

UNIX Time-Travel File System Design

Using Block Level Modifications

Tigran Sloyan/sloyan@mit.edu

Instructor: Prof. Butler W. Lampson

Section R08, TR2

March 18,2010

Introduction and Overview

This design attempts to modify the API and the physical block level structure of the standard UNIX file system in order to implement the Time-Travel feature. The main idea of the design is to save a new copy of each physical block that gets changed. This choice is not the best one in terms of memory efficiency compared to the byte level approach, however it is made to ensure simplicity. It is also not the simplest approach, since we could have employed an even higher level approach by saving a copy of the whole file whenever it would be modified. Nevertheless, such a design would be highly inefficient in terms of memory usage.

To create the illusion for the user that the state of the file system is copied at any point in time, we build a new file hierarchy in addition to the main one. However, in order to efficiently implement the new archive hierarchy, we need tools that we gain by making changes in some of the original procedures of the API. The main idea behind the changes in the API is to have the `open()`, `write()` and `close()` operations initialize and share a common datatype (in this design a table), that will contain the modified blocks and their metadata and will be used by `close()` to construct the “Archive”.

We start by look at the structure of the virtual hierarchy and then we see how the changes in API help us actually build that tree.

Archive hierarchy

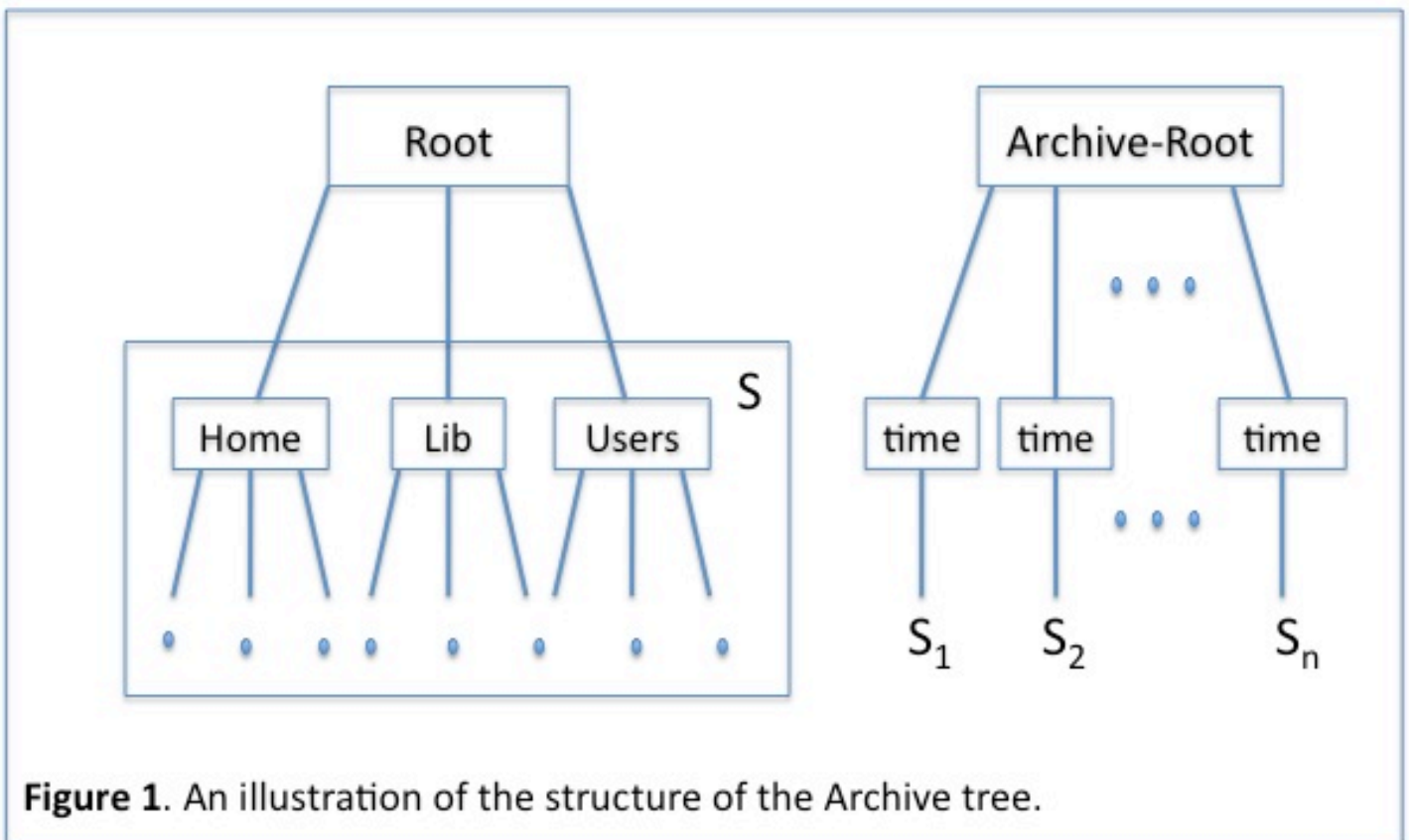
Since we want our documented history of the file system to be “virtual”, i.e. only accessible by path names, we create a separate historical file hierarchy with a new root “Archive-Root”.

We can see the overall structure of the virtual tree in Figure 1. The tree on the left in Figure 1, represents the general hierarchical structure of the file system, whereas the tree on the right is the new Archive tree. The way it is built is that the folders in the Archive-Root directory (marked as “time” in Figure 1) are named corresponding to the times when there was a change in the main file system. For example, if a change to a particular file was committed at time

2010-02-09-09:00:00.000

then there would be a directory in Archive-Root with the name

“time-2010-02-09-09:00:00.000”.



These “time” directories contain the modified sub-tree of the main Root directory. In Figure 1, S is the sub-tree of the Root directory and S₁, S₂, ..., S_n are

the states of S at the times specified by the names of the corresponding parent “time” directories. For example, if the parent directory of S_1 is “*time-2010-02-09-09:00:00.000*”, then S_1 will be the state of S at time *2010-02-09-09:00:00.000*.

Building The Archive Tree

Even though the (S_i) s create the impression that the whole state of the file system was saved at the specified time it is not so. We copy the whole file system only when the first change is made. In other words, S_1 is a copy of original file system at time 0. All other states are mostly hard-linked to the previous state, i.e. all the files and directories of S_{i+1} are hard linked to S_i except for those files that have been modified at time i . Those files are only partially linked to the corresponding files in S_i . The blocks of the modified files that weren’t changed by `write()` would still be linked from S_{i+1} to S_i , as for the modified blocks, we will see below how the `close()` operation will provide copies of those blocks for S_{i+1} .

Modifications in API

In this section we discuss the necessary changes to the `open()`, `write()`, `read()` and `close()` operations. The next section will demonstrate how we make all the other operations compatible with the design.

`open()`: There are three changes necessary in this procedure. First, in order to avoid confusion we should ban the creation of files that have a name starting with “time”. Second, there is a possibility that a superuser might try to modify a historical file, therefore, `open()` would check the references to the given file and if it is in the archive tree, then `open()` would restrict the access to read-only. And finally, `open()` should initialize an extendable table, which we call the “Table of

Changes”, stored on RAM that initially contains only the i-node of the current file and will be filled up by write().

write(): Whenever write() is called to modify a certain part of a file, it identifies the blocks that will be affected by the changes and puts a copy of those blocks’ data and its location into the Table of Changes which will later be accessed by close(). Only after the data is put in the table, write() gets to change the file. Figure 2 illustrates the structure of the Table of Changes.

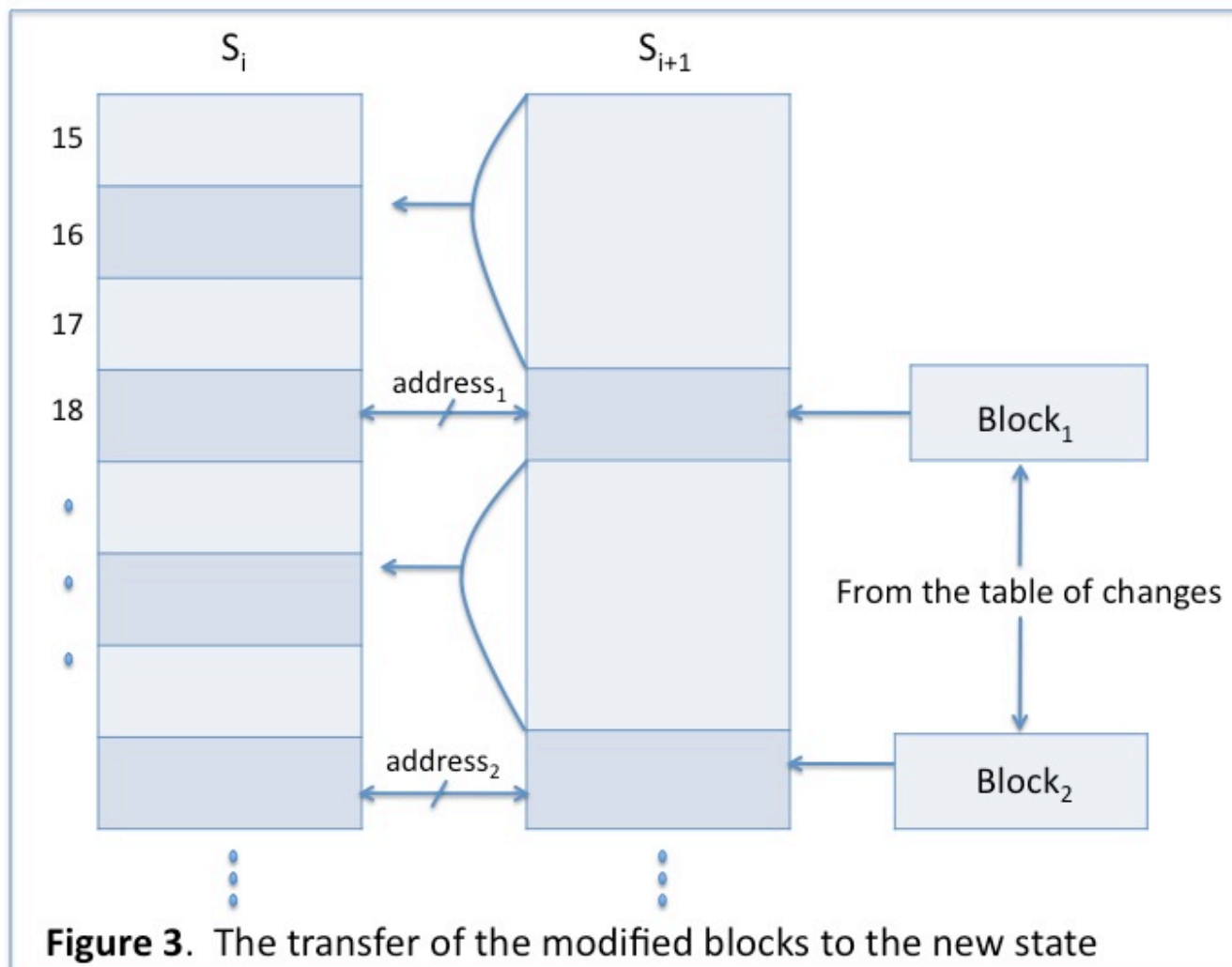
I-node	
block ₁	address ₁
block ₂	address ₂
...	...
block _{n-1}	address _{n-1}
block _n	address _n

Figure 2. The Table of Changes

read(): When we are given a path to open a file, we check whether the path starts with “/time-...” and if it does we need to make sure not to modify the last access time in the i-node of the corresponding file. If the path doesn’t start with “/time-...” read() can proceed as it normally would since we know that we are reading an up to date file-not an archive file.

close(): Close plays the most important role in constructing the Archived Tree. Once it’s called at time i+1, it creates a new directory T-(i+1) in Archive-Root, which contains the state S_{i+1} as shown in Figure 1. Initially, all the files and directories in S_{i+1} are hard-linked to S_i, hence S_{i+1} initializes as S_i. Then close() takes the table that contains the information about changed blocks and traverses the block map of the given file in S_{i+1} looking for blocks with the same physical address as in the table. Once it finds a matching address, it makes a new i-node for this file that initially has the same information as the previous i-node and breaks the hard link. Then it takes the block data from the Table of Changes corresponding to the found address, places it on a free physical block and

updates the block map information of the new i-node to make sure that the block map of the file in S_{i+1} is pointing to the changed data from the Table of Changes. This process is illustrated in Figure 3. After this process is done, close gets rid of the table to free the registers on the RAM and proceeds to close() the file as in the usual UNIX file system.



In Figure 3 we can see how the blocks and correspondingly the block addresses of S_i and S_{i+1} are mostly the same, aside from the blocks at $address_1$, $address_2$ and so on (all the addresses are taken from the Table of Changes). For these addresses, as described above, we get the corresponding blocks from the table and get the block map in the i-node to point at those.

Changing Permissions

To make sure that the user doesn't modify any of the files or directories in the Archive tree, whenever we create a file or directory in the Archive tree we change its permission to read-only. This also handles the issue of making the other API operations consistent with the modified file system. Since in the Archive tree all the files and directories are read-only, operations like *rename()*, *link()*, *unlink()*, *symlink()* and *mkdir()* would not work in the Archive tree, and whenever the user tries to call those on a historical file, it will return an error. Because superusers can modify a file even when it has permission read-only, we handled that case separately by adding a check in the *open()* operation.

General Analysis

In this Time Traveler file system design, the most important choice to make was to decide when and how to update the Archive tree. The sensible choices were to update it during *writ()*, i.e. update the Archive whenever a change is made, or buffer the changes and update all together in *close()*. The advantages of the first approach are firstly not having to deal with RAM or carry around and update a table of data, and secondly updating each changed block one at a time requires less time than updating all the changes at the same time and thus the generated latency is smaller at any particular point of time. Nevertheless, the advantages of the second approach greatly overweight the advantages of the second one.

To name some advantages of buffering the changes and then updating them in the *close()* operation, we should consider the latency issue from the user's perspective. When a user is updating a file, he expects that to happen as fast as possible to be able to continue your work, whereas when you are closing the file, you don't really worry much about the latency. In this case, not having *writ()* do the work, i.e. finding a free spot on the hard drive to copy the blocks that will be changed, and then updating the corresponding i-node in the Archive tree, will speed up the performance of the file system during the time when the

user is actually working. It will of course take longer for the close() operation to complete it's job if we make it update all of the changes together, however, the user wouldn't care much about this and that's our main priority, to make the file system work as fast as possible from the user's perspective.

On the other hand, if we consider the total latency in the two mentioned cases, it turns out that having write() do the work block by block would actually increase the total latency. We know that when building the two rooted trees of the file system, one tries to make each of them as contiguous on the hard drive as possible to improve the overall performance. Consequently, the physical locations of the blocks of the first tree would expectedly be well apart from the block locations of the second tree. Hence buffering the changes and then updating them all together is the optimal choice since the disk head wouldn't have to go back and forth between the two "distant" file hierarchies.

Moreover, having to keep a buffer shouldn't add too much latency because most of the time our Table of Changes wouldn't get too large since it only contains blocks of data and their addresses and thus Cache practically would be able to handle the buffer and we won't have to store the buffer on RAM.

Workload Analysis

Now we can analyze three scenarios and see how our design performs on each of them:

Scenario 1: The first scenario tests the design's ability to document changes in many small files. Given that a user creates a website with many small HTML parts at time T1 and then edits each of them several times and then goes back to time T1 and reads each file, we want to see what performance and space usage would our design provide. Once the changes are made and the process is complete, going back and reading the old state would require the same number of I/O operations our design is build such that after creating the Archive tree it operates like a regular file hierarchy just like in standard Unix file system. The

trade-off here is that when having a lot of small parts, the tree can be quite big with a lot of connections, and the hardlinking process described in the `close()` operation which basically constructs the Archive tree will add a considerable amount of latency. In terms of space use, since we save a copy of each block as soon as it gets modified, the total space usage will depend on the number of times each block gets modified. Thus, if each block is modified k times, and the number of blocks affected is n then we will save $n*k$ extra copies of each block plus the whole original copy.

Scenario 2: The second scenario tests the design's ability to document changes in one large file. In this case, instead of many small files, we have one large movie file (size about 10GB) at T2, and we make changes to its parts that affect about 20% of the used blocks on the hard drive, and we are interested in knowing the relative number of I/O that will be required to get these changes done in our design and in the standard UNIX file system. Since the only major write operation we do is in building the S_1 state, when we take the whole file system and make a copy of it, if we consider T2 to be the starting time, then the number of operations will depend on the size of the whole file system, since we need to copy every single block to a new location. But if we logically assume that T2 is not the starting time, i.e. some changes were already made in the file system, then our Archive tree should be already partially build (the S_1 state is already saved), then after that step we don't do many more writes besides copying every single block that gets modified, which means that for a single write in standard UNIX we do two writes, therefore, the number of I/O operations would be twice as many as in the standard UNIX. To save the historical version of the movie, we only need to save the original version of the movie and a copy of each block that gets modified, therefore the space required would be $(1+0.2=1.2)$ *(size of the movie) assuming that the original size of the movie doesn't change and that each block is modified only ones.

Scenario 3: The third scenario tests the design's ability to playback the historical versions of a large file. Given the set up from scenario 2, we try to go back and read the original movie file. As in scenario 1, after the Archive tree has been built, the file system for the archive operates exactly the same way as the current file system, thus the number of I/O operations would be the same when reading an archive file and when reading a current version. We assume again that the total size of the movie doesn't change.

Conclusion

Overall, the design provides the Time Traveler feature with excellent performance and using a reasonable amount of disk space. In addition, it preserves compatibility with applications programmed for the standard UNIX file system API and it makes sure that the historical files can not be modified and can only be accessed by `chdr()` operation.

Furthermore, the space usage and the performance can be improved given the required specifications. To improve the space usage, since we are only interested to save the state of the file system at the times when `close()` is called, we can modify `write()` not to save a block when it gets changed again but rather get rid of the old version and keep the new one. To improve the performance, if we make a careful choice of the datatype that we are using to buffer the modified blocks, we can effectively rearrange the blocks in the buffer before updating the Archive tree such that contiguous blocks would be updated one after the other providing faster hard drive performance.

Acknowledgments

In the end, I would like to thank Mary Caulfield from the Writing Center for her comments on my writing, which helped me revise my report and make it more efficient and more reader-friendly. I would also like to thank my classmates, Sinchan Banerjee and Rafael Oliveira, for the productive discussions that we had on the UNIX file system before writing the design proposal.

Word count: 2650