CLIO Simplification Solution Architecture

I. Introduction

The Lustre* file system Client IO code (CLIO) was rewritten in Lustre version 2.0. The motivation of this rewrite was to address the following requirements:

- Layer the IO code to get rid of OBD operations
- Improve portability
- Reduce the number of bugs
- Support upcoming features such as file-level replication
- Reduce stack consumption

Today, as the Lustre software prepares for the 2.6 release attention has turned to addressing technical debt inherited from this re-work. This document identifies key areas for improvement and describes the technical approach to address these problems.

II. Solution Proposal

The Client IO implementation is complex and fully understood by only a small number of developers. One reason for this complexity is due to the many levels of abstraction needed to allow the same code to integrate into the Linux VM/VFS layers as well as the MacOS and Windows VM/VFS that are substantially different in design. This complexity creates a high barrier of entry for potential contributors, and changes are more likely to introduce unintended defects. As a result, a measurable regression in performance between the 1.8 and early 2.x series of clients has persisted through a number of releases. This document proposes a solution to simplify CLIO implementation. This is a highly targeted design project to ensure the best aspects of the 2.0 client are maintained and the overall design is simplified – this is not a redesign of CLIO. The following areas will be addressed in this project:

- Simplify `cl_lock` by replacing it with a cache-less lock implementation to reduce complexity that has not shown any performance benefits
- Remove access to `lov_stripe_md` beyond LOV to isolate exposure of layout internals in the Lustre code, and facilitate upcoming changes in this area
- Remove old OBD API operations that clutter the code and confuse developers trying to understand the IO code path
- Implement an ioctl call for `cl_object` to allow passthrough of `ioctl()` requests to lower layers in the client stack and to servers

The purpose of this project is to make CLIO code easier to maintain by reducing complexity.

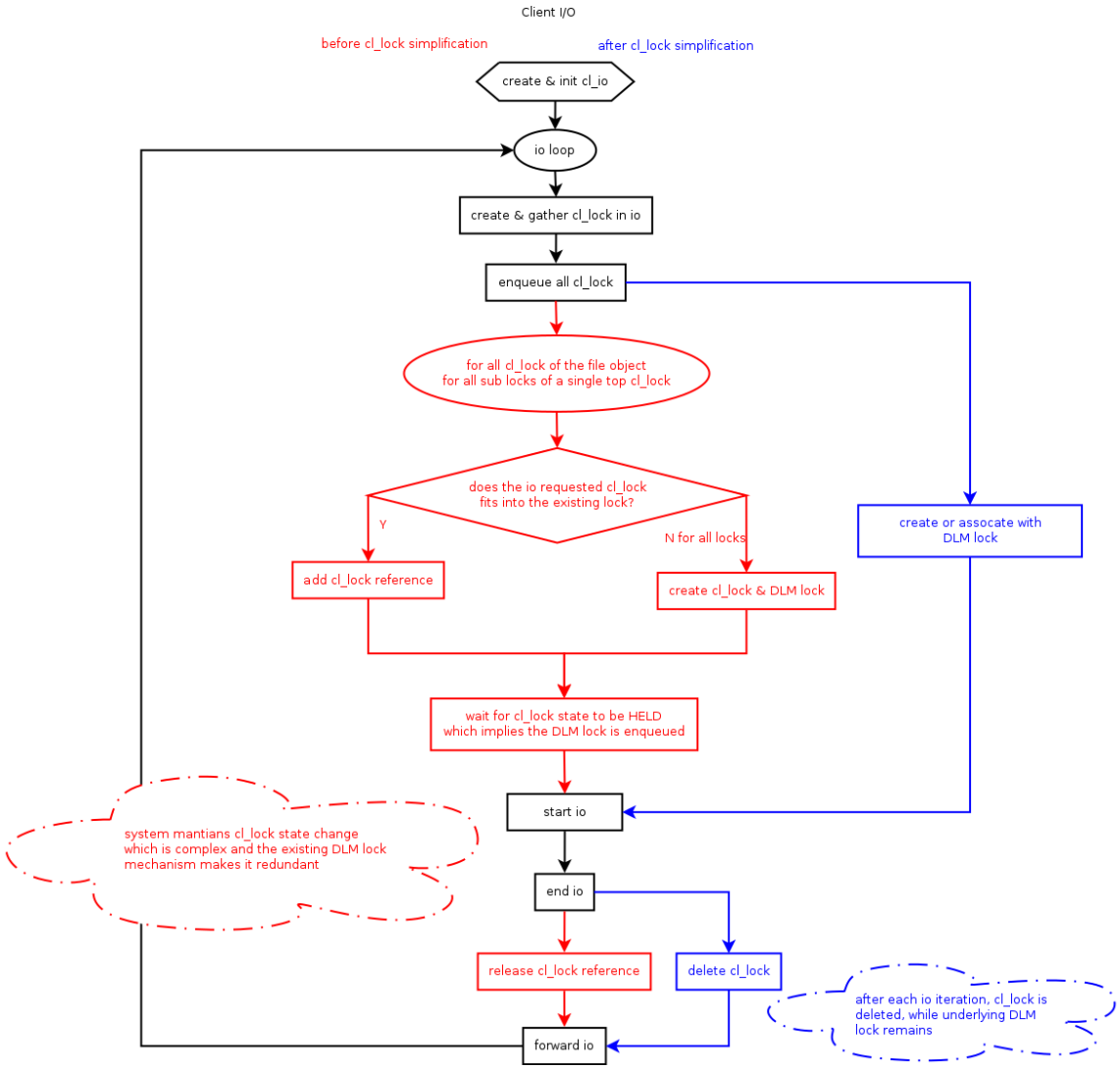
1. Simplify cl_lock

In current implementation of CLIO, `cl_lock` is a data structure that records which parts of the file have been protected by DLM lock. The current design of the current `cl_lock` intended to reduce the overhead of rebuilding memory and data structures so that better performance could be achieved - particularly for small IO. In reality, the complexity of `cl_lock` implementation has meant the anticipated performance improvements could not be realized. In addition to an unrealized performance goal, the complexity of `cl_lock` has also been a source of a number of bugs and presents a high barrier to entry for new developers.

The `cl_lock` cannot be removed altogether as it provides a mechanism to communicate the extent of a specific DLM lock for a specific IO. The new design of `cl_lock` will continue to fulfil this fundamental role: passing information between layers (from llite to OSC) so the DLM module understands the requirements for a given IO.

The `cl_lock` operations can pass as both FTTB (from top to bottom) and FBTT (from bottom to top). This design introduces the possibility of deadlock, so an additional deadlock avoidance mechanism (*closure*) was also introduced. This makes `cl_lock` implementation extremely complex and hard to maintain, as well as difficult to understand.

A cacheless `cl_lock` will be designed to replace the current implementation. In this implementation, DLM lock will still be maintained below the OSC layer. However, before each IO starts, we're going to rebuild cl_lock data structure in memory; after the IO is done, the cl_lock will be destroyed immediately.



The new `cl_lock` will

New `cl_lock` Functional Coverage

File I/O

Almost all file IOs requesting `cl_lock` will be affected by this design:

- Write
- Read
- Fault
- Setattr

After cache-less `cl_lock` is introduced, above operations will use the new `cl_lock` interfaces to request `cl_lock`.

DLM lock cancellation callback

When a DLM extent lock is canceled, it will never be used by any active IOs. It is not necessary to notify CLIO upper layers to remove the DLM lock. This design will remove the FBTT operations: just write back data covered by the DLM lock and destroy it.

Anticipated Post Implementation Effect

+ A cleaner, simpler design that will make the code easier to understand and maintain.

+ Rebuilding the `cl_lock` data structure in memory for each IO may have negitive impact on the performance of the new design under certain

workloads.

2. Remove `lov_stripe_md` access beyond LOV

One of the benefits for CLIO is that it provides a clean interface to the `lov_stripe_data` (layout) structure through the LOV layer. This abstraction allowed the `LAYOUT` lock to be implemented for HSM with a manageable effort. However, legacy code still exists in the client that accesses the `lov_stripe_data` in the `llite` layer. Most of this code is concerned with `ioctl()` operations. In this project, all references to the `lov_stripe_data` data structure outside from LOV layer will be identified and targeted for removal. There will be substantial changes needed for the file layout due to upcoming File-Level Replication and Data on MDT projects, so isolating the parts of the Lustre code that understand the internal details of the file layout is important to ensure the code will be maintainable in the future.

In current CLIO code, the following functions are accessing layout data on 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(). Some of them are obsoleted so that we can delete them; some of them are still used and we're going to reimplement them with ioctl() interface of cl_object.

Anticipated Post Implementation Effect

- Access into `lov_stripe_md` will be aligned with layout lock and layout change (introduced in 2.4).
- `lov_stripe_md` will not only be accessible and visible from the lov layer.

3. Add `ioctl()` method of `cl_object`

Many `ioctl()` functions from `ll_file_ioctl()` still use obsolete OBD API interfaces. A new method of `cl_object_operations` will be designed to replace all of the OBD interfaces. This is a fallout of reducing the access to `lov_stripe_md` outside of the `LOV` layer.

There will be an `coo_ioctl()` method to be added to each layers of CLIO object interfaces. Therefore, we can refresh layout and hold layout guard to access layout in a more protective way in `lov_object_ioctl()`.

There operations to be reimplemented include:

- `find_cbdata(struct inode *inode)`
- `ll_lov_recreate(struct inode *inode, struct ost_id *oi, obd_count ost_idx)`
- `ll_lov_getstripe(struct inode* inode, unsigned long arg)`
- `ll_do_fiemap(struct inode *inode, struct ll_user_fiemap *fiemap, int num_bytes)`
- `ll_data_version()`

Anticipated Post Implementation Effect

- `lov_stripe_md` (section 2: Remove `lov_stripe_md` access beyond LOV) will not be accessible and alternative ioctl interfaces will be provided.

4. Cleanup obsolete OBD API operations

OBD API operations for read, write, setattr, getattr, etc. became obsolete after MDT, OFD and client reconstructing were completed. Redundant code still remains for these operations in different layers of the code. All the interfaces in `obd_operations` (for example `osc_brw()`) that are not referenced by any module will be targeted for removal.

OBD operations are still in use for device configuration and that code will remain in place.

Anticipated Post Implementation Effect

- The abuse of obsolete obd interfaces will be ended.

5. Analysis of removal of non-Linux interfaces

The portability layers for Mac OS, Windows NT, and liblustre platforms consume Lustre developer resources on an ongoing basis. However, they are updated only on an "best effort" basis and are not believed to be functional at this time. There is no publicly available code that is currently using these interfaces, nor is there any mechanism to test them. The liblustre code has not been tested in several years due to limited test and developer resources, and would need a substantial investment of time and effort to become production ready again. While the MacOS client is publicly available, the code was never fully functional, and is several years old. The WinNT client port is not publicly available.

Work has already started to convert the "cross-platform" `cfs_*` wrapper functions back to their original Linux interfaces in order to match the changes needed to get the Lustre code included in the upstream kernel. Those changes are making a "best effort" attempt to keep the MacOS and WinNT wrappers in place (using the Linux API as the "portability" layer) but are not being tested in any way.

Removing this code would allow further cleanups in the CLIO code due to the removal of unnecessary abstractions that are not relevant for Linux. This would allow tighter integration of the Lustre client code with the Linux VM/VFS.

The potential negative impact of this code change is that there is reported project to implement a MacOS FUSE client using the liblustre code (though it does not use the native MacOS client code). It is unclear when or if this project might produce a usable implementation.

There is also the closed-source Windows Native Client (WNC) code that was initially implemented by Oracle and is now an asset of Xyratex. This code was implemented against pre-2.0 Lustre code, and it is unclear what state it currently is in. The main question is whether the open-source Linux client should be burdened by maintaining the WinNT portability layer and extra IO abstractions without any expectation that the WNC code will ever become publicly available.

IV. Acceptance Criteria

CLIO simplification design will be accepted as complete when:

- `cl_lock` has been designed as cache-less lock
- `llite` layer is designed not refer to `lov_stripe_md` directly
- `ioctl()` methods for `cl_object` are redesigned to avoid using OBD API interfaces
- Obsolete OBD API operations are identified and a plan on how to delete them is available

---