The Plaz Provenance File System (PPFS)

 Michael Plasmeier

*theplaz@mit.edu*

 Rudolph 10AM

March 22, 2012

# Introduction

A provenance file system is a file system which stores the history and source of files edited on a local computer system. For example, when one is editing a slide deck, one might want to know from which slide deck a slide was copied from. This paper builds upon the basic file system in the early versions of Unix and introduces a provenance file system known as the *Plaz Provenance File System (PPFS)*. In particular, the PPFS introduces another layer, called the log layer, which maintains pointers to past versions and ancestors of the file.

The PPFS stores a full, verbose set of provenance information. The PPFS also stores the complete data of old versions of files, giving it many of the features of a versioning file system. The PPFS also supports seeing which files are based off a specific file. This is called reverse lookup. These characteristics make the PPFS well suited for businesses that face regulatory or legal requirements to log all changes to files. The PPFS is also well suited to businesses which frequently create derivative works from previous works. For example, a consulting company may what to know which project a slide in a slide deck was copied from.

The system aims for a simple design and it attempts to use a minimal amount of disk space for maintaining provenance information and a de minimis amount of Random Access Memory (RAM) space for the tracking of provenance information. However, it currently uses up a lot of disk space to store old versions of files. Additional disk space could be saved by de-duplication algorithms. Data is laid out so that lookups from both directions (the ancestors of a file and the children of a file) can be performed relatively quickly. As part of the operating system, the PPFS extends the usual (read(), write()) operations. For more complicated or novel functionality, new API calls are introduced.

At the moment, the PPFS operates only on one computer. It is not optimized to work over a network, nor does not track provenance information from files copied from other computers, such as web servers.

# The Log Layer

The PPFS introduces a new layer into the Unix file system called the log layer. This layer is inserted between the file name layer and the inode layer, as shown in Figure 1. The file name layer is modified by redefining the inode number in the directory table to the log entry number.

File Name Layer

Log Layer

Inode

Inode

Inode

Log entry #

Inode #

Figure 1 The Log Layer is inserted between the file name layer and the inode layer

The log layer contains a table of information about the history of each file. Each file has its own table, which is stored on disk in the same way as the inode layer. The log layer table is stored at the beginning of the disk, in a fixed position on the disk, after the inode table. The table consists of a list of log entries, each pointing to the inode number of a *version* of the file, as shown in Figure 2. A version is created automatically each time the file is saved. The last entry in the log layer entry contains a reference to the log layer entry that the file was created from (the *ancestor)*. A bit in each entry designates a row as a version/inode pointer or an ancestor /log layer pointer. If a file was created from scratch (ie using touch) then the last entry in the log layer table will be 0. Although the inode stores the time the file was modified, that information would not be changed if the file was copied, so PPFS also stores that information in the log table. Additional log information, such as the current user’s username and the application that made the change could is also stored here.

The number of incoming links that was stored in the inode in the original Unix file system is redefined to count the number of log layer entries pointing to the inode. The log layer table includes a count of the number of incoming links that was traditionally found in the inode. These counts are manifested in the reverse tables, described below.

Directories are ignored by the PPFS and function as usual. This may differ from certain versioning file systems.

**Log entry 987654
98765**

**Ancestor**: 000000

**Inode**: 123456

**Directory /**

|  |  |
| --- | --- |
| File name | Log Entry # |
| File A | 987654 |
|  |  |
|  |  |
|  |  |
|  |  |

**Inode 123456**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |

**Inode**: 123457

**Inode 123457**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |

Figure 2 File A is created with content and is then edited.

**Log entry 987654**

**(File A)**

**Ancestor**: 000000

**Log entry 987665**

**(File B)**

**Inode**: 123457

**Inode**: 123456

**Ancestor**: 987654

**Inode**: 123458

**Inode**: 123457

Same

Figure 3 File A, from above, is then copied to be File B. File B is then edited. Notice that File B rev 0 shares an inode or version with file A rev 1.

When provenance information is queried (via read\_prov()) for File B, the log entry for File B will be retrieved. Provenance information will then be recursively queried (to A in this example) until an ancestor of 0 is reached.

## Reverse Lookup

One requirement of a provenance file system is to know all of the files which originated from a particular file. This information can be accessed using the search\_prov() system call. In order to support the reverse search case, the log entry and inode tables are modified. A reverse inode table is added for each inode, which contains the list of log entry tables which point to that inode. A reverse log entry table is added to each leg entry o retrieve the files names which each log entry represents.

**Reverse Inode 123457**

**(File A rev 1 and File B rev 0)**

**Log Entry**: 987665

**Log Entry**: 987654

**File**: /A

**Reverse Log 987654**

**(File A)**

**File**: /B

**Reverse Log 987665**

**(File A)**

**0**

**0**

Deleted?

Deleted?

Figure 4 The Reverse Lookup Inode table for Inode 123457 shows that the File A and File B shared the same data at some point in time (i.e. one must be the ancestor of another)

In a search\_prov() query, the system first looks up the log layer table for a particular file. For every inode mentioned in the log layer table, the system retrieves the reverse inode table. The system then retrieves the reverse log tables for each file. The system then outputs the list of files referenced. If needed, the system could also lookup the log tables themselves to find the revision number and timestamp for each file.

## Parts of a File

Provenance information can be stored about the *parts* of a file (for example, the slides in a slide deck). This information is stored by having multiple ancestors in the log layer, as shown in Figure 5. In this example, the difference between Slide Deck E version 2 and version 1 came from Slide Deck D. The pointer to Slide Deck D and a name for this piece would be set by the write\_prov() call. The data that is different would be inferred by looking at the difference between the inode before and after the ancestor entry. The name data is stored in a separate table, as seen in Figure 6.

Applications wishing to take advantage of the *provenance by parts* functionality would need to implement this API call.

**Log entry 987636**

**(Slide Deck E)**

**Ancestor**: 987352

**Inode**: 126268

**Inode**: 126267

**Ancestor**: 987640

**Inode:** 126269

Slide Deck C

Version 0

Version 1

Version 2

“Slide 7”

from Slide Deck D

🡨 Copy in a Slide from D.ppt (4 slides)

🡨 Edit E.ppt to add a Slide (3 slides)

🡨 cp C.ppt E.ppt (2 slides)

Figure 5 Slide Deck E was copied from Slide Deck C with 2 slides; a slide was added from scratch; and then a fourth slide was copied in from Slide Deck D (which was previously called slide 7 in Slide Deck D)

**Piece names 987636**

**(Slide Deck E)**

|  |  |
| --- | --- |
| Log entry # | Name |
| 4 | “Slide 7” |
|  |  |
|  |  |
|  |  |

Figure 6 Piece names for the various pieces for Slide Deck E. In this example, the 4th entry (from bottom) of the log entry for Slide Deck E were previously called “Slide 7” by the application.

## Compilations of Files

Information about the source in compiled binary files can be stored in in a similar way. Multiple ancestor entries are stored at the bottom of the table between the first ancestor and the inode of the newly compiled file, as shown in Figure 7. The first entry will be 0, since the file was created new. Normal log entries will accumulate on top of this information, as before.

**Log entry 875855**

**(Binary H)**

**Ancestor**: 000000

**Ancestor**: 875884

**Ancestor**: 875883

**Inode:** 126269

File Created

Source F

Source G

Compiled

Figure 7 Binary H is compiled from Source F and Source G

## File Archives

File archives present a particular challenge. File archives read information off the disk and then store it in their own proprietary format. In order to be truly portable, this requires all of the provenance information, along with all of the past versions and ancestors of a file to be stored in the file archive. This information would be retrieved using a special call, such as read\_full\_provenance(), which would store the provenance information and past versions in a flat format. This format is a XML format which mirrors the tables in the file system, as shown in Figure 8. The file would then be compressed using normal ZIP or TAR algorithms.

<xml schema=”ppfs-portable”>

<log-entries>

<log-entry id=”875855”>

 <ancestor>000000</ancestor>

<ancestor>875883</ancestor>

 <ancestor>875884</ancestor>

<inode>126269</inode>

<reverse>

 <file>/H</file>

</reverse>

*//Additional metadata (i.e. name, date) removed*

</log-entry>

<log-entry id=”875883”>

 <ancestor>000000</ancestor>

<inode>126267</inode>

<reverse>

 <file>/F</file>

</reverse>

</log-entry>

<log-entry id=”875884”>

 <ancestor>000000</ancestor>

<inode>126268</inode>

<reverse>

 <file>/G</file>

</reverse>

</log-entry>

 </log-entries>

 <inodes>

 <inode id=”126269”>

 <data>*(Binary data)*</data>

 <reverse>

 <log-entry>875855</log-entry>

 </reverse>

 <inode>

<inode id=”126268”>

 <data>*(Binary data)*</data>

 <reverse>

 <log-entry>875883</log-entry>

 </reverse>

 <inode>

<inode id=”126267”>

 <data>*(Binary data)*</data>

 <reverse>

 <log-entry>875884</log-entry>

 </reverse>

 <inode>

 </inodes>

</xml>

Figure 8 The flat file XML for the scenario in Figure 6

When the file archive is extracted, the provenance information is recreated using a special write\_full\_provenance() call. The inode and table entry numbers will change, but the same structure will be created. Those that are interested in preserving the authenticity of the provenance information should disable this feature, because it allows anyone to write provenance information (including old time stamps) to disk.

# Deletion and Thinning

When unlink(filename) is called, the filename to log entry link is removed, and the deleted bit in the reverse log table is flipped (decrementing the traditional link count in the log layer entry). When the count of incoming links in the log entry table reaches 0, the file is no longer accessible. However, the log table and versions are kept in order to preserve provenance information.

In order to save space some intermediate versions can be removed according to a thinning schedule. This schedule is user-settable, but the default values are shown in Table 1. The thinning process is accomplished by a “garbage collection”-style program. A revision will be kept if more than one log entry is present in the reverse inode table – i.e. when a file was copied and is now provenance information for a different file. When versions are thinned, the actual inode/data is removed from the disk, and all references to that version are removed from the log layer.

|  |  |
| --- | --- |
| Days after revision created | Target number of revisions kept |
| < 7 days | 1 / minute |
| > 7 days and < 30 days | 1 / hour |
| > 30 days and < 1 year | 1 / day |
| > 1 year | 1 / week |

Table 1 Intermediate revisions can be thinned after a certain amount of time after their creation. These are the default values

# Performance

PPFS should be not appreciably slower when adding many files to the disk. Principally, the disk must make one additional write (the log layer table) in addition to its other writes. Generally, the non-sequential disk accesses slow a hard drive down. PPFS adds one additional non-sequential access. Thus the system should be no more than 33% slower (adding the log layer to the file system pointer, inode, and file data). Thus the system should be able to easily handle writing 10 files to disk per second. PPFS scales with the size of the disk and is linear with regard to the number of items added to disk per second. The garbage collection process is optional, and can be postponed until the system is relatively idle.

In addition, the system can quickly search for the children of a file (files that are based on that file) by using the reverse inode table. Such lookups should not depend on the number of files on the disk. PPFS scales well with regard to the number of files on disk.

For a file with many ancestors, PPFS handles reads and writes to a file the same as a file without an ancestor. Retrieving the full list of provenance information scales with the number of ancestors. It is envisioned that this will not be a large bottleneck, since the number of ancestors is envisioned to be relatively low and pulling a full list of provenance information is an infrequent operation. Caching could be added to the system to improve this time, but the additional step to update or invalidate the cache would slow the copying of files.

One of the most significant performance impacts is the time to update a file. PPFS rewrites the entire file each time it is saved, in order to maintain a version history of the file. PPFS is not optimized for large files, such as media files, and is likely unsuitable for those use cases.

PPFS uses a significant amount of disk space. PPFS is designed to provide verbosity and maintain provenance information at the expense of disk space. Thus PPFS is best suited to organizations that require comprehensive and persistent logging.

# Conclusion

PPFS is a provenance file system designed to provide a comprehensive and reliable log of the history of each file. PPFS modifies the basic Unix file system to add a log layer that preserves the provenance and version history of each file on the disk. PPFS should add minimal overheard, beyond the keeping of multiple versions. PPFS should be able to easily handle lookups from both directions (the ancestors of a file and the children of a file) in a short amount of time. PPFS should scale well to the size of the disk, the number of files on disk, the number of ancestors of a file. PPFS can do additional work to reduce the disk space that old versions take up.

## Implementation Issues

Modern file systems have advanced beyond the basic Unix file system that PPFS is based on. Care should be taken to maintain the current features of file systems while implementing PPFS.

Where additional API calls have been added, developers must be recruited to update their applications to support the new APIs.

# Future Work

PPFS suffers from a number of limitations, which could be addressed by modifications to the system.

PPFS currently rewrites each file when it is edited, using a lot of disk space. A de-duplication algorithm which, for example, only stored the changes to a file, could save a significant amount of disk space.

PPFS currently only works on a local computer. PPFS could be expanded to work across a network. Provenance information is particularly helpful when there are multiple people working on a group of documents.

PPFS is currently designed to operate on a single disk. Because of the large amount of disk space used by PPFS, it could be expanded to work across multiple disks. For example, the RAID system allows multiple disks to be seen as one disk by a computer. This would allow users to add storage to the system as needed.

# Acknowledgements

## Reviewers

* Dave Custer
* Travis Grusecki

**References**

[1] J. Saltzer, M. F. Kaashoek, Principles of Computer System Design: An Introduction. Burlington, MA: Morgan Kaufmann, 2009.

# Word Count

2,358 words, including captions