Object Index

Danilov Nikita <nikita@clusterfs.com>

2006.07.13

Contents

1 Introduction

Object Index (oi) is a server side component of cmd3 Lustre architecture that maps fid into "storage cookie".

2 Functional Specification

Introduction of fids in cmd3 led to the problem of locating object metadata (inode) by a fid. This problem is solved by a two-level mapping mechanism: first fid sequence is mapped to a sequence home server by fld service, then oi of the home server is used to map fid to the storage cookie (file system dependent unique object identifier. In case of ext3/ldiskfs storage cookie is a (inode number, generation) pair).

To summarize: oi service is local to some server, and maps fids local to the server to storage cookies understandable by local file system. Architecturally, oi service is a part of osd_device, which implements dt_device interface. All layers above dt_device operate with fids, and typical dt_device implementation has to map fid into something that underlying file system is able to work with.

2.1 interface

2.2 data types

```
/*
 * Object Index (oi) instance.
 */
struct osd_oi;
/*
 * Storage cookie. Datum uniquely identifying inode on the underlying file
 * system.
 *
 * XXX Currently this is ext2/ext3/ldiskfs specific thing. In the future this
 * should be generalized to work with other local file systems.
 */
struct osd_inode_id;
```

2.3 initialization/finalization

```
int  osd_oi_init(struct osd_thread_info *info,
                 struct osd_oi *oi, struct dt_device *dev);
void osd_oi_fini(struct osd_thread_info *info, struct osd_oi *oi);
```

2.4 locking

```
void osd_oi_read_lock(struct osd_oi *oi);
void osd_oi_read_unlock(struct osd_oi *oi);
void osd_oi_write_lock(struct osd_oi *oi);
void osd_oi_write_unlock(struct osd_oi *oi);
```

oi locking primitives are exported to the users, so that compound operations like oi lookup, followed by inode lookup can be performed atomically.

2.5 object index manipulation

```
int  osd_oi_lookup(struct osd_thread_info *info, struct osd_oi *oi,
                   const struct lu_fid *fid, struct osd_inode_id *id);
int  osd_oi_insert(struct osd_thread_info *info, struct osd_oi *oi,
                   const struct lu_fid *fid, const struct osd_inode_id *id,
                   struct thandle *th);
int  osd_oi_delete(struct osd_thread_info *info,
                   struct osd_oi *oi, const struct lu_fid *fid,
                   struct thandle *th);
```

Function prototypes are hopefully self-evident.

3 Use Cases

3.1 osd object creation

osd_device uses oi to record fid-to-cookie mapping for the newly created object:

```
static int osd_object_create(const struct lu_context *ctx, struct dt_object *dt,
                             struct lu_attr *attr, struct thandle *th)
{
        const struct lu_fid    *fid  = lu_object_fid(&dt->do_lu);
        struct osd_object      *obj  = osd_dt_obj(dt);
        struct osd_device      *osd  = osd_obj2dev(obj);
        struct osd_thread_info *info = lu_context_key_get(ctx, &osd_key);
        ...
        if (result == 0) { /* object was created successfully */
                struct osd_inode_id *id = &info->oti_id;
                /*
                 * setup storage cookie for new object...
                 */
                id->oii_ino = obj->oo_inode->i_ino;
                id->oii_gen = obj->oo_inode->i_generation;
                /*
                 * ... and insert it into oi.
                 */
                osd_oi_write_lock(&osd->od_oi);
                result = osd_oi_insert(info, &osd->od_oi, fid, id, th);
                osd_oi_write_unlock(&osd->od_oi);
        }
}
```

3.2 osd object initialization

When osd layer initializes new object identified by fid, it uses oi to obtain storage cookie and to load inode from the local file system:

```
static int osd_fid_lookup(const struct lu_context *ctx,
                          struct osd_object *obj, const struct lu_fid *fid)
{
        struct osd_thread_info *info;
        struct lu_device       *ldev = obj->oo_dt.do_lu.lo_dev;
        struct osd_device      *dev;
        struct osd_inode_id    *id;
        struct osd_oi          *oi;
        struct inode           *inode;
        int                     result;
        info = lu_context_key_get(ctx, &osd_key);
        dev  = osd_dev(ldev);
        id   = &info->oti_id;
        oi   = &dev->od_oi;
        osd_oi_read_lock(oi);
        result = osd_oi_lookup(info, oi, fid, id);
        if (result == 0) {
                inode = osd_iget(info, dev, id);
                if (!IS_ERR(inode)) {
                        obj->oo_inode = inode;
                        LASSERT(obj->oo_inode->i_sb == osd_sb(dev));
                        result = 0;
                } else
                        result = PTR_ERR(inode);
        } else if (result == -ENOENT)
                /*
                 * If fid wasn't found in oi, inodeless object is created, for
                 * which lu_object_exists() returns false. This is used in a
                 * (frequent) case when objects are created as locking anchors or
                 * placeholders for objects yet to be created.
                 */
                result = 0;
        osd_oi_read_unlock(oi);
}
```

4 Logic Specification

4.1 generalities

oi uses two mechanisms to implement fid->cookie mapping:

- persistent (iam) index accessed through standard dt_index_operations interface, where cookie is a record and fid is a key, and
- algorithmic mapping for "igif" fids.

4.2 igif

igif is a special type of fid existing for backward compatibility. Initially igifs were introduced so that new fid-based server can run on top of old pre-fid file system, idea being that igifs are in one-to-one correspondence with storage cookies, so that server can generate fids without any additional persistent data. Additionally igifs proved useful to implement oi bootstrapping (e.g., to open oi index file one has to create lu_object identifying this file, but that implies querying oi, etc.). From the oi point of view igif is a special type of fid with the following interface:

```
int   lu_fid_is_igif(const struct lu_fid *fid);
void  lu_igif_to_id (const struct lu_fid *fid, struct osd_inode_id *id);
void  lu_igif_build (struct lu_fid *fid, __u32 ino, __u32 gen);
```

This interface is sufficient to implement oi service for igifs, and this specification won't provide further details on this topic.

4.3 data types

```
struct osd_oi {
        /*
         * underlying index object, where fid->id mapping in stored.
         */
        struct dt_object    *oi_dir;
        /*
         * semaphore, synchronizing access to oi.
         */
        struct rw_semaphore  oi_lock;
};
struct osd_inode_id {
        __u64 oii_ino; /* inode number */
        __u32 oii_gen; /* inode generation */
        __u32 oii_pad; /* alignment padding */
};
```

4.4 pseudo-code

```
static const char oi_dirname[] = "oi";
static const struct dt_index_features oi_index_features = {
        .dif_flags       = DT_IND_UPDATE,
        .dif_keysize_min = sizeof(struct lu_fid),
```

```
        .dif_keysize_max = sizeof(struct lu_fid),
        .dif_recsize_min = sizeof(struct osd_inode_id),
        .dif_recsize_max = sizeof(struct osd_inode_id)
};
int osd_oi_init(struct osd_thread_info *info,
                struct osd_oi *oi, struct dt_device *dev)
{
        int result;
        struct dt_object        *obj;
        const struct lu_context *ctx;
        ctx = info->oti_ctx;
        /*
         * Initialize ->oi_lock first, because of possible oi re-entrance in
         * dt_store_open().
         */
        init_rwsem(&oi->oi_lock);
        obj = dt_store_open(ctx, dev, oi_dirname, &info->oti_fid);
        result = obj->do_ops->do_object_index_try(ctx, obj, &oi_index_features);
        oi->oi_dir = obj;
        return result;
}
void osd_oi_fini(struct osd_thread_info *info, struct osd_oi *oi)
{
        if (oi->oi_dir != NULL) {
                lu_object_put(info->oti_ctx, &oi->oi_dir->do_lu);
                oi->oi_dir = NULL;
        }
}
void osd_oi_read_lock(struct osd_oi *oi)
{
        down_read(&oi->oi_lock);
}
void osd_oi_read_unlock(struct osd_oi *oi)
{
        up_read(&oi->oi_lock);
}
void osd_oi_write_lock(struct osd_oi *oi)
{
        down_write(&oi->oi_lock);
}
void osd_oi_write_unlock(struct osd_oi *oi)
{
        up_write(&oi->oi_lock);
}
static const struct dt_key *oi_fid_key(struct osd_thread_info *info,
                                       const struct lu_fid *fid)
```

```
{
        fid_to_be(&info->oti_fid, fid);
        return (const struct dt_key *)&info->oti_fid;
}
enum {
        OI_TXN_INSERT_CREDITS = 20,
        OI_TXN_DELETE_CREDITS = 20
};
/*
 * Locking: requires at least read lock on oi.
 */
int osd_oi_lookup(struct osd_thread_info *info, struct osd_oi *oi,
                  const struct lu_fid *fid, struct osd_inode_id *id)
{
        int result;
        if (lu_fid_is_igif(fid)) {
                lu_igif_to_id(fid, id);
                result = 0;
        } else {
                result = oi->oi_dir->do_index_ops->dio_lookup
                        (info->oti_ctx, oi->oi_dir,
                         (struct dt_rec *)id, oi_fid_key(info, fid));
                id->oii_ino = be64_to_cpu(id->oii_ino);
                id->oii_gen = be32_to_cpu(id->oii_gen);
        }
        return result;
}
/*
 * Locking: requires write lock on oi.
 */
int osd_oi_insert(struct osd_thread_info *info, struct osd_oi *oi,
                  const struct lu_fid *fid, const struct osd_inode_id *id0,
                  struct thandle *th)
{
        struct dt_object    *idx;
        struct dt_device    *dev;
        struct osd_inode_id *id;
        if (lu_fid_is_igif(fid))
                return 0;
        idx = oi->oi_dir;
        dev = lu2dt_dev(idx->do_lu.lo_dev);
        id = &info->oti_id;
        id->oii_ino = cpu_to_be64(id0->oii_ino);
        id->oii_gen = cpu_to_be32(id0->oii_gen);
        return idx->do_index_ops->dio_insert(info->oti_ctx, idx,
                                             (const struct dt_rec *)id,
```

```
                                                oi_fid_key(info, fid), th);
}
/*
 * Locking: requires write lock on oi.
 */
int osd_oi_delete(struct osd_thread_info *info,
                  struct osd_oi *oi, const struct lu_fid *fid,
                  struct thandle *th)
{
        struct dt_object *idx;
        struct dt_device *dev;
        if (lu_fid_is_igif(fid))
                return 0;
        idx = oi->oi_dir;
        dev = lu2dt_dev(idx->do_lu.lo_dev);
        return idx->do_index_ops->dio_delete(info->oti_ctx, idx,
                                                oi_fid_key(info, fid), th);
}
```

5 State Specification

5.1 resources involved and their state

This module introduces no additional shared state. Some shared state exists in the underlying index implementation (iam).

5.2 locking

Access to oi is protected by read/write semaphore. Locking operations are exported to clients so that compound operations, involving oi manipulation can be made atomic.

5.3 recovery

No special recovery considerations apply. oi operates in the user-supplied transaction context.

6 Environment

6.1 disk format changes

oi introduces new top-level file for its persistent index. Default name of this file is "/oi". It is created by

```
# create index file ...
./lustre/utils/create_iam \
```

```
        -f lfix \   # with fixed size records and fixed size keys...
        -k 16   \   # key size is 16 bytes (sizeof(struct lu_fid))...
        -r 16   \   # record size is 16 bytes (sizeof(struct osd_inode_id))...
    > $mntdir/tmp/oi # and name it /oi
```