Client IO Layering Cleanup

Nikita Danilov <nikita.danilov@sun.com>

Started: 2008.01.25

1 Introduction

This document describes reorganization of the Lustre client IO path (*i.e.*, data access: read, write, truncate, *etc.*, as opposed to the meta-data path), targeting more regular and extensible structure of the client stack. The companion to this paper is [1].

2 Scope

2.1 What is in the scope of this document.

- overall description of the new layering;
- common interfaces between layers;
- assumptions (concurrency, liveness, ordering, *etc.*) that can be made by the layer code;
- certain amount of rationale.

2.2 What is not in the scope of this document.

- structure of the pre-existing layering (necessary to understand reorganization);
- specific planned extensions, like SNS or "parallel IO", that partially motivate re-structuring;
- recovery;
- meta-data operations.

3 Requirements

New interfaces were designed with the following goals in mind:

- MD server stack has already been converted to the different layering model, and client code has to capitalize on that. See [0] for details. In particular, following parts of the server side layering support are immediately re-usable on the client:
  - fid-based object identification;
  - object caching and life cycle management;
  - stack consumption;
  - well-structured interfaces.

- layering support for the client-specific entities, like pages, extent locks, IO requests, *etc.*
- clear and precise semantics for interface entry points.
- no meta-data changes.
- no 2.4 kernels support is necessary.
- portable code.
- patchless client.
- support for certain planned extensions of the client functionality. Specifically:
  - server network striping;
  - parallel IO client design;
  - peer-to-peer data caching.

4 Glossary

io a higher-level I/O operation, like read(2) or write(2) system call, or compound I/O activity initiated by client internally, like write-out of pages from under extent lock being cancelled. This is to be distinguished from transfer.

transfer a lower-layer I/O operation, usually network RPC.

generic code a code in the hosting environment that is not file system specific (*e.g.*, VFS, VM, MM).

cl-code a client code, mostly in cl_{object,lock,page,io}.c, that is not layer-specific.

slice, layer a portion of a compound entity corresponding to the particular device in the device stack. Entities like objects, pages, locks are represented as sequences of slices. Each slice, usually, has a vector (or vectors) of operations and maintains certain layer-specific state. cl-code delegates parts of functionality to the layers, by calling methods in these operation vectors (in top-to-bottom or bottom-to-top order, depending on the semantics) across all layers.

fan out an important difference between client and server side object models, which is that on a client certain entities can be composed of sub-entities of the similar nature. *E.g.*, files are backed up by stripes, extent locks on the files are implemented as a sets of sub-locks on the stripes, *etc.* We say, that corresponding top-entity *fans out* into sub-entities. Objects, locks, and high-level IO do fan out. Pages and transfers—do not.

5 Functional Specification

Due to the large amount of new interfaces and data-types introduced by this design, Functional Specification section is split into 3 parts:

- Functional Specification proper: an overall description of new data-types and their layering is given.
- State Specification: a description of state machines.
- Operation Vectors: a description of layer methods.

In addition to the entities provided by the lu_object infrastructure, (site, device type, device, and object), following new data-types are implemented on the client.

5.1 cl_object

`cl_object` represents a regular file system object, both a file and a stripe. `cl_object` is based on `lu_object`: it is identified by a fid, layered, cached, hashed, and lrued. Important distinction with the server side, where md_object and dt_object are used, is that "fans out" at the lov/sns 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.

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:

- clu
- 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. 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. cl_object is a sub-class of lu_object (in the same sense, as server-side md_object and dt_object are sub-classes of lu_object), see [0] for more details on the latter.

5.2 cl_page

5.2.1 Layered client page.

`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`. `cl_page` doesn't fan out, but as sub-objects of the top-level file object are first class `cl_object`'s, they have their own radix trees of pages and hence page is implemented as a sequence of struct `cl_page`'s, linked into double-linked list through `->cp_parent` and `->cl_child` pointers, each residing in the corresponding radix tree at the corresponding logical offset.

5.2.2 Association with VM page

`cl_page` is associated with VM page of the hosting environment (struct page in 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.

5.2.3 Page 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 synchronization 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 synchronized 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().

5.2.4 Life cycle

cl_page is reference counted (->cp_ref). When reference counter drops to 0, the page is returned to the cache, unless it is in CPS_FREEING state, in which case it is immediately destroyed. The general logic guaranteeing the absence of "existential races" for pages is the following:

- there are fixed known ways for a thread to obtain a new reference to a page:
  - by doing a lookup in the cl_object radix tree, protected by the spin-lock;
  - by starting from VM-locked cfs_page_t and following some hosting environment method (e.g., following ->private pointer in the case of Linux kernel), see cl_vmpage_page();
- when the page enters CPS_FREEING state, all these ways are severed with the proper synchronization (cl_page_invalidate());
- entry into CPS_FREEING is serialized by the VM page lock;
- no new references to the page in CPS_FREEING state are allowed (checked in cl_page_get()).

Together this guarantees that when last reference to a CPS_FREEING page is released, it is safe to destroy the page, as neither references to it can be acquired at that point, nor ones exist. cl_page is a state machine. States are enumerated in enum cl_page_state. Possible state transitions are enumerated in cl_page_state_set(). State transition process (i.e., actual changing of ->cp_state field) is protected by the lock on the underlying VM page.

5.2.5 Linux Kernel implementation.

Binding between `cl_page` and `cfs_page_t` (which is a typedef for `struct page`) is implemented in the clu layer. `cl_page` is attached to the `->private` pointer of the `struct page`, together with the setting of `PG_private` bit in `page->flags`, and acquiring additional reference on the `struct page` (much like `struct buffer_head`, or any similar file system private data structures).

`PG_locked` lock is used to implement both ownership and transfer synchronization, that is, page is VM-locked in `CPS_{OWNED,PAGE{IN,OUT}}` states. No additional references are acquired for the duration of the transfer.

THIS IS NOT the behavior expected by the Linux kernel, where write-out is "protected" by the special `PG_writeback` bit.

5.3 cl_lock

Extent locking on the client.

5.3.1 Layering

The locking model of the new client code is built around struct cl_lock data-type representing an extent lock on a regular file. cl_lock is a layered object (much like cl_object and cl_page), it consists of a header (struct cl_lock) and a list of layers (struct cl_lock_slice), linked to cl_lock.cll_layers list through cl_lock_slice.cls_linkage.

All locks for a given object are linked into cl_object_header.coh_locks list (protected by cl_object_header.coh_lock_guard spin-lock) through cl_lock.cll_linkage. Currently this list is not sorted in any way. We can sort it in starting lock offset, or use altogether different data structure like a tree.

Typical cl_lock consists of the two layers:

- clu_lock (clu specific data), and
- lov_lock (lov specific data).

lov_lock contains an array of sub-locks. Each of these sub-locks is a normal cl_lock: it has a header (struct cl_lock) and a list of layers:

- lovsub_lock, and
- osc_lock

Each sub-lock is associated with a cl_object (representing stripe sub-object or the file to which top-level cl_lock is associated to), and is linked into that cl_object.coh_locks. In this respect cl_lock is similar to cl_object (that at lov layer also fans out into multiple sub-objects), and is different from cl_page, that doesn't fan out (there is usually exactly one osc_page for every clu_page). We shall call clu-lov portion of the lock a "top-lock" and its lovsub-osc portion a "sub-lock".

5.3.2 Life cycle

cl_lock is reference counted. When reference counter drops to 0, lock is placed in the cache, except when CLF_NOREFS bit is set in cl_lock.cll_flags bit-mask. When this bit is set, lock is destroyed when last reference is released. Top-lock keeps additional reference to every sub-lock, and releases this reference when it is destroyed. That is to say, top-locks are a master cache for the sub-locks, and sub-lock cannot normally exist without corresponding top-lock (other way around is possible if some sub-locks of a top-lock were canceled).

5.3.3 Interface and usage

struct cl_lock_operations provide a number of call-backs that are invoked when events of interest occurs. Layers can intercept and handle glimpse, blocking, cancel ASTs and a reception of the reply from the server.

One important difference with the old client locking model is that new client has a representation for the top-lock, whereas in the old code only sub-locks existed as real data structures and file-level locks are represented by "request sets" that are created and destroyed on each and every lock creation.

Top-locks are cached, and can be found in the cache by the system calls. It is possible that top-lock is in cache, but some of its sub-locks were canceled and destroyed. In that case top-lock has to be enqueued again

before it can be used.

Overall process of the locking during an iteration of IO (see description of cl_io iterations below) is as following:

- once parameters for IO are setup in cl_io by ->cio_prep() calls, ->cio_lock() is called on each layer. Responsibility of this method is to add locks, needed by a given layer into cl_io.ci_lockset. Note that this step only collects locks, without actually enqueuing them.
- once locks for all layers were collected, they are sorted to avoid dead-locks (cl_io_locks_sort()), and enqueued.
- when all locks are acquired, IO is performed;
- locks are released into cache.

5.3.4 Striping

Striping introduces major additional complexity into locking. The fundamental problem is that it is generally unsafe to actively use (*hold*) two locks on the same OST servers at the same time, as this introduces inter-server dependency and can lead to cascading evictions.

Basic solution is to sub-divide large read/write IOs into smaller pieces so that no multi-stripe locks are taken (note that this design abandons POSIX read/write semantics). Such pieces ideally can be executed concurrently. At the same time, certain types of IO cannot be sub-divived, without sacrificing correctness. This includes:

- O_APPEND write, where [0, EOF] lock has to be taken, to guarantee atomicity;
- ftruncate(fd, offset), where [offset, EOF] lock has to be taken.

Also, in the case of read(fd, buf, count) or write(fd, buf, count), where buf is a part of memory mapped Lustre file, a lock or locks protecting buf has to be held together with the usual lock on [offset, offset + count].

As multi-stripe locks have to be allowed, it makes sense to cache them, so that, for example, a sequence of O_APPEND writes can proceed quickly without going down to the individual stripes to do lock matching. On the other hand, multi-stripe locks shouldn't be used by normal read/write calls. To achieve this, every layer can implement ->clo_fits_into() method, that is called by lock matching code (cl_lock_lookup()), and that can be used to selectively disable matching of certain locks for certain IOs. For exmaple, lov layer implements lov_lock_fits_into() that allow multi-stripe locks to be matched only for truncates and O_APPEND writes.

5.3.5 Interaction with DLM

In the expected setup, cl_lock is ultimately backed up by a collection of DLM locks (struct ldlm_lock). Association between cl_lock and DLM lock is implemented in osc layer, that also matches DLM events (ASTs, cancellation, *etc.*) into cl_lock_operation calls.

5.4 cl_io

cl_io represents a high level I/O activity like read(2)/write(2)/truncate(2) system call, or cancellation of an extent lock. There is a small predefined number of possible io types, enumerated in enum cl_io_type.

5.4.1 State machine

cl_io is a state machine, that can be advanced concurrently by the multiple threads. It is up to these threads to control the concurrency and, specifically, to detect when io is done, and its state can be safely released.

For read/write io overall execution plan is as following:

1. initialize io state through all layers;
2. loop: prepare chunk of work to do for this iteration;
3. call all layers to collect locks they need to process current chunk;
4. sort all locks to avoid dead-locks, and acquire them;
5. process the chunk: call per-page methods (->cio_read_page() for read, ->cio_prepare_write(), cio_commit_write() for write);
6. release locks;
7. repeat loop, from step 2.

5.4.2 Layering

cl_io is a layered object, much like cl_{object,page,lock}, but with one important distinction. We want to minimize number of calls to the allocator in the fast path, e.g., in the case of read(2) when everything is cached: client already owns the lock over region being read, and data are cached due to read-ahead. To avoid allocation of cl_io layers in such situations, per-layer io state is stored in the session, associated with the io, see struct {clu,lov,osc}_session for example. Sessions allocation is amortized by using free-lists, see cl_env_get().

To implement the "parallel IO mode", lov layer creates sub-io's (lazily to address allocation efficiency issues mentioned above), and returns with the special error condition from per-page method when current sub-io has to block. This causes io loop to be repeated, and lov switches to the next sub-io in its ->cio_prep() implementation.

5.5 cl_req

5.5.1 Transfer model

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

immediate transfer: this is started when 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 CRT_READ and CRT_WRITE;

opportunistic transfer (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.

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 cached 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, whether or not multi-object RPCs are supported by the server, max-rpc-in-flight limitations, size of the dirty cache, etc.

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 ->cpo_prep() layer methods are called. At that moment, page state is atomically changed from CPS_OWNED to CPS_PAGE{OUT,IN}, ->cp_owner is zeroed, and ->cp_req is set to the req. ->cpo_prep() method at the particular layer might return -ENOENT to indicate that it doesn't want page to be submitted immediately. This is possible, for example, if page, submitted for read, became up-to-date in the meantime.

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

5.5.2 Future

Plan is to divide transfers into "priority bands" (indicated when submitting cl_page_list, and queuing a page for the opportunistic transfer) and allow gluing of cached pages to immediate transfers only within single band. This would make high priority transfers (like lock cancellation or memory pressure induced write-out) really high priority.

6 State Specification

6.1 State machines and caching

All data-types described in the Functional Specification section are implemented as state machines, but with different concurrency models. All of them, except for cl_io and cl_req are also reference counted, and are kept in the certain indices. General rule is that a state machine for a reference counted data-type has some dedicated "terminal" state (CPS_FREEING, CLS_FREEING, LU_OBJECT_HEARD_BANSHEE), that cannot be left once reached. No new references can be obtained to the object in that state, and object in the terminal state is destroyed as soon as a last reference is released. If object is not yet in the terminal state when last reference is released, it is placed into a cache.

6.2 cl_object

State machine for the cl_object is the same as for its underlying lu_object. See [0].

6.3 cl_page

The page state machine is rather crude, as it doesn't recognize finer page states like "dirty" or "up to date". This is because such states are not always well defined for the whole stack (see, for example, the implementation of the read-ahead, that hides page up-to-dateness to track cache hits accurately). Such sub-states are maintained by the layers that are interested in them. Following states are present:

CPS_CACHED Page is in the cache, un-owned. Page leaves cached state in the following cases:

- [->CPS_OWNED] io comes across the page and owns it;
- [->CPS_PAGEOUT] page is dirty, the req-formation engine decides that it wants to include this page into an cl_req being constructed, and yanks it from the cache;
- [->CPS_FREEING] VM callback is executed to evict the page form the memory;

State invariants: ->cp_owner == NULL && ->cp_req == NULL

CPS_OWNED Page is exclusively owned by some cl_io. Page may end up in this state as a result of

- io creating new page and immediately owning it;
- [<-CPS_CACHED] io finding existing cached page and owning it;
- [<-CPS_OWNED] io finding existing owned page and waiting for owner to release the page;

Page leaves owned state in the following cases:

- [->CPS_CACHED] io decides to leave the page in the cache,doing nothing;
- [->CPS_PAGEIN] io starts read transfer for this page;
- [->CPS_PAGEOUT] io starts immediate write transfer for this page;
- [->CPS_FREEING] io decides to destroy this page (e.g., as part of truncate or extent lock cancellation).

State invariants: ->cp_owner != NULL && ->cp_req == NULL

CPS_PAGEOUT Page is being written out, as a part of a transfer. This state is entered when req-formation logic decided that it wants this page to be sent through the wire *now*. Specifically, it means that once this state is achieved, transfer completion handler (with either success or failure indication) is guaranteed to be executed against this page independently of any locks and any scheduling decisions made by the hosting environment (that effectively means that the page is never put into CPS_PAGEOUT state "in advance". This property is mentioned, because it is important when reasoning about possible dead-locks in the system). The page can enter this state as a result of

- [<-CPS_OWNED] an io requesting an immediate write-out of this page, or
- [<-CPS_CACHED] req-forming engine deciding that it has enough dirty pages cached to issue a "good" transfer.

The page leaves CPS_PAGEOUT state when the transfer is completed—it is moved into CPS_CACHED state. Underlying VM page is locked for the duration of transfer.

State invariants: ->cp_owner == NULL && ->cp_req != NULL

CPS_PAGEIN Page is being read in, as a part of a transfer. This is quite similar to the CPS_PAGEOUT state, except that read-in is always "immediate"—there is no such thing a sudden construction of read cl_req from cached, presumably not up to date, pages. Underlying VM page is locked for the duration of transfer.

State invariants: ->cp_owner == NULL && ->cp_req != NULL

CPS_FREEING Page is being destroyed. This state is entered when client decides that page has to be deleted from its host object, as, e.g., a part of truncate. Once this state is reached, there is no way to escape it.

State invariants: ->cp_owner == NULL && ->cp_req == NULL

6.4 cl_lock

6.4.1 Rationale

Also, cl_lock is a state machine. This requires some clarification. One of the goals of client IO re-write was 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 -EAGAIN. 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 normal blocking fashion, and it the same time it is possible to switch to the non-blocking locking (simply by returning -EAGAIN from cl_lock.c functions). For a description of state machine states and transitions see enum cl_lock_state.

6.4.2 Concurrency

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

- mutex is taken, and ->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 (for example, if we are trying to obtain a lock, and it's already cached), state is changed, and mutex is released.
- if the transition requires blocking, _try() function returns 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 mutexes 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_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).

6.4.3 States

CLS_NEW Lock that wasn't yet enqueued.

CLS_QUEUING Enqueue is in progress, blocking for some intermediate interaction with the other side.

CLS_ENQUEUED Lock is fully enqueued, waiting for server to reply when it is granted.

CLS_HELD Lock granted, actively used by some IO.

CLS_UNLOCKING Lock granted, but not used, blocking in the communication.

CLS_CACHED Lock granted, not used.

CLS_FREEING Lock is being destroyed.

These states are for individual cl_lock object. Top-lock and its sub-locks can be in the different states. Another way to say this is that we have nested state-machines.

Separate QUEUING and ENQUEUED states are needed to support non-blocking operation for locks with multiple sub-locks. Imagine lock on a file F, that intersects 3 stripes S0, S1, and S2. To enqueue F client has to send enqueue to S0, wait for its completion, then send enqueue for S1, wait for its completion and at last enqueue lock for S2, and wait for its completion. In that case, top-lock is in QUEUING state while S0, S1 are handled, and is in ENQUEUED state after enqueue to S2 has been sent (note that in this case, sub-locks move from state to state, and top-lock remains in the same state).

7 Logical Specification

This section describes interfaces to various parts of client io layering.

7.1 general

7.1.1 data-types

Operations for each data device in the client stack.

```
struct cl_device_operations {
        /* Initialize cl_req. This method is called top-to-bottom on all
         * devices in the stack to get them a chance to allocate layer-private
         * data, and to attach them to the cl_req by calling
         * cl_req_slice_add(). */
        int (*cdo_req_init)(const struct lu_env *env, struct cl_device *dev,
                            struct cl_req *req);
};
```

Device in the client stack.

```
struct cl_device {
        /* Super-class. */
        struct lu_device                    cd_lu_dev;
        /* Per-layer operation vector. */
        const struct cl_device_operations *cd_ops;
};
```

Stats for a generic cache (similar to inode, lu_object, etc. caches).

```
struct cache_stats {
        const char    *cs_name;
        /* how many entities were created at all */
        atomic_t       cs_created;
        /* how many cache lookups were performed */
        atomic_t       cs_lookup;
        /* how many times cache lookup resulted in a hit */
        atomic_t       cs_hit;
        /* how many entities are in the cache right now */
        atomic_t       cs_total;
        /* how many entities in the cache are actively used (and cannot be
         * evicted) right now */
        atomic_t       cs_busy;
};
```

Client-side site. This represents particular client stack. "Global" variables should (directly or indirectly) be added here to allow multiple clients to co-exist in the single address space.

```
struct cl_site {
        struct lu_site cs_lu;
        /* Statistical counters. Atomics do not scale, something better like
         * per-cpu counters is needed.
         * These are exported as /proc/fs/lustre/llite/.../site
         * When interpreting keep in mind that both sub-locks (and sub-pages)
         * and top-locks (and top-pages) are accounted here. */
        struct cache_stats cs_pages;
        struct cache_stats cs_locks;
        struct cache_stats cs_env;
        atomic_t           cs_pages_state[CPS_NR];
        atomic_t           cs_locks_state[CLS_NR];
};
```

7.1.2 entry points

Initialize client site. Perform common initialization (lu_site_init()), and initialize statistical counters.

```
int  cl_site_init (struct cl_site *s, struct cl_device *top);
```

Finalize client site. Dual to cl_site_init().

```
void cl_site_fini (struct cl_site *s);
```

Finalize device stack by calling lu_stack_fini().

```
void cl_stack_fini(const struct lu_env *env, struct cl_device *cl);
```

Output client site statistical counters into a buffer. Suitable for ll_rd_*()-style functions.

```
int cl_site_stats_print(const struct cl_site *s, char *page, int count);
```

Returns lu_env: if there already is an environment associated with the current thread, it is returned, otherwise, new environment is allocated. Allocations are amortized through the global cache of environments.

\param refcheck pointer to a counter used to detect environment leaks. In the usual case cl_env_get() and cl_env_put() are called in the same lexical scope and pointer to the same integer is passed as \a refcheck. This is used to detect missed cl_env_put().

```
struct lu_env *cl_env_get(int *refcheck);
```

Release an environment. Decrement \a env reference counter. When counter drops to 0, nothing in this thread is using environment and it is returned to the allocation cache, or freed straight away, if cache is large enough.

```
void cl_env_put(struct lu_env *env, int *refcheck);
```

Declares a point of re-entrancy. In Linux kernel environments are attached to the thread through current->journal_info pointer that is used by other sub-systems also. When lustre code is invoked in the situation where current->journal_info is potentially already set, cl_env_reenter() is called to save current->journal_info value, so that current->journal_info field can be used to store pointer to the environment.

```
void *cl_env_reenter(void);
```

Exits re-entrancy. This restores old value of current->journal_info that was saved by cl_env_reenter().

```
void cl_env_reexit(void *cookie);
```

7.2 cl_object

7.2.1 data-structures

"Data attributes" of cl_object. Data attributes can be updated independently for a sub-object, and top-object's attributes are calculated from sub-objects' ones.

```
struct cl_attr {
        /* Object size, in bytes */
        loff_t cat_size;
        /* Known minimal size, in bytes.
         * This is only valid when at least one DLM lock is held. */
        loff_t cat_kms;
        /* Modification time. Measured in seconds since epoch. */
        time_t cat_mtime;
        /* Access time. Measured in seconds since epoch. */
        time_t cat_atime;
        /* Change time. Measured in seconds since epoch. */
        time_t cat_ctime;
        /* Blocks allocated to this cl_object on the server file system.
         * \todo XXX An interface for block size is needed. */
        __u64  cat_blocks;
};
```

Fields in cl_attr that are being set.

```
enum cl_attr_valid {
        CAT_SIZE   = 1 << 0,
        CAT_KMS    = 1 << 1,
        CAT_MTIME  = 1 << 3,
        CAT_ATIME  = 1 << 4,
        CAT_CTIME  = 1 << 5,
        CAT_BLOCKS = 1 << 6
};
```

Sub-class of lu_object with methods common for objects on the client stacks.

```
struct cl_object {
        /* super class */
        struct lu_object                  co_lu;
        /* per-object-layer operations */
        const struct cl_object_operations *co_ops;
        /* io operations */
        const struct cl_io_operations     *co_iop;
};
```

Description of the client object configuration. This is used for the creation of a new client object that is identified by a more state than fid.

```
struct cl_object_conf {
        /* Super-class. */
        struct lu_object_conf     coc_lu;
        union {
                /* Object layout. This is consumed by lov. */
                struct lustre_md *coc_md;
                /* Description of particular stripe location in the cluster.
                 * This is consumed by osc. */
                struct lov_oinfo *coc_oinfo;
        } u;
        /* VFS inode. This is consumed by clu. */
        struct inode             *coc_inode;
};
```

Operations implemented for each cl object layer.

```
struct cl_object_operations {
        /* Initialize page slice for this layer. Called top-to-bottom through
         * every object layer when a new cl_page is instantiated. Layer
         * keeping private per-page data, or requiring its own page operations
         * vector should allocate these data here, and attach then to the page
         * by calling cl_page_slice_add(). \a vmpage is locked (in the VM
         * sense). Optional. */
        int  (*coo_page_init)(const struct lu_env *env, struct cl_object *obj,
                              struct cl_page *page, cfs_page_t *vmpage);
        /* Initialize lock slice for this layer. Called top-to-bottom through
         * every object layer when a new cl_lock is instantiated. Layer
         * keeping private per-lock data, or requiring its own lock operations
         * vector should allocate these data here, and attach then to the lock
         * by calling cl_lock_slice_add(). Mandatory. */
```

```
        int  (*coo_lock_init)(const struct lu_env *env,
                              struct cl_object *obj, struct cl_lock *lock,
                              struct cl_io *io);
        /* Fill portion of \a attr that this layer controls. This method is
         * called top-to-bottom through all object layers.
         * \pre cl_object_header::coh_attr_guard of the top-object is locked.
         * \return   0: to continue
         * \return +ve: to stop iterating through layers (but 0 is returned
         * from enclosing cl_object_attr_get())
         * \return -ve: to signal error */
        int (*coo_attr_get)(const struct lu_env *env, struct cl_object *obj,
                            struct cl_attr *attr);
        /* Update attributes.
         * \a valid is a bitmask composed from enum #cl_attr_valid, and
         * indicating what attributes are to be set.
         * \pre cl_object_header::coh_attr_guard of the top-object is locked.
         * \return the same convention as for
         * cl_object_operations::coo_attr_get() is used. */
        int (*coo_attr_set)(const struct lu_env *env, struct cl_object *obj,
                            const struct cl_attr *attr, unsigned valid);
};
```

Extended header for client object.

```
struct cl_object_header {
        /* Standard lu_object_header. cl_object::co_lu::lo_header points
         * here. */
        struct lu_object_header  coh_lu;
        /* \todo XXX move locks below to the separate cache-lines, they are
         * mostly useless otherwise. */
        /* Lock protecting page tree. */
        spinlock_t               coh_page_guard;
        /* Lock protecting lock list. */
        spinlock_t               coh_lock_guard;
        /* Radix tree of cl_page's, cached for this object. */
        struct radix_tree_root   coh_tree;
        /* List of cl_lock's granted for this object. */
        struct list_head         coh_locks;
        /* Parent object. It is assumed that an object has a well-defined
         * parent, but not a well-defined child (there may be multiple
         * sub-objects, for the same top-object). cl_object_header::coh_parent
         * field allows certain code to be written generically, without
         * limiting possible cl_object layouts unduly. */
        struct cl_object_header *coh_parent;
        /* Protects consistency between cl_attr of parent object and
         * attributes of sub-objects, that the former is calculated ("merged")
         * from.
         * \todo XXX this can be read/write lock if needed. */
        spinlock_t               coh_attr_guard;
};
```

7.2.2 entry points

Returns the top-object for a given \a o.

```
struct cl_object *cl_object_top (struct cl_object *o);
```

Returns a cl_object with a given \a fid. Returns either cached or newly created object. Additional reference on the returned object is acquired.

```
struct cl_object *cl_object_find(const struct lu_env *env, struct cl_device *cd,
                                 const struct lu_fid *fid,
                                 struct cl_object_conf *c);
```

Initialize cl_object_header.

```
int  cl_object_header_init(struct cl_object_header *h);
```

Finalize cl_object_header.

```
void cl_object_header_fini(struct cl_object_header *h);
```

Releases a reference on \a o. When last reference is released object is returned to the cache, unless lu_object_header_flags::LU_OBJECT_HEARD_BANSHEE bit is set in its header.

```
void cl_object_put        (const struct lu_env *env, struct cl_object *o);
```

Acquire an additional reference to the object \a o. This can only be used to acquire *additional* reference, i.e., caller
already has to possess at least one reference to \a o before calling this.

```
void cl_object_get        (struct cl_object *o);
```

Locks data-attributes. Prevents data-attributes from changing, until lock is released by cl_object_attr_unlock(). This has to be called before calls to cl_object_attr_get(), cl_object_attr_set().

```
void cl_object_attr_lock  (struct cl_object *o);
```

Releases data-attributes lock, acquired by cl_object_attr_lock().

```
void cl_object_attr_unlock(struct cl_object *o);
```

Returns data-attributes of an object \a obj. Every layer is asked (by calling cl_object_operations::coo_attr_get top-to-bottom to fill in parts of \a attr that this layer is responsible for.

```
int  cl_object_attr_get   (const struct lu_env *env, struct cl_object *obj,
                           struct cl_attr *attr);
```

Updates data-attributes of an object \a obj. Only attributes, mentioned in a validness bit-mask \a v are updated. Calls cl_object_operations::coo_attr_set() on every layer, bottom to top.

```
int  cl_object_attr_set   (const struct lu_env *env, struct cl_object *obj,
                           const struct cl_attr *attr, unsigned valid);
```

Returns true, iff \a o0 and \a o1 are slices of the same object.

```
int cl_object_same(struct cl_object *o0, struct cl_object *o1);
```

7.3 cl_lock

7.3.1 data-structures

Lock mode. For the client extent locks. cl_lock_mode_match() assumes particular ordering here.

```
enum cl_lock_mode {
        CLM_READ,
        CLM_WRITE
};
```

Lock description.

```
struct cl_lock_descr {
        /* Object this lock is granted for. */
        struct cl_object *cld_obj;
        /* Index of the first page protected by this lock. */
        pgoff_t           cld_start;
        /* Index of the last page (inclusive) protected by this lock. */
        pgoff_t           cld_end;
        /* Lock mode. */
        enum cl_lock_mode cld_mode;
};
enum cl_lock_state {
        /* Lock that wasn't yet enqueued */
        CLS_NEW,
        /* Enqueue is in progress, blocking for some intermediate interaction
         * with the other side. */
        CLS_QUEUING,
        /* Lock is fully enqueued, waiting for server to reply when it is
         * granted. */
        CLS_ENQUEUED,
        /* Lock granted, actively used by some IO. */
        CLS_HELD,
        /* Lock granted, but not used, blocking in the communication. */
        CLS_UNLOCKING,
        /* Lock granted, not used. */
        CLS_CACHED,
        /* Lock is being destroyed. */
        CLS_FREEING,
        CLS_NR
};
enum cl_lock_flags {
        /* state change is pending */
        CLF_STATE,
        /* no new references to this lock can be acquired. It also means, that
         * lock is not cached in the object cl_object_header::coh_locks list
         * when last reference to it is released.
         * Protected by cl_object_header::coh_lock_guard. */
        CLF_NOREFS,
        /* blocking ast has been delivered to this lock. */
        CLF_GOT_BLOCK,
        /* lock has been cancelled. */
        CLF_CANCELLED
};
```

Layered client lock.

```
struct cl_lock {
        /* Reference counter. */
        atomic_t              cll_ref;
        /* List of slices. Immutable after creation. */
        struct list_head      cll_layers;
        /* Linkage into cl_lock::cll_descr::cld_obj::coh_locks list. Protected
         * by cl_lock::cll_descr::cld_obj::coh_lock_guard. */
        struct list_head      cll_linkage;
        /* Parameters of this lock. */
        struct cl_lock_descr  cll_descr;
        struct mutex          cll_guard;
        /* Protected by cl_lock::cll_guard. */
        enum cl_lock_state    cll_state;
        /* signals state changes. */
        cfs_waitq_t           cll_wq;
        int                   cll_error;
        /* Number of lock users. Valid in cl_lock_state::CLS_HELD
         * state. Protected by cl_lock::cll_guard. */
        int                   cll_lockcnt;
        /* Atomic flags bit-mask. Values from enum cl_lock_flags. */
        unsigned long         cll_flags;
};
```

Per-layer part of cl_lock

```
struct cl_lock_slice {
        struct cl_lock                  *cls_lock;
        /* Object slice corresponding to this lock slice. Immutable after
         * creation. */
        struct cl_object                *cls_obj;
        const struct cl_lock_operations *cls_ops;
        /* Linkage into cl_lock::cll_layers. Immutable after creation. */
        struct list_head                 cls_linkage;
};
```

Possible (non-error) return values of ->clo_{enqueue,wait,unlock}().

```
enum cl_lock_transition {
        /* operation had to release lock mutex, restart. */
        CLO_REPEAT = 1,
        /* operation cannot be completed immediately. Wait for state change. */
        CLO_WAIT   = 2
};
struct cl_lock_operations {
        /* State machine transitions. These 3 methods are called to transfer
         * lock from one state to another, as described in the commentary
         * above enum #cl_lock_state.
         * \retval 0          this layer has nothing more to do to before
         *                       transition to the target state happens;
         * \retval CLO_REPEAT method had to release and re-acquire cl_lock
         *                    mutex, repeat invocation of transition method
         *                    across all layers;
         * \retval CLO_WAIT   this layer cannot move to the target state
```

```
 *                    immediately, as it has to wait for certain event
 *                    (e.g., the communication with the server). It
 *                    is guaranteed, that when the state transfer
 *                    becomes possible, cl_lock::cll_wq wait-queue
 *                    is signaled. Caller can wait for this event by
 *                    calling cl_lock_state_wait();
 * \retval -ve        failure, abort state transition, move the lock
 *                    into cl_lock_state::CLS_FREEING state, and set
 *                    cl_lock::cll_error.
 * Once all layers voted to agree to transition (by returning 0), lock
 * is moved into corresponding target state. All state transition
 * methods are optional. */
/* Attempts to enqueue the lock. Called top-to-bottom. */
int  (*clo_enqueue)(const struct lu_env *env,
                    struct cl_lock_slice *slice, __u32 enqflags);
/* Attempts to wait for enqueue result. Called top-to-bottom. */
int  (*clo_wait)(const struct lu_env *env, struct cl_lock_slice *slice);
/* Attempts to unlock the lock. Called bottom-to-top. */
int  (*clo_unlock)(const struct lu_env *env,
                   struct cl_lock_slice *slice);
/* A method invoked when lock state is changed (as a result of state
 * transition). This is used, for example, to track when the state of
 * a sub-lock changes, to propagate this change to the corresponding
 * top-lock. Optional */
void (*clo_state)(const struct lu_env *env,
                  struct cl_lock_slice *slice, enum cl_lock_state st);
/* Returns true, iff given lock is suitable for the given io, idea
 * being, that there are certain "unsafe" locks, e.g., ones acquired
 * for O_APPEND writes, that we don't want to re-use for a normal
 * write, to avoid the danger of cascading evictions. Optional. Runs
 * under cl_object_header::coh_lock_guard. */
int (*clo_fits_into)(const struct lu_env *env,
                     const struct cl_lock_slice *slice,
                     const struct cl_io *io);
/* Asynchronous System Traps. All of then are optional, all are
 * executed bottom-to-top. */
/* Blocking ast. Executed when blocking ast arrives for this lock. */
void (*clo_block)(const struct lu_env *env,
                  struct cl_lock_slice *slice);
/* Cancellation callback. This is not, strictly speaking, an ast. This
 * is executed to notify layer that lock is being canceled. */
void (*clo_cancel)(const struct lu_env *env,
                   struct cl_lock_slice *slice);
/* Glimpse ast. Executed when glimpse ast arrives for this
 * lock. Layers are supposed to fill parts of \a lvb that will be
 * shipped to the glimpse originator as a glimpse result. */
int (*clo_glimpse)(const struct lu_env *env,
                   struct cl_lock_slice *slice, struct ost_lvb *lvb);
/* Reception ast. Executed when a reply to enqueue is received from
 * the server. */
int (*clo_receive)(const struct lu_env *env,
                   struct cl_lock_slice *slice, struct ost_lvb *lvb,
                   int rc);
/* Executed top-to-bottom when lock description changes (e.g., as a
```

```
         * result of server granting more generous lock than was requested). */
        int (*clo_modify)(const struct lu_env *env, struct cl_lock_slice *slice,
                          struct cl_lock_descr *updated);
        /* Destructor. Frees resources and the slice. */
        void (*clo_fini)(const struct lu_env *env, struct cl_lock_slice *slice);
        /* Optional debugging helper. Prints given lock slice. */
        int (*clo_print)(const struct lu_env *env,
                         void *cookie, lu_printer_t p,
                         const struct cl_lock_slice *slice);
};
```

Flags to lock enqueue procedure.

```
enum cl_enq_flags {
        /* instruct server to not block, if conflicting lock is found. Instead
         * -EWOULDBLOCK is returned immediately. */
        CEF_NONBLOCK     = 0x00000001,
        /* take lock asynchronously (out of order), as it cannot
         * deadlock. This is for LDLM_FL_HAS_INTENT locks used for glimpsing. */
        CEF_ASYNC        = 0x00000002,
        /* tell the server to instruct (though a flag in the blocking ast) an
         * owner of the conflicting lock, that it can drop dirty pages
         * protected by this lock, without sending them to the server. */
        CEF_DISCARD_DATA = 0x00000004
};
```

7.3.2 entry points

Returns a lock matching description \a need. This is the main entry point into the cl_lock caching interface. First, a cache (implemented as a per-object linked list) is consulted. If lock is found there, it is returned immediately. Otherwise new lock is allocated and returned. In any case, additional reference to lock is acquired.

```
struct cl_lock       *cl_lock_find(const struct lu_env *env, struct cl_io *io,
                                   const struct cl_lock_descr *need);
```

Returns a slice within a lock, corresponding to the given layer in the device stack.

```
struct cl_lock_slice *cl_lock_at  (const struct cl_lock *lock,
                                   const struct lu_device_type *dtype);
```

Prints human readable representation of \a lock to the \a f.

```
void cl_lock_print(const struct lu_env *env, void *cookie,
                   lu_printer_t printer, const struct cl_lock *lock);
```

Acquires an additional reference to a lock. This can be called only by caller already possessing a reference to \a lock.

```
void  cl_lock_get (const struct lu_env *env, struct cl_lock *lock);
void  cl_lock_put (const struct lu_env *env, struct cl_lock *lock);
```

Interface to lock state machine consists of 3 parts:

- "try" functions that attempt to effect a state transition. If state transition is not possible right now (e.g., if it has to wait for some asynchronous event to occur), these functions return cl_lock_transition::CLO_WAIT.
- "non-try" functions that implement synchronous blocking interface on top of non-blocking "try" functions. These functions repeatedly call corresponding "try" versions, and if state transition is not possible immediately, wait for lock state change.
- methods from cl_lock_operations, called by "try" functions. Lock can be advanced to the target state only when all layers voted that they are ready for this transition. "Try" functions call methods under lock mutex. If a layer had to release a mutex, it re-acquires it and returns cl_lock_transition::CLO_REPEAT, causing "try" function to call all layers again.

TRY	NON-TRY	METHOD	FINAL STATE
cl_enqueue_try()	cl_enqueue()	->clo_enqueue()	CLS_ENQUEUED
cl_wait_try()	cl_wait()	->clo_wait()	CLS_HELD
cl_unlock_try()	cl_unlock()	->clo_unlock()	CLS_CACHED

Enqueues a lock.

```
int   cl_enqueue  (const struct lu_env *env, struct cl_lock *lock, __u32 flags);
```

Waits until enqueued lock is granted.

```
int   cl_wait     (const struct lu_env *env, struct cl_lock *lock);
```

Unlocks a lock.

```
void  cl_unlock   (const struct lu_env *env, struct cl_lock *lock);
```

Tries to enqueue a lock. This function is called repeatedly by cl_enqueue() until either lock is enqueued, or error occurs.

\post ergo(result == 0, lock->cll_state == CLS_ENQUEUED)

```
int   cl_enqueue_try(const struct lu_env *env, struct cl_lock *lock,
                     __u32 flags);
```

Tries to unlock a lock. This function is called repeatedly by cl_unlock() until either lock is unlocked, or error occurs.

\post ergo(result == 0, lock->cll_state == CLS_CACHED)

```
int   cl_unlock_try (const struct lu_env *env, struct cl_lock *lock);
```

Tries to wait for a lock. This function is called repeatedly by cl_wait() until either lock is granted, or error occurs.

\post ergo(result == 0, lock->cll_state == CLS_HELD)

```
int   cl_wait_try   (const struct lu_env *env, struct cl_lock *lock);
```

Notifies waiters that lock state changed. Wakes up all waiters sleeping in cl_lock_state_wait(), also notifies all layers about state change by calling cl_lock_operations::clo_state() top-to-bottom.

```
void cl_lock_signal      (const struct lu_env *env, struct cl_lock *lock);
```

Waits until lock state is changed. This function is called with cl_lock mutex locked, atomically released mutex and goes to sleep, waiting for a lock state change (signaled by cl_lock_signal()), and re-acquired mutex before return.

This function is used to wait until lock state machine makes some progress and to emulate synchronous operations on top of asynchronous lock interface.

\retval -EINTR wait for interrupted
\retval 0 wait wasn't interrupted
\pre mutex_is_locked(&lock->cll_guard)

```
int  cl_lock_state_wait  (const struct lu_env *env, struct cl_lock *lock);
```

Changes lock state. This function is invoked to notify layers that lock state changed, possible as a result of an asynchronous event such as call-back reception.

\post lock->cll_state == state

```
void cl_lock_state_set   (const struct lu_env *env, struct cl_lock *lock,
                          enum cl_lock_state state);
```

Check whether \a queue contains locks matching \a need.

\retval +ve there is a matching lock in the \a queue
\retval 0 there are no matching locks in the \a queue

```
int  cl_queue_match      (const struct list_head *queue,
                          const struct cl_lock_descr *need);
```

Locks cl_lock object. This is used to manipulate cl_lock fields, and to serialize state transitions in the lock state machine.

\post mutex_is_locked(&lock->cll_guard)

```
void cl_lock_lock   (const struct lu_env *env, struct cl_lock *lock);
```

Try-locks cl_lock object.

\retval 0 \a lock was successfully locked
\retval -EBUSY \a lock cannot be locked right now
\post ergo(result == 0, mutex_is_locked(&lock->cll_guard))

```
int  cl_lock_trylock(const struct lu_env *env, struct cl_lock *lock);
```

Unlocks cl_lock object.

\pre mutex_is_locked(&lock->cll_guard)

```
void cl_lock_unlock (const struct lu_env *env, struct cl_lock *lock);
```

Invalidate pages protected by the given lock, sending them out to the server first, if necessary. This function does the following:

- collects a list of pages to be invalidated,
- unmaps them from the user virtual memory,
- sends dirty pages to the server,
- waits for transfer completion,
- discards pages, and throws them out of memory.

If \a discard is set, pages are discarded without sending them to the server. If error happens on any step, the process continues anyway (the reasoning behind this being that lock cancellation cannot be delayed indefinitely).

```
int  cl_lock_page_out   (const struct lu_env *env, struct cl_lock *lock,
                         int discard);
```

Returns true iff a lock with the description \a has provides at least the same guarantees as a lock with the description \a need.

```
int  cl_lock_descr_match(const struct cl_lock_descr *has,
                         const struct cl_lock_descr *need);
```

Returns true iff a lock with the mode \a has provides at least the same guarantees as a lock with the mode \a need.

```
int  cl_lock_mode_match (enum cl_lock_mode has, enum cl_lock_mode need);
```

Notifies layers that lock description changed. The server can grant client a lock different from one that was requested (e.g., larger in extent). This method is called when actually granted lock description becomes known to let layers to accommodate for changed lock description.

```
int  cl_lock_modify     (const struct lu_env *env, struct cl_lock *lock,
                         struct cl_lock_descr *desc);
```

Notifies layers (bottom-to-top) that blocking AST was received. Blocking AST notification is delivered to layers at most once.

```
void cl_ast_block  (const struct lu_env *env, struct cl_lock *lock);
```

Destroys this lock. Notifies layers (bottom-to-top) that lock is being destroyed, then destroy the lock. If there are holds on the lock, postpone destruction until all holds are released. This is called when a decision is made to destroy the lock in the future. E.g., when a blocking AST is received on it, or fatal communication error happens.

Caller must have a reference on this lock to prevent a situation, when deleted lock lingers in memory for indefinite time, because nobody calls cl_lock_put() to finish it.

\pre atomic_read(&lock->cll_ref) > 0
\see cl_lock_operations::clo_delete()
\see cl_lock::cll_holds

```
void cl_lock_delete(const struct lu_env *env, struct cl_lock *lock);
```

Notifies layers (bottom-to-top) that lock is being canceled. Cancellation notification is delivered to layers at most once.

```
void cl_lock_cancel (const struct lu_env *env, struct cl_lock *lock);
```

Notifies layers (bottom-to-top) that glimpse AST was received. Layers have to fill \a lvb fields with information that will be shipped back to glimpse issuer.

```
int  cl_ast_glimpse(const struct lu_env *env, struct cl_lock *lock,
                    struct ost_lvb *lvb);
```

Notifies layers (bottom-to-top) that a response for a enqueue request was received.

```
int  cl_ast_receive(const struct lu_env *env, struct cl_lock *lock,
                    struct ost_lvb *lvb, int rc);
```

7.4 cl_page

7.4.1 data-structures

cl_page states.

```
enum cl_page_state {
        CPS_CACHED,
        CPS_OWNED,
        CPS_PAGEOUT,
        CPS_PAGEIN,
        CPS_FREEING,
        CPS_NR
};
```

Fields are protected by the lock on cfs_page_t, except for atomics and immutables.

```
struct cl_page {
        /* Reference counter. */
        atomic_t           cp_ref;
        /* An object this page is a part of. Immutable after creation. */
        struct cl_object  *cp_obj;
        /* Logical page index within the object. Immutable after creation. */
        pgoff_t            cp_index;
        /* List of slices. Immutable after creation. */
        struct list_head   cp_layers;
        /* Parent page, NULL for top-level page. Immutable after creation. */
        struct cl_page    *cp_parent;
        /* Lower-layer page. NULL for bottommost page. Immutable after
         * creation. */
        struct cl_page    *cp_child;
        enum cl_page_state cp_state;
        /* Linkage of pages within some group. */
        struct list_head   cp_batch;
        /* Linkage of pages within cl_req. */
        struct list_head   cp_flight;
        int                cp_error;
```

```
        /* Owning IO in cl_page_state::CPS_OWNED state. Sub-page can be owned
         * by sub-io. */
        struct cl_io      *cp_owner;
        /* Owning IO request in cl_page_state::CPS_PAGEOUT and
         * cl_page_state::CPS_PAGEIN states. This field is maintained only in
         * the top-level pages. */
        struct cl_req     *cp_req;
};
```

Per-layer part of cl_page.

```
struct cl_page_slice {
        struct cl_page                  *cpl_page;
        /* Object slice corresponding to this page slice. Immutable after
         * creation. */
        struct cl_object                *cpl_obj;
        const struct cl_page_operations *cpl_ops;
        /* Linkage into cl_page::cp_layers. Immutable after creation. */
        struct list_head                 cpl_linkage;
};
```

Per-layer page operations.

Methods taking an \a io argument are for the activity happening in the context of given \a io. Page is assumed to be owned by that io, except for the obvious cases (like cl_page_operations::cpo_own()).

```
struct cl_page_operations {
        /* cl_page<->cfs_page_t methods. Only one layer in the stack has to
         * implement these. Current code assumes that this functionality is
         * provided by the topmost layer, see cl_page_disown0() as an example. */
        /* \return the underlying VM page. Optional. */
        cfs_page_t *(*cpo_vmpage)(const struct lu_env *env,
                                  const struct cl_page_slice *slice);
        /* Called when \a io acquires this page into the exclusive
         * ownership. When this method returns, it is guaranteed that the is
         * not owned by other io, and no transfer is going on against
         * it. Optional. */
        void (*cpo_own)(const struct lu_env *env,
                        struct cl_page_slice *slice, struct cl_io *io);
        /* Called when ownership it yielded. Optional. */
        void (*cpo_disown)(const struct lu_env *env,
                           struct cl_page_slice *slice, struct cl_io *io);
        /* Called for a page that is already "owned" by \a io from VM point of
         * view. Optional. */
        void (*cpo_assume)(const struct lu_env *env,
                           struct cl_page_slice *slice, struct cl_io *io);
        /* Announces that page contains valid data and user space can look and
         * them without client's involvement from now on. Effectively marks
         * the page up-to-date. Optional. */
        void  (*cpo_export)(const struct lu_env *env,
                            struct cl_page_slice *slice);
        /* Unmaps page from the user space (if it is mapped). */
        int (*cpo_unmap)(const struct lu_env *env,
                         struct cl_page_slice *slice, struct cl_io *io);
        /* Page destruction. */
```

```
/* Called when page is truncated from the object. Optional. */
void (*cpo_discard)(const struct lu_env *env,
                    struct cl_page_slice *slice, struct cl_io *io);
/* Called when page is removed from the cache, and is about to being
 * destroyed. Optional. */
void (*cpo_delete)(const struct lu_env *env,
                   struct cl_page_slice *slice);
/* Destructor. Frees resources and slice itself. */
void (*cpo_fini)(const struct lu_env *env, struct cl_page_slice *slice);
/* Checks whether the page is protected by a cl_lock. This is a
 * per-layer method, because certain layers have ways to check for the
 * lock much more efficiently than through the generic locks scan, or
 * implement locking mechanisms separate from cl_lock, e.g.,
 * LL_FILE_GROUP_LOCKED in clu. If \a pending is true, check for locks
 * being canceled, or scheduled for cancellation as soon as the last
 * user goes away, too.
 *
 * \return    -EBUSY: page is protected by a lock of a given mode;
 * \return  -ENODATA: page is not protected by a lock;
 * \return         0: this layer cannot decide. */
int (*cpo_is_under_lock)(const struct lu_env *env,
                         struct cl_page_slice *slice, struct cl_io *io,
                         enum cl_lock_mode mode, int pending);
/* Optional debugging helper. Prints given page slice. */
int (*cpo_print)(const struct lu_env *env, struct cl_page_slice *slice,
                 void *cookie, lu_printer_t p);
/* Transfer methods. See comment on cl_req for a description of
 * transfer formation and life-cycle. */
/* Request type dependent vector of operations.
 * Transfer operations depend on transfer mode (cl_req_type). To avoid
 * passing transfer mode to each and every of these methods, and to
 * avoid branching on request type inside of the methods, separate
 * methods for cl_req_type:CRT_READ and cl_req_type:CRT_WRITE are
 * provided. That is, method invocation usually looks like
 *
 *         slice->cp_ops.io[req->crq_type].cpo_method(env, slice, ...); */
struct {
        /* Called when a page is submitted for a transfer as a part of
         * cl_page_list.
         * \return    0       : page is eligible for submission;
         * \return    -ENOENT : skip this page;
         * \return    -ve     : error. */
        int  (*cpo_prep)(const struct lu_env *env,
                         struct cl_page_slice *slice, struct cl_io *io);
        /* Completion handler. This is guaranteed to be eventually
         * fired after cl_page_operations::cpo_prep() or
         * cl_page_operations::cpo_make_ready() call. */
        void (*cpo_completion)(const struct lu_env *env,
                               struct cl_page_slice *slice, int ioret);
        /* Called when cached page is about to be added to the
         * cl_req as a part of req formation.
         * \return    0       : proceed with this page;
         * \return    -EAGAIN : skip this page;
         * \return    -ve     : error. */
```

```
                int  (*cpo_make_ready)(const struct lu_env *env,
                                       struct cl_page_slice *slice);
                /* Announce that this page is to be written out
                 * opportunistically, that is, page is dirty, it is not
                 * necessary to start write-out transfer right now, but
                 * eventually page has to be written out.
                 * Main caller of this is the write path (see
                 * clu_io_commit_write()), using this method to build a
                 * "transfer cache" from which large transfers are then
                 * constructed by the req-formation engine. */
                int  (*cpo_cache_add)(const struct lu_env *env,
                                      struct cl_page_slice *slice,
                                      struct cl_io *io);
        } io[CRT_NR];
        /**
         * Tell transfer engine that only [to, from] part of a page should be
         * transmitted.
         *
         * This is used for immediate transfers.
         *
         * \todo XXX this is not very good interface. It would be much better
         * if all transfer parameters were supplied as arguments to
         * cl_io_operations::cio_submit() call, but it is not clear how to do
         * this for page queues.
         *
         * \see cl_page_clip()
         */
        void (*cpo_clip)(const struct lu_env *env, struct cl_page_slice *slice,
                         int from, int to);
};
```

7.4.2 entry points

Returns a page with given index in the given object, or NULL if no page is found. Acquires a reference on \a page.

Locking: called under cl_object_header::coh_page_guard spin-lock.

```
struct cl_page       *cl_page_lookup(struct cl_object_header *hdr,
                                     pgoff_t index);
```

Returns a cl_page with index \a idx at the object \a o, and associated with the VM page \a vmpage.

This is the main entry point into the cl_page caching interface. First, a cache (implemented as a per-object radix tree) is consulted. If page is found there, it is returned immediately. Otherwise new page is allocated and returned. In any case, additional reference to page is acquired.

```
struct cl_page       *cl_page_find  (const struct lu_env *env,
                                     struct cl_object *o,
                                     pgoff_t idx, struct page *vmpage);
```

Acquires an additional reference to a page. This can be called only by caller already possessing a reference to \a page.

```
void                    cl_page_get   (struct cl_page *page);
```

Releases a reference to a page. When last reference is released, page is returned to the cache, unless it is in cl_page_state::CPS_FREEING state, in which case it is immediately destroyed.

```
void                    cl_page_put   (const struct lu_env *env,
                                       struct cl_page *page);
```

Returns a VM page associated with a given cl_page.

```
cfs_page_t             *cl_page_vmpage(const struct lu_env *env,
                                       struct cl_page *page);
```

Returns a cl_page associated with a VM page, and given cl_object.

```
struct cl_page         *cl_vmpage_page(cfs_page_t *vmpage, struct cl_object *obj);
```

Returns a slice within a page, corresponding to the given layer in the device stack.

```
struct cl_page_slice *cl_page_at    (const struct cl_page *page,
                                     const struct lu_device_type *dtype);
```

Returns the top-page for a given page.

```
struct cl_page         *cl_page_top   (struct cl_page *page);
```

Prints human readable representation of \a pg to the \a f.

```
void cl_page_print(const struct lu_env *env, void *cookie,
                   lu_printer_t printer, struct cl_page *pg);
```

Functions dealing with the ownership of page by io.

Owns page by IO.

Waits until page is in cl_page_state::CPS_CACHED state, and then switch it into cl_page_state::CPS_OWNED state.

\pre !cl_page_is_owned(pg, io)

\post result == 0 iff cl_page_is_owned(pg, io)

\retval 0 success

\retval -ve failure, e.g., page was destroyed (and landed in cl_page_state::CPS_FREEING instead of cl_page_state::CPS_CACHED).

```
int  cl_page_own        (const struct lu_env *env,
                         struct cl_io *io, struct cl_page *page);
```

Assume page ownership. Called when page is already locked by the hosting VM.

\pre !cl_page_is_owned(pg, io)

\post cl_page_is_owned(pg, io)

```
void cl_page_assume     (const struct lu_env *env,
                         struct cl_io *io, struct cl_page *page);
```

Releases page ownership without unlocking the page. Moves page into cl_page_state::CPS_CACHED without releasing a lock on the underlying VM page (as VM is supposed to do this itself).

\pre cl_page_is_owned(pg, io)

\post !cl_page_is_owned(pg, io)

```
void cl_page_unassume   (const struct lu_env *env,
                         struct cl_io *io, struct cl_page *pg);
```

Releases page ownership. Moves page into cl_page_state::CPS_CACHED.

\pre cl_page_is_owned(pg, io)

\post !cl_page_is_owned(pg, io)

```
void cl_page_disown     (const struct lu_env *env,
                         struct cl_io *io, struct cl_page *page);
```

Returns true, iff page is owned by the given IO.

```
int  cl_page_is_owned   (const struct cl_page *pg, const struct cl_io *io);
```

Functions dealing with the preparation of a page for a transfer, and tracking transfer state.

Prepares page for immediate transfer. cl_page_operations::cpo_prep() is called top-to-bottom. Every layer either agrees to submit this page (by returning 0), or requests to omit this page (by returning -ENOENT). Layer handling interactions with the VM also has to inform VM that page is under transfer now.

```
int  cl_page_prep       (const struct lu_env *env, struct cl_io *io,
                         struct cl_page *pg, enum cl_req_type crt);
```

Notify layers about transfer completion.

Invoked by transfer sub-system (which is a part of osc) to notify layers that a transfer, of which this page is a part of has completed. Completion call-backs are executed in the bottom-up order, so that uppermost layer (llite), responsible for the VFS/VM interaction runs last and can release locks safely.

\pre pg->cp_state == CPS_PAGEIN || pg->cp_state == CPS_PAGEOUT

\post pg->cp_state == CPS_CACHED

```
void cl_page_completion (const struct lu_env *env,
                         struct cl_page *pg, enum cl_req_type crt, int ioret);
```

Notify layers that transfer formation engine decided to yank this page from the cache and to make it a part of a transfer.

\pre pg->cp_state == CPS_CACHED

\post pg->cp_state == CPS_PAGEIN || pg->cp_state == CPS_PAGEOUT

```
int  cl_page_make_ready (const struct lu_env *env, struct cl_page *pg,
                         enum cl_req_type crt);
```

Notify layers that high level io decided to place this page into a cache for future transfer. The layer implementing transfer engine (osc) has to register this page in its queues.

\pre cl_page_is_owned(pg, io)

\post pg->cp_state == CPS_PAGEIN || pg->cp_state == CPS_PAGEOUT

```
int  cl_page_cache_add  (const struct lu_env *env, struct cl_io *io,
                         struct cl_page *pg, enum cl_req_type crt);
```

VM interaction

Called when page is to be removed from the object, e.g., as a result of truncate. Calls cl_page_operations::cpo_discard() top-to-bottom.

\pre cl_page_is_owned(pg, io)

```
void cl_page_discard    (const struct lu_env *env, struct cl_io *io,
                         struct cl_page *pg);
```

Called when a decision is made to throw page out of memory. Notifies all layers about page destruction by calling cl_page_operations::cpo_delete() method top-to-bottom. Moves page into cl_page_state::CPS_FREEING state (this is the only place where transition to this state happens). Eliminates all venues through which new references to the page can be obtained:

- removes page from the radix trees,
- breaks linkage from VM page to cl_page.

Once page reaches cl_page_state::CPS_FREEING, all remaining references will drain after some time, at which point page will be recycled.

\pre pg == cl_page_top(pg)

\pre VM page is locked

\post pg->cp_state == CPS_FREEING

```
void cl_page_invalidate (const struct lu_env *env, struct cl_page *pg);
```

Unmaps page from user virtual memory. Calls cl_page_operations::cpo_unmap() through all layers top-to-bottom. The layer responsible for VM interaction has to unmap page from user space virtual memory.

```
int  cl_page_unmap      (const struct lu_env *env, struct cl_io *io,
                         struct cl_page *pg);
```

Marks page up-to-date. Call cl_page_operations::cpo_export() through all layers top-to-bottom. The layer responsible for VM interaction has to mark page as up-to-date. From this moment on, page can be shown to the user space without Lustre being notified, hence the name.

```
void cl_page_export     (const struct lu_env *env, struct cl_page *pg);
```

Checks whether page is protected by any extent lock is at least required mode.

\return the same as in cl_page_operations::cpo_is_under_lock() method.

```
int  cl_page_is_under_lock(const struct lu_env *env, struct cl_io *io,
                           struct cl_page *page, enum cl_lock_mode mode,
                           int pending);
```

Converts a byte offset within object \a obj into a page index.

```
loff_t   cl_offset(const struct cl_object *obj, pgoff_t idx);
```

Converts a page index into a byte offset within object \a obj.

```
pgoff_t  cl_index (const struct cl_object *obj, loff_t offset);
```

7.5 cl_io

7.5.1 data-structures

IO types

```
enum cl_io_type {
        /* read system call */
        CIT_READ,
        /* write system call */
        CIT_WRITE,
        /* truncate system call */
        CIT_TRUNC,
        /* Cancellation of an extent lock. This io exists as a context to
         * write dirty pages from under the lock being canceled back to the
         * server */
        CIT_MISC,
        /* page fault handling */
        CIT_FAULT,
        CIT_OP_NR
};
```

per-layer io operations

```
struct cl_io_operations {
        /* Vector of io state transition methods for every io type. */
        struct {
                /* Initialize io state for a given layer.
                 * called top-to-bottom once per io existence to initialize io
                 * state. If layer wants to keep some state for this type of
                 * io, it has to use lu_env::le_ses for that. It is guaranteed
                 * that all threads participating in this io share the same
                 * session. */
                int  (*cio_init) (const struct lu_env *env,
                                  struct cl_object *obj, struct cl_io *io);
                /* Prepare io iteration at a given layer.
                 * Called top-to-bottom at the beginning of each iteration of
                 * "io loop" (if it makes sense for this type of io). Here
                 * layer selects what work it will do during this iteration. */
                void (*cio_prep) (const struct lu_env *env,
                                  struct cl_object *obj, struct cl_io *io);
                /* Collect locks for the current iteration of io.
                 * Called top-to-bottom to collect all locks necessary for
                 * this iteration. This methods shouldn't actually enqueue
                 * anything, instead it should post a lock through
                 * cl_io_lock_add(). Once all locks are collected, they are
                 * sorted and enqueued in the proper order. */
                int  (*cio_lock) (const struct lu_env *env, struct cl_io *io);
                /* Start io iteration.
                 * Once all locks are acquired, called top-to-bottom to
                 * commence actual IO. In the current implementation,
                 * top-level clu_io_{read,write}_start() does all the work
                 * synchronously by calling generic_file_*(), so other layers
                 * are called when everything is done. */
                int  (*cio_start)(const struct lu_env *env, struct cl_io *io);
```

```
                /* Called top-to-bottom at the end of io loop. Here layer
                 * might wait for an unfinished asynchronous io. */
                void (*cio_end)  (const struct lu_env *env, struct cl_io *io);
                /* Called once per io, bottom-to-top to release io resources. */
                void (*cio_fini) (const struct lu_env *env, struct cl_io *io);
        } op[CIT_OP_NR];
        struct {
                /* Submit pages from \a qin for IO, and move successfully
                 * submitted pages into \a qout. Return non-zero if failed to
                 * submit even the single page. If submission failed after
                 * some pages were moved into \a qout, completion callback with
                 * non-zero ioret is executed on them. */
                int  (*cio_submit)(const struct lu_env *env, struct cl_io *io,
                                   enum cl_req_type crt,
                                   struct cl_page_list *qin,
                                   struct cl_page_list *qout);
        } req_op[CRT_NR];
        /* Read missing page.
         * Called by top-level cl_io_operations::op[CIT_READ]::cio_start()
         * method, when it hits not-up-to-date page in the range. Optional.
         * \pre io->ci_type == CIT_READ */
        int (*cio_read_page)(const struct lu_env *env, struct cl_io *io,
                             struct cl_page_slice *page);
        /* Prepare write of a \a page.
         * \pre io->ci_type == CIT_WRITE */
        int (*cio_prepare_write)(const struct lu_env *env, struct cl_io *io,
                                 struct cl_page_slice *page,
                                 unsigned from, unsigned to);
        /*
         * \pre io->ci_type == CIT_WRITE */
        int (*cio_commit_write)(const struct lu_env *env, struct cl_io *io,
                                struct cl_page_slice *page,
                                unsigned from, unsigned to);
        /* Optional debugging helper. Print given io slice. */
        int (*cio_print)(const struct lu_env *env, void *cookie,
                         lu_printer_t p, const struct cl_io *io);
};
```

Link between lock and io. Intermediate structure is needed, because the same lock can be part of multiple io's simultaneously.

```
struct cl_io_lock_link {
        /* linkage into one of cl_lockset lists. */
        struct list_head  cill_linkage;
        struct cl_lock   *cill_lock;
        __u32             cill_enq_flags;
        /* optional destructor */
        void            (*cill_fini)(const struct lu_env *env,
                                     struct cl_io_lock_link *link);
};
```

Lock-set represents a collection of locks, that io needs at a time. Generally speaking, client tries to avoid holding multiple locks when possible, because

- holding extent locks over multiple ost's introduces the danger of "cascading time-outs";

- holding multiple locks over the same ost is still dead-lock prone, see comment in osc_lock_enqueue(),

but there are certain situations where this is unavoidable:

- O_APPEND writes have to take [0, EOF] lock for correctness;
- truncate has to take [new-size, EOF] lock for correctness;
- SNS has to take locks across full stripe for correctness;
- in the case when user level buffer, supplied to {read,write}(file0), is a part of a memory mapped lustre file, client has to take a dlm locks on file0, and all files that back up the buffer (or a part of the buffer, that is being processed in the current chunk, in any case, there are situations where at least 2 locks are necessary).

In such cases we at least try to take locks in the same consistent order. To this end, all locks are first collected, then sorted, and then enqueued.

```
struct cl_lockset {
        /* locks to be acquired. */
        struct list_head cls_todo;
        /* locks currently being processed. */
        struct list_head cls_curr;
        /* locks acquired. */
        struct list_head cls_done;
};
struct cl_io_rw_common {
        loff_t pos;
        size_t count;
        int    nonblock;
        char  *buf;
};
```

State for io.

```
struct cl_io {
        enum cl_io_type      ci_type;
        /* main object this io is against */
        struct cl_object    *ci_obj;
        /* Upper layer io, of which this io is part of. */
        struct cl_io        *ci_parent;
        /* list of locks (to be) acquired by this io. */
        struct cl_lockset    ci_lockset;
        union {
                struct cl_rd_io {
                        struct cl_io_rw_common rd;
                        struct cl_page_list    rd_queue;
                        struct cl_page_list    rd_sent;
                } ci_rd;
                struct cl_wr_io {
                        struct cl_io_rw_common wr;
                        int                    wr_append;
                        struct cl_page_list    wr_queue;
                        struct cl_page_list    wr_sent;
                } ci_wr;
```

```
                struct cl_io_rw_common ci_rw;
                struct cl_truncate_io {
                        size_t      tr_size;
                } ci_truncate;
                struct cl_fault_io {
                        /* page index within cl_io::ci_fd */
                        pgoff_t         ft_index;
                        /* writable page? */
                        int             ft_writable;
                        /* page of an executable? */
                        int             ft_executable;
                        /* resulting page */
                        struct cl_page *ft_page;
                        /* \todo XXX Linux-specific for now. */
                        struct vm_area_struct *ft_vma;
                        unsigned long          ft_address;
                        int                   *ft_type;
                } ci_fault;
        } u;
        int                ci_result;
        loff_t             ci_size;
        size_t             ci_nob;
        int                ci_continue;
        struct ll_file_data *ci_fd;
};
```

7.5.2 entry points

Initialize \a io, by calling cl_io_operations::cio_init() top-to-bottom.
\pre obj == cl_object_top(obj)

```
int   cl_io_init         (const struct lu_env *env, struct cl_io *io,
                          enum cl_io_type iot, struct cl_object *obj);
```

Initialize sub-io, by calling cl_io_operations::cio_init() top-to-bottom.
\pre obj != cl_object_top(obj)

```
int   cl_io_sub_init     (const struct lu_env *env, struct cl_io *io,
                          enum cl_io_type iot, struct cl_object *obj);
```

Initialize read or write io.
\pre iot == CIT_READ || iot == CIT_WRITE

```
int   cl_io_rw_init      (const struct lu_env *env, struct cl_io *io,
                          enum cl_io_type iot, loff_t pos,
                          char *buf, size_t count);
```

Main io loop. Pumps io through iterations calling

- cl_io_prep()
- cl_io_lock()

- cl_io_start()
- cl_io_end()
- cl_io_unlock()

repeatedly until there is no more io to do.

```
int   cl_io_loop         (struct lu_env *env, struct cl_io *io);
```

Finalize \a io, by calling cl_io_operations::cio_fini() bottom-to-top.

```
void  cl_io_fini         (const struct lu_env *env, struct cl_io *io);
```

Prepares next iteration of io. Calls cl_io_operations::cio_prep() top-to-bottom. This exists to give layers a chance to modify io parameters, e.g., so that lov can restrict io to a single stripe.

```
void  cl_io_prep         (const struct lu_env *env, struct cl_io *io);
```

Takes locks necessary for the current iteration of io. Calls cl_io_operations::cio_lock() top-to-bottom to collect locks required by layers for the current iteration. Then sort locks (to avoid dead-locks), and acquire them.

```
int   cl_io_lock         (const struct lu_env *env, struct cl_io *io);
```

Release locks takes by io.

```
void  cl_io_unlock       (const struct lu_env *env, struct cl_io *io);
```

Starts io by calling cl_io_operations::cio_start() top-to-bottom.

```
int   cl_io_start        (const struct lu_env *env, struct cl_io *io);
```

Wait until current io iteration is finished by calling cl_io_operations::cio_end() bottom-to-top.

```
void  cl_io_end          (const struct lu_env *env, struct cl_io *io);
```

Adds a lock to a lockset.

```
int   cl_io_lock_add     (const struct lu_env *env, struct cl_io *io,
                          struct cl_io_lock_link *link);
```

Allocates new lock link, and uses it to add a lock to a lockset.

```
int   cl_io_lock_alloc_add(const struct lu_env *env, struct cl_io *io,
                           struct cl_lock *lock);
```

Called by read io, when page has to be read from the server.

```
int   cl_io_read_page    (const struct lu_env *env, struct cl_io *io,
                          struct cl_page *page);
```

Called by write io to prepare page to receive data from user buffer.

```
int   cl_io_prepare_write(const struct lu_env *env, struct cl_io *io,
                          struct cl_page *page, unsigned from, unsigned to);
```

Called by write io after user data were copied into a page.

```
int   cl_io_commit_write (const struct lu_env *env, struct cl_io *io,
                          struct cl_page *page, unsigned from, unsigned to);
```

Submits a list of pages for immediate io.

\returns 0 if at least one page was submitted, error code otherwise.

```
int   cl_io_submit_rw    (const struct lu_env *env, struct cl_io *io,
                          enum cl_req_type iot, struct cl_page_list *qin,
                          struct cl_page_list *qout);
```

Records that read or write io progressed \a nob bytes forward.

```
void  cl_io_rw_advance   (struct cl_io *io, size_t nob);
```

Returns top-level io.

```
struct cl_io *cl_io_top(struct cl_io *io);
```

Prints human readable representation of \a io to the \a f.

```
void cl_io_print(const struct lu_env *env, void *cookie,
                 lu_printer_t printer, const struct cl_io *io);
```

7.6 cl_page_list

7.6.1 data-structures

Page list used to perform collective operations on a group of pages.

Pages are added to the list one by one. cl_page_list acquires a reference for every page in it. Page list is used to perform collective operations on pages:

- submit pages for an immediate transfer,
- own pages on behalf of certain io (waiting for each page in turn),
- discard pages.

When list is finalized, it releases references on all pages it still has.

```
struct cl_page_list {
        unsigned         pl_nr;
        struct list_head pl_pages;
};
```

7.6.2 entry points

Iterate over pages in a page list.

```
#define cl_page_list_for_each(page, list)                               \
        list_for_each_entry((page), &(list)->pl_pages, cp_batch)
```

Iterate over pages in a page list, taking possible removals into account.

```
#define cl_page_list_for_each_safe(page, temp, list)                    \
        list_for_each_entry_safe((page), (temp), &(list)->pl_pages, cp_batch)
```

Initializes page list.

```
void cl_page_list_init   (struct cl_page_list *plist);
```

Adds a page to a page list.

```
void cl_page_list_add    (struct cl_page_list *plist, struct cl_page *page);
```

Moves a page from one page list to another.

```
void cl_page_list_move   (struct cl_page_list *dst, struct cl_page_list *src,
                          struct cl_page *page);
```

Removes a page from a page list.

```
void cl_page_list_del    (const struct lu_env *env,
                          struct cl_page_list *plist, struct cl_page *page);
```

Disowns pages in a queue.

```
void cl_page_list_disown (const struct lu_env *env,
                          struct cl_io *io, struct cl_page_list *plist);
```

Owns all pages in a queue.

```
int  cl_page_list_own    (const struct lu_env *env,
                          struct cl_io *io, struct cl_page_list *plist);
```

Discards all pages in a queue.

```
void cl_page_list_discard(const struct lu_env *env,
                          struct cl_io *io, struct cl_page_list *plist);
```

Unmaps all pages in a queue from user virtual memory.

```
int  cl_page_list_unmap  (const struct lu_env *env,
                          struct cl_io *io, struct cl_page_list *plist);
```

Releases pages from queue.

```
void cl_page_list_fini   (const struct lu_env *env, struct cl_page_list *plist);
```

7.7 cl_req

7.7.1 data-structures

Requested transfer type.

```
enum cl_req_type {
        CRT_READ,
        CRT_WRITE,
        CRT_NR
};
struct cl_req_operations {
        int  (*cro_prep)(const struct lu_env *env, struct cl_req_slice *slice);
        void (*cro_completion)(const struct lu_env *env,
                               struct cl_req_slice *slice, int ioret);
};
```

Transfer request.

Transfer requests are not reference counted, because IO sub-system owns them exclusively and knows when to free them.

Life cycle.

cl_req is created by cl_req_alloc() that calls cl_device_operations::cdo_req_init() device methods to allocate per-req state in every layer. Then pages are added (cl_req_page_add()), req keeps track of all objects it contains pages for.

Once all pages were collected, cl_page_operations::cpo_prep() method is called top-to-bottom. At that point layers can modify req, let it pass, or deny it completely. This is to support things like SNS that have transfer ordering requirements invisible to the individual req-formation engine.

On transfer completion (or transfer timeout, or failure to initiate the transfer of an allocated req), cl_req_operations::cro_completion() method is called, after execution of cl_page_operations::cpo_completion() of all req's pages.

```
struct cl_req {
        enum cl_req_type    crq_type;
        struct cl_req      *crq_parent;
        struct cl_req      *crq_child;
        /* A list of pages being transfered */
        struct list_head    crq_pages;
        /* Number of pages in cl_req::crq_pages */
        unsigned            crq_nrpages;
        /* An array of objects which pages are in ->crq_pages */
        struct cl_object  **crq_objs;
        /* Number of objects in cl_req::crq_objs[] */
        unsigned            crq_nrobjs;
        struct list_head    crq_layers;
};
```

Per-layer state for request.

```
struct cl_req_slice {
        struct cl_req    *crs_req;
        struct cl_device *crs_dev;
        struct list_head  crs_linkage;
        const struct cl_req_operations *crs_ops;
};
```

7.7.2 entry points

Allocates new transfer request.

```
struct cl_req *cl_req_alloc(const struct lu_env *env, struct cl_page *page,
                            enum cl_req_type crt, int nr_objects);
```

Adds a page to a request.

```
void cl_req_page_add  (const struct lu_env *env, struct cl_req *req,
                       struct cl_page *page);
```

Removes a page from a request.

```
void cl_req_page_done (const struct lu_env *env, struct cl_page *page);
```

Notifies layers that request is about to depart by calling cl_req_operations::cro_prep() top-to-bottom.

```
int  cl_req_prep      (const struct lu_env *env, struct cl_req *req);
```

Invokes per-request transfer completion call-backs (cl_req_operations::cro_completion()) bottom-to-top.

```
void cl_req_completion(const struct lu_env *env, struct cl_req *req, int ioret);
```

8 References

- [0] MD API DLD (md-api-dld.lyx)
- [1] Client IO stack layering cleanup HLD (client-io-layering)