Versioning File System

Victoria Sun (*vsun@mit.edu*)
6.033 Design Project 1: Report
Hal Abelson, R01A, Tuesdays 10AM
March 22, 2013

Overview

This proposal outlines an implementation plan for a versioning file system (VFS) to automatically back up files and their revisions by storing different versions of a modified file. This system will mitigate consequences of accidental file deletion and loss from human and machine error, while being space and time efficient.

Traditional file systems lack support for maintaining a change history of a file. Once a file is modified and saved, the previous version is no longer accessible. This VFS automates the construction of a version hierarchy to allow users to access older versions of a stored file. It keeps track of differences between files and allows users to revert to a previous version of a file.

Versioning files will be implemented at the UNIX file system level in order to allow files to be versioned regardless of what application is using them, provided the appropriate write functions are called. In general, versioning occurs when files are written to disk: the new file is stored and a difference file containing the differences between the current and older version is created and stored.

We will not version directories; directory versioning typically exists to account for file recovery from file deletions and movement. That recovery is implicitly supported by maintaining file versions.

Design Description

Assumptions

To simplify much of this VFS design, we have made several key assumptions:

- The most up-to-date and current version is the most frequently accessed version of the file.
- The older a version of a file, the less likely the file is to be accessed.

Definitions

The **current version** of a file is the one most recently saved or written to disk.

A **difference file (diff)** is a file that contains the changes from a version to another.

Versioning a file involves opening the file and writing (or deleting) some amount of data from it.

High-level Implementation

This section will describe the high-level implementation of storing and maintaining versioned files.

In-Memory Data Structure

When `write` is executed, the system stores the binary difference in an in-memory buffer, whose size is specified by the user at file system creation. The open, modified version is compared with the current version stored in memory by a prior read call. Though not all of the file may exist in memory, it is reasonable to assume that the relevant portions will be, as it had to have been read in order to be modified. Each write execution overwrites the buffer with a new *diff*. Using an in-memory data structure instead of writing the difference file directly to disk allows us to save in physical memory space and time, allowing this system to be especially efficient for cases when small amounts of data is being written many times.

Inserting a New Version

A difference file is constructed and stored in the following situations: when the in-memory buffer is full, when the system call `close` is executed, and when the user explicitly calls for a new version to be made using the provided API. The following steps are executed:

1. The current version is updated on disk, with its existing inode (which we will refer to as inode A) pointing to it.
2. The difference file is either constructed with the binary difference tool or pulled from the memory buffer and is written to disk. A new inode (which we will refer to as inode B) is created and is denoted as a difference inode (with *isDiff* set to 1). B points to the location of the difference file on disk.
3. B is inserted into the sequence of version differences. Its *previous_version* pointer is updated to point to the inode A's *previous_version* points to.
4. Inode A's *previous_version* is updated to point to B.

Inode Structure Modifications

We will be augmenting the current inode structure with these following fields:

```
integer versionable

inode* previous_version

integer isDiff
```

Verisonable is an integer that represents whether or not this file will be versioned. All files are by default 1, versionable. Setting a directory's *versionable* bit effectively sets all files within the directory to be either versioned or not. Versioned files cannot be deleted. Deleting a versioned file would require one to set the *versionable* bit to 0 before proceeding.

Previous_version refers to the previous version of the file, if the file is versioned. If the file is not versionable or no previous version of the file exists (in the case of a new file), *previous_version* is a null pointer. *Previous_version* points to an inode that represents a file with the difference between the current version and the previous.

Integer *isDiff* indicates if the file is a difference file. A value of 1 indicates that it is a difference file; otherwise, it is set to 0. All files and directories are by default 0; *isDiff* is only set when a *diff file* is explicitly constructed.

Difference Files

The current, or latest version, of a file is represented much as a non-versioned file is; that is, the entire file is stored in the drive intact. Older versions of the file are represented as inodes indicated as a *diff file* and with a pointer that points to the next oldest version (or null, if none exist), as shown in figure 1. The current version is likely to be accessed most frequently and will incur no performance cost per access, with low performance access per write. Retrieving an older version of a file requires one to walk backwards in time and merge appropriate *diff files*, imposing cost which linearly grows as farther back in time we go; however, we anticipate this to be a less frequent use case.

Diff files are created and constructed using the binary difference tool `bindiff`, which specifies changes between individual bytes. This allows the system to not be limited by types of files to be versioned.

It is not necessary for users to access and view individual *diff files*. Thus, *diff files* cannot be linked through symlinks or hard links; only the latest version can be linked.

Garbage Collection

This VFS will merge versions of files to save storage space both automatically and manually. We arbitrarily decide that files will store at most 100 versions of itself; thus, files with greater than 100 diff files will begin to merge older file versions together, starting from the oldest. Garbage collection occurs daily, at an arbitrarily determined time.

Users can also manually call the provided `version_garbageCollect` to merge versioned files and lower storage space.

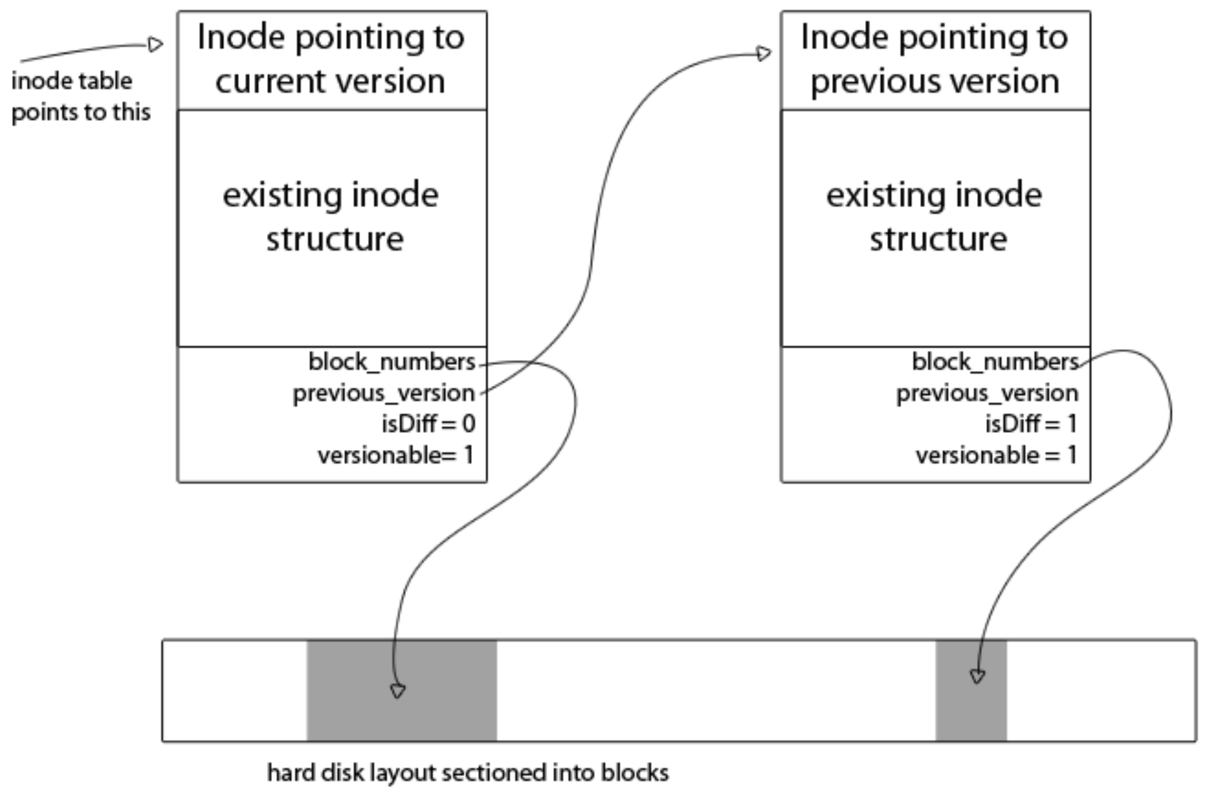

Figure 1: Each inode (n)'s *previous_version* pointer points to an inode that represents the earlier version (n-1). Earlier versions occupy less hard disk space because they contain only the difference between n and n-1. The inode table contains the reference to the current (n) version.

Application Programming Interface

We provide the following API:

```
int version_setVersionable(FILE *fd, int mode);
```

Sets the *versionable* integer in the inode. Values of 0 and 1 specify not versionable or versionable, respectively. Unversioning a previously versioned file includes setting the *previous_version* pointer to null. If the inode represents a directory, recursively apply `setVersionable` to all elements in it. Returns new value of *versionable*.

```
int version_getPreviousVersion(FILE *fd);
```

Reconstructs a read-only previous version of the file. Throws an exception if the file is not versionable. If the file is versionable but no earlier version exits, returns an error.

```
void version_mergeDiffs(FILE *id1, FILE *id2);
```

Merges the *diff files* between and including id1 and id2 for garbage collection. Ensures that the pointers pointing to id1 and from id2 are updated appropriately. Throws an exception if the inodes are not in the same version hierarchy.

`void version_delete(FILE *fd);`

Deletes the file, including all versions of it.

`void version_garbageCollect(FILE *fd, int numberOfVersions);`

Merges difference files starting from the oldest (or first) until only a numberOfVersions number of difference files exist. numberOfVersions is by default 100 if not specified.

`void version_constructNewVersion(FILE *fd);`

Forces the system to write to disk and construct a new version of the file, regardless of whether or not the file has been closed or the buffer has been filled.

`int version_listVersions(FILE *fd);`

Outputs a list of versions that the file system has stored for this particular file. Returns an error if the file is not versionable or is a difference file.

UNIX Modifications

This section will detail exact modifications we will apply to existing Unix system calls. We will not modify any procedure arguments so that existing system calls will still operate as normal.

`close(fd);`

On close, construct a difference file between the previous version (which exists in memory) and the current, or just closed, version. Write the current version and the difference to disk. Insert the previous version to the version hierarchy.

`write(fd, buf, n);`

Construct a difference file between the previous version (which exists in memory) and the open file. Store that difference to the in-memory buffer. If the buffer is full, write the version to disk.

`link(name, link_name);`

If the file is a difference file, this function returns an error.

`symlink(name, link_name);`

If the file is a difference file, this function returns an error.

```
unlink(name);
```

If the file is a difference file, this function returns an error.

Performance Analysis

In this section we will analyze the performance of this system using metrics such as storage requirements and time efficiency. We will refer to *n* as the number of versions that exist in the hierarchy, *v* as the v-th version we are trying to access, n_{size} as the size of the current version, and $diff_{size}$ as the size of the newly constructed difference file.

Storage Analysis

Naturally, this VFS requires more space than a non-VFS system, but it is greatly more space efficient than a system that makes complete copies for backups. This design uses space proportional to the number of changed blocks or $diff_{size}$. However, as this design consumes inodes per version created, it imposes space restrictions on the number of files this system can maintain. Modern day hard disks are capable of maintaining terabytes of storage, and so we believe that such an inode restriction will be nearly imperceptible to the user.

Time Analysis

We next estimate the expected performance of the modified read, write, and close operations. Most complex operations can be constructed with these two operations.

read

Reading the current version of a file incurs no extra performance cost, as the entire file is stored on disk. Reconstructing and reading older versions, however, take

$$t_{seek} * (n - v) + t_{read} * n_{size}$$

The overhead of read increases significantly as users access older versions. However, we believe that the likelihood for users to access older files decreases as the files get older. Furthermore, we suggest that users who want to frequently access a specific old file simply save another copy of it on the disk drive.

write

Write incurs negligible performance cost as long as the in-memory buffer size has been chosen appropriately. In general, write incurs only the small overhead of constructing a *diff* and then storing it into the in-memory buffer.

$$(t_{diff} + t_{memory}) * n_{size} + t_{memory} * diff_{size}$$

In the worst case scenario, the buffer size is chosen inappropriately small and each write is written to disk, which is approximately the same as closing the file multiple times. That analysis is located below in the *close* section.

close

Closing a file initiates the *diff* construction and insertion process. However, because the n-th version of the file exists in memory, we expect this to have only slightly more of performance hit than an unversioned save.

$$(t_{diff} + t_{memory}) * n_{size} + (n_{size} + diff_{size}) * t_{write}$$

Note that calling `version_constructNewVersion` incurs the same cost.

Workload Analysis

We next evaluate this system under three primary workloads: repeatedly writing to a small file, repeatedly writing a block to a large file, and searching through all versions of a small file. Figure 2 documents some useful average performance times that allow us to do numerical analysis.

Average Performance Times (ms)	
Read time: t_{read}	8.2
Write time: t_{write}	9.2
Seek time: t_{seek}	10
Binary difference construction time: t_{diff}	~0.1 (negligible)
Memory access time: t_{memory}	~0.1 (negligible)

Figure 2: System average performance times.

Repeatedly writing to a small file

Assuming that the buffer has been chosen intelligently, repeatedly writing to a small file will incur no extra overhead between writes. A more meaningful analysis discusses the worst-case scenario, where the size of a block exceeds the size of the in-memory buffer. A small file of minimum file size would require 1 block of data. Based on our previous equations, repeatedly writing 8 kb (or 1 block) of data would take ~0.1ms, an essentially nonexistent time cost.

We note that a system log file may be repeatedly written to and incur high performance cost; however, it is unlikely that previous versions of log files to be necessary for access and suggest users to manually exclude these undesirable files from versioning with `version_setVersionable`.

Repeatedly writing a block of a large file

A large file, of maximal file size, can be up to 250 blocks. Assume that we write 1 block of data to that repeatedly. If the buffer size is chosen appropriately, we will notice a marginal write time increase of 0.4 ms. Otherwise, if the block size exceeds the buffer, we will experience a time increase of 18.9 ms. This second scenario gives us a 100% increase in write time, which is unacceptably large in cases of repeated writes. However, on close, such an increase is expected. We expect that users will choose an appropriately sized in-memory buffer.

Searching through all versions of a small file

This VFS does not provide a search algorithm for users to use to efficiently search through all versions of a small file. Searching can be done manually in two ways, both of which are discussed below.

First, users can construct many versions of the file and search through it with `grep`. Such a behavior would incur large performance cost as each version must be constructed before searched.

This VFS's API provides methods such as `version_mergeDiffs` whose constructed merges can be searched through using grep, and allow for users to find strings that may be split amongst different versions. A proposed search algorithm using this merge method includes constructing simple, small merges that exceed the string length of the query. Such a design choice would incur significantly less cost than reconstructing each version.

Use Cases

In this section we will briefly discuss some common use cases keeping the above mentioned performance analysis in mind.

Writing a few blocks of a file

Users will frequently find themselves modifying small parts of a pre-existing file, large or not. The in-memory buffer allows us to, instead of incurring a writing to disk every write call, write when a sufficient amount of changes have been made.

Finding some string across all versions of a file

Locating a string across all versions of a file is simple to do. By calling on `version_mergeDiffs`, users can construct merged difference files that are greater than the length of the query and search for the existence of that query in those merges.

Excluding files and directories from versioning

It is not necessary for all files and directories to be versioned. They can be excluded from versioning with a simple call to `version_setVersionable`.

Constructing new files and directories

This command is indistinguishable in space, time, and efficiency from a non-VFS system.

Conclusion

This document proposes a VFS design that allows maintenance of a file history while adding minimal speed and storage overhead. The modified system allows for backwards compatibility with old files. Its detailed API gives experienced users large flexibility with the system, while being by default easy to use and non-intrusive, allowing even basic users to take advantage of its functionality without complaint.

Acknowledgments and References

I would like to thank Professors Frans Kaashoek and Dina Katabi for their initial input ideas. I would like to thank Alec Thompson and Jesse Stickgold-Sarah for challenging me with their thorough questions and observations.

D. M. Ritchie, K. Thompson, "The UNIX Time-Sharing System," in Communications of the ACM, vol. 17, no. 7, 1974.

J. H. Saltzer and M. F. Kaashoek, M. Frans, Principles of Computer System Design. Waltham, Massachusetts: Morgan Kaufmann, 2009.

Word Count: 2532