MD API

Nikita Danilov <nikita@clusterfs.com>

2006.04.28

1 Introduction

As part of CMD3 development, structure of md server code layering was changed significantly: cmm layer was added to encapsulate clustered meta-data logic, md services were changed to run on top of osd, networking and dlm locking were limited to mdt. It was decided that this is good moment to cleanup code interfaces related to layering.

2 Scope

2.1 What is in the scope of this document.

This specification describes overall structure of server side (and potentially client side) code layering. Specifically, it introduces mechanisms that are used to separate various layers from each other while providing enough communication flexibility without losing type-safety.

2.2 What is not in the scope of this document.

This specification doesn't describe functionality of specific layers in the server stack. Each layer has (or should have) special design document. Details of layer specific locking are omitted (they are subject matter of another design specification). Details of md and dt interface usage are omitted too.

3 Requirements

New interfaces were designed with the following goals in mind:

- *layering.* Devices are cleanly stackable on top of each other. Method invocations propagate through the stack. Objects (representing file system entities) are also stackable. That is, each layer in the device stack has a place to store its per-object data (a slice). Collection of such slices is a compound object having its own identity.

- *fid-based object identification.* Compound object is uniquely identified by fid. It's possible to efficiently search for an object with the given fid, and create one if missing.
- *re-use on clients and in dt stack.* Interfaces and data-structures were designed as to make their future use on the client and ost nodes possible.
- *object caching and life cycle management.* Objects caching and life cycle management (through the reference counting) are controlled by the generic code.
- *stack consumption.* Care has been taken to reduce stack consumption, which is especially important given deeper layering. To this end following techniques are employed:
  - where possible generic code uses iteration instead of recursion;
  - a convenient interface is provided (`lu_context` and `lu_context_key`) allowing clients to associate arbitrary data values with execution contexts (threads). This interface is used to allocate for each thread data that are frequently used by layered code and to avoid allocating it on the stack of the current thread.
- *locking.* Compound objects are entities on which locks are taken. One implication of this is that objects are sometimes created for entities not yet existing on the storage (e.g., on the new file that is going to be created).
- *well-structured interfaces* (no flat tables). Interfaces provided by objects and devices are split into multiple operations vectors. There are per-object, per-device, and per-device-type vectors. This provides flexibility necessary to partition interfaces logically (instead of dumping everything into obd_type as in the previous implementation). Also this implies that interfaces are type-safe where possible (no opaque void * cookies).

4 Glossary

md, md-stack, md-device meta-data, meta-data stack on mds node, a device implementing interfaces to work with meta-data.

dt, dt-stack, dt-device data, data stack on oss node, a device implementing interfaces to work with data.

fid an object identifier, unique across cluster.

compound object an object, representing file system entity (file, directory) and structured as a stack of object slices.

object slice, object layer a portion of compound object corresponding to the particular device in the device stack.

object a compound object or an object slice depending on context.

5 Functional Specification

5.1 Main entities

Main entities represented by md-api are: site, device type, device, object.

site (`struct lu_site`) site represents a particular instance of a device stack, running on a given node. Usually there is one site per server node.

device type (`struct lu_device_type`) device type represents a class of similar devices. Device type is responsible for creation of devices belonging to it. Existing device types are: mdt, cmm, mdd, osd.

device (`struct lu_device`) device is an instance of device type. Device belongs to particular site, and is located within device stack operating in this site. Device is responsible for creation of object slices for this device.

object (`struct lu_object_header`, `struct lu_object`) see glossary.

`lu_site` represents device stack. In addition it keeps track of all objects created for this stack, allows lookup of object by fid, maintains LRU cache of objects, and various statistics.

All other types of entities (device type, device, and object) have operation vectors associated with them. As relationships between these entities are rather involved, they are presented as a diagram below:

Greyed rectangles represent legacy data-types that are used for compatibility with existing obd-based code.

`lu_device_type` operations vector (`lu_device_type_operations`) has methods to initialize/finalize type itself (usually called on module load/unload), and methods to create/destroy devices of this type.

`lu_device` operations vector (`lu_device_operations`) has methods to create/destroy object slices for this device and to process cluster configuration during startup and configuration change.

Object slice is represented by `lu_object`. Object slices are stacked (according to device stack) to form compound object represented by `lu_object_header`. Each slice has its own operations vector (different object slices created for the same device can have different operations vectors): `lu_object_operations`.

`lu_object_header` is linked into site-wide hash table (keyed by its fid) and into site-wide LRU list used to implement cache replacement policy.

Compound object does not necessary represent any "real" file system object existing on the durable storage. Instead it conceptually acts like a place-holder for a given fid. Its usefulness stems from the ability to take locks on it. As a result it's possible to lock a slot in the fid namespace and to operate on this slot. To check for existence of real object `->loo_object_exists()` method (wrapped into `lu_object_exists()` helper function) can be used. By analogy with Linux dcache, compound object not backed by any real storage object is referred to as "negative object".

Different compound objects in the same site may be composed of different slices. Moreover, objects may refer to both local and remote entities. At the bottom of the stack of slices of compound object referring to the local storage, there usually is a slice associated with underlying file system object (inode), while at the bottom of the slice stack for remote object there is a slice dealing with networking.

5.2 Typical server configuration

While beyond the scope of this specification, typical server layering is described below to serve as an illustration and to help understanding of examples below.

Topmost layer of the stack is mdt (`mdt_device` and `mdt_object`). mdt layer is responsible for all networking-related functionality:

- receiving requests from the network, and their unpacking;
- recovery, resends, replays, and difficult-acks;
- network-related optimizations such as intents;
- dlm locking;
- etc.

Under mdt there is cmm layer (`cmm_device` and `cmm_object`) that deals with meta-data clustering. It determines what portions of request are to be handled locally and what remotely and forwards request processing appropriately. Other md clustering related things like cluster-wide rollback protocol are also handled at this layer.

At this point stack splits. For remote operations cmm layer uses underlying mdc devices (`mdc_device` and `mdc_object`). mdc layer acts as a proxy forwarding request over network. Layering for remote objects ends at this layer.

For local operations cmm uses mdd child (`mdd_device` and `mdd_object`). mdd is a layer implementing meta-data operations (like, unlink, rename, create, mkdir, etc.) on top of underlying dt device. mdd also handles per-object locking and file system transactions.

mdd stacks on top of osd (`osd_device` and `osd_object`) that implements dt interface. osd lives directly on top of the underlying file system. Layering for local objects ends here.

5.3 Stacking

The following diagram clarifies relationship between device stack, object slices, and compound objects:

This diagram presents typical md-stack consisting from (top to bottom): mdt (md target, receiving requests from the network), and cmm (managing clustering). Below cmm are mdc targets that communicate with remote md servers and mdd device that implements local md operations. mdd operates on

top of osd — object storage device that implements data operations (osd is used by both md and dt stacks). osd sits on top of local file system.

Accordingly there are two types of compound objects in this site: one for local objects (mdt->cmm->mdd->osd) and one for remote objects (mdt->cmm->mdc). Each compound object is represented by its header (`lu_object_header`) that is embedded into top-level mdt slice (this is am implementation detail). Header is linked into LRU and hash table, anchored in the site. From header hangs off a list of all slices (`->loh_layers`). Slices are represented by `lu_object`. Each device actually allocates some larger data-structure embedding `lu_object` as one of its fields. Device uses this structure to keep per-slice information for each object. In fact, devices do not embed `lu_object` directly. Instead they use either md_object (for md-devices) or dt_object (for dt-device). md_device and `dt_device` in turn include `lu_object`, plus feature some fields common for all md and dt devices respectively (currently only operation vectors).

5.4 Operation vectors

To achieve necessary flexibility and to avoid large flat operation vectors, new md-api design attaches operation vectors to the various entities (to all entities mentioned in 5.1 on page 3 except site). Generic code makes no assumptions about these tables. For example, it's valid for different devices belonging to the same device type to have different `->ld_ops` operation vectors and for different objects created by the same device to have different `->lo_ops` operation vectors.

In addition to that, "sub-classes" add their own operation vectors at both device and object. For example, md_object has additional md_object_operations and md_dir_operations vectors, and dt_object has dt_{object,body,index}_operations vectors. Diagram below clarifies operation vector arrangement.

Note: this area of design is not yet finalized: new methods and new vectors may be added in the future.

5.5 Hashing

Compound objects within site are indexed by their fid. This index is implemented as a hash table, hanging off `lu_site` (`->ls_hash`). Hash-table is protected by a single site-wide spin-lock and follows standard check-allocate-recheck insertion protocol (similar to `iget()`). Hash table guarantees that there is at most one object with the given fid within given site, and allows to check for existence of such object efficiently.

5.6 Caching

Site maintains a cache of recently accessed compound objects. This cache is used to amortize costs of object creation and tear-down (that may involve io).

Compound objects within site are kept in LRU list. Object is moved to the hot end of this list whenever it is found in the cache. New objects are also placed at the hot end of LRU. (Specifically, object is moved at the hot end of

the LRU list just before returning from `lu_object_find()`). LRU list is protected by a single site-wide spin-lock. When a thread acquired a reference to an object (this includes new objects), object is in "busy" state. It leaves this state, when last reference to it is released. At that moment object is usually returns back into cache (moving into "cached" state), unless `lu_object_is_dying()` predicate is true for it. User can assert `lu_object_is_dying()` by setting `LU_OBJECT_HEARD_BANSHEE` bit in `->loh_flags` field. Dying objects are not cached, instead they are freed immediately after release of the last reference.

Function `lu_site_purge()` frees given number of objects from the cold end of the LRU list. This function should be called from VM call-backs to keep cache size under control. Transitions between cache-related states of `lu_object` are described in the following diagram:

5.7 Execution context and context key

To reach the design goal of reducing stack consumption, a notion of "execution context", represented by `lu_context` data-type was introduced. Execution context is an opaque data-structure passed to (almost) all lu methods. It is guaranteed that no two concurrently running lu methods are executed within the same context. That is to roughly say, there is an execution context per-thread. The usefulness of execution context stems from `lu_context_key` interface. This interface allows one to associate an arbitrary piece of data with given execution context. It is supposed to be used by `lu_device` implementations to pre-allocate in each context various data that otherwise would have been allocated from heap of on stack. `lu_context_key` interface is similar in spirit to pthread_key interface (see pthread_key_create(3)). In effect, `lu_context_key` provides some sort of thread-local storage, that maybe used without caring about concurrency issues.

Users should not assume that execution context (and, hence, key values) remain the same from invocation to invocation even for invocations running in the same thread, because:

- in the future implementation may decide to migrate request servicing from one thread to another;
- there are contexts not associated with any thread. Currently these are created during system initialization and configuration. In the future, more non-thread contexts might appear.

That is, key values are only valid during single method invocation. Hence, their semantics is close to local automatic variables (which they are purporting to replace).

Additional advantage of execution context is that it serves as an intermediary step of switching to asynchronous model of execution.

Interface:

```
/*
```

```
 * lu_context. Execution context for lu_object methods. Currently associated
 * with thread.
 *
 * All lu_object methods, except device and device type methods (called during
 * system initialization and shutdown) are executed "within" some
 * lu_context. This means, that pointer to some "current" lu_context is passed
 * as an argument to all methods.
 *
 * All service ptlrpc threads create lu_context as part of their
 * initialization. It is possible to create "stand-alone" context for other
 * execution environments (like system calls).
 *
 * lu_object methods mainly use lu_context through lu_context_key interface
 * that allows each layer to associate arbitrary pieces of data with each
 * context (see pthread_key_create(3) for similar interface).
 *
 */
struct lu_context {
        /*
         * Theoretically we'd want to use lu_objects and lu_contexts on the
         * client side too. On the other hand, we don't want to allocate
         * values of server-side keys for the client contexts and vice versa.
         *
         * To achieve this, set of tags in introduced. Contexts and keys are
         * marked with tags. Key value are created only for context whose set
         * of tags has non-empty intersection with one for key. NOT YET
         * IMPLEMENTED.
         */
        __u32                   lc_tags;
        /*
         * Pointer to the home service thread. NULL for other execution
         * contexts.
         */
        struct ptlrpc_thread  *lc_thread;
        /*
         * Pointer to an array with key values. Internal implementation
         * detail.
         */
        void                 **lc_value;
};
/*
 * lu_context_key interface. Similar to pthread_key.
 */
/*
 * Key. Represents per-context value slot.
 */
```

```
struct lu_context_key {
        /*
         * Value constructor. This is called when new value is created for a
         * context. Returns pointer to new value of error pointer.
         */
        void  *(*lct_init)(struct lu_context *ctx);
        /*
         * Value destructor. Called when context with previously allocated
         * value of this slot is destroyed. @data is a value that was returned
         * by a matching call to ->lct_init().
         */
        void   (*lct_fini)(struct lu_context *ctx, void *data);
        /*
         * Internal implementation detail: index within ->lc_value[] reserved
         * for this key.
         */
        int      lct_index;
        /*
         * Internal implementation detail: number of values created for this
         * key.
         */
        unsigned lct_used;
};
/*
 * Register new key.
 */
int   lu_context_key_register(struct lu_context_key *key);
/*
 * Deregister key.
 */
void  lu_context_key_degister(struct lu_context_key *key);
/*
 * Return value associated with key @key in context @ctx.
 */
void *lu_context_key_get(struct lu_context *ctx, struct lu_context_key *key);
/*
 * Initialize context data-structure. Create values for all keys.
 */
int  lu_context_init(struct lu_context *ctx);
/*
 * Finalize context data-structure. Destroy key values.
 */
void lu_context_fini(struct lu_context *ctx);
/*
 * Called before entering context.
 */
```

```
void lu_context_enter(struct lu_context *ctx);
/*
 * Called after exiting from @ctx
 */
void lu_context_exit(struct lu_context *ctx);
```

5.8 Generic code

Generic `lu_object` code performs basic operations on `lu_site`, `lu_device_type`, `lu_device`, and `lu_object`.

5.8.1 Constructors/destructors

```
/*
 * Initialize site @s, with @d as the top level device.
 */
int  lu_site_init(struct lu_site *s, struct lu_device *d);
/*
 * Finalize @s and release its resources.
 */
void lu_site_fini(struct lu_site *s);
/*
 * Acquire additional reference on device @d
 */
void lu_device_get(struct lu_device *d);
/*
 * Release reference on device @d.
 */
void lu_device_put(struct lu_device *d);
/*
 * Initialize device @d of type @t.
 */
int lu_device_init(struct lu_device *d, struct lu_device_type *t);
/*
 * Finalize device @d.
 */
void lu_device_fini(struct lu_device *d);
/*
 * Initialize compound object.
 */
int lu_object_header_init(struct lu_object_header *h);
/*
 * Finalize compound object.
 */
void lu_object_header_fini(struct lu_object_header *h);
/*
```

```
 * Initialize object @o that is part of compound object @h and was created by
 * device @d.
 */
int lu_object_init(struct lu_object *o,
                   struct lu_object_header *h, struct lu_device *d);
/*
 * Finalize object and release its resources.
 */
void lu_object_fini(struct lu_object *o);
/*
 * Add object @o as first layer of compound object @h.
 *
 * This is typically called by the ->ldo_object_alloc() method of top-level
 * device.
 */
void lu_object_add_top(struct lu_object_header *h, struct lu_object *o);
/*
 * Add object @o as a layer of compound object, going after @before.1
 *
 * This is typically called by the ->ldo_object_alloc() method of
 * @before->lo_dev.
 */
void lu_object_add(struct lu_object *before, struct lu_object *o);
```

5.8.2 Caching and reference counting.

```
/*
 * Acquire additional reference to the given object. This function is used to
 * attain additional reference. To acquire initial reference use
 * lu_object_find().
 */
void lu_object_get(struct lu_object *o);
/*
 * Return true of object will not be cached after last reference to it is
 * released.
 */
int lu_object_is_dying(struct lu_object_header *h);
/*
 * Decrease reference counter on object. If last reference is freed, return
 * object to the cache, unless lu_object_is_dying(o) holds. In the latter
 * case, free object immediately.
 */
void lu_object_put(struct lu_context *ctxt, struct lu_object *o);
/*
 * Free @nr objects from the cold end of the site LRU list.
 */
```

```
void lu_site_purge(struct lu_context *ctx, struct lu_site *s, int nr);
/*
 * Search cache for an object with the fid @f. If such object is found, return
 * it. Otherwise, create new object, insert it into cache and return it. In
 * any case, additional reference is acquired on the returned object.
 */
struct lu_object *lu_object_find(struct lu_context *ctxt,
                                 struct lu_site *s, const struct lu_fid *f);
```

5.8.3 Helpers.

```
/*
 * First (topmost) sub-object of given compound object
 */
struct lu_object *lu_object_top(struct lu_object_header *h);
/*
 * Next sub-object in the layering
 */
struct lu_object *lu_object_next(const struct lu_object *o);
/*
 * Pointer to the fid of this object.
 */
const struct lu_fid *lu_object_fid(const struct lu_object *o);
/*
 * return device operations vector for this object
 */
struct lu_device_operations *lu_object_ops(const struct lu_object *o);
/*
 * Given a compound object, find its slice, corresponding to the device type
 * @dtype.
 */
struct lu_object *lu_object_locate(struct lu_object_header *h,
                                   struct lu_device_type *dtype);
/*
 * Print human readable representation of the @o to the @f.
 */
int lu_object_print(struct lu_context *ctxt,
                    struct seq_file *f, const struct lu_object *o);
/*
 * Returns true iff object @o exists on the stable storage.
 */
static inline int lu_object_exists(struct lu_context *ctx, struct lu_object *o);
```

6 Use Cases

6.1 Device method invocation

Handling of `MDS_GETSTATUS` request (returning root fid) is implemented as a call to `->mdo_root_get()` method from `md_device_operations`:

```
static int mdt_getstatus(struct mdt_thread_info *info)
{
        struct md_device *next  = info->mti_mdt->mdt_child;
        return next->md_ops->mdo_root_get(info->mti_ctxt,
                                          next, &info->mti_body->fid1);
}
```

Top-level device of the stack (mdt), invokes `->mdo_root_get()` method of its child device (`->mdt_child`), passing to it current context, and a pointer where fid is to be stored.

Child device is usually cmm:

```
int cmm_root_get(struct lu_context *ctx,
                 struct md_device *md, struct lu_fid *fid)
{
        struct cmm_device *cmm_dev = md2cmm_dev(md);
        return cmm_child_ops(cmm_dev)->mdo_root_get(ctx,
                                                    cmm_dev->cmm_child, fid);
}
```

cmm obtains from upper layer (mdt) a pointer to `md_device`, and it knows that this pointer points to the `md_device` portion of `cmm_device`. cmm casts this pointer to obtain `cmm_device` and forwards `->mdo_root_get()` call down through the stack to its child (`->cmm_child`).

cmm child is usually mdd:

```
static int mdd_root_get(struct lu_context *ctx,
                        struct md_device *m, struct lu_fid *f)
{
        struct mdd_device *mdd = lu2mdd_dev(&m->md_lu_dev);
        return mdd_child_ops(mdd)->dt_root_get(ctx, mdd->mdd_child, f);
}
```

`mdd_root_get()` is similar to `cmm_root_get()`: it also casts received pointer to `md_device` to obtain `mdd_device` and forwards method down. The difference is that mdd_device sits on top of `dt_device`, so `->mdo_root_get()` call transforms into `->dt_root_get()` call.

`dt_device` below `mdd_device` is usually osd:

```
static int osd_root_get(struct lu_context *ctx,
                        struct dt_device *dev, struct lu_fid *f)
```

```
{
        osd_inode_get_fid(osd_dt_dev(dev)->od_root_dir->d_inode, f);
        return 0;
}
```

osd is bottom device and `osd_root_get()` actually implements the functionality, by returning fid associated with the root inode.

6.2 Object method invocation

Handling of `MDS_GETATTR` request is implemented through invocation of `->moo_attr_get()` method from `md_object_operations` vector.

As in the previous case, handling starts in the top-level device (mdt):

```
static int mdt_getattr(struct mdt_thread_info *info, struct ptlrpc_request *req)
{
        int                result;
        struct mdt_object *obj;
        struct md_object  *next;
        obj = mdt_object_find(info->mti_ctxt, info->mti_mdt, req_fid(req));
        if (!lu_object_exists(info->mti_ctxt, &obj->mot_obj.md_lu))
                return -ENOENT;
        next   = mdt_object_child(obj);
        result = next->mo_ops->moo_attr_get(info->mti_ctxt, next, &info->mti_attr);
        mdt_pack_attr2body(info->mti_body, &info->mti_attr);
        return result;
}
```

First, `mdt_getattr()` finds object corresponding to fid received in request (`mdt_object_find()`) by calling `lu_object_find()` function. Then, after checking that object exists, `->moo_attr_get()` method of child object is called. This method fills out `lu_attr` structure. Ultimately, `mdt_getattr()` packs obtained `lu_attr` into `mdt_body` that will be shipped back to the client.

Child of mdt object is usually `cmm_object`:

```
int cmm_attr_get(struct lu_context *ctxt, struct md_object *obj,
                 struct lu_attr *attr)
{
        struct md_object *next = cmm2child_obj(md2cmm_obj(obj));
        return next->mo_ops->moo_attr_get(ctxt, next, attr);
}
```

Method invocation is delegated through the stack much like in the previous use case, until it reaches bottom object slice (usually `osd_object`):

```
static int osd_attr_get(struct lu_context *ctxt, struct dt_object *dt,
                        struct lu_attr *attr)
{
        LASSERT(lu_object_exists(ctxt, &dt->do_lu));
        return osd_inode_getattr(ctxt, dt2osd_obj(dt)->oo_inode, attr);
}
```

`osd_attr_get()` implements real functionality, by taking object attributes from inode and packing them into `lu_attr`.

6.3 lu_context_key usage

mdt layer defines structure mdt_thread_info:

```
/*
 * Common data shared by mdt-level handlers. This is allocated per-thread to
 * reduce stack consumption.
 */
struct mdt_thread_info {
        struct lu_context     *mti_ctxt;
        struct mdt_device     *mti_mdt;
        /*
         * number of buffers in reply message.
         */
        int                    mti_rep_buf_nr;
        /*
         * sizes of reply buffers.
         */
        int                    mti_rep_buf_size[MDT_REP_BUF_NR_MAX];
        /*
         * Body for "habeo corpus" operations.
         */
        struct mdt_body       *mti_body;
        /*
         * Lock request for "habeo clavis" operations.
         */
        struct ldlm_request   *mti_dlm_req;
        /*
         * Host object. This is released at the end of mdt_handler().
         */
        struct mdt_object     *mti_object;
        /*
         * Object attributes.
         */
        struct lu_attr         mti_attr;
```

```
        /*
         * reint record. Containing information for reint operations.
         */
        struct mdt_reint_record mti_rr;
        /*
         * Additional fail id that can be set by handler. Passed to
         * target_send_reply().
         */
        int                     mti_fail_id;
        /*
         * A couple of lock handles.
         */
        struct mdt_lock_handle mti_lh[MDT_LH_NR];
};
```

It contains fields for data-items frequently used by mdt methods: sizes of buffers in the reply message, lock handles, pointer to mdt_body passed together with request, etc.

During device type initialization, mdt module registers `lu_context_key`:

```
static int mdt_type_init(struct lu_device_type *t)
{
        return lu_context_key_register(&mdt_thread_key);
}
static struct lu_context_key mdt_thread_key = {
        .lct_init = mdt_thread_init,
        .lct_fini = mdt_thread_fini
};
static void mdt_device_free(struct lu_device *d)
{
        struct mdt_device *m = mdt_dev(d);
        mdt_fini(m);
        OBD_FREE_PTR(m);
}
static void *mdt_thread_init(struct lu_context *ctx)
{
        struct mdt_thread_info *info;
        OBD_ALLOC_PTR(info);
        if (info != NULL)
                info->mti_ctxt = ctx;
        else
                info = ERR_PTR(-ENOMEM);
        return info;
}
```

Key constructor (`->lct_init()`, `mdt_thread_init()` in this case) is called whenever new execution context is created, in particular, when new thread

is started. Key destructor (->lct_fini(), mdt_thread_fini() in this case) is called whenever execution context is torn down.

To access data allocated by constructor, lu_context_key_get() is used:

```
static int mdt_handle(struct ptlrpc_request *req)
{
        struct lu_context      *ctx;
        struct mdt_thread_info *info;
        ctx = req->rq_svc_thread->t_ctx;
        LASSERT(ctx != NULL);
        LASSERT(ctx->lc_thread == req->rq_svc_thread);
        info = lu_context_key_get(ctx, &mdt_thread_key);
        LASSERT(info != NULL);
        ...
}
```

6.4 lu_object_alloc() invocation

lu_object_alloc() is invoked as a part of object creation by lu_object_find() function. Due to desire to avoid recursive method calls, lu_object_alloc() follows rather involved protocol, and is also instructive to understand various bits of object architecture. Source code for this function is listed below (7 on page 18) in Logic Specification section.

Overall idea is to split allocation of every layer into two steps:

- allocation proper that only allocates data object for this layer;
- initialization that allocates child layers.

Allocation proper is implemented in ->ldo_object_alloc() method, and initialization in ->loo_object_init() method. Generic code calls ->loo_object_init() that is supposed to call ->ldo_object_alloc() of the underlying layers. After that control returns to the generic code that calls ->loo_object_init() method of newly allocated slices and so on until whole compound object is built.

Let's look step-by-step at typical lu_object_alloc() execution.

As the first step, ->ldo_object_alloc() method of the top-level device is called. Let this be mdt_object_alloc(). ->ldo_object_alloc() of the top level device is somewhat of exception, because it has to allocate both header (lu_object_header) and object slice (mdt_object). After linking just allocated slice into layering list of the header, and initializing few operation vector pointers this method returns.

Once object header is allocated, lu_object_alloc() sets compound object fid to the supplied value. After this it enters alloc/init loop described above.

At this point, layering list contains single slice (mdt_object) so its ->loo_object_init() method is called (mdt_object_init()). mdt_object_init() calls ->ldo_object_alloc() of the underlying device (cmm_object_alloc()) and inserts resulting slice into

layering list. cmm_object_alloc() allocates its object slice (cmm_object) and immediately returns.

Now control is back to lu_object_alloc() and ->loo_object_init() of the newly allocated cmm object is called (cmm_object_init()).

cmm_object_init() is more complicated than for other layers, because at this point we have to decide whether object is local or remote. To this end fld query is made. In any case, cmm_object_init() finds underlying device for this object. This can be either mdc_device (for remote object) or mdd_device (for local object). cmm_object_init() calls ->ldo_object_alloc() of this device and returns back to lu_object_alloc().

This loop continues until no new objects are allocated. This happens when bottom slice (mdc_object or osd_object) is allocated.

It should be noted, that ->loo_object_init() is free to allocate more than one object on the underlying layer, or even allocate slices belonging to the different devices.

7 Logic Specification

Logic specification for some generic code:

```
/*
 * Decrease reference counter on object. If last reference is freed, return
 * object to the cache, unless lu_object_is_dying(o) holds. In the latter
 * case, free object immediately.
 */
void lu_object_put(struct lu_context *ctxt, struct lu_object *o)
{
        struct lu_object_header *top;
        struct lu_site          *site;
        top = o->lo_header;
        site = o->lo_dev->ld_site;
        spin_lock(&site->ls_guard);
        if (-- top->loh_ref == 0) {
                /*
                 * When last reference is released, iterate over object
                 * layers, and notify them that object is no longer busy.
                 */
                list_for_each_entry(o, &top->loh_layers, lo_linkage) {
                        if (o->lo_ops->loo_object_release != NULL)
                                o->lo_ops->loo_object_release(ctxt, o);
                }
                -- site->ls_busy;
                if (lu_object_is_dying(top)) {
                        /*
                         * If object is dying (will not be cached), remove it
```

```
                         * from hash table and LRU.
                         *
                         * This is done with hash table and LRU lists
                         * locked. As the only way to acquire first reference
                         * to previously unreferenced object is through
                         * hash-table lookup (lu_object_find()), or LRU
                         * scanning (lu_site_purge()), that are done under
                         * hash-table and LRU lock, no race with concurrent
                         * object lookup is possible and we can safely destroy
                         * object below.
                         */
                        hlist_del_init(&top->loh_hash);
                        list_del_init(&top->loh_lru);
                }
        }
        spin_unlock(&site->ls_guard);
        if (lu_object_is_dying(top))
                /*
                 * Object was already removed from hash and lru above, can
                 * kill it.
                 */
                lu_object_free(ctxt, o);
}
/*
 * Allocate new object.
 *
 * This follows object creation protocol, described in the comment within
 * struct lu_device_operations definition.
 */
struct lu_object *lu_object_alloc(struct lu_context *ctxt,
                                  struct lu_site *s, const struct lu_fid *f)
{
        struct lu_object *scan;
        struct lu_object *top;
        int clean;
        int result;
        /*
         * Create top-level object slice. This will also create
         * lu_object_header.
         */
        top = s->ls_top_dev->ld_ops->ldo_object_alloc(ctxt, s->ls_top_dev);
        if (IS_ERR(top))
                RETURN(top);
        /*
         * This is the only place where object fid is assigned. It's constant
         * after this point.
```

```
         */
        o->lo_header->loh_fid = *fid;
        do {
                /*
                 * Call ->loo_object_init() repeatedly, until no more new
                 * object slices are created.
                 */
                clean = 1;
                list_for_each_entry(scan,
                                    &top->lo_header->loh_layers, lo_linkage) {
                        if (scan->lo_flags & LU_OBJECT_ALLOCATED)
                                continue;
                        clean = 0;
                        scan->lo_header = top->lo_header;
                        result = scan->lo_ops->loo_object_init(ctxt, scan);
                        if (result != 0) {
                                lu_object_free(ctxt, top);
                                RETURN(ERR_PTR(result));
                        }
                        scan->lo_flags |= LU_OBJECT_ALLOCATED;
                }
        } while (!clean);
        s->ls_stats.s_created ++;
        RETURN(top);
}
/*
 * Free object.
 */
static void lu_object_free(struct lu_context *ctx, struct lu_object *o)
{
        struct list_head splice;
        struct lu_object *scan;
        /*
         * First call ->loo_object_delete() method to release all resources.
         */
        list_for_each_entry_reverse(scan,
                                    &o->lo_header->loh_layers, lo_linkage) {
                if (scan->lo_ops->loo_object_delete != NULL)
                        scan->lo_ops->loo_object_delete(ctx, scan);
        }
        -- o->lo_dev->ld_site->ls_total;
        /*
         * Then, splice object layers into stand-alone list, and call
         * ->ldo_object_free() on all layers to free memory. Splice is
         * necessary, because lu_object_header is freed together with the
         * top-level slice.
```

```
         */
        INIT_LIST_HEAD(&splice);
        list_splice_init(&o->lo_header->loh_layers, &splice);
        while (!list_empty(&splice)) {
                o = container_of0(splice.next, struct lu_object, lo_linkage);
                list_del_init(&o->lo_linkage);
                LASSERT(lu_object_ops(o)->ldo_object_free != NULL);
                lu_object_ops(o)->ldo_object_free(ctx, o);
        }
}
/*
 * Free @nr objects from the cold end of the site LRU list.
 */
void lu_site_purge(struct lu_context *ctx, struct lu_site *s, int nr)
{
        struct list_head         dispose;
        struct lu_object_header *h;
        struct lu_object_header *temp;
        INIT_LIST_HEAD(&dispose);
        /*
         * Under LRU list lock, scan LRU list and move unreferenced objects to
         * the dispose list, removing them from LRU and hash table.
         */
        spin_lock(&s->ls_guard);
        list_for_each_entry_safe(h, temp, &s->ls_lru, loh_lru) {
                if (nr-- == 0)
                        break;
                if (h->loh_ref > 0)
                        continue;
                hlist_del_init(&h->loh_hash);
                list_move(&h->loh_lru, &dispose);
        }
        spin_unlock(&s->ls_guard);
        /*
         * Free everything on the dispose list. This is safe against races due
         * to the reasons described in lu_object_put().
         */
        while (!list_empty(&dispose)) {
                h = container_of0(dispose.next,
                                  struct lu_object_header, loh_lru);
                list_del_init(&h->loh_lru);
                lu_object_free(ctx, lu_object_top(h));
                s->ls_stats.s_lru_purged ++;
        }
}
/*
```

```
 * Search cache for an object with the fid @f. If such object is found, return
 * it. Otherwise, create new object, insert it into cache and return it. In
 * any case, additional reference is acquired on the returned object.
 */
struct lu_object *lu_object_find(struct lu_context *ctxt, struct lu_site *s,
                                 const struct lu_fid *f)
{
        struct lu_object  *o;
        struct lu_object  *shadow;
        struct hlist_head *bucket;
        /*
         * This uses standard index maintenance protocol:
         *
         *     - search index under lock, and return object if found;
         *     - otherwise, unlock index, allocate new object;
         *     - lock index and search again;
         *     - if nothing is found (usual case), insert newly created
         *       object into index;
         *     - otherwise (race: other thread inserted object), free
         *       object just allocated.
         *     - unlock index;
         *     - return object.
         */
        bucket = s->ls_hash + (fid_hash(f) & s->ls_hash_mask);
        spin_lock(&s->ls_guard);
        o = htable_lookup(s, bucket, f);
        spin_unlock(&s->ls_guard);
        if (o != NULL)
                return o;
        /*
         * Allocate new object. This may result in rather complicated
         * operations, including fld queries, inode loading, etc.
         */
        o = lu_object_alloc(ctxt, s, f);
        if (IS_ERR(o))
                return o;
        ++ s->ls_total;
        LASSERT(lu_fid_eq(lu_object_fid(o), f));
        spin_lock(&s->ls_guard);
        shadow = htable_lookup(s, bucket, f);
        if (shadow == NULL) {
                hlist_add_head(&o->lo_header->loh_hash, bucket);
                list_add_tail(&s->ls_lru, &o->lo_header->loh_lru);
                shadow = o;
                o = NULL;
        } else
```

```
                s->ls_stats.s_cache_race ++;
        spin_unlock(&s->ls_guard);
        if (o != NULL)
                lu_object_free(ctxt, o);
        return shadow;
}
/*
 * Register new key.
 */
int lu_context_key_register(struct lu_context_key *key)
{
        int result;
        int i;
        result = -ENFILE;
        spin_lock(&lu_keys_guard);
        for (i = 0; i < ARRAY_SIZE(lu_keys); ++i) {
                if (lu_keys[i] == NULL) {
                        key->lct_index = i;
                        key->lct_used = 1;
                        lu_keys[i] = key;
                        result = 0;
                        break;
                }
        }
        spin_unlock(&lu_keys_guard);
        return result;
}
/*
 * Deregister key.
 */
void lu_context_key_degister(struct lu_context_key *key)
{
        LASSERT(key->lct_used >= 1);
        LASSERT(0 <= key->lct_index && key->lct_index < ARRAY_SIZE(lu_keys));
        if (key->lct_used > 1)
                CERROR("key has instances.\n");
        spin_lock(&lu_keys_guard);
        lu_keys[key->lct_index] = NULL;
        spin_unlock(&lu_keys_guard);
}
/*
 * Return value associated with key @key in context @ctx.
 */
void *lu_context_key_get(struct lu_context *ctx, struct lu_context_key *key)
{
        LASSERT(0 <= key->lct_index && key->lct_index < ARRAY_SIZE(lu_keys));
```

```
        return ctx->lc_value[key->lct_index];
}
static void keys_fini(struct lu_context *ctx)
{
        int i;
        if (ctx->lc_value != NULL) {
                for (i = 0; i < ARRAY_SIZE(lu_keys); ++i) {
                        if (ctx->lc_value[i] != NULL) {
                                struct lu_context_key *key;
                                key = lu_keys[i];
                                LASSERT(key != NULL);
                                LASSERT(key->lct_fini != NULL);
                                LASSERT(key->lct_used > 1);
                                key->lct_fini(ctx, ctx->lc_value[i]);
                                key->lct_used--;
                                ctx->lc_value[i] = NULL;
                        }
                }
                OBD_FREE(ctx->lc_value,
                         ARRAY_SIZE(lu_keys) * sizeof ctx->lc_value[0]);
                ctx->lc_value = NULL;
        }
}
static int keys_init(struct lu_context *ctx)
{
        int i;
        int result;
        OBD_ALLOC(ctx->lc_value, ARRAY_SIZE(lu_keys) * sizeof ctx->lc_value[0]);
        if (ctx->lc_value != NULL) {
                for (i = 0; i < ARRAY_SIZE(lu_keys); ++i) {
                        struct lu_context_key *key;
                        key = lu_keys[i];
                        if (key != NULL) {
                                void *value;
                                LASSERT(key->lct_init != NULL);
                                LASSERT(key->lct_index == i);
                                value = key->lct_init(ctx);
                                if (IS_ERR(value)) {
                                        keys_fini(ctx);
                                        return PTR_ERR(value);
                                }
                                key->lct_used++;
                                ctx->lc_value[i] = value;
                        }
                }
                result = 0;
```

```
        } else
                result = -ENOMEM;
        return result;
}
/*
 * Initialize context data-structure. Create values for all keys.
 */
int lu_context_init(struct lu_context *ctx)
{
        memset(ctx, 0, sizeof *ctx);
        keys_init(ctx);
        return 0;
}
/*
 * Finalize context data-structure. Destroy key values.
 */
void lu_context_fini(struct lu_context *ctx)
{
        keys_fini(ctx);
}
```

8 State Specification

See 5.6 on page 6 for a cache-related state transitions description.

8.1 Resources Involved and Their State

Shared resources, protected by lock in brackets:

- `lu_site`
  - `->ls_hash` [`->ls_guard`]
  - `->ls_lru` [`->ls_guard`]
- `lu_device`
  - `->ld_ref` [atomic]
- `lu_object_header`
  - `->loh_hash` [`->ls_guard`]
  - `->loh_lru` [`->ls_guard`]
  - `->loh_ref` [`->ls_guard`]
  - `->loh_layers`, `->loh_fid` [immutable after object setup]
  - `->loh_flags` [atomic]

- `lu_object`
  - `->lo_linkage` [immutable after object setup]
- `lu_context`
  - `->lc_value` [immutable after context setup]
- `lu_context_key`
  - `->lct_index` [immutable after key setup]
  - `->lct_used` [immutable after key setup]

8.2 Locking

LRU list, hash table and object reference counters are protected by single per-site spin-lock. If need arise, this lock can be split into multiple locks. All locking logic is currently encapsulated in few lu_object.c functions. One obvious split is separation of LRU lock, from lock covering both hash table and object counters. In the case of multiple locks, lock ordering has to be defined.