Computer System Engineering

Design Project 1 Report

Create a Versioning File System

Author:
Bo Song 11302010003
YiTing Cheng 11302010050
YuWei Zhou 11302010067

April 9, 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

In Unix file system, we treat file and directory as inode. Similarly, VFS also use inode and there are several fields are added to both the on-disk and in-memory representation of the inode as mentioned below.

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 be allocated on the next write and bit value 1 indicates that a new allocation of this block has been performed.

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.

Record When a user want excludes a file or directory from versioning, an extra bit is employed to indicate it. a bit of value 0 is versioning which is default, and one is not versioning. When the system wants to create a new version, it will first exam this bit and decide whether to do subsequent things.

Head Pointer A pointer to a block containing addresses of the head of sublists of inodes 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 Record and Head 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 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. As a result, it extremely improves the memory utilization in the system.

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 Bitmap and Garbage Collection

It is a very common case that applications truncate a file to zero length as a first step when rewriting that file. On truncate, 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.

When the disk is almost full, the system also support garbage collection by scanning inode table and freeing the oldest inode sublist of each current inode which discussed in layout section.

2.3 Layout

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/create** - A new inode is duplicated or created with different inode number and some other fields. But they share the same block pointers.

- **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 sublist 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

3.2 Alternative Approach

4 Conclusion

5 Acknowledgment