A Versioning File System Based on UNIX V6

Ellen Finch
Rudolph T11, F1
March 1, 2013

Overview

The goal of this design project is to produce a space-efficient versioning file system based on the UNIX V6 file system. The system will save both an old and a new version of a file each time a file is saved; the old version will be read-only to prevent branching. Directories will not be versioned, because it is assumed that all important files will be versioned. The proposed system will provide the desired versioning functionality by copying a file's inode when it is opened and only modifying the blocks that are directly affected by a write. This system is space-efficient because versions will share most of the same memory blocks, and will only use additional space for the blocks that are changed.

Design Description

Modifications to Data Structures

To support the desired operations, additional metadata will be added to the inodes and to the file descriptors. The inodes will be extended to keep track of whether or not a file is to be versioned, a maximum lifetime for versions, and a timestamp. Each inode will also be given a `prev_inode` field which points to the inode of the immediately preceding version. The file descriptors will be extended with a `modified` flag that tracks whether or not an opened file has been written to.

Summary of Basic Operations

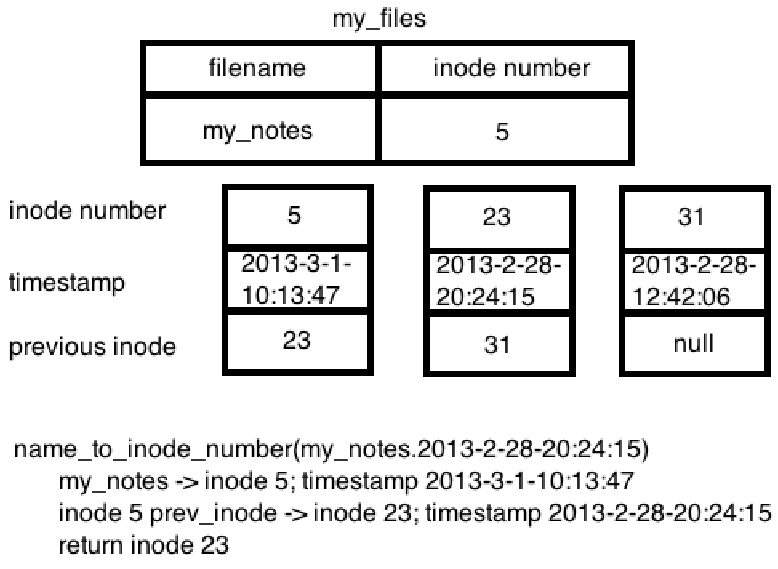
When the current version of a versioned file is opened, its inode information is copied to a new inode; the returned file descriptor refers to the original. The timestamp associated with the original inode is set to the current time, and its `prev_inode` field is set to the copy. The copy is not associated with any name, but it is referenced by the original inode and has a `refcnt` of 1. The copy is always created in read-only mode.

On a write call, the system copies each block which is to be modified into a new block, edits the contents of the new block, and updates the file to reference the new block instead of the old one. The write call also sets the `modified` flag in the file descriptor to true.

On close, if the `modified` flag is set, the system closes the file and does a garbage-collection check: it runs back through the `prev_inode` chain of versions and checks timestamps, deleting any that are older than the specified maximum lifetime. If the modified flag is not set, the system closes the open file, changes its `prev_inode` to the `prev_inode` of the copy, and deletes the copy.

Viewing Older Versions

The names of older versions are not explicitly stored in a directory. Instead, when the user calls `ls` to list the contents of a directory, the system will traverse the `prev_inode` list and list each older version as `filename.[timestamp]`. The user can then specify the version to open by this name, and the system will look through the `prev_inode` links from `filename` until it finds the appropriate timestamp, and will then return that inode.



```
name_to_inode_number(my_notes.2013-2-28-20:24:15)
    my_notes -> inode 5; timestamp 2013-3-1-10:13:47
    inode 5 prev_inode -> inode 23; timestamp 2013-2-28-20:24:15
    return inode 23
```

Figure 1. Older Version Lookup By Timestamp

Links

The implementation of symbolic links will be unmodified. Since only the current version of a file is stored by name, symbolic links will only allow linkage to the most recent version of a file.

The implementation of hard links relies on the `prev_inode` field. Since the inode number of the current version remains unchanged while the `prev_inode` field is updated, a single hard link to the current version inode gives access to all saved versions.

The unlink function will be essentially unchanged, with the addition that if any inode's `refcnt` is decremented to 0, the system should set the `refcnt` of its `prev_inode` to 0 as well, and so on until an inode with no previous inode is found.

Additional System Calls

In order to make the system more user-friendly, two new system calls will be added to support marking files as to-be-versioned or not and setting the maximum lifetime for a file's versions. There will also be a garbage-collect call added to delete expired versions of a file without having to open and close it. The current version of a file is never garbage collected.

Conclusion

The proposed design provides a space-efficient solution for a versioning file system by storing minimal additional data for each version of a file. The remaining concerns for this implementation are the specific implementation of the copy-on-write functionality, which involves heavy modification of the write system call, as well as the implementations of other modified or added system calls.

References

The description of the UNIX file system on which this versioning file system is built is from *Principles of Computer Design* by Jerome H. Saltzer and M. Frans Kaashoek. The design is indebted to the paper "Ext3cow: A Time-Shifting File System for Regulatory Compliance" by Zachary N.J. Peterson and Randall Burns, located at http://hssl.cs.jhu.edu/~zachary/papers/peterson-tos05.pdf. The design development was also aided by discussion with Eugenio Fortanely.