Fossil, an Archival File Server

Sean Quinlan
Jim McKie
Russ Cox
jmk,rsc@plan9.bell-labs.com

ABSTRACT

This paper describes the internals and operation of Fossil, an archival file server built for Plan 9. Fossil has not yet replaced the current Plan 9 file server and `kfs`, but that is our eventual intent. Both fossil and this documentation are works in progress. Comments on either are most welcome.

1. Introduction

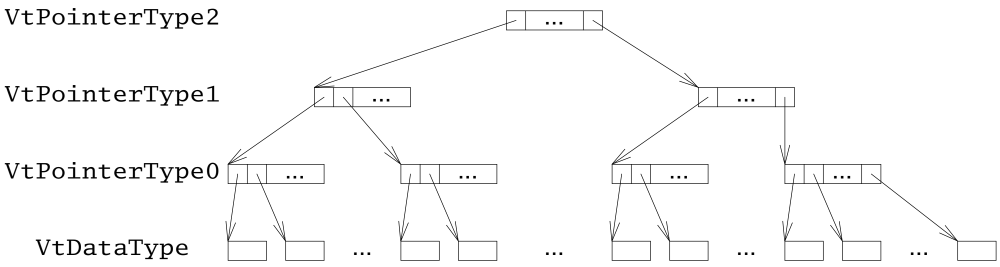
Fossil is an archival file server built for Plan 9. In a typical configuration, it maintains a traditional file system in a local disk partition and periodically archives snapshots of the file system to a Venti server. These archives are made available through a file system interface. Fossil can also be run without a Venti server, in which case the snapshots (if any) occupy local disk space.

The bulk of this paper explains the underlying data structures: Venti trees, the Venti archival file system format, and finally Fossil's file system format. The end of the paper discusses the architecture of the Fossil server.

The presentation of the data structures is very detailed, perhaps too detailed for most readers. The intent is to record all the details necessary to make structural changes to the file system format. Feel free to jump ahead when boredom strikes.

2. Venti trees and directory hierarchies

Venti [3] is an archival block storage server. Once a block is stored, it can be retrieved by presenting the 20-byte SHA1 hash of its contents, called a *score*. Blocks on Venti have a maximum length of about 56 kilobytes, though in practice smaller blocks are used. To store a byte stream of arbitrary length, Venti uses a hash tree. Conceptually, the data stream is broken into fixed-size (say, *dsize*-byte) chunks, which are stored on the Venti server. The resulting scores are concatenated into a new pointer stream, which is broken into fixed size (say, *psize*-byte) chunks, which are stored on the Venti server. (*Psize* is different from *dsize* so that we can ensure that each pointer block holds an integral number of pointers.) This yields a new pointer stream, and so on, until there is a single block and finally a single score describing the entire tree. The resulting structure looks like:



The leaves are the original data stream. Those blocks have type `VtDataType`. The first pointer stream has type `VtPointerType0`, the next has type `VtPointerType1`, and so on. The figure ends with a single block of type `VtPointerType2`, but in general trees can have height up to `VtPointerType6`. For a *dsize* of 8192 bytes and *psize* of 8180 bytes (409 pointers), this gives a maximum stream size of approximately 10 zettabytes (2^{73} or 10^{22} bytes).

Data block are truncated to remove trailing runs of zeros before storage to Venti; they are zero-filled back to *dsize* bytes after retrieval from Venti. SImilarly, trailing runs of pointers to zero-length blocks are removed from and added back to pointer blocks. These simple rules happen to make it particularly efficient to store large runs of zeros, as might occur in a data stream with ''holes:'' the zero-length block itself can be interpreted as a tree of any depth encoding an all-zero data stream.

Reconstructing the data stream requires the score and type of the topmost block in the tree, the data chunk size, the pointer chunk size, and the data stream size. (From the data stream size and the chunk sizes we could derive the depth of the tree and thus the type of the topmost block, but it is convenient to allow trees that are deeper than necessary.) This information is kept in a 40-byte structure called a `VtEntry`:

```
VtEntry:
    gen[4]        generation number
    psize[2]      size of pointer blocks
    dsize[2]      size of data blocks
    flags[1]
    zero[5]
    size[6]       length of file
    score[20]     score of root block in tree
```

(In this notation, `name[sz]` indicates a `sz`-byte field called `name`. Integers are stored in big-endian order. `Size` really is a 48-bit field.) `Flags` is made up of the following bit fields. '*or*' of the following flags:

```
0x01  VtEntryActive         entry is allocated
0x02  VtEntryDir            entry describes a Venti directory (q.v.)
0x1C  VtEntryDepthMask      mask for tree depth
0x20  VtEntryLocal          reserved (q.v.)
```

The depth of the described tree is stored in the 5 bits indicated: a tree with a topmost node of type `VtPointerType3` has depth 4.

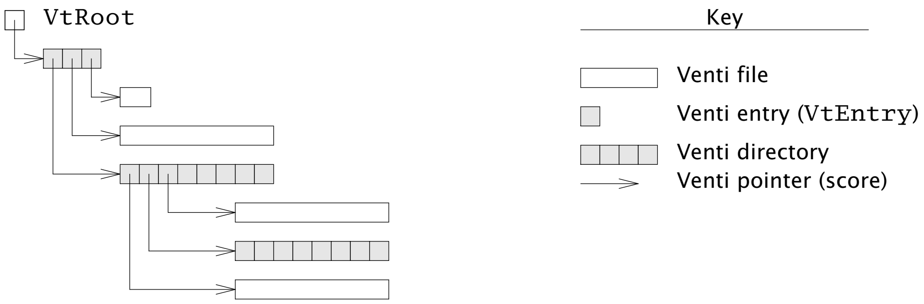
With `VtEntry` we can build more complicated data structures, ones with multiple or nested data streams. A data stream consisting of `VtEntry` structures is called a Venti directory. It is identical in structure to the Venti data stream we described earlier except that the bottom-level type is `VtDirType`, and the `VtEntry` describing a Venti directory has the `VtEntryDir` flag bit set. The *dsize* for a Venti directory is a multiple of 40 so that each data chunk holds an integer number of `VtEntry` structures. By analogy with Venti directories, we call the original data stream a Venti file. Note that Venti files are assumed *not* to contain pointers to other Venti blocks. The only pointers to Venti blocks occur in `VtEntry` structures in Venti directories (and in the internal hash tree structure of the individual files and directories). Note also that these directories are nothing more than pointer lists. In particular, there are no names or metadata like in a file system.

To make it easier to pass hierarchies between applications, the root of a hierarchy is described in a 300-byte structure called a `VtRoot`:

```
VtRoot:
    version[2]       2
    name[128]        name of structure (just a comment)
    type[128]        string describing structure (vac)
    score[20]        pointer to VtDirType block
    blockSize[2]     maximum block size in structure
    prev[20]         previous VtRoot in chain, if any
```

This structure is stored to Venti and its score is passed between applications, typically in the form ‘‘*type*:*rootscore*,’’ where *type* is the type field from the `VtRoot` structure, and *rootscore* is the score of the `VtRoot` block. `VtRoot` structures can be chained together using the *prev* field to encode an archival history of the data structure.

For example, a small Venti hierarchy might look like:



Venti files are shown as white boxes, while directories are shown as shaded boxes. Each shaded square represents a `VtEntry`. Arrows represent pointers from `VtEntry` structures to other Venti files or directories.

The hierarchical structure provided by Venti files and directories can be used as the base for more complicated data structures. Because this structure captures all the information about pointers to other blocks, tools written to traverse Venti hierarchies can traverse the more complicated data structures as well. For example, *venti/copy* (see *ventiaux*(8)) copies a Venti hierarchy from one Venti server to another, given the root `VtEntry`. Because the traditional file system described in later sections is layered on a Venti hierarchy, *venti/copy* can copy it without fully understanding its structure.

3. Vac file system format

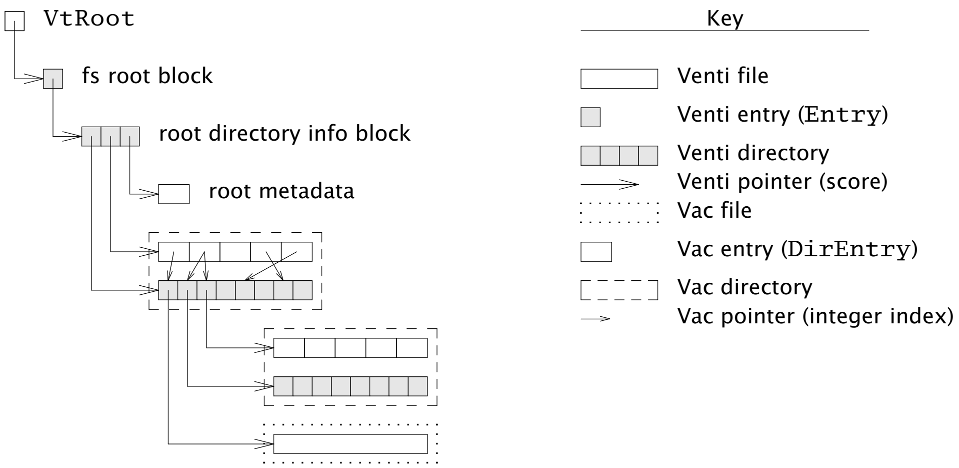
The Venti archive format *vac* builds a traditional file system using a Venti hierarchy. Each vac file is implemented as a Venti file; each vac directory is implemented as a Venti directory and a Venti file to provide traditional file system metadata. The metadata is stored in a structure called a `DirEntry`:

```
DirEntry:
    magic[4]     0x1c4d9072 (DirMagic)
    version[2]   9
    elem[s]      name (final path element only)
    entry[4]     entry number for Venti file or directory
    gen[4]       generation number
    mentry[4]    entry number for Venti file holding metadata
    mgen[4]      generation number
    qid[8]       unique file serial number
    uid[s]       owner
    gid[s]       group
    mid[s]       last modified by
    mtime[4]     last modification time
    ctime[4]     creation time
    atime[4]     last access time
    mode[4]      mode bits
```

The notation `name[s]` denotes a string stored as a two-byte length and then that many bytes. The above describes Version 9 of the `DirEntry` format. Versions 7 and 8 are very similar; they can be read by the current *vac* source code but are not written. Earlier versions were not widespread. A `DirEntry` may be followed by optional extension sections, though none are currently used. The `mode` bits include bits commonly used by Unix and Windows, in addition to those used by Plan 9.

The `entry` field is an index into the parallel Venti directory. The `gen` field must match the `gen` field in the corresponding `VtEntry` in the directory; it is used to detect stale indices. Similarly, `mentry` and `mgen` are the index and generation number for the metadata Venti file, if the `DirEntry` describes a vac directory.

The relation between Venti files and directories and vac files and directories can be seen in this figure:



In reality, the story is slightly more complicated. The metadata file in a Vac directory is not just the concatenation of `DirEntry` structures. Instead, it is the concatenation of `MetaBlocks`. A `MetaBlock` contains some number of `DirEntry` structures along with a sorted index to make it easy to look for a particular `DirEntry` by its `elem` field. The details are in the source code.

As shown in the diagram, the root directory of the file system is summarized by three `VtEntry` structures describing the Venti directory for the children of the root, the Venti file for the metadata describing the children of the root, and a Venti file holding metadata for the root directory itself. These `VtEntry` structures are placed in a Venti directory of their own, described by the single `VtEntry` in the root block.

4. Fossil file system format

Fossil uses the vac format, with some small changes. The changes only affect the data on the local disk; the data archived to Venti is exactly in vac format.

Blocks stored on local disk may contain scores pointing at local disk blocks or at Venti blocks. Local block addresses are stored as 20-byte scores in which the first 16 bytes are all zero and the last 4 bytes specify a block number in the disk. Before a block is archived, all the blocks it points to must be archived, and the local scores in the block must be changed to Venti scores. Using block addresses rather than content hashes for local data makes the local file system easier to manage: if a local block's contents change, the pointer to the block does not need to change.

4.1. Snapshots

Fossil is an archival file server. It takes periodic snapshots of the file system, which are made accessible through the file system. Specifically, the active file system is presented in `/active`. Ephemeral snapshots (those that are kept on local disk and eventually deleted) are presented in `/snapshot/`*yyyy*`/`*mmdd*`/`*hhmm*, where *yyyy* is the full year, *mm* is the month number, *dd* is the day number, *hh* is the hour, and *mm* is the minute. Archival snapshots (those that are archived to Venti and persist forever) are presented in `/archive/`*yyyy*`/`*mmdds*, where *yyyy*, *mm*, and *dd* are year, month, and day as before, and *s* is a sequence number if more than one archival snapshot is done in a day. For the first snapshot, *s* is null. For the subsequent snapshots, *s* is `.1`, `.2`, `.3`, etc.

To implement the snapshots, the file server maintains a current *epoch* for the active file system. Each local block has a label that records, among other things, the epoch in which the block was allocated. If a block was allocated in an epoch earlier than the current one, it is immutable and treated as copy-on-write. Taking a snapshot can be accomplished by recording the address of the current root block and then incrementing the epoch number. Notice that the copy-on-write method makes snapshots both time efficient and space efficient. The only time cost is waiting for all current file system requests to finish and then incrementing a counter. After a snapshot, blocks only get copied when they are next modified, so the per-snapshot space requirement is proportional to the amount of new data rather than the total size of the file system.

The blocks in the archival snapshots are moved to Venti, but the blocks in the ephemeral snapshots take up space in the local disk file. To allow reclamation of this disk space, the file system maintains a *low epoch*, which is the epoch of the earliest ephemeral snapshot still available. Fossil only allows access to snapshots with epoch numbers between the low epoch and the current epoch (also called the high epoch). Incrementing the low epoch thus makes old snapshots inaccessible. The space required to store those snapshots can then be reclaimed, as described below.

4.2. Local blocks

The bulk of the local disk file is the local blocks. Each block has a 14-byte label associated with it, of the format:

```
Label:
    state[1]          block state
    type[1]           block type
    epoch[4]          allocation epoch
    epochClose[4]     close epoch
    tag[4]            random tag
```

The `type` is an analogue of the block types described earlier, though different names are used, to distinguish between pointers blocks in a hash tree for a data stream and pointer blocks for a directory stream. The `epoch` was mentioned in the last section.

The other fields are explained below.

There are two distinguished blocks states `BsFree` (`0x00`) and `BsBad` (`0xFF`), which mark blocks that are available for allocation and blocks that are bad and should be avoided. If `state` is not one of these values, it is a bitwise '*or*' of the following flags:

`0x01`	`BsAlloc`	block is in use
`0x02`	`BsCopied`	block has been copied
`0x04`	`BsVenti`	block has been stored on Venti
`0x08`	`BsClosed`	block has been unlinked from active file system

The flags are explained as they arise in the discussions below.

It is convenient to store some extra fields in the `VtEntry` structure when it describes a Venti file or directory stored on local disk. Specifically, we set the `VtEntryLocal` flag bit and then use the bytes 7-16 of the score (which would otherwise be zero, since it is a local score) to hold these fields:

`archive[1]`	boolean: this is an archival snapshot
`snap[4]`	epoch number if root of snapshot
`tag[4]`	random tag

The extended `VtEntry` structure is called an `Entry`. The `tag` field in the `Label` and the `Entry` is used to identify dangling pointers or other file system corruption: all the local blocks in a hash tree must have tags matching the tag in the `Entry`. If this `Entry` points at the root of a snapshot, the `snap` field is the epoch of the snapshot. If the snapshot is intended to be archived to Venti, the `archive` field is non-zero.

4.3. Block reclamation

The blocks in the active file system form a tree: each block has only one parent. Once a copy-on-write block *b* is replaced by its copy, it is no longer needed by the active file system. At this point, *b* is unlinked from the active file system. We say that *b* is now *closed*: it is needed only for snapshots. When a block is closed, the `BsClosed` bit is set in its state, and the current epoch (called the block's closing epoch) is stored in the `epochClose` label field. (Open blocks have an `epochClose` of ~0).

A block is referenced by snapshots with epochs between the block's allocation epoch and its closing epoch. Once the file system's low epoch grows to be greater than or equal to the block's closing epoch, the block is no longer needed for any snapshots and can be reused.

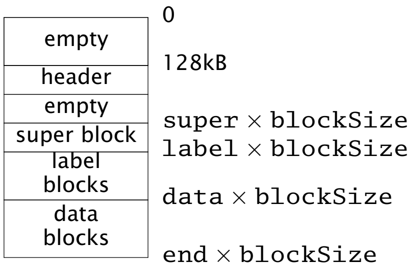
In a typical configuration, where nightly archival snapshots are taken and written to Venti, it is desirable to reclaim the space occupied by now-archived blocks if possible. To do this, Fossil keeps track of whether the pointers in each block are unique to that block. When a block *bb* is allocated, a pointer to *bb* is written into exactly one active block (say, *b*). In the absence of snapshots, the pointer to *bb* will remain unique to *b*, so that if the pointer is zeroed, *bb* can be immediately reused. Snapshots complicate this invariant: when *b* is copied-on-write, all its pointers are no longer unique to it. At time of the copy, the `BsCopied` state bit in the block's label is set to note the duplication of the pointers contained within.

4.4. Disk layout

The file system header describes the file system layout and has this format:

```
Header:
    magic[4]          0x3776AE89 (HeaderMagic)
    version[2]        1 (HeaderVersion)
    blockSize[2]      file system block size
    super[4]          block offset of super block
    label[4]          block offset of labels
    data[4]           data blocks
    end[4]            end of file system
```

The corresponding file system layout is:



The numbers to the right of the blocks are byte offsets of the boundaries.

The super block describes the file system itself and looks like:

```
Super:
    magic[4]          0x2340A3B1 (SuperMagic)
    version[2]        1 (SuperVersion)
    epochLow[4]       file system low epoch
    epochHigh[4]      file system high (active) epoch
    qid[8]            next qid to allocate
    active[4]         data block number: root of active file system
    next[4]           data block number: root of next file system to archive
    current[4]        data block number: root of file system currently being archived
    last[20]          Venti score of last successful archive
    name[128]         name of file system (just a comment)
```

5. Fossil server

The Fossil server is a user-space program that runs on a standard Plan 9 kernel.

5.1. Process structure

The file server is structured as a set of processes synchronizing mostly through message passing along queues. The processes are given names, which can be seen in the output of `ps -a`.

`Listen` processes announce on various network addresses. A `con` process handles each incoming connection, reading 9P requests and adding them to a central message queue. `Msg` processes remove 9P requests from the queue, handle them, and write the responses to the appropriate file descriptors.

The `disk` process handles disk I/O requests made by the other processes. The `flush` process writes dirty blocks from the in-memory block cache to disk. The `unlink` process frees previously linked blocks once the blocks that point at them have been written to disk.

A `consI` reads from each console file (typically a pipe posted in `/srv`), adding the typed characters to the input queue. The `cons` process echoes input and runs the commands, saving output in a ring buffer. Because there is only one `cons` process, only one console command may be executing at a time. A `consO` process copies this ring buffer to the each console file.

The `periodic` process runs periodic events, like flushing the root metadata to disk or taking snapshots of the file system.

5.2. Block cache

Fossil maintains an in-memory block cache which holds both local disk blocks and Venti blocks. Cache eviction follows a least recently used policy. Dirty blocks are restricted to at most half the cache. This can be changed by editing `DirtyPercentage` in `dat.h`.

The block cache uses soft updates [1] to ensure that the on-disk file system is always self-consistent. Thus there is no *halt* console command and no need to check a file system that was shut down without halting.

5.3. Archiving

A background process writes blocks in archival snapshots to Venti. Although `/archive/`*yyyy*`/`*mmdds* is a copy of only `/active` at the time of the snapshot, the archival process archives the entire file tree rather than just the subtree rooted at `/active`. The snapshots `/snapshot/`*yyyy*`/`*mmdd*`/`*hhmm* are stored as empty directories. Once all the blocks have been archived, a `VtRoot` header for the file system is archived. The score of that header is recorded in `super.score` and also printed on the file server console. The score can used by *flfmt* to restore a file system (see *fossil*(4)).

5.4. Contrast with the old file server

The most obvious difference between Fossil and the old Plan 9 file server [2] is that Fossil uses a Venti server as its archival storage in place of a WORM juke box. There are a few other architectural differences to be aware of.

Fossil is a user-level program run on a standard kernel.

Fossil does not have any way to concatenate, stripe, or mirror disk files. For functionality similar to the old file server's configuration strings, use the experimental file stack device (see *fs*(3)).

Fossil speaks only 9P2000. Old 9P (aka 9P1) is not supported.

6. References

[1] Gregory R. Ganger, Marshall Kirk McKusick, Craig A. N. Soules, and Yale N. Patt. ''Soft Updates: A Solution to the Metadata Update Problem in File Systems,'' *ACM Transactions on Computer Systems*, Vol 18., No. 2, May 2000, pp. 127-153.

[2] Sean Quinlan, ''A Cached WORM File System,'' *Software—Practice and Experience*, Vol 21., No 12., December 1991, pp. 1289-1299.

[3] Sean Quinlan and Sean Dorward, ''Venti: A New Approach to Archival Storage,'' *Usenix Conference on File and Storage Technologies*, 2002.