SEV: a **S**torage-**E**fficient **V**ersioning File System

Xunjie (Helen) Li
xunjieli@mit.edu
Design Proposal 1
Strauss T12
March 22, 2013

1 Introduction

Conventional file systems do not support automatic file versioning. To keep a record of file changes, users would need to manually save copies of a file at different point in time. This consumes a large amount of disk space, and brings users unnecessary operational overhead.

In this report, we outline the design of SEV -- a **s**torage-**e**fficient **v**ersioning file system, designed specifically to solve the above-mentioned problem. SEV augments file inode structure, implements an accompanying *vnode* for each versioned file, and uses an in-memory copy-on-write scheme. Retrieving old versions does not involve any sort of reconstruction, and is as fast as reading files in an ordinary file system. In addition, creating new user contents through standard file system calls can be performed efficiently without compromising the integrity of versioned data.

In order to keep the design as lightweight as possible, and to achieve reasonable speed and space efficiency at the same time, we have made a few design trade-offs: (1) We choose not to implement directory versioning. Even though it is a very desirable feature in some circumstances, there are many tricky edge cases involved. (2) In preference to B-trees, indirection is used to keep track of on-disk file blocks. This is to adhere more closely to the basic UNIX File System. However, it is relatively straightforward to adapt SEV to the implementation of B-trees, if a need for quicker random access to file blocks arises. (3) all old versions will be read-only. The reason is that in SEV, versions can reference to the same file blocks. To avoid massively rewiring during WRITEs, the design does not allow modification on old versions.

2 Design

The main components of SEV include an on-disk data structure, *vnode* (“version node”), and an in-memory data structure, *cblock* (“cloned block”). Regular file inode and the in-memory file_table are augmented. In addition, additional mappings are maintained in order to achieve fast lookup and access.

2.1 Data Structures

2.1.1 On-Disk Data Structures

Every file inode contains a *version-enabled* bit, specifying whether the file is versioned by SEV. It also contains a pointer to the *vnode* that records the version data of this file.

A *vnode* is stored in the same way as an inode, and is referenced by an inode number. The link count of an initialized *vnode* is one. When the all references to the actual file inode is removed, the link count of *vnode* drops to zero.

The data blocks of a *vnode* contain the following structured data:

- ***metadata_table:*** a table that keeps track of metadata on the versioning status of the file. It includes
    - ***file_inode_pointer***: a pointer to the actual file inode,
    - ***cur_num_versions***: current number of versions,
    - ***cur_total_blocks***: current number of total file blocks used,
    - ***last_commit_timestamp***: timestamp of last commit.
- ***block_usage_map***: a block usage map that contains block numbers used in the versioning of this file, and how many versions are referencing to each. This map includes indirect blocks and data blocks.
- ***version_object_list***: a doubly linked list that contains version objects, which are snapshots of file inodes taken at the time of version creation. In addition, two pointers, one to the oldest version object and the other to the newest version object, are maintained.

The structure of the *vnode* and the relationship between an augmented file inode and its accompanying *vnode* are illustrated in Figure 1.

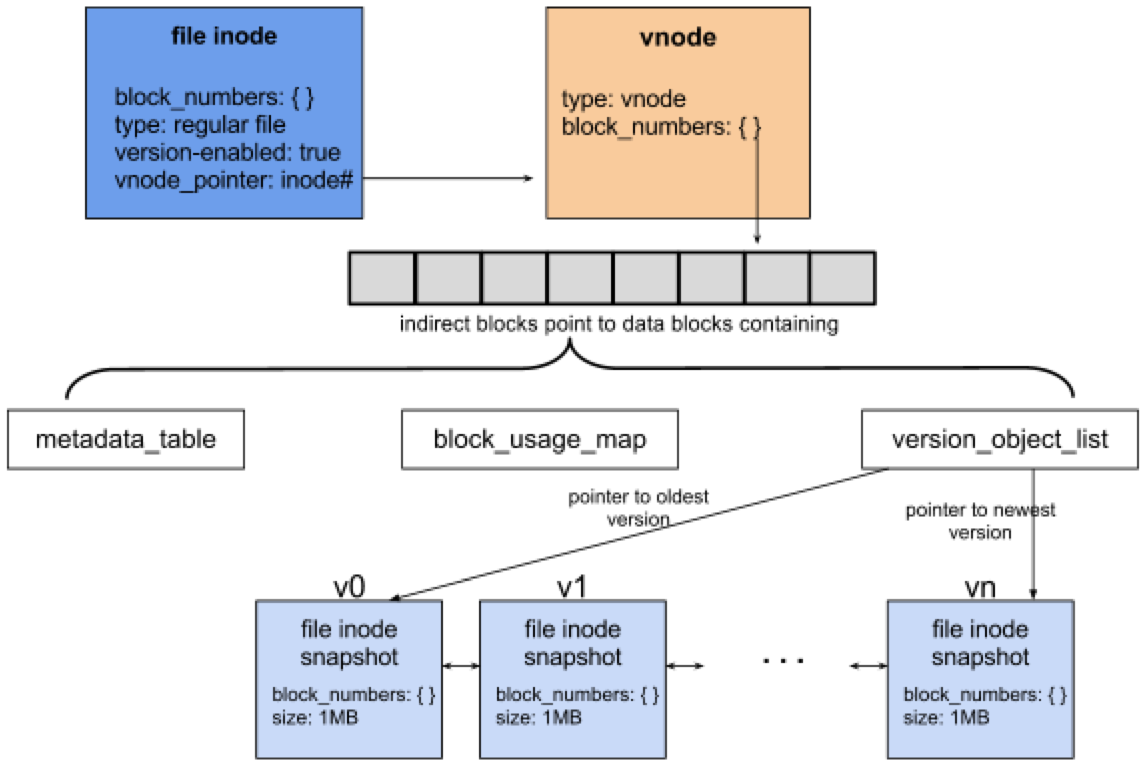

Figure 1. Augmented file inode and the structure of a vnode

Note that each *vnode* only has an inode number. It is otherwise unnamed. Even though a file inode can have multiple links, version objects are bound to the *vnode* of the file, independent of which name (hard link) is used by processes.

2.1.2 In-Memory Data Structures

cblock

In SEV, all WRITE and READ system calls operate on in-memory block caches. When a process attempts to modify an in-memory file block that belongs to a versioned file, an in-memory clone of that block, called *cblock* (stands for "cloned block"), is created. Instead of directly modifying the file block, the process will modify the corresponding *cblock*.

To enable fast-lookups, mappings from processes to *cblock* numbers are also maintained in memory.

file_table

SEV augments the in-memory file_table to contain a counter, ***num_of_writes***, which records how many WRITEs a process and its children have made to a file.

2.2 Versioning Policy

A system-wide SEV configuration file is stored in the root directory. This configuration file is a formatted text file containing,

- ***max_num_ver***: maximum number of versions allowed. Initial value is 50.
- ***max_num_block***: maximum number of file blocks allowed for a versioned file. Initial value is 2^{14}.
- ***num_write_calls***: the number of WRITEs after which a new version is created. Initial value is 10.
- ***lines_of_regular_expressions***: All files are versioned by default. However, if a file is matched by any line of regular expressions at its creation time, the inode of the file has *version_enabled* bit set to false. Similarly, if a directory path is matched, all files in that directory are opted out of versioning.

Root users can customize this configuration file according to their needs through a few SEV utility functions.

2.3 Versioning Procedures

SEV versioning procedures consist of 4 stages: (1) file creation, (2) copy-on-write, (3) version creation, and (4) file deletion.

As illustrated in Figure 2, Stage 1 and 4 only occur once in the lifetime of a file, and Stage 2 followed by Stage 3 can repeat many times.

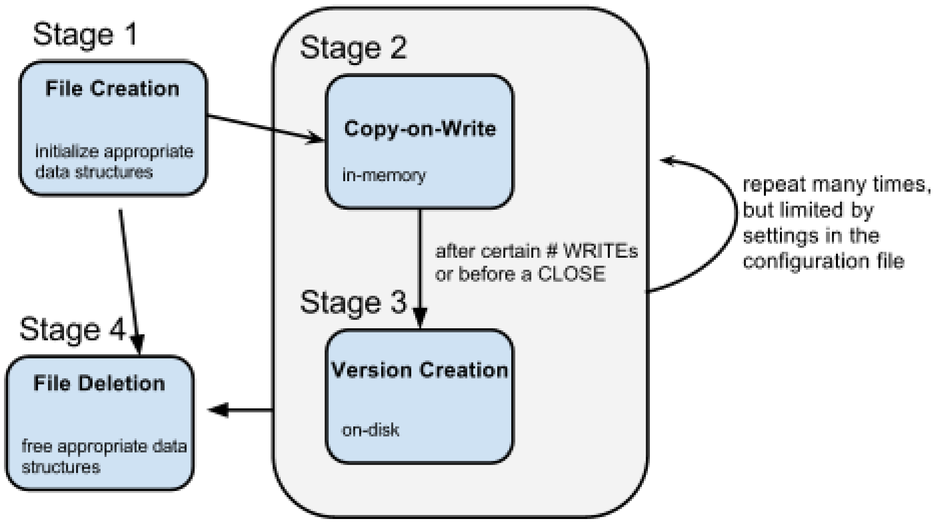
Stage 1
File Creation
initialize appropriate data structures
Stage 2
Copy-on-Write
in-memory
after certain # WRITEs or before a CLOSE
Stage 3
Version Creation
on-disk
repeat many times, but limited by settings in the configuration file
Stage 4
File Deletion
free appropriate data structures

Figure 2. Stages of versioning procedures

Stage 1: File Creation

When a file is created, the *version_enabled* bit is set according to the configuration file. An inode is allocated to be the accompanying *vnode*, if the *version_enabled* bit is true. A user is unable to change the *version_enabled* bit afterwards.

Stage 2: Copy-on-Write Scheme

For each process that tries to write to an in-memory file block, a *cblock* is created. Subsequently the process will execute READ and WRITE calls on the assigned *cblocks*. If new data cannot be contained in a *cblock*, ordinary in-memory blocks are allocated. After **num_write_calls** WRITEs, or before a CLOSE on files that have *cblocks*, Stage 3 is invoked.

Stage 3: Creation of a New Version

A new version is created by

- i taking a snapshot of the current file inode to make a new version object,
- ii updating **metadata_table, block_usage_map** and **version_object_list** in *vnode*,
- iii allocating new disk blocks to write each *cblock*, and any additional in-memory blocks back to disk,
- iv updating **block_numbers** in file inode to replace block numbers of the overwritten blocks with block numbers of the new disk blocks in (iii). If an overwritten block is a data block listed by an indirect block, the pointer in the indirect block is also updated. This is a recursive process.

This stage is illustrated in Figure 3.

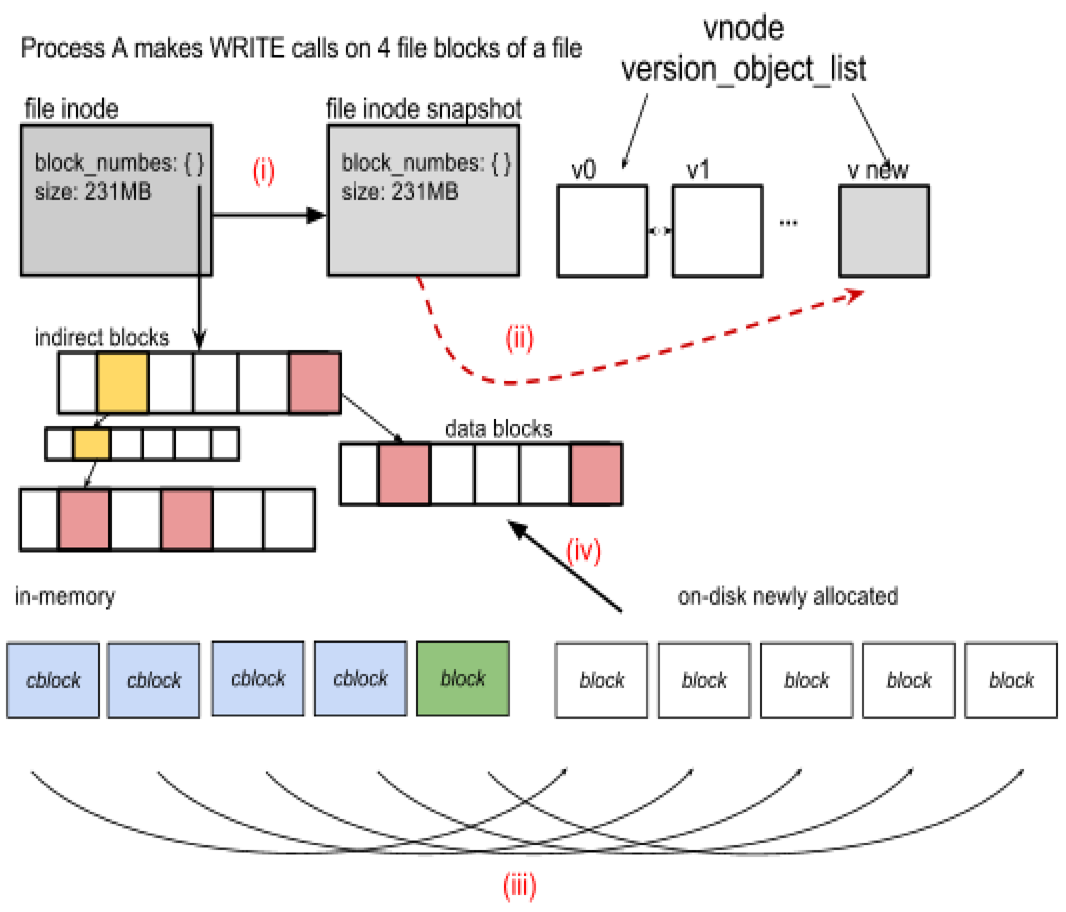

Figure 3. Creation of a new version (The green block represents an additional in-memory block used to contain new data. The pointers to red data blocks are replaced with the pointers to the newly allocated blocks. In the process, two indirect blocks, illustrated in yellow, are written with updated references.)

Stage 4: File Deletion

When a file is deleted by an application, SEV recursively frees all data blocks used by all versions by going through **block_usage_map**. It also frees the *vnode* after all data blocks are freed.

2.4 Garbage Collection

SEV periodically reclaims versions of file to restore disk space. It deletes the oldest version of a file, if the version data of that file exceeds the limits set in the configuration file (Section 2.2).

In deleting an oldest version, the following is handled by the garbage collection routine:

- updating reference counts in **block_usage_map**. If a reference count drops to 0, the block is freed.

- removing the corresponding version object from **version_object_list**, and updating its end pointer to point to the next version.

2.5 Application Programming Interface (API)

SEV supports all standard file system calls of Unix V6. There are a few modifications and additions.

Modifications:

- **WRITE** is intercepted to implement the copy-on-write scheme. In addition, WRITE is also modified to read the **num_of_writes** stored in **file_table** to determine whether it should invoke the creation of a new version.
- **READ** is intercepted to read old versions, if the filename is of the format of a version (i.e. name followed by "_v", a version number and the file extension). Reading an old version is straightforward, which only involves finding the correct inode snapshot in **version_object_list**, and reading the **block_numbers** in that snapshot.
- **CLOSE** is intercepted to execute creation of a new version.
- **STAT** is intercepted to parse a filename. If the filename specifies a file version, STAT finds the corresponding snapshot in **version_object_list**, and prints out the metadata in the snapshot inode.

Additions:

- **SEV_GREP**(*regexPattern, filename*) searches a file or a specific version of it (if *filename* specifies a version) for strings matching the regular expression *regexPattern*.
- **SEV_GREP_ALL**(*regexPattern, filename*) searches all versions of a file for strings matching the regular expression *regexPattern*.
- **SEV_STATUS**(*filename*) displays metadata stored in metadata_table of the *vnode* of the file with filename *filename*.

SEV_GREP and SEV_GREP_ALL employ SEV's READ system call and the GREP utility of Unix file system.

3 Analysis

The following sections present several use cases and performance metrics to determine how well SEV performs.

3.1 Uses Cases

File Creation

When a file is created, we need to parse the configuration file to determine whether the file should be versioned. If we regard every regex rule is of the same complexity, each file creation call incurs a delay that is linear to the number of regex rules in the configuration file.

A possible mitigation to the increased latency of creation calls could be to cache configuration settings in memory. However, this is not currently implemented.

Version Retrieving and String Searching

Each version is stored as a directly readable format in *vnode*. Retrieving an version only takes an additional amount of time that is bounded by **max_num_ver** in the configuration setting, as SEV needs to walk the doubly linked list (**version_object_list**) to find the version object.

Usually **max_num_ver** is small, ~O(100). Therefore, reading a version is as fast as reading a file in a non-versioning file system.

For string searching in old versions, SEV also performs well, as version retrieving is just as efficient as retrieving an ordinary file.

Frequent Commits

Some applications, such as a text-editor, might make frequent WRITEs. In order to avoid creating numerous new versions in a short period of time, SEV delays the creation of a new version after **num_write_calls** WRITEs or until a CLOSE (whichever happens earlier).

However, this design is not optimal, as **num_write_calls** is a system-wide setting. Some applications might need to create a version after a single WRITE, while others do not. A more granular approach is to customize settings on an application-specific level rather than on a system level.

Other Edge Cases

SEV does not optimize for WRITEs that only insert small amount of new data in a file block. Data with offset larger than the inserted data might all be shifted and re-written. SEV relies on the application layer to do such optimization.

An alternative design is to do byte-level diffing on new data and old data, and to store byte-level changes on disk. However, this makes the storage of the file system too fragmented, which might affect garbage collection negatively. Therefore, in SEV, a block is considered as the smallest unit.

3.2 Performance Analysis on Different Workloads

The performance of SEV is evaluated on three different workloads: (1) repeatedly writing to a small file; (2) repeatedly writing a block of a large file; and (3) searching through all versions of a small file.

The following assumptions are made:

- size of a file block: 4 KB
- size of a small file ~ 32 KB = 8 file blocks
- size of a large file ~ 2 TiB = 2^{29} file blocks
- number of versions made ~ n
- system-wide configuration,
  - **num_write_calls** = 10
  - **max_num_block** = 2^{30}

Table 1. Performance of SEV under different workloads

workloads	disk blocks read	on-disk storage	in-memory space
(1)	8 for every 10 WRITEs	O(8n x 4) = O(32n) KB	8 in-memory block, 8 *cblocks*
(2)	1 for every 10 WRITEs	O(2^{19} + 4n) KB	1 in-memory block, 1 *cblock*
(3)	n x 8 = 8n	worst-case*: O(8n*4) = O(32n) KB	worst-case*: O(8n in-memory blocks)

*For the third workload, the on-disk storage and in-memory space depend on whether and how the application stores the query results.

It is clear from the calculation that the amount of storage and space needed is proportional to the amount of changes (in terms of the number of modified blocks). Small edits on large files are less costly in version creation. Search time on versions is linear to the number of versions searched, and is as efficient as searching ordinary files.

3.3 Scalability Limits

SEV does have some scalability limits. Even though the additional time spent in retrieving a specific version is negligible, the time required by garbage collection increases linearly with the number and the size of versions. Therefore, when deleting versions of enormous sizes, SEV might experience temporary hiccups. In addition, versioning depletes disk space at a faster rate, which places a limit on the extensiveness of version control the user could do.

4 Conclusion

SEV provides a storage-efficient way to create and keep track of different versions of files. It does so by employing a copy-on-write scheme, minimally augmenting existing data structures, and creating an additional data structure (*vnode*) to systematically store snapshots of file inodes. With handy utilities and standard file system calls in place, SEV presents users and applications an easy interface to interact with versioned data and to customize version control settings.

Before the design of SEV is final, challenges that remain include handling hardware crashes, ensuring better access control of versioned data, and supporting version control settings with finer granularity.

5 Acknowledgement

I am grateful for the numerous writing comments received from Janis Melvold, for the discussion with my classmates (Siyao Xu, Kelly Zhang and Kojo Acquah), and for the extremely helpful design feedback given by my TA, Jorge Simosa.

6 References

[1] Z.N.J. Peterson and R. Burns. *Ext3cow: A Time-Shifting File System for Regulatory Compliance*. In: ACM Transactions on Storage, 1(2), May, 2005
[2] Saltzer, J. H., and Frans Kaashoek. *Principles of Computer System Design: An Introduction*. Burlington, MA: Morgan Kaufmann, 2009. Print.

Word Count: 2419