CLIO TOI

WARNING: text below contains ascii art that requires fixed width font, and uses British spelling.

```
0. Contents
-----------
```

```
        . operations

    - locks

        . life cycle

        . top-lock and sub-locks

        . state machine

        . concurrency

        . sub-lock sharing

        . use case: lock invalidation

    - io

        . fixed io types

        . state machine

        . parallel io

        . data-flow: from stack to io slice

    - transfer

        . immediate vs. opportunistic

        . page lists

        . states: prepare, completion

        . page completion handlers, synchronous transfer

    - lu_env

        . motivation, server usage

        . client usage

        . sub-environments

    - use cases

        . inode creation

        . first IO to a file

            + read, read-ahead

            + write

        . cached IO

        . lock-less and no-cache IO
```

1. Overview

1.1. Goals:

....

CLIO is a re-write of interfaces between layers in the client data-path (read, write, truncate). Its goals are:

- reduce the number of bugs in the IO path;
- introduce more logical layer interfaces instead of current all-in-one obd device interface;
- define clear and precise semantics for the interface entry points;
- simplify structure of the client code.
- support upcoming features:
    - . SNS,
    - . p2p caching,
    - . parallel non-blocking IO;
    - . pNFS;
- reduce stack consumption.

Restrictions:

- no meta-data changes;
- no 2.4 kernels support;
- portable code;
- no changes to recovery;
- the same layers with mostly the same functionality;
- as few changes to the core logic of each Lustre data-stack layer as possible (e.g., no changes to the read-ahead or osc RPC logic).

1.2. Terminology:

....

Any discussion of client functionality has to talk about `read' and `write' system calls on the one hand and about `read' and `write' requests to the server on the other hand. To avoid confusion, the former high level operations are called `IO', while the latter are called `transfer'.

Many concepts apply uniformly to pages, locks, files, and io contexts, for example, reference counting, caching, etc. To describe such situations, a common term is needed to denote things from any of the above classes. `Object' would be a natural choice, but files are especially stripes are already called objects, so `entity' is used instead.

Due to the striping it's often a case that some entity is composed of the multiple entities of the same kind: a file is composed of stripe objects, a logical lock on a file is composed of stripe locks on file's stripes, etc. In these cases we shall talk about top-object, top-lock, top-page, top-io, etc. being constructed from sub-objects, sub-locks, sub-pages, sub-io's respectively.

The topmost module in the Linux client, is traditionally known as `llite'. Corresponding CLIO layer is called `vvp' (Vfs, Vm, Posix) to reflect its functional responsibilities. Top-level layer for liblustre is called `slp'. vvp and slp share a lot of logic of data-types. Their common functions and types are prefixed with `ccc' prefix.

1.3. Main differences with the existing client code:

- locks on files (as opposed to locks on stripes) are first-class objects;

- sub-objects (stripes) are first class objects;

- stripe-related logic is moved out of llite (almost);

- io control flow is different:

  . HEAD: llite implement control flow, calling underlying obd methods as necessary;

  . CLIO: generic code (cl_io_loop()) controls IO logic calling all layers, including vvp.

  In other words, vvp (or any other top-layer) instead of calling some pre-existing `lustre interface', also implements parts of this interface.

- lu_env allocator from MDT is used on a client.

1.4. Layered objects:

CLIO continues the layered object approach that was found to be useful for cmd3 mdt stack. In this approach instances of key object types (files, pages, locks, etc.) are represented as a header, containing attributes shared by all layers, from which hangs off a linked list of per-layer `slices'. Each slice contains a pointer to a vector of function pointers. Generic operations on layered objects are implemented by going through the list of slices and invoking corresponding function from the operation vector at every layer. This way generic object behavior is delegated to the layers.

For example, page is represented by struct cl_page, from which hangs off a list of cl_page_slice structures, one for each layer in the stack. cl_page_slice contains a pointer to struct cl_page_operations. cl_page_operations has

```
        void (*cpo_completion)(const struct lu_env *env,
                               const struct cl_page_slice *slice, int ioret);
```

field. When transfer of a page is finished, ->cpo_completion() methods are in a particular order (bottom to top in this case).

Allocation of slices is done during instance creation. If layer needs some private state for an object, it embeds slice into its own data structure. For example, osc layer defines

```
struct osc_lock {
        struct cl_lock_slice  ols_cl;
        struct ldlm_lock     *ols_lock;
        ...
};
```

When an operation from cl_lock_operations is called, it is given a pointer to struct cl_lock_slice, and layer casts it to its private structure (for example, struct osc_lock) to access per-layer state.

Following types of layered objects exist in CLIO:

- file system objects (files and stripes): struct cl_object_header, slices are of type struct cl_object;

- cached pages with data: struct cl_page, slices are of type cl_page_slice;

- extent locks: struct cl_lock, slices are of type cl_lock_slice;

- IO content: struct cl_io, slices are of type cl_io_slice;

- transfer request: struct cl_req, slices are of type cl_req_slice.

1.5. Instantiation:

Entities with different sequences of slices can co-exist. Typical example of this is a local vs. remote object on mdt server. Local object, based on some file in the local file system has mdt, cmm, mdd and osd as its layers, whereas remote object (representing an object local to some other mdt) has mdt, cmm, mdc layers.

When client is being mounted, its device stack is configured according to llog configuration records. Typical configuration is

```
          vvp_device
               |
               V
          lov_device
               |
     +---+---+---+---+
     |   |   |   |   |
     V   V   V   V   V
  .... osc_device's ....
```

In this tree every node knows its descendants. When new file (inode) is created, every layer, starting from the top, creates a slice, with a state and an operation vector for this layer, appends this slice to the tail of a list anchored at the object header, and then calls corresponding lower layer device to do the same. That is, file object structure is determined by the configuration of devices to which this file belongs.

Pages and locks, in turn, belong to the file objects, and when a new page is created for a given object, slices of this object are iterated through and every slice is asked to initialise a new page, which includes (usually) allocation of a new page slice and its insertion into a list of page slices. Similarly for locks and IO contexts.

1.6. Life cycle:

All layered objects except io contexts and transfer requests (which leaves file objects, pages and locks) are reference counted and cached. They have uniform caching mechanism:

- objects are kept in some sort of an index (global fid hash for file objects, per-file radix tree for pages, and per-file list for locks);

- a reference for an object can be acquired by cl_{object,page,lock}_find() functions, that check the index, and if object is not there, create new one and insert it into the index;

- a reference is released by cl_{object,page,lock}_put() function. When the last reference is released, the object is returned to the cache (still in the index), except when the user explicitly set `do not cache' flag for this object. In the latter case the object is destroyed immediately.

IO contexts are owned by a thread (or, potentially a group of threads) doing IO, and need neither reference counting nor indexing. Similarly, transfer requests are owned by osc device, and their life time is from RPC creation until completion notification.

1.7. State machines:

All types of layered objects contain a state-machine inside, although for the transfer requests this machine is trivial (CREATED -> PREPARED -> INFLIGHT -> COMPLETED), and for the file objects it is very simple, see object-state.png.

Page (in page-state.png), lock (in lock-state.png) and io state machines are described in more detail below.

As a generic rule, state machine transitions are made under some kind of lock: VM lock for a page, a per-lock mutex for a cl_lock, and site spin-lock for an object. After some event that might cause state transition happens, such lock is taken, and object state is analysed to check whether transition is possible. If it is, state machine is advanced to the new state and lock is released. IO state transitions do not require concurrency control.

1.8. Finalisation:

State machine and reference counting interact during object destruction. In addition to temporary pointers to an entity (that are counted in its reference counter), entity is reachable through

- indexing structures described above, and

- pointers internal to some layer of this entity. For example, page is reachable through a pointer from VM page, lock might be reachable through a ldlm_lock::l_ast_data pointer, and sub-{lock,object,page} might be reachable through a pointer from its top-entity.

Entity destruction happens in three phases:

- first, a decision is made to destroy an entity, when, for example, when a lock is cancelled, or a page is truncated from a file. At this point `do not cache' bit is set in the entity header, and all ways to reach entity from `internal' pointers are severed.

  cl_{page,lock,object}_get() functions never return entity with `do not cache' bit set, so from this moment no new internal pointers can be obtained too.

  See: cl_page_delete(), cl_lock_delete();

- for some time pointers `drain' as existing references are released. In this phase entity is reachable through

```
        . temporary pointers, counted in its reference counter, and

        . possibly a pointer in the indexing structure.

    - when last reference is released, entity can be safely freed (after
      possibly removing it from the index).

      See lu_object_put(), cl_page_put(), cl_lock_put().
```

1.9. Code structure:
....

All global CLIO data-types are defined in include/cl_object.h header that contains detailed documentation. Generic clio code is in obdclass/cl_{object,page,lock,io}.c

An implementation of CLIO interfaces for a layer foo is located in foo/foo_{dev,object,page,lock,io}.c files, with (temporary) exception of liblustre code that is located in liblustre/llite_cl.c.

Definitions of data-structures shared within a layer are in foo/foo_cl_internal.h

vvp and slp share most of functionality and data-structures. Common functions are defined in the lustre/lclient directory, and common types are in the lustre/include/lclient.h header.

```
{llite,lov,osc}/*_{dev,object,lock,page,io}.c
liblustre/llite_cl.c
lclient/*.c
obdclass/cl_*.c
include/cl_object.h
```

FILE	DECLARATIONS	LINES	NON-BLANK	NON-COMMENT
llite/vvp_dev.c	30	559	470	404
llite/vvp_object.c	7	144	114	74
llite/vvp_lock.c	3	90	69	29
llite/vvp_page.c	32	555	462	377
llite/vvp_io.c	30	991	845	696
lov/lov_dev.c	23	515	432	391
lov/lovsub_dev.c	12	212	168	121
lov/lov_object.c	31	651	561	441
lov/lovsub_object.c	7	143	111	76
lov/lov_lock.c	24	784	677	558
lov/lovsub_lock.c	15	433	365	255
lov/lov_page.c	10	227	187	141
lov/lovsub_page.c	3	83	63	28
lov/lov_io.c	32	851	740	658
lov/lovsub_io.c	0	55	38	2
osc/osc_dev.c	17	253	204	163
osc/osc_object.c	14	219	176	133
osc/osc_lock.c	47	1559	1342	929
osc/osc_page.c	20	388	320	264
osc/osc_io.c	17	411	338	231
liblustre/llite_cl.c	42	836	680	614
lclient/glimpse.c	5	253	225	163
lclient/lcommon_cl.c	63	1134	952	738
obdclass/cl_io.c	69	1589	1383	1051
obdclass/cl_lock.c	69	1900	1634	1218
obdclass/cl_object.c	50	1014	863	637

```
obdclass/cl_page.c          59       1521       1325        948
include/cl_object.h         63       2989       2519        855

TOTAL                      794      20359      17263      12195
```

2. Layers

This section briefly outlines responsibility of every layer in the stack. More detailed description of functionality is in the following sections on objects, pages and locks.

2.1. vvp, slp, echo-client:

There are currently 3 options for the top-most Lustre layer:

- vvp: linux kernel client,
- slp: liblustre client, and
- echo-client: special client used by the Lustre testing sub-system.

Other possibilities are:

- client ports to other operating systems (OSX, Windows, Solaris),
- pNFS and NFS exports.

The responsibilities of the top-most layer include:

- definition of the entry points through which Lustre is accessed by the applications;
- interaction with the hosting VM/MM system;
- interaction with the hosting VFS or equivalent;
- implementation of the desired semantics of top of Lustre (e.g., POSIX, or Win32 semantics).

Let's look at vvp in more detail. First, vvp implements VFS entry points required by the Linux kernel interface: ll_file_{read,write,sendfile}(). Then, vvp implements VM entry points: ll_{write,invalidate,release}page().

For file objects, vvp slice (ccc_object, shared with liblustre) contains a pointer to inode.

For pages, vvp slice (ccc_page) contains a pointer to the VM page (cfs_page_t), `defer up to date' bit to track read-ahead hits (similar to the HEAD client), and fields necessary for synchronous transfer (see below). vvp is responsible for implementation of interaction between client page (cl_page) and VM.

There is no special vvp private state for locks.

For io, vvp implements

- mapping from Linux specific entry points (readv, writev, sendfile, etc.) to Lustre IO loop,
- mmap,

- POSIX features like short reads, O_APPEND atomicity, etc.

- read-ahead (this is arguably not the best layer to implement read-ahead in, as existing read-ahead algorithm is network aware).

2.2. lov, lovsub:

lov layer implements raid0 striping. It maps top-entities (file objects, locks, pages, io's) to one or more sub-entities. lovsub is a companion layer doing reverse mapping.

2.3. osc:

osc layer deals with networking stuff:

- it decides when efficient rpc can be formed from cached data;

- it calls LNET to initiate a transfer and to get notification of completion;

- it calls LDLM to implement distributed cache coherency, and to get notifications of lock cancellation requests;

3. Objects

3.1. Fid, hashing, caching, LRU:

Files and stripes are collectively known as (file system) `objects'. CLIO client re-uses support for layered objects from MDT stack. Both client and MDT objects are based on struct lu_object type, representing a slice of a file system object. lu_object's for a given object are linked through ->lo_linkage field into a list hanging off field ->loh_layers of struct lu_object_header, that represents whole layered object.

lu_object and lu_object_header provide functionality common between a client and a server:

- object is uniquely identified by a fid, all objects are kept in a hash table, indexed by a fid;

- objects are reference counted. When last reference to an object is released it is returned back into the cache, unless it has been explicitly marked for deletion, in which case it is immediately destroyed;

- objects in the cache are kept in a LRU list, that is scanned to keep cache size under control.

On MDT, lu_object is wrapped into struct md_object where additional state that all server-side objects have, is stored. Similarly, on a client, lu_object and lu_object_header are embedded into struct cl_object and struct cl_object_header where additional client state is stored.

cl_object_header contains following additional state:

- ->coh_tree: a radix tree of cached pages for this object. In this tree pages are indexed by their logical offset from the beginning of this object. This tree is protected by ->coh_page_guard spin-lock;

- ->coh_locks: a double-linked list of all locks for this object. Locks in all possible states (see Locks section below) are threaded on this list without any particular ordering.

3.2. Top-object, sub-object:
....

Important distinction with the server side, where md_object and dt_object are used, is that cl_object "fans out" at the lov level: depending on the file layout, single file is represented as a set of "sub-objects" (stripes). At the implementation level, struct lov_object contains an array of cl_objects. Each sub-object is a full-fledged cl_object, having its fid, living in the lru and hash table. Each sub-object has its own radix tree of pages, and its own list of locks.

This leads to the next important difference with the server side: on the client, it's quite usual to have objects with the different sequence of layers. For example, typical top-object is composed of the following layers:

- vvp

- lov

whereas its sub-objects are composed of

- lovsub

- osc

layers. Here "lovsub" is a mostly dummy layer, whose purpose is to keep track of the object-subobject relationship:

```
                               cl_object_header-+--->radix tree of pages
                                       |        |
                                       V        +--->list of locks
                        inode<----ccc_object
                                       |
                                       V
                                  lov_object
                                       |
                               +---+---+---+---+
                               |   |   |   |   |
                               V   |   |   |   |
                  cl_object_header |   .   .   .
                        |          |   .   .   .
                        |          V
                        .  cl_object_header-+--->radix tree of pages
                                   |        |
                                   V        +--->list of locks
                             lovsub_object
                                   |
                                   V
                              osc_object
```

Sub-objects are not cached independently: when top-object is about to be discarded from the memory, all its sub-objects are torn-down and destroyed too.

3.3. Operations:

....

In addition to lu_object_operations vector, each cl_object slice has cl_object_operations. lu_object_operations deal object creation and destruction of objects. Client specific cl_object_operations fall into two categories:

- creation of dependent entities: these are ->coo_{page,lock,io}_init() methods called at every layer when new page, lock or io context are being created, and

- object attributes: ->coo_attr_{get,set}() methods that are called to get or set common client object attributes (struct cl_attr): size, [mac]times, etc.

3.4. Attributes:
....

cl_object comes with a set of attributes, defined by struct cl_attr. Attributes include object size, object known-minimum-size (KMS), access, change and modification times and ownership identifiers. Description of KMS is beyond the scope of this document, refer to the (non-)existent Lustre documentation on the subject.

Both top-objects and sub-objects have attributes. Consistency of the attributes is protected by a lock on the top-object, accessible through cl_object_attr_{un,}lock() calls. This allows to change sub-object and its top-object attributes atomically.

Attributes are accessible through cl_object_attr_{g,s}et() functions that call per-layer ->coo_attr_{s,g}et() object methods. Top-object attributes are calculated from the sub-object ones by lov_attr_get() that optimises for the case when none of sub-object attributes have changed since last call to lov_attr_get().

As a further potential optimisation (proposed by Oleg) one can recalculate top-object attributes at the moment when any sub-object attribute is changed. This would allow to avoid collecting cumulative attributes over all sub-objects. To implement this optimisation _all_ changes of sub-object attributes must go through cl_object_attr_set().

4. Pages
----

cl_page represents a portion of a file, cached in the memory. All pages of the given file are of the same size, and are kept in the radix tree hanging off the cl_object_header.

cl_page is associated with a VM page of the hosting environment (struct page in the Linux kernel, for example), cfs_page_t. It is assumed, that this association is implemented by one of cl_page layers (top layer in the current design) that

- intercepts per-VM-page call-backs made by the environment (e.g., memory pressure),

- translates state (page flag bits) and locking between lustre and environment.

The association between cl_page and cfs_page_t is immutable and established when cl_page is created. It is possible to imagine a setup where different pages get their backing VM buffers from different sources. For example, in the

case if pNFS export, some pages might be backed by local DMU buffers, while
others (representing data in remote stripes), by normal VM pages.

4.1. Indexing:
....

Pages within given object are linearly ordered. Page index is stored it
->cpo_index field. In a typical Lustre setup, top-object has an array of
sub-objects, and every page in a top-object corresponds to a page in some of
its sub-object. This second page (a sub-page of a first), is a first class
cl_page, and, in particular, it is inserted into sub-object radix tree, where
it is indexed by its offset within sub-object. Sub-page and top-page are
linked together through ->cp_child and ->cp_parent fields in struct cl_page:

```
                      +------>radix tree of pages
                      |                  /|\
                      |                 /.|.\
                      |                 ..V..
          cl_object_header<------------cl_page<-----------------+
                  |          ->cp_obj     |                     |
                  V                       V                     |
   inode<----ccc_object<---------------ccc_page---->cfs_page_t  |
                  |        ->cpl_obj      |                     |
                  V                       V                     | ->cp_child
             lov_object<---------------lov_page                 |
                  |        ->cpl_obj                            | ->cp_parent
          +---+---+---+---+                                     |
          |   |   |   |   |                                     |
          .   |   .   .   .                                     |
              |                                                 |
              |   +------>radix tree of pages                   |
              |   |                       /|\                   |
              |   |                      /.|.\                  |
              V   |                      ..V..                  |
      cl_object_header<-----------------cl_page<----------------+
              |            ->cp_obj        |
              V                            V
        lovsub_object<-----------------lovsub_page
              |            ->cpl_obj       |
              V                            V
         osc_object<--------------------osc_page
                           ->cpl_obj
```

4.2. Ownership:
....

cl_page can be "owned" by a particular cl_io (see below), guaranteeing this io
an exclusive access to this page w.r.t. other io attempts and various events
changing page state (such as transfer completion, or eviction of the page from
the memory). Note, that in general cl_io cannot be identified with a
particular thread, and page ownership is not exactly equal to the current
thread holding a lock on the page. Layer implementing association between
cl_page and cfs_page_t has to implement ownership on top of available
synchronisation mechanisms.

While Lustre client maintains the notion of an page ownership by io, hosting
MM/VM usually has its own page concurrency control mechanisms. For example, in
Linux, page access is synchronised by the per-page PG_locked bit-lock, and
generic kernel code (generic_file_*()) takes care to acquire and release such
locks as necessary around the calls to the file system methods (->readpage(),
->prepare_write(), ->commit_write(), etc.). This leads to the situation when
there are two different ways to own a page in the client:

- client code explicitly and voluntary owns the page (cl_page_own());

- VM locks a page and then calls the client, that has "to assume" the ownership from the VM (cl_page_assume()).

Dual methods to release ownership are cl_page_disown() and cl_page_unassume().

4.3. Transfer locking:
....

cl_page implements simple locking design: As noted above, page is protected by VM lock while IO owns it. The same lock is kept while page is in transfer. Note that this is different from the standard Linux kernel behavior where page write-out is protected by a lock (PG_writeback) separate from VM lock (PG_locked). It is felt that single-lock design is more portable and moreover, Lustre cannot benefit much from separate write-out lock, due to LDLM locking.

4.4. Operations:
....

See documentation for cl_object.h:cl_page_operations. See cl_page state descriptions in documentation for cl_object.h:cl_page_state.

5. Locks
----

struct cl_lock represents an extent lock on cached file or stripe data. cl_lock is used only to maintain distributed cache coherency and provides no intra-node synchronisation. It should be noted that, as other Lustre DLM locks, cl_lock is, actually, a lock _request_, rather than lock itself.

As locks protect cached data, and the unit of data caching is a page, locks are of page granularity.

5.1. Life cycle:
....

Locks for a given file are cached in a per-file double linked list. Overall lock life cycle is as following:

- lock is created in CLS_NEW state. At this moment lock doesn't actually protects anything;

- lock is enqueued, that is, sent to server, passing through CLS_QUEUING state. In this state some multiple network communications with multiple servers can occur;

- once fully enqueued, lock moves into CLS_ENQUEUED state where it waits for a final reply from the server or servers;

- when a reply, granting this lock, is received, lock moves into CLS_HELD state. In this state lock protects file data, and pages in the lock extent can be cached (and dirtied for a write lock);

- when lock is no actively used, it is `unused' and, moving through CLS_UNLOCKING state, lands in CLS_CACHED state. In this state lock still protects cached data. The difference with CLS_HELD state is that in CLS_CACHED state lock can be cancelled;

- ultimately, lock is either cancelled, or destroyed without cancellation. In any case, it is moved in CLS_FREEING state and eventually freed.

  Lock can be cancelled by a client either voluntary (to react to the memory pressure, by explicit user request, or as part of early cancellation), or involuntary, when blocking AST arrives.

  Lock can be destroyed without cancellation when its object is destroyed (there should be no cached data at this point), or during eviction (when cached data are invalid too);

- if an unrecoverable error occurs at any point (e.g., due to network timeout, or server's refusal to grant a lock), lock is moved into CLS_FREEING state.

The description above matches slow IO path. In the common fast path there already is a cached lock covering the extent IO is going against. In this case, cl_lock_find() function finds cached lock. If found lock is in CLS_HELD state, it can be used to for IO immediately. If found lock is in CLS_CACHED state, it is yanked from the cache and transitions to CLS_HELD. If lock is in CLS_QUEUING or CLS_ENQUEUED states, some other IO is currently in the process of en-queuing it, and current thread helps that other thread by continuing enqueue.

Actually the process of finding a lock in the cache is more involved, because there are cases when lock matching IO extent and mode still cannot used for this IO. For example, locks covering multiple stripes cannot be used for regular IO, due to the danger of cascading evictions. For such situations, every layer can optionally define cl_lock_operations::clo_fits_into() method that might declare a given lock unsuitable for a given IO. See lov_lock_fits_into() as an example.

5.2. Top-lock and sub-locks:
....

Top-lock protects cached pages of a top-object, and is based on a set of sub-locks, protecting cached pages of sub-objects:

```
                      +--------->list of locks
                      |                    |
                      |                    V
          cl_object_header<-------------cl_lock
                   |          ->cld_obj    |
                   V                       V
             ccc_object<----------------ccc_lock
                   |        ->cls_obj      |
                   V                       V
             lov_object<----------------lov_lock
                   |        ->cls_obj      |
          +---+---+---+---+            +---+---+---+---+
          |   |   |   |   |            |   |   |   |   |
          .   |   .   .   .            .   .   .   |   .
              |                                    |
              |   +--------------->list of locks   |
              |   |                            |   |
              V   |                            V   V
      cl_object_header<-----------------------cl_lock
              |            ->cp_obj              |
              V                                  V
        lovsub_object<----------------------lovsub_lock
              |            ->cls_obj             |
              V                                  V
         osc_object<-------------------------osc_lock
                           ->cls_obj
```

When top-lock is created, it creates sub-locks based on the striping method (raid0 currently). Sub-locks are `created' in the same manner as top-locks: by calling cl_lock_find() function, so that cache works. To enqueue a top-lock all of its sub-locks have to be enqueued also, with ordering constraints defined by enqueue options:

- to enqueue a regular top-lock, each sub-lock has to be enqueued and granted before next one can be enqueued. This is necessary to avoid dead-locks;

- for `try-lock' style top-lock (e.g., a glimpse request, or O_NONBLOCK IO locks), requests can be enqueued in parallel, because dead-lock is not possible.

Sub-lock state depends on its top-lock state:

- when top-lock is being enqueued, its sub-locks are in QUEUING, ENQUEUED, or HELD state;

- when top-lock is in HELD state, its sub-locks are in HELD state too;

- when top-lock is in CACHED state, its sub-locks are in CACHED state too;

- when top-lock is in FREEING state, it detaches itself from all sub-locks, that are usually deleted too.

Sub-lock can be cancelled while top-lock is in CACHED state. To maintain an invariant that CACHED lock is immediately ready for re-use by IO, top-lock is moved into NEW state. Next attempt to use this lock will enqueue it again, resulting in creation and enqueue of a missing sub-lock.

As follows from the description above, top-lock provides somewhat weaker guarantees than one might expect:

- sub of its sub-locks can be missing, and

- top-lock does not necessary protects whole its extent.

In other words, top-lock is potentially porous, and in effect, it is just a hint, describing what sub-locks are likely to exist. Nonetheless, in the most important cases of a file per client, and of clients working in the disjoint areas of a shared file this hint is precise.

5.3. State machine:

....

cl_lock is a state machine. This requires some clarification. One of the goals of CLIO is to make IO path non-blocking, or at least to make it easier to make it non-blocking in the future. Here `non-blocking' means that when a system call (read, write, truncate) reaches a situation where it has to wait for a communication with the server, it should --instead of waiting-- remember its current state and switch to some other work. E.g,. instead of waiting for a lock enqueue, client should proceed doing IO on the next stripe, etc. Obviously this is rather radical redesign, and it is not planned to be fully implemented at this time, instead we are putting some infrastructure in place, that would make it easier to do asynchronous non-blocking IO easier in the future. Specifically, where old locking code goes to sleep (waiting for enqueue, for example), new code returns cl_lock_transition::CLO_WAIT. When enqueue reply comes, its completion handler signals that lock state-machine is ready to transit to the next state. There is some generic code in cl_lock.c that sleeps, waiting for these signals. As a result, for users of this cl_lock.c code, it looks like locking is done in the normal blocking fashion, and it the same time it is possible to switch to the non-blocking locking (simply by returning cl_lock_transition::CLO_WAIT from cl_lock.c functions).

For a description of state machine states and transitions see enum cl_lock_state.

There are two ways to restrict a set of states which lock might move to:

- placing a "hold" on a lock guarantees that lock will not be moved into cl_lock_state::CLS_FREEING state until hold is released. Hold can be only acquired on a lock that is not in cl_lock_state::CLS_FREEING. All holds on a lock are counted in cl_lock::cll_holds. Hold protects lock from cancellation and destruction. Requests to cancel and destroy a lock on hold will be recorded, but only honoured when the last hold on a lock is released;

- placing a "user" on a lock guarantees that lock will not leave cl_lock_state::CLS_NEW, cl_lock_state::CLS_QUEUING, cl_lock_state::CLS_ENQUEUED and cl_lock_state::CLS_HELD set of states, once it enters this set. That is, if a user is added onto a lock in a state not from this set, it doesn't immediately enforce lock to move to this set, but once lock enters this set it will remain there until all users are removed. Lock users are counted in cl_lock::cll_users.

  User is used to assure that lock is not cancelled or destroyed while it is being enqueued, or actively used by some IO.

  Currently, a user always comes with a hold (cl_lock_invariant() checks that a number of holds is not less than a number of users).

Lock "users" are used by the top-level IO code to guarantee that a lock is not cancelled when IO it protects is going on. Lock "holds" are used by a top-lock (lov code) to guarantee that its sub-locks are in expected state.

5.4. Concurrency:

This is how lock state-machine operates. struct cl_lock contains a mutex cl_lock::cll_guard that protects struct fields.

- mutex is taken, and cl_lock::cll_state is examined.

- for every state there are possible target states where lock can move into. They are tried in order. Attempts to move into next state are done by _try() functions in cl_lock.c:cl_{enqueue,unlock,wait}_try().

- if the transition can be performed immediately, state is changed, and mutex is released.

- if the transition requires blocking, _try() function returns cl_lock_transition::CLO_WAIT. Caller unlocks mutex and goes to sleep, waiting for possibility of lock state change. It is woken up when some event occurs, that makes lock state change possible (e.g., the reception of the reply from the server), and repeats the loop.

Top-lock and sub-lock has separate mutices and the latter has to be taken first to avoid dead-lock.

To see an example of interaction of all these issues, take a look at the lov_cl.c:lov_lock_enqueue() function. It is called as a part of cl_enqueue_try(), and tries to advance top-lock to ENQUEUED state, by advancing state-machines of its sub-locks (lov_lock_enqueue_one()). Note also, that it uses trylock to grab sub-lock mutex to avoid dead-lock. It also has to handle CEF_ASYNC enqueue, when sub-locks enqueues have to be done in parallel, rather than one after another (this is used for glimpse locks, that cannot dead-lock).

```
             +------------------>NEW
             |                    |
             |                    | cl_enqueue_try()
             |                    |
             |    cl_unuse_try()  V
             |  +--------------QUEUING (*)
             |  |                 |
             |  |                 | cl_enqueue_try()
             |  |                 |
             |  | cl_unuse_try()  V
   sub-lock  |  +-------------ENQUEUED (*)
  cancelled  |  |                 |
             |  |                 | cl_wait_try()
             |  |                 |
             |  |                 V
             |  |                HELD<---------+
             |  |                 |            |
             |  |                 |            |
             |  |  cl_unuse_try() |            |
             |  |                 |            |
             |  |                 V            | cached
             |  +------------>UNLOCKING (*)    | lock found
             |                    |            |
             |     cl_unuse_try() |            |
             |                    |            |
             |                    |            | cl_use_try()
             |                    V            |
             +------------------CACHED---------+
                                  |
                             (Cancelled)
                                  |
                                  V
                               FREEING
```

5.5. Shared sub-locks:
....

For various reasons, the same sub-lock can be shared by multiple top-locks. For example, large sub-lock can match multiple small top-locks. In general, a sub-lock keeps a list of all its parents, and propagates certain events to them, e.g., as described above, when a sub-lock is cancelled, it moves _all_ of its top-locks from CACHED to NEW state.

This leads to a curious situation, when an operation on some top-lock (e.g., enqueue), changes state of one of its sub-locks, and this change has to be propagated to the other top-locks of this sub-lock. Resulting locking pattern is top->bottom->top, which is, obviously not dead-lock safe. To avoid dead-locks, try-locking is used in such situations. See cl_object.h:cl_lock_closure documentation for details.

5.6. Use case: lock invalidation:

....

To demonstrate how object, page and lock data-structures interact, let's look at the example of stripe lock invalidation.

Imagine that on the client C0 there is a file object F, striped over stripes S0, S1 and S2 (file and stripes are represented by cl_object_header). Further, there is a write lock LF, for the extent [a, b] (recall, that lock extents are measured in pages) of file F. This lock is based on write sub-locks LS0, LS1 and LS2 for the corresponding extents of S0, S1 and S2 respectively.

All locks are in CACHED state. Each LSi sub-lock has osc_lock slice, where a pointer to the struct ldlm_lock is stored. A ->l_ast_data field of ldlm_lock pointer to, points back to sub-lock's osc_lock.

Client caches clean and dirty pages for F, some in [a, b], some outside of it (these latter are necessary covered by some other locks). Each of these pages is in F's radix tree, and points through cl_page::cp_child to a sub-page which is in radix-tree of one of Si's.

Some other client requests a lock that conflicts with LS1. OST, where S1 lives, sends blocking AST to C0.

C0's ldlm invokes lock->l_blocking_ast(), which is osc_ldlm_blocking_ast(), that eventually calls acquires a mutex on the sub-lock and calls cl_lock_cancel(sub-lock). cl_lock_cancel() ascends through sub-lock's slices (which are osc_lock and lovsub_lock), calling ->clo_cancel() method at every stripe, that is, calling osc_lock_cancel() (lovsub layer doesn't define ->clo_cancel()).

osc_lock_cancel() calls cl_lock_page_out() to invalidate all pages cached under this lock after sending dirty ones back to the S1's server.

To do this, cl_lock_page_out() obtains sub-lock's object and sweeps through its radix tree from start to end offset of a sub-lock (recall that a sub-lock extent is measured in page offsets within a sub-object). For every page thus found cl_page_unmap() function is called to invalidate it. This function goes through sub-page slices bottom-to-top, then follows ->cp_parent pointer to go to the top-page and repeats the same process. Eventually vvp_page_unmap() is called that unmaps a page (top-page by this time) from page tables.

After page is invalidated, it is prepared for transfer if it is dirty. This step also includes bottom-to-top scan of page and top-page slices, and calls to ->cpo_prep() methods at each layer, allowing vvp_page_prep_write() to announce to VM that VM page is being written.

Once all pages were written, they are removed from radix-trees and destroyed.

This completes invalidation of a sub-lock, and osc_lock_cancel() exits.

Note that:

- no special cancellation logic for the top-lock is necessary;

- specifically, vvp knows nothing about striping and there is no need to handle the case where only part of top-lock is cancelled;

- there is no need to convert between file and stripe offsets during this process;

- there is no need to keep track of locks protecting given page.

6. IO

IO context (struct cl_io) is a layered object where data describing the state of an ongoing IO operation (such as a system call) are stored.

6.1. Fixed io types:

There are two classes of IO contexts, represented by cl_io:

- an IO for a specific type of client activity, enumerated by enum cl_io_type:
    - CIT_READ: read system call including read(2), readv(2), pread(2), sendfile(2);
    - CIT_WRITE: write system call;
    - CIT_TRUNC: truncate system call;
    - CIT_FAULT: page fault handling;
- and a `catch-all' CIT_MISC IO type for all other IO activity:
    - cancellation of an extent lock,
    - VM induced page write-out,
    - glimpse,
    - other miscellaneous stuff.

The difference between CIT_MISC and other IO types is that CIT_MISC io is merely a context in which pages are owned and locks are enqueued, whereas other io types, in addition to being a context, are also state machines.

6.2. State machine:

The idea behind cl_io state machine is that initial `work' that has to be done (e.g., writing a 3MB user buffer into a given file) is done as a sequence of `iterations', and an iteration is executed as a following idiomatic sequence of steps:

- prepare: determine what work is done at this iteration;
- lock: enqueue and acquire all locks necessary to perform this iteration;
- start: either perform iteration work synchronously, or post it asynchronously, or both;
- end: wait for the completion of asynchronous work;
- unlock: release locks, acquired at the "lock" step;
- finalise: finalise iteration state.

cl_io is a layered entity and each step above is performed by invoking corresponding cl_io_operations method on every layer. As will be explained below, this is especially important in the `prepare' step, as it allows layers to cooperate in determining the scope of the current iteration.

For CIT_READ or CIT_WRITE io, typical scenario is splitting original user buffer into chunks that map completely inside of a single stripe in the target file, and processing each chunk as a separate iteration. In this case, it is lov layer that (in lov_io_rw_iter_init() function) determines the extent of the current iteration.

Once iteration is prepared, "lock" step acquires all necessary DLM locks to cover a region of a file that is affected by the current iteration, and "start" step does the actual processing, which for write means placing pages from the user buffer into the cache, and for read means fetching pages from the server, including read-ahead pages (see `immediate transfer' below).

Truncate and page fault are executed in one iteration (currently that is, it's easy to change truncate implementation to, for instance, truncate each stripe in a separate iteration, should the need arise).

6.3. Parallel io:

....

One important planned generalisation of this model is an out of order execution of the iterations.

Motivating example for this is a write of a large user level buffer, overlapping with multiple stripes. Typically, busy Lustre client has its per-osc caches for the dirty pages nearly full, which means that the write often has to block, waiting for the cache to drain. Instead of blocking whole IO operation, CIT_WRITE might switch to the next stripe and try to do IO there. Without such a `non-blocking' io, a slow OST or an unfair network degrades the performance of the whole cluster.

Another example is a legacy single-threaded application running on a multi-core client machine, where IO throughput is limited by the single thread copying data between the user buffer to the kernel pages. Multiple concurrent IO iterations that can be scheduled independently on the available processors eliminate this bottleneck by copying the data in parallel.

Obviously, parallel io is not compatible with the usual `sequential io' semantics. For example, POSIX read and write have very simple failure model, where some initial (possibly empty) segment of a user buffer is processed successfully, and none of the remaining bytes were read and written. Parallel io can fail in much more intricate ways.

For now, only sequential iterations are supported.

6.4. Data-flow: from stack to io slice:

....

Parallel IO design, outlined above, implies that an ongoing IO can be preempted by other IO and later resumed, all potentially in the same thread. This means that IO state cannot be kept on a stack, as it is customarily done in the UNIX file system drivers. Instead, layered cl_io is used to store information about current iteration, and progress within this iteration. Coincidentally (almost) this is similar to the way IO requests are used by the Windows driver stack.

struct cl_io contains a set of common fields, shared by all layers, describing IO:

- IO type;

- a file (struct cl_object) against which this IO is executed;

- a position in a file, where read and write are going to, and a count of bytes remaining to be processed (for CIT_READ and CIT_WRITE);

- a size to which file is being truncated or expanded (for CIT_TRUNC)

- a list of locks, acquired for this IO;

etc.

Each layer keeps IO state in its `io slice', described below, with all slices chained to the list hanging off struct cl_io:

- vvp_io, ccc_io: these two slices are used by the top-most layer of Linux kernel client. ccc_io is a state common between kernel client and liblustre, and vvp_io is a state private to the kernel client.

  Most important state in ccc_io is an array of struct iovec, describing user space buffers from or to which IO is taking place. Note that other layers in the IO stack have no idea that data actually come from the `user space'.

  vvp_io contains kernel specific fields, such as VM information describing a page fault, or sendfile target.

- lov_io: IO state private for LOV layer is kept here. Most important IO state at the LOV layer is an array of sub-io's. Each sub-io is a normal struct cl_io, representing a part of IO process for a given iteration.

  With current sequential iterations, only one sub-io is active at a time.

- osc_io: this slice, storing IO state private to the osc layer exists within each sub-io created by LOV.

7. Transfer

7.1. Immediate vs. opportunistic:

There are two possible modes of transfer initiation on the client:

- immediate transfer: this is started when a high level io wants a page or a collection of pages to be transferred right away. Examples: read-ahead, synchronous read in the case of non-page aligned write, page write-out as a part of extent lock cancellation, page write-out as a part of memory cleansing. Immediate transfer can be both cl_req_type::CRT_READ and cl_req_type::CRT_WRITE;

- opportunistic transfer (cl_req_type::CRT_WRITE only), that happens when io wants to transfer a page to the server some time later, when it can be done efficiently. Example: pages dirtied by the write(2) path. Pages submitted for an opportunistic transfer are kept in a "staged area".

In any case, transfer takes place in the form of a cl_req, which is a representation for a network RPC.

Pages queued for an opportunistic transfer are placed into a staging area (represented as a set of per-object and per-device queues at the osc layer) until it is decided that efficient RPC can be composed of them. This decision is made by "a req-formation engine", currently implemented as a part of osc layer. Req-formation depends on many factors: the size of the resulting RPC, RPC alignment, whether or not multi-object RPCs are supported by the server, max-rpc-in-flight limitations, size of the staging area, etc. CLIO uses unmodified RPC formation logic from osc, so it is not discussed here.

For the immediate transfer io submits a cl_page_list, that req-formation engine slices into cl_req's, possibly adding cached pages to some of the resulting req's.

Whenever a page from cl_page_list is added to a newly constructed req, its cl_page_operations::cpo_prep() layer methods are called. At that moment, page state is atomically changed from cl_page_state::CPS_OWNED to cl_page_state::CPS_PAGEOUT or cl_page_state::CPS_PAGEIN, cl_page::cp_owner is zeroed, and cl_page::cp_req is set to the req. cl_page_operations::cpo_prep() method at the particular layer might return -EALREADY to indicate that it does not need to submit this page at all. This is possible, for example, if page, submitted for read, became up-to-date in the meantime; and for write, the page don't have dirty bit set. \see cl_io_submit_rw()

Whenever a staged page is added to a newly constructed req, its cl_page_operations::cpo_make_ready() layer methods are called. At that moment, page state is atomically changed from cl_page_state::CPS_CACHED to cl_page_state::CPS_PAGEOUT, and cl_page::cp_req is set to req. cl_page_operations::cpo_make_ready() method at the particular layer might return -EAGAIN to indicate that this page is not eligible for the transfer right now.

RPC engine guarantees that once ->cpo_prep() or ->cpo_make_ready() method has been called, page completion routine (->cpo_completion() layer method) will be eventually called (either as a result of successful page transfer completion, or due to timeout).

To sum, there are two main entry points into transfer sub-system:

- cl_io_submit_rw(): submits a list of pages for immediate transfer;

- cl_page_cache_add(): places a page into staging area for future opportunistic transfer.

7.2. Page lists:

....

To submit a group of pages for immediate transfer struct cl_2queue is used. It contains two page lists: qin (input queue) and qout (output queue). Pages are linked into these queues by cl_page::cp_batch list heads. Qin is populated with the pages to be submitted to the transfer, and pages that were actually submitted are placed onto qout. Not all pages from qin might end up on qout due to

- ->cpo_prep() methods deciding that page should not be transferred, or

- unrecoverable submission error.

Pages not moved to qout remain on qin. It is up to the transfer submitter to decide when to remove pages from qin and qout. Remaining pages on qin are usually removed from this list right after (partially unsuccessful) transfer submission. Pages are usually left on qout until transfer completion. This way caller can determine when all pages from the list were transferred.

Association between a page and a immediate transfer queue is protected by cl_page::cl_mutex: this mutex is acquired when a cl_page is added in a cl_page_list and released when a page is removed from a list.

When an RPC is formed, all its constituent pages are linked together through cl_page::cp_flight list hanging off cl_req::crq_pages. Pages are removed from this list just before transfer completion method is invoked. No special lock protects this list, as pages in transfer are under VM lock.

7.3. States: prepare, completion:

Transfer (cl_req) state machine is trivial, and is not explicitly coded: newly created transfer is in "prepare" state while pages are collected. When all pages are gathered, transfer enters "in-flight" state where it remains until it reaches "completion" state where page completion handlers are invoked.

Per-layer ->cro_prep() transfer method is called when transfer preparation is completed and transfer is about to enter in-flight state. Similarly, per-layer ->cro_completion() method is called when transfer completes before per-page completion methods are called.

Additionally, before moving a transfer out of prepare state RPC engine calls cl_req_attr_set() function invoking ->cro_attr_set() methods on every layer to fill in RPC header that server uses to determine where to get or put data. This replaces old ->ap_{update,fill}_obdo() methods.

Further, cl_req's are not reference counted and access to the is not synchronised, because they are accessed only by the RPC engine in osc that fully controls RPC life-time, and uses osc internal lock (client_obd::cl_loi_list_lock spin-lock) for serialisation.

7.4. Page completion handlers, synchronous transfer:

When a transfer completes, cl_req completion methods are called on every layer. Then, for every transfer page, per-layer page completion methods ->cpo_completion() are invoked. The page is still under VM lock at this moment. Completion methods are called bottom-to-top and it is responsibility of the last of them (i.e., the completion method of the top-most layer---vvp) to release the VM lock.

Both immediate and opportunistic transfers are asynchronous in the sense that control can return to the called before the transfer completion. CLIO doesn't provide synchronous transfer interface at all and it is up to particular caller to implement it if necessary. The simplest way to wait for the transfer completion is wait on a page VM lock. This approach is used implicitly by the Linux kernel. There is a case though, where one wants to do transfer completely synchronous without releasing page VM lock: when ->prepare_write() method determines that a write goes from a non page-aligned buffer into a not up to date page, a portion of a page has to be fetched from the server. VM page lock cannot be used to synchronise transfer completion in this case, because it is used to mark the page as owned by IO. To handle this, vvp attaches struct cl_sync_io to struct vvp_page. cl_sync_io contains a number of pages still in IO and a synchronisation primitive (struct completion) signalled when transfer of the last page completes. vvp page completion handler (vvp_page_completion_common()) checks for attached cl_sync_io and if it is here, decreases number of in-flight pages there and signals completion when that number drops to 0. Similar mechanism is used for direct-io.

8. lu_env

8.1. Motivation, server usage:

lu_env and related data-types (struct lu_context and struct lu_context_key) together implement a memory pre-allocation interface that Lustre uses to decrease stack consumption without resorting to fully dynamic allocation.

Stack space is severely limited in the Linux kernel. Lustre traditionally allocated a lot of automatic variables, resulting in spurious stack overflows that are hard to trigger (they usually need a certain combination of driver calls and interrupts to happen, making them extremely difficult to reproduce) and debug (as stack overflow can easily result in corruption of thread-related data-structures in the kernel memory, confusing the debugger).

The simplest way to handle this is to replace automatic variables with calls to generic memory allocator, but

- generic allocator has scalability problems, and

- additional code to free allocated memory is needed.

lu_env interface was originally introduced for CMD3 mdt stack and matches server-side threading model very well. Roughly speaking, lu_context represents a context in which computation is executed and lu_context_key is a description of per-context data. In the simplest case lu_context corresponds to a server thread; then lu_context_key is effectively a thread-local storage (TLS), see user level pthreads interface pthread_key_create() for a similar idea.

More formally, lu_context_key defines a constructor-destructor pair and a tags bit-mask. When lu_context is initialised (with a given tag bit-mask), a global array of all registered lu_context_keys is scanned, constructors for all keys with matching tags are invoked and their return values are stored in lu_context.

Once lu_context has been initialised, a value of any key allocated for this context can be retrieved very efficiently by indexing in the per-context array. lu_context_key_get() function is used for this.

When context is finalised, destructors are called for all keys allocated in this context.

The typical server usage is to have a lu_context for every server thread, initialised when the thread is started. To reduce stack consumption by the code running in this thread, a lu_context_key is registered that allocates in its constructor a struct containing as fields values otherwise allocated on the stack. See {mdt,osd,cmm,mdd}_thread_info for examples. Instead of doing

```
int function(args) {
         /* structure "bar" in module "foo" */
         struct foo_bar bar;
         ...
```

code does roughly

```
struct foo_thread_info {
         struct foo_bar fti_bar;
         ...
};

int function(const struct lu_env *env, args) {
         struct foo_bar *bar;
         ...

         bar = &lu_context_key_get(&env->le_ctx, &foo_thread_key)->fti_
```

etc.

struct lu_env contains 2 contexts:

- le_ctx: this context is embedded in lu_env. By convention, this context is used _only_ to avoid allocations on the stack, and it should never be used to pass parameters between functions or layers. The reason for this restriction is that using contexts for implicit state sharing leads to a code that is difficult to understand and modify.

- le_ses: this is a pointer to a context shared by all threads handling given RPC. Context itself is embedded into struct ptlrpc_request. Currently a request is always processed by a single thread, but this might change in the future with the design where a small pool of threads processes RPCs non-blockingly.

  Additionally, state kept in env->le_ses context is shared by multiple layers. For example, remote user credentials are stored there.

8.2. Client usage:

....

On a client there too is a lu_env, associated with every thread executing Lustre code. Again, it contains &env->le_ctx context used to reduce stack consumption. env->le_ses is used to share state between all threads handling given IO. Again, currently an IO is processed by a single thread. env->le_ses is used to efficiently allocate cl_io slices ({vvp,lov,osc}_io).

There are three important differences with lu_env usage on the server:

- while on the server there is a fixed pool of threads, any client thread can execute Lustre code. This makes in impractical to pre-allocate and pre-initialise lu_context for every thread. Instead, contexts are constructed on demand and after use returned into a global cache that amortises creation cost;

- client call-chains cross Lustre-VFS and Lustre-VM boundaries all the time. This means that just passing lu_env as a first parameter to every Lustre function and method is not enough. To work around this problem, a pointer to lu_env is stored in a field in the kernel data-structure associated with the current thread (task_struct::journal_info), from where it is recovered when Lustre code is re-entered from VFS or VM;

- sometimes, client code is re-entered in a fashion that precludes re-use of the higher level lu _env. For example, when a read or write incurs a page fault in the user space buffer memory mapped from a Lustre file, page fault handling is a separate IO, independent of the already ongoing system call. Lustre page fault handler allocates new lu_env (by calling lu_env_get_nested()) in which nested IO is going on. Similar situation occurs when client DLM lock LRU shrinking code is invoked in the context of a system call.

8.3. Sub-environments:

....

As described above, lu_env (specifically, lu_env->le_ses) is used on a client to allocate per-IO state, including foo_io data on every layer. This leads to a complication at the lov layer, that maintains multiple sub-ios. As layers below lov allocate their io slices in lu_env->le_ses, lov has to allocate an lu_env for every sub-io and to carefully juggle them when invoking lower layer methods. The case of a single IO is optimised by re-using top-environment.

9. Use cases

9.1. inode creation:

Lookup ends up calling ll_update_inode(), to setup a new inode with a given meta-data descriptor (obtained from the meta-data path). cl_inode_init() calls cl_object_find() eventually calling lu_object_find_try() that either finds a cl_object in the cache or allocates new one, calling lu_device_operations::ldo_object_{alloc,init}() methods on every layer top to bottom. Every layer allocates its private data structure ({vvp,lov}_object) and links it into an object header (cl_object_header) by calling lu_object_add(). At the vvp layer, vvp_object contains a pointer to the inode. lov layer allocates lov_object, containing an array of pointers to sub-objects, that are found in the cache or allocated by calling cl_object_find (recursively). These sub-objects have lovsub and osc layer data.

Top-object and its sub-objects are inserted into global fid-based hash table and global LRU list.

9.2. First IO to a file:

After object is instantiated as described in the previous use case, first IO call against this object has to create DLM locks. Following operations re-use cached locks, see below.

Read call starts at --depending on the kernel version-- ll_file_readv() that eventually calls ll_file_io_generic(). This function calls cl_io_init() to initialises IO context, that calls cl_object_operations::coo_io_init() method on every layer. As in the case of object instantiation, these methods allocate layer-private IO state ({vvp,lov}_io), and add it to the list hanging off IO context header cl_io, by calling cl_io_add(). At the vvp layer, vvp_io_init() handles special cases (like count == 0), updates statistic counters and in case of write takes per-inode semaphore to avoid possible deadlock.

At the lov layer, lov_io_init_raid0() allocates struct lov_io, and stores in it original IO parameters (starting offset and byte count). This is needed because lov is going to modify these parameters. Sub-ios are not allocated at this point---they are instantiated later lazily.

Once top-io has been initialised, ll_file_io_generic() enters main IO loop cl_io_loop() that drives IO iterations, going through

```
    - cl_io_iter_init()  calling cl_io_operations::cio_iter_init() top-to-bottom
    - cl_io_lock()       calling cl_io_operations::cio_lock()      top-to-bottom
    - cl_io_start()      calling cl_io_operations::cio_start()     top-to-bottom
    - cl_io_end()        calling cl_io_operations::cio_end()       bottom-to-top
    - cl_io_unlock()     calling cl_io_operations::cio_unlock()    bottom-to-top
    - cl_io_iter_fini()  calling cl_io_operations::cio_iter_fini() bottom-to-top
    - cl_io_rw_advance() calling cl_io_operations::cio_advance()   bottom-to-top
```

repeatedly until cl_io::ci_continue remains 0 after an iteration. These "io iterations" drag IO context through consecutive states, see enum cl_io_state. ->cio_iter_init() decides at each layer what part of the remaining IO is to be done during current iteration. Currently, lov_io_rw_iter_init() is the only non-trivial implementation of this method. It

    - except for the cases of truncate and O_APPEND write, it shrinks IO extent recorded in the top-io (starting offset and bytes count) so that this extent is fully contained within a single stripe. This avoids "cascading evictions";

- it allocates sub-ios for all stripes intersecting with the resulting IO range (which, in case of non-append write or read means creating single sub-io) by calling cl_io_init() that (as above) creates a cl_io context with lovsub_io and osc_io layers. Initialised cl_io is primed from the top-io (lov_io_sub_inherit()) and cl_io_iter_init() is called against it;

- finally all sub-ios for the current iteration are linked together into a lov_io::lis_active list.

Now we have a top-io and its sub-io in CIS_IT_STARTED state. cl_io_lock() collects locks on all layers without actually enqueuing them: vvp_io_rw_lock() requests a lock on IO extent (possibly shrunk by lov, see above) and optionally on extents of Lustre files that happen to be memory mapped onto the user level buffer used for this IO. In the future layers like SNS might request additional locks, e.g., to protect parity blocks.

Locks requested by ->cio_lock() methods are added to the cl_lockset embedded into top cl_io. Lockset contains 3 lock queues: "todo", "current" and "done". Locks are initially placed in the todo queue. Once locks from all layers have been collected, they are sorted to avoid deadlocks (cl_io_locks_sort()) and them enqueued by cl_lockset_lock(). The latter can enqueue multiple locks concurrently if enqueuing mode guarantees this is safe (e.g., lock is a try-lock). Locks being enqueued are in "current" queue, from where they are moved into "done" queue when the lock is granted.

At this stage we have top- and sub-io in CIS_LOCKED state with all needed locks held. cl_io_start() moved cl_io into CIS_IO_GOING mode and calls ->cio_start() method. On vvp layer this method invokes some version of generic_file_{read,write}() function.

In the case of read, generic_file_read() calls for every non-uptodate page a_ops->readpage() method that eventually (after obtaining cl_page corresponding to the VM page supplied to it) calls cl_io_read_page() that calls cl_io_operations::cio_read_page().

vvp_io_read_page() populates a queue by a target page and pages from read-ahead window. Resulting queue is then submitted to the immediate transfer by calling cl_io_submit_rw(), ending up calling osc_io_submit_page() for every not-up-to-date page in the queue.

->readpage() returns at this point, and VM waits on a VM page lock, released by the transfer completion handler before copying page date to the user buffer.

In the case of write, generic_file_write() calls a_ops->prepare_write() and a_ops->commit_write() address space methods that end up calling cl_io_prepare_write() and cl_io_commit_write() respectively. These functions follow normal Linux protocol for write, including possible synchronous read of a non-overwritten part of a page (vvp_page_sync_io() call in vvp_io_prepare_partial()) and in the normal case end up by placing the dirtied page into staging area (cl_page_cache_add() call in vvp_io_commit_write()). If staging area is full already, cl_page_cache_add() fails with -EDQUOT and page is transferred immediately by calling vvp_page_sync_io().

9.3. Cached IO:
....

Subsequent IO calls will, most likely, find suitable locks already cached on the client. This happens because server tries to grant as large lock as possible, to reduce future enqueue RPC traffic for a given file from a given client. Cached locks are kept (in no particular order) on a cl_object_header::coh_locks list. When, in cl_io_lock() step, a layer requests a lock, this list is scanned for matching lock. If found lock is in HELD or CACHED state it can be re-used immediately by simply calling cl_lock_use() method, that eventually calls ldlm_lock_addref_try() protecting underlying DLM lock from a concurrent cancellation while IO is going on. If a lock in other (NEW, QUEUING or ENQUEUED) state is found, it is enqueued as usual.

9.4. Lock-less and no-cache IO:

....

IO context has a "locking mode" selected from MAYBE, NEVER or MANDATORY set (enum cl_io_lock_dmd), that specifies what degree of distributed cache coherency is assumed by this IO. MANDATORY mode requires all caches accessed by this IO to be protected by distributed locks. In NEVER mode no distributed coherency is needed at the expense of not caching the data. This mode is required for the cases where client can not or will not participate in the cache coherency protocol (e.g., a liblustre client that cannot respond to the lock blocking call-backs while in compute phase). In MAYBE mode some of the caches involved in this IO are used and are globally coherent, and some other caches are bypassed.

O_APPEND writes and truncates are always executed in MANDATORY mode. All other calls are executed in NEVER mode by liblustre (see below) and in MAYBE mode by a normal Linux client.

In MAYBE mode every osc decides individually whether to use DLM. OST might return -EUSERS result to enqueue rpc indicating that the stripe in question is contended and client should switch to the lockless IO mode. If this happens, osc, instead of using ldlm_lock, creates special "lockless osc lock" that is not backed up by a DLM lock. This lock conflicts with any other lock in its range and self-cancels when its last user is removed. As a result, when IO proceeds to the stripe that is in lockless mode, all conflicting extent locks are cancelled, purging the cache, and when IO against this stripe ends, the lock is cancelled, sending dirty pages (just placed in the cache by IO) back to the server and invalidating the cache again. "Lockless locks" allow lockless and no-cache IO mode to be implemented by the same code paths as cached IO.

* * * END * * *