Notes on solutions to 6.033 DP1, Spring 2012

Peter Szolovits

The design project asks you to design a provenance tracking file system with several required characteristics:

1. It should support the idea that files may have *parts*, each of which may have its own provenance. There must be API support to allow programs that manipulate such parts to specify provenance for them.
2. Provenance tracking of files should work even when files are manipulated by programs (such as `mv` and `cp`) that have not been specifically modified to support tracking.
3. Your system must be able to operate continuously, without exhausting resources on "dead" provenance information.
4. Provenance information must be stored persistently, to survive power cycling, though you do not need to worry about crashes, malicious behavior, bugs, etc.
5. You should be able to copy files at a rate of at least 10 files/sec.
6. You should be able to identify the complete provenance of a file in a million-file system in a few seconds.
7. Your system should not use "excessive" resources – which are not precisely defined.

In your write-up, you are asked to consider how you would design each of the following:

1. The RAM and disk data structures that implement provenance information.
2. Whether entire provenance trees are stored in total with the files to which they apply or are computed on the fly from only "local" provenance data that connects to immediate ancestors and/or children.
3. How parts are named, and what mechanisms can look up those names to find corresponding provenance information.
4. How your design deals with deleting or renaming files that participate in provenance tracking.
5. How you deal with avoiding false attributions of provenance when the time sequence in which operations take place is critical to determine provenance. The example given is

   "Consider a user that copies a PowerPoint slide from file B to C, and then replaces that slide in B with a different slide from file A. If the provenance for this slide in file C points to the corresponding slide in B, and the provenance for B's slide now points to A, it may now appear that C's slide depends on A's slide, even though this could not have happened."

   The assignment suggests that a *versioning file system* may help, although it does not specify what is meant by such a system. There is also a suggestion

that file systems extended attributes (xattrs) may be helpful, though not necessarily for this consideration.

6. Four use cases are to be analyzed:
    a. Copying powerpoint slides
    b. Compiling software using make
    c. Copying files with unmodified programs
    d. Handling tar or zip files, retaining provenance information among the files included in the archive.

Abstract Solution

There is no single "best" solution to this assignment, so I will try to present an abstract view of some of the challenges faced by all the students and then suggest various good ways to meet these, as well as a few common bad ways. This document is *not* intended to be a model solution, because such a solution must actually make design choices that are only described here.

Identifying Files

Most people chose to use inodes and their inode numbers as the representation of files and pointers to files. This is clearly within the spirit of *nix systems. A few used file names, and many of these designs needed to map from an inode number to a file name. Of course given the possibility of multiple links to the same file, the file name is not a unique identifier for a file, so such designs often led to additional problems.

Parts

We need some machinery to permit naming of parts of a file. The simplest idea is to store a mapping per file that associates a name with a part number, and to leave details of how those names/part numbers are used to the application. PowerPoint, for example, could name its slides "Slide1", "Slide2", etc. We want the mapping to numbers so we can have a compact representation of a file part when we store provenance information. E.g., if we limit ourselves to 64K parts in a file, then two bytes suffice. Typically, part number 0 is reserved to refer to the whole file.

There are various places one could store the mapping from names to numbers, including in the file's inode's extended attributes, in the file itself, or in some data store elsewhere in the file system. The mapping can be implemented by a hash table, b-tree, or other efficiently searchable data structures.

Should parts be recursive? To make it simpler to support aggregations that have multiple levels of parts, we might consider allowing parts to have parts. For example, a zip file could consider the files it includes to be its parts, and if among those are PowerPoint files, they could in turn have their own parts. Slides could in turn be decomposed into text fields, diagrams, etc., each of which is a part of the slide. If we take this approach, then we could use a hierarchic naming system such

as what Unix uses for files, but this time to name hierarchic arrangements of parts. This can be simpler than file names/inodes if we disallow aliasing/linking. If 64K parts in a file suffice, we could still map these hierarchic part names to a compact number, much as Unix maps file names to inode numbers. Doing this would require some syntax for file parts, and might be combined with file name syntax to create parts such as

/u/psz/6.033/dp1/presentations/dp1.ppt:slide5:body:leftdiagram:circle1

where the (hierarchic) part name separator is a colon (:). In this syntax, the reference to the file itself is just the file name without any parts. If we use four-byte inode numbers to identify files and a two byte number to identify file parts, then a provenance reference to part 17 of the file with inode number 256 would be x000000100011.

A few students simply use separate files for each part, and then create a bundle of these that looks like a traditional file. (This technique is fairly commonly used in Mac OS X, where large items such as iPhoto archives and applications have a lot of internal structure that consists of separate files, but are made to look like a single file in the GUI.) However, at the *nix interface, this exposes at the application level a design choice belonging to the file system. Some students suggested using separate blocks in the underlying file system to store each part, but to keep them under a single inode. Then, the block number can be associated with the part. Either of these approaches wastes a lot of space if parts are very small and numerous, and potentially creates a problem if one tries to copy such a multi-part file with unmodified utilities such as `cp`, unless the file inode makes them look like part of the content data. Such a solution runs the risk of failing to carry information that identifies and names the parts when using an unmodified program to copy the file.

Versions

To deal with consideration 5 in our list above, we need to be able to distinguish among different versions of a file. This can be handled in a variety of ways, but depends on a few overall design choices:

a. Do multiple versions of a file all share the same inode structure, or are they stored via different inodes? Sharing a single structure can facilitate sharing of blocks that don't differ in different versions, though copy-on-write schemes could achieve that goal via other organizations as well.
b. We need some user-level way to specify different versions of a file. For example, we could append a version number or a timestamp to the filename. Depending on choice (a), this would require either augmentation of the filename-to-inode translation method or segregating the version identifier and using it along with the inode to find the blocks of the appropriate version.

If we were to append a timestamp to the inode number, as one possible implementation, and if we assume a six-byte timestamp, then a reference to a versioned file part would require 4+6+2 = 12 bytes.

We may also explicitly thread the versions of a file, so one can move backward and forward among them, or store version information in the inode depending on the decision about issue (a) above. Alternatively, a version dependency could also be recorded as a type of provenance.

Provenance

If we consider all the provenance relationships within a file system, they form a directed acyclic graph (DAG), in which each edge represents a *dependency* of the head end (child) on the tail end (parent). If we have a good solution to the problem of consideration 5 (false parenthood), then the laws of causality will prevent cycles. Note that any node can have from none to many parents and/or children. An important design choice is whether to store only the local links in this graph and to compute their transitive closure if we need, say, all ancestors of a file, or whether to store all the ancestors with the file. This is a classic time/space tradeoff, and can cause severe problems either way because the total ancestor list of a file can be enormous, thus taking either much storage or many disk operations to recompute. Consider, for example, the ancestry of the Linux kernel file on a developer's machine, where probably every source file, header file, library, etc., should be included in the ancestry.

To add provenance for parts, we can augment this graph by nodes that represent the parts (perhaps those 12-byte identifiers) and another type of arrow, which represents the part relationship, where an arrow points from a file to its parts. A design question arises: should a part that depends on a part from a different file (e.g., a PowerPoint slide) also depend explicitly on the file from which the slide originated? In fact, should it depend on the file of which it is a part? If parts point back to their containing files, we have the same choice as above for whether to compute compound relations on the fly or to store them, with the same trade-off.

Another question is how the data about provenance is supposed to interact with versioning. In most cases, I assume that a new version of a file will be derived from a previous one, so that previous one will not only be in its version history but also appear as part of its dependency graph. However, the two might be distinct considerations. For example, I could create a brand new header file for a program to replace an old one of the same name, in which case the old one should probably count as a previous version but not a dependency.

Storage of the dependency graph

As described for the question of where to store parts naming information, we have many choices for where to store the DAG that represents all of these types of dependencies, ancestries, part dependencies and versions. Abstractly, any storage capable of holding a DAG with multiple types of arrows would suffice. It could be in (a) a database, (b) within the inode structure, its xattrs, or the machinery created to name file parts, (c) a low-level parallel to inodes (sometimes called pnodes) especially created to store dependencies, (d) special files distributed thoughout the file system, etc. Some students created `prov` and `anti-prov` directories to

parallel each directory and hold files with the same names as the files about which they held backward and forward link information (or similar naming conventions). This requires no change to the underlying file system, but does expose these files to user scrutiny and manipulation.

Evaluation

I was willing to accept most designs as long as they were internally consistent and seemed to meet the specifications of the assignment. I suspect that actually trying to implement all the mechanisms proposed would uncover uncomfortable problems in many of the designs that would require revision, but neither the students nor I tried to implement these, so I probably only caught the more egregious flaws.

Use Cases

Unmodified Programs such as cp and mv

Most reasonable solutions to this challenge proposed modifying file `read` and `write` commands (or `open` and `close`) to record files that were opened/read by a process as it was writing another. Then, they assume that the files read are part of the provenance of the files written. There are two different versions of this assumption, but both create problems.

1. If we assume that the provenance of a file contains only those files that are open simultaneously with its writing, we may miss some provenance. For example, if a program reads and caches a file (closing it when done reading) and then later writes another that depends on the first, then this technique will fail to recognize that the first file belongs to the provenance of the second. Most students assumed that a program such as `cp` will not cache entire files, so the input and output will be open simultaneously.
2. If each process maintains a list of all the files it has ever opened and attributes them all to the provenance of a file it later writes, it may over-generate. For example, a web server might open myriad files to deliver them to clients, but on a subsequent `POST` transaction that updates one file, it may be unreasonable to consider all the previously-read files as part of its provenance.

Wherever the provenance graph is stored, it will typically need to be updated to reflect the provenance of files created by such unmodified programs. This might be done by modifying `close`, but in general is a challenge.

Adding provenance to a file system is probably a major enough change that most programs will need to be updated to deal with it correctly. Work-arounds such as this are only heuristic solutions that will often be incorrect.

PowerPoint

We have to assume that the PowerPoint program will be modified to deal with parts. It can, therefore, obey whatever design choices were made for how parts are to be represented and provenance relations among them are to be kept.

Make

With the changes to support unmodified programs, make could work without change, with the caveat that it might make the kinds of errors discussed above. A binary load file will wind up depending on essentially everything that went into its build.

Tar or Zip

The special challenge here is to retain provenance information for the files in the archive when they are extracted. This requires some machinery to maintain provenance among elements of the archive as well as provenance links between files in the archive and those in the rest of the file system. Depending on where the provenance graph is stored, the relevant parts of it must be reflected inside the archive. This poses problems if that graph is in terms of inode numbers, because different files within an archive do not have their own inode numbers. Further, when the archive is expanded, that process will create new files with new inodes, so the provenance information must be installed on these new inodes to reflect relationships within the archive and (if expanded on the same system as where it was created) to other files that were not in the archive. There are numerous possible solutions to this challenge, but I did not see any that were completely satisfactory.

API

The proposed API needed to support the required operations and to be consistent with the logic of the overall design. Some students described only the API calls listed in the assignment; others introduced new ones as needed by their designs.

Analysis

Most students assumed that they would use SSD storage, which gives a much larger, comfortable margin for performance requirements. Grading on this section looked mainly at whether you had done some sensible analysis, not necessarily on whether I was convinced that your analysis would hold up if your system were actually implemented. I was looking for analyses of storage volumes, provenance retrieval times, and continuous-copy behavior. The major failing here was that a number of students did not even attempt any serious quantitative analysis but just hand-waved (often incorrectly) about their system. An acceptable analysis considers real data structures, disk accesses, etc., and provides estimates in terms of bytes of overhead and milliseconds of time.

4/15/2012