Luxor

A Performant Versioning Filesystem

John Wang
6.033 Computer Systems Engineering
Design Project 1 Report

Introduction

Luxor is a versioning filesystem which merges performance with a simple command line interface. Based on chunking and deduplication, Luxor enables frequent automatic versioning with little extra disk space. The system breaks files down into chunks and reuses each chunk, identifying them with a SHA-3 hash. Luxor uses version files and chunk-storage blocks to enforce two layers of abstraction. These abstractions allow Luxor to provide simple commands to the user while retaining performance. This paper overviews the design of Luxor and analyzes its performance.

Interface

For simplicity, Luxor only implements a small set of commands. The following is a description of Luxor's user commands:

- "luxor show"
    - Example: ***luxor show -f /home/john/my_file.txt -t '9/15/1995 5:34pm EST'***
    - Shows a particular file at a specified date in the past. The resulting file is read-only.
- "luxor grep"
    - Example: ***luxor grep 'some search string' -f /home/john/my_file.txt***
    - Searches for a string in history, can optionally specify a file.
- "luxor exclude" / "luxor include"
    - Example: ***luxor exclude /home/john/large_database_file.sqlite***
    - Excludes or includes a file or directory for versioning.

Design Description

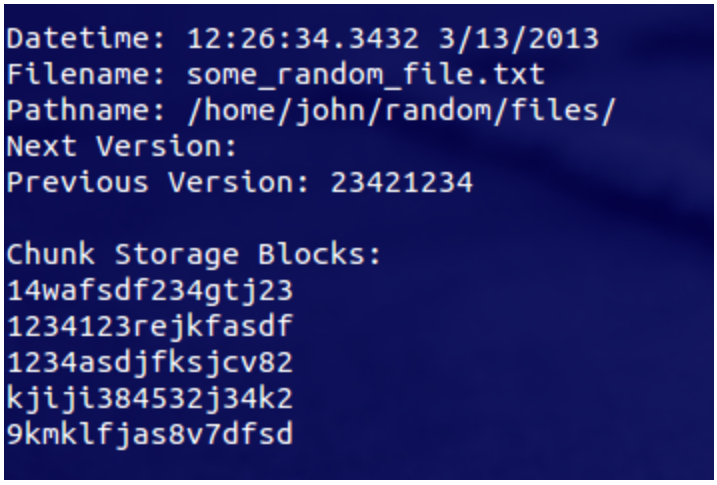
In order to accomplish the dual goals of usability and performance, Luxor adopts an abstraction model which hides the complexity of the versioning system from the user.

Versioning Abstraction

Version files are the fundamental medium for storing history. Each file's inode contains a reference to its most recent version file called "most_recent_version", which is the first element in a doubly linked list of version files. Each version file contains the following header information:

- Datetime of Version
- Filename and path
- Pointers to the next and previous versions of the file

In addition, version files contain references to storage-chunk blocks (explained in the next section), which are used to reconstruct the file. Large files will contain multiple version files to store a sufficient number of storage-chunk block references.



The above figure provides an example of a version file in human readable format (real version files would be byte-encoded). The figure shows the header information of the version file and the chunk storage blocks that it references.

Version Chunks and Deduplication

Luxor stores new versions efficiently by using deduplication to save only one copy of each unique chunk of a file. Chunks are k=256 byte partitions of a file, each having its own 32 bit SHA-3 hash. Chunks can be reached by providing a block number and a chunk offset. Each chunk-storage block stores 32 SHA-3 hashes of its chunks, along with a reference count for each chunk. As soon as the reference count for a chunk reaches zero, the space for that chunk is freed.

Inside the chunk-storage block, the hashes are stored sequentially, as the chunks would appear in the original file. For 32 bit SHA-3 hashes and a filesystem with 8kB blocks, each chunk-storage block will contain hashes for roughly 2000 chunks. Every chunk-storage block also contains a reference count. As soon as the reference count reaches zero, the chunk-storage block is deallocated.

CNode Index

The CNode index acts like an inode index for chunks. The CNode index contains a mapping from SHA-3 hashes to a block number and chunk offset pair. The root of the index is stored in a well

known location so that it can be accessed from anywhere.

When a new chunk is created, its information is written to the CNode index. Chunks are retrieved by using the chunk's SHA-3 hash to find the block number and chunk offset, then reading the 256 bytes in the file after the chunk offset.

Below is figure of a block containing chunks. Each block contains about 30 chunks each. As discussed above, chunks can be obtained by provided a block number and a chunk offset. To access chunk 3 in the figure below, the operating system would read the chunk with offset of 3.

Creating New Versions

The ***new_version(file, datetime)*** command is called by Luxor to create new versions. This command creates a new version file and updates the file's inode with a reference to this version file. The new version file is then added to the front of the linked list of versions. The ***new_version*** command also reads the specified file and breaks up the output into chunks of k=256 bytes, which are pushed onto a thread-safe queue.

A different thread pops chunks from the queue, hashes them to a 32-bit address using SHA-3, and performs a lookup on the CNode index for the resulting hash. If the CNode index does not contain the hash, then the ***create_chunk(hash, chunk)*** command is called. Finally, the SHA-3 hash is written to a chunk-storage block. If the block is full, a new one is created and a reference is inserted in the version file. If the version file is full, a new version file is created and a reference to the new version file is created.

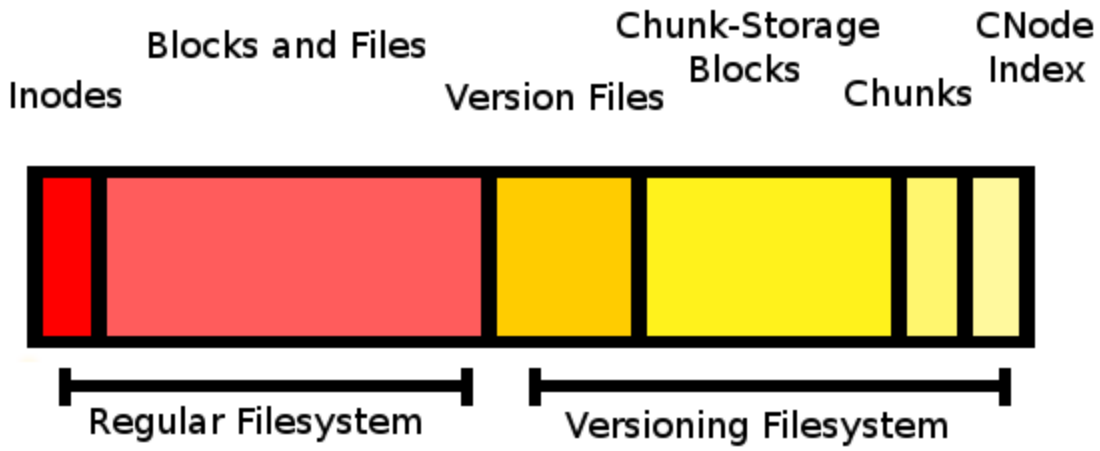
The ***create_chunk*** command finds the next chunk-storage block with available space, writes the specified chunk to that block, then creates an entry in the CNode index using the block number and chunk offset of the filled up chunk. The new CNode entry will also contain the hash argument from ***create_chunk.***

To improve performance, an in-memory cache keeps valid SHA-3 hashes, with a LRU replacement policy. This saves the time needed to lookup a SHA-3 hash in the CNode index if it already exists. Chunks are deallocated with the ***delete_chunk*** command, which removes chunks from the CNode index and the chunk's SHA-3 hash from the in-memory cache.

To improve disk usage, chunk-storage blocks are reused. Version files will reference the previous chunk storage blocks for all blocks which have not been modified. Only chunk storage blocks which have been modified will be rewritten in the version file.

Layout on Disk

The figure below shows how the storage on disk will be divided among the regular filesystem (in red) and the versioning filesystem (in yellow). Version files, chunk-storage blocks, and chunks will each receive their own continugous space on disk. The CNode index resides in a well-known location at the end of the disk.



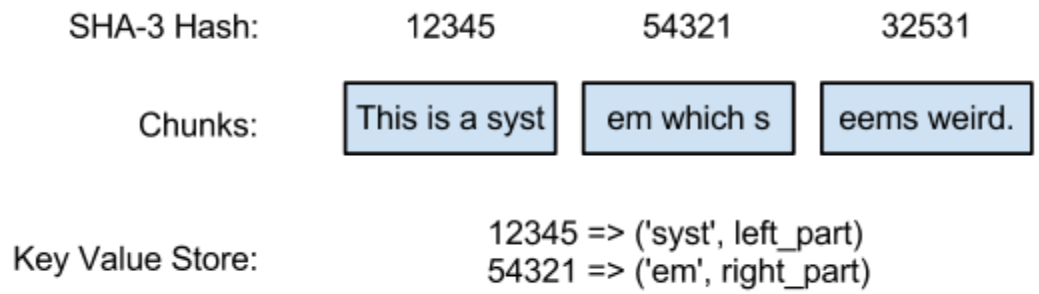
String Searches

Luxor searches for a string in all previous versions of the filesystem by searching through the chunks. The ***luxor grep*** command goes through each chunk looking for the specified string using the Boyer-Moore string search algorithm. Whenever the string is found within a chunk, the chunk's SHA-3 hash is appended to an array.

However, this method does not match strings which are broken across two chunks. Therefore, Luxor makes a second pass over the chunks. In this pass, Luxor keeps in memory a set of all possible ways of breaking up a string into two parts. For example, the figure below shows all ways of splitting the word "system" into left and right parts.

Left Part	Right Part
s	ystem
sy	stem
sys	tem
syst	em
syste	m

Luxor looks for chunks which end in any of the left parts or begin with any of the right parts. If a chunk matches, it is included in a key value store which maps the chunk's SHA-3 hash (the key) to a tuple of the matching string and match type (the value). The match type is either "left_part" or "right_part." The figure below shows a key-value store populated for three small chunks.



Once all chunks have been processed in the second pass, Luxor adds the SHA-3 hashes of all chunks from the first pass to the key-value store. Luxor can now query the key-value store with a chunk's SHA-3 hash, and check if the string was found inside the chunk, or if the chunk begins or ends with some part of the string.

After the key-value store is constructed, ***luxor grep*** iterates over all chunk-storage blocks and queries the key-value store for each SHA-3 hash that appears. Searching involves sequentially scanning each chunk-storage block for SHA-3 hashes. If the SHA-3 hash appears in a key-value store from the first pass (i.e. the chunk contains the string), then the filename is appended to the return array. If a SHA-3 hash indicates that the chunk matches the left part of the string, then ***luxor grep*** will append the filename to the return array if the next chunk completes the string.

The algorithm requires that search strings be less than 256 characters (the size of a block) since it does not handle multipart storage. This is a reasonable restriction since relatively few queries will exceed this size.

To perform string searches in a particular file's version history, Luxor takes the string search algorithm and confines it to check chunks from a particular file's version history. This can be performed quickly because each version has a link to its ancestor, so Luxor can iterate through an entire file's history.

Excluding Files or Directories

Luxor excludes a file or directory from versioning using ***luxor exclude***. To handle exclusion, Luxor adds an "excluded" bit to each inode. Luxor ensures that a file's excluded bit is 0 before calling the ***new_version*** command on a file. Excluding a file leaves the previous versions of the file, but new versions are not created until the excluded bit is set to 0 again.

To exclude a directory, Luxor sets the directory's excluded bit to 1 and does a breadth first search for all files within the directory. The files within the directory will have their excluded bits set to 1. A user can manually negate the exclude on any file in an excluded directory.

New and moved files or directories will have their excluded bits set to the excluded bit of their working directory. If a file appears in more than one directory, the excluded bit is set to 1 if any parent directory has its excluded bit set to 1, and 0 otherwise. The root directory's excluded bit is permanently set to 0 so files and directories are versioned by default.

Binary Files and Directories

Luxor has no problem handling binary files or directories. Binary files are simply broken up into 256 byte chunks and placed into chunk-storage blocks like normal files. Directories are versioned normally, with their inodes referencing the lastest version file, except that new version files are created only when the directory structure changes. This occurs whenever files are renamed, added, or removed.

Adding and Deleting Files

Adding or deleting files pose no problem to Luxor. Whenever a new file is created, a new version file is initialized with a null child pointer, signifying that it has no previous version. Whenever a file is deleted, its version file persists and will eventually get garbage collected (see Garbage Collection section).

Hard and Symbolic Links

Luxor handles hard links without problems since, fundamentally, Luxor only changes the inode

and block layers. The versioning of directories ensures that Luxor has history for the addition and removal of hard links.

Luxor easily versions symbolic links. Since symbolic links are nothing more than files containing a pathname in their blocks, Luxor can create version files whenever symbolic links are created or modified.

Thread Safety

Race conditions can occur when multiple processes close a single file in rapid succession. To prevent this, Luxor adds a “new_version_counter” to each inode, which defaults to 0. Each call to ***new_version*** begins by atomically incrementing the counter if the counter is below 64, and finishes by atomically decrementing the counter. If the counter is greater than 0, then the file is placed in read only mode. This prevents other proccesses from modifying the file while ***new_version*** is running. The counter limit of 64 prevents too many processes from creating new versions, though it should not typically be reached in practice.

Notice that Luxor allows multiple ***new_version*** commands to run concurrently, since Luxor uses atomic reads and writes to the “most_recent_version” field in an inode. Atomic operations prevent the “most_recent_version” field from being invalid. Thus, if the underlying operating system allows multiple processes to overwrite each other’s modifications to a file, Luxor will faithfully record the history. For example, if two processes opened a file at the same time, made conflicting modifications, then closed the file at the same time, Luxor would create two versions in the history.

Garbage Collection

When over 90% of the disk space allocated to the versioning system has been used, the garbage collector initializes a max-heap (using 5% of memory) and begins a new process which iterates over all version files on disk. If a version file has a datetime which is smaller than the maximum datetime on the heap, then it is inserted and the current maximum is popped from the heap (unless the heap is not yet full). After iterating over all files, the max-heap will contain the earliest version files. Since inserts and deletes on a max-heap require *O(log n)* time, the entire process is accomplished in *O(n log n)* time, where *n* is the number of version files.

The version files left in the max-heap will be deallocated from disk. Additionally, each of the storage-chunks blocks in the version files and their corresponding chunks will have their reference counts decremented. Once it’s reference count reaches zero, a chunk or storage-chunk block will be deallocated and have its disk space freed. Multiple passes of garbage collection occur until disk usage drops by at least 10%.and this will have been

Automatic Saving

When the user closes a file, Luxor begins to create a new version if the file was modified and has not been excluded. Luxor creates versions on close (rather than on write for example) because closing a file usually signifies that the user is completely finished with his or her changes. Other file system hooks, such as write and save, occur too rapidly to provide useful versions and would clutter the filesystem without much benefit.

Analysis and Performance

This section analyzes the performance of Luxor for a number of important use cases.

Writing and Reading Chunks

Luxor improves the performance of reading and writing chunks by writing them sequentially on disk. Since the ***new_version*** command reads from a file and creates chunks sequentially, the chunks are usually written in the order in which they are read from the chunk-storage file. This improves performance because reads from a chunk-storage block tend to be sequential.

Deduplication of Chunks

Luxor can afford to create a new file for every version because of the high probability of deduplication. Since chunks are reused for files, one can save substantial disk space, especially for frequently used chunks. If chunks of files are repeated often, Luxor will cache the SHA-3 hash of the chunk and use the same chunk for multiple files. There is a high probability of collision as most files contain very similar character patterns.

Caching of SHA-3 Hashes

Since the most recently used SHA-3 hashes will be cached in memory, the CNode index can be completely bypassed quite frequently. Reads from the CNode index are somewhat expensive because of they are located on a different section of disk from the chunk-storage blocks.

Batching

Luxor improves performance by batching operations. For example, Luxor keeps SHA-3 hashes in memory before checking for their presence in the CNode index. This batching improves performance by making sequential disk reads on the CNode index.

Multiple Processes Creating New Versions

Luxor uses multiple processes in order to speed up the performance of the ***new_version***

command. Since this command is thread safe, as argued in the “Thread Safe” section, one can close multiple files concurrently and Luxor will create new versions in the background.

Modifications to a File

Making modifications to a file requires only a number of writes linear in the number of bytes k of modifications, for both large and small files. A version file will be created for each new version, which is a single write. Then, new chunks will be written for the modifications made to the log (if the chunks are not already in memory). This requires at most k/256 new chunks. Finally, at most k/512000 + 1 chunk-storage blocks will be created and the rest will be referenced from the previous version (since each chunk-storage block contains 8000 bytes / 4 (bytes/chunk) = 2000 chunks * (256 bytes/chunk) = 512000 bytes). Thus, a total of 2 + k/256 + k/512000 blocks will be written.

String Search Operations

The string search algorithm requires O(n) time and O(n) memory, where n is the number of chunks searched, because at most n chunks will have their SHA-3 hashes stored in the key-value store and the algorithm makes three passes over all chunks. The first two passes construct the key-value store, and the third pass outputs the files which match the string. However, note that O(n) memory usage is highly unlikely since only a small subset of chunks are likely to contain the string and be in the key-value store.

Overall Performance

Although string search operations require a significant amount of time and memory, normal operations of the filesystem are extremely fast. The speed of normal operations are achieved by caching, filesystem deduplication, and multiprocessing of the ***new_version*** command. These allow the filesystem to keep its speed in normal operation and make expensive operations only when users would like to examine file history. An extremely high volume of file closes would crash the system. However, this only occurs when the available memory runs out, and is extremely unlikely normal operation.

Conclusion

Luxor provides an easy way to track filesystem history. Built on version files, Luxor can create new snapshots quickly and in parallel. The speed of Luxor allows it to automatically take snapshots whenever files are closed, reducing work and complexity for the user. Luxor accomplishes this through deduplication and hashing of chunks. Unlike a diff-based system,

Luxor's file deduplication handles directories, links, and binary files without any extra effort. Luxor operates seamlessly and enables history tracking without the user even noticing.

Acknowledgements

Judy Deng
Sashko Stubailo

Word Count: 2853