# HLD for OSD API Changes

Huang Hua

Feb 27, 2006

## 1 Introduction

In order to achieve better performance, better architecture, and better code organization, we change the call stack layering of Lustre, especially on MDS. The mds-layering HLD describes this topic in detail. According to the change in layering on MDS, there will also be some changes in API.

This document descirbes the changed OSD API.

## 2 Requirements

According to the mds-layering HLD and the managements, the main requirement for OSD is to change its API to provide access to MDD. In detail, the API changes will happen in the following aspects:

- Objects operations API;
- Locking API;
- Transaction API;
- FLD API;

## 3 Functional specification

Object Storage Device (OSD) provides storage space and management operations to its user. Compared with the logical block address in block device, OSD is accessed by object. OSD in Lustre manages the physical device itself.

There are five different sets of APIs that OSD should export:

- Objects management operations;
- Lock related operations;
- Transaction related operations;
- FLD related operations;
- Object indexing operations; (These functions will be discussed in another HLD.)

## 3.1 Objects management API

- Lookup an object

```
struct osd_object *osd_object_lookup(IN struct obd_device *obd,
                                     IN struct osd_object *parent,
                                     IN char *object_name);
```

- Increase reference count

```
int osd_object_get(IN struct obd_device *obd,
                   IN struct osd_object *object);
```

- decrease reference count

```
int osd_object_put(IN struct obd_device *obd,
                   IN struct osd_object *object);
```

- Create an object

```
int osd_object_create(IN struct obd_device *obd,
                      IN struct osd_object *parent,
                      IN OUT struct osd_object *child,
                      IN char *child_name,
                      IN mode_t mode,
                      IN create_info_t *ci);
```

- Remove an object

```
int osd_object_remove(IN struct obd_device *obd,
                      IN struct osd_object *parent,
                      IN struct osd_object *object);
```

- Get attributes for an object

```
int osd_object_getattr(IN struct obd_device *obd,
                       IN struct osd_object *object,
                       OUT struct attr *attr);
```

- Set attributes for an object

```
int osd_object_setattr(IN struct obd_device *obd,
                       IN struct osd_object *object,
                       IN struct attr *attr);
```

- Get metadata EA for an object

```
int osd_object_getmea(IN struct obd_device *obd,
                      IN struct osd_object *object,
                      OUT struct mea **mea);
```

- Set metadata EA for an object

```
int osd_object_setmea(IN struct obd_device *obd,
                      IN struct osd_object *object,
                      IN struct mea *mea);
```

- Add an entry in a directory object

```
int osd_object_add_entry(IN struct obd_device *obd,
                         IN struct osd_object *parent,
                         IN char *object_name,
                         IN struct osd_object *object,
                         IN int type);
```

- Delete an entry in a directory object

```
int osd_object_del_entry(IN struct obd_device *obd,
                         IN struct osd_object *parent,
                         IN char *object_name);
```

- Increase the link count for an object

```
int osd_object_inc_link(IN struct obd_device *obd,
                        IN struct osd_object *object);
```

- decrease the link count for an object

```
int osd_object_dec_link(IN struct obd_device *obd,
                        IN struct osd_object *object);
```

- Read from an object

```
int osd_object_read(IN struct obd_device *obd,
                    IN struct osd_object *object,
                    OUT char *buffer,
                    IN size_t buflen,
                    IN OUT off_t *off);
```

- Write to an object

```
int osd_object_write(IN struct obd_device *obd,
                     IN struct osd_object *object,
                     IN char *buffer,
                     IN size_t buflen,
                     IN OUT off_t *off);
```

- Get status of this OSD

```
int osd_object_getstatus(IN struct obd_device *obd,
                         OUT struct status *status);
```

- Get file system information for this OSD

```
int osd_object_statfs(IN struct obd_device *obd,
                      OUT struct statfs *statfs);
```

## 3.2 Locking API

The locking functions are used by user to acquire lock and release lock again some objects.

- Acquire a lock on an object

```
int osd_lock(IN struct obd_device *obd,
             IN struct osd_object * object,
             IN lock_mode_t mode);
```

- Try to acquire a lock on an object

```
int osd_trylock(IN struct obd_device *obd,
                IN struct osd_object * object,
                IN lock_mode_t mode);
```

- Release a lock on an object

```
int osd_unlock(IN struct obd_device *obd,
               IN struct osd_object * object);
```

- Check if an object is locked

```
int osd_islocked(IN struct obd_device *obd,
                 IN struct osd_object * object);
```

## 3.3 Transaction API

These functions are used to complete transaction based atomic operations.

- Start a transaction

```
handle_t *osd_trans_start(IN struct obd_device *obd,
                          IN trans_info_t *info);
```

- Stop a transaction

```
int osd_trans_stop(IN struct obd_device *obd,
                   IN handle_t *handle,
                   IN int force_commit);
```

- Abort a transaction

```
int osd_trans_abort(IN struct obd_device *obd,
                    IN handle_t *handle);
```

- Register a callback

```
int osd_trans_register_cb(IN struct obd_device *obd,
                          IN callback_func_t *cb,
                          IN callback_data_t * cbd);
```

### 3.4 FLD API

- Create a FID location mapping

```
int osd_fld_create(IN struct obd_device *obd,
                   IN struct lustre_fid *fid,
                   IN fld_mapping_t *mapping);
```

- Delete a FID location mapping

```
int osd_fld_delete(IN struct obd_device *obd,
                   IN struct lustre_fid *fid);
```

- Lookup a FID location mapping

```
int osd_fld_lookup(IN struct obd_device *obd,
                   IN struct lustre_fid *fid,
                   OUT fld_mapping_t *mapping);
```

- Update a FID location mapping

```
int osd_fld_update(IN struct obd_device *obd,
                   IN struct lustre_fid *fid,
                   OUT fld_mapping_t *mapping);
```

## 4 Use cases

According to the mds-layering HLD, OSD in Lustre should provide those APIs listed in the previous section. In this section, I will describe how to use these interfaces.

### 4.1 Objects management

In CMD environment, objects can be created and/or manipulated locally or remotely. A remote operation initiated by another MDT is called a partial operation, or a dependent operation. So some of the APIs listed above may be used to complete such an partial operation. For example, creating a hard link to an object which locates on another MDT, will create an object locally (by calling osd_object_create), and increase the link count of an object which locates on another MDT (by calling osd_object_inc_link). So the user of these APIs should pay attention to this situation.

## 4.2 Lock operations

The user may want to access some objects exclusively. This can be achieved by locking an object with proper mode. Write lock is exclusive, and read lock is sharable with another read lock. Exclusive lock can be acquired twice. Calling the obd_lock may be blocked if the object has held a conflict lock. The user may use osd_trylock if she/he does not to be blocked. Lock against object should be release after use.

OSD lock is local to this OSD. OSD lock is volatile, which means that the lock is invalid after umount/reboot.

## 4.3 Atomic operations based on transactions

Atomic operations of file system/object storage are very important. Journaling is a very common technique to meet this requirement. OSD of Lustre also provides such journaling based on ext3/reiserfs.

The user starts an transaction by calling osd_trans_start, and got a valid opaque handle on success. Then user can advance with his/her objects operations. After completed those operations, the user call osd_trans_stop to close this transaction. Also he/she can abort this transaction by calling osd_trans_abort.

The upper layer of OSD also may register a callback function which will be called every time a transaction is committed. The committed transaction number will be returned back as a parameter for the callback function. The user may return this transaction number to clients to release request buffers.

## 4.4 FID location management

In CMD environment, we need to know on which MDT an object is stored. An object is represented by a FID. So we need to map an object to its MDT. This is achieved by FID location management.

The user add such a mapping information by calling osd_fld_create. The mapping information will be stored in this OSD and can be lookup-ed later by calling osd_fld_lookup. The mapping information may be deleted by calling osd_fld_delete. When object is migrated, the migrate server may want to modify the mapping information with osd_fld_update.

All these fld related function calls are synchronously flushed to persistent storage. This can promise atomic update.

# 5 Logic specification

## 5.1 Objects management API

Objects in OSD are identified externally by a memory data struct called “struct osd_object”. We call this is an object’s handle. The upper layer on OSD never knows about operating system specific elements, such as inode/dentry. The internal representation of objects is hidden from upper layer in order to achieve portability, security.

These objects management interfaces are totally reenterable if the underlying file system is reenterable. We will take much care on this topic. At the current time, OSD is implemented on ldiskfs (patched/enhanced Ext3). So these interfaces will be implemented by using the ldiskfs as the back end facilities.

### 5.1.1 Lookup an object

```
struct osd_object *osd_object_lookup(IN struct obd_device *obd,
                                     IN struct osd_object *parent,
                                     IN char *object_name);
```

1. Description:

   The osd_object_lookup function looks up an object in the parent object according to a name.

2. Parameters:

   - obd: this OSD;
   - parent: the parent object;
   - object_name: the child name.

3. Return value:

   On success, the child’s handle is returned. Otherwise, error pointer is returned. Caller of this function should check the return value by using IS_ERR macro.

   If the parent is NULL, then the root object will be returned back.

### 5.1.2 Increase reference count

```
int osd_object_get(IN struct obd_device *obd,
                   IN struct osd_object *object);
```

1. Description:

   The osd_object_get function increases an object’s reference count. The user should call this function before using an object.

2. Parameters:

   - obd: this OSD;
   - object: this object's reference will be increased;

3. Return value:

   On success, zero is returned. Other value indicates error.

#### 5.1.3 decrease reference count

```
int osd_object_put(IN struct obd_device *obd,
                   IN struct osd_object *object);
```

1. Description:

   The osd_object_put function decreases an object's reference count. The user should call this function after having finished using an object. When the object's reference count drops to zero, OSD will deallocate the memory.

2. Parameters:

   - obd: this OSD;
   - object: this object's reference will be decreased;

3. Return value:

   On success, zero is returned. Other value indicates error.

#### 5.1.4 Create an object

```
int osd_object_create(IN struct obd_device *obd,
                      IN struct osd_object *parent,
                      IN OUT struct osd_object *child,
                      IN char *child_name,
                      IN create_info_t *ci,
                      IN mode_t mode);
```

1. Description:

   The osd_object_create function create an object named child_name under parent object. The newly created child has the user specified mode. Other object attributes are initializes to default.

2. Parameters:

   - obd: this OSD;
   - parent: parent object under which child will be created;

- child: the newly created child;
- child_name: child's name specified by user;
- ci: create information for this child object. For example, the symlink name/default EA/mds_num hint/etc.
- mode: child's mode specified bu user.

3. Return value:

   On success, zero is returned and child pointer to a valid child object. Other return value indicates error.

#### 5.1.5 Remove an object

```
int osd_object_remove(IN struct obd_device *obd,
                      IN struct osd_object *parent,
                      IN struct osd_object *object);
```

1. Description:

   The osd_object_remove function removes an object under parent physically from the storage.

2. Parameters:

   - obd: this OSD;
   - parent: the parent which contains the child object;
   - object: the object to be removed;

3. Return value:

   On success, zero is returned. Other return value indicates error.

#### 5.1.6 Get attributes for an object

```
int osd_object_getattr(IN struct obd_device *obd,
                       IN struct osd_object *object,
                       OUT struct attr *attr);
```

1. Description:

   The osd_object_getattr function gets attributes of the specified object. The attributes include standard file attributes.

2. Parameters:

   - obd: this OSD;
   - object: the object to obtain attributes from;

- attr: user supplied memory to store attributes;

3. Return value:

   On success, zero is returned. Other return value indicates error.

#### 5.1.7 Set attributes for an object

```
int osd_object_setattr(IN struct obd_device *obd,
                       IN struct osd_object *object,
                       IN struct attr *attr);
```

1. Description:

   The osd_object_setattr function sets attributes for the specified object. The attributes include standard file attributes.

2. Parameters:

   - obd: this OSD;
   - object: the object to set attributes for;
   - attr: user supplied attributes;

3. Return value:

   On success, zero is returned. Other return value indicates error.

#### 5.1.8 Get metadata EA for an object

```
int osd_object_getmea(IN struct obd_device *obd,
                      IN struct osd_object *object,
                      OUT struct mea **mea);
```

1. Description:

   The osd_object_getmea function gets metadata extended attributes for the specified object. OSD will allocate the needed memory, and the caller are responsible to deallocate them.

2. Parameters:

   - obd: this OSD;
   - object: the object to get MEA from;
   - mea: user supplied pointer to receive MEA;

3. Return value:

   On success, zero is returned. Other return value indicates error.

### 5.1.9 Set metadata EA for an object

```
int osd_object_setmea(IN struct obd_device *obd,
                      IN struct osd_object *object,
                      IN struct mea *mea);
```

1. Description:

   The osd_object_setmea function sets metadata extended attributes for the specified object.

2. Parameters:

   - obd: this OSD;
   - object: the object to get MEA from;
   - mea: user supplied MEA;

3. Return value:

   On success, zero is returned. Other return value indicates error.

### 5.1.10 Add an entry in a directory object

```
int osd_object_add_entry(IN struct obd_device *obd,
                         IN struct osd_object *parent,
                         IN char *object_name,
                         IN struct osd_object *object,
                         IN int type);
```

1. Description:

   The osd_object_add_entry function add the specified object to the parent. This is a partial (depend) operation in CMD.

2. Parameters:

   - obd: this OSD;
   - parent: the parent object to add entry to;
   - object_name: the name of entry to be added;
   - object: the object handle of this entry;
   - type: type of the entry/object;

3. Return value:

   On success, zero is returned. Other return value indicates error.

#### 5.1.11 Delete an entry in a directory object

```
int osd_object_del_entry(IN struct obd_device *obd,
                         IN struct osd_object *parent,
                         IN char *object_name);
```

1. Description:

   The osd_object_del_entry function delete the specified object from the parent. This is a partial (depend) operation in CMD.

2. Parameters:

   - obd: this OSD;
   - parent: the parent object to delete entry from;
   - object_name: the name of entry to be deleted;

3. Return value:

   On success, zero is returned. Other return value indicates error.

#### 5.1.12 Increase the link count for an object

```
int osd_object_inc_link(IN struct obd_device *obd,
                        IN struct osd_object *object);
```

1. Description:

   The osd_object_inc_link function increases the link count of the specified object. This is a partial (depend) operation in CMD. The link count is stored persistently on storage.

2. Parameters:

   - obd: this OSD;
   - object: the user supplied object to increase link count;

3. Return value:

   On success, zero is returned. Other return value indicates error.

#### 5.1.13 decrease the link count for an object

```
int osd_object_dec_link(IN struct obd_device *obd,
                        IN struct osd_object *object);
```

1. Description:

   The osd_object_dec_link function decreases the link count of the specified object. This is a partial (depend) operation in CMD. The link count is stored persistently on storage.

2. Parameters:

   - obd: this OSD;
   - object: the user supplied object to decrease link count;

3. Return value:

   On success, zero is returned. Other return value indicates error.

#### 5.1.14 Read from an object

```
int osd_object_read(IN struct obd_device *obd,
                    IN struct osd_object *object,
                    OUT char *buffer,
                    IN size_t buflen,
                    IN OUT off_t *off);
```

1. Description:

   The osd_object_read function reads some data/metadata from an object.

2. Parameters:

   - obd: this OSD;
   - object: the user supplied object to read data/metadata from;
   - buffer: the user supplied memory to store data/metadata;
   - buflen: the user supplied memory size;
   - off: the logical/internal offset from the beginning of the file. This offset will be updated to new value after this read operation;

3. Return value:

   On success, the actual number of bytes read is returned (zero indicates end of the object), and the offset is advanced by this number. It is not an error if this number is smaller than the number of bytes requested. On error, a negative value is returned, and the offset is undefined.

#### 5.1.15 Write to an object

```
int osd_object_write(IN struct obd_device *obd,
                     IN struct osd_object *object,
                     IN char *buffer,
                     IN size_t buflen,
                     IN OUT off_t *off);
```

1. Description:

   The osd_object_write function writes some data/metadata to an object.

2. Parameters:

   - obd: this OSD;
   - object: the user supplied object to write data/metadata to;
   - buffer: the user supplied memory which holds data/metadata;
   - buflen: the user supplied memory size;
   - off: the logical/internal offset from the beginning of the file. This offset will be updated to new value after this write operation;

3. Return value:

   On success, the actual number of bytes written is returned (zero indicates nothing was written), and the offset is advanced by this number. On error, a negative value is returned.

#### 5.1.16 Get status of this OSD

```
int osd_object_getstatus(IN struct obd_device *obd,
                         OUT struct status *status);
```

1. Description:

   The osd_object_getstatus function gets status information of this OSD.

2. Parameters:

   - obd: this OSD;
   - status: the user supplied memory to receive status information;

3. Return value:

   On success, zero is returned. Other value indicates error.

#### 5.1.17 Get file system information for this OSD

```
int osd_object_statfs(IN struct obd_device *obd,
                      OUT struct statfs *statfs);
```

1. Description:

   The osd_object_statfs function gets file system information of this OSD.

2. Parameters:

   - obd: this OSD;
   - statfs: the user supplied memory to receive statfs information;

3. Return value:

   On success, zero is returned. Other value indicates error.

## 5.2 Locking API

The locking functions are used by user to acquire lock and release lock again some objects.

I have two proposal to implement the lock for OSD.

1. VFS Lock.

   All the OSD locks can be implemented as VFS locks in Linux. OSD is an OS specific and network-free component in Lustre. No other server components except OSD will use VFS directly. So we can use VFS lock to fulfil the lock requirements, that is by down(&inode->i_sem) and up(&inode->i_sem);

2. OSD private lock.

   FIXME: We also can do locks without the help of VFS. We can add a semaphore in some struct which can be access in OSD and has a one-one relationship to Lustre object. All the OSD locks are implemented as locks agianst this per-object lock. For Example, we can add a semaphore into obdo, and do OSD locking against this semaphore.

#### 5.2.1 Acquire a lock on an object

```
int osd_lock(IN struct obd_device *obd,
             IN struct osd_object * object,
             IN lock_mode_t mode);
```

1. Description:

   This function locks the specified object with specified lock mode, and may be blocked if someone has already holds some conflict locks on this object.

2. Parameters:
   - obd: this OSD;
   - object: the object to take lock on;
   - mode: lock mode; /* FIXME: the mode can be write lock or read lock; write lock is exclusive; read lock can be shared with another read lock; */
3. Return value:

   On success, zero is returned. Other value indicates error.

#### 5.2.2 Try to acquire a lock on an object

```
int osd_trylock(IN struct obd_device *obd,
                IN struct osd_object * object,
                IN lock_mode_t mode);
```

1. Description:

   This function does the same thing as the above one except that it will return an error immediately if someone has already held a lock on this object.
2. Parameters:
   - obd: this OSD;
   - object: the object to take lock on;
   - mode: lock mode; the mode can be write lock or read lock; write lock is exclusive; read lock can be shared with another read lock;
3. Return value:

   On success, zero is returned. Other value indicates error.

#### 5.2.3 Release a lock on an object

```
int osd_unlock(IN struct obd_device *obd,
               IN struct osd_object * object);
```

1. Description:

   This function release a lock against the specified object.
2. Parameters:
   - obd: this OSD;
   - object: the object to release lock on;
3. Return value:

   On success, zero is returned. Other value indicates error.

#### 5.2.4 Check if an object is locked

```
int osd_islocked(IN struct obd_device *obd,
                 IN struct osd_object * object);
```

1. Description:

   This function tests if the specified object is locked.

2. Parameters:

   - obd: this OSD;
   - object: the object to test;

3. Return value:

   If the object is already locked, return value is 1; If is not locked, return value is 0; Negative value indicates error.

### 5.3 Transaction API

These functions are used to complete transaction-based atomic operations. We will implement thses interfaces by using patched JBD & Ext3 file system journaling functions.

These interfaces are helpful to complete object management, partial object operations and so on.

/*FIXME Should we add the handle parameter to every objects operation in 0.3.1/0.5.1? */

#### 5.3.1 Start a transaction

```
handle_t *osd_trans_start(IN struct obd_device *obd,
                          IN trans_info_t *info);
```

1. Description:

   This function starts a new transaction. This function may block until the underlying log can guarantee that much space. The caller use the returned handle to continue on later operations.

2. Parameters:

   - obd: this OSD;
   - info: the required information for starting a new transaction, such as number of blocks affected.

3. Return value:

   On success, the newly created handle is returned. Otherwise, NULL is returned to indicate an error.

### 5.3.2 Stop a transaction

```
int osd_trans_stop(IN struct obd_device *obd,
                   IN handle_t *handle,
                   IN int force_commit);
```

1. Description:

   This function stops a transaction with the specified handle.

2. Parameters:

   - obd: this OSD;
   - handle: handle of the transaction to stop;
   - force_commit: indicate whether to force commit.

3. Return value:

   On success, zero is returned. Other value indicates an error.

### 5.3.3 Abort a transaction

```
int osd_trans_abort(IN struct obd_device *obd,
                    IN handle_t *handle);
```

1. Description:

   This function abort a transaction with the specified handle. This function may be used during CMD to complete partial operations. /*FIXME JBD can not abort a started transcation, only can abort the entire journal. So is this API necessary? esspecially for rollback and recovery? and for partial operation?*/

2. Parameters:

   - obd: this OSD;
   - handle: handle of the transaction to abort;

3. Return value:

   On success, zero is returned. Other value indicates an error.

### 5.3.4 Register a callback

```
int osd_trans_register_cb(IN struct obd_device *obd,
                          IN callback_func_t *cb,
                          IN callback_data_t * cbd);
```

1. Description:

   This function registers a callback function for the user to monitor the commit of transaction. This may be useful to complete Client-MDT transactions. The callback will be called when a transaction is committed.

2. Parameters:

   - obd: this OSD;
   - cb: the user supplied callback function. This function will be called every time a transaction is committed;
   - cbd: when the callback function is called, the callback data specified by cbd is passed as a parameter to the function;

3. Return value:

   On success, zero is returned. Other value indicates an error.

## 5.4 FLD API

All these FLD functions are synchronous and all there modifications are flushed to disk when the function calls return. This may help to achieve atomic. At the same time, the mostly called function is lookup, and that deos not modify disk content. All FLD users also have FLD cache. This will greatly reduce the overhead.

Though we use the “struct lustre_fid” as argument to these APIs, but we only use the fid sequence as the key. The main purpose here is to map fid sequence to MDT number (also some other information).

Internally, these FLD functionalities are implemented by using IAM interfaces.

### 5.4.1 Create a FID location mapping

```
int osd_fld_create(IN struct obd_device *obd,
                   IN struct lustre_fid *fid,
                   IN fld_mapping_t *mapping);
```

1. Description:

   This function creates a mapping information for the specified fid and stores them in the OSD. The user may lookup and delete this mapping in a later time.

2. Parameters:

   - obd: this OSD;
   - fid: the user specified fid to add mapping information to;

- mapping: the mapping information of the fid; The mapping information mainly includes MDT number, also includes some other information about objects. This will be discussed in DLD.

3. Return value:

   On success, zero is returned. Other value indicates an error.

### 5.4.2 Delete a FID location mapping

```
int osd_fld_delete(IN struct obd_device *obd,
                   IN struct lustre_fid *fid);
```

1. Description:

   This function deletes a mapping information for the specified fid and stores them in the OSD.

2. Parameters:

   - obd: this OSD;
   - fid: the user specified fid to delete mapping information;

3. Return value:

   On success, zero is returned. Other value indicates an error.

### 5.4.3 Lookup a FID location mapping

```
int osd_fld_lookup(IN struct obd_device *obd,
                   IN struct lustre_fid *fid,
                   OUT fld_mapping_t *mapping);
```

1. Description:

   This function looks up a mapping information for the specified fid.

2. Parameters:

   - obd: this OSD;
   - fid: the user specified fid to find mapping information for;
   - mapping: the user supplied memory to store mapping information;

3. Return value:

   On success, zero is returned. Other value indicates an error.

#### 5.4.4 Update a FID location mapping

```
int osd_fld_update(IN struct obd_device *obd,
                   IN struct lustre_fid *fid,
                   IN fld_mapping_t *mapping);
```

1. Description:

   This function updates a mapping information for the specified fid transactionally and atomically. By using the delete and create function may modify a mapping, but that may cause problem in failure situation.

2. Parameters:

   - obd: this OSD;
   - fid: the user specified fid to update mapping information for;
   - mapping: the user supplied mapping information;

3. Return value:

   On success, zero is returned. Other value indicates an error.

# 6 State management

## 6.1 State invariants

## 6.2 Scalability & performance

## 6.3 Recovery changes

## 6.4 Locking changes

## 6.5 Disk format changes

## 6.6 Wire format changes

## 6.7 Protocol changes

## 6.8 API changes

The auguments for APIs are changed to use FID. Now FID is the only universal identification for an object in Lustre.

### 6.9 RPCs order changes

# 7 Alternatives

## 7.1 Locks

The OSD lock can be implemented by VFS lock, or by adding semaphore into some object-related data structure as a per-object lock?

The two kinds are both simple. The former can be used to synchronize with other file system operations. The later is Lustre-owned lock and is depenent on OS.

# 8 Focus for inspections

1. Are these APIs complete? redundant? especially APIs about object operations.
2. Are these APIs reasonable? especially the transaction operations.
3. Are these APIs feasible?
4. What about the lock? Lock by VFS, or private per-object semaphore?