Constructing a Unix-Based Versioning File System

MIT 6.033 Design Project 1
Proposal
February 28th 2013
Xiao Meng Zhang

Overview

Current file systems are highly inefficient in terms of both space and productivity. Users often have to create duplicate copies of the same file saved under different names if they want to track previous versions of a file. I propose a Unix-based versioning file system that automatically versions a file upon close if modified, and tracks all versions of the file using a Version-Tree data structure. This system utilizes the copy-on-write protocol to record each modified block of a file, and constructs a version_log documenting the status of the file for each version. Under this design, users are able to review all versions of a file, exclude files from versioning, and revert to any previous version in a fast and space efficient manner.

Design Description

Opening and creating files

Construction of a new file begins with the creation of two inodes: *base_node* and *reg_node*. Reg_node behaves as a regular inode file, recording basic metadata about the file, as well as the numbers of blocks storing the file data. Base_node has a pointer to reg_node, and stores additional metadata about the file (whether it is versioned, the number of versions it has, and a link to the most recent version)

Base_node is the primary inode used to reference the file. When a user tries to open an existing file, the inode number stored in the directory of the file points to its base_node. From base_node, the user can use the link to reg_node to access basic metadata and the most updated version of the file, or the link to the Version-Tree to access previous versions.

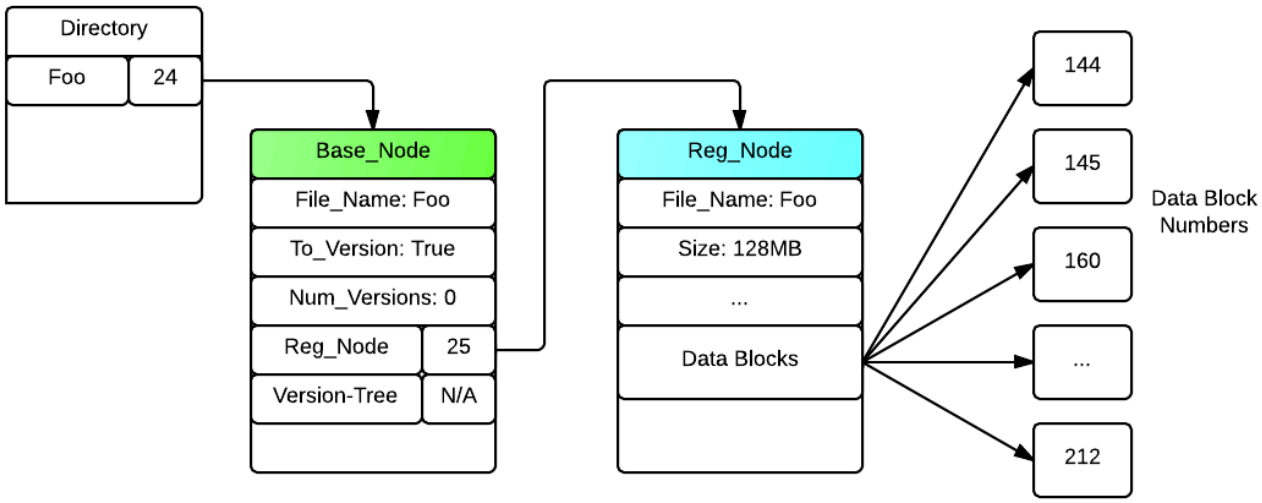

Figure 1: Relationship between Base_Node, Reg_Node, and Data_blocks.

Writing To A File

After opening (or creating) a file, users can begin writing to and deleting data from it. The file system will first check whether the opened file is set to be versioned using the To_Version field in base_node.

If the file is not set to be versioned, modifications made by the user will be done on the original block directly.

Otherwise, the system will adhere to a copy-on-write protocol with respect to the modified blocks. Under this protocol, every block that the user modifies will first be copied into a separate block. All modifications done by the user will be on the newly replicated block. The old one remains unchanged. Once the user finishes modifying the block, the link to the original block written on the Reg_node will be changed to the newly replicated one.

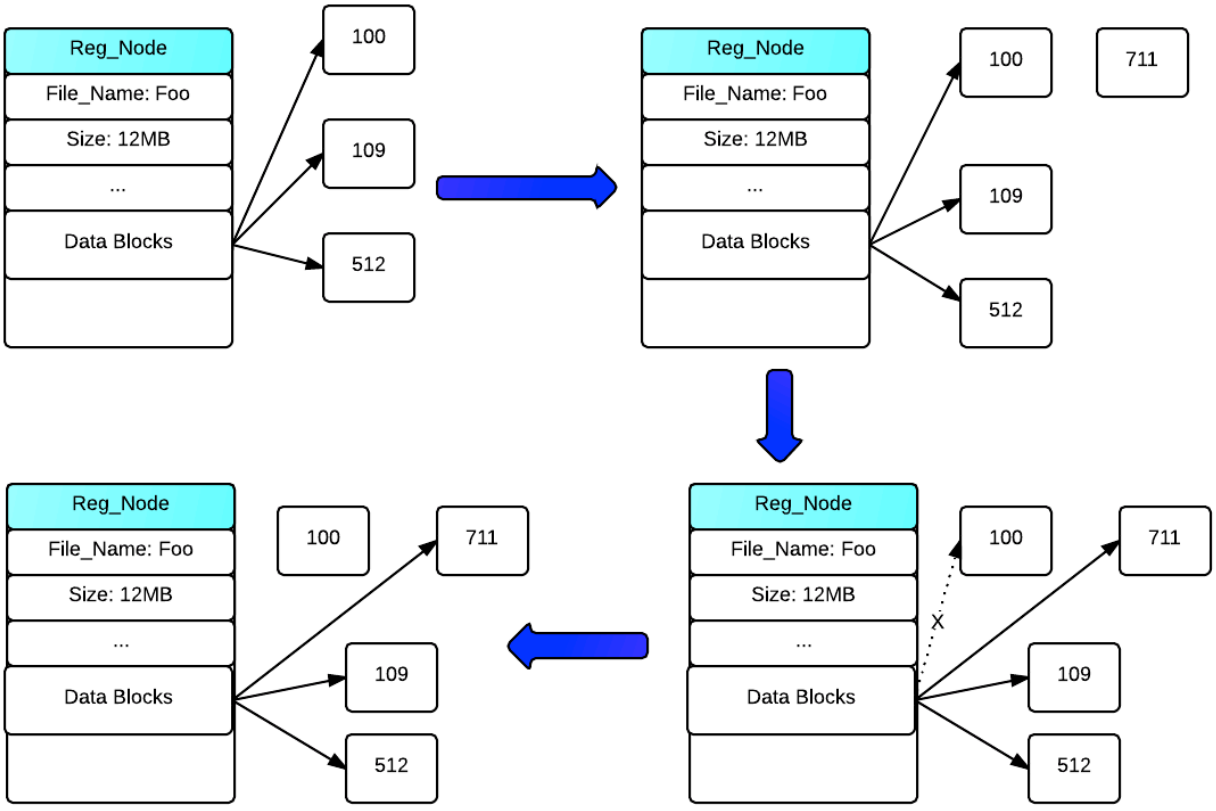

Figure 2: User starts modifying the data in block 100. Because the file is set to be versioned, the data in block 100 is first replicated to block 711. User's changes is then recorded on block 711. After the user finishes making changes to that block, the block number recorded on Reg_Node file is changed from 100 to 711.

Versioning the file

Versioning is done once the close system call is revoked on the file. Upon closing, a new version_log for that file will be created, documenting all the data blocks recorded on the Reg_Node file. This new version_log will also retain a pointer to the preceding version_log. Afterwards, the Version-Tree field in Base_Node will be updated to point to this new version_log. A file that has been versioned many times will have a chain of version_logs, with the head of the chain being referenced by the Version-Tree field in the Base_Node file. A diagram of this is shown in Figure 3.

This implementation is optimal because it allows the user to form links to any previous version of the file. In the case that the user wants a link to a previous version, all this file system has to do is visit the Base_Node for the file, traverse the chain of version_logs until it reaches the correct version, and return a pointer to the version_log for the specified version. Similarly, the user can easily revert to any previous version of the file.

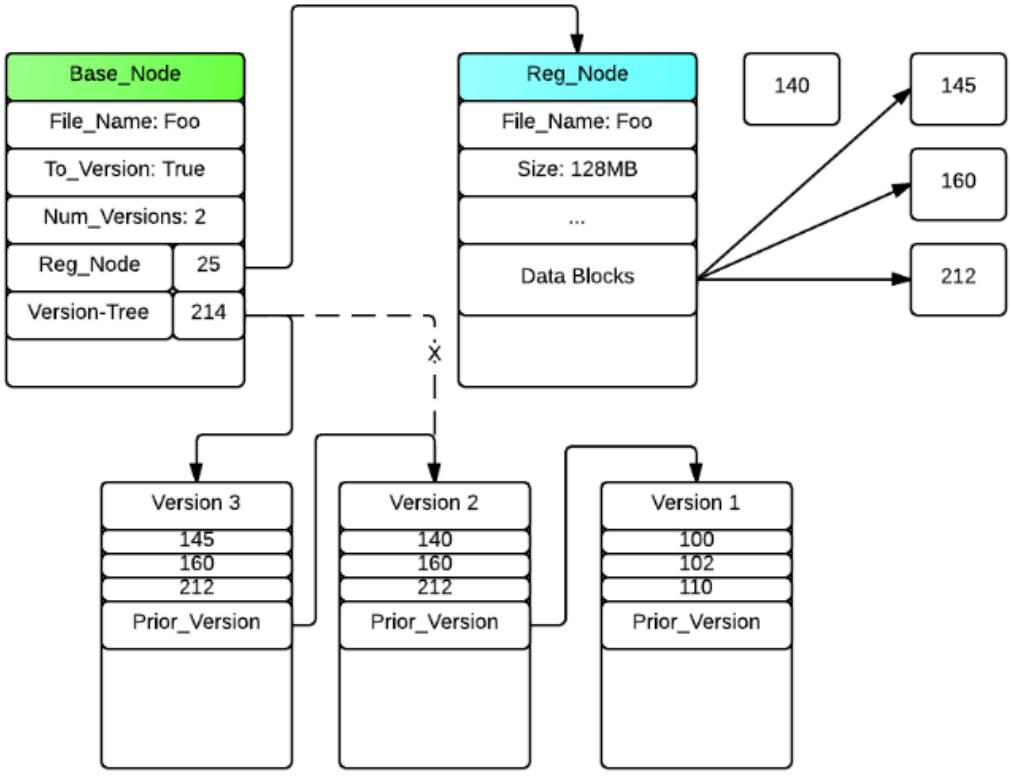

Figure 3: The user has revoked the close system call on the file after changing the data in block 140. This caused a new version_log called Version 3 to be created, listing all the data block numbers in the updated file. This new version_log holds a reference to the previous version_log (Version 2), and the Version-Tree reference in Base_Node is changed from version_log 2 to version_log 3.

Directories

Directories will be treated differently from regular files in the sense that they are unable to be versioned. However, this design will include a system call allowing users to set the remove_versioning field of a directory (in its Reg_Node), disabling versioning for all files created in that directory. An important issue to note is that directories do not have Base_Nodes, only Reg_Nodes.

Additional System Calls

The file system will also support the following system calls to users:

`SearchFile(File f, String str);`
Iterates through all versions of a file and searches for the specified input. Returns names of all versions in which the input has been found.

`QuitVersion(File f);`
Sets the To_Version field of the specified file to false. This stops versioning for the specified file.

`LinkFile(File f);`
Adds a link to the version_log of a specified file and its version in another file.

`RenameFile(File f, String new_name);`
Changes the name of a file in its directory.

Conclusion

The above design satisfies the requirements outlined in the problem statement by utilizing Version-Trees to track different versions of a file. This implementation allows for optimal space efficiency and supports additional functionalities such as renaming, linking, un-versioning, and searching across all versions of a file. Future questions that still remain are efficient garbage collection protocols and limiting the maximum number of versions a file can have.

References

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

Kasampalis, Sakis. “Copy on Write Based File Systems Performance Analysis and Implementation”. 11 January 2013.

Word count: 811