Client IO stack layering cleanup

Nikita Danilov <nikita@clusterfs.com>

Started: 2007.05.21

1 Introduction

This document describes changes to the lustre client io stack code, that are supposed to reduce number of bugs in the data-paths. Basic idea behind these changes is to introduce more systematic layering on the client. This HLD describes only core functionality of the new layering. Companion documents will describe specific parts of the design with more details.

Understanding of current client layering is assumed.

2 Requirements

Main goals of new layering are:

- clear layering, with well-defined functionality at each layer;
- as little of state sharing between layers as practically possible (*e.g.*, avoid current situation when Linux VM `struct page` is visible at both llite and osc layers).
- simplified layer interface;
- real stacking, *i.e.*, one where it is possible to insert or remove layers where it makes sense. Specifically, local mount and lov-less configuration have to be supported;
- support for SNS (within raidframe framework [0]);
- support for read-only peer-to-peer client caching;
- support for lock-less IO and ost intents;
- reuse of server layering from cmd3 (see `../DLD/md-api-dld.lyx`).

Other desirable properties include: portability, not changing meta-data code this time, patch-less client. Implementation of full-fledged SNS, lock-less IO, and peer-to-peer caching is not a part of current effort: new layering is designed to accommodate for these features, but not to actually implement them.

3 Functional specification

3.1 Definitions

First important principle of proposed design is that overall structure of client layers and their respective functionality remains unchanged: it is layers api that is modified (see Alternatives (section 7 on page 9) for more ambitious redesign plans). For sake of clarity, client layers will be named by their current names:

llite top-level layer, responsible for interaction with VFS, converting incoming requests (read, write, *etc.*) into api implemented by lower layers. It also implements POSIX operation semantics (*e.g.*, short reads, flock), and read-ahead.

lov layer that implements raid/SNS striping. This layer implements various raid patterns on top of underlying layers. It is supposed that real implementation will be based on raidframe library [0].

osc layer that interacts with VM, lnet and dlm, and implements data caching,

Main structuring concept behind new layering is introduction of new "**layered**" data-types similar to ones implemented for cmd3 mds stack. Layered object has private data for each layer in the stack. One general advantage is that by having explicit list of layers (referred to as "**slices**") certain operations can be reduced to iteration over layers (as opposed to recursion), thus decreasing stack space usage. Following layered types are considered:

cl_object Client object. This represents a file and an object (that is, stripe). `cl_object` is based on `lu_object`, which automatically gives us fids, hashing, lru, *etc.* Object caches pages (see below), and keeps track of dlm locks. Object can be composed of other objects.

cl_page Client page. Represents fixed-size chunk of file data, associated with a buffer in memory. Page size is the same for all pages on the client. Note, that the same page can be part of multiple `cl_object`'s — *e.g.*, page might belong to some file, and to some stripe of this file. This property is crucial for interlinking files and stripes. Within `cl_object`, page is uniquely identified by its logical offset from the beginning of file (index).

cl_lock Client lock. Represents a region (measured in page indices) of a particular file, covered by cluster lock. `cl_lock` is ultimately backed up by a dlm lock (except for the case of completely local lustre setup).

cl_io Client IO batch. A handle used to accumulate pages for IO, start IO, wait for its completion and inspect IO outcome. It is not entirely clear at this point that IO handle needs layer private data, and simpler implementation remains an option.

3.2 Striping and stacking

Existence of lov and striping is a major difference between client and server stacks. While detailed description of implementation of striping is beyond HLD scope (refer to DLD for details), few notes are in order:

- in contrast with the current code that frequently maps file offset to stripe number and offset within stripe and other way around, new code will do such mapping only once, when constructing new objects (pages and locks). At that point, object and its sub-objects are linked through pointers that are chased afterward. Perceived advantage of this is that it becomes possible to localize a state required for mapping.
- file (`cl_object`) fans out into stripes at lov level. That is, lov-private portion of `cl_object` contains an array of `cl_object`'s, each representing a stripe within file. Each of these is a first-class `cl_object` with its own pages and locks, except that set of layers is different: instead of llite and lov, stripe object contains lov_stub and osc layers, where lov_stub is a special layer used to link stripe object back to its host file. Note that top-level `cl_object` for file ends at lov layer: it doesn't have osc part.
- similarly `cl_lock` fans out into stripe sub-locks.
- `cl_page`, on the other hand, does not split in any way, because single page in a file corresponds to the single page in the stripe.

3.3 Life-time rules

In a typical scenario, new objects are created by the top-level layer in the stack (or, in the "sub-stack", corresponding to the single stripe within file, see DLD for details). Bottom layer triggers changes in the object state (*e.g.*, change of lock state on reception of AST from DLM, and change of page state on IO completion), and ultimately decides when object is to be destroyed. For file objects, destruction is initiated by the top layer.

When new object is created, layers in order are asked for "permission". This is the place where osc implements grants and cache limits, dlm lock lru size limits, *etc.*, and where lov creates sub-objects for files (stripes) and locks (stripe locks). When existing object is destroyed, layers are notified, and release per-layer resources. When object state changes, again, call-backs are called on every layer. At that point lov splits parent lock (for cancellation AST event), reads parity pages (for page write-back event), llite updates KMS (lock cancellation event), *etc.* See use cases for details.

3.4 Threading

An important and useful feature of server-side layering is ubiquitous `struct lu_context`, which is used as a cheap memory allocator (supplying a precious kernel stack), and an

additional mechanism for argument passing. Implementation of `lu_context` is tied to the server threading model, where limited threads from a pool of fixed size execute incoming requests. Number of threads on a client is not limited, and more importantly, invocations of lustre code are short lived. To efficiently use `lu_context` under these circumstances, creation and destruction of context has to be extremely fast and cheap, and `lu_context` implementation will be optimized accordingly (see DLD).

4 Logic specification

General diagram of relations between data-types:

Following table shows what state is maintained in private per-layer parts of layered objects:

	`cl_object`	`cl_page`	`cl_lock`
generic	radix tree of `cl_page`'s	index within file	start offset, end offset, lock mode
`llite`	`struct inode`	read-ahead state	tree of locks, keyed by end offset
`lov`	array of stripe `cl_object`'s	linkage into stripe row	array of stripe `cl_lock`'s
`osc`	`struct address_space`, io page queues	`struct page`	`struct ldlm_lock`

Few comments:

- ordered tree of locks is maintained by llite to simplify KMS handling: when llite is notified about destruction of last lock in the tree, it steps back to the previous lock and reduces KMS. This allows to completely encapsulate handling of KMS inside of llite.
- `struct page` refers back to the `cl_page` through `->private` pointer.
- there is no explicit linkage between locks and pages. Pages covered by lock can be efficiently found by doing gang lookups in `cl_object` page tree. Compare this with bug 10718 ("*slow lock cancellation due to excessive page walking*"), where similar solution was rejected because llite doesn't see stripe boundaries.

5 Use cases

Authoritative list of use cases is provided at [1]. In this HLD only few typical use cases are examined, the rest is covered in companion HLDs and DLDs.

5.1 read and read-ahead

- llite starts processing `read(file, buf, count)` request.
- DLM lock on file region being read is obtained (together with usual precautions about reading into mmapped buffer). New `cl_lock` object is created, and initialization call-backs are executed int bottom-to-top order:
  - lov: depending on striping parameters, calls lock creation function again (recursively) in a loop, to create a sub-lock for each stripe. This again, results in call-back execution:
    - osc: tries to match a lock locally, if that fails, enqueues lock with ldlm. Once lock is obtained, it is linked back to `cl_lock` through `->l_ast_data` pointer.
    - lov_stub: does nothing. This layer exists only to link per-stripe locks into host lock.

    Once stripe sub-lock is created, lov puts a pointer to it into array in its private part of `cl_lock`, and links the sub-lock back into the host lock (through the pointer in lov_stub-private part of the sub-lock).
  - llite: inserts lock into llite-private per-file tree (or list), indexed by end offset of the lock. This index is used to maintain KMS.
- short read detection is done completely within llite layer: KMS is maintained within llite-private portion of `cl_object` by tracking cancellation of locks (see tree of locks, keyed by end offset mentioned in the table above, that is used as an optimization for KMS calculation).
- llite calls into `generic_file_read()`.
- VM creates new `struct page`, and calls `->readpage()` method (`ll_readpage()`).
- `ll_readpage()` obtains `cl_object` from inode and calls `cl_page_get()`, that either returns existing page (found in radix tree), or creates a new one.
- To create new page, methods on each level (in the bottom-up order) are called:
  - lov: lov page creation method calls `cl_page_get()` recursively, to create new page in the `cl_object` corresponding to the stripe object this page belongs to. This call, again, results in invocation of page creation methods in bottom-up order:
    - osc: osc page creation method links `struct page` with `cl_page`, and adds `cl_page` into osc io queues. It might also implement cache size restrictions, by triggering page write-back and freeing, and waiting until cache size drops below desirable limit.
    - lov_stub: does nothing.

Once nested `cl_page_get()` call returned, lov links newly created page with top-level `cl_page`. Also, page is added into stripe row, *etc.*, as necessary. In case of mirroring, `cl_page_get()` is called multiple times, creating several `cl_pages`, all referring to the same `struct page`. It is the responsibility of osc layer to detect this situation and to coordinate further operations with shared `struct page`.

  - llite: does nothing.

- new page is added into `cl_io` handle (attached to the file descriptor in the manner similar to `struct ll_ra_read`), and read-ahead is started, then next page is processed, *etc*. When a page is added into `cl_io`, call-backs are executed top-to-bottom. At that point lov performs read balancing.

- Once enough pages are accumulated in `cl_io`, read is initiated:

  - osc forms rpc, and ships it to lnet.
  - io completion handler for each page, executes callbacks in bottom-up order:
    - osc: modifies `struct page` flags
    - lov_stub: executes callbacks in the "parent" page:
      - lov: modifies stripe state.
      - llite: does nothing.

- once all pages are processed, control returns to llite, and dlm lock is released. This is (as it looks at this stage) generic operation, requiring no per-layer processing.

As page and lock creation was described above in detail, further use cases will just mention in briefly. Moreover, examples below will not repeat tedious description of propagation of method invocation from host object (page, lock, file) to its dependent stripe sub-object(s) and back, and will instead simply assume that methods are executed through all layers.

5.2 write

Write path is quite similar to the read one with obvious changes:

- `generic_file_write()` calls `->prepare_write()` and then `->commit_write()` (note: latest 2.6 kernels have different methods).

- `->prepare_write()` creates page and reads portion of it if necessary.

- `->commit_write()` executes call-backs to notify layers that `cl_page` is now dirty. Call-backs are executed top-to-bottom:

  - llite: nothing.

  - lov: sub-pages in all mirrors are dirtied. It also might queue asynchronous read of parity page (immediately, or by a timer).
  - osc: waits for grant space.

5.3 lock LRU overflow

Once creation of new lock or changes in desirable LRU parameters (see bug 2262 "*LRU size should be controlled by the server*") cause LRU list overflow, osc takes least recently used lock and starts its cancellation:

- note, that lock being cancelled is a stripe sub-lock, attached to some stripe sub-object. Lock contains its start and end offsets within stripe.
- osc iterates over all `cl_pages` in the region of stripe `cl_object` radix tree covered by lock (by using gang lookup). For each found dirty page, page-out call-backs are executed bottom-to-top:
  - osc: modifies `struct page` flags to notify VM that page is about to be written up.
  - lov: reads in parity page if necessary. Note, that ascending through page layers, we are now at the `cl_page` attached to the host `cl_object`, representing file, and call-back is called on the `cl_page`, associated with file (rather than with stripe).
  - llite: nothing
- io completion handler executes call-backs (see read case above (subsection 5.1 on page 5)).
- once all pages were flushed to the server, lock itself is cancelled. Call-backs are invoked bottom-to-top:
  - osc: cancels dlm lock.
  - lov_stub, lov: jumps from stripe sub-lock to file lock. If "utmost" (*e.g.*, first or last) sub-lock of host lock is cancelled, start or end offset of host lock is adjusted. Otherwise (when sub-lock in the middle of host lock is cancelled), a hole is punched inside of host lock, and the latter is split into two locks:
    * new lock is created, and
    * start or end offset of existing host lock is adjusted.

    This mechanism allows upper layers (llite) to maintain precise knowledge of existing locks, which, in turn, allows handling of KMS (and probably other lock related data) to be encapsulated.

This use case also covers reception of cancellation AST from server, and invocation of `->writepage()` by VM.

5.4 Write RPC formation

When osc decides it has enough dirty pages to form an efficient RPC, it collects pages to be sent and calls the same call-backs as in case of `->writepage()` (covered in section 5.3 on the preceding page).

6 State management

Detailed changes to the state are described in DLD. It is assumed that objects, locks, and pages are reference counted, and their caching is controlled through VM memory pressure call-backs.

6.1 State invariants

Consistency invariants are described in DLD. Basic reference-consistency is assumed: e.g., if page is in a radix tree of a certain object, it contains pointer to that object, and stripe sub-lock of a file lock, is owned by stripe sub-object of corresponding file, etc. Invariants are maintained under per-object locks.

6.2 Scalability & performance

Lists of `cl_page`'s and `cl_lock`'s are protected by a spin-lock within corresponding `cl_object`. Per-layer state is protected by per-level locks. If scalability proves to be inadequate for multiprocessor clients, critical data structures will be switched to RCU.

6.3 Recovery changes

N/A

6.4 Locking changes

Detailed description of locking rules is in the scope of DLD.

6.5 Disk format changes

N/A

6.6 Wire format changes

N/A

6.7 Protocol changes

N/A

6.8 API changes

This document is all about API changes.

6.9 RPCs order changes

N/A

7 Alternatives

7.1 lov as a file system (Alex's proposal)

In this model, lov owns pages, performs caching and striping. lov provides file-system-like interface with read/write methods. osc acts as a block device: it accepts canned batches of pages (similar to `struct bio` in Linux block device layer), and implements elevator/io-scheduler functionality. Llite directly maps VFS calls onto lov interface. Advantage of this approach is that pages (and DLM locks) are owned by a single layer (lov), which allows to get rid of layered data-types, described above. It also better matches structure of a standard Linux file system. Possible disadvantage is that lov layer effectively assumes functionality of current llite and (larger part of) osc, and, also, requires more fundamental and pervasive changes to the client code base.

7.2 osc as a file system (Nikita's original proposal)

In this model, osc completely owns pages, and provides file-system-like interface: it accepts read/write/truncate calls. `generic_file_*`() functions are called within osc. Perceived advantage of that approach is that interaction with VM and DLM can be completely encapsulated within osc and, in particular extent lock cancellation requires no llite help.

Unfortunately, this design was found to contain a major flaw: lov is hard to implement on top of file-system-like layer, because it requires delicate manipulation of page state and depends on page sharing (for efficient mirroring). Putting osc *above* lov is not going to help either, as such osc would not see stripe boundaries and, hence, would inevitably fail to form efficient RPCs.

8 References

[0] RAIDframe: http://www.pdl.cmu.edu/RAIDframe/

[1] arch.wiki page with Quality Attribute Scenarios for client io cleanup: http://arch.lustre.org/index.php?titl

9 Focus for inspections

Check that all use cases are covered.

Check that goals stated in Requirements section are fulfillable with selected design.