Provenance Tracking in the Unix Filesystem

Merritt Boyd (mboyd@mit.edu)

6.033 Design Project 1
March 22, 2012
Recitation: Shavit / Moll 2PM

Introduction

This document presents a design for adding provenance tracking to an existing versioning filesystem. Provenance tracking provides answers the following question: given some part *p* of some file *f*, from what parts of other files is *p* derived, and what files have parts derived from *p*. By default, we define one part per file containing all of the file's data, and define a file *f1* to be derived from a file *f2* if the processes that wrote *f1* read *f2* during its execution. We provide an interface allowing applications to specify provenance explicitly using their own notions of "parts" and "derivation" to support finer-grained provenance tracking, and an interface to read the provenance data for all parts in a file.

The resulting provenance data encodes a bidirectional graph representing the movement of data between files. We require traversal of this graph to compute the "full" provenance for a file.

Design

Data model

Our design builds on a versioning filesystem, from which we require the following features: first, for each file on the disk, the versioning filesystem should provide storage for multiple "versions" of the file, containing its state at some past point in time. These versions should be retrievable directly using some identifier, for instance the file name plus a timestamp. Second, the versioning filesystem is allowed to periodically garbage collect versions to save space, either automatically or at user request; however, we require a hook in the versioning filesystem allowing our code to be run before a version is deleted, in order to maintain the integrity of the provenance data. Third, the versioning filesystem must allow us to disable versioning for our internal files. We make no changes to the on-disk storage model of the versioning filesystem; all of our modifications occur at the inode layer or above.

We represent the provenance data for a file *f* using two tables, *parent* and *child*. *Parent* stores records for parts of *f* that were derived from other files, while *child* stores records of parts in other files derived from *f*. Each record encodes a derivation relationship between one part in *f* and one part in some other file, and contains four fields: *src_part*, *dest_file*, *dest_part*, and *flags*.

The *src_part* field identifies a part of *f*. Part identifiers may be any unicode string. The part identifier "*" has special meaning, as it is used by the system to indicate the default part, encompassing the entire file. All other identifiers are defined by the applications that write them. Parts need not refer to contiguous regions of the file on disk, and may overlap — they are intended to be used to identify logical sections within a file, and are thus dependent on the data model of the originating application.

The *dest_file* field identifies the file containing the analogous part to *src_part*. This should be a identifier known to the filesystem, representing a specific version of some file, prefixed with the "file://" URL schema.

The *dest_part* field identifies which part of *dest_file* has a provenance relationship with *src_part*. It is a part identifier, following the same schema as *src_part*.

The *flags* field encodes metadata about this provenance entry. It is represented as a 64-bit bitfield. Currently defined flags are: *deleted*, which specifies that *dest_file* has been deleted and is no longer available.

The *parent* and *child* tables are stored in files themselves, although we do not intend for these files to be user-visible. Rather, we augment versioning filesystem's inode structure with two additional fields containing pointers to the inodes for the *parent* and *child* tables. The table pointers for the tables themselves are set null, and we disable versioning for the table inodes.

Implicit provenance tracking

Provenance tracking functions properly for files written by provenance-unaware applications, although the usefulness of the resulting provenance records is greatly diminished. To accomplish this, we modify the *open()* and *close()* syscalls in the operating system kernel. For each process, we augment the kernel's data structures to store a list of all files ever opened by the process, and modify *open()* to update this list. We then modify *close()* such that when a write-mode file is closed, we write each file on the list as a *parent* provenance entry for "*" in the closing file, and update the *child* tables appropriately.

Explicit provenance tracking

For provenance-aware applications, we provide the following API:

```
void prov_setMode(int mode);
```

Set the provenance tracking mode for the current process. Values of 0 and 1 specify implicit and explicit modes, respectively.

```
int prov_write(FILE src, char *srcPart, FILE dest, char *destPart);
```

Write a provenance record, indicating part srcPart of the file src was used to derive destPart of the the file dest. This call updates both src's *child* table and dest's *parent* table appropriately; raw access to these tables is not provided to maintain consistency. The return value indicates whether or not an error occurred.

```
(prov_rec *) prov_read(FILE f);
```

Read the provenance table for a file. Returns a pointer to a null-terminated array of provenance records, which contain the fields described in the Data Model section. Applications are responsible for using the provenance records to traverse the provenance graph as required.

Garbage collection

It is expected that the versioning file system will periodically reclaim versions of files to save storage space. Before the system reclaims file *f*, we walk both *f*'s *parent* and *child* tables. For each entry (*src_part*, *dest_file*, *dest_part*, *flags*), we open the provenance table for *dest_file*, find the entry back-referencing *f*, and set the *deleted* flag in that entry. This indicates that applications traversing the provenance graph for *dest_file* (including the system itself) should not attempt to traverse *f*'s subgraph. This is robust even in the case of filename re-use, as the *deleted* flag prevents applications from mistakenly reading the wrong file's provenance tables.

External files

The system is capable of recording the origin of files imported into the system via a network. The *dest_file* field of a provenance record is normally prefixed with "file://" to indicate a local file. However, other prefixes may be used, such as "http://" URLs, to

indicate files of non-local origin. Applications must handle such alternate prefixes appropriately when traversing the provenance graph.

Provenance blacklisting

The system provides a mechanism to exclude regions of the filesystem from provenance tracking. This might be useful for security or privacy reasons, or to simplify the provenance graph. The superuser may load a list of exclusion regular expressions into the kernel filesystem driver. Before writing any provenance information with `prov_write`, the driver checks if either the source or destination are matched by any of the exclusion patterns, and if so cancels the write and reports an error.

Analysis

Use cases

PowerPoint slide copying:

> Large user-facing applications such as Microsoft PowerPoint are expected to add support for explicit provenance tracking and integrate it into their UI. In the case of PowerPoint, this could be accomplished by defining parts of the form “slide1,” “slide2,” etc., and placing part and file information in the system pasteboard when a slide is copied. When a slide is pasted into a document, the source file and part can be used in a call to `prov_write`. PowerPoint could additionally provide an interface built around `prov_read` to allow the user to browse the provenance of individual slides.

Compiling software:

> Unix tools such as make and gcc are unlikely to add explicit mode provenance support, nor do they require it. Source files read as the input to a build stage will be tagged implicitly as parents of the build output, and if the download tool (e.g. wget) supports explicit mode, the binary could be traced back to the URL used to download the source.

Copying files:

> The standard UNIX cp command will work properly in implicit mode. Copies made in this manner will have a single *parent* record pointing to their source file — the provenance tables of the source file are not copied. This is done because without augmenting all calls to *read()* and *write()* (which would induce significant

performance overhead), it is impossible for the system to determine the difference between cp and a mutating command such as tail, which should clearly not produce a provenance table copy.

Handling tar/zip archives:

Since tar and zip are required to produce provenance table copies, they must be modified to support explicit provenance mode. This would entail reading the source files' provenance tables and writing them into the archive. When the archive is unpacked, the provenance tables must then be written back, with the file names changed to reflect the new locations of the unpacked files.

Performance

Our design does not involve modification to the system *read()* or *write()* calls or require any background processing beyond what is done by the versioning filesystem, so for the most part the system performance will be unaffected. Overhead in implicit mode is strictly limited to the *open()* and *close()* syscalls. Our modification to *open()* is trivial and requires no disk access and only a small increase in memory usage due to storage of the open file list — no more than 10K or so for typical workloads. Overhead on *close()* is more severe, incurring a seek + write for the closing file and all historically opened files. For processes that open large numbers of files, this could result in a large amount of seeking for each write-mode file closed. As a workaround, some files could be added to the provenance blacklist, or the offending process could be run under a wrapper that turns off implicit provenance tracking.

Reading back provenance data involves a graph traversal, with the performance dominated by the seek across each edge. This is acceptable, and a better solution than storing an entire provenance tree with each file. First, provenance readbacks are expected to be done relatively infrequently, usually in response to user input. The graph traversal makes it easy to load results asynchronously, so the user need not wait for the entire structure to load to see the most immediate entries, which are likely the most useful. Furthermore, a graph structure allows efficient storage of far more data than a user might typically request — for instance, an application may limit its traversal to 10 files deep by default, but may allow the user to search deeper. We allow storage of an arbitrarily large provenance graph, with no overhead except during readback.

With regard to copy performance, the exact behavior depends on the implementation of the copy command. Standard UNIX cp will easily meet the goal of 10 copies per

second, as even in implicit mode only a single pair of provenance records are written (for the source and destination). A copy operation should, then, involve four seeks: first to the src, then to the destination, then two additional seeks for the provenance tables. A relatively modern, low-performance drive such as the Western Digital Caviar Green has an average seek time of 8.9ms, and assuming that the seek times dominate a copy, that places the copy time at approximately 35.6ms, for a rate of around 28 copies/sec.

Deletion handling

When a file is deleted, either as a result of versioning filesystem garbage collection or user request, the provenance graph is broken at that point — files derived from the deleted file maintain their references, but become unable to the provenance subgraph behind the deleted node. We believe this is a correct solution for several reasons. First, it makes it simple to handle deletions correctly, and reduces the amount of background processing necessary to maintain the provenance graph. Second, we believe this model corresponds with user expectations with regard to deletion. Deleting a file should reclaim all storage associated with the file, and make information about its contents unavailable. To keep provenance information for deleted files would prevent the amount of storage consumed by provenance from decreasing without complicated garbage-collection schemes and their associated performance impact. Second, hoisting provenance entries from a deleted file into its children preserves information about the contents of the deleted file, which is strongly undesirable from a security / privacy standpoint.

Conclusion

Our design provides a system for tracking the provenance of a file's parts, and allows efficient lookup of a part's parents and children in the provenance graph. By leaving the definition of file parts and their derivation relationships to the application, we allow a large degree of flexibility in the use of the system, while still providing useful behavior for applications ignorant of provenance. Performance impact is mostly minimal, with no impact to bulk read and write performance and no required background processing. Provenance readback is efficient, and allows the application to specify how much data to retrieve at one time.

Some issues remain, particularly in the handling of implicit mode. We have chosen to be very general in the way we imply derivation relationships, which has a cost both in lower performance when writing the provenance information and in cluttering the

provenance graph. It may well be the case that a more limited heuristic-based approach is superior. Additionally, some users may be unhappy with the way we handle file deletions, and prefer that provenance information be preserved for deleted files. Finally, requiring applications to perform their own graph traversal puts additional burden on the application developers, although this will likely be mitigated by the creation of convenience libraries wrapping our API.

References

J. Saltzer and M. Kaashoek, Principles of Computer System Design: An Introduction. Burlington, MA: Morgan Kaufmann, 2009.

Word Count

2,274, including these.