NEXT3 Snapshots Design

Rev. 1.3, April 2010
Amir G.

Contents

Design Goal

Provide snapshot capabilities with thin provisioning, high performance and scalability.
Snapshot management capabilities should include creating as many snapshots as desired and deleting any past snapshots to free up file system space.

Introduction

LVM Snapshots

Snapshot capabilities are available at the volume level with LVM snapshots, but there are some caveats to LVM snapshots:

1. The need to define a fixed volume size per snapshot
2. Copy overhead per snapshot for every write operation
3. Not scalable to O(1TB), requires O(1GB) RAM and long loading time
4. There is some memory overhead per mounted snapshot volume
5. Not possible to define a snapshot on a directory (sub volume)

Some of these caveats can be tackled by changes to the LVM snapshot implementation, but some are fundamental to the volume level snapshot approach.
For this reason it makes sense to try to reach the design goals by implementing snapshots at the file system level.

Ext3cow

Ext3cow is an experimental patch for ext3, which implements snapshots at file system level.
Some implementation ideas were taken from ext3cow.
Ext3cow is not backward compatible with ext3.
Ext3cow includes many changes in the ext3 file system internals and changes the on disk format. These changes set the stability and consistency of Ext3cow at high risk.

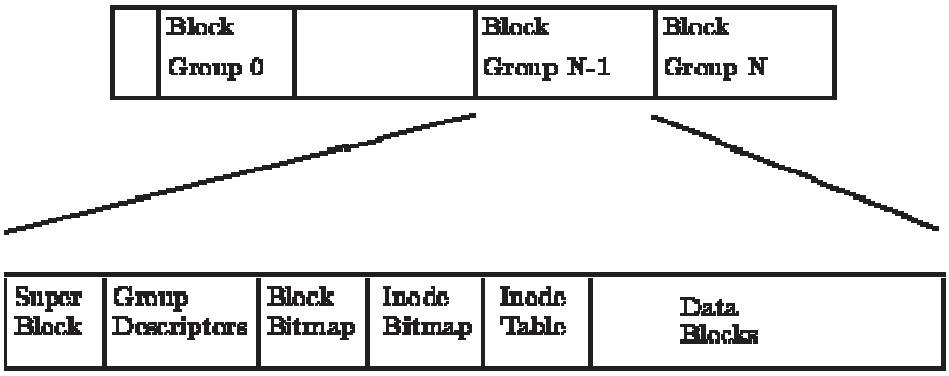
NEXT3™ Snapshots

The proposed design combines the simplicity of the LVM snapshots with some ideas from Ext3cow to achieve most of the design goals and preserve the stability and consistency of good old ext3.

Technical Background

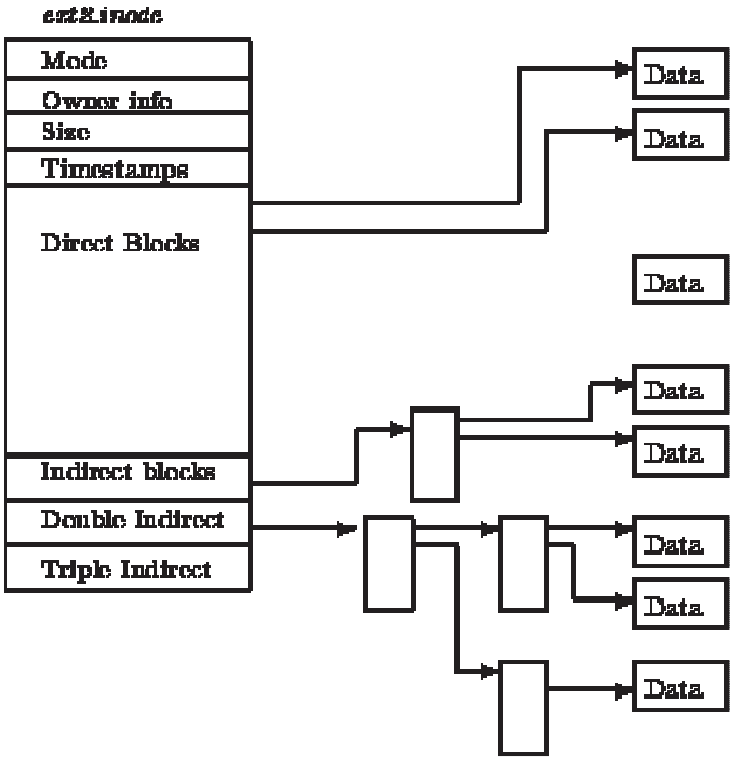
Ext2 File System

Ext3 is built over the simple ext2 layout.
The file system is divided into block groups as shown in the figure below.
For example, a 4TB file system consists of 32K block groups X 32K blocks X 4KB.
Each block group contains a few global metadata blocks, which are allocated at mkfs time and can only be reallocated using offline tools.



Ext2 Inode

Every allocated block in the file system, except the global metadata blocks, is referenced by exactly one inode, either directly, or in-directly, as shown in the figure below.
A single inode can map 12 direct blocks + 1K indirect blocks + 1M double indirect blocks + 1G triple indirect blocks, which sets an upper limit of a little over 4TB per file with 4KB blocks.
There is a lower upper limit of 2TB per file, imposed by the sectors count field of the inode.



Implementation Notes

Metadata blocks

Metadata blocks writes are monitored by the journal block device.
Before the journalled metadata blocks are committed modified, a copy of the original metadata block should be made.
For every snapshot, there is a special snapshot file that represents it.
Every snapshot file is associated with a read-only loop block device.
Block X in the snapshot file represents a private copy of block X in the snapshot block device.
A hole at block X in the snapshot file signifies that block X in the snapshot block device should be read from a later snapshot (or from the origin).
To avoid an endless loop, snapshot file blocks writes are ignored by the COW mechanism.

Data blocks

Data blocks in this context refer to data blocks of regular files.
They can be defined as every block which is not journalled as metadata.
Unlike metadata blocks, which are copied on write, data blocks are moved on write, which means there are fewer (possibly zero) extra I/O oprerations per data block write.
The MOW technique of data blocks was taken from ext3cow.
In order to maintain compatibility to ext3 and keep track of the space used by the snapshot,
All data blocks which are de-allocated in the file system are moved to the active snapshot.
This prevents the file system from re-using these blocks.
The snapshot file disk usage indicates the amount of space occupied by the snapshot.
The deletion of the snapshot file results in reclamation of snapshot blocks to the file system.

COW bitmap

Snapshot COW bitmaps are the block bitmaps copies of that snapshot,
meaning that if a block was allocated at the time the snapshot was taken,
it is referenced by that snapshot and therefor, should be preserved unchanged.
The latest (active) snapshot COW bitmap determines which blocks need to be COWed.
On the first block group access after snapshot take, the COW bitmap is created by COWing the block group block bitmap to snapshot.
A direct reference to the new COW bitmap block is stored in its block group descriptor along with the (active) snapshot id, which it belongs to.

Exclude bitmap

The exclude bitmap indicates all the blocks which are not to be COWed to snapshot, specifically blocks of snapshot files.
Exclude bitmap blocks are stored in a special EXCLUDE inode.
The EXCLUDE inode is pre-allocated on mkfs and resize2fs (one block per block group).
A direct reference for each exclude bitmap block is stored in its block group descriptor.
Unlike the active COW bitmap blocks, which are allocated lazily for every new snapshot, the active exclude bitmap blocks are the same blocks (with different content) for every active snapshot.

Inodes and directories

Inodes and directories are copied as part of the metadata COW mechanism.
The copied inode numbers and directory paths remain the same but belong to a different file system, which is accessible from the snapshot block device.

Snapshot Management

Listing snapshots

The snapshot files are kept in a dedicated snapshots directory.
The snapshot file name is the name of the snapshot (defaults to time of snapshot creation).
The snapshot file size (of enabled snapshots) is the entire volume size.
The snapshot file disk usage (du) implies the amount of space used by the snapshot.
Snapshot inodes are chained by a linked list, starting at the super block, similarly to the way that orphan inodes are chained.

Accessing a snapshot

1. Mount the snapshot file via loop device as read-only ext2

Taking a snapshot

1. Allocate inode with some initial blocks for snapshot file
2. Lock journal updates and flush journal to disk
3. Copy data to initial blocks of snapshot file (and fix super block)
4. Set new snapshot file as active snapshot file
5. Enable snapshot mount and unlock journal updates

Deleting a snapshot

1. Disable snapshot mount
2. Mark snapshot as 'deleted'
3. Merge snapshot with older not 'in-use' snapshot
4. Verify that the snapshot is not 'in-use' by older enabled snapshots
5. Delete the snapshot file (now or when the snapshot becomes not in-use)

Snapshot enable

1. Set snapshot file size to the entire volume size
2. Grant user read access

Snapshot disable

1. Set snapshot file size to zero
2. Revoke user read access

Dependency rules

1. Only last snapshot can be 'active' (target of copy-on-write)
2. Last snapshot can only become non 'active' when it is deleted
   or when a new snapshot becomes the 'active' snapshot
3. Snapshot is 'in-use' if an enabled older snapshot exist
4. Snapshot 'in-use' cannot be deleted nor merged (only marked as 'deleted')

Pseudo Code Implementation

Helper functions

```
snapshot_map_block (f, n, b)
        Map block b to file f at offset n
snapshot_get_block (f, n)
        Return the block at offset n in file f or zero for a hole at offset n
snapshot_get_cow_bitmap (s, b)
        Return the COW bitmap block of snapshot s that holds the bit for block b
snapshot_get_exclude_bitmap (s, b)
        Return the exclude bitmap block of snapshot s that holds the bit for block b
snapshot_should_cow_block (s, b)
        test_bit (b, snapshot_get_cow_bitmap (s, b)) &&
        !test_bit (b, snapshot_get_exclude_bitmap (s, b)) &&
        !snapshot_get_block (s, b)
```

snapshot_take ()

```
        s = new_inode ()
        s->prev = SNAPSHOT
         SNAPSHOT->next = s
        get_super_block () ->snapshot = s
        SNAPSHOT = s
```

snapshot_delete (s)

```
        // don't delete active snapshot
        Assert (SNAPSHOT != s)
        // mark the snapshot deleted so we can continue to free blocks after crash
        s->deleted = true
        // traverse all snapshot data blocks
        For every offset n in file s do
                If (b = snapshot_get_block (s, n))
                        If (snapshot_should_cow_block (s->prev, n))
                                // move block to previous snapshot
                                snapshot_map_block (s->prev, n, b)
                                snapshot_map_block (s, n, 0)
                        Else
                                // free block and un-exclude it from snapshots([1])
                                free_block(b);
                                unset_bit (b, snapshot_get_exclude_bitmap (SNAPSHOT,b))

        // remove deleted snapshot from snapshots list
        s->prev->next = s->next
        s->next->prev = s->prev
        // delete snapshot inode
        delete_inode (s)
```

[1] *All snapshot blocks are excluded from active snapshots by setting their bits in the EXCLUDED inode bitmap.*
when a snapshot block is reclaimed by the file system, it should be un-excluded, so it may be COWed into future snapshots, after it is re-allocated for a regular file.

On prepare_write_file_data_block (f, n)

```
// called before writing data to file f at offset n
If (b = snapshot_get_block (f, n))
        If (snapshot_should_cow_block (SNAPSHOT, b))
                // move block to active snapshot and exclude it from snapshots([2])
                snapshot_map_block (SNAPSHOT, b, b)
                set_bit (b, snapshot_get_exclude_bitmap (SNAPSHOT,b))
                snapshot_map_block (f, n, 0)
                // a new block will be allocated for the data
```

On prepare_write_metadata_block (b)

```
// called before writing to any metadata block
If (snapshot_should_cow_block (SNAPSHOT, b))
        // copy metadata into a new block
        nb = new_block ()
        copy_block (nb, b)
        // map copy to active snapshot and exclude it from snapshots
        snapshot_map_block (SNAPSHOT, nb)
        set_bit (nb, snapshot_get_exclude_bitmap (SNAPSHOT,nb))
```

On new_block ()

```
b = alloc_block ()
// new allocated blocks should not be referenced by snapshots
Assert (! snapshot_should_cow_block (SNAPSHOT, b))
// new allocated blocks should not be excluded from snapshots
Assert (!test_bit (b, snapshot_get_exclude_bitmap (SNAPSHOT,b))
```

On delete_block (b)

```
// called when deleting/truncating files
If (snapshot_should_cow_block (SNAPSHOT, b))
        // map block to active snapshot and exclude it from snapshots
        snapshot_map_block (SNAPSHOT, b)
        set_bit (b, snapshot_get_exclude_bitmap (SNAPSHOT,b))
Else
        free_block (b)
```

[2] *All snapshot blocks are excluded from active snapshots by setting their bits in the EXCLUDED inode bitmap.*

Journal and Recovery

Ext3 tracks all metadata write transactions in the journal, which can be used to replay the transaction at mount time after crash.
An additional mechanism called orphan inodes list is used to maintain a persistent list of inodes that need to be deleted or truncated.
In case of a crash, the orphan inodes will be deleted on the next mount after journal recovery is complete.

With NEXT3 snapshots, snapshots don't have journals and the snapshot file changes are journalled as part of the file system transaction that triggered them.
For example, an ext3 transaction of truncating a single block b from file f looks like this:

```
{map_block (f, 0, 0); unset_bit (block_bmap, b); update_inode (f)}
```

With NEXT3 snapshots enabled the transaction would look like this:

```
{ map_block (s, b, b); set_bit (exclude_bmap, b); update_inode (s);
  map_block (f, 0, 0); update_inode (f) }
```

After crash, journal recovery will bring the file system and snapshot file to consistent state.
The file system must be successfully mounted before any snapshot may be mounted.
Before taking a snapshot, the journal is flushed to the disk, using the file system operation write_super_lockfs (), which brings the file system to a consistent state on-disk with no pending transactions in the journal.
Then, the super block is copied to the snapshot file and it is fixed to look like an ext2 super block (no journal).

Extending journal credits to include snapshot metadata

When starting an ext3 journal transaction, one must specify in advance the amount of buffer credits (N) that will be required for the transaction.
In order to include the snapshot metadata in the journal, each NEXT3 transaction requests 18N+3 credits to account for allocating up to 5N snapshot blocks:
N COW blocks + N COW bitmap blocks([3]) + 2N indirect blocks + N double indirect block.
(3 credits each for journalling block bitmap, exclude bitmap and group descriptor)
Another 2N credits for journalling:
N snapshot indirect block + N snapshot double indirect block
Add finaly, another 3 credits are reserved in every transaction for jouranlling:
1 super block + 1 snapshot inode block + 1 snapshot triple indirect block.
The credits are extended on journal start, restart, extend and release functions.
The snapshot file data blocks themselves are not journalled, but 'ordered' (written before the journalled), so they don't use any credits.
The snapshot COW bitmaps and their indirect blocks are synced to disk immediately after they are created, to avoid using any credits([4]).

File truncate and file sytem resize transactions in ext3 are special, because they can be very large and it's hard to calculate their credits in advance.
To overcome this difficulty, their credits are extended every time that they drop under a threshold (EXT3_RESERVE_TRANS_BLOCKS). In NEXT3, this threshold is extended to include snapshot metadata reserve credits.

[3] On first block group access there is an extra COW operation to copy the COW bitmap.
[4] Syncing the COW bitmap block to disk is time consuming but it doesn't happen very often.

Avoiding Recursions

The concept of a snapshot file, which is a clone of the file system in which it is stored, invites potential recursions. These recursions have to be identified and handled carefully.

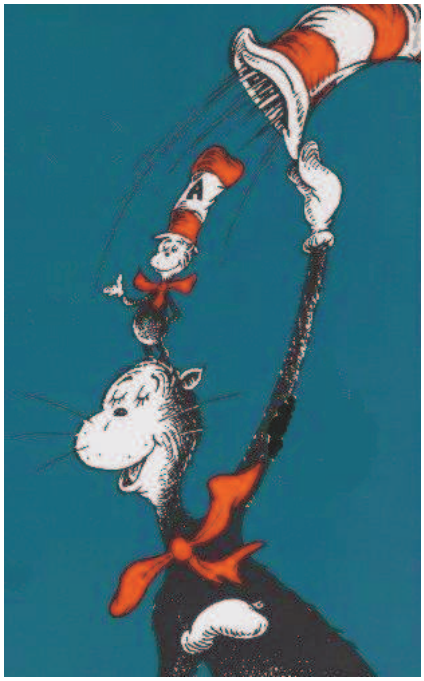
Ignoring journal writes

As mentioned in the "journal and recovery" section, snapshots do not have journals.
This helps avoiding the recursion of having to COW journal blocks to the snapshot and add them to the running transaction, while the transaction itself is being committed.

Growing snapshot file side effects

A snapshot file starts as an empty inode and grows slowly, while file system blocks are being written and deleted.
The growth of the active snapshot file causes some extra file system block writes, which may trigger additional blocks COW:

1. snapshot blocks
2. snapshot inode (table)
3. super block (and group descriptors)
4. block bitmaps



All snapshot blocks writes are ignored and it makes sense, since the snapshot blocks were not allocated at the time that the snapshot was taken.

The snapshot inode and super block are copied while taking the snapshot to prevent the recursion from happening.

The blocks bitmap recursion is more challenging.
In a file system of 4TB, with 32K block groups the potential recursion may have 32K levels. It is also not good practice to copy 32K bitmap blocks while taking the snapshot.
Instead, we trigger the bitmap block COW on first access to the block group, that is, the first time that the COW bitmap is accessed.
We always try to allocate snapshot file blocks from the same block group as the file system blocks that they map. If we succeed in doing that, the recursion will end after one level.

If the block group of the mapped blocks is full, the snapshot file blocks may be allocated from a block group that hasn't been accessed yet.
To avoid the recursion from exploding, we ignore these bitmap block COW requests.
This may cause the snapshot to contain a non accurate copy of the file system bitmaps at the time that the snapshot was taken.
However, we are guaranteed that the inaccuracy of these bitmaps is restricted to bits that represent blocks of the active snapshot file.
This 'problem' can be easily fixed, since all these snapshot blocks are marked in the exclude bitmap, which we use to clear all snapshot blocks from the COW bitmap.

Avoiding Race Conditions

The journal is responsible for on-disk consistency of the snapshots.
This means that if a snapshot is mounted after boot, it is guaranteed to be consistent and to reflect the file system at the time it was taken.
However, the snapshot may be mounted and accessed while the file system is changing and while the snapshot file itself is active (growing).
Some special attention is required in order to maintain in-memory consistency of the snapshots.

Tracked read operations

The read page operation is composed of 2 separate atomic file system operations:

1. map the page to a block device/number
2. read the block from the block device

Normally, the map of a page doesn't change (only created and deleted).
But with snapshots, the COW operation changes the mapping of a snapshot block.
In case of snapshot block COW that happens between the map and read operations,
the map operation will map the page to the block device before the block was copied and the read operation will read the block from the block device after the block was copied and possibly after it has been changed by the file system.
In order to avoid this race condition, page reads are 'tracked' and COW operation completion is delayed until there are no tracked reads of the block.

Pending COW operations

The COW operation is composed of 3 separate atomic file system operations:

1. test if snapshot file block is mapped (and if not)
2. allocate the snapshot file block
3. copy the block data to the snapshot block

A race condition can occur when 2 concurrent COW operations on the same block get a negative result on the snapshot map test.
Both tasks try to allocate the snapshot file block. Since block allocations are protected by inode truncate mutex, only the first one allocates the block and the second gets the first allocation.
The task that allocates the snapshot block, grabs the buffer cache entry (sb_getblk) and marks the buffer 'new'. When COW operation is completed, the task clears the 'new' flag from the buffer and drops the buffer cache reference count.
The task that did not allocate the snapshot block, or any task that finds a mapped snapshot block (read from snapshot for example), checks if a buffer cache entry exists and if the buffer is marked 'new', it waits until the pending COW operation completes.

Pending bitmap COW operations

The bitmap COW operation is called on first block group access, so it may be invoked by 2 concurrent tasks COW-ing 2 different blocks in the same block group.
It is protected with a special block group lock.
The first task takes the lock, performs the bitmap COW operation and writes the COW bitmap block number in the group descriptor.
The second task waits until the COW bitmap block is written to the group descriptor.

Future work

1. New features
    a. exclude regular files and directories from snapshots
    b. merge ext4 features, but not extents, to NEXT3 (NEXT4)
    c. snapshots and files writable clones
2. Performance
    a. run file system benchmarks
    b. reduce usage of buffer_heads and duplicate pages of the same block
3. Unsupported ext3 features
    a. user/group quotas (copy user/group quota blocks on snapshot take)
    b. journaled data
    c. direct I/O
    d. block size != page size