Versioning File System

Contents

File System Overview

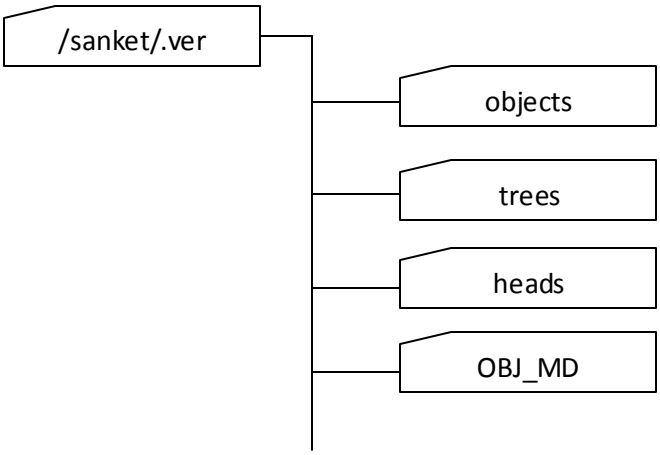
Versioning

Versioning file system is supposed to provide the basic functions of a versioning system, but on the same hand free the user from taking the 'tough' decisions about doing the Branching, Committing and provide an easy to Checkout Plan. The ideal characteristics of a file system according to me will be:

1. Commit a new version on the present branch when some file changes
2. Supply default tagging using time stamps, for user to checkout old versions
3. Provide branching more naturally, without the user worrying about it
4. Provide the actual semantics of Checkout, Revert, Branching and Commit
5. Efficient way to store version metadata, and policy for version compression
6. Version Compression may be lossy

We will look at how our VFS(Versioning File System) takes care of the above tasks, the required data structures and the file hierarchy required.

Directory Structure



The above diagram analyses the basic directory structure of our versioning file system. We now describe the utility of each of the directory/file and their semantic.

Objects

Raw data in VFS is stores in the form of Objects. There are majorly 2 types of Objects that we store.

- Loose Objects: These are the actual files stored in a compressed format(gzipped)
- Packed Objects: They are diff objects, that is difference b/w two files

Object directory stores all Loose Objects and Pack Objects for files inside the referenced directory. For example in the above case, / is the root mount(it can be something like /home/sanket/mount in real scenarios, but I call it simply /). /sanket/.ver is the versioning metadata for all files inside /sanket. Hence all versioning information about /sanket/<files> will be stores strictly in /sanket/.ver folder!

Storage

Objects are all stored by their hash values. Suppose there is a file, /sanket/heloo.txt containing

Heloo.txt

Hello world text file

Then the loose object will be the SHA-1 of the contents the file, and the Loose Object in this case will be named SHA-1(Hello.txt). This helps us to reuse same data over multiple files without replicating(a rare use case though)

Trees

Trees folder(/sanket/.ver/trees) stores the hierarchy tree for each file inside /sanket/<file>. Because we are concentrating on a 'versioning file system' the versions need to be maintained for each file separately. One one hand that is more overhead, but less implementation issues.

Trees files will be described in more detail in Data Structures

Heads

Heads folder(/sanket/.ver/heads/) stores the present branch heads as well as the present working head in a single file, it helps to perform the various Versioning Functions.

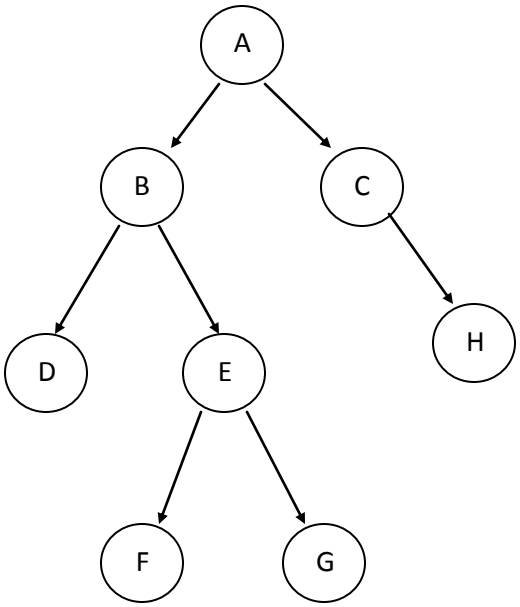
Heads files will be explained in more detail in Data Structures

OBJ_MD

OBJ_MD is a single file, which is a map between the objects and the reference count. Reference Count is simply the number of times an object occurs inside each of the tree of each file, all summed up. Though this looks tricky, maintaining it is rather trivial.

Data Structures

A typical use File Hierarchy

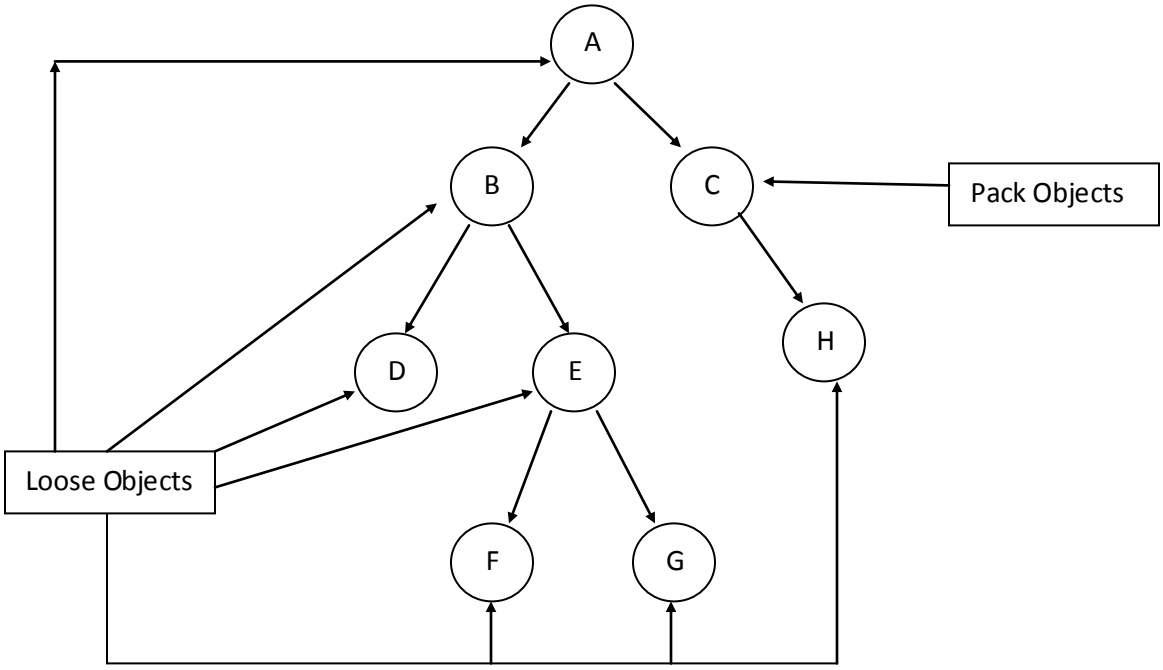


The above tree shows typically how a file might grow. Thus it is a snapshot of a single tree version! Let's understand what is happening:

- The root A may be the first version that existed
- User made changes and version B was committed
- But then he decided to checkout A and branch. Version C was created
- He did not like C right now, he went to B and saved D
- Again he goes back to B and saves new branch E
- … and so on …

Interestingly the above data structure encapsulates all we want to know about a single file versioning, henceforth we will be using a tree structure(assuming no merges), for our file system too.

But in order to use such a structure, we typically would need to store the tree somewhere. Hence we now start defining out data structures and the actual semantics of the .ver/ metadata.



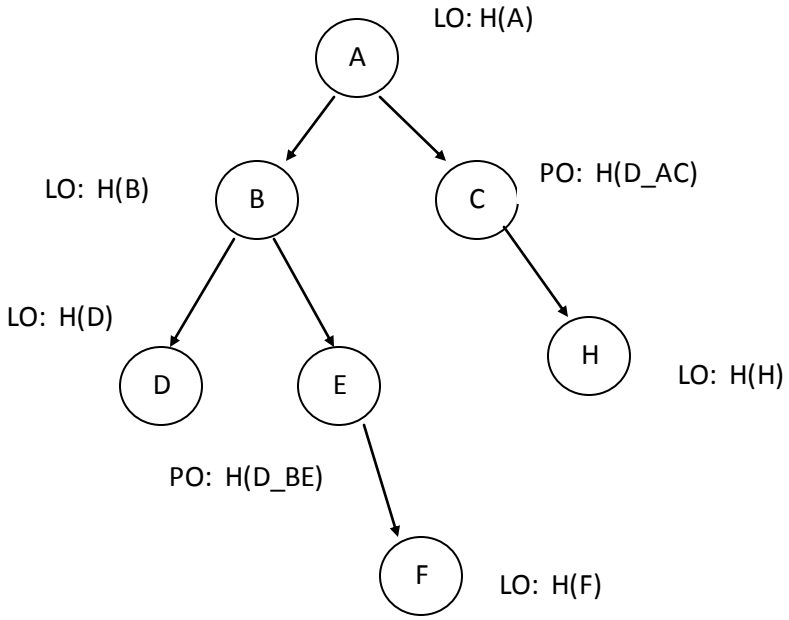
Terminologies

- Leaf Nodes: Show all the possible branch heads
- Junction: A, B and E in this case. All nodes which have branches diverging
- Stem Nodes/Stems: Sequence of nodes forming a single branch. A-C-H only in this case

Interestingly what follows from the above discussion is, the following invariant:

*Theorem: All Junctions and Leaf Nodes will *have* to contain the Loose Object corresponding to that version!*

Thus, it means that except C, everything is a junction or a leaf node(cannot be both)and hence only C can store a diff object whereas all others will store a Loose Object for the version at it.



Versioning of /sanket/file.txt

In the above figure LO represent the Loose Objects, and PO depict the packed objects. Let's now try to understand how to build the basic blocks of versioning metadata.

Tree structure

[File: .ver/tree/file.txt.tree, sizeof(Metadata+Parent Pointer) = s]

Offset	Node	Metadata	Parent Pointer
0	A	xxxxx	-1
S	B	xxxxx	0
2S	C	xxxxx	0
3S	D	xxxxx	S
4S	E	xxxxx	S
5S	F	xxxxx	4S
6S	H	xxxxx	2S

Note that Node and offset are for depiction, they are not stored in the file

Explanation

- Assume the order of insertion of nodes be, A,B,C,D,E,F,H. The nodes are nothing but versions
- When A is added, it corresponds to new file add event, hence file.txt.tree is constructed afresh. With first entry pointing to A. No parent hence -1
- When B is added we know the HEAD(explained later) and hence just put the metadata of B along with the parent, which is A
- Similarly for C, D, E, F, H.
- See that each of the nodes maintains their parents.

Metadata structure

As metadata will store the versioning specific information relevant to the end user:

- Timestamp
- Tag(if present)
- Hash of the object
- Type of object

In C structure

```
#define LO 0

#define PO 1

typedef struct _metadata_ts{

 char ts[MAX_TS];

 char tag[MAX_TAG];

 char obj_hash[HASH_SIZE];

 bool obj_type; // LO or PO

}
```

Heads Structure

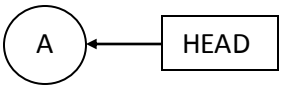
The format for the generic heads/file.head structure:

Branch Name	Offset in file.tree(S)
B1	PRESENT_HEAD
B2	BRANCH1 HEAD
B3	BRANCH2 HEAD
….	….

Hence:

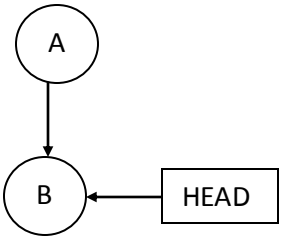
- The first row represents simply where the present head is
- The second row onwards represent all branch heads that are present currently

Let's try to simulate this table generation for the above sequence A,B,C,H of instructions



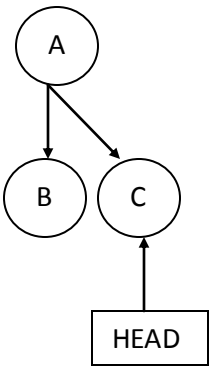
BranchName	Offset in file.tree(S)
B1	0
B1	0

Offset	Node	Metadata	Parent Pointer
0	A	xxxxx	-1



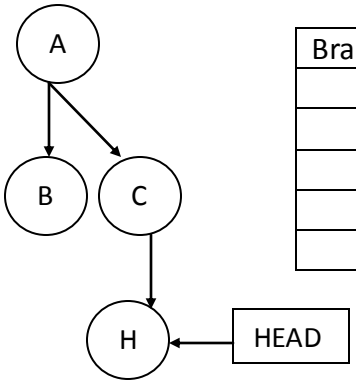
BranchName	Offset in file.tree(S')
B2	S
B1	INV
B2	S

Offset	Node	Metadata	Parent Pointer
0	A	xxxxx	-1
S	B	xxxxx	0



BranchName	Offset in file.tree(S')
B3	2S
B1	INV
B2	S
B3	2S

Offset	Node	Metadata	Parent Pointer
0	A	xxxxx	-1
S	B	xxxxx	0
2S	C	xxxxx	0



BranchName	Offset in file.tree(S')
B4	2S
B1	INV
B2	S
B3	INV
B4	3S

Offset	Node	Metadata	Parent Pointer
0	A	xxxxx	-1
S	B	xxxxx	0
2S	C	xxxxx	0
3S	H	xxxxx	2S

Object Store

The object store is populated in the following manner:

- Each object that is created, whether a LO or a PO is a file
- Each such object is hashed with it's contents, then gzipped and stored with HASH value as the file name
- HASH used is SHA-1 [http://library.gnome.org/devel/glib/2.16/glib-Data-Checksums.html]
- When storing a new object, you apply the following routine:
    - Find x = SHA1(obj)
    - If x not in store:
        - Store the object
    - Else
        - Don't store

OBJ_MD

File: .ver/OBJ_MD

The format for this file is:

SHA1(Obj)	Ref Count
SHA1(A)	1
SHA1(B)	1
SHA1(C)	2

An example OBJ_MD file

You may use the glib hash table implementation
[http://library.gnome.org/devel/glib/unstable/glib-Hash-Tables.html]

Versioning Operations

Add new version

The following is the workflow for adding a new version(for file heloo.txt):

- Read the present HEAD from heads/heloo.txt.head
- Add a new entry to tree/heloo.txt.tree with hash of the current version data(PO/LO)
- Update the HEAD, add the current head also as a branch. Invalidate the parent in heads/hello.txt.head
- Update OBJ_MD and increment the ref_count

The RAW file operations will be:

- Finding the diff in the right manner
- Finding the SHA1
- Making sure the right version is in the working directory

Checkout

Checking out a version has the following semantics:

- Checkout is always in the same branch
- When you checkout, your head should point to the given checkout point
- The working copy should be the checkout out copy

Workflow

- Read the present HEAD from heads/heloo.txt.head
- You know where to jump(either a HASH of the object or a TAG)
- Jump to the position in tree/heloo.txt.tree where the HEAD points
- Now make your way to the root(just traverse using the parent pointer)
- If you reach the root without seeing the checkpoint => incorrect checkpoint
- Else, you reach the checkpoint
- Change the HEAD

RAW File Operations

- Checkout the file applying the Diffs in a proper manner
- Make sure you start applying diff from the closes LO position

Revert

Revert operation takes you back into history, with removal of all those objects which are no longer referenced/needed. The following is the workflow(heloo.txt):

- Read the HEAD from heads/heloo.txt.head
- Just as in Checkout try to reach the destination timestamp(tag)
- If tag/ts not found then do not revert
- Else go till the first junction is reached
- Decrement the count of the objects encountered and delete if ref count = 0
- Checkout from the first junction reached to the timestamp asked

Change Branch and Delete Version

Do as for above operations, fill it in yourself.