Lineage File System

Can Sar and Pei Cao
Department of Computer Science
Stanford University

Abstract

We propose that file systems keep record of the lineage information of each file. Lineage records of files are easy to track, maintain and query. With today's exploding storage capacity and ever-increasing amount of user data, lineage information helps users manage their files and facilitates a range of applications.

In this paper, we define a model of lineage records for files, and describe an implementation to track, store and query the lineage information. We also list several applications, and provide experimental data for a particular application.

1 Introduction

We propose "Lineage File System", which is a file system that stores not only a file's data and its standard attributes, but also its lineage information.

Loosely speaking, lineage information is the information needed to "recreate" a file. By "recreate", we mean re-enact the series of actions that generated the content of the file. For example, the lineage information for file "paper.dvi" is "run latex on paper.tex". Though in extreme cases, a lot of information needs to be recorded to truly re-enact a process, for most applications it suffices to record a few pieces of information, including the executable, command line arguments, and the list of (timestamped) input files read by the process.

We motivate the need for Lineage File System with an application in the high-performance computing community. Today, over 500 physicists at 76 universities around the world study the data gathered by the BaBar experiment at SLAC (Stanford Linear Accelerator Center) [Mou03]. Over a period of four years, the physicists have generated millions of data products, i.e. datasets derived from the original data through expensive computations. The storage requirement is now over one petabyte. For BaBar physicists, "automated tracking of data provenance (i.e. lineage) and maximized reuse of data products are becoming a requirement" [Mou03]. Without a system to track the lineage of data files, physicists generate the same data products repeatedly, wasting both time and storage on the supercomputing cluster. While a tool can be built specifically for the high-energy physicists, there is no reason why the file server cannot provide lineage information as a general facility.

There are many other applications of lineage information, including backups, storage management, and security monitoring. Essentially, lineage information is a form of "structured" information about "unstructured" file data, and aids in the management of file data.

2 Lineage File System

2.1 Lineage Records

Ideally, a file's lineage information should contain all and only data that causally affect its content. Unfortunately, the operating system cannot readily deduce such information.

One solution would be to record all potential sources of causal information. Unfortunately, the list of sources is very long. During the execution of a process, all of the following inputs could affect its outputs: the executable, the command line arguments, disk files read by the process, external inputs from the user and the network, system call return values, library call return values, kernel and compiler versions, etc. Capturing all these data is neither practical nor necessary.

Instead, we observe that in most cases, three pieces of information suffice to capture a file's lineage: the process executable, command line arguments, and files read by the process. They form the default components of lineage records in our system. In addition, a system call is provided for a process to insert specific data into a file's lineage record, so that applications such as the browser can add the URL origin to the lineage record of the local copy of a web page.

Access Control Determining who can access a file's lineage information is surprisingly tricky. The simple choice, i.e. giving the lineage information the same set of permissions as the file itself, does not work in many cases. For example, the lineage information of a person's annual performance review file might reveal who wrote the reviews, which is undesirable[1]. The sophisticated choice, which is to set the permission to be the "intersection" (or "maximum lower bound") of the permission of the file and permissions of all files in its lineage record, is difficult to implement, particularly since permissions of the involved files can be changed at any time.

To keep things simple, we decide to use the following access control model. The owner of a file can view its lineage information. In addition, if the file is readable by all and all its input files' are readable by all, then the file's lineage information is also readable by all. We think this model works for most cases. We also plan to provide the means by which a file's owner can set permissions on its lineage record.

2.2 Implementation Modules

There are three aspects to the implementation of lineage records: tracking, storing and querying. In addition, for files that reside at a file server, the lineage records need to be sent from the client machines to the file server.

Tracking Our current implementation modifies the Linux kernel to log all process creation and file-related system calls in the *printk* buffer[2]. A user-level daemon then wakes up periodically to read the buffer and generate lineage records. Specifically, the kernel logs the following information:

- Upon process creation (i.e. fork and exec), the pid, the executable name (i.e. absolute pathname), and the command line arguments are logged. Upon process exit, the pid is logged.
- Upon file open, the file name (i.e. absolute pathname), open mode, and the resultant file descriptor are logged; upon file close, the file descriptor is logged.
- When the process reads or writes a file, the fact that the file descriptor is read from or written to is logged *once* in the buffer. This is implemented by adding a flag to the file descriptor data structure to remember whether a file descriptor has been logged or not.
- For dup and pipe system calls, the involved file descriptors are logged.
- In the case of the socket system call creating a communication endpoint, the involved file descriptor and the fact that the file descriptor is a socket are logged.
- For link, symlink and unlink system calls, the involved file names are logged. If unlink results in the deletion of a file, that fact is logged as well.
- If a file is truncated to zero byte, the involved file name or file descriptor is logged.

The user-level daemon processes the log records in the following fashion. For each process, it uses an internal table to track, for each

[1]We thank Dr. Drew Dean for giving us this example.

[2]We increased the size of the printk buffer to 8MB.

file descriptor of the process, the file name and whether the file was read or written. Sockets and pipes are assigned specially constructed unique names. When processing the fork call, the parent process' internal table is replicated to the child process' table, but with the "read" and "written" bits cleared. The dup and pipe system calls are also handled appropriately. Then, for each file that is written by a process, the process information, along with the list of files that have been read by the process, constitute the lineage record of the file.

The kernel modifications are mostly `printk` statements and hence incur little overhead. Testing with the file system micro-benchmark in lmbench [bit05] shows that only 15 microseconds are added to the creation of 0-byte and 10KB files. The user-level daemon is currently implemented in perl. It can process about 2000 records each second; during our normal usage of the desktop the daemon consumes about 10 seconds of CPU time each hour. We have not felt the need to optimize its implementation.

Storage There are basically two choices to store lineage records. One is to store them as meta files. Since the lineage record is an attribute of the file, it can be stored in a "meta" file along with the file. The advantage of this approach is that the file system can be used to enforce access control on the lineage records. The disadvantage is that querying is not supported well. The alternative is to store records in a SQL database, which provides flexible and efficient querying capabilities. Since most applications of lineage information rely on extensive querying of the records, we decide to adopt this approach and use MySQL as the backend.

The database has two tables, one for processes and one for files. The schema are shown in Table 1. The field "pid" is the key that links the two tables, and is generated by appending the timestamp of the process creation to the process pid, so that it is a unique key. To obtain the lineage record of a file "A", all a user needs to do is to run the following query:

```
SELECT prog.exec, prog.argv, input.name
FROM process_table AS prog, file_table
AS input, file_table AS output WHERE
input.mode = 'R' AND output.mode = 'W'
AND output.name = "A" AND output.pid =
input.pid AND output.pid = prog.pid AND
input.open_t <= output.close_t;
```

Note that, though logically the lineage record of a file is one "record", to store the information in SQL databases we need to break it up and store the elements as multiple rows.

The user-level daemon inserts records into the database. Upon seeing the last file write made by a process, it writes the record about the process and the records of all involved files to the database. If a process did not write any file or only wrote to a `tty` device, no record is inserted. Furthermore, when a file is deleted or truncated to zero byte, its (old) lineage information is deleted as well.

Thus, the size of the database is roughly proportional to the number of files in the file system. In our current implementation, tracking 1000 files' lineage records consumes about 360KB of storage.

Querying To implement access control, users cannot query the MySQL database directly. Rather, a phantom user, *linfs*, is created. The MySQL database file is writable only by root and readable only by *linfs*. A wrapper application, owned by linfs and having the setuid bit set, acts as a front-end to the MySQL query interface and enforces access control based on the owner information and permission bits of the file being queried[3].

SQL queries are quite versatile and can answer a variety of questions. For example, to find all existing files that are results of running "run_experiment" on a particular input file A, the following query can be used:

```
SELECT output.name
FROM process_table AS prog, file_table
AS input, file_table AS output WHERE
prog.exec LIKE "%experiment" AND
input.mode = 'R' AND output.mode = 'W'
```

[3] We have not yet implemented the wrapper application. Rather, we currently use "sudo" to issue queries.

SQL Schema for the Process Table					
Field	pid	exec	argv	uid	euid
Type	long	blob	blob	int	int

SQL Schema for the File Table							
Field	pid	name	mode	owner	permission	open_t	close_t
Type	long	blob	char	int	short	timestamp	timestamp

Table 1: Schema for the process table and file table. "blob" is a MySQL type that means arbitrary string.

```
AND input.name = 'A' AND output.pid =
input.pid AND output.pid = prog.pid;
```

File Server Implementation In networked file systems, the lineage information is tracked by the client machine rather than the file server. However, since lineage information is a kind of file attribute, it should be stored at the file server. Hence, the records need to be passed from client machines to the file server, and queries should be answered by the file server.

Our current design is that clients use a set of RPC calls to send lineage records to the file server, and to send users' queries to the file server. Standard authentication methods such as Kerberos can be used to establish users' identities to the file server, and the file server enforces access control on read/write of lineage records. However, we have not yet implemented this scheme.

3 Applications of Lineage Records

There are a variety of applications for lineage records. Below, we list a few:

- *Remembering what a file is about*: Lineage File System is motivated in part by the difficulty of managing our own files. Due to the plunge in storage prices, there are fewer and fewer occasions when we have to delete files. As a result, our directories have files from many years ago. While it's easier to figure out what a file is about if the file is somehow readable by human (e.g. text), for a binary file the only clue is its name. Unfortunately, since we tend to use short file names, the names stopped making sense after a few years. Lineage records can help us remember what a file is about and whether it needs to be kept or deleted.
- *Space management on small backup drives*: Many people use portable USB drives or web-based network drives for backups. These devices usually have limited capacities and cannot hold all files. Currently, the users manually decide what files are backed up. If lineage records are available, a tool can be written to classify files into "root" files, whose contents involve user input or external events that are hard to replicate (e.g. source codes), and "generated" files, whose contents are generated from others by applications (e.g. compiled object files). Then, only "root" files need to be stored on the USB drive or the web-based backup service.
- *Security monitoring and forensic analysis*: One can build a host-based intrusion detection tool by observing normal patterns of lineage records and sounding an alarm when a pattern is broken. For example, if a file (e.g. registry file) has always been written by one particular executable (e.g. registry editor) but is suddenly being written by a different executable (e.g. the virus), an alarm can be triggered. Lineage records are also very useful for forensic analysis, since they capture which executable last changed a file and what input files were used.

- *Determining the type of a file*: The name of the executable that created and wrote a file is an extended "type" information of the file. This "type" information is much more extensible and usually more reliable than file typing using suffix or magic numbers. It can be used in any application that uses "type" information, for example, restricting search results to files of a particular type, or applying a specific compression tool to all files of a certain type.

- *Automated "make"*: In application domains such as scientific computing, when a commonly used tool has been updated or has a bug fixed, many simulations need to be rerun. The lineage records allow the system to do so automatically without asking users to explicitly write makefiles. Of course, the automated make needs to be under user direction, so that it doesn't inadvertently change a file when it shouldn't.

In essence, Lineage File System provides some "structured" information for the "unstructured" files, which can be used in a variety of storage management applications.

Experience with one application We have written a space management tool for our own use. The tool uses a simplified criteria to determine whether a file is a "root" file; if the process that modified the file read from its standard input, then the file is considered a "root" file. During a day when the main activities on the desktop consisted of working on this paper and making kernel changes, we found that out of the 314 MB of new data, only 2.5MB were root files and needed to be stored on the USB.

4 Related Work

The file system research community have long looked into making the file system more intelligent and easing the task of storage management. For example, the Elephant file system [SFH^{+}99] automatically retained all important versions of files to provide an undo facility at the file system level and to protect against users' accidental overwrites; disks that provide time-travel capabilities [MG03, WCG04] have been used to debug configuration changes [WCG04], among other applications. More recently, a number of projects focused on associating rich attributes with files (either manually provided [DJ91, Bow97, XKTK03] or automatically derived though context analysis [SG03]), and providing flexible querying and view-constructing capabilities over the attributes [DJ91, XKTK03, GM99, SM, Neu92].

While sharing the same high-level goals with these projects, our work differs from them by focusing exclusively on lineage records and exploring their potential applications. Lineage records are a type of file attribute and can be managed by file systems that handle rich attributes. In practice, however, it is easier to centralize lineage records in a MySQL database and leverage its rich query interface. Lineage File System is complimentary to other intelligent file system efforts by providing more information about files, for example, lineage records are a source of information for automated file attribute assignment [SG03].

The database research community have long studied data lineage issues, particularly reasoning the precision and correctness of derived data. Recently, the Trio project [Wid05] tries to build a database that manages not only data, but also the accuracy and lineage of data. In these studies, lineage means lineage through database queries and constructions, which are much more complex and have much richer semantics than the "lineage" discussed in this paper. The "lineage" discussed in this paper is only a *generic* lineage in the operating system context. Thus said, Lineage File System can use lineage databases to manage its data, and lineage databases can use Lineage File System to provide input about external dependencies.

Finally, unlike file systems that are implemented on top of databases (e.g. the inversion file system [Ols93] and the Microsoft Longhorn file system), lineage file system does not make changes to the underlying file system, and uses databases only to store the lineage records.

5 Conclusions and Future Work

Through a prototype implementation, we have demonstrated that lineage records are easy to obtain, store and query. We have also shown that they enable a rich set of applications, and provided data on a particular application.

There are two focuses for our future work. One is to implement a way to track lineage records without changing the kernel, as the kernel changes are a deployment impediment. A potential approach is to change the shell to trace the file system calls of all its children processes, and to implement a separate daemon to process those calls to obtain lineage records. The second focus is to explore ways to use lineage records for security monitoring and intrusion detection. We plan to build a tool that can be trained to learn normal lineage patterns, detect abnormal operations and issue alerts.

References

[bit05] bitmover.com. Lmbench - tools for performance analysis. In *http://www.bitmover.com/lmbench/*, 2005.

[Bow97] M. Bowman. Managing diversity in wide-area file systems. In *Second IEEE Metadata Conference*, September 1997.

[DJ91] Mark A. Sheldon David K. Gifford, Pierre Jouvelot and James W. O'Toole Jr. Semantic file systems. *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 16–25, October 1991.

[GM99] Burra Gopal and Udi Manber. Integrating content-based access mechanisms with hierarchical file systems. In *Operating Systems Design and Implementation*, pages 265–278, 1999.

[MG03] C. B. Morrey and D. Grunwald. Peabody: the time-travelling disk. In *20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies*, April 2003.

[Mou03] Richard P. Mount. High-engergy physics data-storage challenges. In *SuperComputing 2003: Workshop on Storage on the Lunatic Fringe: Beyond Peta-Scale Storage Systems*, 2003.

[Neu92] B. Clifford Neuman. The prospero file system: A global file system based on the virtual system model. *Computing Systems*, 5(4):407–432, 1992.

[Ols93] M. A. Olson. The design and implementation of the Inversion file system. In *Proceedings of the USENIX Winter 1993 Technical Conference*, pages 205–217, San Diego, CA, USA, 25–29 1993.

[SFH+99] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton, and Jacob Ofir. Deciding when to forget in the elephant file system. In *Symposium on Operating Systems Principles*, pages 110–123, 1999.

[SG03] Craig A.N. Soules and Greg Ganger. Why can't i find my files? new methods for automating attribute assignment. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems*, pages 145–150, 2003.

[SM] Stuart Sechrest and Michael McClennen. Blending hierarchical and attribute-based file naming. In *International Conference on Distributed Computing Systems (Yokohama, Japan, 9–12, June 1992.*

[WCG04] Andrew Whitaker, Richard S. Cox, and Steven D. Gribble. Configuration debugging as search: Finding the needle in the haystack. In *Operating Systems Design and Implementation*, 2004.

[Wid05] Jennifer Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *Proceedings of the 2005 CIDR Confernece*, January 2005.

[XKTK03] Z. Xu, M. Karlsson, C. Tang, and C. Karamanolis. Towards a semantic-aware file store. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems*, pages 145–150, 2003.