Persistent data

Introduction

The more-sophisticated device-mapper targets require complex metadata that is managed in kernel. In late 2010 we were seeing that various different targets were rolling their own data structures, for example:

- Mikulas Patocka's multisnap implementation
- Heinz Mauelshagen's thin provisioning target
- Another btree-based caching target posted to dm-devel
- Another multi-snapshot target based on a design of Daniel Phillips

Maintaining these data structures takes a lot of work, so if possible we'd like to reduce the number.

The persistent-data library is an attempt to provide a re-usable framework for people who want to store metadata in device-mapper targets. It's currently used by the thin-provisioning target and an upcoming hierarchical storage target.

Overview

The main documentation is in the header files which can all be found under drivers/md/persistent-data.

The block manager

dm-block-manager.[hc]

This provides access to the data on disk in fixed sized-blocks. There is a read/write locking interface to prevent concurrent accesses, and keep data that is being used in the cache.

Clients of persistent-data are unlikely to use this directly.

The transaction manager

dm-transaction-manager.[hc]

This restricts access to blocks and enforces copy-on-write semantics. The only way you can get hold of a writable block through the transaction manager is by shadowing an existing block (ie. doing copy-on-write) or allocating a fresh one. Shadowing is elided within the same transaction so performance is reasonable. The commit method ensures that all data is flushed before it writes the superblock. On power failure your metadata will be as it was when last committed.

The Space Maps

dm-space-map.h dm-space-map-metadata.[hc] dm-space-map-disk.[hc]

On-disk data structures that keep track of reference counts of blocks. Also acts as the allocator of new blocks. Currently two implementations: a simpler one for managing blocks on a different device (eg. thinly-provisioned data blocks); and one for managing the metadata space. The latter is complicated by the need to store its own data within the space it's managing.

The data structures

dm-btree.[hc] dm-btree-remove.c dm-btree-spine.c dm-btree-internal.h

Currently there is only one data structure, a hierarchical btree. There are plans to add more. For example, something with an array-like interface would see a lot of use.

The btree is 'hierarchical' in that you can define it to be composed of nested btrees, and take multiple keys. For example, the thin-provisioning target uses a btree with two levels of nesting. The first maps a device id to a mapping tree, and that in turn maps a virtual block to a physical block.

Values stored in the btrees can have arbitrary size. Keys are always 64bits, although nesting allows you to use multiple keys.