NestVFS: A File-Backed Versioning File System

Ben Weissmann
bsw@mit.edu
Recitation: R08A, T2 with Chlipala

March 21, 2013

Introduction

NestVFS is a versioning file system that stores information about previous version of files and directories in a simple, human-navigable, nested-directory structure. This architecture means that standard tools such as `ls` and `grep` can be used to browse and search versions.

NestVFS adds additional behavior to system calls that manipulate files. It causes these calls to maintain version files in the `/versions` directory (referred to as the "Top-Level Versions Direcory" or "TLVD") that track the changes to a file or directory. NestVFS is designed with Unix principles in mind: these system call changes are as minimal as possible, and a number of independent utilities are provided to perform tasks such as garbage collecting old files and restoring files to their previous state.

Design

Structure of the TLVD

For each i-node that represents a directory or regular file, the TLVD contains an i-node directory (or "ID") that stores previous versions of the i-node, as shown in Figure 1.

A new version of an i-node that represents a regular file is created the first time a given file descriptor is used in a `write` call. Subsequent writes using the same file descriptor use the same version of the i-node, but a new version is created if the file is closed and then opened again. For directories, a new version of an i-node that represents a directory is created each time a link is added to or removed from the directory. NestVFS does not track metadata changes, thus restoring a file to its previous state only restores the content of the file, not the permission bits or other associated data. This means that if a file changes ownership, it can be safely reverted to an earlier version without the current owner losing ownership.

For each version of an i-node, an ID contains a "Specific Version Directory" or "SVD". These SVDs are named after the nanosecond timestamp at the time the file was opened (the output of `date +%s%N`). For example, a a version of i-node 123 created at time 1361777404709973359 would be stored in `/versions/123/1361777404709973359`.

SVDs for Regular Files

For i-nodes that represent a regular file, each SVD contains one or more version files, which store the contents of a block that was modified as part of the version. These version files are named after the block and contain 512 bytes: the contents of the block when the file was opened.

As an example, if the file with i-node 123 was opened at time 1361777404709973359 and block 456 was changed from "abc" to "def", there would be a file at `/versions/123/1361777404709973359/456` containing "abc".

File deletion is handled in the same way as truncating a file to zero length: a new revision is created that contains a version file for each block in the file.

To restore a file to the previous version, NestVFS finds the most recent SVD and replaces the contents of each block referenced by a version file with the contents of that version file. The process is described in greater detail in the discussion of the `restore` utility below.

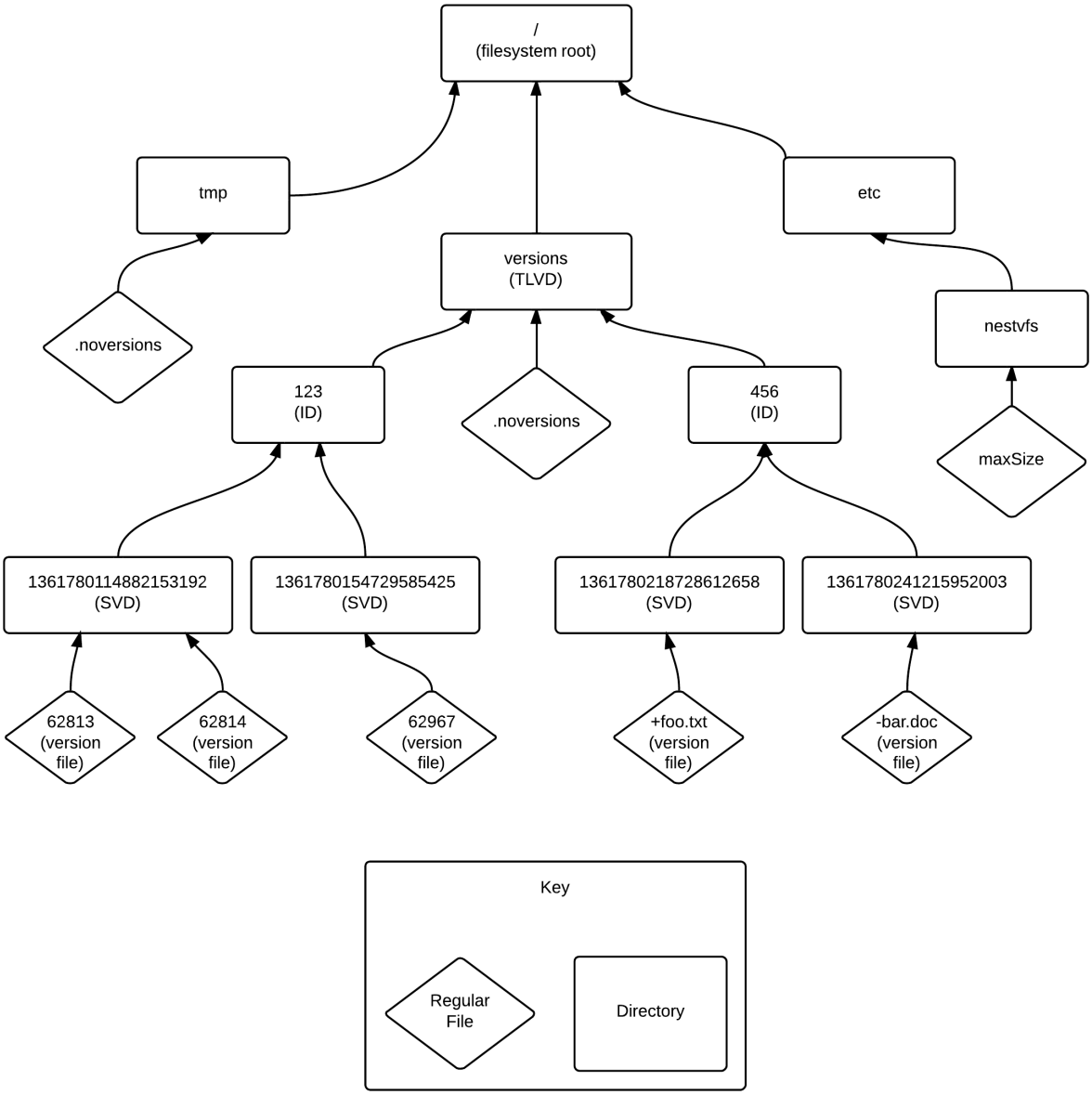

Figure 1: Structure of NestVFS, illustrating the nesting of version files inside specific version directories (SVDs) inside i-node directories (IDs) inside the top-level version directory (TLVD). Also shown are the configuration directory and `.noversion` files. i-node 123 corresponds to a regular file and i-node 456 corresponds to a directory.

SVDs for Directories

Each SVD for an i-node that represents a directory contains a file for each link that was added or removed during the version. If a link was added to the directory during this version, the file is named "+[name]", where [name] is the name of the link. If a link was removed from the directory, the file is named "-[name]". In either case, the file contains the i-node of the link. An exception is made for symlinks: instead of containing the inode the link points to, a version file for a symlink contains the path of the linked file.

To restore a directory to a previous version, NestVFS finds the most recent SVD. It then adds and removes links or symlinks based on the version files in the SVD.

Configuration

NestVFS can be configured to ignore particular files. There are two ways to ignore files: first, if a file named `.noversion` is present in a directory, no versions are kept for that directory or any files or subdirectories in that directory. `/tmp` and `/versions` for example, would contain a `.noversion` file. This checking is done when the file is opened, and is based on the name. Thus, if `/tmp/foo` is hardlinked as `/home/bsw/foo`, versions will be kept only if it is opened with `open("/home/bsw/foo")`, not if it is opened with `open("/tmp/foo")`. Files can also be excluded from versioning based on size: the file `/etc/nestvfs/maxSize` contains a single integer, which is the maximum size of a versioned file, in bytes. If a file exceeds this size, it is not versioned.

System Calls

NestVFS adds additional behavior to a number of system calls. If a system call is not discussed below, it is left unmodified.

open

The `open` system call must be altered to keep track of the time a file was opened, so the `write` system call can create a correctly-named SVD. NestVFS maintains an internal mapping of file descriptors to timestamps. When `open` is invoked, NestVFS generates a nanosecond timestamp and maps the file descriptor to this timestamp. NestVFS does not check whether the file is eligible for versioning at this point, because this process is expensive and is irrelevant for files that are never written. NestVFS performs this check during the `write` system call, described below.

write

The `write` system call must be altered to write version files. When a `write` call is issued, the file descriptor is looked up in the internal mapping to find the timestamp for the version and locate the SVD. If the SVD does not exist, NestVFS must determine if the file is eligible for versioning. If the file size indicated by the file's i-node exceeds the number of bytes specified by `/etc/nestvfs/maxSize`, NestVFS does not version the file. If the file passes this test, NestVFS examines the directory containing the file, then its parent directory, then its parent's parent, and continues until either one of these directories contains a `.noversion` file or it reaches the root directory. If it reaches the root directory, then NestVFS must version the file.

If NestVFS determines the file should be versioned, an SVD is created based on the timestamp associated with the file descriptor. If NestVFS determines it should not be versioned, an SVD is not created, and the file descriptor is removed from the internal mapping. On subsequent writes, if the mapping does not contain en entry for a file descriptor, NestVFS can immediately know that the file should not be versioned.

If NestVFS versions the file, then during the write, each time a new block is entered, the contents of the block are copied to a version file in the SVD.

close

The `close` system call removes the file descriptor's timestamp mapping, if it exists.

link

The `link` system call must update the ID for the directory containing the new link. First, NestVFS checks that the directory is eligible for versioning, as in `write`. If it is, then a new SVD is created based on the current timestamp, and a file named "+[link name]" is created, containing the i-node of the linked file.

unlink

The `unlink` system call is modified similarly to the `link` system call, but it adds a file named "-[link name]" to the SVD.

rename

The `rename` system call acts as a combination of `link` and `unlink`: The destination directory gets a new version representing the addition of a link, and the source directory gets a new version representing the removal of a link.

create

The `create` system call is modified to create a new ID and also perform the functions of `open` and `link`. If the parent directory is eligible for versioning, it creates an ID for the new i-node. It then creates a new version of the parent directory, representing the insertion of a link to the file, as with `link`. Finally, as with `open`, it inserts a new entry into the internal file descriptor mapping.

symlink

The `symlink` system call operates identically to the `link` system call, but creates a version file that contains the path the the linked file rather than its i-node.

Utilities

While many tasks (such as searching through past versions of a file) are easily accomplished using standard utilities such as `grep` and `find`, NestVFS provides a number of additional programs to help users manage and navigate versions.

ls-versions

The `ls-versions` utility lists past versions of a file or directory. To do this, it simply looks up the i-node and uses `ls` to list the ID of the i-node. It accepts a `-h` flag that causes it to print dates in a human-readable format.

Example: finding past versions of `myfile.txt`.

```
$ ls-versions -h myfile.txt
Tue Mar 19 04:44:49 EDT 2013
Mon Mar 18 22:12:43 EDT 2013
Wed Mar 13 05:10:23 EDT 2013
```

restore

The `restore` utility restores a file or directory to a previous version. It takes two parameters: a file or directory, and a time. The time is intelligently interpreted as either a nanosecond timestamp, a second timestamp, or as a human-readable date. The time need not match any version; the file will be restored the first revision before the given date. `restore` then opens the file and then iterates over all SVDs for the i-node of the given file whose timestamps are greater than the given timestamp, seeking to the appropriate block in the file and writing the contents of version files. Note that this creates a new version of the file. For directories, `restore` just creates and deletes links as specified by the version files.

Example: restoring myfile.txt to 1 hour ago.

```
$ restore myfile.txt $((`date +"%s"` - 3600))
Tue Mar 19 04:44:49 EDT 2013
Mon Mar 18 22:12:43 EDT 2013
Wed Mar 13 05:10:23 EDT 2013
```

backview

The `backview` utility prints an old version of a file to `stdout`. It is roughly the same as restore, except it reads the file into memory and modifies the blocks of its in-memory copy. This is less efficient than `restore`, because it needs to read the entire file into memory, but doesn't change the file. It can be used to pipe to other commands or to open a previous version for editing without creating a copy.

Example: diff myfile.txt with a previous version.

```
backview myfile.txt "Tue Mar 19 04:44:49 EDT 2013" | diff myfile.txt -
```

nestvfs-gc

The `nestvfs-gc` utility garbage-collects versions. It accepts as parameters a maximum amount of space a file's versions can take up, a maximum age of versions, and a minimum amount of time between versions.

First, the `nestvfs-gc` deletes old versions: it iterates over all IDs and deletes SVDs that are older than the maximum age. Second, it deletes versions that are too large: it iterates over all IDs and for each one, it calculates the size of its SVDs and deletes them, oldest first, until the total size is less than the maximum space allotted per file. Finally, it combines versions: for each ID, it iterates over all SVDs, oldest first. If

there are any SVDs that were created less than maximum-time-between-versions seconds after this SVD, their version files are copied into the older SVD and they are deleted.

`nestvfs-gc` is designed to be run as a cronjob at a convenient time. Example: crontab entry to garbage-collect at 3 AM every night, with a maximum age of 1 month, maximum space of 10 megabyte, and maximum time-between-versions of 10 minutes:

```
0 3 * * * /usr/bin/nestvfs-gc --age=1mo --space=10M --time=10m
```

Analysis

This system performs efficiently is a number of common use cases.

Use Cases

Repeated Writes to a Small File

A new version is created each time the file is re-opened. The SVD contains one file for each block changed during the version. The space needed is proportional to the total number of blocks changed across all writes. Opening the files is fast, but the write operation takes twice as long as normal: the old data must be copied before new data can be written.

Repeated Writes to a Single Block of a Large File

A new version is created each time the file is re-opened. Each SVD contains a single file that has no content (because we're appending to the end of the file, so there was no previous content). NestVFS adds almost no overhead: because we're appending, we don't need to copy old data to a version file, just create the file.

Searching Through All Versions of a Small File

Standard searching utilities (e.g. `grep`) can be used to search the SVDs for a desired string. If it is found, the name of the SVD corresponds to the most recent time at which the string was present in the file. Because `grep` is only searching changes, this is highly efficient.

Searching Through All Versions in a Filesystem

`grep` will find a string across all versions if the user points it at `/versions`.

Creating a New File in a Large Directory

A new SVD is created for the directory with a single very small file. This is both space and time efficient.

Overall Analysis

Efficiency

The design is, overall, efficient. It's particularly well-suited for search, as only changed portions of a file are searched in each version. In addition, garbage collection is very fast: compacting multiple version into one is a simple move operation – no reads or writes needed.

The major inefficiency is in restoring old versions: this requires a large number of reads to open each version file. This could be avoided by combining version files into larger files, but this would lose the garbage collection efficiency and searchability discussed above.

In addition, searching all of a directory's parents for a `.noversion` file is slow for deeply-nested directories. This could be resolved by added a cache that stores which directories are and aren't eligible for versioning, and expiring this cache when a `.noversion` file is created, deleted, or moved.

Application Integration

The design degrades gracefully for legacy applications: non-NestVFS-aware applications simply use the filesystem as normal. Creating NestVFS-aware applications is simple – a text editor, can, for example, use `ls-versions` to show versions of a file and `backview` to open these old versions. A file manager can do the same, and use `restore` to restore files and directories.

Conclusion

NestVFS maintains version files, which specify how to restore an i-node to a previous state, in a global versioning directory. NestVFS is both simple to understand and implement, and allows users to navigate previous versions of files using standard UNIX tools such as `ls` and `grep` while also providing a suite of custom utilities that leverage NestVFS's versioning capabilities. NestVFS differs from existing designs in the simplicity and minimalism of its core design, complemented by tools that integrate tightly with the standard Unix toolchain: searching handled by `grep`, garbage collection handled by `cron`, and difference viewing handled by `diff`.

During the implementation phase, careful attention must be paid to efficiency, particularly with respect to restoring old versions and searching for `.noversion` files. Solutions to potential problems have been discussed above and can be implemented as needed.

Overall, NestVFS provides an elegant design for a versioning file system that respects the Unix philosophy and toolchain, provides easy integration for application developers, and degrades gracefully for legacy applications.

Word Count: 2563

Collaboration: I designed NestVFS and wrote this document without help from anyone other than the course staff.