Creating a Providence-Tracking File System

Max Nelson
6.033 Design Project 1
Proposal
March 1, 2012

Overview

The goal of this design project is to create a provenance tracking file system. The provenance of a file is defined to be the set of files that have influenced some part of the file's data. The provenance can be changed in two ways: when a process writes to a file, all files read by this process will be added to the provenance of the initial file, and when an application manuals updates the provenance using a system call. This system can be realized through the creation of ".prov" files, outlined below.

Design Description

Definitions

First, let's define two terms, parent and child:

If file C is influenced by file P then P is the *parent* of C and C is the *child* of P.

Data Structures

The overarching plan to maintain the correct parent-child relationships for all files is to keep two data structures:

- An on-disk ".prov" file for each file we are tracking the provenance of (i.e. example.ppt would have a hidden example ".prov" file in the same directory) that holds the "child-of" provenance information for that file in a tree structure. For example, if a line from file A is copied into file B, then file A would be in B.prov.
- An in-memory tree structure that stores the "parent of" information for all open files. This structure is used for bookkeeping as well as allowing the user to search for a file's ancestors. This structure must be stored to a specific ".prov" file on disk during system shutdown to persist the "parent-of" information for all files.

The on-disk ".prov" files and in-memory trees can be a standard B-tree (an example tree structure as well as an example grammar for a ".prov" file is shown in Figure 1) or a more optimized tree, such as an van Emde Boas tree. These options will be explored during implementation of the system.

Importantly, the trees will store their information by reference as opposed to by value. Storing by reference only keeps the pointer to a specific ".prov" file instead of keeping the entire copy of the file. For example, imagine a scenario where file A is a parent of file B, and file B is a parent of file C. C.prov would only contain a reference to B.prov, which would in turn keep a reference to A.prov. While storing by reference is efficient on space (which is critical when we are storing so many “.prov” files), it requires some extra bookkeeping when modifying or deleting files, which will be further detailed below.

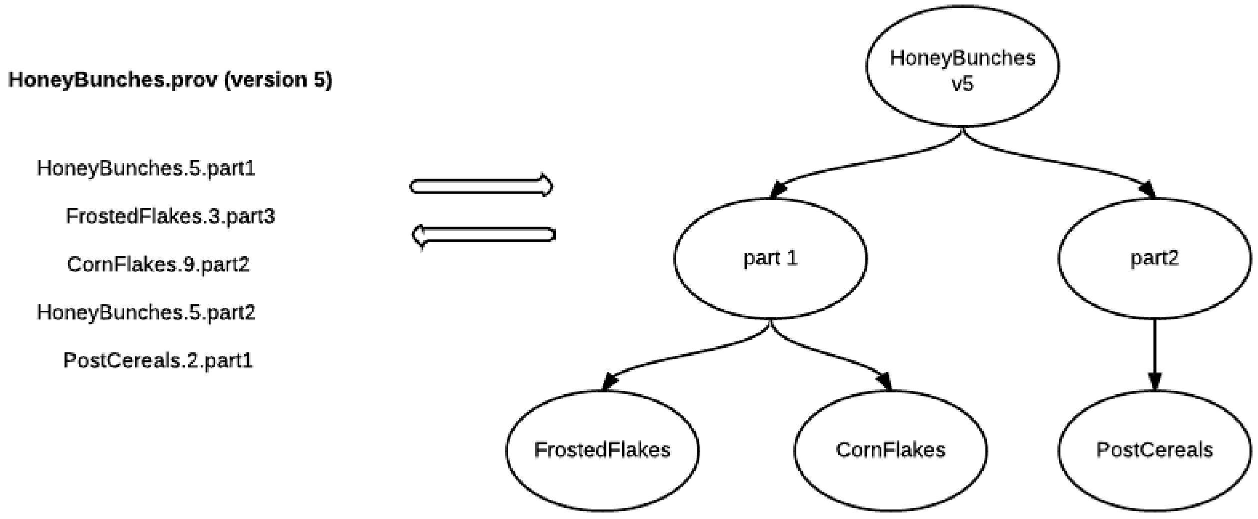

Figure 1
In this example, we see a sample grammar for ".prov" files. We can see the grammar is [filename].[version].part[partNumber]. On the right we can see the actual tree structure

To maintain these data structures, we will need to concern ourselves with the following operations:

- **Creation** - When a file is created we need to also create its associated “.prov” file
- **Moving/Renaming** - When the absolute path of a file is changed, we must update all children and parent provenance files with the new path
- **Modifying Data in the File** - Operations such as “copy paste” must add the file to the provenance information of the recipient of the operation
- **Deletion** - Similar to moving/renaming, the parents and children of a deleted file must update their tree structure with the deleted file. This in turn, means that our system must forge new “connections” between files during deletion. For example, if A1 and A2 are parents of B, and C1 and C2 are children of B, if B is deleted, it should be the case that A1 and A2 are now immediate parents of both C1 and C2 (as shown in Figure 2).
- **Garbage Collection** - As mentioned in Figure 1, a versioning file system will be most likely used. A versioning file system allows us bypass many

tricky cases with deletion/modifying. However, this means we must concern ourselves with garbage collection. After some number of versions of a ".prov" file (i.e. 10) have been created, we should delete the oldest disk ".prov" file as well as references in the children and parents.

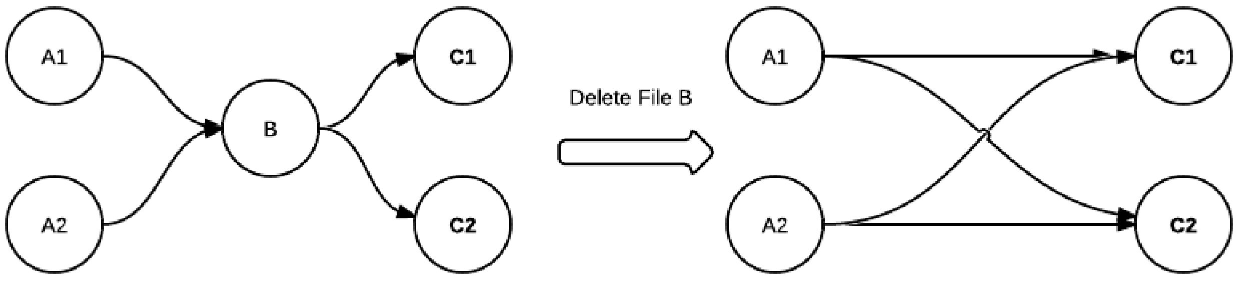

Figure 2
When File B is deleted, we extend B's parenthood to A1 and A2

Applications Interacting With Provenance Information

The OS will provide the following API to applications:

`read_prov(fd,part)`

Specification: returns the provenance info for a file

`write_prov(fd,part,provinfo)`

Specification: sets the provenance info for a file

`search_prov(fd,part)`

Specification: returns the children info for a file

Supported Applications/Operations

Powerpoint-the copying, rearranging, modifying of slides will preserve the provenance info for each part (i.e. if slide 1 and 2 are swapped the on-disk and in-memory trees will mirror this change.
Compiling Software-the sources use to compile a binary will be in the provenance file of the binary.
Coping Files- the copying operation should add/update the provenance info for the correct file, propagating by reference.

tar/zip Operation- a tar/zip operation should preserve the provenance information of the compressed files, and the compressed file's own provenance information should contain all files within the compressed file.

Conclusion

The above design meets the requirements as outlined in the problem statement by utilizing the idea of ".prov" files pointing to parents and children of files. Future questions that still remain are the choice of tree to use (or possibly a hash table if space is available), as well as the specifics of the implementation, which will involve the modification of, among others, the read/write system calls.