# Provenance: Sensible, Extensible Tracking

6.033 Design Project 1
Vladislav Kontsevoi, vok@mit.edu
Szolovits, Fang, and Volaitis, 2PM
March 22, 2012

## Introduction

Conventional file systems do not support tracking file *provenance*, or where the contents of a file come from. In addition, they cannot track how data evolves and how files influence one another. This is problematic, for example, when a user wishes to determine the source of PowerPoint slides or files in a ZIP archive.

In this report, we outline the design of PEST, a sensible and extensible system for tracking provenance. Provenance can be visualized as a graph, linking each file with its *ancestors*, the files that have directly affected its contents, and its *descendants*. PEST uses on-disk data structures associated with each file to store links in the provenance graph, and a central in-memory data structure to track file access. The primary design goals are high performance under common use cases, and simplicity to facilitate implementation. Additionally, to ensure greater portability of provenance data, PEST stores provenance in a decentralized manner. This portability might come at the expense of provenance propagation overhead.

## Design

PEST represents the provenance graph implicitly by storing pointers to a file's ancestors and descendants. PEST's data structures are updated through UNIX-like [1, 2] system calls.

The file system also enables tracking provenance across *parts*, which are abstractions that might, for example, represent slides in a PowerPoint presentation or files in a ZIP archive. While the storage of parts within a file is application-defined, parts are referenced by 64-bit *part numbers*, with part 0 referring to the file itself.

### Data Structures

Two main types of data structures work together to enable provenance tracking within PEST: on-disk data structures associated with each file to store immediate provenance and an in-memory data structure that tracks file access.

This schema is efficient because it balances the time and space complexity of provenance tracking. Storing both immediate ancestor and immediate descendant pointers makes querying for the full set of *descendants* of a given file just as efficient as querying for the full set of ancestors. Because PEST does not store the entire provenance tree of a given file, we must determine it through a depth-first search. On the other hand, not storing the entire provenance tree with each file makes the overhead from storing provenance relatively small.

#### On-Disk Data Structures

To store immediate ancestor and descendant provenance information, PEST associates every file with a B-tree mapping part numbers to a pair of arrays: the ancestor array and the descendant array. These arrays are comprised of **(inumber, part, timestamp)** tuples that represent pointers to file versions in a versioning file system.

The B-tree is stored in a *provenance file* associated with the given *regular file*. Provenance files are stored in the /prov directory and are sequentially numbered for simplicity. The relationship between regular files, provenance files, and B-trees is shown in Figure 1. To forge a reference between regular files and provenance files, each regular file has an entry of the form **("provenance_file_pointer", inumber)** in its file system extended attributes, where the **inumber** field refers to the **inode** of the associated provenance file. Conversely, each provenance file as an entry of the form **("regular_file_pointer", inumber)**, where the **inumber** field refers to the **inode** of the associated regular file. This asymmetry allows for regular files and provenance files to be distinguished, and for provenance tracking to be disabled for provenance files.

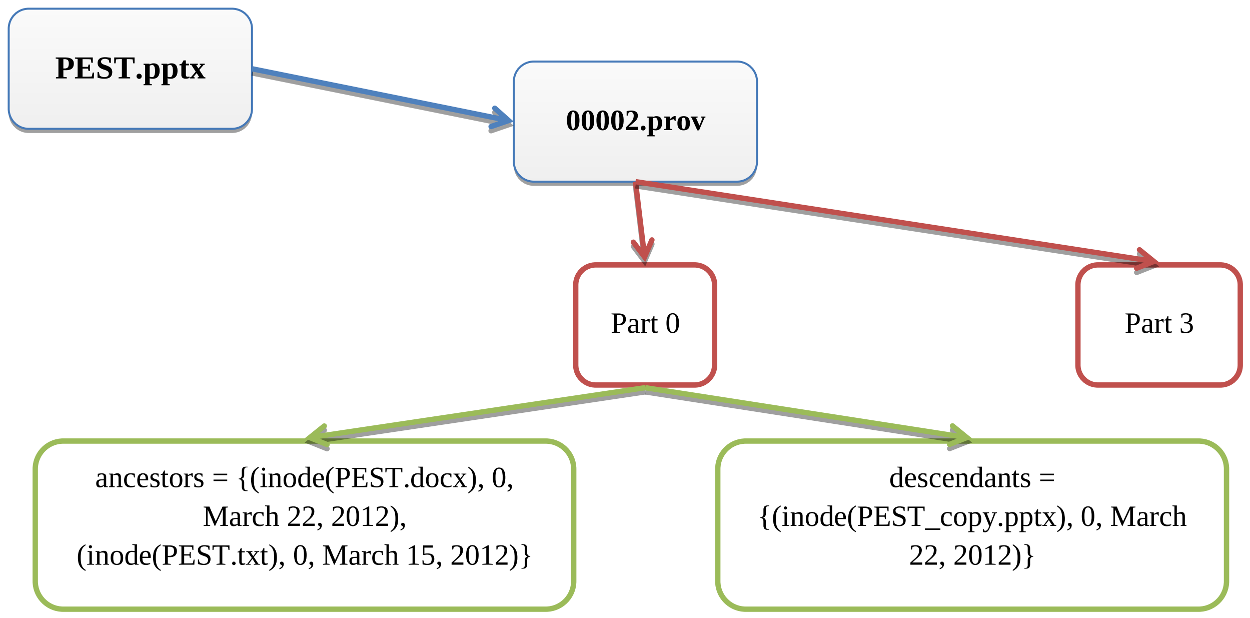
PEST.pptx
00002.prov
Part 0
Part 3
ancestors = {(inode(PEST.docx), 0, March 22, 2012), (inode(PEST.txt), 0, March 15, 2012)}
descendants = {(inode(PEST_copy.pptx), 0, March 22, 2012)}

*Figure 1. Regular files, provenance files, and the B-tree.*

One consideration in designing PEST was whether to use file paths or references to **inodes** when referring to ancestors and descendants. Storing file paths simplifies the **read_prov** operation, which returns a tree of all files that have directly or indirectly influenced the content of a given file. The advantage of referencing **inodes** via **inumbers**, however, is that **inodes** do not change when files are moved or renamed.

Another consideration was whether to use the file system extended attributes to store provenance data or to create a provenance file layer. The former approach requires that the file system extended attributes are dynamically-sized, which places a constraint on the file system. The latter approach has the added advantage of centralizing provenance information.

**In-Memory Data Structure**

To track file access and update provenance as necessary, PEST maintains an additional in-memory data structure called the *process-provenance tree (PPT)*: a B-tree mapping process IDs to *provenance objects*, each consisting of a bit flag **track_provenance** and an array **read_inodes** of **(inode, timestamp)** pairs**.** The structure of the PPT is illustrated in Figure 2.

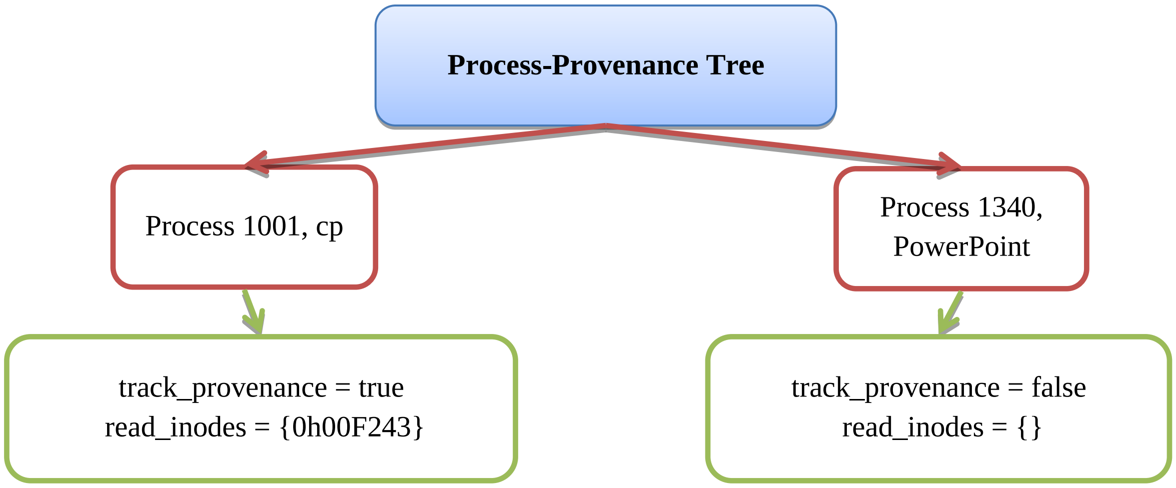

*Figure 2. Process-provenance tree.*

The **track_provenance** bit flag indicates whether or not provenance tracking is enabled for the process. A program such as PowerPoint might want to manage its own provenance and would set the flag to **false**.

The **read_inodes** array keeps track of files that have been input by a process. If a process writes to a file, the file's provenance will include the files that have been read by the process. The **timestamp** field tracks file version.

Notably, provenance updates are not explicitly cached in any way. Caching could be implemented at a lower level of abstraction and is left to the **write_prov** implementation.

When **fork** is called, the child process is assigned a clone of the parent's provenance object in the PPT, because the child process might make use of data that was read in by the parent process, and because the child process inherits open file handles.

PEST does not keep track of process-to-process communication via pipes because doing so sacrifices simplicity. Because PEST is extensible, however, implementing “pipe tracking” is relatively straightforward.

## API and Algorithms

To dynamically update these data structures with provenance information, PEST uses a modified version of the UNIX system API [1] as well as additional provenance query/update commands. The latter commands manipulate the on-disk data structures. Because they involve disk I/O, they are relatively expensive.

**write_prov(fd, part, provinfo)**
It updates the provenance information for **fd**.
The input **provinfo** is a tuple **(inumber_ancestor, part_ancestor, timestamp_ancestor)**. First, look up the provenance file associated with **fd**. Loop through the ancestor array stored in the B-tree entry for **part**, searching for **inumber_ancestor** and **part_ancestor.** If no entry exists, add **(inumber_ancestor, part_ancestor, timestamp_ancestor)** to the ancestor array. If there is an entry with an older **timestamp**, update **timestamp.** Otherwise, do nothing.
Similarly, update the descendant array of **inumber_ancestor** using **(fd, part, current_version_timestamp(fd))**.
If two versions of a file contribute to the provenance of a given file, only the latest version is stored in the ancestor or descendant array, reducing provenance overhead in the presence of circular references.

**read_ancestors(fd, part)**
It returns the ancestor array for **(fd, part)** by looking up **part** in the provenance file associated with **fd.** The ancestor array has entries of the form **(inumber, part, timestamp)**,

**read_descendants(fd, part)**
It returns the descendant array for **(fd, part)** by looking up **part** in the provenance file associated with **fd.**

**read_prov(fd, part)**
It generates the full ancestor tree for **(fd, part)** by performing a depth-first transversal starting from **(fd, part)** and returns an array with entries of the form **(inumber, part, timestamp)**.
The code in Figure 3 illustrates the depth-first transversal: repeatedly call **read_ancestor** until every file and part is visited, ignore older versions of files that have already been seen, and avoid circular references.

```
read_prov(fd, part) {
      set prov = read_ancestors(fd, part);
      set todo = prov;
      set done = {};
      while (next = prov.pop() && next is not (fd, part)) {
            add next to done;
            add read_ancestors(next) to prov, avoiding duplicates
                  unless a file has a later timestamp;
            add elements of read_ancestors(next) to todo, avoiding
                  files already in done;
      }
      return prov;
}
```

*Figure 3. Pseudocode for read_prov.*

**search_prov(fd, part)**

It generates the full descendant tree for **(fd, part)** using the same algorithm as in **read_prov**, but calling **read_descendant** instead of **read_ancestor**.

**inode_to_path(inumber)**
This is a convenience function for converting **inodes** generated by commands such as **read_prov** to file paths. It recursively searches through every directory in the file system until an **inode** with the given **inumber** is found and returns its **path**, which might not be unique for the **inode**.

**create_prov_file(fd)**
It creates an empty provenance file for **fd**. The B-tree should only have the key “0”, corresponding to the entire file, and the ancestor and descendant arrays should be empty.
This API call should only be called by **open**.

**delete_prov_file(fd, part)**
It deletes the provenance file associated with **fd**, while ensuring that the provenance graph is not broken. As the code in Figure 4 illustrates, **delete_prov_file** links all of **fd**’s immediate ancestors with its immediate descendants and does the same for immediate descendants. Finally, it deletes the provenance associated with **fd**.
This API call should only be called by **unlink**, when a file is being deleted.

```
delete_prov_file(fd, part) {
     for every file/part/version fd_an in read_ancestors(fd, part) {
          delete fd from the descendant array of fd_an;
          add read_descendants(fd, part) to the descendants array of
               fd_an;
     }
     for every file/part/version fd_de in read_descendants(fd, part) {
          delete fd from the ancestor array of fd_de;
          add read_ancestors(fd, part) to the ancestor array of
               fd_de;
     }
     set fd_pr to be the provenance file of fd;
     if fd is an entire file (part 0) {
          delete fd_pr;
     }    else, fd is a part {
          delete this part from the B-tree stored in fd_pr;
     }
}
```

*Figure 4. Pseudocode for delete_prov_file.*

The UNIX-like system calls update both the in-memory and on-disk data structures. The overhead from tracking provenance in this case is relatively cheap. The calls are aware of the **regular_file_pointer** key in the file system extended attributes, which indicates that a file is a provenance file, and do not track provenance for such files.

**open(name, flag)**
It proceeds as in the normal **open** system call. If a new file is created, it calls **create_prov_file(filep)**, where **filep** is the file pointer.

**read(filep, buffer, count)**
It reads in data from a file and adds the file to the PPT.
Look up the process ID of the process calling **read** in the PPT. Add the **(inumber, timestamp) pair** referenced by **filep** to **read_inodes** if it is not already present. Afterwards, proceed as in the normal **read** system call.

**write(filep, buffer, count)**
It writes to a file from a buffer, updating provenance data as well.
Look up the process ID of the process calling **write** in the PPT. Loop through the **(inumber, timestamp)** pairs contained in **read_inodes** and call **write_prov(filep, 0, (i, t))** for each one. This represents adding a file that was read earlier to the provenance of the file being currently written. Afterwards, proceed as in the normal **write** system call.

**fork()**
It creates a new process.
First, proceed as in the normal **fork** system call. Then, look up the process ID of the process calling **fork** in the PPT. Create a deep copy of the parent's provenance object and store it in the PPT under the child process ID.

**exit()**
It terminates the process, deleting the provenance object corresponding to the process ID of the process calling **exit** from the PPT. Afterwards, it proceeds as in the normal **exit** system call.

**unlink(name)**
It removes a hard link to a name and removes provenance information if necessary.
Proceed as in the normal **unlink** system call. If there are no links remaining to the file referenced by **name**, call **delete_prov_file(filep, 0)** to remove the provenance file associated with **filep**, where **filep = name_to_inode(name)**.

**set_provenance_flag(value)**
It sets **track_provenance** for the current process ID to **value**. If **value** is **false**, provenance tracking will be disabled, and vice versa.

## Analysis

### Use Cases

There are four main use cases that illustrate the extensibility of PEST: PowerPoint slide copying, compiling software, copying files, and handling tar/zip files.

#### PowerPoint Slide Copying

When a user copies PowerPoint slides from one presentation to another, PowerPoint makes a **write_prov** call for each slide (part) of a presentation that is sourced from another presentation. Furthermore, PowerPoint refers to the specific slide (part) of the ancestor presentation that is sourced. This way, PowerPoint tracks slide provenance across presentations.

**Compiling Software**
When a user compiles files, the compiled file's provenance will include the source files. Because the compiler must **read** all source files before writing the compiled file, they will be added to the process-provenance tree. When the compiled software is written, its provenance will be augmented with the list of source files. Any temporary files that are created and deleted by the compiler will automatically transfer their provenance to the compiled software via **delete_prov_file**.

**Copying files**
When a user uses **cp** to make a copy of a file **A**, naming it **B**, **B**'s provenance, initially blank, will consist of **A**. This is because **cp** calls **read** and **write**, which track provenance.

A consideration in designing the behavior of **cp** was whether or not **A**'s provenance is inherited. PEST does not create a copy of **B**'s provenance, which speeds up **cp** slightly and results in smaller provenance files.

**Handling tar/zip files**
When a user adds files to a ZIP archive, the ZIP program will track its own provenance and represent files in the archive as parts. Depending on third-party design decisions, it may choose to keep the original files' links or to make links between files in the ZIP archive internal when possible. While the former approach may not require any modification to the ZIP program, the latter approach requires keeping track of the **inodes** of files being added to the archive and changing provenance pointers to be internal when necessary. The flexibility afforded to the third-party highlights PEST's extensibility.

## Performance and Scalability

Due to design decisions such as only storing immediate links in the provenance graph, PEST is reasonably performant and scales well in terms of disk usage.

**Disk Usage**
Assume that the average file has 5 parts and 20 ancestor/descendant pointers per part. Assuming that timestamps use 4 bits, each direct ancestor and descendant takes up 4+8+4=16 bytes, accounting for the **inumber**, **part**, and **timestamp**. This means that the average file has 1600 bytes of provenance data. The B-tree overhead is at most 1600 additional bytes, meaning that provenance files have 3200 bytes of data. This amount of data fits into 7 512-byte blocks [1] on disk. The overhead from the provenance file's **inode** and the file system extended attributes is equivalent to about one additional block.

Thus, the provenance data of the average file takes up about 4KB. If we assume that the file system has 1,000,000 files, this is about 4GB of overhead.

**Copying Workflow**
Copying a file once introduces a provenance overhead, as a single provenance file must be created and a single write to the provenance file must be done. There are additional operations performed in memory, but these are on the order of microseconds. Assuming that we can read/write amounts of data less than 4KB in 12ms, the provenance overhead is 24ms. Assuming that it takes about 50ms to copy the file data itself and update the file system extended attributes, we can copy about 13 files per second while propagating provenance data. This is an underestimate.

**Provenance Tracking**
Calling **read_prov** on a file with 10 total ancestors would involve 20 **read**s of 4KB on average, each of which takes about 12ms. PEST must read both file system extended attributes and provenance files. Thus, **read_prov** would take 240ms, meaning that it can be called 4 times per second, which is acceptable.

## Conclusion

PEST offers sensible and extensible provenance tracking by storing data in *provenance files* associated with every regular file, in addition to using an in-memory data structure that tracks file access. It is applicable to common use cases including PowerPoint slide copying, compiling software, copying files, and handling tar/zip files.

PEST's strengths include portability and simplicity. Challenges that remain before it can be implemented include preventing hardware crashes, ensuring security, and tracking provenance resulting from pipe-based communication between processes. Future directions for work include augmenting the types of provenance that can be represented, expanding interoperability with non-PEST file systems, and tracking provenance across computers.

## Acknowledgements

We are grateful for Peter Szolovits' technical instruction, for the numerous writing comments received from Lydia Volaitis, and for the design feedback given by Katherine Fang.

## References

[1] D. M. Ritchie, K. Thompson, "The UNIX Time-Sharing System," in *Communications of the ACM*, vol. 17, no. 7, 1974.
[2] J. H. Saltzer and M. F. Kaashoek, M. Frans, Principles of Computer System Design. Waltham, Massachusetts: Morgan Kaufmann, 2009.

(Word count: 2,624, including references and headings. Sorry!)