OPERATING SYSTEM PROJECT

Snapshots To A File System

Submitted by

Team: Geeks

NAME	BRANCH	ROLL NO
Ashwini Rangnekar	CSE	20172006
Himanshi Sharma	CSE	20172061
Parul Jain	CSE	20172107
Shipta Golechha	CSE	20172053

INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY

HYDERABAD

Under the guidance of

Mr. Manish Shrivastava	Mr. Avinash Sharma	Ms. Rupali Aher
Professor	Professor	Teaching Assistant

INTRODUCTION

In computer system a **snapshot** is the state of system at a particular point in time. A snapshot of a file system can be created to preserve the contents of the file system at a single point in time. Snapshots of the entire file system are also known as global snapshots. The storage overhead for maintaining a snapshot is keeping a copy of data blocks that would otherwise be changed or deleted after the time of the snapshot.

The snapshot function allows a backup or mirror program to run concurrently with user updates and still obtain a consistent copy of the file system as of the time that the snapshot was created. Snapshots also provide an online backup capability that allows easy recovery from common problems such as accidental deletion of a file, and comparison with older versions of a file.

Here, in this project we are maintaining snapshots by calculating differences with the help of rolling checksum as weak checksum and md5 as strong checksum.

DIFFERENT SNAPSHOT TECHNOLOGIES

- Copy-on-write: Before a write is allowed to a block, copy-on-write moves the original data block to the snapshot storage. This keeps the snapshot data consistent with the exact time the snapshot was taken. Read requests to the snapshot volume of the unchanged data blocks are redirected to the "copied" blocks in the snapshot, while read requests to active data blocks that have been changed are directed to the original volume. Snapshot contains the meta-data that describes the data blocks that have changed since the snapshot was first created. Note that original data blocks are copied only once into the snapshot storage when the first write request is received. However, this method is highly space efficient, because the storage required to create a snapshot is minimal to hold only the data that is changing. Additionally, the snapshot requires original copy of the data to be valid. IBM flash copy, AIX JFS2 snapshot, IBM SAN File System snapshot, IBM General Parallel File System snapshot, Linux Logical Volume Manager, and IBM Tivoli Storage Manager Logical Volume Snapshot Agent (LVSA) are all based on copy-on-write.
- Redirect –on-write: It offers storage space and performance efficient snapshots. New writes to the original volume are redirected to another location set aside for snapshot. The advantage of redirecting the write is that only one write takes place, whereas with copy-on-write, two writes occur (one to copy original data onto the storage space, the other to copy changed data).

- An **incremental backup** is one in which successive copies of the data contain only that portion that has changed since the preceding backup copy was made. When a full recovery is needed, the restoration process would need the last full backup plus all the incremental backups until the point of restoration. Incremental backups are often desirable as they reduce storage space usage, and are quicker to perform. The most basic form of incremental backup consists of identifying, recording and, thus, preserving only those files that have changed since the last backup. Since changes are typically low, incremental backups are much smaller and quicker than full backups. For instance, following a full back up on Friday, a Monday backup will contain only those files that changed since Friday. A Tuesday backup contains only those files that changed since Monday, and so on. A full restoration of data will naturally be slower, since all increments must be restored. Should any one of the copies created fail, including the first (full), restoration will be incomplete.

We have tried to implement copy-on-write backup strategy to take snapshots of a file system. Let's discuss this in detail:

DIRECTORY STRUCTURE

Inode:

Technically, in Unix, the term inode refers to an unnamed file in the file system, but the precise meaning can be one of two depending on context.

- First, there is the on disk data structure which contains metadata about the inode like its size and the list of blocks storing its data.
- Second, there is the in-kernel data structure, which contains a copy of the on disk structure but adds extra metadata needed within the kernel.

File systems internally refer to files and directories via inodes. Inodes are unique identifiers of the entities stored in a filesystem. Whenever an application has to operate on a file/directory (read/modify), the filesystem maps that file/directory to the right inode and start referring to that inode whenever an operation has to be performed on the file/directory.

HOW we have implemented the directory structure:

dirent.h

It is the header in C POSIX library that contains constructs that facilitate directory traversing.

The <*dirent.h*> header defines the following data type through typedef:

- DIR: A type representing a directory stream.
- Struct dirent: A structure with the following members:
  - Ino_t d_ino- file serial number
  - Char d_name[]- name of entry
- Int closedir (DIR* dirp): closes the directory stream referred to by dirp.

- DIR* opendir(const char* dirname): opens a directory named by dirname.
- Struct dirent* readdir(DIR* dirp): returns a pointer to a structure representing the directory entry at the current position in the directory stream specified by the argument dirp, and positions the directory stream at the next entry.

The rsync algorithm mentioned below takes a complete path to files as input to create a snapshot for a complete directory.

It recursively calls a function that traverse the complete directory at different levels. This functionality is implemented by applying dfs on path provided at command line. It finally gives full structure of directory to be backed up. It creates exact copy of original directory structure in destination folder mentioned by user in command line.

Also, with the help of above mentioned implementation we have extracted filename on which finally rsync algorithm has been applied.

Thus, this creates a skeleton directory structure in which files will be backed up by applying rsync algorithm mentioned below:

BACKUP ALGORITHM

Rsync algorithm is used for updating a file on one machine to be identical to a file on another machine. We assume that the two machines are connected by a low-bandwidth high-latency bi-directional communications link. The algorithm identifies parts of the source file which are identical to some part of the destination file, and only sends those parts which cannot be matched in this way. So, this algorithm is appropriate for the purpose of taking backup of a filesystem.

How Rsync detect differences?

The computer holding the newest version of the file is here called NEW and the computer holding the oldest file is called OLD.

OLD divides the oldest version of the file into blocks of, say 1024 or 2048 bytes. The file is not divided on the disk. It's just something OLD does logically, internally in the memory.

For each block OLD calculates a checksum and the list of block checksums are then sent to NEW.

NEW searches the newest version of the file for blocks of data that has the same checksum as those found in the old version of the file. This is done by fist calculating the checsum for the very first block of data (1024 or 2048 bytes). If this checksum does not match any checksum in the old file, NEW moves 1 byte down the new file and calculates the checsum for this 1024 checksum. NEW thus calculates checksums for every possible 1024 (or 2048) byte block in the new file, to search for matches to blocks in the old file.

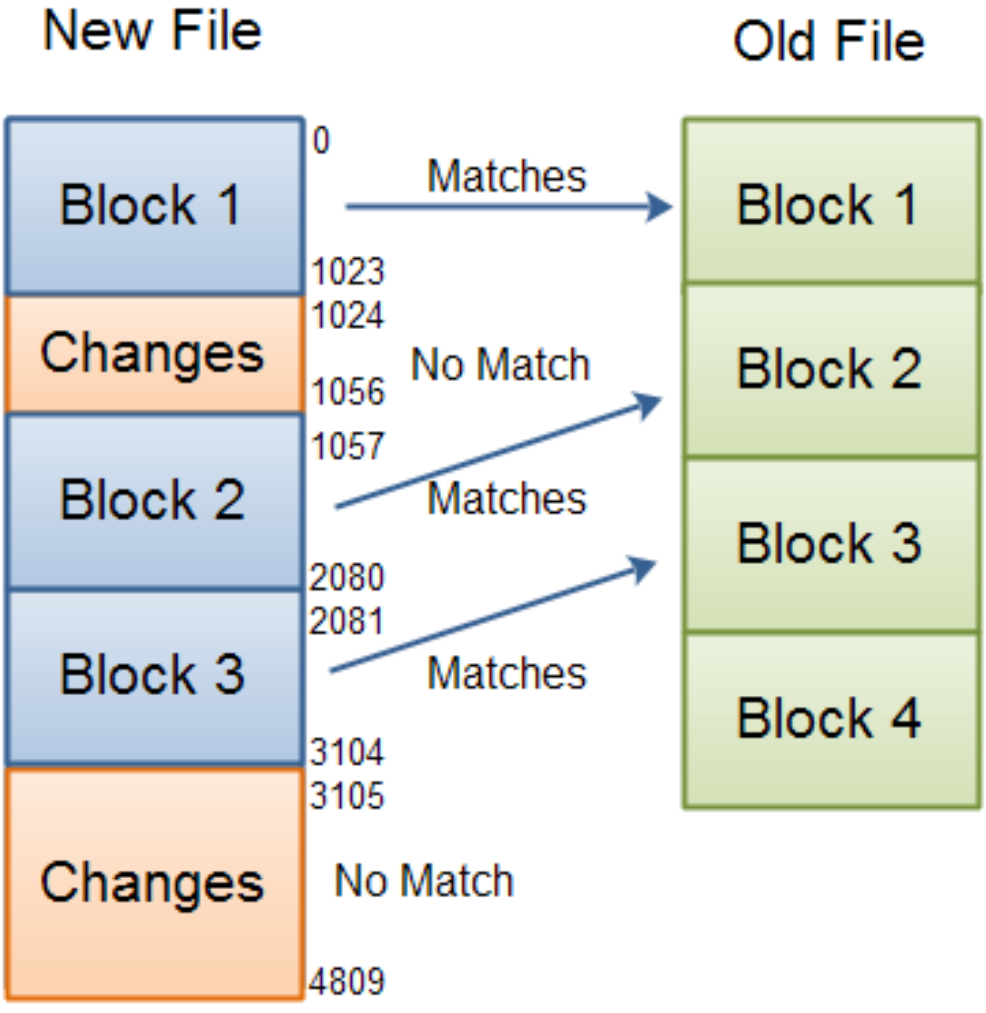
If NEW finds a 1024 byte block with the same checksum as one of the checksums received from OLD, then it considers that block to exist in the old version. It doesn't matter if the sequence of blocks is not the same as in the old version. NEW now skips to the end of this block and continues searching for checksum matches from there.

NEW will thus find X blocks of data matching checksums in the old file. This is the data that has not changed between the old and the new version of the file. In between these blocks will be data that was not part of a 1024 block that matched a checksum in the old file. This data is CHANGES, either new or modified data.

NEW sends back instructions to OLD on how to create a copy of the newest version of the file. NEW does this by sending a list of block references in the old file for the sections of the newest file that has not changed. For the parts that has changed NEW sends back the changed data in full.

OLD receives the list of block references and literal data (the changes), and from the old file and the literal data, constructs the new version of the file.

Detecting differences between files using block checksums.



Rolling Checksum

The weak rolling checksum used in the rsync algorithm needs to have the prop-erty that it is very cheap to calculate the checksum of a buffer X2...X n+1 given the checksum of buffer X1...Xn and the values of the bytes X 1 and Xn+1 .

Our checksum is defined by:

$a (k, l) = \sum (Xi) \bmod M$, where i = l to k

$s (k,l) = a (k,l)$

where s (k,l) is the rolling checksum of the bytes Xk...Xl .

For simplicity and speed, we use $M = 2^{16}$

The important property of this checksum is that successive values can be computed very efficiently using the recurrence relations

$a (k + 1, l + 1) = (a (k,l) , Xk + X l+1) \bmod M$

Snapshot Algorithm

To implement the delta snapshot technique, that is, after the first snapshot we need to only

calculate the differences between the files stored at the source and destination directories.

1. We have taken two files OLD and NEW. OLD represents the first snapshot and

NEW represents the actual data.

2. We divide the OLD file into blocks of a fixed size(say bsize= 30 bytes) and calculate checksum for every block using the given formula:

$a (k, l) = \sum (X_i) \bmod M$, where i = l to k

Then we calculate the checksum using the previous checksum value(rolling checksum).

$a (k + 1, l + 1) = (a (k,l) , X_k + X_{l+1}) \bmod M$

v=a(k+1,l+1)

Since, rolling checksum is a weak checksum so we have also calculated the md5 checksum for each block whish is a strong checksum.

3. We have send the rolling checksum for every block to the rch(NEW,rchk,bsize,len).

4. In the rch() we parse the NEW file and calculate its checksum as follows:-

initially flag=0;

case 1: if(flag==0)

$a (k, l) = \sum (X\ i) \bmod M$, where i = l to k

v=a(k,l)

case 2: if(flag==1)

Then we calculate the checksum using the previous checksum value(rolling

checksum).

a (k + 1, l + 1) = (a (k,l) , X k + X l+1) mod M

v=a(k+1,l+1)

5. Now, we will maintain a recovery string which will contain block references to the

data common between OLD and NEW and the "modified data" for the unmatched

parts of the two strings.

recovery string (str) = references (matched data) + data (unmatched data)

6. We will match v with the entries in rchk :

case 1: if v == rchk[i], then we will also check if their md5 checksums are matching or not if they are also matching then ,

set flag=0

k=k+bsize

push i into a queue, where i denotes the reference to the block matched between

OLD and NEW

str = str+i

case 2: if v does not match any of the enteries in rchk, then

flag=1

str = str+NEW[k]

k=k+1

7. So, finally we get the recovery string(str) , which can be used to update the OLD string using only the differences.

SUMMARY

To summarize, snapshot is a technique to maintain consistency of a file system. With the help of snapshot we create a backup copy, and write back only the differences between the backed up data and modified data.

We have tried to make a similar snapshot and recovery technique which uses rsync algorithm. It calculates the differences between the two files and generates a recovery string which contains the modified data (unmatched data) and the block references to the blocks corresponding to the matched data. We use this recovery string to recover file and reflect back the changes made in the live data to the backed up data.