Provenance Tracking System (PTS)

Ashwini Gokhale
<agokhale@mit.edu>
6.033, Prof. Rudolph, TR 10 AM
March 22, 2012

1. Overview

Provenance is critical, especially when storing important data. Tracking provenance is especially applicable in sensitive systems like hospital databases in the event of a quality control audit. This report outlines an approach to track provenance for the second extended (ext2) filesystem created by Linux. Ext2 is advantageous to work with because it has extra space in many of its on-disk data structures which allows for extensibility. I assume a versioning file system in which every file has a unique inode number and all subsequent modified versions of that file also have unique inode numbers to handle file modification.

The Provenance Tracking System (PTS) is implemented by modifying the data structure "inode" and creating a new data structure called "pnode". One of the major design tradeoffs is the decision to place 2 Gigabytes of memory on-disk to accommodate new and modified PTS data structures. This on-disk overhead is necessary to successfully track provenance for sensitive systems like hospital databases. A design strength is the implementation of garbage collection; although retaining information is important, removing stale information to make room for newer, more relevant information is crucial. In this paper I describe the PTS design, demonstrate its implementation with specific use cases, and analyze performance.

2. Design Description

The goal of PTS is to build upon the existing ext2 filesystem in a minimalistic manner. Existing provenance-unaware programs continue operating correctly because PTS maintains transparent file-to-file provenance with null pnodes. For provenance-aware programs PTS provides calls to implement part-to-part provenance tracking. Provenance information is stored persistently in PTS because everything (except a temporary table) is stored on-disk.

2.1 Physical Layout of PTS

The physical (on-disk) layout of PTS is shown in Figure 1. The "pnode" (which stands for "part node"), is stored in the same way as inodes. The pnode table represents a segment of contiguous blocks allocated to store pnodes indexed by (unique) pnode numbers. PTS also stores a table mapping inode numbers to filenames; this table would be used in provenance calls to return filenames to the application.

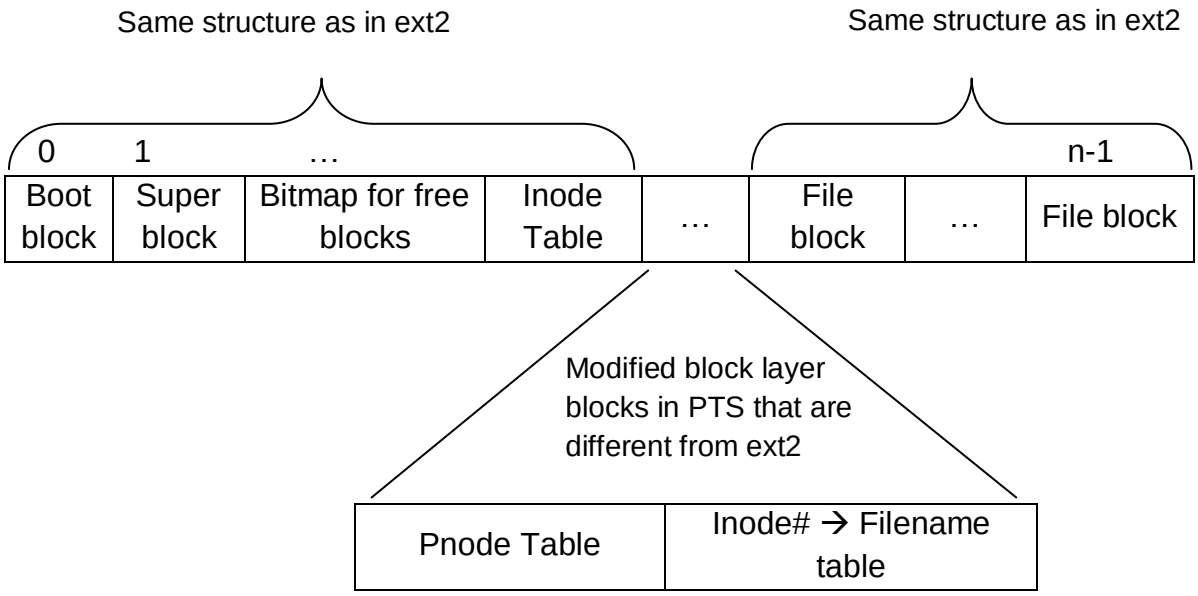

Figure 1: PTS disk layout. The first few blocks (corresponding to the boot block, super block, bitmap, and inode table) and the file blocks in PTS are the same as those in ext2. Blocks after the inode table and before the file blocks have been modified to include the pnode table and inode number to filename table.

2.2 Data Structure Creation and Modification

When a new file (or version) is created, its inode and other components (such as file size and type) are initialized. The unique inode number is a name for the inode that holds the metadata about a particular file.

To implement provenance tracking, PTS relies upon the extended attributes (xattrs) of the filesystem and "pnode". All inodes of type *file* will have a list (called *listOfParts*) containing the pnode numbers of their parts. Every inode is initialized with a "null" pnode to keep track of file-to-file provenance in provenance-unaware applications; further details are provided in Figure 2.

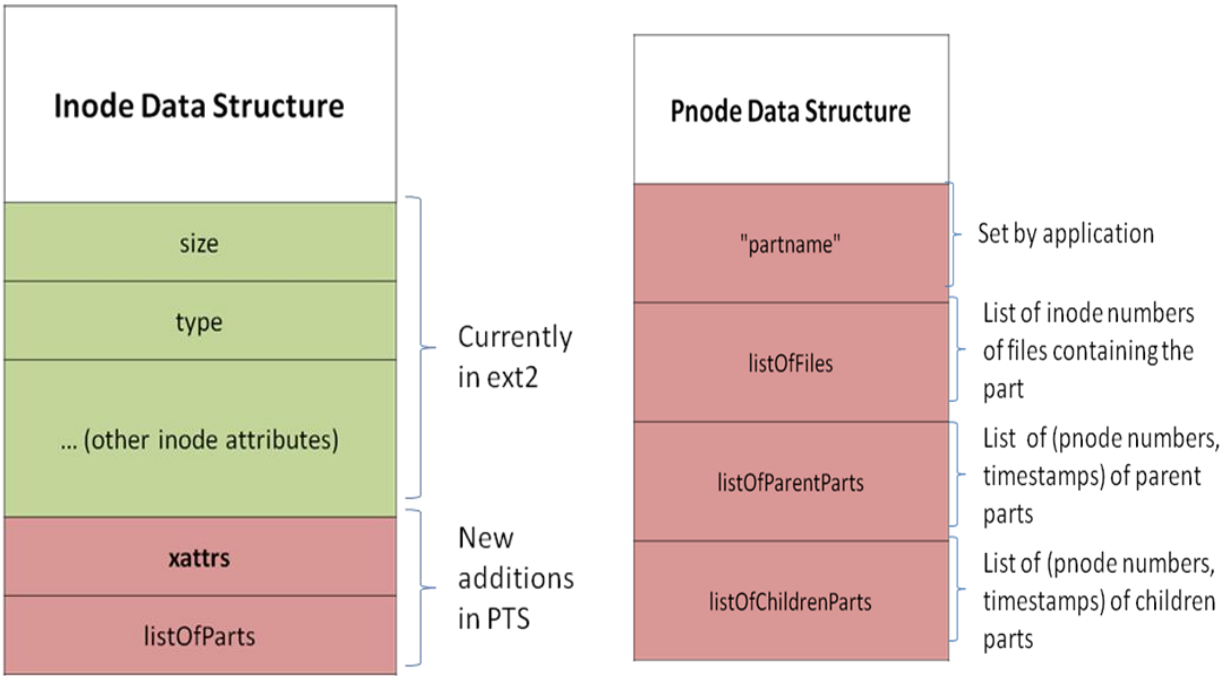

Figure 2: Data structures used in PTS. The data structure "inode" is modified using its extended attributes, and a new data structure "pnode" is introduced.

2.3 Setting and Returning Provenance

PTS allows provenance-aware applications to make read- and write-calls in order to determine and store provenance information of specific parts of files. For provenance-unaware applications, PTS will update the provenance of the entire file using pointers within the null pnodes and thereby propagate provenance by reference.

PTS stores the provenance information of file parts within a directed acyclic graph; pnodes store (pnode number, timestamp) values in their listOfParentParts and listOfChildrenParts. Timestamps stored along with the pnode numbers prevent cycles from occurring, and specify the reference-time for a given part's provenance. For example, assume that a user copies a PowerPoint slide from files B to C, and then replaces that slide in B with a different slide from file A. When an application queries the provenance of C, PTS returns only B as being a parent of C because PTS will only return provenance data that has a timestamp which is *less* than the specified timestamp.

When provenance-aware applications query a part's provenance information, it is appropriate to return a filename rather than an inode number. In Unix, given a filename, an inode number can easily be found but not vice-versa. Thus, PTS contains a table on-disk with (key, value) pairs mapping to (inode number, absolute pathname(s)). This

table is called “INFT” for **I**node **N**umber to **F**ilename **T**able. (The time required to update INFT is explained in Section 3.4.) If there are two or more hard-links to a file, then a given inode number can have more than one absolute pathname.

2.3.1 System Calls for Provenance

The system call read_prov(fd, part) returns the ancestors (files containing the parent parts) for the specified part of the file. The search_prov(fd, part) call returns files which have parts that are children of the specified file part. Both procedures implement similar steps, except that read_prov and search_prov recursively follow the list of parent parts and children parts, respectively. Pseudocode for the implementation of provenance calls is given in Figure3 and 4.

```
Procedure read_prov(fd, “partname”):
   //similar to search_prov, see step 2b for details
   1.  Go through the parts in listOfParts corresponding to the
       file descriptor (fd)
   2.  If a part named “partname” is found
       a.  Call read_recursive_part with the following arguments:
           pnode number of specified part, timestamp of specified
           part, and type of provenance (either read_prov or
           search_prov)
       b.  Read_recursive_part will return a list of all parent
           (for read_prov) or children (for search_prov) files'
           inode numbers corresponding to the specified part. PTS
           ensures that the parts the procedure recursively
           follows have a timestamp less than the specified
           timestamp.
   3.  Take the inode numbers and convert them into absolute
       pathnames using the inode number to absolute pathname table
   4. Return the absolute pathnames
```

Figure 3: Read_prov will return filenames of files that contain parent parts of the specified “partname”. This function recursively goes through the parent pnodes and returns all associated filenames.

```
Procedure write_prov(fd_child, "partname_child", fd_parent,
"partname_parent"):
  1. Go through the parts in listOfParts corresponding to the
     file descriptor of the child
  2. If a part named "partname_child" is found
      a.  Store the (parent pnode number, timestamp) in the
          listOfParentParts of the child pnode
      b.  Store the (child pnode number, timestamp) in the
          listOfChildrenParts of the parent pnode.
  3. If a part named "partname_child" is not found, allocate a
     new pnode and execute Steps 2a and 2b.
```

Figure 4: Write_prov will store provenance information for the specified part. There is no recursion required; the function simply updates the child part and parent part to have parent/child pointers to one another.

2.4 Handling Unmodified Filesystem Programs

PTS allows unmodified programs (such as move, remove, and copy) to run correctly, while modifying the inner workings of certain system calls in order to track provenance.

The "move" program contains a `link()` and `unlink()` call while "remove" contains an `unlink()` call. The `link()` and `unlink()` system calls have been modified as follows:

1. `Link(from_name,to_name)`: This system call is modified because the filesystem must update INFT by creating a new entry mapping the inode number for `from_name` to filename `to_name`
2. `Unlink(from_name)`: This system call is modified to both update INFT in addition to executing the garbage collection system. `Unlink()` will delete the INFT entry mapping the relevant inode number to `from_name`. If the reference count of the inode is zero, then the following steps will be executed:
    - The corresponding data blocks will be freed
    - All parts comprising the file `from_name` will have the relevant inode number removed from their listOfFiles
    - The garbage system will be called on each part of the file to recursively free-up pnodes that have no children or files that depend on it. The system deletes entries for the specified pnode number from the listOfChildren of all corresponding parent parts.

The garbage collection system allows PTS to handle inode reuse and renaming. When an inode is freed, the garbage system frees all provenance corresponding to the inode to avoid future provenance read/write errors.

PTS tracks file-to-file provenance for programs that are provenance-unaware by maintaining a temporary table (called "TEMP") which stores a list of files that have been read by a particular process (say, Microsoft Word). This table is stored in memory and has (key, value) pairs mapping to (process_id, list of files being read). TEMP is useful to maintain provenance when a program such as "copy" (which includes `open()`, `read()`, `write()`, and `close()`) is called. The system calls `read()` and `write()` have been modified to track file-to-file provenance as follows:

3. `Read(file_descriptor)`: Make entry in TEMP corresponding to the process_id for the inode number specified by the file_descriptor
4. `Write(file_descriptor)`: Add parent/child pointers between the null pnode corresponding to file_descriptor and the null pnodes of all files being read by the process.

3. Design Analysis with Use Cases

There are several common use cases which require subtle implementations in PTS. PowerPoint copying, compiling, and zipping are explored in more detail below.

3.1 PowerPoint Slide Copying

PowerPoint developers will have to modify their program in order to read and write provenance of individual parts (or slides) of a file (or presentation) using PTS. PowerPoint should keep track of a slide's corresponding pnode number and associated pnode name.

Let us assume the user opens two presentations and copies a slide from each presentation into a new presentation. The following series of steps will occur:

1. User opens PresentationA.ppt and PresentationB.ppt
2. User creates a new presentation called PresentationC.ppt
3. User copies Slide1 from PresentationA.ppt to PresentationC.ppt
    a. PowerPoint tells PTS to create a new part. PTS will initialize a new pnode and give it a unique pnode number
    b. PowerPoint names the part that was copied into PresentationC.ppt
    c. PowerPoint calls read_prov in order to learn the provenance of the original copied slide
    d. PowerPoint calls write_prov in order to store the provenance of the new slide created in PresentationC.ppt

4. User copies Slide2 from PresentationB.ppt to PresentationC.ppt
    a. PowerPoint executes steps 3a through 3d

With the read_ and write_prov calls, PowerPoint is able to successfully track provenance of individual file parts.

3.2 Compiling Software and Copying Files

PTS takes advantage of the similarities between compiling and copying files and addresses these use cases with a common solution. Compiling (“make”) and copying (“cp”) require common steps: open, read, write, and close, described in Section 2.4. It is assumed that “make” and “cp” are provenance-unaware programs.

3.2.1 Example of Compiling Software and Copying Files

As shown in Figure 5, when reading and writing files with provenance-unaware applications, PTS will use TEMP (explained in Section 2.4) to propagate provenance information of entire files (not parts) by reference, in order to retain efficiency in memory usage. Every time a process is closed, PTS will empty the list corresponding to that process. Whenever write(fileX) is called and fileX has the same process_id as one or more files that were read, PTS updates the null pnodes of fileX and the corresponding “read” files with the same process_id.

Creating parent and child pointers between null pnodes is acceptable though every part of a file is not necessarily related to another file. However, there is no way for a provenance-unaware application to make the required read- and write-provenance calls, so the user must accept that PTS may lose provenance correctness with provenance-unaware programs.

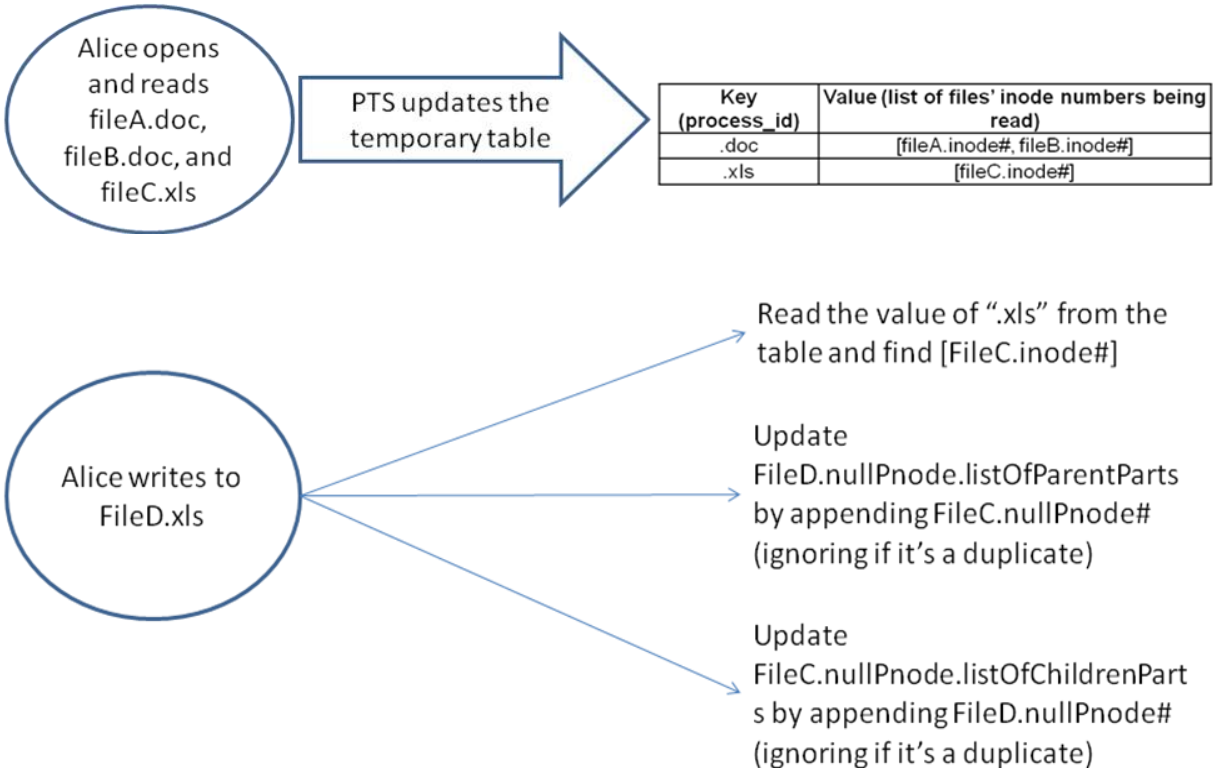

Figure 5: How PTS handles copying files in provenance-unaware applications.

3.3 Handling tar/zip Files

Zip could be modified in order to take advantage of PTS. When the "zip" call is made, a temporary file should be created that stores the xattrs (listOfParts) of all files to be zipped (and this temporary file should also be zipped). The xattrs will be stored as a lookup table in the temporary file; the inode number will map to the listOfParts of the file in question. Zip should also be modified to update the listOfParts associated with the .zip file itself; the listOfParts of the .zip file should contain the union of the listOfParts of its component files. When "unzip" is called, the program must first read the temporary file, and update the metadata of all the other files. It should not put the temporary file into the filesystem, but rather delete it. PTS propagates provenance by reference in order to maintain efficiency. This process is depicted in Figure 6.

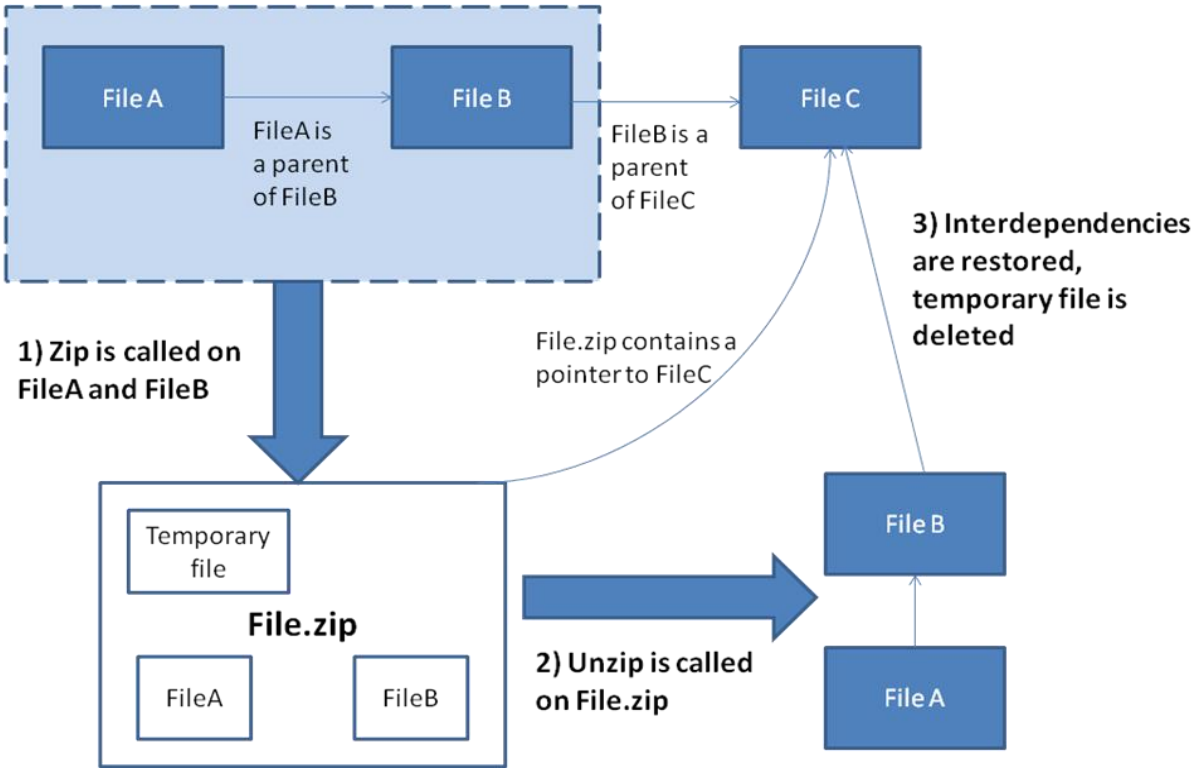

Figure 6: An example of how PTS handles zipping and unzipping files.

3.4 Comprehensive Performance Evaluation

The benefits of tracking provenance with PTS outweigh the time and space overheads. Assuming an average of two parts per file, and two parents and children per file, the total on-disk space PTS requirements increases by 2 Gigabytes as shown in Table 1.

PTS supports provenance storage, search, and continuous file copying at reasonable rates. The approximations in Table 2 are used throughout. Assuming each part has two ancestors or descendants, PTS requires 0.06 seconds on average to conduct a provenance search of a specified part (using Equation 1), and 0.02 seconds to write provenance (using Equation 2). The time required to access the TEMP table is negligible. Assuming an average user has 50 processes running with two open files being read per process, the size of TEMP is 493.75 bytes (using Equation 3). Thus, the total additional time required by PTS to copy (or search and write) a file is 0.08 seconds. The garbage collection runs during every unlink operation and requires 0.019 seconds (using Equation 4). Thus, copying can be sustained at a rate of 10 file copies per second.

Returning filenames to applications calling provenance system calls requires storing INFT (explained in Section 2.3) on-disk. Assuming an average of one hard-link per inode and Table 2 approximations, the size of INFT is 259 Megabytes (using Equation 5). The link(), unlink(), and open() system calls are modified to initiate an INFT update. Whenever a file is created or deleted, a hard-link is created or deleted, and if a directory is moved or renamed, PTS recursively updates filenames and preserves accuracy. Using Equation 6, a recursive update on INFT would take one second; I have assumed an average directory depth of 5 and an average directory size of 7 files [1].

Table 1: Space analysis of PTS.

Implementation	Space Usage in bytes
xattrs of inodes	20*10^6
pnodes	1.8*10^9
Size of inode number --> filename table	259*10^6
Total	**2*10^9**

Table 2: The approximations listed in this chart are used for performance analysis in the report

Space and Performance Approximations		
Parameter	**Symbol**	**Approximation**
Number of files [3]	F	10^6
Length of inode number field in bytes [4]	I	4
Maximum String Lengths in bytes [4]	Str	255
Seek time in seconds [5]	S	0.01
Table lookup in seconds [2]	TL	0.5
Read and write rate in Megabytes/sec [5]	R or W	100
Number of processes	P	50
Maximum Process_id length in bytes [6]	P_id	4

$$\textit{Time to search\_prov (or read\_prov)} = (S+R+(S+R)*2)*2 \quad (1)$$

$$\textit{Time to write\_prov} = S+R+S+R \quad (2)$$

$$\textit{Size of TEMP} = P*(P\_id+(2*I)) \quad (3)$$

$$\textit{Time to Garbage Collect} = S*2*2 \quad (4)$$

$$\textit{Space of INFT} = F*(I + Str) \quad (5)$$

$$\textit{Time to Update INFT} = S+TL+Str/W*7*5 \quad (6)$$

3.5 Scalability Issues

The PTS design does have some scalability limits. The time required to recursively find the provenance for a part increases linearly with the number of ancestors (or descendants). The time required by garbage collection also increases linearly with the number of ancestors of a part. Thus, PTS may take longer when faced with unwieldy databases; however, in the hospital database scenario, the time overhead is acceptable since audits do not require fast access time (unlike emergency-room records).

4. Conclusion

The proposed PTS allows users to track provenance of files and parts of files, without placing unreasonable constraints on space and time performance. Implementation of PTS is relatively simple and supports both provenance-aware and provenance-unaware applications. PTS is appropriate for use in a variety of settings including hospital databases.

5. Acknowledgments

I sincerely thank the Writing Advisors and all the TAs who asked me leading questions and made me think deeply about my design.

6. References

[1] J. R. Bolosky and Douceur, A Large-Scale Study of File-System Contents, *Proceedings of the international conference on Measurement and modeling of computer systems (SIGMETRICS)*, Association for Computing Machinery, Inc., 1999.

[2] (2012, Mar.). C# Dictionary Versus List Lookup Time [Online]. Available: http://www.dotnetperls.com/dictionary-time

[3] (2012, Mar.). DP1 Handout [Online]. Available: http://mit.edu/6.033/www/assignments/dp1.html

[4] (2012, Mar.). The Second Extended File System [Online]. Available: http://www.nongnu.org/ext2-doc/ext2.html#DEF-INODES

[5] (2012, Mar.). TA Office Hours and by appointment.

[6] (2012, Mar.). UNIX Processes [Online]. Available: http://www.cs.miami.edu/~geoff/Courses/CSC521-04F/Content/UNIXProgramming/UNIXProcesses.shtml

Word Count: 2707