Interoperability at server side

Rahul Deshmukh

5 August 2008

1 Introduction

1.1 Definitions

Following terminologies are used in this HLD (These definitions are taken from arch page):

- **OLD:** any major release in b1_6 line of development. (release 1.6)
- **OLD.X:** a release in b1_6 line containing client that is able to interact with a NEW.0 md server. (Tentatively 1.8.)
- **NEW.0:** first release based on HEAD. This features kernel server, and uses ldiskfs as a back-end. This is (tentatively) 2.0. It is important to note that NEW.0 is a temporary intermediate release whose purpose is to effect transition from ldiskfs-based to DMU-based clusters.
- **OLD object, NEW object:** In NEW.0 release, MDS will be upgraded from fid-less to fid-enable, keeping the underlying ldiskfs storage in tact. Due to this in NEW.0 release, NEW objects (created by NEW.0 fid-enabled MDS) and OLD objects (already present, created by OLD.X fid-less MDS) will be present.
- **NEW.1:** next release based on HEAD. This release introduces support for fids on OST, and DMU as a back-end, in addition to continued support for ldiskfs. This is (tentatively) 2.x.
- **Fill-in-fid:** a special not otherwise used fid value, reserved to indicate in a CREATE RPC that client requests server to generate fid for newly created object on client's behalf. This fid is taken from one of the system-reserved fid sequences.

1.2 Background

In OLD release both Meta Data Server (MDS) and it's storage, do not use fid. In NEW.1 release both MDS and it's storage will be fid-enabled. To provide interoperability between these two releases (OLD/1.6 and NEW.1/2.X release) according to the (+-1) policy, OLD.X and NEW.0 is introduced.

- OLD.X will address network protocol related interoperability changes (i.e. here client will have ability to talk fid-less and fid-enabled protocol).
- And NEW.0 release having interoperability changes which will address disk interoperability part (i.e. fid-enabled MDS running on OLD disk format).

This HLD will concentrate on disk interoperability related events i.e. changes introduced in NEW.0 release.

ID	Type of MDS	Type of MDS-Storage	Description
OLD	fid-less	fid-less	Fid-less version
OLD.X	fid-less	fid-less	Interoperability: Network protocol
NEW.0	fid	fid-less	Interoperability: Disk
NEW.1	fid	fid	Fid-enabled version

Understanding of fid and NEW MDS stack [1] is assumed. In brief, the MDS stack consists of MDT (networking details), CMM (placement policies), MDD (meta data operations) and OSD layer (object storage details). (i.e. MDT->CMM->MDD->OSD)

2 Architecture

Link to the architecture design: *http://arch.lustre.org/index.php?title=Interoperability_fids_zfs*

Understanding of interoperability details mention in the link are assumed.

3 Requirements

Requirements for running fid-enabled MDS on fid-less storage.

[1] MDS stack overview is present in *mds-layering-hld.lyx*

3.1 Functional requirements

3.1.1 NEW.0 Release

ID	Quality	Trigger	Affected	Description
Index operations: fid_inode_ea	Usability	Create, lookup, delete operations on newly created files, from user point of view OR index operation on NEW objects, from Lustre point of view	OSD	For NEW object, name->fid mapping should work i.e. fid should be stored in inode's ea to make it persistent.
Index operations: igif	Usability	Lookup or delete operations on already created files, from user point of view OR index operation on OLD objects, from Lustre point of view.	OSD	OLD object will not have fid stored in inode ea, so to support name->fid mapping, igif (which is dynamically generated fid, from ino and inode generation or in simpler words it is used as fid for those files who do not have formal fid) support must be present
Iterator operations	Usability	Directory read operation, empty directory check (say while removing directory) from user point of view OR iterator operations from Lustre point of view.	OSD	Support for iterator based on, fid-less storage.This also must handle OLD as well as NEW objects.
mount	Usability	Mount command	OSD	Fid-enabled mds should be able to mount on fid-less storage, transparently.

3.1.2 OLD.X release

ID	Quality	Trigger	Affected	Description
To remove fid ea	Usability	Any operation that will access a file	N/A	To support upgrade and downgrade multiple times, Fid-less (or OLD.X) MDS needs to remove fid from inode ea (inserted by NEW.0 MDS), if present.

3.2 Architecture requirements

- All the changes needs to be handled at OSD layer. This will make sure minimal or no changes needed at the layer above OSD (MDS stack is MDT->CMM->MDD->OSD).

4 External Functional specifications

In NEW.0 release we will upgrade fid-less MDS to fid-enabled MDS, keeping the underlying storage same (i.e. ldiskfs or can be called as fid-less). To make this possible, following main conditions needs to be fulfilled (restating the requirement here for better understanding).

- To maintain index mapping for OLD as well as NEW object (requirement ID from section 3.1 “fid_inode_ea” and “igif”), to support basic operations (like file lookup). Note here that index mappings are introduced in fid-enabled MDS to support Cluster Meta Data (CMD) operations.
- To support iterator running on fid-less storage (requirement ID from section 3.1 “Iterator operations”) which is required for operations related to reading directory (e.g. ls <no input>)

4.1 Prototypes

This section describe what needs to be done to fulfill above mention conditions.

4.1.1 Index Operation APIs:

Index operations, helps to maintain persistent mapping between key (e.g. fid, ino) and it’s value (e.g. name, fid). Fid-enable MDS mainly maintains three types of persistent mappings. These are name->fid, fid->ino and fid->mdt.

To maintain name->fid mapping, fid needs to be stored in such a way that, it can be retried back for given name during lookup and also it should not introduced any disk format change (As in NEW.0 release, meta data is stored in OLD.X format or ldiskfs format). If fid is stored inode’s extended attributes (ea), then it will satisfy both conditions to maintain name->fid mapping.

Note: This is just for reference, when fid-enabled MDS runs on fid-enabled storage i.e. ldiskfs+iam, then fid is stored in directory entry to maintained name->fid mapping.

Prototypes of functions which will be implemented in NEW.0 release, to maintain name->fid mapping, for both NEW and OLD objects are as follows (These APIs are self explanatory)

- Index insert:

```
int osd_index_ea_insert (const struct lu_env *env,
                         struct dt_object  *dt,
                         const struct dt_rec *rec,
                         const struct dt_key *key,
                         struct thandle *handle,
                         struct lustre_capa *capa);
```

- Index lookup:

```
int osd_index_ea_lookup (const struct lu_env *env,
                         struct dt_object *dt,
                         struct dt_rec *rec,
                         const struct dt_key *key,
                         struct lustre_capa *capa);
```

- Index delete:

```
int osd_index_ea_delete (const struct lu_env *env,
                         struct dt_object *dt,
                         const struct dt_key *key,
                         struct thandle *handle,
                         struct lustre_capa *capa);
```

Other mappings (fid->ino and fid->mdt) can be maintain, with less or no change, as these mappings are stored in special files (viz. /oi and /fld) . These mappings will be maintained by existing functions osd_index_insert(), osd_index_lookup(), osd_index_delete().

4.1.2 Igif handling:

In NEW.0 release, MDS will be upgraded from fid-less to fid-enable, keeping the underlying ldiskfs storage in tact. Due to this in NEW.0 release, NEW objects (created by NEW.0 fid-enabled MDS) and OLD objects (already present, created by OLD.X fid-less MDS) will be present. So to support name->fid mapping for OLD object igif will be used. Igif is nothing but dynamically generated fid, from ino and inode generation or in simpler words it is used as fid for those files who do not have formal fid.

In NEW.0 release igif format will be as (This format will also helps to maintain fid version, which will be useful in version based recovery):

- sequence = (0:33 and ino:31)
- object_id = gen

- version = LUSTRE_FID_VERSION

Due to change in igif format, following function will be impacted or need to be modified:

- Build Igif:

  To build igif following function is used. This function is mainly responsible to generate igif (i.e. dynamic fid) from ino and inode generation number.

  ```
  void lu_igif_build(struct lu_fid *fid, __u32 ino, __u32 gen)
  ```

- Check igif:

  This function checks if the input value is igif or not (in detailed, it uses sequence no to make a decision)

  ```
  static inline int fid_is_igif(const struct lu_fid *fid)
  ```

- Extracting values from igif:

  Following functions extracts the particular(ino, inode generation number) value from igif format. These will modified considering changed in igif format.

  ```
  __u32 lu_igif_ino(const struct lu_fid *fid)
  __u32 lu_igif_gen(const struct lu_fid *fid)
  ```

4.1.3 Iterator Operation API:

Iterator operations are used to traverse the index mapping (i.e. key->value pair). In NEW.0 release, to traverse fid->ino and fid->mdt mapping generic IAM iterator can be used. But to traverse name->fid mapping new iterator API needs to be implemented (as fid is stored in inode's ea). Their prototypes are as follows (These APIs are self explanatory):

- dio_it->init()

```
struct dt_it *osd_it_ea_init (const struct lu_env *env,
                              struct dt_object *dt,
                              int writeable,
                              struct lustre_capa *capa);
```

- dio_it->fini()

```
void osd_it_ea_fini(const struct lu_env *env,
                    struct dt_it *di);
```

- dio_it->get()

```
int osd_it_ea_get (const struct lu_env *env,
                   struct dt_it *di,
                   const struct dt_key *key);
```

- dio_it->put()

```
void osd_it_ea_put (const struct lu_env *env, struct dt_it *di);
```

- dio_it->next()

```
int osd_it_ea_next (const struct lu_env *env, struct dt_it *di);
```

- dio_it->del()

```
int osd_it_ea_del (const struct lu_env *env, struct dt_it *di,
                   struct thandle *th);
```

- dio_it->key()

```
struct dt_key *osd_it_ea_key (const struct lu_env *env,
                              const struct dt_it *di);
```

- dio_it->key_size()

```
int osd_it_ea_key_size (const struct lu_env *env,
                        const struct dt_it *di);
```

- dio_it->rec()

```
struct dt_rec *osd_it_ea_rec (const struct lu_env *env,
```

- dio_it->store()

```
__u64 osd_it_ea_store (const struct lu_env *env,
                       const struct dt_it *di)
```

- dio_it->load()

```
int osd_it_ea_load(const struct lu_env *env,                                         co
```

4.2 Layering of API’s

4.2.1 Index API

The layering of API, applicable to index operations are as follows:

- fid->ino and fid->mdt mapping

Layer 1: User of the API	FLD, OSD (e.g. osd_oi_insert())
Layer 2: Indexing API used (Wrapper API)	Indexing API at OSD (e.g. osd_index_insert())
Layer 3: Generic API used by Layer 2	IAM Indexing API (e.g. iam_index_insert())

  To understand it completely e.g. for fid->ino mapping functions will be called in sequence as osd_oi_inset()->osd_index_insert->iam_index_insert()

- name->fid mapping

Layer 1: User of the API	MDD (e.g __mdd_index_insert())
Layer 2: Indexing API used	Indexing API at OSD (e.g. osd_index_ea_insert())

4.2.2 Iterator API

The API layering for iterator API is as follows:

Layer 1: User of the API	MDD Layer (e.g __mdd_readpage())
Layer 2: Iterator API	Iterator API at OSD (e.g. osd_it_init())

4.2.3 Igif:

The layering of mostly used API, are as follows:

- lu_igif_build

  This function is used by osd_index_lookup(), when fid for OLD object needs to be returned.

Layer 1: User of the API (lu_igif_build)	OSD, osd_index_lookup, fid for OLD object
Layer 2: Igif check	lu_igif_build()

- fid_is_igif():

  This function is mostly used by Object Index (fid->ino) mapping functions to check that fid is igif or not. It will make sure that for igif, fid->ino mapping is not check. (as there will not be any entry) and also make sure that (osd_oi_insert->osd_index_insert()) osd_index_insert will never get igif as input.

Layer 1: User of the API (fid_is_igif)	OSD, Object index functions (i.e. osd_oi_insert())
Layer 2: Igif check	fid_is_igif()

5 Use-Case Scenarios

5.1 Describe use cases for all normal and abnormal uses of externally visible functions.

5.1.1 Index operations

Following are the use-cases for create, lookup, rename and delete operations considering new interoperability index APIs and Igif changes or requirement ID from section 3.1 “fid_inode_ea” and “igif” for NEW.0 release.

1. Object creation in NEW.0 release (i.e. touch file_name or other alternative way to create file.):

   (a) Client allocates fill-in-fid and sends file creation request to MDS.

   (b) MDS gets the request, it will first checks if the fid is equals to special fill-in-fid, then it generates new fid from an internal fid sequence.

   (c) MDS then looks for that file name, to check if it is present.

   (d) If file is not present then object allocation and initialization will happen at MDT and then at CMM layer.

   (e) Then at OSD layer, inode will be created for that file. After that fid is inserted into inode’s ea (that will be used in name->fid mapping) and fid->ino mapping is created by inserting object index entry into global file using IAM functions. Here there is no change, the way fid->ino mapping done.

   (f) After that at MDD layer, ldiskfs style directory entry (i.e. {name, ino}) is added using osd_index_ea_insert() (i.e. osd_index_ea_insert()->ldiskfs_add_entry()). This will helps to maintain name->fid mapping (i.e. name->dir_lookup()->inode->read_ea()->fid).

   (g) Success is returned to client.

2. Object lookup in NEW.0 release

   - For NEW object (i.e ls new_file).

     (a) Consider any request from client to MDS, to access inode of newly created file.

     (b) MDS gets the request, it will first lookup for that file-name, to check if it is present. For that MDD layer requests interoperability mode index lookup function (These are the function which will be build using ldiskfs functions. These functions read fid stored in inode ea), to get fid for given name. This is the first index lookup (key=name, value=fid).

     (c) If file is present, then fid stored in inode’s ea will be read. If fid exists then it will be returned by interoperability mode index lookup function.

  (d) Got the fid, OSD layer check if second index lookup (key=fid, value=ino) is needed or not by checking if fid is igif or not. In this case fid is not igif. So it requests IAM index lookup function to get ino for given fid. Here IAM index lookup functions will be used.

  (e) If fid is present in object index mapping, then IAM index lookup function returns ino.

  (f) Using ino, load the inode and read the required data.

  (g) Fid is returned to client if necessary.

- For OLD object (i.e ls old_file).

  (a) Consider any request from client to MDS, to access inode of already existing file.

  (b) MDS gets the request, it will first lookup for that file-name, to check if it is present. For that MDD layer requests interoperability mode index lookup function to get fid for given name. This is the first index lookup (key=name, value=fid).

  (c) If file is present, then fid stored in inode's ea will be read. Fid will not present in inode ea as this file is created by fid-less server. In this case interoperability mode index lookup function will dynamically generate fid, called as igif (Igif will be build using ino + generation no, pair)

  (d) Got the fid, OSD layer check if second index lookup (key=fid, value=ino) is needed or not by checking if fid is igif or not. In this case fid is igif. So it will not request to second index lookup, instead inode number and inode generation stored in igif will be returned.

  (e) Using ino, load the inode (already present in memory) and read the required data.

  (f) Fid is returned to client if necessary.

3. Object rename in NEW.0 release (i.e. mv src_file tgt_file)

   (a) Client send request to MDS to rename the file meta-data.

   (b) MDS gets the request, it will first looks for that src file name, to check if it is present.

   (c) MDS gets the request, it will first looks for that tgt file name, to check if it is present.

   (d) If the src file present and tgt file is not present then, MDD layer request to remove directory entry of src object and tgt object. This will done using interoperability mode index function. (i.e.osd_index_ea_delete()->ldiskfs_delete_entry()). This description holds true for NEW object as well for OLD object

   (e) After that MDD layer request to add directory entry for tgt object, using interoperability mode index function. (i.e. osd_index_ea_insert()->ldiskfs_add_entry()). Hence we have replaced the src name with tgt name (by removal of src dir_entry and addition of tgt dir_entry) without touching the fid stored in inode ea.This description holds true for NEW tgt object as well for OLD tgt object.

(f) Success is returned to client.

4. Object Delete in NEW.0 release (i.e. rm file_name)

   (a) Client send request to MDS to delete the file meta-data.

   (b) MDS gets the request, it will first looks for that src file name, to check if it is present.

   (c) MDD layer request to interoperability index functions to remove the name->fid mapping (i.e. remove directory entry only).

   (d) OSD layer request to IAM functions to remove the fid->ino mapping.

   (e) Object cleanup is done layer wise.Note here that inode->nlink count == 0 then only object cleanup is done.

   (f) Success is returned to client.

5.1.2 Iterator operations

Following are the use-cases for directory read, empty directory check operations considering new interoperability iterator functionality or requirement ID “Iterator operations” for NEW.0 release.

1. Directory read in NEW.0 release (i.e. ls <no_input>)

   (a) Client send directory read request to MDS.

   (b) MDS get the request, through MDD layer it iterate over the directory using interoperability mode iterator functions to get required data.

   (c) Put the required data in the format requested by client and send it to client.

2. Empty directory check in NEW.0 release (i.e. rmdir dir_name)

   (a) Client send to remove the directory meta data.

   (b) MDS get the request, through MDD layer it iterate over the directory using interoperability mode iterator functions to check if directory is empty or not.

   (c) If the directory is empty then MDS remove the directory meta data.

   (d) Success is returned to client.

5.2 Describe use cases demonstrating interoperability between the software with and without this module.

5.2.1 Interoperability between the software with this module (Interoperability within the scope of this HLD is discussed)

This HLD deals with changes done in MDS for interoperability mode. Those changes comes into picture when upgrading/downgrading from OLD.X/NEW.0 to NEW.0/OLD.X. So only those cases are considered here.

1. Up-gradation/Down-gradation from OLD.X/NEW.0 to NEW.0/OLD.X,

    - In NEW.0 release, to run fid-enabled NEW.0 MDS on fid-less OLD.X mds-storage some functionality addition will be done in NEW.0 MDS. These functionality additions are nothing but tasks discussed in this HLD (except for orphan handling task discussed in separate document, which comes into picture in case of recovery). These details are highlighted here, just to understand the place where these changes fit into. Also to understand how these changes will help, possible upgrade(+1) and downgrade(-1).
    - To start with, all clients, MDT and OSTs are running OLD.X release. OLD.X clients and OSTs are capable of talking to NEW.0 protocol.
    - First MDS will we upgraded from OLD.X to NEW.0 keeping the MDS-Storage same (i.e. fid-less or OLD.X MDS-Storage) using fail-over mechanism without losing‘ any functionality. Here clients can continue without evictions. This will be achieved considering completion of changes in MDS mention in the HLD and also the clients and OSTs ability to talk NEW.0 protocol.
    - After MDS, clients and OSTs will be upgraded (one by one) from OLD.X to NEW.0. Client and OST can be upgraded in any order. Note here that, now we have NEW.0 clients and OSTs, which only speak NEW.0 wire protocol.
    - All clients, OSTs and MDS are upgraded to NEW.0 release.
    - From here first NEW.0 client/OST can be downgraded to OLD.X in any order. Even after downgrade, they will still have ability to talk NEW.0 protocol.
    - Now we have OLD.X clients and OSTs. We know OLD.X clients and OSTs are able to talk NEW.0 protocol. So MDS can be downgraded from NEW.0 to OLD.X release (i.e. fid-less MDS and fid-less MDS-Storage). without losing any functionality (This possible also because there is not any disk-level changes in NEW.0 release).
    - Finally all clients, MDT and OSTs will be running OLD.X release.

2. Upgrade and downgrade multiple times

To support the upgrade and downgrade operation multiple times (i.e. OLD.X->NEW.0->OLD.X->NEW.0), we need to take care of (or to remove) the fid inserted into inode ea, by NEW.0 mds in NEW.0 release. The following two cases will describe the purpose to do this and how it will taken care respectively.

- The purpose, to remove ea storing fid by OLD.X mds.
  - OLD.X mds is upgraded to NEW.0 mds keeping OLD.X ldiskfs mds-storage in tact.
  - New file X is created by NEW.0 mds, with fid F.
  - NEW.X mds is downgrade to OLD.X mds, again keeping mds-storage in tact.
  - OLD.X Client C0 accesses X, and gets (ino, gen) back. C0 takes a lock on (ino, gen)
  - Again OLD.X mds is upgraded to NEW.0 mds, keeping mds-storage in tact.
  - Client C1 accesses X, and gets F back. C1 takes a lock on F.
  - Now we have two different clients accessing the same file and taking locks in different name-spaces. So to avoid it, ea storing fid needs to be removed.
- When/how OLD.X mds will remove the fid ea.
  - Any file access trigger, OLD.X mds to load an inode.
  - OLD.X mds will first check whether the inode has fid ea.
  - If the fid ea is present then remove it. This is how fid ea will be taken care off.

5.3 Describe use cases demonstrating any scalability use cases mentioned in the architecture document.

N/A

6 High Level Logic

6.1 Index Operations:

In NEW.0 release, name->fid mapping will be maintained by storing fid into inode's ea. Following sections describe how name->fid mapping will be handled in file create, lookup and delete operation.

6.1.1 File creation

When file is created in fid-enabled MDS, following main events (in which we are interested) happened:

- name->fid mapping

  In NEW.0 release, for name to fid mapping we want fid into inode ea and at the same time fid should not touched (i.e. no removal or reinsertion) till it's life time.To server this purpose we will insert fid into inode ea when object is created (i.e. in osd_object_create()) for a given file (of-course after inode allocation). When actual name->fid mapping will be done (i.e. when osd_index_ea_insert() is called) then ldiskfs style ({name, ino}) directory entry will be inserted using ldiskfs_add_entry().

  Hence using fid into inode ea and addition of directory entry, we will able to create persistent name->fid mapping (i.e. name->dir_lookup()->inode->get_ea(fid)).

  Note: There will be addition of file system specific calls while talking to ldiskfs storage (e.g. ldiskfs_add_entry()) in osd module, in NEW.0. At the same time NEW.1 has dmu-osd module to talk to ZFS. So it is worth to mention that in case of merge of osd module and dmu-osd module file-system specific calls added for interoperability, should be handled. It is also true that it might not required as there will not be interoperability support when dmu will be used.

- fid->ino mapping

  Fid->ino mapping is maintain in special file i.e. /oi. This mechanism can also be used in NEW.0 release. So in NEW.0 release, no changes will be required for fid->ino mapping (i.e. it will be handled through IAM only or using osd_index_insert()).

6.1.2 File lookup

In fid-enabled MDS lookup is done by:

- Name

  In NEW.0 release, for object lookup by name, osd_index_ea_lookup() will be implemented to get fid for given name. Implementation details are as:

  - When NEW object will be looked-up, fid will be fetched from inode's ea (.i.e. name->dir_lookup()->gut_ea(fid)) and
  - When OLD object will be looked-up, igif will be generated, as no fid stored in inode ea for OLD object (.i.e. name->dir_lookup()->gut_ea(fid)->no_fid_found->generate_igif()). Igif is nothing but dynamically generated fid, which consists of ino and inode generation.

- Fid

  In NEW.0 release there will not be any change in a way fid->ino mapping is done. So there will not be any change for lookup by fid as well (i.e. exiting osd_index_lookup() can be used).

  Note: Also before calling to osd_index_lookup(), igif check is done to confirm that osd_index_lookup will always get fid. So no need consider lookup by igif.

6.1.3 File delete

When file is delete in fid-enabled MDS, following main things happened (These are the things in which we are interested in):

- name->fid mapping

  In NEW.0 release. when to remove name->fid mapping (i.e. call to osd_index_ea_delete()) ldiskfs style directory entry will be removed (i.e. call to ldiskfs_delete_entry()). This is applicable to OLD as well as NEW object. (or there will not be any difference in OLD and NEW object handling)

  Note: inode and fid related cleanup will be done by system when nlink count becomes zero.

- fid->ino mapping

  No changes will be required to remove fid->ino mapping (i.e. it will be handled through IAM only or using osd_index_delete()).

6.2 Igif

6.2.1 To build igif

Now to support the new igif format, following things will be done:

```
fid->f_seq = ino;
fid->f_oid = gen;
fid->f_ver = LUSTRE_FID_VERSION;
```

6.2.2 To check igif

This will implemented considering following facts:

- SEQ == 1, igif
- 1<SEQ<0x100000000; Reserved
- SEQ >= 0x100000000, normal FID.

6.3 Iterator operations

In NEW.0 release to support iterator operation, we need fill struct dt_index_operations->dio_it with interoperability (i.e. ldiskfs storage) based function pointers.Their functionality is described below:

- struct dt_it

  It will contains fields to store file pointer (i.e. struct file * oie_file) and dentry (i.e. struct dirent64 * oie_dentry)

- dio_it->init()

  To initialize the iterator data structure i.e. it->oie_file and it->oie_dentry.

- dio_it->fini()

  To destroy the iterator context

- dio_it->get()

  To traverse the iterator's in memory structure and return the value (i.e. fid) for given input key (i.e. name).

- dio_it->put()

  Just to decrement the reference count.

- dio_it->del()

  To delete the value for current key position stored in iterator's in memory structure.

- dio_it->key()

  To return the key (i.e. name) at current position from iterator's in memory structure.

- dio_it->key_size()

  return the value of key size at current position from iterator's in memory structure.

- dio_it->rec()

  To return the value (i.e. packed fid/igif) at current position

  - new_obj: load_inode->getxattr(fid)->return it
  - old_obj: load_inode->generate_igif-> return it

- dio_it->store()

  To return a cookie for current position of the iterator head,so that user can use this cookie to load/start the iterator next time.

- dio_it->next()

  It will call function (say osd_ldiskfs_it_fill()) which will use ldiskfs_readdir() to load the one directory entry at a time and stored it, in-memory iterator's data structure.

- dio_it_load

  It will call function (say osd_ldiskfs_it_fill()) which will use ldiskfs_readdir() to load the one directory entry at a time and stored it, in-memory iterator's data structure.

6.4 Mount

In NEW.0 release, fid-enabled MDS will be running on fid-less (i.e. OLD) storage and it will use OLD control

files (e.g. /last_rcvd, /ROOT, /PENDING, etc.) where possible. These things will be taken care to make up-grade transparent to user.

6.5 To remove fid ea

When OLD.X MDS loads an inode, it will has to check whether the inode has fid ea (i.e. created by NEW.0 server) and if so, such fid ea will be removed.

7 State Machine Design

7.1 Locking

N/A

7.2 Cache Usage

N/A

7.3 Recovery

Recovery related details not in the scope of this document.

7.4 Disk state changes

N/A as no disk changes.

8 Test Plan

For each test configuration, all the test-cases mention in test case table will be executed.

8.1 Test case:

Test case ID	Test case description	Expected Result
obj_create	object creation through file create (e.g. touch file_name)	file creation success
obj_lookup_new	new object lookup through file lookup (e.g. ls file_name)	lookup returns valid values
obj_lookup_old	old object lookup through file lookup (e.g. ls file_name)	lookup returns valid values
obj_delete_new	new object deletion through file deletion (e.g. rm file_name)	delete returns success
obj_delete_old	old object deletion through file deletion (e.g. rm file_name)	delete returns success
obj_rename_new	rename of new object through file rename (e.g. mv a b)	rename returns success
obj_rename_old	rename of old object through file rename (e.g mv a b)	rename returns success
read_dir	execute a command to read the directory (i.e. ls <no input>)	command returns valid values
del_dir	execute a command to delete the directory (i.e. rmdir file)	delete returns success

8.2 Test configuration:

Sr.No.	Description	Client	OST	MDS
1	Clients and OSTs and MDT running OLD.X version	OLD.x	OLD.x	OLD.X
2	MDS upgraded to NEW.0 version	OLD.X	OLD.x	NEW.0
3	Client & OST is upgraded to NEW.0 version	NEW.0	NEW.0	NEW.0
4	Client & OST is downgraded to OLD.X version	OLD.X	OLD.x	NEW.0
5	MDS downgraded to OLD.X version	OLD.X	OLD.x	OLD.X

9 Plan Review