CEPHFS SNAPSHOTS

CephFS supports snapshots, generally created by invoking mkdir against the (hidden, special) .snap directory.

OVERVIEW

Generally, snapshots do what they sound like: they create an immutable view of the filesystem at the point in time they're taken. There are some headline features that make CephFS snapshots different from what you might expect:

- Arbitrary subtrees. Snapshots are created within any directory you choose, and cover all data in the filesystem under that directory.
- Asynchronous. If you create a snapshot, buffered data is flushed out lazily, including from other clients. As a result, "creating" the snapshot is very fast.

IMPORTANT DATA STRUCTURES

- SnapRealm: A *SnapRealm* is created whenever you create a snapshot at a new point in the hierarchy (or, when a snapshotted inode is moved outside of its parent snapshot). SnapRealms contain an *sr_t srnode*, links to *past_parents* and *past_children*, and all *inodes_with_caps* that are part of the snapshot. Clients also have a SnapRealm concept that maintains less data but is used to associate a *SnapContext* with each open file for writing.
- sr_t: An *sr_t* is the on-disk snapshot metadata. It is part of the containing directory and contains sequence counters, timestamps, the list of associated snapshot IDs, and *past_parents*.
- snaplink_t: *past_parents* et al are stored on-disk as a *snaplink_t*, holding the inode number and first *snapid* of the inode/snapshot referenced.

CREATING A SNAPSHOT

Because CephFS snapshot currently is an experimental feature, we are supposed to enable it explicitly by the command below before testing.

```
$ ceph fs set <fs_name> allow_new_snaps true --yes-i-really-mean-it
```

To make a snapshot on directory "/1/2/3/foo", the client invokes "mkdir" on "/1/2/3/foo/.snap" directory. This is transmitted to the MDS Server as a CEPH_MDS_OP_MKSNAP-tagged *MClientRequest*, and initially handled in Server::handle_client_mksnap(). It allocates a *snapid* from the *SnapServer*, projects a new inode with the new SnapRealm, and commits it to the MDLog as usual. When committed, it invokes *MDCache::do_realm_invalidate_and_update_notify()*, which triggers most of the real work of the snapshot.

If there were already snapshots above directory "foo" (rooted at "/1", say), the new SnapRealm adds its most immediate ancestor as a *past_parent* on creation. After committing to the MDLog, all clients with caps on files in "/1/2/3/foo/" are notified (MDCache::send_snaps()) of the new SnapRealm, and update the *SnapContext* they are using with that data. Note that this *is not* a synchronous part of the snapshot creation!

UPDATING A SNAPSHOT

If you delete a snapshot, or move data out of the parent snapshot's hierarchy, a similar process is followed. Extra code paths check to see if we can break the *past_parent* links between SnapRealms, or eliminate them entirely.

GENERATING A SNAPCONTEXT

A RADOS *SnapContext* consists of a snapshot sequence ID (*snapid*) and all the snapshot IDs that an object is already part of. To generate that list, we generate a list of all *snapids* associated with the SnapRealm and all its *past_parents*.

STORING SNAPSHOT DATA

File data is stored in RADOS “self-managed” snapshots. Clients are careful to use the correct *SnapContext* when writing file data to the OSDs.

STORING SNAPSHOT METADATA

Snapshotted dentries (and their inodes) are stored in-line as part of the directory they were in at the time of the snapshot. *All dentries* include a *first* and *last* snapid for which they are valid. (Non-snapshotted dentries will have their *last* set to CEPH_NOSNAP).

SNAPSHOT WRITEBACK

There is a great deal of code to handle writeback efficiently. When a Client receives an *MClientSnap* message, it updates the local *SnapRealm* representation and its links to specific *Inodes*, and generates a *CapSnap* for the *Inode*. The *CapSnap* is flushed out as part of capability writeback, and if there is dirty data the *CapSnap* is used to block fresh data writes until the snapshot is completely flushed to the OSDs.

In the MDS, we generate snapshot-representing dentries as part of the regular process for flushing them. Dentries with outstanding *CapSnap* data is kept pinned and in the journal.

DELETING SNAPSHOTS

Snapshots are deleted by invoking “rmdir” on the “.snap” directory they are rooted in. (Attempts to delete a directory which roots snapshots *will fail*; you must delete the snapshots first.) Once deleted, they are entered into the *OSDMap* list of deleted snapshots and the file data is removed by the OSDs. Metadata is cleaned up as the directory objects are read in and written back out again.

HARD LINKS

Hard links do not interact well with snapshots. A file is snapshotted when its primary link is part of a SnapRealm; other links *will not* preserve data. Generally the location where a file was first created will be its primary link, but if the original link has been deleted it is not easy (nor always determnistic) to find which link is now the primary.

MULTI-FS

Snapshots and multiiple filesystems don’t interact well. Specifically, each MDS cluster allocates *snapids* independently; if you have multiple filesystems sharing a single pool (via namespaces), their snapshots *will* collide and deleting one will result in missing file data for others. (This may even be invisible, not throwing errors to the user.) If each FS gets its own pool things probably work, but this isn’t tested and may not be true.