1
$ alamofs --branch new dir
$ alamofs --branch experimental dir/file1
```

```
$ echo "World" >> file1
$ alamofs --branch master dir
$
```

It is unclear how this should work. Should the contents of *file1* be reverted or not? We did switch back to the master branch, which includes *file1*, but branch *experimental* is still active on *file1*.

Although these ambiguities could be resolved by picking some conventions and sticking to them, the benefits would be marginal. It is a feature that would only be used by power users, and could potentially confuse less experienced users. Having your data unexpectedly be reverted when you were not expecting it would be disastrous.

2.6 Other file modifications

Although file writes are probably the most useful feature a user would want a file system to keep track of, there are a few others as well.

2.6.1 Deletion

Deletion of a file is done through the *unlink* system call. This is the function that is called when, for example, the user runs the *rm* command or deletes a file in his file browser.

When AlamoFS receives an *unlink* call, it simply removes the copy of the file which is viewable to the user and notes internally that it was deleted. All the versions of the file are still available on the disk. AlamoFS allows users to undelete their files, using the "–revert" command.

```
$ ls
file1
$ rm file1
$ ls
$ alamofs --search file1
Filename: /file1@1205568925; Time: 2008-03-15 04:15:25
Filename: /file1@1205568947; Time: 2008-03-15 04:15:47
$ alamofs --revert file1
$ ls
file1
$
```

Note that no file is listed as being current.

The command "–revert" also works with directories, but is slightly different. The system call to remove a directory is *rmdir*. This call differs from *unlink* in that it only works if the directory is empty. When a user requests that a directory be deleted, the command he issues (e.g., "rm -r dir/") recursively goes through the directory and its subdirectories, unlinks each file, and then calls *rmdir* on each empty subdirectory, and finally on the directory that the user wanted deleted. Thus, AlamoFS must process several requests when the user asks to delete a directory.

The lack of an atomic *rmdir* system call makes it hard to implement an undelete command for directories that works in the way the user would expect. When he reverts his directory, it will be empty and filled with deleted files. If he wishes to have those back, he most undelete all of those as well.

It would be possible to add some logic to the file system to try to figure out which deletions were made as a result of deleting the directory. If the deletion of a file in the directory and the deletion of the directory were done within a few seconds of each other, it might infer that the two were related. This might not be easy, however. What if the "rm -r" command is interrupted in the middle, and then continued later? Or what if the user is deleting a directory with many files in it? Such a deletion might take several seconds, and if AlamoFS's threshold is too small, some of the files might be missed.

One solution already available is that the user can ask the file system to restore the directory to the way it was at a specific point in time. This is cumbersome since the user would have to find the time when the directory was deleted, then subtract a few seconds or minutes from it for good measure, and then run the command.

2.6.2 Renaming

Renaming a file (which is semantically the same as moving a file) adds some complications to versioning, and could have been done in several ways. Unlike recursive branches, renaming is a feature which is essential to file systems and cannot be ignored. The author attempted to use the most intuitive option in each decision involved.

A few of the issues that came up will be discussed. When a file is moved, what should be done with the versioning data? Should it be moved over to the new location? Or should it be copied? Or maybe the new version should not contain any of the versioning data from the previous location.

Before that question can be answered, some boundary cases must be looked at. The important ones are renaming a file to another file which already exists or used to exist, or

creating a new file where the old file used to exist. What should happen in these cases? If we're moving a file to a location where a file used to exist, what happens if the user later tries to perform a search on that file? Should results for both the old and new file match, or just one? Should the two files be considered part of the same versioning set?

The decision taken is that moving a file should still keep it in the same versioning set, and the fact that two files shared a name at two points in time does not make them part of the same set. Shown is an example of these principles.

```
$ ls
file1
$ alamofs --search file1
Filename: /file1@1205568925; Time: 2008-03-15 04:15:25
Filename: /file1@1205568947; Time: 2008-03-15 04:15:47 (current)
$ echo "I am file 2" >> file2
$ alamofs --search file2
Filename: /file2@1205568970; Time: 2008-03-15 04:16:10
$ mv file1 file2
$ ls
file2
$ alamofs --search file1
Filename: /file1@1205568925; Time: 2008-03-15 04:15:25
Filename: /file1@1205568947; Time: 2008-03-15 04:15:47
Filename: /file2@1205569023; Time: 2008-03-15 04:17:13 (current)
$ alamofs --search file2
Filename: /file1@1205568925; Time: 2008-03-15 04:15:25
Filename: /file1@1205568947; Time: 2008-03-15 04:15:47
Filename: /file2@1205569023; Time: 2008-03-15 04:17:13 (current)

Filename: /file2@1205568970; Time: 2008-03-15 04:16:10
$ mv file2 file1
$ alamofs --search file1
Filename: /file1@1205568925; Time: 2008-03-15 04:15:25
Filename: /file1@1205568947; Time: 2008-03-15 04:15:47
Filename: /file2@1205569023; Time: 2008-03-15 04:17:13
Filename: /file1@1205569043; Time: 2008-03-15 04:17:33 (current)
$
```

When the user searches for *file1*, it is unambiguous which versioning set is being referred to. The user is looking for the file which is named *file1* or any name it was later changed to. So when the user searches for the file, he gets results from both names.

A search for *file2* is, however, ambiguous. Was the user hoping to to find a version of the file that was originally called *file2* or a version of the file that was at some point renamed to *file2*? Since there is no way to tell, AlamoFS searches for both. In order to tell the user which files belong to which versioning set, it separates the two sets by a blank line.

2.7 File meta-data

AlamoFS does not keep versioning information for file meta-data, which includes attributes such as owner, permissions, access time, etc. The user can modify the meta-data of the currently active file, but not those of previous versions. AlamoFS is mainly concerned with the preservation of data, not meta-data. Still, tracking these changes is a possible improvement to the file system.

Chapter 3

Implementation

Since AlamoFS is a userland file system, it uses the already existing file operations to contain its versioning data, so most of the code deals with implementing versioning features. If you compare this to a regular file system, which can be tens of thousands of lines of code, most of which is dedicated to figuring out how to store the data on the disk, it is easy to see how much this implementation saves on code.

3.1 Code organization

The code of AlamoFS comes out to about 3,000 lines and is divided up into four main files: *main.c*, *versioning.c*, *match.c*, and *branch.c*, along with their associated header files, as well as *misc.c* which has a few functions that do not fit in the others.

The file *main.c* implements the file system functions which FUSE calls. Whenever a write request is made, etc., *main.c* handles it. It mostly consists of wrapper code, and is not particularly interesting.

The "–search" command and related functions are held in the file *match.c*. The file also deals with parsing the *match.vdata* file and checking what policy to assign to a given file depending on its contents.

The majority of the versioning code is contained in *versioning.c*, with the exception of the branch code. It is responsible for reading and writing the versioning data from the disk, as well as managing the associated data structures.

The code in *branch.c* could have logically been included in *versioning.c*, but since that file already contained several hundreds lines of codes, it was decided that the branching code would be contained in its own file.

The functions in *misc.c* are mostly uninteresting. They are functions that implement a linked list, that get the current time, and other such tasks. For the most part they are

unrelated to versioning.

3.2 On-disk format

Besides the older versions of files which are normally hidden from the user, the file system also keeps several other files hidden which the user cannot modify. [1] The files are stored in a human-readable format.

The first file is *super.vdata*, which sits at the root of the file system. It is analogous to the superblock found in disk-based file systems. It contains general parameters and configuration data for the entire file system, although it is currently mainly used to keep track of branches.

Another file which sits at the root is *log.vdata*. This file contains a list of every versioning-related operation that is performed. Whenever a file is written to, renamed, deleted, etc., it is written to this log. While this increases the amount of spaced used, the storage costs are minimal compared to all the extra data the file system must keep in order to keep previous versions. In order to cut down on the number of writes performed to this file, the file system only updates it every few seconds or after a certain amount of time has passed. It is possible that if the computer is unexpectedly shut-off, some updates may be missing from the log. [2]

The last type of file is the generic *.vdata* file associated with every file (or, rather, the versioning set of every file). This keeps track of all the versions of one file. While this is redundant with the *log.vdata* file, these files are kept for performance purposes. The file *log.vdata* grows quickly in a file system, and it would be too much of a performance penalty to read the entire file or waste too much memory to keep it there, so it acts as a sort of cache.

The *log.vdata* file is used mostly for file system wide searches, such as when the user wants to find all files modified between two dates. The file-specific *.vdata* files are used when the user wants to do something with files which are all part of the same versioning set.

[1] It is technically possible for a user to view and modify these files if the file system is not mounted, but this might corrupt the file system if done improperly and should generally not be done.

[2] A possible solution to this problem is to perform a scan of all *.vdata* files in the file system and use those to check if there are missing entries in *log.vdata*, although this would be slow.

3.2.1 Versioned files

Unlike most versioned file systems, AlamoFS stores all older files uncompressed on disk, and without storing them as a smaller diff file. If a large file just has a few bytes changed in between versions, it should not be necessary to copy the entire file. However, this is how it is done in the current implementation of AlamoFS.

Each version of a file is permanently identified by its path and time, although the most recent file is available by just the path. AlamoFS manages this by first creating the versioned file with the full name, and then creating a hard link to it, so the data is only used once. When the user makes the *getattr* call on a file, the file system decreases the hard link count returned by one to hide this fact.

3.2.2 .vdata files

The *.vdata* files are the most commonly used files, and are read on many operations. Below is an example of a simple *.vdata* file:

```
current experimental

branch master /file1@1208293420
branch experimental /file1@1208293455

tag mytag /file1@1208293420

0 1208293420 none /file1@1208293420
1 1208293455 /file1@1208293420 /file1@1208293455
```

The first part of the file lists the branches and which file they each point to, and the second part lists the tags.

The last part of the file contains a list of every file in the versioning set. The first number indicates whether it is the currently active file or not. The second number is the timestamp associated with the file. The third field is the parent of the file, and the final one is the path where it is located.

3.2.3 log.vdata

The *log.vdata* file consists of a list of logs entries which look like:

```
1208293420 create /file1@1208293420
1208293455 write /file1@12082934255
```

3.2.4 super.vdata

The *super.vdata* file contains some general file system data. Currently it is only used to keep track of branches, but further information may be added in the future. The last number on each line indicates whether the branch is fully updated.

```
branch master / 1208293420 1
branch experimental /file1 1208293455 0
```

3.3 Data Structures

Internally, AlamoFS uses several data structures to organize information. There is usually a close correspondence between these data structures and the on-disk files. For example, when reading from a *.vdata* file, AlamoFS uses both *struct version_data* and *struct version_data_file*.

```
struct version_data_file {
        char *path;
        char *parent;

        long int time;
        int is_current;
};

struct version_data {
        char *path;

        struct version_data_file *current_vdf;
        struct array *vdf_list;

        struct branch *current_branch;
        struct array *branch_list;
};
```

A *struct version_data_file* contains the information for one specific version of a file. It contains information on its location, parent, time of modification, a flag that tells whether it is the current file or not, and pointers to its children.

There is one *struct version_data* generated per versioning set, and it is mostly used as a container of pointers to *struct version_data_file*s.

Due to the similarity of tags and branches, a *struct branch* can be used for both of them. This struct is rather simple.

```
struct branch {
        char *name;
        char *path;

        int is_branch;
        int needs_update;
};
```

The only difference is that *is_branch* is set to 1 for branches and 0 for tags.

The same data structure is used by the superdata to keep track of branches, although it is used differently. When used in relation with a *.vdata* file, the path refers to the particular version in the file to which it points, such as /filename@timestamp. When used by the superblock, it refers to the general location of the file or directory, not a specific version. The path in this case would look like /filename. The *needs_update* member is used to allow lazy reversions.

The rest of the data structures used in the file system are pretty basic and are mainly things like simple linked lists.

3.4 Algorithms

In essence, there are three sorts of algorithms that touch the data structures: populating data structures from information on disk, changing them based on the user's command, and writing them back to disk. The first and third ones are basically the same process but in reverse, so only the first will be discussed.

3.5 Reading from disk

The *.vdata* files must be read each time a new version of a file is created, and when the user is making use of the various versioning commands available. For the most part, this is

pretty simple. The lines are read in with the *fscanf* function, broken up into the different tokens, and then put into the data structures.

Some complications arise due to renames, which cause the data for one file to appear in different places on disk. If a file is moved from *file1* to *file2*, part of the versioning data will be in *file1.vdata*, and part of it in *file2.vdata*. Usually, this does not matter. When reading or writing to a *.vdata* file, the most important piece of information is knowing what the current file is named.

In some scenarios, it becomes necessary to open the *.vdata* files associated with the other names of a file. Suppose that a user has a file *file1* in which he has some data. At some later point, he moves it to *file2* and makes some more updates. Eventually, he deletes it. One day, he decides to get this file back, so he does a search. But during this time, he forgot that at one point he renamed the file *file2*, so he just does a search for *file1*. If our algorithm did not look at the new location of the file, he would not have a complete list of files, and more importantly, would not have the most recent ones.

For this reason, some operations may have to read several *.vdata* files. To know when a file has been renamed, AlamoFS notes that a file was renamed in the original *.vdata* file. In the scenario above, *file1.vdata* might contain a line that reads:

```
0 1208293420 /file1@1208293410 /file2@1208293420
```

When reading this from the disk, AlamoFS notes that the path of the file differs from that of its parent, so if necessary, it can read the *.vdata* file for the new one.

3.6 Managing data

The algorithms presented in this section are the main part of the file system. Luckily, due to the nature of versioning control, they are not very complicated. Since structures on disk are immutable, most of the algorithms just consist of adding a line to the end of the file system or updating where a branch is pointing to.

3.6.1 Searching

The first step in doing a search is determining whether the user is searching for one file, or for many. If the user gives an absolute path to a file, such as */file1*, we only need to search for one file. If he does a more general search, either not specifying a file at all or searching for a file regardless of where it is in the file system, a larger search will need to be done.

In the first case, AlamoFS reads the associated *.vdata* file (and others as well in case of a rename). In the latter, it reads *log.vdata*. Regardless of which one was done, the algorithm proceeds the same. There are now a large number of files, and we need to match the ones which meet the user's criteria.

Although criteria can be matched in any order, AlamoFS uses a specific order to increase efficiency. If a user specifies both a time range and text that the file should contain, then the time criterion should be applied first. Since the time is already included in the *.vdata* file and is ordered chronologically, the algorithm just needs to find the first file that is after the minimum time and the last file that is before the maximum time range.

After that step, each file must be opened, and each line scanned to check if it contains the specified text (in the case of "–hastext") or if it does not have it (for "–nothastext").

The list of files is kept as a linked list. Once all the files which do not meet the criteria have been taken out, the remaining files are printed.

3.6.2 Revert

Reverting a file is a simple process. All that is necessary is to hard link to the appropriate version and to set that version as active in the *.vdata* file.

Reverting a directory is a bit more complicated. Say the user wants to revert a directory to the way it was at a specific time. To update every *.vdata* file in each of its subdirectories could potentially take a long time, especially if the entire file system is being reverted. Since any such delays decrease the usefulness of branches, AlamoFS takes a lazy approach to reversions.

The only time that the user actually cares about which version is active in a directory is when he actually goes to that directory, tries to list all the files in it, or tries to open a file in it. Until then, it does not matter. To save time, AlamoFS does not update the current file in a directory which has been reverted until the user accesses the directory. By keeping track of what time the user tried to revert to, the next time the directory is accessed, AlamoFS can make the appropriate version active.

3.6.3 Diffs and merges

AlamoFS cheats in these implementations. It would be beneficial to have these algorithms in the core of the source code, but AlamoFS instead runs the *diff* and *patch* programs which are included in all Linux distributions (as far as the author knows). The *LibXDiff* [5] library may be suitable.

3.6.4 Branching

Implementing branching was the most difficult feature to write. The key fact to realize is that a branch is in essence a tag. AlamoFS does not need to keep all the data separated according to what branch they are in, the file system only needs a tag for each branch that points to the current file.

The algorithms for branching a single file are easy. Say the user is in the master branch, and branches a file called *file1* into a branch called *experimental*, and then make a modification to it. All that needs to be done is to add some information to *file1*'s *.vdata* file. First, the file system tells the superdata to record the fact that the user is performing a branching. Then it adds a line to the *.vdata* file which would indicate which file is currently active in the new branch. The line would look like "branch experimental file1@time" .

When the user makes a modification in the experimental branch, AlamoFS reads the *.vdata* file to see where the branch currently points. The new version of the file is this file with the modifications the user just made. After the user switches back to the master branch, AlamoFS looks up in the *.vdata* file where the master branch points, and that becomes the new active file.

When modifying the *.vdata* file, the file system needs to check if the branch being switched to already exists. We scan through the list of branches which point to the file. If any of them match the branch name being switched to, we make that the current branch. If the file does not yet exist, then we add a new entry to it.

Branching a directory is more complicated. Since there are potentially many files inside a directory, it would be slow to open every *.vdata* file and edit all of them. As with reversion, AlamoFS takes a lazy approach in this case. The *.vdata* files are not modified until it is necessary.

Suppose the user branches a directory *dir1* into a branch named *experimental*. The only action the file system takes at this point is to inform the superdata that the new branch was created. If at some later point the user modifies a file in the directory, then AlamoFS will note in the file's *.vdata* where the *experimental* branch points to. This is possible because until any modifications, all of the current files will be the current ones in the master branch.

This algorithm works until we change back to the *master* branch. If all of the files in the directory have been modified since then there is nothing further to be done. But if not all of the files have been modified, their *.vdata* files will not yet contain a mention of *experimental*. The next time that the user makes a modification in the *master* branch, we use the above trick of having the current file in the *experimental* be the one active in the

master branch. So again we take the approach of not updating the files until necessary.

Switching branches can be seen as a sort of revert, so the same algorithm can be used. Whenever the user attempts to list a directory or access a file in a directory which is out-of-date, we first check the superdata to check if we had reverted a branch in the directory and not yet updated the contents. If so, AlamoFS updates the *.vdata* files at this time.

Chapter 4

Conclusion

AlamoFS was written to explore the possibilities in designing a user-friendly file system with support for branching history. It is not a finished file system, but most of the major features have been included.

4.1 Possible Improvements

4.1.1 Security

During the design of AlamoFS, security was not considered as much as should be the case in a file system designed for practical use.

Besides the usual security considerations, a versioning file system must handle the question about the security of older files. Several situations which were not given proper consideration are:

What happens if a user would like to permanently delete a file? Perhaps the file contained some secret information which he would not like to be on his hard drive. In a versioning file system, deleting the file is not enough, and there is no way to delete the older files without unmounting the file system and then manually editing it. Even if it were automated by AlamoFS, this would not be a simple problem. Do you delete every single previous file? What about ones that were branched off of years ago? What if you just want to delete one particular version? Do you have to re-parent everything that had it as a parent, or leave those hanging?

4.1.2 Storage policies

Other versioning file systems have more complex storage policies than AlamoFS does. AlamoFS allows you to assign one of three policies to files, depending on certain criteria.

They can be fully versioning, not versioned at all, or have only one version at a time.

In contrast, other versioning file systems have a greater variety of choices. For example, one which might be useful is "Keep Landmarks". This strategy, introduced by the Elephant File System[10], is a bit more clever in deciding which files to delete. It uses an algorithm to figure out if it is a major change or not.

Another policy is "Keep Recent", which can be generalized to "Keep In Time Range". A user might decide that there is a file where he would like recent changes to be kept, since he might save too hastily, but does not want the file system to bother keeping older versions since he knows he will never look at them again.

These policies would not be difficult to implement. One of the main reasons they were kept out is that the author wanted to stick to the idea that data should never be lost. It is conceivable that a policy such as "Keep Recent" would throw away data the user would later want back. The policy used by AlamoFS could lead to this if the user was too liberal in choosing what types of files not to version, but short of keeping every single bit of data, this approach is not feasible due to space constraints.

4.1.3 Input parsing

All the input to AlamoFS must be well-formed. Instead of spending time writing an input parser that would handle a variety of formats, the number of formats was kept small.

For example, in supplying a time for files to search, dates must be in the format of YY[YY[-MM[-DD[HH[:MM[:SS]]]]]], as opposed to any of the other commonly used time formats. There are several other occurrences of this in the file system.

This violates the maxim of "Be liberal with input, strict with output".

4.1.4 File naming

The permanent names given to files are of the form "filename@timestamp". It may improve usability to change this naming convention. As mentioned earlier, typing out such a name is cumbersome and prone to errors. While a GUI interface would help with the problem, there are several other possibilities.

One solution comes from the *git* project, which is a revision control project. In *git*, commits are referred to by their SHA1 hash. (A hash is a function which maps a file to string.) Whenever a commit is made, *git* calculates the SHA1 of the commit, and this hash becomes the permanent identifier of the file. Hashes change drastically between commits, so the first few characters of a hash are usually enough to uniquely identify a commit. Since

the hashes are composed of hexadecimal digits, just the first four characters of the hash allow us to refer up to $16^4 = 65,536$ different commits (although the actual number before a clash occurs is likely to be much smaller). Timestamps, however, do not work nearly as well for this. The first few digits are probably going to be the same between versions, especially for versions created on the same day. In the examples from the second chapter, often only the last two digits differed. The reason this was not implemented is that SHA1 hashes are expensive to calculate. Since it would have to be done every time a file was modified, this was deemed too large of a performance penalty.

Another solution is to give the files a simpler naming convention. Some versioning file systems give old versions simple names such as "file1@1" to indicate that this is the first version of the file. The second version would be "file1@2", etc.

One last possibility is to add a "tab completion" feature to the user's shell. Tab completion is a fairly common shell feature. Instead of having to type out the file name, "foo-bar.baz", the user can type foo`<tab>`, and if those few letters uniquely identify the file, the shell will fill in the rest of the filename. Several modern shells, such as *zsh*, provide powerful tab completion functionality which can do much more than that. One example of this is an *scp* completion feature. After a user authenticates himself to the remote machine, he can tab complete files that are sitting on the remote computer. It would therefore be possible to add functionality to the shell that would query AlamoFS and let it tab complete the full versioning names of files. The drawbacks of this are that it requires modification of other programs on the computer, only works with certain shells, and requires that both of them be updated when the user updates to a newer version of either one.

4.1.5 Graphical interface

The interface to AlamoFS would be more usable if it were presented as a graphical interface. The command line in Linux is a powerful tool for effectively performing tasks, but many users prefer to use something graphical.

AlamoFS does not require a command line to use the file system regularly, but it is required to use any of the versioning functionality. If a user regularly uses a graphical file browser to access his files, it would be a pain to switch to the command line to search for files or create a new branch.

An example of a program that does something similar is Apple's *Time Machine*[14]. Time Machine is not a versioning file system, but is a backup utility. Nonetheless, the interface which Apple provides is easy to use, and a similar feature for AlamoFS would go a long way toward making it more usable for the general user.

4.1.6 On-disk storage

For efficiency purposes, the latest version of a file should be kept as-is on the disk. For older versions of files, however, this is not necessary. Huge storage gains can be made by either compressing the data or recording only the parts that changed from the previous or next files.

AlamoFS does neither of these. Since one of the major drawbacks of a versioning file system is the extra space it uses, some ability to reduce the amount used would be greatly welcomed.

4.1.7 Performance

As mentioned in the first chapter, performance was given little priority in the design of AlamoFS. Still, for moderate workloads, AlamoFS is quite usable. There is never any noticeable slowdown during regular usage during regular usage.

Regardless, the file system has not been tested on a heavy workload, where many operations are happening at once, and it is possible that there would be serious performance degradation. If so, the algorithms AlamoFS uses must be improved to remove the bottlenecks.

4.1.8 Thread safety

AlamoFS is not completely thread-safe. It is not unusual for the OS to make several requests in a short period of time, before the file system has finished all the previous requests made. There are some rare situations in which two simultaneous file system calls may result in a corrupt file system, and these problems need to be addressed.

The author considered placing some sort of lock which would have solved the problem. The idea can be added to the list of the features which should have been included.

4.2 Future

AlamoFS was not designed with the intent that it would one day become a polished file system used by many people. It was written as an academic project to explore the design decisions involved in such a file system. While it would be great if other programmers would work on AlamoFS to make it a robust file system, the author does not expect this to happen.

Instead, the author hopes that others who are designing or modifying a versioning file system will have a look at AlamoFS and this thesis as inspiration on where to go with it. AlamoFS is far from the file system the author would look to see in use one day, but the author hopes others can build on the ideas explored in AlamoFS.

Bibliography

1. Jeff Bonwick. ZFS. http://opensolaris.org/os/community/zfs/, 2008.

2. Concurrent Versions System (CVS). http://www.nongnu.org/cvs/, 2008.

3. Brian Cornell, Peter A. Dinda, and Fabian E. Bustamante. Wayback: A user-level Versioning File System for Linux, *Proceedings of the 2004 USENIX Annual Technical Conference*, 2004.

4. Kiran-Kumar Muniswamy-Reddy. A Versatile And User-Oriented Versioning File System, *Proceedings of the Third USENIX Conference on File and Storage Technologies*, March 2004.

5. Davide Libenzi. LibXDiff. http://www.xmailserver.org/xdiff.html, 2008.

6. J.J. Kistler and M. Satyanaryanan. Disconnected operations in the Coda file system. *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, October, 1991.

7. Kirby McCoy. *VMS File System Interals*, Digital Press, 1990.

8. Z.N. Peterson and R. C. Burns. Ext3cow: The design, Implementation, and Analysis of Metadata for a Time-Shifting File System. Tech. Report HSSL-2003-03, Compuer Science Department, The John Hopkins University, 2003.

9. David Roundy. Darcs. http://darcs.net/, 2008.

10. D.S. Santry, M.J. Feeley, N.C. Hutchinson, A.C. Veitch, R.W. Carton and J. Ofir. Deciding When to Forget in the Elephant File System. *Proceedings of the Seventeenth ACM Symposium on Operating System Principles*, December, 1999.

11. Subversion. http://subversion.tigris.org/, 2008.

12. Miklos Szeredi. Filesystem in USErspace. http://fuse.sourceforge.net/, 2008.

13. Walter Tichy, RCS: A System for Version Control. *Software — Practice and Experience*, 15(7), July 1985.

14. Time Machine. http://www.apple.com/macosx/features/timemachine.html, 2008.

15. Linux Torvards and Junio Hamano. Git – Fast Version Control System. http://git.or.cz/, 2008.

Appendix A

Source Code

The entire source code for AlamoFS is presented in this appendix. To download the most recent version of AlamoFS, please visit http://www.simons-rock.edu/ erk/alamofs/

AlamoFS is released under the BSD license. The terms of this license may be found in LICENSE.txt included in the download.

versioning.h

```
/*
 * This structure provides the data for one particular commit
 */
struct version_data_file {
        char *path;
        char *parent;

        long int time;
        int is_current;
};

/*
 * This structure provides the versioning data for a versioning set
 */
struct version_data {
        char *path;

        struct version_data_file *current_vdf;
        struct array *vdf_list;

        struct branch *current_branch;
        struct array *branch_list;
};

/*
 * Our file system doesn't have a superblock, exactly, but it does have
 * some global information about the file system which is kept in this
 * struct
 */
struct super_data {
        struct match_data_file *mdf;
        struct array *branch_list;

        char *data_dir;
};

/*
 * Various strings associated with a file name
 */
struct file_data {
        char *path;
        char *name;
        char *full;
        char *real_path;
        long int time;
};


#define WRITE 1
#define TRUNC 2
#define RENAME 3
#define RMDIR 4
#define UNLINK 5

#define DATA_FILE_SUFFIX ".vdata"

char *get_data_file_name(char *real_path);
char *get_name_after_delim(char *path, char delim);
char *get_path_and_time(char *path, long int time);
int is_hidden_file(char *str);
struct file_data *get_file_data(char *path);

int copy_file(char *old_path, char *new_path);

struct version_data *read_version_data(char *real_path, int just_headers);
int read_pointers(FILE *fp, struct version_data *vd);
int read_version_info(FILE *fp, struct version_data *vd);
void skip_headers(FILE *fp);
int write_version_data(struct version_data *vd);
int write_pointers(FILE *fp, struct version_data *vd);
int write_version_info(FILE *fp, struct version_data *vd);
```

```
int add_new_version_data(struct version_data *vd);
int append_vdf(struct version_data *vd, struct version_data_file *vdf);

int write_log();
int make_current(struct version_data *vd, struct version_data_file *vdf);
int revert_file(int argc, char *argv[]);
struct version_data_file *add_rename(struct version_data *vd, char *path, long int time);

int init_super();
void update_super_data();
int write_super();
struct branch *get_branch_from_super_data(char *branch_name);

/* memory */
void free_operations(struct operation *op);
void free_version_data_file(void *ptr);
void free_version_data(struct version_data *vd);
void free_file_data(struct file_data *fd);

/* Debug functions */
void print_version_data(struct version_data *vd);
void print_version_data_file(struct version_data_file *vdf);
```

versioning.c

```
#define _XOPEN_SOURCE 500

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <dirent.h>
#include <errno.h>
#include <unistd.h>
#include <sys/mman.h>

#include "main.h"
#include "versioning.h"
#include "branch.h"
#include "misc.h"

/*
 * When a file is modified, we must make a copy of it. This function
 * does this. It copies the file at old_path to new_path. The method is
 * based on  section 14.9 of Advanced Programming in the UNIX
 * Environment, Second Edition
 */

int copy_file(char *old_path, char *new_path) {

        int fd1 = open(old_path, O_RDONLY);
        int fd2 = open(new_path, O_RDWR | O_CREAT | O_TRUNC);

        if(fd1 == -1 || fd2 == -1) {
                printf("Could not cp\n");
                return -1;
        }

        struct stat *statbuf = c_malloc(sizeof(struct stat));
        fstat(fd1, statbuf);

        lseek(fd2, statbuf->st_size -1, SEEK_SET);
        write(fd2, "", 1);

        void *src = mmap(0, statbuf->st_size, PROT_READ, MAP_SHARED, fd1, 0);
        void *dst = mmap(0, statbuf->st_size, PROT_READ | PROT_WRITE, MAP_SHARED, fd2, 0);

        if(src == MAP_FAILED || dst == MAP_FAILED) {
                printf("Count not cp\n");
                return 0;
        }

        memcpy(dst, src, statbuf->st_size);
        fchmod(fd2, statbuf->st_mode);

        close(fd1);
        close(fd2);

        return 0;
}


/*
 * Given the path to a file, returns the path to its .vdata file
 */

char *get_data_file_name(char *real_path) {
        int isdir = is_dir(real_path);
        int len = strlen(real_path) + strlen(DATA_FILE_SUFFIX) + 1 + isdir;

        char *path = c_malloc(len);
        *path = 0;

        strcat(path, real_path);
        if(isdir) {
                strcat(path, "/");
```

```
        }

        strcat(path, DATA_FILE_SUFFIX);
        path[len - 1] = 0;

        return path;
}

/*
 * Initializes the vd and vdf structures from the .vdata file
 * on disk.
 *
 * If `just_headers` is set to 1, the function will only read the branch data.
 * Note that if any modifications are to be written to disk, the entire file
 * must be read.
 *
 * Returns NULL if the file is in the wrong format.
 */

struct version_data *read_version_data(char *real_path, int just_headers) {

        char *version_path = get_data_file_name(real_path);
        struct version_data *vd = c_malloc(sizeof(struct version_data));

        vd->vdf_list = init_array(free_version_data_file);
        vd->branch_list = init_array(free_branch);
        vd->path = strdup(real_path);
        vd->current_branch = 0;


        FILE *fp = fopen(version_path, "r");
        if(!fp) { /* No versioning data yet */
                vd->current_branch = c_malloc(sizeof(struct branch));
                vd->current_branch->name = strdup("master"); /* todo? */
                vd->current_branch->path = get_path_and_time(vd->path, get_current_time());
                vd->current_branch->is_branch = 1;
                add_element(vd->branch_list, vd->current_branch);
                return vd;
        }

        read_pointers(fp, vd);
        if(!just_headers) {
                read_version_info(fp, vd);
        }

        fclose(fp);

        return vd;
}

/*
 * Read all the branch and tags in the vdf file and update the vd struct
 * appropriately
 */

int read_pointers(FILE *fp, struct version_data *vd) {
        char *type = c_malloc(BUF_SIZE);
        char *name = c_malloc(BUF_SIZE);
        char *path = c_malloc(BUF_SIZE);
        char *current = c_malloc(BUF_SIZE);

        if(fscanf(fp, "current %s\n", current) != 1) { /* Current branch name */
                printf("Corrupt versioning file\n");
                return -1;
        }


        while(fscanf(fp, "branch %s %s\n", name, path) == 2) {
                struct branch *br = c_malloc(sizeof(struct branch));
                br->name = strdup(name);
                br->path = strdup(path);
                br->is_branch = 1;
                add_element(vd->branch_list, br);
```

```
                if(strcmp(name, current) == 0) {
                        vd->current_branch = br;
                }
        }

        while(fscanf(fp, "tag %s %s\n", name, path) == 2) {
                struct branch *br = c_malloc(sizeof(struct branch));
                br->name = strdup(name);
                br->path = strdup(path);
                br->is_branch = 0;
                add_element(vd->branch_list, br);
                if(strcmp(name, current) == 0) {
                        vd->current_branch = br;
                }
        }

        free(type);
        free(path);
        free(current);
        free(name);

        return 0;
}

/*
 * Read and parse information in a .vdata file regarding the different
 * versions of a file. Called by read_version_data. It adds a vdf struct
 * to 'vd's vdf_list member.
 *
 * Returns -1 on error.
 */
int read_version_info(FILE *fp, struct version_data *vd) {
        int type;
        long int time;
        char *parent = c_malloc(BUF_SIZE);
        char *path = c_malloc(BUF_SIZE);

        while(fscanf(fp, "%d %li %s %s\n", &type, &time, parent, path) == 4) {
                struct version_data_file *vdf = c_malloc(sizeof(struct version_data_file));

                if(type == 1) { /* branch */
                        vd->current_vdf = vdf;
                        vdf->is_current = 1;
                }
                else if (type == 0) { /* tag */
                        vdf->is_current = 0;
                }
                else {
                        printf("Corrupt versioning file\n");
                        return -1;
                }

                vdf->parent = strdup(parent);
                vdf->time = time;
                vdf->path = strdup(path);

                add_element(vd->vdf_list, vdf);
        }

        return 0;
}

void skip_headers(FILE *fp) {
        char *tmp = c_malloc(BUF_SIZE);
        while(fscanf(fp, "current %s\n", tmp));
        while(fscanf(fp, "branch %s %s\n", tmp, tmp));
        while(fscanf(fp, "tag %s %s\n", tmp, tmp));
        free(tmp);
}

/*

void print_version_data(struct version_data *vd) {
```

```
        printf("Path: %s\n", vd->path);
        printf("Pos: %d : Size: %d\n", vd->pos, vd->size);

        printf("Current:\n");
        print_version_data_file(vd->current);

        printf("Others:\n");
        int i;
        for(i = 0; i < vd->pos; i++) {
                print_version_data_file(vd->old[i]);
        }
}

void print_version_data_file(struct version_data_file *vdf) {
        printf("Path: %s\n", vdf->path);
        printf("Parent: %s\n", vdf->parent);
        printf("Time: %li\n", vdf->time);
        printf("Current? %s\n\n", vdf->is_current ? "yes" : "no");
}
*/


/*
 * Writes the vd structure to the disk in the appropriate format.
 *
 * Returns -1 on error.
 */

int write_version_data(struct version_data *vd) {
        char *version_path = get_data_file_name(vd->path);

        FILE *fp = fopen(version_path, "w");
        if(!fp) {
                printf("Could not open version file\n");
                return -1;
        }

        fprintf(fp, "current %s\n", vd->current_branch->name);


        fclose(fp);

        unlink(vd->path);
        link(vd->current_vdf->path, vd->path);

        return 0;
}

/*
 * Called by write_version_data. Writes the branches and tags of the file.
 *
 * Returns -1 on error.
 */
int write_pointers(FILE *fp, struct version_data *vd) {
        int size = get_num_elements(vd->branch_list);
        int i;

        for(i = 0; i < size; i++) {
                struct branch *br = get_nth_element(vd->branch_list, i);
                if(br->is_branch) {
                        fprintf(fp, "branch %s %s\n", br->name, br->path);
                }
                else {
                        fprintf(fp, "tag %s %s\n", br->name, br->path);
                }
        }

        return 0;
}

/*
 * Called by write_version_data. Writes the different versions of a file.
 *
```

```
 * Returns -1 on error.
 */
int write_version_info(FILE *fp, struct version_data *vd) {
        int size = get_num_elements(vd->vdf_list);
        int i;

        for(i = 0; i < size; i++) {
                struct version_data_file *vdf = get_nth_element(vd->vdf_list, i);
                fprintf(fp, "%d %li %s %s\n", vdf->is_current, vdf->time, vdf->parent, vdf->path);
        }

        return 0;
}

/*
 * Frees a vdf structures
 */

void free_version_data_file(void *ptr) {
        struct version_data_file *vdf = ptr;
        if(vdf->path) {
                free(vdf->path);
        }
        if(vdf->parent) {
                free(vdf->parent);
        }
        free(vdf);
}

/*
 * Frees a vd structure and all of its vdf members
 */

void free_version_data(struct version_data *vd) {
        if(vd->vdf_list) {
                free_array(vd->vdf_list);
        }
        if(vd->branch_list) {
                free_array(vd->branch_list);
        }
        if(vd->path) {
                free(vd->path);
        }
        free(vd);
}

/*
 * This function is called when a file is modified.
 * It takes care of modifying the vd structure appropriately.
 * It also modifies the path names if necessary
 */

int add_new_version_data(struct version_data *vd) {
        struct version_data_file *vdf = c_malloc(sizeof(struct version_data_file));
        vdf->time = get_current_time();
        vdf->path = get_path_and_time(vd->path, vdf->time);


        if(get_num_elements(vd->vdf_list) == 0) {
                vdf->parent = strdup("none");
                vdf->is_current = 1;
                vd->current_vdf = vdf;
                free(vd->current_branch->path);
                vd->current_branch->path = strdup(vdf->path);
                add_element(vd->vdf_list, vdf);
                return 0;
        }

        vdf->parent = strdup(vd->current_vdf->path);
        vd->current_branch->path = strdup(vdf->path);

        add_element(vd->vdf_list, vdf);
```

```
        vd->current_vdf->is_current = 0;
        vd->current_vdf = vdf;
        vdf->is_current = 1;

        return 0;
}

/*
 * Changes the currently active file to the one specified by ‘vdf’
 * This file does not write these changes to disk, though it does
 * change the handlink of the non-versioned file.
 */

int make_current(struct version_data *vd, struct version_data_file *vdf) {
        unlink(vd->path);
        link(vdf->path, vd->path);

        vd->current_vdf->is_current = 0;
        vd->current_vdf= vdf;
        vdf->is_current = 1;

        free(vd->current_branch->path);
        vd->current_branch->path = strdup(vdf->path);

        return 0;
}


/*
 * Given a path and time, allocates and returns a string in the form of
 * path@time. If time == 0, it gets the current timestamp and uses that.
 */

char *get_path_and_time(char *path, long int time) {
        char *pnt = c_malloc(strlen(path) + TIME_BUF_SIZE);

        if(time == 0) {
                time = get_current_time();
        }

        sprintf(pnt, "%s@%li", path, time);

        return pnt;
}

/*
 * Checks if the file name is not allowed in our file system
 * Currently, files are not allowed to have an @ or the string
 * .vdata
 */

int is_hidden_file(char *str) {
        if(strstr(str, "@") || strstr(str, ".vdata")) {
                return 1;
        }

        return 0;
}

int init_super() {
        sd = c_malloc(sizeof(struct super_data));

        char *real_path = get_real_path("super.vdata");

        sd->branch_list = init_array(free_branch);

        FILE *fp = fopen(real_path, "r");
        if(!fp) {
                printf("Could not open super file\n");
                return -1;
        }

        char *name = c_malloc(BUF_SIZE);
```

```
        char *path = c_malloc(BUF_SIZE);

        while(fscanf(fp, "branch %s %s\n", name, path) == 2) {
                struct branch *br = c_malloc(sizeof(struct branch));
                br->name = strdup(name);
                br->path = strdup(path);
                br->is_branch = 1;
                add_element(sd->branch_list, br);
        }

        return 0;
}

int write_super() {
        char *real_path = get_real_path("super.vdata");

        FILE *fp = fopen(real_path, "w");
        if(!fp) {
                printf("Could not open super file\n");
                return -1;
        }

        int size = get_num_elements(sd->branch_list);
        int i;

        for(i = 0; i < size; i++) {
                struct branch *br = get_nth_element(sd->branch_list, i);
                fprintf(fp, "branch %s %s\n", br->name, br->path);
        }
        fclose(fp);

        return 0;
}

struct branch *get_branch_from_super_data(char *branch_name) {
        int size = get_num_elements(sd->branch_list);
        int i;

        for(i = 0; i < size; i++) {
                struct branch *br = get_nth_element(sd->branch_list, i);
                if(strcmp(br->name, branch_name) == 0) {
                        return br;
                }
        }

        return 0;
}

/*
void update_super_data() {
        sd->num_ops++;

        long int time = get_current_time();

        if(sd->num_ops > 10 || time - sd->time_of_last_log > 30) {
                write_log();
        }
}


int add_to_log(struct operation *op) {
        if(sd->first_op == 0) {
                sd->first_op = op;
                sd->last_op = op;
        }
        else {
                sd->last_op->next = op;
                op->next = 0;
        }

        return 0;
}
```

```
int write_log()  {
        char *log_path = get_real_path("/log.vdata");
        FILE *fp = fopen(log_path, "a");
        printf("writing log file\n");
        if(!fp) {
                printf("Could not open log file: %d\n", errno);
                return -1;

        }

        struct operation *op = sd->first_op;

        while(op) {
                switch(op->type) {
                        case WRITE:
                                fprintf(fp, "%li write %s %s\n", op->time, op->path1, op->parent);
                                break;
                        case TRUNC:
                                fprintf(fp, "%li trunc %s %s\n", op->time, op->path1, op->parent);
                                break;
                        case RENAME:
                                fprintf(fp, "%li rename %s %s %s\n", op->time, op->path1, op->path2, op->parent);
                                break;
                        case RMDIR:
                                fprintf(fp, "%li rmdir %s %s\n", op->time, op->path1, op->parent);
                                break;
                        case UNLINK:
                                fprintf(fp, "%li unlink %s %s\n", op->time, op->path1, op->parent);
                                break;
                        default:;
                }

                op = op->next;
        }

        fclose(fp);

        free_operations(sd->first_op);
        sd->first_op = 0;
        sd->last_op = 0;
        sd->time_of_last_log = get_current_time();
        sd->num_ops = 0;

        return 0;
}

void free_operations(struct operation *op) {

        if(op) {
                if(op->path1) {
                        free(op->path1);
                }
                if(op->path2) {
                        free(op->path2);
                }
                if(op->parent) {
                        free(op->parent);
                }

                free_operations(op->next);
                free(op);
        }
}
*/

/*
 * --revert filename
 *
 *  If a  non-versioned file name is given, revert to the parent of the file.
 *  Otherwise, revert to the version given.
 */
int revert_file(int argc, char *argv[]) {
        if(argc != 1) {
```

```
                printf("Bad arguments to --revert\n");
                return -1;
        }

        struct file_data *fd = get_file_data(argv[0]);
        struct version_data *vd = read_version_data(fd->real_path, 0);

        if(!fd->time && strcmp(vd->current_vdf->parent, "none") == 0) {
                printf("File %s does not have a parent\n", argv[0]);
                return -1;
        }


        int size = get_num_elements(vd->vdf_list);
        while(--size + 1) {
                struct version_data_file *vdf = get_nth_element(vd->vdf_list, size);
                if(fd->time && vdf->time == fd->time) {
                        make_current(vd, vdf);
                        write_version_data(vd);
                        break;
                }
                else if (!fd->time && strcmp(vdf->path, vd->current_vdf->parent) == 0) {
                        make_current(vd, vdf);
                        write_version_data(vd);
                        break;
                }
        }

        free_file_data(fd);

        if(size == -1) {
                printf("File %s does not exist\n", argv[0]);
                return -1;
        }

        return 0;

}

/*
 * A struct file_data breaks down a (possibly versioned) file name. It
 * contains the directory, time, file name without directory, etc.
 */
struct file_data *get_file_data(char *path) {
        struct file_data *fd = c_malloc(sizeof(struct file_data));

        int len = strlen(path);

        int slash_pos = get_last_pos(path, '/');
        if(slash_pos == -1) {
                fd->path = 0;
                slash_pos = 0; /* Needed to get the rest right */
        }
        else {
                fd->path = c_malloc(slash_pos + 2); /* Need to include the trailing / */
                strncpy(fd->path, path, slash_pos + 1);
        }


        char *name = c_malloc(len - slash_pos + 1);
        strncpy(name, path + slash_pos, len - slash_pos);

        int at_pos = get_last_pos(name, '@');

        if(at_pos == -1) {
                fd->time = 0;
                fd->name = strdup(name);
                fd->full = strdup(path);
        }
        else {
                fd->time = atoi(name + at_pos + 1);
                fd->name = c_malloc(at_pos + 1);
                strncpy(fd->name, name, at_pos);
```

```
                fd->full = c_malloc(slash_pos + at_pos + 1);
                strncpy(fd->full, path, slash_pos + at_pos);
        }


        fd->real_path = get_real_path(fd->full);
        free(name);
        return fd;
}

/*
 * Frees memory for a struct file_data
 */

void free_file_data(struct file_data *fd) {
        if(fd->full) {
                free(fd->full);
        }
        if(fd->path) {
                free(fd->path);
        }
        if(fd->name) {
                free(fd->name);
        }
        if(fd->real_path) {
                free(fd->real_path);
        }

        free(fd);
}


/*
 * Called whenever a rename is performed. [todo?]
 */
struct version_data_file *add_rename(struct version_data *vd, char *path, long int time) {
        struct version_data_file *vdf = c_malloc(sizeof(struct version_data_file));
        vdf->path = get_path_and_time(path, time);
        vdf->parent = strdup(vd->current_vdf->path);
        vdf->time = time;
        vdf->is_current = 0;

        vd->current_vdf->is_current = 0;
        vd->current_vdf = 0;
        vd->current_branch = 0;

        return vdf;
}
```

branch.h

```
/* also works for tags */
struct branch {
        char *name;
        char *path;

        int is_branch;
        int needs_update;
};

int handle_branch(int argc, char *argv[]);
int branch_dir(char *branch_name, char *path);
int branch_file(char *branch_name, char *path);
int switch_to_branch(struct version_data *vd, char *branch_name);
struct branch *get_current_branch(char *real_path);
struct branch *get_branch(char *real_path, char *branch_name);
struct branch *add_old_branch(struct version_data *vd, char *branch_name);
struct branch *copy_branch(struct branch *br);

int handle_merge(int argc, char *argv[]);
int merge_dir(struct branch *br, char *real_path);
int merge_file(char *orig_real_path, char *new_real_path);

char *resolve_tag(char *path);
void tag_file(int argc, char *argv[]);

/* memory */
void free_branch(void *ptr);

/* debug */
void print_branch(char *path);
```

branch.c

```
#define _XOPEN_SOURCE 500

#include <fuse.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <dirent.h>
#include <errno.h>
#include <sys/time.h>

#include "main.h"
#include "versioning.h"
#include "match.h"
#include "branch.h"
#include "misc.h"

/*
 * Creates a new branch. The logic is quite different for files and
 * directories, so the two cases are handled seperately.
 */
int handle_branch(int argc, char *argv[]) {
        if(argc <= 1) {
                printf("Not enough arguments to --branch\n");
                return -1;
        }
        if(argc == 2 && strcmp(argv[0], "--list") == 0) {
                print_branch(argv[1]);
        }
        else if(argc == 3 && strcmp(argv[0], "--diff") == 0) {
                char *args[2];
                char *real_path = get_real_path(argv[1]);
                struct branch *br = get_current_branch(real_path);
                args[0] = get_fake_path(br->path);

                br = get_branch(real_path, argv[2]);
                args[1] = get_fake_path(br->path);
                diff_files(2, args);
                free(real_path);
        }
        else if(argc == 2) {
                char *real_path = get_real_path(argv[1]);
                int isdir = is_dir(real_path);

                if(isdir) {
                        branch_dir(argv[0], argv[1]);
                }
                else {
                        branch_file(argv[0], argv[1]);
                }
        }
        else {
                printf("Bad arguments to --branch\n");
                return -1;
        }

        return 0;
}

/*
 * Returns the currently active branch associated with the file name.
 * The file name must not be versioned.
 */
struct branch *get_current_branch(char *real_path) {
        struct version_data *vd = read_version_data(real_path, 1);
        struct branch *br = copy_branch(vd->current_branch);
        free_version_data(vd);

        return br;
}
```

```
/*
 * Given a file name a branch name, returns the branch struct
 * associated with it. File name must not be versioned.
 */
struct branch *get_branch(char *real_path, char *branch_name) {
        struct version_data *vd = read_version_data(real_path, 1);

        int size = get_num_elements(vd->branch_list);
        int i;

        for(i = 0; i < size; i++) {
                struct branch *br = get_nth_element(vd->branch_list, i);
                if(br->is_branch && strcmp(br->name, branch_name) == 0) {
                        struct branch *ret_branch = copy_branch(br);
                        free_version_data(vd);
                        return ret_branch;
                }
        }

        free_version_data(vd);

        return 0;
}

/*
 * Prints the currently active branch given a file name.
 *
 * Used, for example, by the --branch --list command
 */
void print_branch(char *path) {
        char *real_path = get_real_path(path);


        if(is_dir(real_path)) {
                int size = get_num_elements(sd->branch_list);
                int i;
                for(i = 0; i < size; i++) {
                        struct branch *br = get_nth_element(sd->branch_list, i);
                        if(is_subdir(br->path, path)) {
                                printf("%s\n", br->name);
                                free(real_path);
                                return;
                        }
                }
                printf("master\n");
                free(real_path);
                return;
        }

        struct branch *br = get_current_branch(real_path);

        if(br == 0) {
                printf("File does not exist\n");
                return;
        }

        printf("%s\n", br->name);

        free(real_path);
}

/*
 * Creates a new branch in the directory.
 * This involves telling the super data where you are creating the branch.
 */
int branch_dir(char *branch_name, char *path) {
        char *real_path = get_real_path(path);

        int size = get_num_elements(sd->branch_list);
        int i;

        for(i = 0; i < size; i++) {
```

```
                struct branch *br = get_nth_element(sd->branch_list, i);
                if(is_subdir(br->path, path)) {
                        printf("Recursive branches are not permitted\n");
                        free(real_path);
                        return -1;
                }
        }

        struct branch *new_branch = c_malloc(sizeof(struct branch));
        new_branch->is_branch = 1;
        new_branch->path = strdup(path);
        new_branch->name = strdup(branch_name);
        add_element(sd->branch_list, new_branch);
        write_super();

        free(real_path);
        free_branch(new_branch);

        return 0;
}

/*
 * Creates a new branch on the file.
 * Updates the .vdata file with this fact.
 */
int branch_file(char *branch_name, char *path) {
        int size = get_num_elements(sd->branch_list);
        int i;

        for(i = 0; i < size; i++) {
                struct branch *br = get_nth_element(sd->branch_list, i);
                if(is_subdir(br->path, path)) {
                        printf("Recursive branches are not permitted\n");
                        return -1;
                }
        }

        char *real_path = get_real_path(path);
        struct version_data *vd = read_version_data(real_path, 0);

        switch_to_branch(vd, branch_name);
        write_version_data(vd);

        free_version_data(vd);
        free(real_path);

        return 0;
}

/*
 * Switches the currently active branch on a file.
 */
int switch_to_branch(struct version_data *vd, char *branch_name) {
        int i;
        int size = get_num_elements(vd->branch_list);
        int found = 0;
        struct branch *br = 0;
        for(i = 0; i < size; i++) {
                br = get_nth_element(vd->branch_list, i);
                if(strcmp(br->name, branch_name) == 0) {
                        found = 1;
                        break;
                }
        }

        if(found) { /* existing branch */
                vd->current_branch = br;
                size = get_num_elements(vd->vdf_list);
                for(i = 0; i < size; i++) {
                        struct version_data_file *vdf = get_nth_element(vd->vdf_list, i);
                        if(strcmp(vdf->path, br->path) == 0) {
                                make_current(vd, vdf);
                        }
```

```
                }
        }
        else {
                struct branch *new_branch = c_malloc(sizeof(struct branch));
                new_branch->name = strdup(branch_name);
                new_branch->path = strdup(vd->current_vdf->path);
                new_branch->is_branch = 1;
                vd->current_branch = new_branch;
                add_element(vd->branch_list, new_branch);
                add_element(sd->branch_list, br);
                write_super();
        }

        return 0;
}

/*
 * todo
 */
struct branch *add_old_branch(struct version_data *vd, char *branch_name) {
        struct branch *br = malloc(sizeof(struct branch));
        br->name = strdup(branch_name);
        br->path = vd->current_branch->path;
        br->is_branch = 1;
        add_element(vd->branch_list, br);
        return br;
}

/*
 * The first function called when the user requests a merge.
 * File and directory merges are performed in different functions.
 */
int handle_merge(int argc, char *argv[]) {
        if(argc != 1) {
                printf("Bad input to --merge\n");
                return -1;
        }

        struct branch *br = get_branch_from_super_data(argv[0]);
        if(br == 0) {
                printf("Branch %s does not exist\n", argv[0]);
                return -1;
        }

        char *real_path = get_real_path(br->path);

        struct branch *br2 = get_branch(br->path, argv[0]);
        char *real_path2 = get_real_path(br2->path);

        if(is_dir(real_path)) {
                merge_dir(br, real_path);
        }
        else {
                merge_file(real_path, real_path2);
        }

        free_branch(br);
        free_branch(br2);
        free(real_path);
        free(real_path2);

        return 0;
}

/*
 * Recursively merges a directory.
 */
int merge_dir(struct branch *br, char *real_path) {
        DIR *dp;
        struct dirent *de;

        dp = opendir(real_path);
```

```
        if (dp == NULL)
                return -errno;

        while ((de = readdir(dp)) != NULL) {
                char *new_real_path = c_malloc(strlen(real_path) + strlen(de->d_name) + 2);
                new_real_path = strdup(real_path);
                strcat(new_real_path, "/");
                strcat(new_real_path, de->d_name);
                if(is_dir(de->d_name)) {
                        merge_dir(br, new_real_path);
                }
                else {
                        struct branch *cur_branch = get_branch(br->name, new_real_path);
                        char *n_real_path = get_real_path(cur_branch->path);
                        merge_file(n_real_path, new_real_path);
                }
        }

        closedir(dp);

        return 0;
}

/*
 * Merges a file. This is called both when a file merge is specifically
 * requested and by merge_dir
 */
int merge_file(char *orig_real_path, char *new_real_path) {

        char *cmd = malloc(BUF_SIZE);
        sprintf(cmd, "diff %s %s", new_real_path, orig_real_path);

        FILE *fp = popen(cmd, "r");
        FILE *fp2 = popen("patch", "w");
        if(!fp) {
                printf("Cold not run diff\n");
                return -1;
        }
        if(!fp2) {
                printf("Could not run patch\n");
                return -1;
        }

        char *buf = malloc(BUF_SIZE);

        while(fgets(buf, BUF_SIZE - 1, fp)) {
                fprintf(fp, "%s", buf);
        }

        fclose(fp);
        fclose(fp2);
        free(buf);

        return 0;
}

/*
 * Frees the memory associated with a branch
 */
void free_branch(void *ptr) {
        struct branch *br = ptr;
        if(br->name) {
                free(br->name);
        }
        if(br->path) {
                free(br->path);
        }
        free(br);
}

/*
 * Deep copies a branch
 */
```

```
struct branch *copy_branch(struct branch *br) {
        struct branch *new_branch = c_malloc(sizeof(struct branch));
        new_branch->is_branch = br->is_branch;
        new_branch->name = strdup(br->name);
        new_branch->path = strdup(br->path);
        new_branch->needs_update = br->needs_update;

        return new_branch;
}


/*
 * Adds a tag to a file. Edits the .vdata file to add it.
 */
void tag_file(int argc, char *argv[]) {
        if(argc != 2) {
                printf("Invalid arguments to --tag\n");
                return;
        }

        char *path = argv[0];
        char *tag = argv[1];

        struct file_data *fd = get_file_data(path);
        struct version_data *vd = read_version_data(fd->full, 0);

        int size = get_num_elements(vd->branch_list);

        int found = 0;

        struct branch *br;
        while(--size + 1) {
                br = get_nth_element(vd->branch_list, size);
                if(br->is_branch == 0 && strcmp(br->name, tag) == 0) {
                        found = 1;
                        break;
                }
        }

        if(fd->time == 0) {
                path = vd->current_vdf->path;
        }

        free_file_data(fd);

        if(found == 0) {
                struct branch *new_tag = c_malloc(sizeof(struct branch));
                new_tag->path = strdup(path);
                new_tag->name = strdup(tag);
                new_tag->is_branch = 0;
                add_element(vd->branch_list, new_tag);
                free_branch(br);
        }
        else {
                printf("Tag %s already exists\n", tag);
                return;
        }

        free_version_data(vd);
}

/*
 * The user may refer to a file by it's tag. Given a file name, the function
 * returns it's real version name, whether it is a tag or not.
 *
 * Returns 0 on error.
 */

char *resolve_tag(char *path) {
        int pos = get_last_pos(path, '@');
        if(pos == -1) {
                return strdup(path);
        }
```

```
        char *end = path + pos;

        struct version_data *vd = read_version_data(path, 1);

        int size = get_num_elements(vd->branch_list);
        int i;

        for(i = 0; i < size; i++) {
                struct branch *br = get_nth_element(vd->branch_list, i);
                if(!br->is_branch && strcmp(end, br->name) == 0) {
                        return strdup(br->path);
                }
        }

        return strdup(path);
}
```

main.h

```
struct super_data *sd;

void cat_file(int argc, char *argv[]);
int diff_files(int argc, char *argv[]);
void print_help();
```

main.c

```
#define FUSE_USE_VERSION 26

#define _XOPEN_SOURCE 500

#include <fuse.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <dirent.h>
#include <errno.h>
#include <sys/time.h>

#include "main.h"
#include "versioning.h"
#include "match.h"
#include "branch.h"
#include "misc.h"

/* Global variables which tells if the super data has been initiazlied yet */
int sd_init = 0;


static int alamofs_getattr(const char *path, struct stat *stbuf)
{
        int res;

        char *real_path = get_real_path(path);

        res = lstat(real_path, stbuf);

        free(real_path);

        if (res == -1)
                return -errno;

        return 0;
}

static int alamofs_access(const char *path, int mask)
{
        int res;

        char *real_path = get_real_path(path);

        res = access(real_path, mask);

        free(real_path);

        if (res == -1)
                return -errno;

        return 0;
}

static int alamofs_readlink(const char *path, char *buf, size_t size)
{
        int res;

        char *real_path = get_real_path(path);

        res = readlink(real_path, buf, size - 1);

        free(real_path);

        if (res == -1)
                return -errno;

        buf[res] = '\0';
        return 0;
```

```
}


/*
 * When getting a listing of files in a directory, we ignore files that are
 * used for versioning, so they won't show up in ls, etc.
 */

static int alamofs_readdir(const char *path, void *buf, fuse_fill_dir_t filler,
                       off_t offset, struct fuse_file_info *fi)
{
        DIR *dp;
        struct dirent *de;

        (void) offset;
        (void) fi;

        int size = get_num_elements(sd->branch_list);
        int i;

        struct branch *br;
        int needs_update = 0;

        for(i = 0; i < size; i++) {
                br = get_nth_element(sd->branch_list, i);

                if(is_subdir(br->path, path) && strcmp(br->name, "master")) {
                        if(br->needs_update) {
                                needs_update = 1;
                        }
                        break;
                }
        }

        char *real_path = get_real_path(path);

        dp = opendir(real_path);


        if (dp == NULL)
                return -errno;

        while ((de = readdir(dp)) != NULL) {

                if(is_hidden_file(de->d_name)) {
                        continue;
                }

                if(needs_update) {
                        char *new_real_path = malloc(strlen(real_path) + strlen(de->d_name) + 2);
                        new_real_path = strdup(real_path);
                        strcat(new_real_path, "/");
                        strcat(new_real_path, de->d_name);

                        if(!is_dir(new_real_path)) {
                                struct version_data *vd = read_version_data(new_real_path, 0);
                                add_old_branch(vd, br->name);
                                write_version_data(vd);
                        }
                }
                struct stat st;
                memset(&st, 0, sizeof(st));
                st.st_ino = de->d_ino;
                st.st_mode = de->d_type << 12;
                if (filler(buf, de->d_name, &st, 0))
                        break;
        }

        free(real_path);

        closedir(dp);
        return 0;
}
```

```
static int alamofs_mknod(const char *path, mode_t mode, dev_t rdev)
{
        int res;
        char *real_path = get_real_path(path);

        res = mknod(real_path, mode, rdev);

        free(real_path);

        if (res == -1) {
                return -errno;
        }

        return 0;
}

static int alamofs_mkdir(const char *path, mode_t mode)
{
        int res;
        char *real_path = get_real_path(path);

        res = mkdir(real_path, mode);

        free(real_path);

        if (res == -1) {
                return -errno;
        }

        return 0;
}

static int alamofs_unlink(const char *path)
{
        int res;
        char *real_path = get_real_path(path);

        res = unlink(real_path);

        free(real_path);

        if (res == -1) {
                return -errno;
        }

        return 0;
}

static int alamofs_rmdir(const char *path)
{
        int res;
        char *real_path = get_real_path(path);

        res = rmdir(real_path);

        free(real_path);

        if (res == -1) {
                return -errno;
        }

        return 0;
}

static int alamofs_symlink(const char *from, const char *to)
{
        int res;
        char *real_from = get_real_path(from);
        char *real_to = get_real_path(to);

        res = symlink(real_from, real_to);
```

```
        free(real_from);
        free(real_to);

        if (res == -1) {
                return -errno;
        }

        return 0;
}

static int alamofs_rename(const char *from, const char *to)
{
        int res;
        char *real_from = get_real_path(from);
        char *real_to = get_real_path(to);

        /* struct version_data *vd = read_version_data(from); */

        /* struct version_data_file *vdf = add_rename(vd, real_to, time); */

        res = rename(real_from, real_to);

        free(real_from);
        free(real_to);

        if (res == -1) {
                return -errno;
        }

        return 0;
}

static int alamofs_link(const char *from, const char *to)
{
        int res;
        char *real_from = get_real_path(from);
        char *real_to = get_real_path(to);

        res = link(real_from, real_to);

        free(real_from);
        free(real_to);

        if (res == -1) {
                return -errno;
        }

        return 0;
}

static int alamofs_chmod(const char *path, mode_t mode)
{
        int res;
        char *real_path = get_real_path(path);

        res = chmod(real_path, mode);

        free(real_path);

        if (res == -1) {
                return -errno;
        }

        return 0;
}

static int alamofs_chown(const char *path, uid_t uid, gid_t gid)
{
        int res;
        char *real_path = get_real_path(path);

        res = lchown(real_path, uid, gid);
```

```
        free(real_path);

        if (res == -1) {
                return -errno;
        }

        return 0;
}

/*
 * The file system doesn't keep versioning info for truncates, since
 * a truncate is usually followed or proceeded by another operation.
 *
 * Perhaps it would be better to do a check to see if any other modifications
 * are made within a second of a truncation.
 */

static int alamofs_truncate(const char *path, off_t size)
{
        int res;
        char *real_path = get_real_path(path);

        res = truncate(real_path, size);

        free(real_path);

        if (res == -1) {
                return -errno;
        }

        return 0;
}

static int alamofs_utimens(const char *path, const struct timespec ts[2])
{
        int res;
        struct timeval tv[2];
        char *real_path = get_real_path(path);

        tv[0].tv_sec = ts[0].tv_sec;
        tv[0].tv_usec = ts[0].tv_nsec / 1000;
        tv[1].tv_sec = ts[1].tv_sec;
        tv[1].tv_usec = ts[1].tv_nsec / 1000;

        res = utimes(real_path, tv);

        free(real_path);

        if (res == -1) {
                return -errno;
        }

        return 0;
}

/*
 * We first need to check if the file is a versiosing file, in which case
 * we want to pretend it doesn't exist. Access to old files should only
 * be done through the interface provided, not the regular file system
 * interface.
 *
 * An error of EACCES is returned if this is the case. Otherwise, it
 * behaves as a regular open call would.
 */

static int alamofs_open(const char *path, struct fuse_file_info *fi)
{
        int res;

        char *real_path = get_real_path(path);
        printf("in open: %s\n", real_path);
```

```
        res = open(real_path, fi->flags);

        if (res == 0) {
                free(real_path);
                return -errno;
        }
        else {
                if(is_hidden_file(real_path)) {
                        printf("wat\n");
                        return -EACCES;
                }
        }


        free(real_path);

        close(res);
        return 0;
}

/*
 * Like with alamofs_open, we need to check if it is an old file or not
 */
static int alamofs_read(const char *path, char *buf, size_t size, off_t offset,
                    struct fuse_file_info *fi)
{
        int fd;
        int res;
        char *real_path = get_real_path(path);

        (void) fi;
        fd = open(real_path, O_RDONLY);

        if (res == 0) {
                if(is_hidden_file(real_path)) {
                        free(real_path);
                        return -EACCES;
                }
        }
        else {
                free(real_path);
                return -errno;
        }

        free(real_path);


        res = pread(fd, buf, size, offset);
        if (res == -1) {
                res = -errno;
        }

        close(fd);
        return res;
}



/*
 * This is perhaps the most important function in terms of versioning.
 * First we make sure that this is a valid file (aka not an old file).
 * Then we must read the versioning data from the disk.
 * We add an entry to it telling it about the new versioning, copy the
 * contents of the file, and the make the modifications.
 */

static int alamofs_write(const char *path, const char *buf, size_t size,
                     off_t offset, struct fuse_file_info *fi)
{
        int fd;
        int res;

        (void) fi;
```

```
        char *real_path = get_real_path(path);

        if(is_hidden_file(real_path)) {
                free(real_path);
                return -EACCES;
        }

        struct version_data *vd = read_version_data(real_path, 0);

        free(real_path);

        if (vd == 0) {
                free(real_path);
                return -EACCES;
        }

        char *old_path;
        if(get_num_elements(vd->vdf_list) > 0) {
                old_path = strdup(vd->current_vdf->path);
        }

        add_new_version_data(vd);

        if(get_num_elements(vd->vdf_list) > 1) {
                copy_file(vd->path, vd->current_vdf->path);
        }

        fd = open(vd->current_vdf->path, O_WRONLY | O_CREAT);

        if (fd == -1) {
                printf("Could not write\n");
                return -errno;
        }

        res = pwrite(fd, buf, size, offset);

        if (res == -1) {
                printf("Could not write2\n");
                res = -errno;
        }

        close(fd);

        write_version_data(vd);
        free_version_data(vd);

        char *log = c_malloc(200);
        struct version_data_file *vdf = vd->current_vdf;
        sprintf(log, "%li write %s %s", vdf->time, vdf->path, vdf->parent);
        /* update_super_data(log); */
        free(log);

        return res;
}


static int alamofs_statfs(const char *path, struct statvfs *stbuf)
{
        int res;

        char *real_path = get_real_path(path);

        res = statvfs(real_path, stbuf);

        free(real_path);

        if (res == -1)
                return -errno;

        return 0;
}
```

```
static int alamofs_release(const char *path, struct fuse_file_info *fi)
{
        /* Just a stub.  This method is optional and can safely be left
           unimplemented */

        (void) path;
        (void) fi;
        return 0;
}

static int alamofs_fsync(const char *path, int isdatasync,
                     struct fuse_file_info *fi)
{
        /* Just a stub.  This method is optional and can safely be left
           unimplemented */

        (void) path;
        (void) isdatasync;
        (void) fi;
        return 0;
}


static struct fuse_operations alamofs_oper = {
        .getattr        = alamofs_getattr,
        .access         = alamofs_access,
        .readlink       = alamofs_readlink,
        .readdir        = alamofs_readdir,
        .mknod          = alamofs_mknod,
        .mkdir          = alamofs_mkdir,
        .symlink        = alamofs_symlink,
        .unlink         = alamofs_unlink,
        .rmdir          = alamofs_rmdir,
        .rename         = alamofs_rename,
        .link           = alamofs_link,
        .chmod          = alamofs_chmod,
        .chown          = alamofs_chown,
        .truncate       = alamofs_truncate,
        .utimens        = alamofs_utimens,
        .open           = alamofs_open,
        .read           = alamofs_read,
        .write          = alamofs_write,
        .statfs         = alamofs_statfs,
        .release        = alamofs_release,
        .fsync          = alamofs_fsync,
};

void cat_file(int argc, char *argv[]) {
        if(argc != 1) {
                printf("Bad arguments to --cat\n");
                return;
        }

        char *real_path = get_real_path(argv[0]);
        char *tmp_path;

        tmp_path = resolve_tag(real_path);
        free(real_path);
        real_path = tmp_path;


        FILE *fp = fopen(real_path, "r");
        if(!fp) {
                printf("Could not open %s\n", argv[0]);
                free(real_path);
                return;
        }

        char *buf = c_malloc(BUF_SIZE);
        while(fgets(buf, BUF_SIZE, fp)) {
                printf("%s", buf);
        }
```

```
        free(real_path);
        free(buf);
}

/*
 * The  implementation for this function is ugly and insecure. It's terrible.
 * Since I could not easily find a diff function with a suitable license,
 * this function links the files to in a temporary directory, starts a diff
 * process with the name of the temporary files as arguments. Terrible.
 */

int diff_files(int argc, char *argv[]) {
        if(argc != 2) {
                printf("Bad arguments to --diff\n");
                return -1;
        }

        char *buf = c_malloc(200);
        char *path1 = get_real_path(argv[0]);
        char *path2 = get_real_path(argv[1]);
        sprintf(buf, "diff -u %s %s", path1, path2);
        free(path1);
        free(path2);

        /*
        FILE *fp = popen(buf, "r");
        if(!fp) {
                return -1;
        }
        */
        system(buf);
        free(buf);

        return 0;
}

void print_help() {
        printf("Usage: alamofsfs mountpoint, or alamofsfs [OPTIONS]\n");
        printf("Notation: Text in brackets -- [] -- are optional. ");
        printf("Text in capitals refer to a format defined below.\n");
        printf("OPTIONS:\n");
        printf("  --search filename [SEARCHARGS], -s filename [SEARCHARGS}\n");
        printf("    Search for old versions of files matching filename. The following SEARCHARGS may be used\n");
        printf("      time TIMEARG\n");
        printf("        TIMEARG: >|<|=TIME\n");
        printf("          TIME: [YYYY[-MM[-DD[ HH[:MM[:SS]]]]]] or a unix timestamp\n");
        printf("      hastext \"text\"\n");
        printf("      nothastext \"text\"\n");
        printf("  --cat file, -c file\n");
        printf("    Prints the contents of a file\n");
        printf("  --diff file1 file2, -d file1 file2\n");
        printf("    Prints the diff between the two files\n");
}

int parse_opts(int argc, char *argv[]) {

        if(argc == 0 || strcmp(argv[0], "help") == 0 || strcmp(argv[0], "--help") == 0 || strcmp(argv[0], "-h") == 0) {
                print_help();
        }
        if(strcmp(argv[0], "--search") == 0 || strcmp(argv[0], "-s") == 0) {
                do_match(argc - 1, argv + 1);
        }
        else if (strcmp(argv[0], "--branch") == 0) {
                handle_branch(argc - 1, argv + 1);
        }
        else if(strcmp(argv[0], "--cat") == 0 || strcmp(argv[0], "-c") == 0) {
                cat_file(argc - 1, argv + 1);
        }
        else if(strcmp(argv[0], "--diff") == 0 || strcmp(argv[0], "-d") == 0) {
                diff_files(argc - 1, argv + 1);
        }
        else if(strcmp(argv[0], "--revert") == 0 || strcmp(argv[0], "-r") == 0) {
                revert_file(argc - 1, argv + 1);
```

```
        }
        else if(strcmp(argv[0], "--merge") == 0 || strcmp(argv[0], "-n") == 0) {
                handle_merge(argc - 1, argv + 1);
        }
        else if(strcmp(argv[0], "--tag") == 0 || strcmp(argv[0], "-t") == 0) {
                tag_file(argc - 1, argv + 1);
        }

        else {
                return 1;
        }

        return 0;
}

int main(int argc, char *argv[])
{
        umask(0);

        if(sd_init == 0) {
                init_super(sd);
                sd_init = 1;
        }

        /* printf("dir: %s\n", getcwd(NULL, 0)); */

        if(parse_opts(argc - 1, argv + 1) == 1) {
                sd = c_malloc(sizeof(struct super_data));

                char *data_dir = argv[2];
                char **args = c_malloc(sizeof(char *) * argc - 1);

                args[0] = argv[0];
                args[1] = argv[1];

                int i = argc;

                while(--i - 2) {
                        args[i - 1] = argv[i];
                }


                sd->data_dir = strdup(data_dir);

                /*
                if(read_super_data() == -1) {
                        printf("Could not read super data\n");
                        exit(1);
                }
                */

                return fuse_main(argc - 1, args, &alamofs_oper, NULL);

        }
        else {
                return 0;
        }
}
```

match.h

```
struct match_data {
        char *file_name;

        long int before_time;
        long int after_time;
        char *has_text;
        char *not_has_text;
};

struct match_data_file {
        struct array *time_matches; /* struct time_match */
        struct array *size_matches; /* struct size_match */
        int read_only;
        struct array *indir_matches; /* struct path_match */
};

struct time_match {
        int type;
        int policy;
        long int time;
};

#define LESS_THAN 1
#define GREATER_THAN 2

struct size_match {
        int size; /* in bytes */
        int policy;
        int type;
};

struct path_match {
        char *path;
        int policy;
};

/* Policy defs */
#define KEEP_ALL 1
#define KEEP_ONE 2
#define KEEP_NONE 3

struct match_info {
        struct file_info *head;
        struct file_info *tail;
};

struct file_info {
        long int time;
        int current;


        char *path;
        char *parent;
        struct file_info *next;
};

#define MATCH_BUF_SIZE 200
#define MATCH_FILE_NAME "/match.vdata"

int do_match(int argc, char *argv[]);
int perform_scan(struct match_data *md);

int get_has_text(struct match_data *md, struct match_info *mi);
int get_between_time(FILE *fp, struct match_data *md, struct match_info *mi);

struct file_info *get_next_file_info(FILE *fp, struct match_data *md);
struct match_data *get_match_data(int argc, char *argv[]);
void get_time_range(const char *str);
int init_match_data_from_disk();

int add_has_match(struct match_data *md, char *str);
```

```
int add_not_has_match(struct match_data *md, char *str);
int add_time_match(struct match_data *md, char *str);
int add_file_info(struct match_info *mi, struct file_info *fi);

int get_size_in_bytes(char *str);
long int parse_time(char *time, long int *end);

/* memory */
struct match_data *init_match_data();
void free_match_data(struct match_data *md);
void free_file_info(struct file_info *fi);
void free_match_info(struct match_info *mi);

/* debug */
void print_match_data(struct match_data *md);
void print_file_info(struct file_info *fi);
```

match.c

```
#define _XOPEN_SOURCE 500

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <dirent.h>
#include <errno.h>
#include <sys/time.h>
#include <time.h>
#include <unistd.h>

#include "main.h"
#include "match.h"
#include "misc.h"
#include "versioning.h"

extern int sd_init;

/*
 * This is the initial file called to perform a search.
 * It contains all the command line arguments after --search
 *
 * This function doesn't do much processing. Printing the results is left to
 * perform_scan().
 */
int do_match(int argc, char *argv[]) {

        struct match_data *md = get_match_data(argc, argv);

        if(!md) {
                return 0;
        }

        int ret = perform_scan(md);
        free_match_data(md);

        return ret;
}

/*
 * This function is called after get_match_data has been called. That function
 * is responsible for parsing the arguments and populating the data structures
 * for the scan. This function does the actual scanning.
 *
 * Although theoretically the different search criteria could be applied in
 * any order, the time criteria is applied first. This is done for efficiency
 * reasons. Since all the files in the log are ordered chronologically, there
 * are no need to do any file reads (besides the log file) to get the ones
 * that match. If, for example, hastext were done first, many more file reads
 * would need to be performed.
 *
 * Both  hastext and nothas text are done at the same time to avoid having to
 * read the file twice.
 *
 * The current order used is as follows:
 * -filename
 * -time
 * -hastext, -nothastext
 *
 *  At the end, all the resulting files are printed.
 */
int perform_scan(struct match_data *md) {

        char *real_path;
        if(md->file_name) {
                char *tmp = get_real_path(md->file_name);
                real_path = get_data_file_name(tmp);
                free(tmp);
        }
        else {
```

```
                real_path = get_real_path("/log.vdata");
        }

        FILE *fp = fopen(real_path, "r");
        free(real_path);
        if(!fp) {
                printf("Could not read versioning data\n");
                return -1;
        }
        skip_headers(fp);

        struct match_info *mi = c_malloc(sizeof(struct match_info));
        if(!mi) {
                return -1;
        }

        mi->head = 0;
        mi->tail = 0;

        get_between_time(fp, md, mi);

        if(!mi->head) {
                if(md->has_text || md->not_has_text) {
                        /* printf("No file name matches\n"); */
                }
                free_match_info(mi);
                return 0;
        }

        if(md->has_text || md->not_has_text) {
                get_has_text(md, mi);
        }

        print_file_info(mi->head);
        free_match_info(mi);

        return 0;
}

/*
 * Debug function
 */
void print_match_data(struct match_data *md) {
        printf("File: %s\n", md->file_name);
        printf("Before, After: %li, %li\n", md->before_time, md->after_time);
        printf("Has text: %s\n", md->has_text);
        printf("Not has text: %s\n", md->not_has_text);
}


struct file_info *get_next_file_info(FILE *fp, struct match_data *md) {
        struct file_info *fi = c_malloc(sizeof(struct file_info));

        char *parent = c_malloc(BUF_SIZE);
        char *path = c_malloc(BUF_SIZE);
        long int time;
        int current;

        if(md->file_name) {
                if(fscanf(fp, "%d %li %s %s\n", &current, &time, parent, path) == 4) {
                        fi->current = current;
                        fi->time = time;
                        fi->parent = strdup(parent);
                        fi->path = strdup(path);
                        fi->next = 0;

                        free(parent);
                        free(path);

                        return fi;
                }
        }
```

```
        free(parent);
        free(path);
        free(fi);

        return 0;
}

/*
 * This is called after a search is finished. The struct file_info contains all
 * the files which matched. This function prints the name and some basic information
 * about each.
 */

void print_file_info(struct file_info *fi) {
        while(fi) {
                printf("File name: %s; Time: %li", fi->path, fi->time);
                if(fi->current) {
                        printf(" (current)");
                }
                printf("\n");
                fi = fi->next;
        }
}

/*
 * This function opens every file which has matched the time criteria and searches
 * for matches which have a certain text, or do not have one. Both are done at
 * same time, if applicable. Since all criteria must be met, if the nothastext
 * is matched, the file does not match.
 *
 * Unfortunately, the linked list implementation in struct match_info creates an ugly
 * first-case scenerio, and so a temporary prev struct must be used.
 */
int get_has_text(struct match_data *md, struct match_info *mi) {
        FILE *fp;
        struct file_info *fi;
        struct file_info *prev;
        char *buf = c_malloc(1000);

        fi = mi->head;
        prev = 0;

        while(fi) {
                fp = fopen(fi->path, "r");
                if(!fp) {
                        free(buf);
                        return 0;
                }
                int has = 0;
                int not_has = 1;
                while(fgets(buf, 1000, fp)) {
                        if(md->has_text && strstr(buf, md->has_text)) {
                                has = 1;
                                if(!md->not_has_text) {
                                        break;
                                }
                        }
                        if(md->not_has_text && strstr(buf, md->not_has_text)) {
                                not_has = 0;
                                if(!md->has_text) {
                                        break;
                                }
                        }

                }
                if((md->not_has_text && !not_has) || (md->has_text && !has)) {
                        if(prev == 0) {
                                mi->head = fi->next;
                                free_file_info(fi);
                                fi = mi->head;
                        }
                        else {
```

```
                                prev->next = fi->next;
                                prev = fi;
                                free_file_info(fi);
                                fi = prev->next;
                        }
                }
                else {
                        prev = fi;
                        fi = fi->next;
                }
        }

        fclose(fp);
        free(buf);
        return 0;
}

/*
 * get_between_time populates a struct match_info with all the files that match
 * md->file_name that also fall within the time constraints.
 *
 * This function both reads the log files from the disk and checks to see if
 * they meet the time criteria, although the file is opened previously.
 * Semantically, it would be nicer to seperate this into two different
 * functions, but this implementation is a lot ``cleaner'' and more efficient.
 *
 * Returns: This function updates `mi' with a list of files that matches.
 * It returns 0 on success and -1 if an error occurs (which currently
 * cannot occur.
 */
int get_between_time(FILE *fp, struct match_data *md, struct match_info *mi) {
        int at_min = md->after_time ? 0 : 1;

        /* while(fscanf(fp, "%li %s %s %s", &time, type, path, parent) == 4) { */
        /* while(fscanf(fp, "%d %li %s %s\n", &current, &time, parent, path) == 4) { */
        struct file_info *fi;

        while((fi = get_next_file_info(fp, md))) {
                if(!at_min) {
                        if(fi->time > md->after_time) {
                                at_min = 1;
                        }
                }
                if(at_min) {
                        if(md->before_time && fi->time > md->before_time) {
                                free_file_info(fi);
                                break;
                        }

                        if(strstr(fi->path, md->file_name)) {
                                add_file_info(mi, fi);
                        }
                }
        }
        return 0;
}

/*
 * This function is called whenever by get_between_time() to add files to mi.
 * This is just a standard linked list add function.
 *
 * Returns: Modifies mi to add `fi' to the end. Returns 0 on success.
 */
int add_file_info(struct match_info *mi, struct file_info *fi) {
        if(mi->head == 0) {
                mi->head = fi;
                mi->tail = fi;
        }
        else {
                mi->tail->next = fi;
                mi->tail = fi;
        }
```

```
        return 0;
}

/*
 * This function populates a struct match_data from the arguments given on the
 * command line. It does not actually search for any files. This is left to
 * perform_scan(), which uses 'md' to know what to look for.
 *
 * Returns: 'md', with all the fields minus 'file_name' filled in. If the user
 * did not specifiy anything for a certain criteria, it is set to 0.
 *
 * Technically, this does not allow the user to use a time value of 0, but it
 * is rare that a user would want to do this.
 */
struct match_data *get_match_data(int argc, char *argv[]) {
        int res;

        struct match_data *md = init_match_data();
        if(!md) {
                return 0;
        }

        if(argc >= 1) {
                if(strcmp(argv[0], "--time") != 0 && strstr(argv[0], "hastext") == 0) {
                        md->file_name = strdup(argv[0]);
                        argc--;
                        argv++;
                }
                else {
                        md->file_name = 0;
                }
        }
        else {
                return 0;
        }

        /* All arguments come in pairs, e.g., time >2008-04-02 */
        while(argc - 1 > 0) {
                if(strcmp(argv[0], "--time") == 0) {
                        char *tmp = argv[1];
                        if(argc >= 3) { /* Time may be in two strings due to space character */
                                if(strstr(argv[2], "hastext") == 0) {
                                        tmp = strdup(argv[1]);
                                        strcat(tmp, " ");
                                        strcat(tmp, argv[2]);
                                        argc--;
                                        argv++;
                                }
                        }
                        res = add_time_match(md, tmp);
                        if(res < 0) {
                                printf("Could not parse time: %s\n", argv[1]);
                                return 0;
                        }
                }
                else if(strcmp(argv[0], "--hastext") == 0) {
                        md->has_text = strdup(argv[1]);
                }
                else if(strcmp(argv[0], "--nothastext") == 0) {
                        md->not_has_text = strdup(argv[1]);
                }
                else {
                        return 0;
                }

                argc -= 2;
                argv += 2;
        }

        if(argc) {
                return 0;
        }
```

```
        return md;
}

/*
 * Please see the comments for parse_time for information on the format of the
 * 'str' variable.
 *
 * This function checks if the first character is a '<', '>', or '=', calls
 * parse_time() to parse the actual time, and fills in the appropriate 'md'
 * members.
 *
 * Returns: The function modifies 'md' with the information in the string.
 * The function returns 0 on success. On error, it prints a message and
 * returns -1. 'md' is untouched in this case, but it shouldn't really matter
 * since the entire thing needs to be aborted.
 */
int add_time_match(struct match_data *md, char *str) {
        long int end = 0;

        if(*str == '>') {
                if(md->after_time) {
                        return -1;
                }
                md->after_time = parse_time(str + 1, 0);
                if(!md->after_time) {
                        return -1;
                }
        }
        else if(*str == '<') {
                if(md->before_time) {
                        return -1;
                }
                md->before_time = parse_time(str + 1, 0);
                if(!md->before_time) {
                        return -1;
                }
        }
        else if(*str == '=') {
                if(md->after_time || md->before_time) {
                        return -1;
                }

                md->after_time = parse_time(str + 1, &end);
                md->before_time = end;
                if(!md->after_time || !md->before_time) {
                        return -1;
                }
        }
        else {
                return -1;
        }

        return 0;
}

/*
 * 'str' contrains a time value, which can be in various formats.
 *
 * add_time_match first checks if the starting character is a '=', '<', or '>'.
 * If it one of the second two, the 'end' parameter is left untouched, and the
 * return value is the unix timestamp associated with the date. If it is '=',
 * the returned value is the start data, and 'end' is modified to contain the
 * end data. 'end' must be a null pointer if a '<' or '>' was present.
 *
 * It would be slightly cleaner to use two different functions depending on
 * if '=' was used or not, but this keeps all the code together.
 *
 * There are two formats that are acceptable. The first is a unix timestamp.
 * The second is a human readable data, in the form of
 * [YY[YY][-MM[-DD[ HH[:MM[:SS]]]]]].
 *
 * If the '=' is used, it uses a range dating from the date given, the
 * data + smallest_time_unit_used. For example, =2008 would match starting
```

```
 * from the first second of 2008 to a year later. =2008-03 would match dates
 * in March of 2008, etc.
 *
 * It would probably be more helpful to provide a more lenient time parsing
 * function. That way, the user would not have to provide the data if he is
 * just looking for files modified in the past day, or use a format such as
 * MM/DD/YY (although that runs into ambiguity issues, since DD/MM/YY is also
 * used).
 *
 * There is a slight ambiguity issue, in that a string such as >2008 could
 * mean a date after the year 2008, or after the 2008th second since the
 * epoch. In such cases, it is assumed that the year is meant. It is unlikely
 * that a user would need to match files created almost 40 years before this
 * file system was developed.
 *
 * Returns: If the string was parsed succesfully, the timestamp is returned.
 * In the case of a
 */
long int parse_time(char *str, long int *end) {

        char *ptr = str;
        char *buf = c_malloc(TIME_BUF_SIZE);
        int pos = 0;
        int time;
        struct tm *t = c_malloc(sizeof(struct tm));

        t->tm_year = 0;
        t->tm_mon = 0;
        t->tm_mday = 0;
        t->tm_hour = 0;
        t->tm_min = 0;
        t->tm_sec = 0;

        do {
                while(*ptr >= '0' && *ptr <= '9') {
                        ptr++;
                }

                /* Good ol' timestamp */
                if(!*ptr && ptr - str != 2 && ptr - str != 4) {
                        free(buf);
                        return atoi(str);
                }

                strncpy(buf, str, ptr - str);
                buf[ptr - str] = 0;
                time = atoi(buf);

                /*
                 * The struct tm format is a bit strange, which is why we need
                 * to subtract numbers from a few of these.
                 */
                switch(pos) {
                        case 0:
                                if(time < 1900 || time > 3000) {
                                        goto parse_exit;
                                }
                                t->tm_year = time - 1900;
                                break;
                        case 1:
                                if(time < 1 || time > 12) {
                                        goto parse_exit;
                                }
                                t->tm_mon = time - 1;
                                break;
                        case 2:
                                if(time < 0 || time > 31) {
                                        goto parse_exit;
                                }
                                t->tm_mday = time;
                                break;
                        case 3:
                                if(time < 0 || time > 23) {
```

```
                                        goto parse_exit;
                                }
                                t->tm_hour = time;
                                break;
                        case 4:
                                if(time < 0 || time > 59) {
                                        goto parse_exit;
                                }
                                t->tm_min = time;
                                break;
                        case 5:
                                if(time < 0 || time > 59) {
                                        goto parse_exit;
                                }
                                t->tm_sec = time;
                                break;
                        default:
                                goto parse_exit;
                }

                pos++;
                str = ++ptr;

        } while (ptr[-1]);

        free(buf);
        long int start = mktime(t);

        if(pos < 6) {
                if(!end) {
                        return start;
                }
                switch(pos) {
                        case 5:
                                *end = start + 60;
                                break;
                        case 4:
                                *end = start + 60 * 60;
                                break;
                        case 3:
                                *end = start + 24 * 60 * 60;
                                break;
                        case 2:
                                *end = start + 31 * 24 * 60 * 60;
                                break;
                        case 1:
                                *end = start + 365 * 24 * 60 * 60;
                                break;
                        default:
                                return 0;
                }
        }

        return start;

parse_exit:
        free(buf);
        return 0;
}

/*
 * Just allocates and zeros out everything in the struct.
 */
struct match_data *init_match_data() {
        struct match_data *md = c_malloc(sizeof(struct match_data));
        if(!md) {
                return 0;
        }

        md->file_name = 0;
        md->after_time = 0;
        md->before_time = 0;
        md->has_text = 0;
```

```
        md->not_has_text = 0;

        return md;
}

/* Frees all the strings in the struct */
void free_match_data(struct match_data *md) {
        if(md->has_text) {
                free(md->has_text);
        }
        if(md->not_has_text) {
                free(md->not_has_text);
        }

        free(md);
}

/*
 * Frees the struct and all the file_info structs in it.
 */
void free_match_info(struct match_info *mi) {
        struct file_info *tmp;
        tmp = mi->head;

        while(tmp) {
                tmp = tmp->next;
                free_file_info(mi->head);
                mi->head = tmp;
        }

        free(mi);
}

/*
 * Frees the memory associated with a struct file_info
 */
void free_file_info(struct file_info *fi) {
        if(fi->path) {
                free(fi->path);
        }
        if(fi->parent) {
                free(fi->parent);
        }
        free(fi);
}


/*
 * This function is called to determine which files need versioning
 */
int init_match_data_from_disk() {
        return 0;
}


/*
 * This function converts a human readable size to a number of bytes.
 *
 * Format: 234G, for example
 * Available sizes: k, K, m, M, g, G
 *
 * The function assumes 1k = 1024 bytes, not 1000.
 */

int get_size_in_bytes(char *str) {
        char *ptr = str;

        while(*ptr >= '0' && *ptr <= '9') {
                ptr++;
        }

        if(ptr[1] != 0) {
                return -1;
```

```
        }

        int size = ptr - str;
        char *num = c_malloc(size + 1);

        strncpy(num, ptr, size);
        int n = atoi(num);

        int mod;

        switch(*ptr) {
                case 'k':
                case 'K':
                        mod = 1024;
                        break;
                case 'm':
                case 'M':
                        mod = 1024 * 1024;
                        break;
                case 'g':
                case 'G':
                        mod = 1024 * 1024 * 1024;
                        break;
                default:
                        return -1;
        }

        return n * mod;
}

struct match_data_file *init_match_data_file() {
        struct match_data_file *mdf = c_malloc(sizeof(struct match_data_file));

        mdf->time_matches = init_array(free);
        mdf->size_matches = init_array(free);
        mdf->indir_matches = init_array(free);
        mdf->read_only = 0;

        return mdf;
}

/*
struct match_data_file *read_match_file() {
        FILE *fp = fopen(MATCH_FILE_NAME, "r");

        struct match_data_file *mdf = init_match_data_file();

        char *buf = c_malloc(500);
        while(fgets(buf, 499, fp)) {
                char *type = strsep(&buf, ' ');

                if(strcmp(type, "size") == 0) {
                        char *size = strsep(&buf, ' ');
                        char *policy = buf;
                        struct size_match *sm = c_malloc(sizeof(struct size_match));
                        if(size[0] == '>') {
                                sm->type = GREATER_THAN;
                        }
                        else if(size[0] == '<') {
                                sm->type = LESS_THAN;
                        }
                        else {
                                printf("Invalid size\n");
                                return 0;
                        }
                        sm->size = get_size_in_bytes(size + 1); // get rid of > or <
                        sm->policy = parse_policy(policy);
                        add_element(mdf->size_matches, sm);
                }
                else if(strcmp(type, "read-only") == 0) {
                        char *policy = buf;
                        if(mdf->read_only) {
                                printf("More than one read-only policy specified");
```

```
                                return 0;
                        }
                        mdf->read_only = parse_policy(policy);
                }
                else if(strcmp(type, "indir") == 0) {
                        char *dir = strsep(&buf, ' ');
                        char *policy = buf;

                        struct path_match *pm = c_malloc(sizeof(struct path_match));
                        pm->path = strdup(dir);
                        pm->policy = parse_policy(policy);
                        add_element(mdf->path_matches, pm);
                }
                else if(strcmp(type, "time") == 0) {
                        char *time = strsep(&buf, ' ');
                        char *policy = buf;

                        struct time_match *tm = c_malloc(sizeof(struct time_match));
                        tm->policy = parse_policy(policy);

                        if(time[0] == '>') {
                                tm->type = GREATER_THAN;
                                tm->time = parse_time(time + 1, 0);
                        }
                        else if(time[0] == '<') {
                                tm->type = LESS_THAN;
                                tm->time = parse_time(time + 1, 0);
                        }
                        else if(time[0] == '=') {
                                struct time_match *tm2 = c_malloc(sizeof(struct time_match));
                                tm->type = GREATER_THAN;
                                tm2->type = LESS_THAN;
                                tm->time = parse_time(time + 1, &tm2->time);
                                tm->policy = parse_policy(policy);
                                tm2->policy = tm->policy;
                                add_element(mdf->time_matches, tm2);
                        }
                        else {
                                printf("Invalid time specified: %s", time);
                                return 0;
                        }

                        add_element(mdf->time_matches, tm);

                }
        }

        sd->mdf = mdf;
}
*/

int parse_policy(char *policy) {
        if(strcmp(policy, "all") == 0 || strcmp(policy, "keep_all") == 0) {
                return KEEP_ALL;
        }
        else if(strcmp(policy, "none") == 0 || strcmp(policy, "keep_none") == 0) {
                return KEEP_NONE;
        }
        else if(strcmp(policy, "one") == 0 || strcmp(policy, "keep_one") == 0) {
                return KEEP_ONE;
        }
        else {
                return -1;
        }
}

#define min(X, Y)  ((X) < (Y) ? (X) : (Y))
#define max(X, Y)  ((X) > (Y) ? (X) : (Y))
int find_policy_of_file(struct version_data_file *vdf) {
        struct match_data_file *mdf = sd->mdf;
        int policy = 100; /* Random big number */

/*
```

```
        if(mdf == 0) {
                mdf = read_match_file();
                if(!mdf) {
                        return -1;
                }
        }
*/

        struct stat *stbuf = c_malloc(sizeof(struct stat));
        lstat(vdf->path, stbuf);
        int file_size = stbuf->st_size;

        int size = get_num_elements(mdf->time_matches);

        while(--size + 1) {
                struct time_match *tm = get_nth_element(mdf->time_matches, size);
                if(tm->type == GREATER_THAN && vdf->time > tm->time) {
                        policy = min(policy, tm->policy);
                }
                else if(tm->type == LESS_THAN && vdf->time < tm->time) {
                        policy = min(policy, tm->policy);
                }
        }

        size = get_num_elements(mdf->size_matches);

        while(--size + 1) {
                struct size_match *sm = get_nth_element(mdf->size_matches, size);
                if(sm->type == GREATER_THAN && file_size > sm->size) {
                        policy = min(policy, sm->policy);
                }
                else if(sm->type == LESS_THAN && file_size < sm->size) {
                        policy = min(policy, sm->policy);
                }
        }

        size = get_num_elements(mdf->indir_matches);

        while(--size + 1) {
                /* lol no */
        }

        return policy;
}
```

misc.h

```
struct array {
        void **ptr;
        int size;
        int pos;
        void (*free_func)(void *);
};

#define BUF_SIZE 1024
#define TIME_BUF_SIZE 15

char *get_real_path(const char *path);
char *get_fake_path(const char *real_path);
int is_subdir(const char *big, const char *little);
int is_dir(char *real_path);

int get_last_pos(char *path, char delim);
long int get_current_time();

/* arrays */
void * get_nth_element(struct array *ar, int n);
struct array *init_array(void (*free_func)(void *));
int get_num_elements(struct array *ar);
void add_element(struct array *ar, void *element);
void free_array(struct array *ar);

void *c_malloc(size_t size);
void *c_realloc(void *ptr, size_t size);
```

misc.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/time.h>

#include "misc.h"

/*
 * FUSE gives us the name of each file as an absolute address using the
 * mount directory as the root. Since we do not actually store the files
 * in there, we need to do some translation.
 *
 * This function just concatenates ‘path’ to the appropriate base string.
 * The caller is responsible for free’ing the string.
 */


char *get_real_path(const char *path) {
        char *real_path = c_malloc(strlen(path) + strlen(BASE_PATH) + 1);
        *real_path = 0;

        strcat(real_path, BASE_PATH);
        if(*path == ’/’) { /* No need for two foward slashes */
                real_path[strlen(BASE_PATH) - 1] = 0;
        }
        strcat(real_path, path);

        return real_path;
}

char *get_fake_path(const char *real_path) {
        return strdup(real_path + strlen(BASE_PATH) - 1);
}

/*
 * This function is used to get the file name (not including path)
 * or extension of a file. It returns the position of the last
 * occurance of ‘delim‘ or -1 if it is not found;
 */

int get_last_pos(char *path, char delim) {
        int pos = strlen(path);

        while(path[--pos] != delim && pos + 1);
        return pos;
}

struct array *init_array(void (*free_func)(void *)) {
        struct array *ar = c_malloc(sizeof(struct array));
        ar->size = 8; /* Random number */
        ar->ptr = c_malloc(sizeof(void *) * ar->size);
        ar->pos = 0;
        ar->free_func = free_func;

        return ar;
}

void * get_nth_element(struct array *ar, int n) {
        if(n > ar->pos) {
                return 0;
        }

        return  ar->ptr[n];
}

int get_num_elements(struct array *ar) {
        return ar->pos;
}

void add_element(struct array *ar, void *element) {
```

```
        if(ar->pos == ar->size) {
                ar->size *= 2;
                ar->ptr = c_realloc(ar->ptr, sizeof(void *) * ar->size);
        }

        ar->ptr[ar->pos] = element;
        ar->pos++;
}

void free_array(struct array *ar) {
        int i;

        for(i = 0; i < ar->pos; i++) {
                (*(ar->free_func))(ar->ptr[i]);
        }

        free(ar);
}

void *c_malloc(size_t size) {
        void *ptr = malloc(size);
        if(!ptr) {
                printf("Out of memory\n");
                abort();
        }

        return ptr;
}

void *c_realloc(void *ptr, size_t size) {
        void *p = realloc(ptr, size);
        if(!ptr) {
                printf("Out of memory\n");
                abort();
        }

        return p;
}

int is_dir(char *real_path) {
        struct stat *statbuf = c_malloc(sizeof(struct stat));
        lstat(real_path, statbuf);
        int isdir = S_ISDIR(statbuf->st_mode);
        free(statbuf);

        return isdir;
}

/*
 * A wrapper that returns the current unix timestamp
 */

long int get_current_time() {
        struct timeval tv;
        gettimeofday(&tv, 0);
        return (long int) tv.tv_sec;
}
```