CLIO Simplification High Level Design

Added by Zhenyu Xu, last edited by Richard Henwood on Jan 28, 2014

Introduction

The Lustre* file system client implementation on the IO path (CLIO) was rewritten in Lustre software 2.0. The motivation to redesign the CLIO stack is recorded in the CLIO Simplification Solution Architecture. This document also contains a detailed analysis of successes and shortfalls of the CLIO 2.0 stack in today's HPC deployments.

The purpose of this project is to make CLIO easier to maintain. Since Lustre software version 2.0.0 it is estimated that the CLIO portions of the code have received 460 patches. Even with this enormous effort, bugs still remain. With the current CLIO implementation, Lustre file systems are radically different from a common abstraction used by other Linux file systems. This document is concerned with describing the specific approach to designing a simpler CLIO stack that retains the best parts of the 2.0 Stack and cuts away costly complexity - as described in the CLIO Simplification Solution Architecture.

Definitions and Acronyms

CLIO

CLient IO module.

CL_LOCK

Lock implementation on CLIO. Cl_lock is based on DLM lock implementing a cacheable lock at the LLITE and LOV layer

LLITE

Lustre file system VFS implementation

VVP

VVP refers to VFS, VM and Posix. VVP is called by llite to interpret VFS operations to client IO operations

SLP

SLP refers to Sysio Library Posix. This is actually liblustre implementation. It has this name because liblustre uses sysio to intercept libc calls

CCC

CCC refers to Client Common Code. If the code can be shared by SLP and VVP, it is usually moved into CCC subdirectory

LOVSUB

LOVSUB is a substitute OSC layer at LOV layer. In CLIO implementation stripe data is hidden and inaccessible at LOV layer. To make OSC call up to LLITE layer, LOVSUB is introduced which is at LOV layer and has access to stripe info.

LDLM

LDLM stands for Lustre Distributed Lock Mechanism

AST

AST stands for Asynchronous System Trap, it is the callback of LDLM.

Requirements

Simplify `cl_lock`

In current implementation of CLIO, `cl_lock` is implemented as two layer cacheable lock. There are two different categories of operations which can access the lock: FTTB (from top to bottom) and FBTT (from bottom to top). This structure can easily introduce deadlock so deadlock avoidance mechanism has to be invented. This makes cl_lock implementation extremely complex and hard to maintain. A cache-less `cl_lock` will be worked out to replace the current implementation. There will be no FBTT operations so deadlock will no longer be an issue.

Client IO cleanup

Remove `lov_stripe_md` access beyond LOV

One of the benefits for 2.0 CLIO is that it limits the access to `lov_stripe_data` at the LOV layer. This enables a convenient implementation of layout lock. However, legacy code continues to exist in the client that requires `lov_stripe_data` from the LLITE layer. Typically, these are `ioctl` operations. In this project, all references to `lov_stripe_data` will be eliminated from LOV.

Cleanup obsoleted OBD operations

OBD operations became obsolete after MDT, OFD, and client reconstructing were complete but code remains. Most of the interfaces in obd_operations, for example `osc_brw()`, is not referenced by module and can be removed in this project. OBD operations are still employed for configuration. That code will remain intact.

`ioctl` method of `cl_object`

There are many ioctl functions from `ll_file_ioctl()` that still use obsolete OBD interfaces. A new method of `cl_object_operations` will be created to replace all of those OBD interfaces.

Functional specification

`cl_lock` simplification

`cl_lock` data structure and method will be retained. However, `cl_lock` will be degenerated as a data container of lock enqueue information such as lock mode, lock extent and enq flags etc. `cl_lock` will be stateless. it will simply provide a way to transfer enqueue information among layers. A mutex for lock becomes redundant as only the owner of lock, who is the process to queue the lock, can access the lock at any time.

Data structure

Data structure of `cl_lock` will be simplified as follows:

```
struct cl_lock {
        /* List of slices. Immutable after creation. */
        cfs_list_t            cll_layers;
        /* locks in cl_object_headers, for statistic */
        cfs_list_t            cll_linkage;
        /* lock attribute, extent, cl_object, etc. */
        struct cl_lock_descr  cll_descr;
        /* Flag bit-mask for lock attribute */
        unsigned long         cll_flags;
};
```

There are still two level of `cl_lock`: top `cl_lock` is composed of `vvp_lock` and `lov_lock`; sub `cl_lock` is composed of `lovsub_lock` and `osc_lock`. The data structure of `vvp_lock` will not change. `lov_lock` and `lovsub_lock` will be simplified significantly, as follows:

```
struct lov_lock_sub {
        struct cl_lock  sub_lock;
        int         sub_stripe;
        int         sub_flags;
};

struct lov_lock {
        struct cl_lock_slice   lls_cl;
        /** Number of sub-locks in this lock */
            int         lls_nr;
        int         lls_index;
        /* sublock array */
        struct lov_lock_sub *lls_sub;
};
```

Since the queuing process is changed, old data structures such as `cl_lock_closure` will not need. A `spin_lock` will be added to `osc_lock` because ptlrpc daemon and the owner can both access it on the same time. `osc_lock` will be revised as follows:

```
struct osc_lock {
        /* Internal lock to protect states, etc. */
        spinlock_t  ols_lock;
        /* owner sleeps on this channel for state change */
        struct cl_sync_io *ols_owner;
        /* list of waiting for this lock to be canceled */
        cfs_list_t      ols_waiting_list;
        /* wait_entry of ols_waiting_list */
        cfs_list_t  ols_wait_entry;

        /* original contents of osc_lock */
        …
};
```

After `cl_lock` becomes cache-less, only the following methods will be available in `cl_lock_operations`:

- `->clo_enqueue`: to start enqueue of `cl_lock`
- `->clo_cancel`: cancel a `cl_lock, release its DLM lock reference`
- `->clo_print`: print the attribute of `cl_lock`
- `->clo_free: destroy a cl_lock, release its memory use`.

One principle of `cl_lock` simplification is that only the lock owner can access the lock so there is no refcount in `cl_lock` at all.

APIs

Two interfaces will be introduced with `cl_lock` simplification:

```
int cl_lock_enqueue(struct cl_env *env, struct cl_lock *lock, struct cl_sync_io
*anchor)
```

This function is called to enqueue a `cl_lock`. `cl_sync_io` data structure is reused here in the event resources are not immediately available. For example, if a conflicting lock already exists the lock cancellation must be waited for.

Return Value:

0: enqueue has been successful

< 0: error code

> 0: have to wait on anchor for resources

`cl_lock_enqueue()` can be called repeatedly until the lock is granted successfully. As long as `cl_lock_enqueue()` is invoked, `cl_lock_cancel()` must be called to clean up resources even if an error occurred.

```
int cl_lock_cancel(struct cl_env *env, struct cl_lock *lock)
```

After the process finishes using `cl_lock`, `cl_lock_cancel()` is called to release locks and any potential resources have been held during enqueue process. The change of `cl_lock` will not affect `cl_io`. For each VFS IO, it will still enter into `cl_io_loop()` to initialize IO. If the IO needs lock, `cl_lock_request()` will be called. Instead of matching `cl_lock` as what we did before, new lock will be allocated and then enqueued by `cl_lock_enqueue()`.

Client IO cleanup

Cleanup work can be split into multiple phases. In this design, only most severe problems will be covered.

Remove `lov_stripe_md` access beyond LOV

The following functions are referring `lov_stripe_md` directly at llite layer:

- `obd_find_cbdata`
- `ll_inode_getattr`
- `ll_glimpse_ioctl`
- `ll_lov_recreate`
- `ll_lov_setstripe_ea_info`
- `ll_lov_getstripe`
- `ll_do_fiemap`
- `ll_data_version`
- `ll_iocontrol`

Some of these functions of can be implemented by `ioctl` method of `cl_object`:

- `obd_find_cbdata`
- `ll_lov_getstripe`
- `ll_do_fiemap`
- `ll_data_version`

Some of them have already been obsoleted so that we can delete them:

- `ll_inode_getattr` (obsolete, used by SoM)
- `ll_lov_recreate` (currently used by FSCK, becomes obsolete with the arrivial of LFSCK phase 2)
- `ll_iocontrol`

`ll_glimpse_ioctl()` can be revised to use glimpse functionality and `ll_lov_setstripe_ea_info()` doesn't need to check if stripe data exists in prior.

Cleanup obsoleted OBD operations

Most of the operations in `obd_ops` can be deleted or provided by an alternative solution. Specific OBD IO operations targetted for removal include:

- `o_precreate`
- `o_create`
- `o_create_async`
- `o_destroy`
- `o_setattr`
- `o_setattr_async`
- `o_getattr`
- `o_getattr_async`
- `o_brw`
- `o_merge_lvb`

- o_adjust_kms
- o_punch
- o_sync
- o_migrate
- o_copy
- o_preprw
- o_commitrw
- o_enqueue
- o_cancel
- o_change_cbdata
- o_find_cbdata
- o_change_cbdata
- o_extent_calc

All of these operations in the OSC and LOV layers will be diminished. Further clean-up of the configuration is possible but it is out of the scope of this design.

`ioctl` method of `cl_object`

An `ioctl` method of `cl_object` is needed to address the problem of OBD interfaces still called by other code. These remaining interfaces will be clio-ized. Prototype of `ioctl` method for `cl_object_operations`:

```
int (*coo_ioctl)(struct lu_env *env, struct cl_object *obj, unsigned int cmd,
unsigned long arg);
```

The following `ioctl` command will be supported to replace their corresponding implementation in `ll_file_ioctl()`:

- IOC_LOV_GETSTRIPE
- IOC_FIEMAP
- IOC_DATA_VERSION
- IOC_FIND_CBDATA

LLITE, LOV and OSC layer must implement their own method of `ioctl`. This enables each layer to fulfill separate requests. For example, `IOC_LOV_GETSTRIPE` can be finished at LOV layer, but `IOC_FIND_CBDATA` must descend to the OSC layer to discover if there is caching lock for the inode.

Function `cl_object_ioctl()` is worked out to fulfill this requirement. The prototype of `cl_object_ioctl()` will be:

```
int cl_object_ioctl(struct lu_env *env, struct cl_object *obj, unsigned int cmd,
unsigned long arg);
```

In `cl_object_ioctl()`, `->coo_ioctl` for each layer will be called. If the object includes multiple sub objects, LOV will be responsible for fan out of the requests and merging of the results.

Removal of non-Linux stub

There are two part of code can be removed for this purpose: some libcfs portability code and client common code (with "`ccc_`" prefix). ccc is an abbreviation for client common code.

- `libcfs` portability code changes are tabulated below:

current code pattern	change to/remove	comment
`cfs_atomic_xxx`	`atomic_xxx`	to comply with Linux kernel code
`cfs_get_blocked_sigs`	**remove**	no definition, no usage
`cfs_strncasecmp`	`strncasecmp`	to comply with Linux kernel code
`cfs_vsnprintf`	`vsnprintf`	to comply with Linux kernel code
`cfs_snprintf`	**remove**	no usage
`cfs_list_xxx`	`list_xxx`	to comply with Linux kernel code
`cfs_hlist_xxx`	`hlist_xxx`	to comply with Linux kernel code

- client common code change
  The ccc layer will be completely removed provided removal liblustre is removed and the remaining functions are merged into vfs vm posix layer.

Logic specification

`Cl_lock` Simplification

In this section, to fully understand the requirement of `cl_lock`, the implementation details of the current `cl_io_lock` will be discussed firstly. This section will then describe what will be changed in the new `cl_lock` implementation. Finally it will cover implementation details to support common use cases.

Background of `cl_io_lock` and `cl_lock`

Before making a change to the code it is necessary understand the current behaviour and design. In this section, `cl_lock` workings in current implementation are discussed to provide context for the designed changes.

In CLIO, IO is driven by `cl_io_loop()` as follows:

```
do {
    io->ci_continue = 0;
    result = cl_io_iter_init(env, io);
    if (result == 0) {
        nob    = io->ci_nob;
        result = cl_io_lock(env, io);
        if (result == 0) {
            result = cl_io_start(env, io);
            cl_io_end(env, io);
            cl_io_unlock(env, io);
            cl_io_rw_advance(env, io, …);
        }
     }
     cl_io_iter_fini(env, io);
} while (result == 0 && io->ci_continue);
```

An IO is performed by multiple stages:

- `cl_io_iter_init()`: Lustre file system does IO stripe by stripe. The major functionality of `cl_io_iter_init()` is to align the IO by stripe size.
- `cl_io_lock()`: ask around each layer on client side to decide what locks will be needed to perform the IO. The IO needs multiple locks if the read/write buffer is composed of mmaped region from Lustre file system files. After all locks have been collected, it will sort those locks and request them.
- `cl_io_start()`: where an IO actually happens. Typically, kernel component interfaces will be called to perform IO.
- `cl_io_end()`: post mortem analysis for statistic etc. For `setattr` request all RPCs must first complete.
- `cl_io_unlock()`: release `cl_locks` held for IO.
- `cl_io_iter_fini()`: finish the current stripe and get ready for the next one.

As the code snippet shows, the above functions are called repeatedly until the entire buffer is handled. In `cl_io_lock()`, `cl_io_operations::cio_lock` will first be called to request each layer for which locks are needed to perform this specific IO. At present LLITE layer can determine the required locks. In future, supporting network RAID will require the LOV layer to provide lock parity. Taking a look at `vvp_io_rw_lock()`:

```
static int vvp_io_rw_lock(…)
{
    …
     result = vvp_mmap_locks(env, cio, io);
     if (result == 0)
            result = ccc_io_one_lock(env, io, …);
     …
}
```

`vvp_mmap_locks()` is called to check if the read/write buffer is a mmaped region of Lustre file system files, then `ccc_io_one_lock()` is called to add the lock region for IO object itself:

```
ccc_io_one_lock
-> ccc_io_one_lock_index
-> cl_io_lock_add
```

The required lock region will be added into `cl_io::cl_lockset` by function `cl_io_lock_add()`. Thereafter, `cl_io_locks_lock()` is called to enqueue each lock in the lockset by calling `cl_lockset_lock()`, then `cl_lock_request()` is called to start the enqueue process.

At lock enqueue stage `{vvp|lov|lovsub|osc}_lock_enqueue()` will be called to enqueue the lock. In `lov_lock_enqueue()`, the lock may be fanned out if this is a multiple stripe file and the operation is append write or setattr. After lock is granted, its state must be in `CLS_HELD`.

After the IO is finished, `cl_io_unlock()` will be called to release all locks for this IO. For each specific lock, `cl_unuse()` is called to release the lock and put them back to lock cache. `clo_lock_unuse()` method for each layer will be called to release the lock. After lock is unused, its state will become `CLS_CACHED`.

New `cl_lock` Enqueue and Request

After required lock regions are collected in `cl_lockset`, `cl_lock_request()` will be called to request each lock and enqueue them. Since there is no lock cache any more, `cl_lock_alloc()` is called directly in `cl_lock_request()`, then `cl_lock_enqueue()` will be called to enqueue the lock.

`cl_lock_enqueue()` is designed to be called repeatedly until lock is granted, or error occurs. In `cl_lock_request()`, it will be implemented as:

```
cl_lock_request()
{
    struct cl_sync_io anchor;
    int rc;

    …
    lock = cl_lock_alloc();
    while(1) {
        rc = cl_lock_enqueue(env, lock, &anchor);
        if (rc <= 0)
            break;

        rc = l_wait_event(anchor->csi_waitq,
                atomic_read(&anchor->csi_sync_nr) == 0);
        if (rc < 0)
            break;
    }
    if (rc < 0)
        cl_lock_cancel(env, lock);
    …
}
```

At LOV layer, `lov_lock_enqueue()` will call `cl_lock_enqueue()` on each sub locks.

```
lov_lock_enqueue(…, struct cl_sync_io *anchor)
{
    …
    for (idx = 0; idx < lck->lls_nr; idx++) {
        get subenv and subio for sub lock;
        rc = cl_lock_enqueue(subenv->lse_env, subenv->lse_io,
                        &lov->lls_sub[idx].sub_lock,
                        anchor);
        if (rc < 0 ||
            rc > 0 && !(descr->cll_enq_flags & CFS_ASYNC))
            /* error occurred or need to wait for resource */
            break;
    }
    …
}
```

As the above code snippet shows, the process of queuing the lock may sleep with if sub locks are held. This can be changed to release previous sub locks if necessary.

At OSC layer, `osc_lock_enqueue()` will be called. Here in this function, a DLM lock will be found or queued and attached to the `osc_lock`. `osc_lock` is actually a cache-less substitute of DLM lock. Multiple `osc_locks` may use the same DLM lock in the interim.

Lockless lock is used to support lockless IO, which has no DLM lock attached; also lockless lock is conflicted with any other overlapped lock. To support lockless lock, `osc_locks` are linked into a list of `osc_object`. Enqueuing a lock must wait until any conflicting locks are resolved or cancelled. This method will be applied to address lockless problem, and for early lock cancel. This avoids needing to request the server to cancel the locks. The pseudo code of `osc_lock_queue()` is as follows:

```
osc_lock_enqueue()
{
    if (osc->ols_state == OLS_GRANTED)
        return 0;

    add the osc_lock into tail of osc_object’s lock list;

restart:
    list_for_each(tmp, osc_object’s lock list) {
        if (tmp == osc_lock)
            break;
        if (tmp is conflicted with current lock) {
            add osc_lock into tmp’s ols_waiting_list;
            wait for the lock to be canceled;
            goto restart;
        }
    }

    if (osc_lock is lockless) {
        osc->ols_state = OLS_GRANTED;
        return 0;
    }

    osc_lock->ols_state = OLS_ENQUEUED;
    if (glimpse lock) {
        cfs_atomic_inc(&anchor->csi_sync_nr);
        osc_lock->ols_owner = anchor;
    }

    /* dlm lock’s ast data must be osc_object;
     * if glimpse or AGL lock, async of osc_enqueue_base() must
         * be true, DLM’s enqueue callback set to osc_lock_upcall            * with
cookie as osc_lock; */
    call osc_enqueue_base() to find or allocate a DLM lock;
    osc_lock->ols_lock = dlmlock;
    if (glimpse lock)
        return +1;

    wait for the lock to be granted;
    return 0;
}
```

In `osc_lock_enqueue()`, it will handle the lock request synchronously, except for glimpse request. Glimpse requests must be asynchronous and `osc_lock_upcall()` is registered as the callback of enqueue so that it will be called after enqueue is finished. Finally `cl_sync_io_note()` is called to awake the enqueue process provided by `cl_lock_enqueue()`. `osc_lock_enqueue()` will be discussed later case by case.

`cl_lock` Cancel

`cl_lock_cancel()` can be called either after the lock is used, or error occurs. There are corresponding `lov_lock_cancel()` and `osc_lock_cancel()`; lock cancel request should be fanned out at `lov_lock_cancel()`. Before canceling a lock, the caller has to make sure `cis_sync_nr` in `cl_sync_io` equals to zero. This is to make sure that all outstanding glimpse RPCs have been finished. In `osc_lock_cancel()`, `osc_lock` state will be checked to make sure DLM lock is released. In addition, it will check if there are pending requests for this lock. The pseudo code of `osc_lock_cancel()` is be:

```
osc_lock_cancel()
{
    osc_lock->ols_state = OLS_RELEASED;
    if (osc_lock->ols_lock != NULL)
        release DLM  lock;

    take this lock out of osc_object’s list;
    list_for_each(tmp, osc_lock’s waiting list) {
        wake it up by cl_sync_io_note(tmp->ols_ower, 1);
        take it out of the list;
    }
}
```

After a lock is canceled, `cl_lock_free()` will be called to free lock slices. `osc_lock_cancel()` must be implemented carefully to ensure all references to `osc_lock` are cleaned up.

Lock Wait Policy

The last parameter of `cl_lock_enqueue()` is a `cl_sync_io` data structure. This is used to wait on some resources during enqueue, for example, to wait a conflicting lock to be canceled, or glimpse RPCs to finish. Whenever it needs to wait for resources, `osc_lock::ols_owner` will be assigned to the `cl_sync_io`, and then the process will yield the CPU and sleeping on `cl_sync_io::csi_waitq` .

At present, the enqueue process can await three kinds of resources:

- Conflicting lock exists

The `osc_lock` will be added into conflicting lock’s waiting list. When the conflicting lock is canceled, it will iterate this list and wake them up.

- Glimpse lock

For glimpse lock, `cl_sync_io::csi_nr` is the number of glimpse RPCs have been sent. In `osc_lock_upcall()`, it will call `cl_sync_io_note()` on `osc_lock` so that the process will be woken up after all RPCs are finished.

- Lock completion wait

For multiple stripe lock of append write and setattr, sub locks have to be queued one by one by a specific order. In this case, we have to wait in `osc_lock_enqueue()` for the lock to be granted before returning to the upper layer. When waiting for lock completion, `osc_lock::ols_owner` will be assigned and the process will be woken up by completion AST of DLM lock.

`osc_lock` and DLM Lock

`cl_lock` becomes associated with `cl_io` if an IO is finished. All related `cl_lock` must be destroyed by the DLM lock remains in cache. In `osc_lock_enqueue()` the DLM cache will be identified through `osc_enqueue_base()`. If a DLM lock is being used by an `osc_lock`, it can not be canceled because `osc_lock` holds a reader/writer count of DLM lock.

A DLM lock can be shared by multiple `osc_locks`. `osc_lock` cannot be used as AST data of DLM lock because it is associated with IO. As a consequence, `osc_object` is used instead. This will impact lock AST handling as follows:

- Blocking AST

When blocking AST is called there must be no `osc_lock` being attached to the DLM lock. `osc_lock_flush()` will be called to write back dirty pages of this `osc_object` by the extent of DLM lock, then `osc_lock_discard_pages()` will be called to discard covering pages.

- Glimpse AST

`cl_object_glimpse()` will be called to collect up-to-date lvb.

- Completion AST

Given `osc_object` is employed as lock AST data there is no direct way to locate the `osc_lock` in the completion AST. The `osc_lock` list of `osc_object` will be iterated to find the corresponding `osc_locks` and wake up the sleeping process if it exists. There may be multiple `osc_locks` attaching to the same DLM lock.

- Weigh AST

Weigh AST is used to determine if a DLM lock should be early canceled. Another use case of weigh AST would be to check if a lock needs replaying in a recovery. In the weigh AST, `cl_object_weigh()` will be called to return weight of this lock. Locks covering mmap region will have heavier weight so less likely to be canceled.

Glimpse and AGL

Glimpse must be asynchronous. In order to make it happen the glimpse processes will enqueue the glimpse RPCs asynchronously and await all RPCs to complete. The pseudo code for glimpse handling in `osc_lock_enqueue()` is as follows:

```
osc_lock_enqueue(…, struct cl_sync_io *anchor)
{
    …
    rc = osc_enqueue_base();
    if (rc == 0 && lock is glimpse) {
        cfs_atomic_inc(&anchor->csi_nr);
        osc_lock->ols_owner = anchor;
    }
    …
}
```

At the LLITE layer the glimpse process will sleep on `cl_sync_op::csi_waitq`. In `osc_lock_upcall()`, `cl_sync_io_note()` will be called so that the waiting process will be woken after all `osc_locks` have been handled.

AGL is similar to glimpse, but AGL does not wait for the lock. In addition the `cl_lock` for AGL will be destroyed immediately after enqueue is finished so that a dangling DLM lock will be created and cached in the system. When the glimpse is called later, hopefully we can find DLM lock in cache so RPCs will not be issued. The glimpse process will be fast because all DLM locks are already in cache so only local memory data will be operated. There is no lock upcall for AGL.

Readahead

Readahead pages have to be covered by a granted lock in a Lustre file system. Therefore, CLIO uses `osc_page::cpo_is_under_lock` to check if a page to be read ahead is covered by an `osc_lock`.

However `osc_page_is_under_lock()` implementataion is implement during a cancel on recovery. A lock will be canceled instead of replayed during a recovery, if it does not cover any pages. In the new implementation, AST weight will be employed to solve this problem. In the weigh callback, we will check the page radix tree of the corresponding object, and then return a hint to ptlrpc to make decision.

Estimation of Code Change

After this `cl_lock` implementation is complete only `cl_lock_{alloc|free}` will be retained in `obdclass/cl_lock.c`; new functions of `cl_lock_{enqueue|cancel}` will be implemented as a simple encapsulation of `->clo_enqueue` and `->clo_cancel`.

At LOV layer most of the code in `lov_lock.c` will be removed. Since the relationship between top and sub lock is determined and immutable after initialization the functions of `sublock_{lock|unlock}` will become redundant; in addition, `cl_lock_closure` is not needed as well because there is no possibility of deadlock.

At OSC layer, the code in `osc_lock.c` is still needed because it has to handle the DLM cache. And most of them have to be reimplemented to accommodate the new lock structure.

Wire protocol changes

None.

Risk Analysis

Background

Locking mechanisms are crucial to the correct behaviour of the Lustre software. Central to the CLIO Simplification Design project is the re-factorization of `cl_lock` client locking mechanism. The intention of this work is to reduce the complexity of `cl_lock` to enhance maintainability but the importance of locking function demands a separate risk analysis at this stage.

RISK

A modified locking mechanism may affect the behaviour of the clients.

LIKELIHOOD and IMPACT

medium and high

STRATEGY

Mitigate. All code changes to the CLIO `cl_lock` must complete the existing test suite. In addition, all code must be reviewed by two senior engineers and complete the gate-keeping requirements and testing.

RISK

A modified locking mechanism may not correctly lock the contended resource.

LIKELIHOOD and IMPACT

medium and high

STRATEGY

Mitigate. Passing of the 'racer' test suite with 6 OSTs is introduced as a requirement for the completed simplified `cl_lock`. racer is currently disabled as it does not predictably pass with the existing, complex `cl_lock` implementation.

RISK

Issues with the new lock implementation only become visible at scale.

LIKELIHOOD and IMPACT

medium and high

STRATEGY

Mitigate. Testing at scale will be performed as part of routine Lustre software release process. We are confident that we can find large scale early adopters who will provide feedback on their HPC workloads. A proportion of engineering will be reserved after completion of the project for solving issues that only become visible at scale.

Open issues and future work

None.

---

30 Comments

Nathan Rutman

"There will be no FBTT operations so deadlock will no longer be an issue" – presumably FBTT was done for a reason - will we lose performance?

Jinshan Xiong

Yes, it may add per IO overhead because now we have to rebuild the cl_lock{} data structure in memory. However, we will take this into consideration in the implementation and do performance tune so hopefully it won't affect performance at all.

Nathan Rutman

```
struct lov_lock_sub *lls_sub;
```

How is this array reconciled with the layout enhancement project? There may be multiple sets of stripes...

Jinshan Xiong

If there exists multiple set of stripes, one set has to be decided to use at IO initialization time, which means we already know which stripes will be used for lock request.

Nathan Rutman

There's a decent amount of risk associated with these extensive locking changes – is there any chance for a prototype or proof-of-concept to prove out these changes? Is there a level of confidence in this solution?

Jinshan Xiong

We already did some work to verify the ideas - thanks for reminding.

it indeed has a lot of using scenarios.

Richard Henwood

Hi Nathan,

Jinshan and the team have completed a risk analysis, and attached it to the end of this document. Please let me know if you can see areas for improvement.

Cory Spitz

First, thanks for the opportunity for non-Lustre architects to pose questions and make comments.

In the solution architecture it is stated that CLIO will not be redesigned. However, I think that making cl_lock cache-less brings into question why cl_lock still exists onto of DLM lock. A statement about the necessity of cl_lock may go a long way. Then, as a courtesy to the reader, could you please define each and describe each lock type and its role: cl_lock, vvp_lock and lov_lock, and then lovsub_lock and osc_lock? What does each lock/sublock do/protect?

Also, from Patrick Farrell @ Cray: cl_lock enqueue seems to be called by various layers. Will there be multiple client locks for a single IO?

Jinshan Xiong

Thank you for taking time on reading the HLD and posting comments here.

Cl_lock will become a data structure to carry lock requirements among layers. This information is necessary to enqueue a DLM lock.

In Lustre, since a file can be striped, so there are two layers of cl_lock to describe the lock requirements on file level and stripe level. vvp_lock and lov_lock belong to file level requirements; lovsub_lock and osc_lock describes stripe level requirements. Moreover, lov_lock and lovsub_lock is defined at LOV layer where it converts file level to stripe level by the stripe info, lov_stripe_md. Those sub locks doesn't protect anything, they are just data structure to describe lock requirements, the only thing can protect data is DLM lock.

"cl_lock enqueue seems to be called by various layers. Will there be multiple client locks for a single IO?"

Yes, if read/write buffer is a lustre mmaped region, an IO will have multiple locks.

Cory Spitz

cl_lock cleanup seems OK, one set of cl_lock creations per I/O? If so, if a cl_lock doesn't exist for a set of pages can one assume that the file extent (pages) are on stable storage? (Because previously the cl_lock would transition from held->cached) What about mmap'd I/O? Is that a bad case for performance as you fault in new pages? Will every page fault result in cl_lock enqueue?

Jinshan Xiong

Yes, it will have to rebuild cl_lock for each IO.

No, we can't assume that. If DLM lock exists, the page can still be cached in client memory.

Yes, every fault-in page has to request cl_lock, but comparing to the overhead of reading a page from OST, the overhead of building cl_lock in memory should be trivial.

Cory Spitz

If cl_lock enqueue will block on conflicting locks the caller will wait/sleep. What kind of use cases would trigger this? What if the kernel is doing page reclaim or running shrinkers?

If a client had multiple threads writing to the same single shared file, would the thread I/O end on conflicting on lock enqueue for the same stripe sub-lock? What would that mean for shared file I/O performance? What does that look like and compare with the current design?

Jinshan Xiong

For example, if an application reads file [0, 4095] and then write [0,4095] may trigger this use case. Not sure why page reclaiming is related.

It totally depends on the IO pattern of the process. Since read write lock can be reused so the lock conflicting may not be severe. There is no difference comparing to current design, or b1_8 in this case.

Cory Spitz

A spin lock for osc_lock? How long is it expected to be held? Will there be a lot of contention? Now that there are lots of ptlrpc threads, do we expect trouble?

Jinshan Xiong

it should be fine. Usually there is not a lot of processes accessing one file in parallel on a single client.

Cory Spitz

Why have a separate cancel and free cl_lock operation if a lock will be destroyed when it is canceled anyway?

Jinshan Xiong

indeed. We can combine these two operations.

Cory Spitz

If cl_lock will only be used to provide a way to transfer enqueue information among layers can it be further curtailed? What else has been considered?

Jinshan Xiong

that's all for now.

Cory Spitz

Suggestion: If cl_io_iter_init is IO by stripe, can it be renamed to indicate such?

Jinshan Xiong

iter -> iteration. This is a good name. One iteration does IO to a single stripe, but a stripe can be iterated multiple times in an IO.

Cory Spitz

Re: cleanup of OBD API it was stated that “further clean-up of the configuration is possible but it is out of the scope of this design”. What further clean-up is possible? It need not be designed.

Jinshan Xiong

To move configuration callbacks somewhere else, for example, in lu_device_operations. This way we can get rid of obd_ops{} data structure. There may be other components currently using obd_ops, I didn't take a look.

Cory Spitz

If the ccc layer will be moved, what will be the new names of the interfaces in the VFS VM Posix layer?

Jinshan Xiong

vvp.

Cory Spitz

From Patrick Farrell @ Cray about the scale testing comments:

"What do they mean by scale? Client count? If so, why? Stripe count,
and/or server count? If so, again, why? It seems like this would be a
straight up performance loss at all scales, by some % due to not re-using
client locks, rather than a specifically scale related issue."
"I don't see that this would particularly result in increased network
traffic - I don't think it would harm LDLM lock re-use much, if at all,
but I may be wrong there - and if not that, then I'd think the client
changes would more closely resembled fixed per operation overhead than
something with scale issues."
Is Patrick correct in thinking that the operational behavior changes will largely, if not entirely, be constrained to a single client?

Jinshan Xiong

Sorry if we offend you by some descriptions, please let us know so that we can reword them.

By scale we mean to get a lot of hardware to exercise the code, internally and externally. It usually takes long time to stabilize lock code because there are so many use scenarios and race condition out there. Obviously we don't have that much resource to cover all the use cases.

It MAY have performance loss, and the loss may be noticeable for small IOs. It's not that "straight up" by the way, because it still has some top-to-bottom calls to reuse a lock in current design. So really hard to say how much performance loss it will be.

Cory Spitz

About stress testing with racer: Cray has been using the LTP test, mmstress, run multi-node and multi-process, which has proven to be an excellent stressor to the current CLIO code. Please consider it in addition to racer.

About testing in general: Please consider passing all tests with --enable-invariants to help ensure that the new code is entirely sane.

Jinshan Xiong

Sure, thanks for offering. Definitely I will ask you to exercise the new code before releasing it.

I know lots of customers have their own test cases, so we will try to get them involved and use their hardware to run the code.

Cory Spitz

Jinshan, thank you for all of your replies!