RBD LAYERING

RBD layering refers to the creation of copy-on-write clones of block devices. This allows for fast image creation, for example to clone a golden master image of a virtual machine into a new instance. To simplify the semantics, you can only create a clone of a snapshot - snapshots are always read-only, so the rest of the image is unaffected, and there's no possibility of writing to them accidentally.

From a user's perspective, a clone is just like any other rbd image. You can take snapshots of them, read/write them, resize them, etc. There are no restrictions on clones from a user's viewpoint.

Note: the terms *child* and *parent* below mean an rbd image created by cloning, and the rbd image snapshot a child was cloned from.

COMMAND LINE INTERFACE

Before cloning a snapshot, you must mark it as protected, to prevent it from being deleted while child images refer to it:

```
$ rbd snap protect pool/image@snap
```

Then you can perform the clone:

```
$ rbd clone [--parent] pool/parent@snap [--image] pool2/child1
```

You can create a clone with different object sizes from the parent:

```
$ rbd clone --order 25 pool/parent@snap pool2/child2
```

To delete the parent, you must first mark it unprotected, which checks that there are no children left:

```
$ rbd snap unprotect pool/image@snap
Cannot unprotect: Still in use by pool2/image2
$ rbd children pool/image@snap
pool2/child1
pool2/child2
$ rbd flatten pool2/child1
$ rbd rm pool2/child2
$ rbd snap rm pool/image@snap
Cannot remove a protected snapshot: pool/image@snap
$ rbd snap unprotect pool/image@snap
```

Then the snapshot can be deleted like normal:

```
$ rbd snap rm pool/image@snap
```

IMPLEMENTATION

DATA FLOW

In the initial implementation, called 'trivial layering', there will be no tracking of which objects exist in a clone. A read that hits a non-existent object will attempt to read from the parent snapshot, and this will continue recursively until an object exists or an image with no parent is found. This is done through the normal read path from the parent, so differing object sizes between parents and children do not matter.

Before a write to an object is performed, the object is checked for existence. If it doesn't exist, a copy-up operation is performed, which means reading the relevant range of data from the parent snapshot and writing it (plus the original write) to the child image. To prevent races with multiple writes trying to copy-up the same object, this copy-up operation will include an

atomic create. If the atomic create fails, the original write is done instead. This copy-up operation is implemented as a class method so that extra metadata can be stored by it in the future. In trivial layering, the copy-up operation copies the entire range needed to the child object (that is, the full size of the child object). A future optimization could make this copy-up more fine-grained.

Another future optimization could be storing a bitmap of which objects actually exist in a child. This would obviate the check for existence before each write, and let reads go directly to the parent if needed.

These optimizations are discussed in:

http://marc.info/?l=ceph-devel&m=129867273303846

PARENT/CHILD RELATIONSHIPS

Children store a reference to their parent in their header, as a tuple of (pool id, image id, snapshot id). This is enough information to open the parent and read from it.

In addition to knowing which parent a given image has, we want to be able to tell if a protected snapshot still has children. This is accomplished with a new per-pool object, *rbd_children*, which maps (parent pool id, parent image id, parent snapshot id) to a list of child image ids. This is stored in the same pool as the child image because the client creating a clone already has read/write access to everything in this pool, but may not have write access to the parent's pool. This lets a client with read-only access to one pool clone a snapshot from that pool into a pool they have full access to. It increases the cost of unprotecting an image, since this needs to check for children in every pool, but this is a rare operation. It would likely only be done before removing old images, which is already much more expensive because it involves deleting every data object in the image.

PROTECTION

Internally, protection_state is a field in the header object that can be in three states. "protected", "unprotected", and "unprotecting". The first two are set as the result of "rbd protect/unprotect". The "unprotecting" state is set while the "rbd unprotect" command checks for any child images. Only snapshots in the "protected" state may be cloned, so the "unprotected" state prevents a race like:

1. A: walk through all pools, look for clones, find none
2. B: create a clone
3. A: unprotect parent
4. A: rbd snap rm pool/parent@snap

RESIZING

Resizing an rbd image is like truncating a sparse file. New space is treated as zeroes, and shrinking an rbd image deletes the contents beyond the old bounds. This means that if you have a 10G image full of data, and you resize it down to 5G and then up to 10G again, the last 5G is treated as zeroes (and any objects that held that data were removed when the image was shrunk).

Layering complicates this because the absence of an object no longer implies it should be treated as zeroes - if the object is part of a clone, it may mean that some data needs to be read from the parent.

To preserve the resizing behavior for clones, we need to keep track of which objects could be stored in the parent. We can track this as the amount of overlap the child has with the parent, since resizing only changes the end of an image. When a child is created, its overlap is the size of the parent snapshot. On each subsequent resize, the overlap is *min(overlap, new_size)*. That is, shrinking the image may shrinks the overlap, but increasing the image's size does not change the overlap.

Objects that do not exist past the overlap are treated as zeroes. Objects that do not exist before that point fall back to reading from the parent.

Since this overlap changes over time, we store it as part of the metadata for a snapshot as well.

RENAMING

Currently the rbd header object (that stores all the metadata about an image) is named after the name of the image. This makes renaming disrupt clients who have the image open (such as children reading from a parent). To avoid this, we can name the header object by the id of the image, which does not change. That is, the name of the header object could be *rbd_header.$id*, where $id is a unique id for the image in the pool.

When a client opens an image, all it knows is the name. There is already a per-pool *rbd_directory* object that maps image

names to ids, but if we relied on it to get the id, we could not open any images in that pool if that single object was unavailable. To avoid this dependency, we can store the id of an image in an object called *rbd_id.$image_name*, where $image_name is the name of the image. The per-pool *rbd_directory* object is still useful for listing all images in a pool, however.

HEADER CHANGES

The header needs a few new fields:

- int64_t parent_pool_id
- string parent_image_id
- uint64_t parent_snap_id
- uint64_t overlap (how much of the image may be referring to the parent)

These are stored in a “parent” key, which is only present if the image has a parent.

CLS_RBD

Some new methods are needed:

```
/****************** methods on the rbd header ********************/
/**
 * Sets the parent and overlap keys.
 * Fails if any of these keys exist, since the image already
 * had a parent.
 */
set_parent(uint64_t pool_id, string image_id, uint64_t snap_id)

/**
 * returns the parent pool id, image id, snap id, and overlap, or -ENOENT
 * if parent_pool_id does not exist or is -1
 */
get_parent(uint64_t snapid)

/**
 * Removes the parent key
 */
remove_parent() // after all parent data is copied to the child

/*************** methods on the rbd_children object *****************/

add_child(uint64_t parent_pool_id, string parent_image_id,
          uint64_t parent_snap_id, string image_id);
remove_child(uint64_t parent_pool_id, string parent_image_id,
             uint64_t parent_snap_id, string image_id);
/**
 * List ids of a given parent
 */
get_children(uint64_t parent_pool_id, string parent_image_id,
             uint64_t parent_snap_id, uint64_t max_return,
             string start);
/**
 * list parent
 */
get_parents(uint64_t max_return, uint64_t start_pool_id,
            string start_image_id, string start_snap_id);


/************ methods on the rbd_id.$image_name object **************/

set_id(string id)
get_id()

/************** methods on the rbd_directory object *****************/

dir_get_id(string name);
dir_get_name(string id);
dir_list(string start_after, uint64_t max_return);
dir_add_image(string name, string id);
dir_remove_image(string name, string id);
dir_rename_image(string src, string dest, string id);
```

Two existing methods will change if the image supports layering:

```
snapshot_add - stores current overlap and has_parent with
               other snapshot metadata (images that don't have
               layering enabled aren't affected)

set_size     - will adjust the parent overlap down as needed.
```

LIBRBD

Opening a child image opens its parent (and this will continue recursively as needed). This means that an ImageCtx will contain a pointer to the parent image context. Differing object sizes won't matter, since reading from the parent will go through the parent image context.

Discard will need to change for layered images so that it only truncates objects, and does not remove them. If we removed objects, we could not tell if we needed to read them from the parent.

A new clone method will be added, which takes the same arguments as create except size (size of the parent image is used).

Instead of expanding the rbd_info struct, we will break the metadata retrieval into several API calls. Right now, the only users of rbd_stat() other than 'rbd info' only use it to retrieve image size.