Thin provisioning

Introduction

This document describes a collection of device-mapper targets that between them implement thin-provisioning and snapshots.

The main highlight of this implementation, compared to the previous implementation of snapshots, is that it allows many virtual devices to be stored on the same data volume. This simplifies administration and allows the sharing of data between volumes, thus reducing disk usage.

Another significant feature is support for an arbitrary depth of recursive snapshots (snapshots of snapshots of snapshots ...). The previous implementation of snapshots did this by chaining together lookup tables, and so performance was O(depth). This new implementation uses a single data structure to avoid this degradation with depth. Fragmentation may still be an issue, however, in some scenarios.

Metadata is stored on a separate device from data, giving the administrator some freedom, for example to:

- Improve metadata resilience by storing metadata on a mirrored volume but data on a non-mirrored one.
- Improve performance by storing the metadata on SSD.

Status

These targets are considered safe for production use. But different use cases will have different performance characteristics, for example due to fragmentation of the data volume.

If you find this software is not performing as expected please mail dm-devel@redhat.com with details and we'll try our best to improve things for you.

Userspace tools for checking and repairing the metadata have been fully developed and are available as 'thin_check' and 'thin_repair'. The name of the package that provides these utilities varies by distribution (on a Red Hat distribution it is named 'device-mapper-persistent-data').

Cookbook

This section describes some quick recipes for using thin provisioning. They use the dmsetup program to control the device-mapper driver directly. End users will be advised to use a higher-level volume manager such as LVM2 once support has been added.

Pool device

The pool device ties together the metadata volume and the data volume. It maps I/O linearly to the data volume and updates the metadata via two mechanisms:

- Function calls from the thin targets
- Device-mapper 'messages' from userspace which control the creation of new virtual devices amongst other things.

Setting up a fresh pool device

Setting up a pool device requires a valid metadata device, and a data device. If you do not have an existing metadata device you can make one by zeroing the first 4k to indicate empty metadata.

```
dd if=/dev/zero of=$metadata_dev bs=4096 count=1
```

The amount of metadata you need will vary according to how many blocks are shared between thin devices (i.e. through snapshots). If you have less sharing than average you'll need a larger-than-average metadata device.

As a guide, we suggest you calculate the number of bytes to use in the metadata device as 48 * $data_dev_size / $data_block_size but round it up to 2MB if the answer is smaller. If you're creating large numbers of snapshots which are recording large amounts of change, you may find you need to increase this.

The largest size supported is 16GB: If the device is larger, a warning will be issued and the excess space will not be used.

Reloading a pool table

You may reload a pool's table, indeed this is how the pool is resized if it runs out of space. (N.B. While specifying a different metadata device when reloading is not forbidden at the moment, things will go wrong if it does not route I/O to exactly the same on-disk location as previously.)

Using an existing pool device

```
dmsetup create pool \
```

```
--table "0 20971520 thin-pool $metadata_dev $data_dev \
         $data_block_size $low_water_mark"
```

$data_block_size gives the smallest unit of disk space that can be allocated at a time expressed in units of 512-byte sectors. $data_block_size must be between 128 (64KB) and 2097152 (1GB) and a multiple of 128 (64KB). $data_block_size cannot be changed after the thin-pool is created. People primarily interested in thin provisioning may want to use a value such as 1024 (512KB). People doing lots of snapshotting may want a smaller value such as 128 (64KB). If you are not zeroing newly-allocated data, a larger $data_block_size in the region of 256000 (128MB) is suggested.

$low_water_mark is expressed in blocks of size $data_block_size. If free space on the data device drops below this level then a dm event will be triggered which a userspace daemon should catch allowing it to extend the pool device. Only one such event will be sent.

No special event is triggered if a just resumed device's free space is below the low water mark. However, resuming a device always triggers an event; a userspace daemon should verify that free space exceeds the low water mark when handling this event.

A low water mark for the metadata device is maintained in the kernel and will trigger a dm event if free space on the metadata device drops below it.

Updating on-disk metadata

On-disk metadata is committed every time a FLUSH or FUA bio is written. If no such requests are made then commits will occur every second. This means the thin-provisioning target behaves like a physical disk that has a volatile write cache. If power is lost you may lose some recent writes. The metadata should always be consistent in spite of any crash.

If data space is exhausted the pool will either error or queue IO according to the configuration (see: error_if_no_space). If metadata space is exhausted or a metadata operation fails: the pool will error IO until the pool is taken offline and repair is performed to 1) fix any potential inconsistencies and 2) clear the flag that imposes repair. Once the pool's metadata device is repaired it may be resized, which will allow the pool to return to normal operation. Note that if a pool is flagged as needing repair, the pool's data and metadata devices cannot be resized until repair is performed. It should also be noted that when the pool's metadata space is exhausted the current metadata transaction is aborted. Given that the pool will cache IO whose completion may have already been acknowledged to upper IO layers (e.g. filesystem) it is strongly suggested that consistency checks (e.g. fsck) be performed on those layers when repair of the pool is required.

Thin provisioning

i. Creating a new thinly-provisioned volume.

To create a new thinly- provisioned volume you must send a message to an active pool device, /dev/mapper/pool in this example:

```
dmsetup message /dev/mapper/pool 0 "create_thin 0"
```

Here '0' is an identifier for the volume, a 24-bit number. It's up to the caller to allocate and manage these identifiers. If the identifier is already in use, the message will fail with -EEXIST.

ii. Using a thinly-provisioned volume.

Thinly-provisioned volumes are activated using the 'thin' target:

```
dmsetup create thin --table "0 2097152 thin /dev/mapper/pool 0"
```

The last parameter is the identifier for the thinp device.

Internal snapshots

i. Creating an internal snapshot.

Snapshots are created with another message to the pool.

N.B. If the origin device that you wish to snapshot is active, you must suspend it before creating the snapshot to avoid corruption. This is NOT enforced at the moment, so please be careful!

```
dmsetup suspend /dev/mapper/thin
dmsetup message /dev/mapper/pool 0 "create_snap 1 0"
dmsetup resume /dev/mapper/thin
```

Here '1' is the identifier for the volume, a 24-bit number. '0' is the identifier for the origin device.

ii. Using an internal snapshot.

Once created, the user doesn't have to worry about any connection between the origin and the snapshot. Indeed the snapshot is no different from any other thinly-provisioned device and can be snapshotted itself via the same method. It's perfectly legal to have only one of them active, and there's no ordering requirement on activating or removing them both.

(This differs from conventional device-mapper snapshots.)

Activate it exactly the same way as any other thinly-provisioned volume:

```
dmsetup create snap --table "0 2097152 thin /dev/mapper/pool 1"
```

External snapshots

You can use an external **read only** device as an origin for a thinly-provisioned volume. Any read to an unprovisioned area of the thin device will be passed through to the origin. Writes trigger the allocation of new blocks as usual.

One use case for this is VM hosts that want to run guests on thinly-provisioned volumes but have the base image on another device (possibly shared between many VMs).

You must not write to the origin device if you use this technique! Of course, you may write to the thin device and take internal snapshots of the thin volume.

i. Creating a snapshot of an external device

This is the same as creating a thin device. You don't mention the origin at this stage.

```
dmsetup message /dev/mapper/pool 0 "create_thin 0"
```

ii. Using a snapshot of an external device.

Append an extra parameter to the thin target specifying the origin:

```
dmsetup create snap --table "0 2097152 thin /dev/mapper/pool 0 /dev/image"
```

N.B. All descendants (internal snapshots) of this snapshot require the same extra origin parameter.

Deactivation

All devices using a pool must be deactivated before the pool itself can be.

```
dmsetup remove thin
dmsetup remove snap
dmsetup remove pool
```

Reference

'thin-pool' target

i. Constructor

```
thin-pool <metadata dev> <data dev> <data block size (sectors)> \
          <low water mark (blocks)> [<number of feature args> [<arg>]*]
```

Optional feature arguments:

skip_block_zeroing:
: Skip the zeroing of newly-provisioned blocks.

ignore_discard:
: Disable discard support.

no_discard_passdown:
: Don't pass discards down to the underlying data device, but just remove the mapping.

read_only:
: Don't allow any changes to be made to the pool metadata. This mode is only available after the thin-pool has been created and first used in full read/write mode. It cannot be specified on initial thin-pool creation.

error_if_no_space:
: Error IOs, instead of queueing, if no space.

Data block size must be between 64KB (128 sectors) and 1GB (2097152 sectors) inclusive.

ii. Status

```
<transaction id> <used metadata blocks>/<total metadata blocks>
<used data blocks>/<total data blocks> <held metadata root>
ro|rw|out_of_data_space [no_]discard_passdown [error|queue]_if_no_space
needs_check|- metadata_low_watermark
```

transaction id:
: A 64-bit number used by userspace to help synchronise with metadata from volume managers.

used data blocks / total data blocks

If the number of free blocks drops below the pool's low water mark a dm event will be sent to userspace. This event is edge-triggered and it will occur only once after each resume so volume manager writers should register for the event and then check the target's status.

held metadata root:

The location, in blocks, of the metadata root that has been 'held' for userspace read access. '-' indicates there is no held root.

discard_passdown|no_discard_passdown

Whether or not discards are actually being passed down to the underlying device. When this is enabled when loading the table, it can get disabled if the underlying device doesn't support it.

ro|rw|out_of_data_space

If the pool encounters certain types of device failures it will drop into a read-only metadata mode in which no changes to the pool metadata (like allocating new blocks) are permitted.

In serious cases where even a read-only mode is deemed unsafe no further I/O will be permitted and the status will just contain the string 'Fail'. The userspace recovery tools should then be used.

error_if_no_space|queue_if_no_space

If the pool runs out of data or metadata space, the pool will either queue or error the IO destined to the data device. The default is to queue the IO until more space is added or the 'no_space_timeout' expires. The 'no_space_timeout' dm-thin-pool module parameter can be used to change this timeout -- it defaults to 60 seconds but may be disabled using a value of 0.

needs_check

A metadata operation has failed, resulting in the needs_check flag being set in the metadata's superblock. The metadata device must be deactivated and checked/repaired before the thin-pool can be made fully operational again. '-' indicates needs_check is not set.

metadata_low_watermark:

Value of metadata low watermark in blocks. The kernel sets this value internally but userspace needs to know this value to determine if an event was caused by crossing this threshold.

iii. Messages

create_thin <dev id>
: Create a new thinly-provisioned device. <dev id> is an arbitrary unique 24-bit identifier chosen by the caller.

create_snap <dev id> <origin id>
: Create a new snapshot of another thinly-provisioned device. <dev id> is an arbitrary unique 24-bit identifier chosen by the caller. <origin id> is the identifier of the thinly-provisioned device of which the new device will be a snapshot.

delete <dev id>
: Deletes a thin device. Irreversible.

set_transaction_id <current id> <new id>
: Userland volume managers, such as LVM, need a way to synchronise their external metadata with the internal metadata of the pool target. The thin-pool target offers to store an arbitrary 64-bit transaction id and return it on the target's status line. To avoid races you must provide what you think the current transaction id is when you change it with this compare-and-swap message.

reserve_metadata_snap
: Reserve a copy of the data mapping btree for use by userland. This allows userland to inspect the mappings as they were when this message was executed. Use the pool's status command to get the root block associated with the metadata snapshot.

release_metadata_snap
: Release a previously reserved copy of the data mapping btree.

'thin' target

i. Constructor

```
thin <pool dev> <dev id> [<external origin dev>]
```

pool dev:
: the thin-pool device, e.g. /dev/mapper/my_pool or 253:0

dev id:
: the internal device identifier of the device to be activated.

external origin dev:

an optional block device outside the pool to be treated as a read-only snapshot origin: reads to unprovisioned areas of the thin target will be mapped to this device.

The pool doesn't store any size against the thin devices. If you load a thin target that is smaller than you've been using previously, then you'll have no access to blocks mapped beyond the end. If you load a target that is bigger than before, then extra blocks will be provisioned as and when needed.

ii. Status

<nr mapped sectors> <highest mapped sector>

If the pool has encountered device errors and failed, the status will just contain the string 'Fail'. The userspace recovery tools should then be used.

In the case where <nr mapped sectors> is 0, there is no highest mapped sector and the value of <highest mapped sector> is unspecified.