Computer System Engineering

Design Project 1 Report

Create a Versioning File System

Author:
Bo Song 11302010003
YiTing Cheng 11302010050
YuWei Zhou 11302010067

April 14, 2013

1 Introduction

The goal of this design project is to create a versioning file system. The versioning or continuous snapshotting file system is defined to store all versions of each file over time. When user modifies a file or a directory, the versioning file system create a new version after closing the file or directory and store the old version which can be only read by users.

To implement this system, The whole design will solve two major problem: how to maintain the old version and how to manipulate them, as well as achieving low access time and better space utilization under common use cases.

2 Design Description

The versioning file system(VFS) represents different versions of a file or directory as different inodes which is based on the inodes in Unix file system. Therefore, we need add extra fields to origin inode data structure. Then, a new layout and management is required to maintain the huge number of inodes and data blocks. Finally, some interfaces is adopted to let users manipulate different versions easily.

2.1 Data Structure

2.1.1 On-disk Structure

In Unix file system, we treat file and directory as inode on the disk. Similarly, VFS also use inode and there are several fields are added to it as mentioned below(Figure 1).

Timestamp When a new version is created, a unsigned integer is recorded, representing the number of seconds passed since a epoch which is predefined in the system. If the unsigned integer is 32 bit, the timestamp allows the system to represent about 132 years after the epoch(typically 1970.1.1) which is enough to current system, and it is easy to extend to 64 bit when we need to represent more time.

Bitmap The system embeds bitmaps in inodes and indirect blocks that allow system to record which blocks have had a copy-on-write performed which will discuss in memory management section. A bit of value 0 indicates that a new block needs to

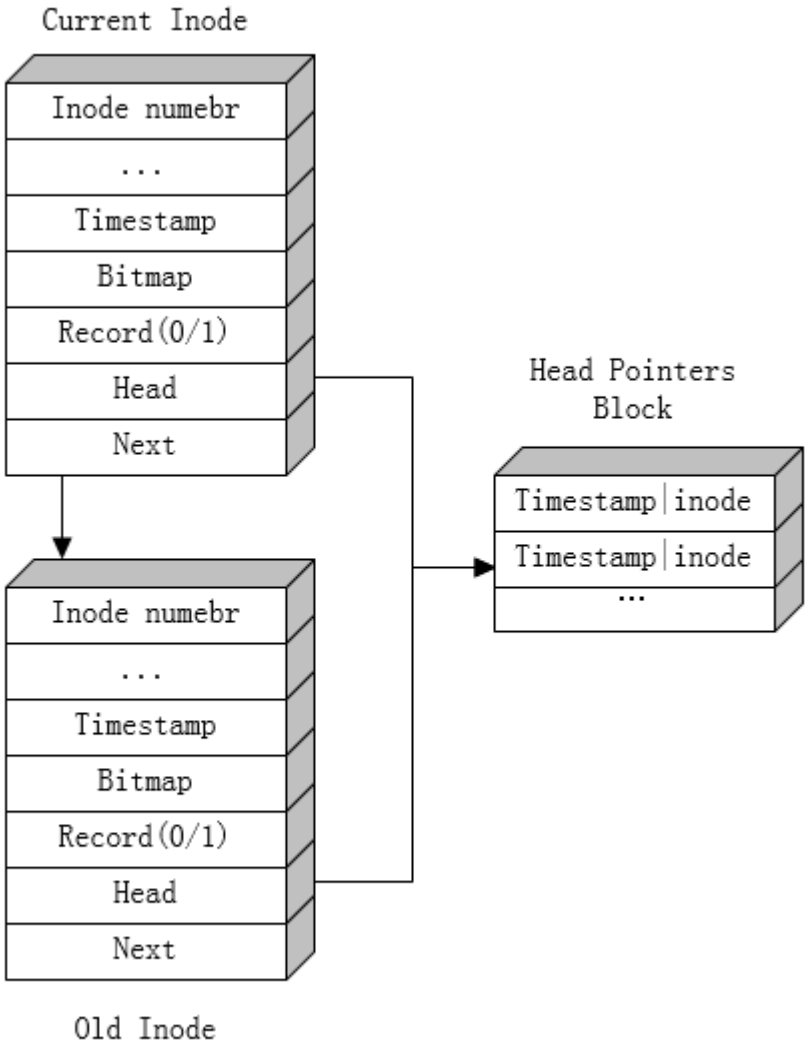

Figure 1: Inode Structure

be allocated on the next write and bit value 1 indicates that a new allocation of this block has been performed. The size of Bitmap is mainly the overhead of extra fields but we think it worth costing it to optimize performance for its three uses mentioned below.

Next A pointer to the next old version of an inode. Through this field, we can search a specific version of an inode like an linked list, and the current version inode is the head of linked list. Since the performance of it may not so considerable, a optimized structure will discuss in layout section.

Root Pointer A pointer to a block containing addresses of the root list of B+-tree layout structure discussed in layout section.

Since the size of inode is fixed(typically 2K in fast file system), the number of indirect block pointer fields will decrease after adding new fields and it will result in reduced max file size under approximately 10% which is acceptable.

Although the Root Pointer field is only used in current version inode and waste some space in old versions, the old version is read only and inode is fixed size. Consequently, it is meaningless to free these fields in the old version inode.

2.2.2 In-memory Structure

The in-memory structure of the system is similarly to the origin Unix file system which containing a open file table which points to its vnode. In VFS, the in-memory copy of an inode is always kept up-to-date with the current version, allowing quick access for standard operations since the number of operations of current file is dominated. To ensure that the current version can always be accessed directly off the disk, the VFS could check it on a cache flush.

2.2 Memory Management

2.2.1 Copy-on-write

When a new version is created, it is too waste to copy all the block in the old version. So copy-on-write(abbreviated COW) is adopted to implement multiple versions of data compactly. The system needs to create a new physical version of a file only when data changes. Frequently, physical versions have much data in common. The COW allows versions to share a single copy of file system blocks for common data and have their own copy of data that have changed(Figure 2). As a result, it extremely improves the memory utilization in the system.

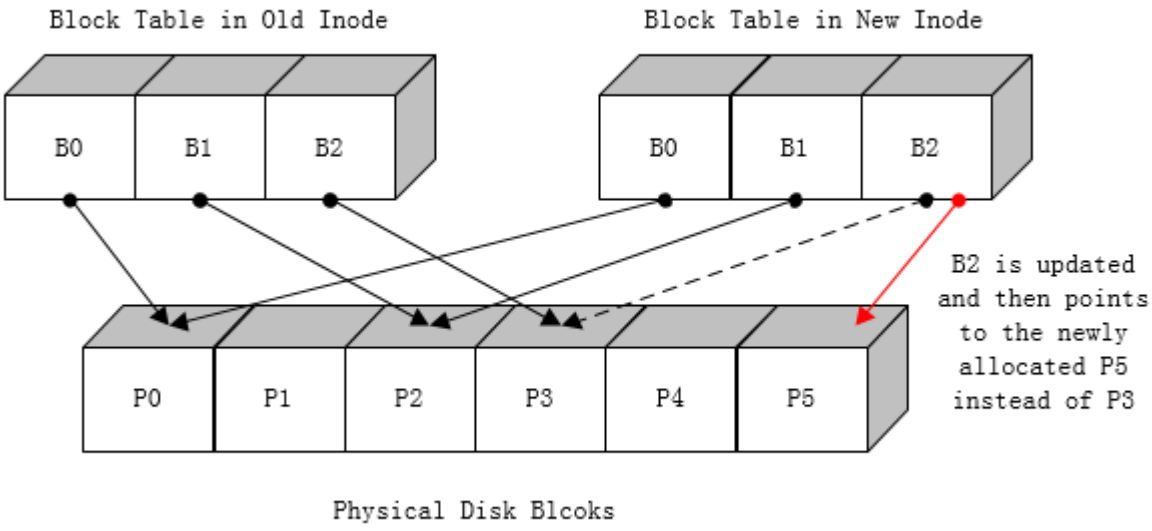

Figure 2: Copy-On-Write Process

How COW works is showed in the following steps:

1. When users open a file or directory, a new inode is duplicated with different inode number and same block references.

2. When users have modified something, the system will allocate new blocks to store modified blocks. The block reference in the new inode will point to the new allocated block and the old one remains unchanged, at the same time, the bitmap in the new inode will also update.

3. When users finally closed the file, the system check whether the bitmap is all zero. If it is, it means nothing changes and the new inode will free. Otherwise, a new version of inode is finished.

2.2.2 Garbage Collection

When the disk is almost full(A predefined constant such as 90%), the system also support garbage collection by repeatedly scanning inode table and freeing the oldest inode sublist in the layout structure(discussed in section2.3) of each current inode till the space decreased to another predefined constant such as 80%.

2.3 Layout

Considering there will be a large quantity of version, the of work searching for a target version will be difficult and cost a lot of time. A structure that maintains all versions is obviously needed.

2.3.1 B+-tree Structure

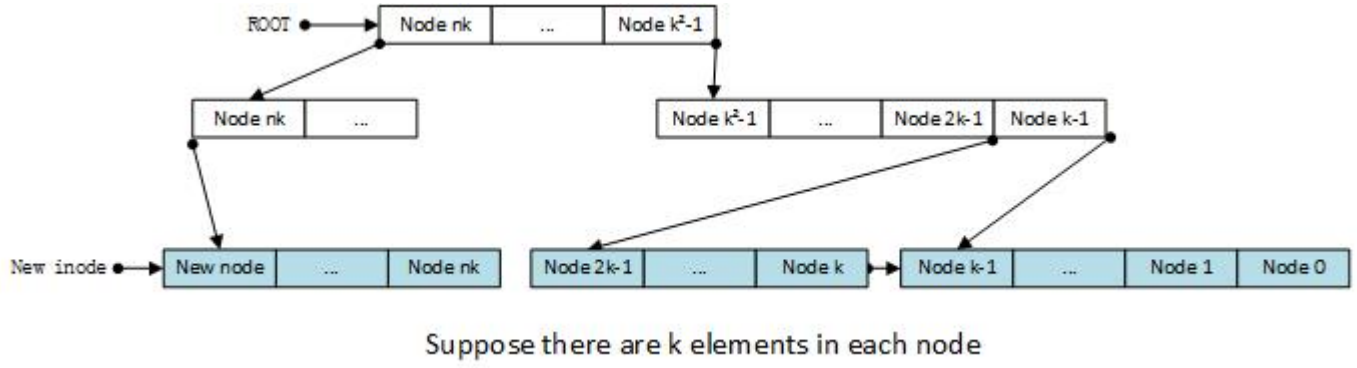

Figure 3: B+-tree Layout

To solve the searching problem, we use B+-tree structure to maintain these versions in our file system. As the figure3 shows, the tree uses timestamp to be the key and inode to be the value, and stores all the value in the leaf-nodes. When searching a certainly version, it will start at the root and straightly to the leaf, with the binary search in each node. For finding the target inode number, it costs time $\Theta(\lg n)$.

To store the structure, we use a few blocks (referred as log-block in the following report) to store it. As shown in the figure4, at the head of the block, there are two interfaces, one is the pointer of current file while the other is the pointer of the root of the B+-tree.

For the purpose of maximizing the effect of B+-tree, each log-block only stores a single node. As we study in the ICS course, when accessing a value in disk, it loads the whole block in the memory. B+-tree puts a few values in a single node according to the phenomenon. As a non-leaf node, it stores keys/log-block numbers in the field. The leaf node stores the keys and inode number of file instead.

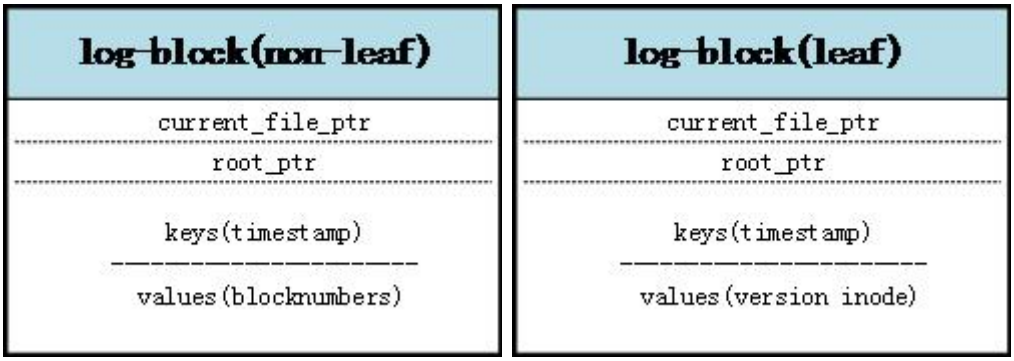

Figure 4: Log-block Structure

2.3.2 Abandoned Sublists-only Layout

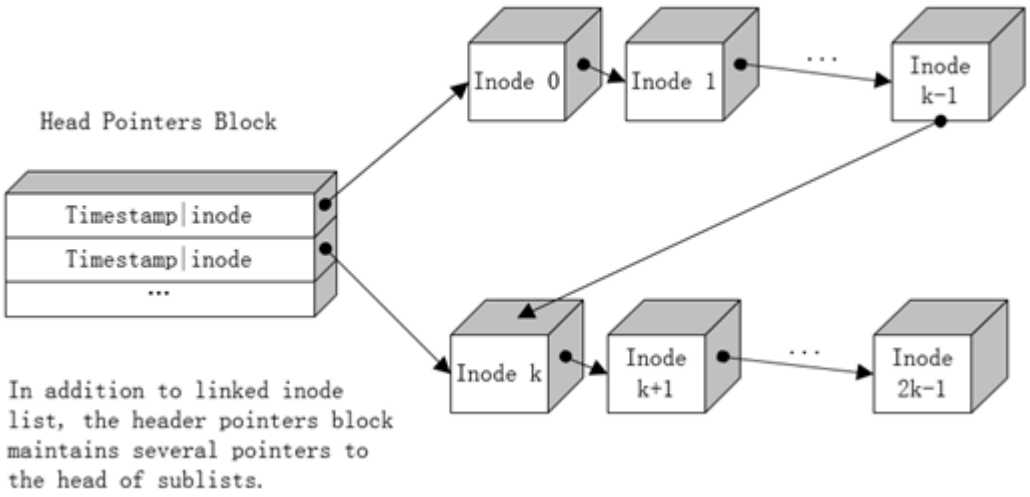

Figure 5: Sublists-only Layout

When the length of linked inode list exceeds a predefined number k, the system divides it into small sublists. The head of each sublist which contains the address of first inode in the sublist and its timestamp is recorded in a block in the inode structure(Figure 5). When the user find a version of file with given timestamp, the system use the timestamp as a key and search the sublist containing that inode

efficiently with a function. We abandon this method because this effective function does not exist.

2.4 Interfaces and Manipulating Old Versions

2.4.1 File System Operation

The system supports all the standard Unix file system operations as follows:

- **write** - When write something, the system will check the bitmap, then COW is triggered to allocate new blocks or only update the newly allocated blocks.
- **open** - A new inode is duplicated with different inode number and some other fields, but they share the same block pointers. If a read-only file is opened or the Root field is null, the system dose not duplicated the inode like the Unix file system.
- **create** - A new inode is created as Unix file system. However, user can explicitly set the the file is not versioning which sets the Root field to null. Finally the directory containing it will do versioning too.
- **close** - Check bitmap to decide whether the system free the inode. If not, add timestamp, next fields and do correspondent layout operation.
- **rename** - Combination with unlink and link as follows.
- **unlink** - If link count drops to zero, the inode will not free and the user can still access the old version of that unlinked file.
- **mkdir** - Create a new inode related to the new directory
- **read, chdir, symlink, link, stat** - The same as origin Unix file system

2.4.2 Accessing Old Version

User can access any version of file or directories by appending @ and timestamp, such as: **cd /home/sb@362480234**

When the system resolve the file or directory names, it reads from right to left and regard the last @ and number as version specifier. Hence, it can distinguish the

version specifier from @ and numbers in its origin name.

When the system gets wrong timestamp, it will search the inode layout structure and give a nearest version.

To meet the demand that users want to know how many versions created. A new operations is introduced:

- **lstv name** - name is a file or directory name, and lstv lists all the versions of it with correspondent timestamps.

In our consideration, there is another layer between the VFS and users. The application in that layer translates annoy things like timestamp to more user-friendly interfaces such as index search. Therefore, for simplicity, the VFS only provides uniformed timestamp interface.

3 Analysis

3.1 Use Cases

3.1.1 Search

When users search a string across all versions of a file, there are two strategy to accomplish this task.

The first one is searching all versions thoroughly. The system get the current inode of the file to search, scanning all the blocks of it, then going through the inode linked list and searching all blocks of each inode.

The second approach does not search search all the blocks to save time. When the system has finished searching the blocks of one inode and begins to search the next inode, it only searches the blocks that has been changed according to the bitmap of the last inode.

It seems that the first strategy is naive, but it always gives the right answer. However, the second strategy may be skip the answer when the target string is too long to fit in a block or occasionally across two blocks. So we need some complex mechanism(such as check the length of string first and choose appropriate number of blocks as smallest unit to search) to handle this problem. As a result, if the overhead of that mechanism

is not big and works well, we choose the second one. If not, the first one is chose according to 'Correctness comes first.' principle.

When users search a string across all the file system, we can also search all blocks of inodes or skip the same block in different version or even in different files with complex method. The trade-off is similarly to the above one.

3.1.2 Truncate

Truncate is a frequent file system operation, it is very common to see that applications or users truncate a file to zero length to rewrite that file again and again. If there is no bitmaps in the inode, we will deallocate all truncated blocks and allocate new blocks which leads to waste space and let the block of a file become more separate. With checking the bitmap, the system deallocates only those blocks that have been written in the current open progress instead of allocating new blocks to duplicated COW. Therefore, although bitmaps cost a relative big space, it is very useful to optimize truncating time and save a huge memory space.

3.1.3 Multithreads and Log File

One advantage of the COW execution flow is that it is not influenced by multithreads. If different threads of an server program open the log file simultaneously, then they will get their own duplicated inode, allocate their own new blocks and generate separate versions. It enforces the isolation between different threads and different versioning inodes under processing.

3.1.4 Repeatedly Writie

- **Case1** - Repeatedly writing to a small file.
  Since the system duplicate inode to create new version, if the overhead of inode size is extremely huge when the file is small. Under extreme case that the file only occupied nearly one block, the overhead of versioning will rise to about 50%.

- **Case2** - Repeatedly writing a block of a large file
  Thanks to COW and the inode space can be ignored when file is large, we only

allocate one more block per versioning in this situation which is ideal and cam also be ignored.

3.1.5 Exclude Files From Versioning

When a user want excludes a file or directory from versioning, the system sets the root field in the inode structure to null, since the root field and the layout structure is useless when there is no versioning. But the side-effect is that if the user exclude an exist versioning file from versioning, the system will free all its old versions and we consider it is acceptable.

Another method is that we introduce a new field indicating whether the file is versioning. The advantage of it is that it maintains old version in the above case, and the drawback is that it spends extra space which we think is unnecessary.

3.2 Alternative Approach

There is another promising approach realizing VFS without duplicating the inode. Only an inode is used to represent all versions of a file. There is no timestamp, bitmap, next fields in the inode but it has a pointer which points to a series of blocks containing many bitmaps. each bitmap makes changed blocks according to last version. If users want to get an indexed old version, the system will retrieve the blocks according to each bitmap one version by one version(Figure 6).

It is obviously that this approach saves the space of inode, but it will cost huge time to retrieve an old version. In our consideration, the less retrieving time is more important than the bigger space of metadata. However, if there is optimized mechanism based on this approach which reduce retrieving time, we will then select this one. In conclusion, the duplicated inodes strategy is more suitable to VFS under the current research situation.

4 Conclusion

The above design meets the requirements as outlined in the problem statement mainly by duplicated inodes and COW strategy. The remaining problem is a correct edge check in searching and cache related issue. Perfect unduplicated inode approach and failure tolerance are further research.

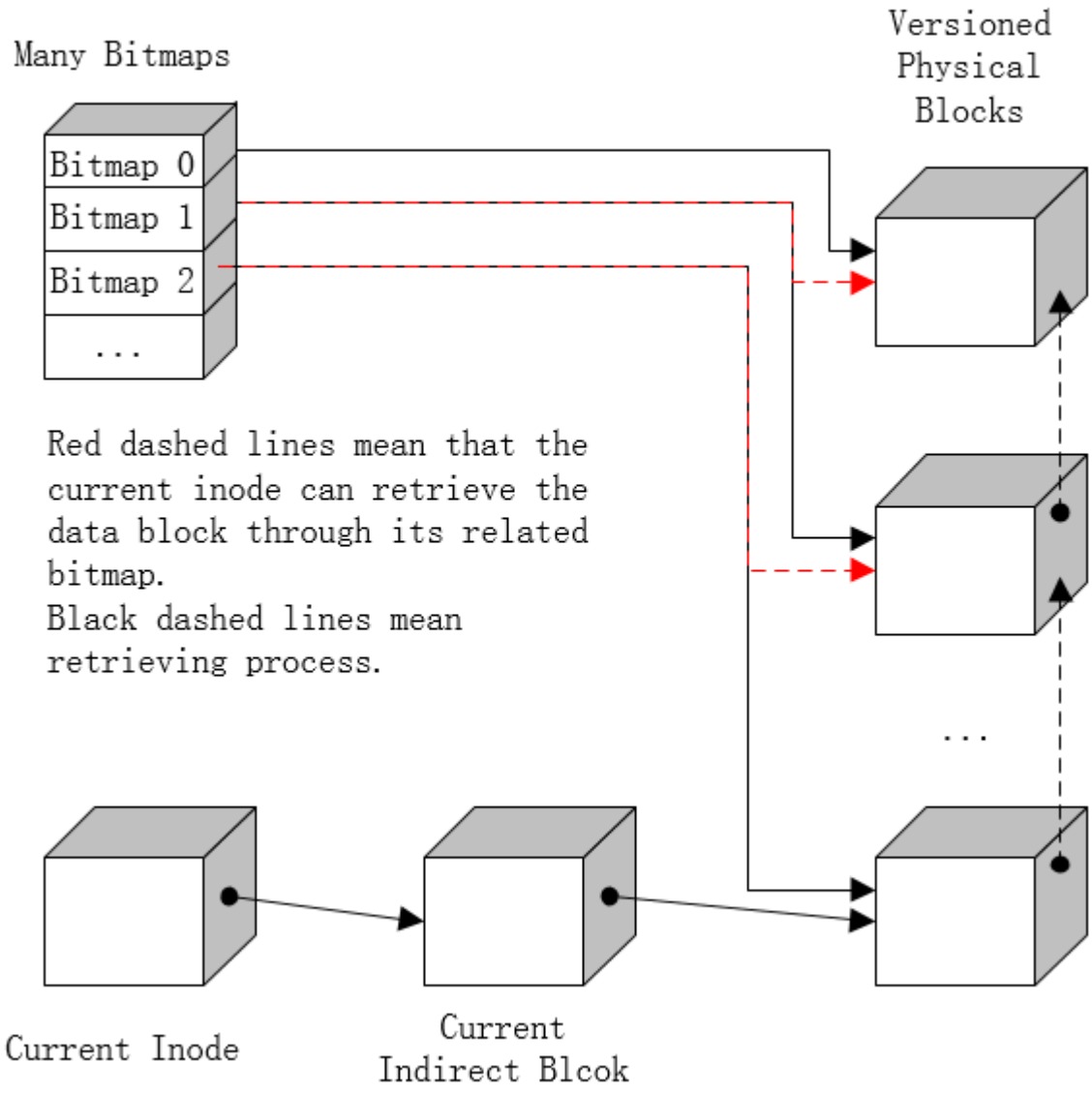

Figure 6: No Inode Duplication Approach

Word Count:2500(exclude title page)