Commit Callback for DMU Transactions

Author: Andreas Dilger

12/24/2007

1 Introduction

In order to know when asynchronous transactions have been committed to stable storage, Lustre requires a mechanism to update a given OBD device's `last_committed_transno` in memory for replying to clients in each RPC. In ext3/ldiskfs this is accomplished by hooking a callback function and associated callback data to a given JBD journal handle while that handle is still open, transferring the handle to the transaction at handle commit time, and in the JBD code after a transaction has successfully committed to the journal the commit callback is called.

Implementing a similar mechanism to notify the Lustre code when operations have committed to stable storage for the ZFS DMU is required in order to maintain the Lustre functionality and allow asynchronous commit notification to the clients.

2 Architecture

The DMU Commit Callback API implements a mechanism to allow the upper layers of the software stack to get asynchronous notification of low level transaction events in order to allow a coherent distributed transaction mechanism to be implemented for Lustre, but without having to know the internal details of the transaction mechanism.

This should be accomplished on the DMU by having the caller associate callback functions and data with each atomic operation and the caller is notified via the callback when this has been committed to stable storage in some manner.

3 External Functional specifications

3.1 Prototypes

```
/* This magic number is internal to the dmu_tx_callback_*() functions */
```

```
#define DMU_CALLBACK_MAGIC 0xca11bac0ca11bacfull
typedef char dmu_callback_data_t;
typedef (void dmu_callback_func_t)(dmu_callback_data_t *dcb_data, int error);
typedef struct dmu_callback {
        list_node            dcb_list;    /* linked to tx_callbacks list */
        __u64                dcb_magic;   /* magic number to verify header */
        dmu_callback_func_t *dcb_func;    /* caller function pointer */
        dmu_callback_data_t  dcb_data[0]; /* caller private data */
} dmu_callback_t;
dmu_callback_data_t *dmu_tx_callback_data_create(size_t bytes);
int dmu_tx_callback_commit_add(dmu_tx_t *tx, dmu_callback_func_t *dcb_data, int error),
                               dmu_callback_data_t *dcb_data);
int dmu_tx_callback_data_destroy(dmu_callback_data_t *dcb_data);
```

4 High Level Logic

The DMU makes a transaction handle available to the caller once it has been created via `dmu_tx_create()`. While the handle is active, before `dmu_tx_commit()` is called, new callbacks can be registered against that handle. When the handle is committed the callbacks will be transferred to the transaction group proper. When the transaction group is committed to disk all of the callbacks are called, in no defined order.

If there is an error in transaction group processing (e.g. corrupt filesystem, filesystem read-only, `dmu_tx_abort()` on transaction handle) any registered callback functions are called with errno != 0 to indicate to the caller that the operation was not completed and to give the caller an opportunity to clean up the allocated memory and any associated caller state. If a created callback is to be destroyed before it is added to the transaction handle the caller is responsible to destroy it itself.

The structure of `dmu_callback_t` is opaque to the caller, so a new function `dmu_tx_callback_data_create()` will be used to allocate space for the callback data. A separate allocation function is used to avoid having two small allocations and frees for each callback (one for the DMU-internal state in `dmu_callback_t`, and one for the caller's data). Only the `dmu_commit_data_t` part of the allocated data is returned to the caller, and the rest of the structure is private to the DMU. The function `dmu_tx_commit_callback_add()` regenerates the `dmu_callback_t` pointer via `container_of()` or equivalent and verifies the `dcb_magic` before registering the callback on the `dmu_tx_t`. Multiple callbacks can be registered on a `dmu_tx_t`.

A new `tx_callbacks` list is added to the `dmu_tx` structure to hold the list of callbacks and their data:

```
struct dmu_tx {
        /*
         * No synchronization is needed because a tx can only be handled
```

```
         * by one thread.
         */
        list_t tx_holds; /* list of dmu_tx_hold_t */
        objset_t *tx_objset;
        struct dsl_dir *tx_dir;
        struct dsl_pool *tx_pool;
        uint64_t tx_txg;
        uint64_t tx_lastsnap_txg;
        uint64_t tx_lasttried_txg;
        txg_handle_t tx_txgh;
        void *tx_tempreserve_cookie;
        struct dmu_tx_hold *tx_needassign_txh;
        list_t tx_callbacks; /* list of dmu_callback_t on this dmu_tx */
        uint8_t tx_anyobj;
        int tx_err;
#ifdef ZFS_DEBUG
        uint64_t tx_space_towrite;
        uint64_t tx_space_tofree;
        uint64_t tx_space_tooverwrite;
        refcount_t tx_space_written;
        refcount_t tx_space_freed;
#endif
};
```

The per-transaction list of callbacks is moved from `dmu_tx_t` to `tx_state_t` in `dmu_tx_commit()` by calling a new internal function `txg_rele_commit_cb()`. The `tx_state_t` now tracks the list of all `dmu_callback_t` that need to be run after a particular transaction group is completed. The `tx_commit_callbacks` list is an array of `TXG_SIZE` elements, which results in a separate callback list per in-flight transaction group. The `tx_commit_callbacks[]` array is indexed by the `dmu_tx_t.tx_txg & TXG_MASK`, which is constant for the lifetime of the `dmu_tx_t`.

```
typedef struct tx_state {
        tx_cpu_t        *tx_cpu;        /* protects right to enter txg  */
        kmutex_t        tx_sync_lock;   /* protects tx_state_t */
        krwlock_t       tx_suspend;
        uint64_t        tx_open_txg;    /* currently open txg id */
        uint64_t        tx_quiesced_txg; /* quiesced txg waiting for sync */
        uint64_t        tx_syncing_txg; /* currently syncing txg id */
        uint64_t        tx_synced_txg;  /* last synced txg id */
        uint64_t        tx_sync_txg_waiting; /* txg we're waiting to sync */
        uint64_t        tx_quiesce_txg_waiting; /* txg we're waiting to open */
        kcondvar_t      tx_sync_more_cv;
        kcondvar_t      tx_sync_done_cv;
        kcondvar_t      tx_quiesce_more_cv;
```

```
        kcondvar_t      tx_quiesce_done_cv;
        kcondvar_t      tx_timeout_exit_cv;
        kcondvar_t      tx_exit_cv;     /* wait for all threads to exit */
        uint8_t         tx_threads;     /* number of threads */
        uint8_t         tx_exiting;     /* set when we're exiting */
        list_t          tx_commit_callbacks[TXG_SIZE]; /* post-commit callbacks */
        kthread_t       *tx_sync_thread;
        kthread_t       *tx_quiesce_thread;
        kthread_t       *tx_timelimit_thread;
} tx_state_t;
```

In `lib/libzpool/txg.c::txg_sync_thread()`, after `spa_sync()` is finished writing the data to disk the pending callbacks are called. In the majorify of cases the error parameter is 0, but when there is read-only support for the DMU then the callbacks need to be called with a non-zero error (e.g. EROFS) to indicate that the operation was not committed to disk. The callbacks are responsible for freeing the callback memory, so there is little to do other than removing the items from the list and calling the callbacks.

```
/* iterate over commit callbacks on this txg */
for (dcb = list_head(&txg->txg_commit_callbacks[txg & TXG_MASK]),
     next = dcb ? list_next(&txg->txg_callbacks, dcb) : NULL;
     dcb;
     dcb = next,
     next = dcb ? list_next(&txg->txg_callbacks, dcb) : NULL) {
        dmu_callback_func_t *dcb_func = dcb->dcb_func;
        list_remove(&txg->txg_callbacks, dcb);
        dcb_func(dcb->dcb_data, 0 /* non-zero if SPA read-only */);
}
```

Each callback function is required to free the allocated data itself when it is called and has finished with the data. Because the callback data contains private state that is needed by the DMU, the callback data must be freed with the function `dmu_tx_data_callback_destroy()`. If the callback is not yet registered with the `dmu_tx_t` then the caller must also destroy the callback data with `dmu_tx_data_callback_destroy()`.

The callback function receives as parameters the `dmu_callback_data_t` passed as the parameter to `dmu_tx_commit_callback_add()` and an `error` parameter that indicates if there was an error committing the data to disk.

5 Use-Case Scenarios

5.1 Describe use cases for all normal and abnormal uses of externally visible functions.

dmu_tx_callback_data_create() returns a pointer to a `dmu_commit_data_t` at least as large as the requested transaction data size, or NULL if there is an allocation failure. The allocated memory must be freed by the caller using `dmu_tx_callback_data_destroy()`, either within the registered callback function (after `dmu_tx_callback_commit_add()` is called) or directly if the callback will never be registered for some reason.

dmu_tx_callback_commit_add() returns 0 for success, or EINVAL if there is an invalid parameter passed to the function. This function validates the passed `tx` (if possible), and `db_data.dcb_magic`. As the callback data should be preallocated via `dmu_tx_callback_data_create()` and the `tx` should be always be held, an error can only happen in the case of coding errors or memory corruption.

dmu_tx_callback_data_destroy() will free the supplied dcb and return 0 for success. If the the `dcb_magic` value is incorrect EINVAL will be returned and no action taken. This can only fail if there is a coding error or memory corruption. This function is always used by the external caller, either from within the registered commit callback function or directly for unregistered callbacks, and not the DMU.

dcb_callback_func_t() will be called when the transaction group has committed to disk. If `error` is 0 then the operation completed successfully, otherwise the operation did not complete. In either case, the allocated memory must be freed with dmu_tx_callback_data_destroy().

5.2 Describe use cases demonstrating interoperability between the software with and without this module.

For ZFS there should be no interoperability issues, as the callback functionality will remain unused in the code and the `tx_commit_callbacks` list will always be empty so no action will be taken.

For Lustre this change is required for proper functionality of recovery, so no release can be made without it. All code changes are internal to the DMU and the Lustre `udmu` calling code so no interoperability issues should arise.

5.3 Describe use cases demonstrating any scalability use cases mentioned in the architecture document.

For uses by Lustre the commit callback has a very low overhead (non-sleeping locks, and the free of the callback data), so there is not anticipated to be any scalability issues

different than those already existing with ldiskfs. The list of callbacks is only walked once at commit time, and each element is being removed from the list. As there is no defined order for the callbacks there are no sorting or other ordering requirements and items can be inserted into the `tx_callbacks` list in O(1).

6 State Machine Design

6.1 Locking

The transaction handle's `tx_callbacks` list does not need to be locked, as only a single processor can be active on a single transaction at one time, per comment at `struct dmu_tx` declaration. The transaction group's `tx_commit_callbacks` list also does not itself need any locking at callback time, as the transaction commit is also handled by a single processor, and the transactions are removed from the list before the transaction is called, so even if the `dmu_callback_t` is passed to another thread before freeing there is no risk of list corruption.

The `tx_commit_callbacks` list *does* need to be locked during the short time that the callbacks are moved from `dmu_tx_t` to `tx_state_t` in `dmu_tx_commit->txg_rele_commit_cb()`. This would use the per-txg lock in `dmu_tx_t->tx_txgh` similar to `txg_rele_to_sync()`.

6.2 Disk state changes

None.

7 Test Plan

A simple test program that links into the DMU could be constructed, or PIOS could be modified to register callbacks to monitor the time taken for each write to commit to disk.

8 Plan Review

9 Alternatives/Questions

Polling the underlying transaction group number to determine when it is committed to disk would also be possible, but is considerably less flexible.