DLD of MDD over OSD

Author WangDi

2006/04/04

1 Introduction

According to the new metadata stack layer, MDD will run on top of DT device. This DLD describes how MDD running on DT device.

2 Requirement

- MDD running on top of DT device;
- MDD API only needs to care about the local stuff, which means there is no export and import in this layer;
- MDD API does not need to care about recovery and replay in the API, which will be handled in MDT or CMM;
- MDD API should present the objects with fid or mdd object, no filesystem objects(dentry or inode) should exist in the API.

3 Function Specification

According to the new layer stack, MDD will handle metadata related operations in the local node. And it has 2 groups operations, MDD device operations and MDD metadata operations. MDD device operations are used to manage the objects in the server layers, and metadata operations are used to implement the metadata related process.

3.1 MDD generic operations

3.1.1 mdd device operations

```
static struct lu_device_operations mdd_lu_ops = {
   .ldo_object_alloc   = mdd_object_alloc,
   .ldo_process_config = mdd_process_config };
struct lu_object *mdd_object_alloc(const struct lu_context *ctxt,
                                   const struct lu_object_header *hdr,
                                   struct lu_device *d);
```

1. Description:

   This API is used to allocate the object in the MDD layer.

2. Parameters:

   - ctxt: the context of the operation.
   - hdr: the header of this object list.
   - d: lu device of the object.

3. Return values:

Return the pointer of the object when success, NULL when failed.

```
int mdd_process_config(const struct lu_context *ctx, struct lu_device *d,
                       struct lustre_cfg *cfg)
```

1. Description:

   This API is used to process the log of the device.

2. Parameters:

   - ctx: the context of the operation.
   - d: lu device of the object.
   - cfg: the lustre config.

3. Return values:

Return 0 when successs, other value when failed.

3.1.2 mdd object operations

```
static struct lu_object_operations mdd_lu_obj_ops = {
      .loo_object_init    = mdd_object_init,
      .loo_object_release = mdd_object_release,
      .loo_object_free    = mdd_object_free,
      .loo_object_print   = mdd_object_print,
      .loo_object_exists  = mdd_object_exists };
static int mdd_object_init(const struct lu_context *ctxt, struct lu_object *o)
static void mdd_object_release(const struct lu_context *ctxt, struct lu_object *o)
```

1. Description:

   This API is used to init and release the object.

2. Parameters:

- ctx: the context of the operation.
- o: the object to be init/released

3. Return values:

Return 0 when success, other value when failed.

```
static void mdd_object_free(const struct lu_context *ctxt, struct lu_object *o)
```

1. Description:
   This API is used to free the object.
2. Parameters:
   - ctxt: the context of the operation.
   - o: the object to be freed
3. Return values:

Return 0 when successs, other value when failed.

```
static int mdd_object_print(const struct lu_context *ctxt,
                            struct seq_file *f, const struct lu_object *o)
```

1. Description:
   This API is used to print the object.
2. Parameters:
   - ctxt: the context of the operation.
   - seq_file: the seq_file of the print.
   - o: the object to be print.
3. Return values:

Return 0 when successs, other value when failed.

```
static int mdd_object_exists(const struct lu_context *ctx, struct lu_object *o)
```

1. Description:
   This API is used to check whether the object exists in the storage
2. Parameters:
   - ctx: the context of the operation.
   - o: the object to be checked.
3. Return values:

Return true if exist, false if not exist.

3.2 MDD metadata operations

3.2.1 MDD metadata device operation

```
struct md_device_operations mdd_ops = {
        .mdo_root_get       = mdd_root_get,
        .mdo_statfs         = mdd_statfs,
};
static int mdd_root_get(const struct lu_context *ctx, struct md_device *m, struct   lu_
```

1. Description:

   This API is used to get root fid of mdd.

2. Parameters:

   - ctx: the context of the operation.
   - md: the mdd device.
   - f: the root fid will be filled here.

3. Return values:

   0 means success, other value indicate the error.

```
static int mdd_statfs(const struct lu_context *ctx,
                      struct md_device *m, struct kstatfs *sfs)
```

1. Description:

   This API is used to get stat info of the bottom filesystem.

2. Parameters:

   - ctx: the context of the operation.
   - m: the mdd device.
   - kstatfs: the stat info will be filled here.

3. Return values:

   0 means success, other value indicate the error.

3.2.2 MDD metadata dir operations

```
static struct md_dir_operations mdd_dir_ops = {
        .mdo_lookup        = mdd_lookup,
        .mdo_create        = mdd_create,
        .mdo_rename        = mdd_rename,
        .mdo_link          = mdd_link,
        .mdo_name_insert   = mdd_name_insert,
        .mdo_name_remove   = mdd_name_delete,
        .mdo_rename_tgt    = mdd_rename_tgt,
```

```
};
static int mdd_lookup(const struct lu_context *ctxt, struct md_object *pobj,
                      const char *name, struct lu_fid* fid)
```

1. Description:

   This API is used to lookup the fid according to the name.

2. Parameters:

   - ctx: the context of the operation.
   - pobj: the parent object.
   - f: the name of the child.
   - fid: the fid found will be filled here.

3. Return values:

   0 means success, other value indicate the error.

```
static int mdd_create(const struct lu_context *ctxt,
                      struct md_object *pobj, const char *name,
                      struct md_object *child, struct lu_attr* attr)
```

1. Description:

   This API is used to create a object by name, and insert the name index into the namespace of parent.

2. Parameters:

   - ctxt: the context of the operation.
   - attr: the created dir attribute.
   - pobj: the parent obj.
   - name: the object name.
   - child: the object to be created.

3. Return values:

   0 means success, other value indicate the error.

```
static int mdd_rename(const struct lu_context *ctxt
                  struct md_object *spobj, struct md_object *tpobj,
                  struct md_object *sobj, const char *sname,
                  struct md_object *tobj, const char *tname);
```

1. Description:

   This API is used to rename a object.

2. Parameters:

- spobj: source parent object in rename.
- tpobj: target parent object in rename.
- sobj: source object in rename.
- tobj: target object in rename.
- sname: source object name.
- tname: target object name.
- ctxt: the context of this API.

3. Return values:

   0 means success, other value indicate the error.

```
static int mdd_link(const struct lu_context *ctxt, struct md_object *tgt_obj,
                        struct md_object *src_obj, const char *name)
```

1. Description:

   This API is used to link an object according to the args.

2. Parameters:

   - tgt_obj: parent dir for the new link.
   - src_obj: the source object in the link.
   - name: the name of the created link.
   - ctxt: the context of the link

3. Return values:

   0 means success, other value indicate the error.

```
static int mdd_unlink(const struct lu_context *ctxt, struct md_object *pobj,
                       struct md_object *cobj, const char *name)
```

1. Description:

   This API is used to unlink an object according to the args.

2. Parameters:

   - pobj: the parent object in the unlink.
   - cobj: the object to be unlink.
   - name: the name of the unlink object.
   - ctxt: the context of the link

3. Return values:

   0 means success, other value indicate the error.

```
static int mdd_name_insert(const struct lu_context *ctxt, struct md_object *pobj,
                           const char *name, const struct lu_fid *fid)
static int mdd_name_delete(const struct lu_context *ctxt, struct md_object *pobj,
                           const char *name)
```

1. Description:

   This API is used to insert/delete name/fid into the pobj index.

2. Parameters:

   - ctxt: the context of the operation.
   - pobj: the parent obj.
   - fid: the fid to be insert.
   - name: the name to be insert/delete.

3. Return values:

   0 means success, other value indicate the error.

```
static int mdd_name_rename(const struct lu_context *ctxt, struct md_object *pobj,
                           struct md_object *sobj, struct md_object *tobj,
                           const char *name);
```

1. Description:

   This API is used to add new name into dir and remove old object if exists.

2. Parameters:

   - ctxt: the context of the operation.
   - pobj: the parent obj.
   - sobj: the source object.
   - tobj: the target object.
   - name: the name to be insert/delete.

3. Return values:

   0 means success, other value indicate the error.

3.2.3 MDD metadata object operation

```
static struct md_object_operations mdd_obj_ops = {
        .moo_attr_get      = mdd_attr_get,
        .moo_attr_set      = mdd_attr_set,
        .moo_xattr_get     = mdd_xattr_get,
        .moo_xattr_set     = mdd_xattr_set,
        .moo_object_create  = mdd_object_create,
        .moo_ref_add       = mdd_ref_add,
        .moo_ref_del       = mdd_ref_del
};
static int mdd_attr_get(const struct lu_context *ctxt,
                        struct md_object *obj, struct lu_attr *attr)
static int mdd_attr_set(const struct lu_context *ctxt,
                        struct md_object *obj, struct lu_attr *attr)
```

1. Description:

   This API is used to get/set attr to the object.

2. Parameters:

   - obj: the obj will be get/set attr.
   - attr: the attr to be get/set.
   - ctxt: the context of the API.

3. Return values:

   0 means success, other value indicate the error.

```
static int mdd_xattr_get(const struct lu_context *ctxt, struct md_object *obj,
                         void *buf,int buf_len, const char *name)
static int mdd_xattr_set(const struct lu_context *ctxt, struct md_object *obj,
                         void *buf, int buf_len, const char *name)
```

1. Description:

   These API are used to set/get xattr of the object

2. Parameters:

   - obj: the object will be set/get xattr
   - name: the xattr name.
   - buf: the xattr buf.
   - buf_len: the length of the buf.
   - ctxt: the context of the API.

3. Return values:

   0 means success, other value indicate the error.

```
int mdd_object_create(struct md_object *pobj, struct md_object *child,
                      struct context *uctxt);
```

1. Description:

   This API is used to create an object under the parent pobj.

2. Parameters:

   - pobj: the parent object.
   - child: the object being created.
   - uctxt: the context of the create

3. Return values:

   0 means success, other value indicate the error.

```
static int mdd_ref_add(struct lu_context *ctx, struct md_object *object)
static int mdd_ref_del(struct lu_context *ctx, struct md_object *object)
```

1. Description:

   This API is used to increase/decrease the nlink of the object.

2. Parameters:

   - obj: the object.
   - ctx: the context of this operation.

3. Return values:

   0 means success, other value indicate the error.

4 Use Cases

4.1 MDD device operation

4.1.1 mdd_device operation

1. mdd_object_alloc

   In MDS server stack, the upper layer of MDD(CMM) will call this API to allocate the object for MDD layer.

2. mdd_process_config

   When MDS metadata stack init, the upper layer will call this API to process the config log (for example LCFG_SETUP).

4.1.2 mdd_object operation

These mdd object operations will be used by server stack to init/release/print the object.

4.2 MDD metadata operation

4.2.1 MDD metadata device operation

1. mdd_root_get

   The upper layer will call this API to get the root fid of the MD device.

2. mdd_statfs

   The upper layer will call this API to get the statinfo of the bottom file system.

4.2.2 MDD metadata dir operation

These metadata dir operations will be used to handle those metadata requests for dir objects. After MDT unpack the request, the CMM will check whether the operation should be handled locally, if it is, these MDD metadata dir API will be called to process those metadata related operation.

4.2.3 MDD metadata object operation

These metadata object operations will be used to handle those metadata requests(get/set attr, get/set xattr) for all the objects.

5 Logic Specification

5.1 Data structure

According to the new layer of MDS, mdd is a metadata device, which is based on the top of dt device.

```
struct mdd_device {
    struct md_device mdd_md_dev;    /* common md device*/
    struct dt_device *mdd_child;  /* lower device*/
    ......  /*other mdd device specific stuff*/
}
struct md_device {
    struct lu_device md_lu_dev;             /*common lu device*/
    struct md_device_operations *md_ops;    /*md device operations*/
    ..... /*other metadata device specific stuff*/
};
struct dt_device {
    struct lu_device  dd_lu_dev;
```

```
        struct dt_device_operations *dd_ops;
        ..........
};
```

As a lustre metadata device, MDD must also connect with lov to handle those OSS stuff, so it should include a lov device, and the details will be discussed in another DLD. Note: mdd will also handling some llog work, and the details will be discussed in other DLD. In the new MDS server stack, for each object, it has layer-specific objects, which are linked together according to the position of the each layer.

```
struct mdd_object {
     struct md_object  mod_obj;
     ..... /*mdd specificial stuff*/
};
struct md_object {
     struct lu_object mo_lu; /* common lustre object*/
     ..... /*other metadata specific stuff*/
};
struct dt_object {
     struct lu_object  do_lu;
     struct dt_object_operations *do_ops;
     struct dt_body_operations   *do_body_ops;
     struct dt_index_operations  *do_index_ops;
};
```

5.2 MDD API

5.2.1 MDD objects api

MDD objects API is used to create/destory the mdd objects for the metadata stack.

1) mdd_object_alloc

Allocate the mdd object for mdd layer, and init the mdd_object, where the mdd_object_init will be called to init the object.

2) mdd_object_init

Init the mdd object, link the mdd objects to the top layer object, and allocate the object of dt layer.

3) mdd_object_free

Free the mdd object, and cleanup the objects for mdd.

4)mdd_object_release

Release the mdd object, and it is called when last active reference to the object is released.

5) mdd_object_print

Print the mdd object.

5.2.2 MDD MD API

With the support of DT API, each MDD API only needs to care about metadata related stuff, and make MDD working on top of DT.

In the following API, mdd_lock/unlock will be discussed in State specification, and transaction will be discussed in other DLD. Note: some simple MDD API are implemented only by called related DT API, for example mdd_ref_add/del, so we will not discussed here.

1) Get root fid

```
int mdd_root_get(struct md_device *m, struct lu_fid *f)
    {
          struct mdd_device *mdd = lu2mdd_dev(&m->md_lu_dev);
           *f = mdd->mdd_root_fid;
    }
```

Here, mdd_root_fid is set in mdd_mount, which is called in mds stack init.

```
int mdd_mount(const struct lu_context *ctx, struct mdd_device *mdd)
   {
        int result;
        struct dt_object *root;

        root = dt_store_open(ctx, mdd->mdd_child, mdd_root_dir_name,
                             &mdd->mdd_root_fid);
        if (!IS_ERR(root)) {
                LASSERT(root != NULL);
                lu_object_put(ctx, &root->do_lu);
                result = 0;
        } else
                result = PTR_ERR(root);
        return result;
   }
```

2) Create method.

```
static int mdd_create(const struct lu_context *ctxt, struct lu_attr* attr,
```

```
                        struct md_object *pobj, const char *name,
                        struct md_object *child)
{
        struct mdd_device *mdd = mdo2mdd(pobj);
        struct mdd_object *mdo = mdo2mddo(pobj);
        struct thandle *handle;
        int rc = 0;
        ENTRY;
        mdd_txn_param_build(ctxt, &MDD_TXN_MKDIR);
        handle = mdd_trans_start(ctxt, mdd);
        if (IS_ERR(handle))
                RETURN(PTR_ERR(handle));
        mdd_lock(ctxt, mdo, DT_WRITE_LOCK);
        rc = __mdd_object_create(ctxt, mdo2mddo(child), attr, handle);
        if (rc)
               GOTO(cleanup, rc);
        rc = __mdd_index_insert(ctxt, mdo, lu_object_fid(&child->mo_lu),
                                name, handle);
        if (rc)
                GOTO(cleanup, rc);
cleanup:
        mdd_unlock(ctxt, mdo, DT_WRITE_LOCK);
        mdd_trans_stop(ctxt, mdd, handle);
        RETURN(rc);
}
```

3) Rename method

```
static int mdd_rename(const struct lu_context *ctxt, struct md_object *src_pobj,
                      struct md_object *tgt_pobj, struct md_object *sobj,
                      const char *sname, struct md_object *tobj, const char *tname)
{
     struct mdd_device *mdd = mdo2mdd(src_pobj);
     struct mdd_object *mdd_spobj = mdo2mddo(src_pobj);
     struct mdd_object *mdd_tpobj = mdo2mddo(tgt_pobj);
     struct mdd_object *mdd_sobj = mdo2mddo(sobj);
     struct mdd_object *mdd_tobj = mdo2mddo(tobj);
     int rc;
     struct thandle *handle;
     mdd_txn_param_build(ctxt, &MDD_TXN_RENAME);
     handle = mdd_trans_start(ctxt, mdd);
     if (IS_ERR(handle))
           RETURN(PTR_ERR(handle));
     mdd_rename_lock(mdd, mdd_spobj, mdd_tpobj, mdd_sobj, mdd_tobj);
     rc = __mdd_index_delete(ctxt, mdd, mdd_spobj, mdd_sobj, sname, handle);
```

```
        if (rc)
                GOTO(cleanup, rc);
        rc = __mdd_index_delete(ctxt, mdd, mdd_tpobj, mdd_tobj, tname, handle);
        if (rc)
                GOTO(cleanup, rc);
        rc = __mdd_index_insert(ctxt, mdd_spobj, lu_object_fid(&tobj->mo_lu),
                                tname, handle);
        if (rc)
                 GOTO(cleanup, rc);
        if (mdd_object_exists(ctxt, &tobj->mo_lu)) { /*check whether the object exists*/
                rc = __mdd_object_destroy(ctxt, mdd_tobj, handle);
                if (rc)
                        GOTO(cleanup, rc);
        }
    cleanup:
        if (rc) {
                /* error handling*/
        }
        mdd_rename_unlock(mdd, mdd_spobj, mdd_tpobj, mdd_sobj, mdd_tobj);
        mdd_trans_stop(ctxt, mdd, handle);
        RETURN(rc);
    }
```

MDD_rename_lock will be discussed in other DLD.

4) Link method

```
    static int mdd_link(const struct lu_context *ctxt, struct md_object *tgt_obj,
                        struct md_object *src_obj, const char *name)
    {
           struct mdd_object *mdd_tobj = mdo2mddo(tgt_obj);
           struct mdd_object *mdd_sobj = mdo2mddo(src_obj);
           struct mdd_device *mdd = mdo2mdd(src_obj);
           struct thandle *handle;
           int rc, nlink;
           ENTRY;
           mdd_txn_param_build(ctxt, &MDD_TXN_LINK);
           handle = mdd_trans_start(ctxt, mdd);
           if (IS_ERR(handle))
                    RETURN(PTR_ERR(handle));
           mdd_lock2(ctxt, mdd_tobj, mdd_sobj);
           rc = __mdd_index_insert(ctxt, mdd_tobj, lu_object_fid(&src_obj->mo_lu),
                                   name, handle);
           if (rc)
                   GOTO(exit, rc);
           rc = mdd_ref_add(ctxt, src_obj);
```

```
exit:
        mdd_unlock2(ctxt, mdd_tobj, mdd_sobj);
        mdd_trans_stop(ctxt, mdd, handle);
        RETURN(rc);
}
```

5) Attr get/set method

```
static int mdd_attr_get(struct md_object *obj, void *buf, int buf_len,
                        const char *name,struct context *uctxt)
{
        struct mdd_object *mdd_obj = mdo2mddo(obj);
        struct mdd_device *mdd = mdo2mdd(obj);
        int rc;

        mdd_object_get(mdd, mdd_obj);
        rc = mdd_child_ops(mdd)->osd_attr_get(mdd_object_child(mdd_obj), buf,
                                              buf_len, name, uctxt);
        mdd_object_put(mdd, mdd_obj);
        RETURN(rc);
}
static int __mdd_attr_set (struct mdd_device *mdd, struct mdd_object *obj,
                           void *buf,int buf_len, const char *name,
                           struct context *uc_context, void *handle)
{
        return mdd_child_ops(mdd)->osd_attr_set(mdd_object_child(obj), buf,
                                                buf_len, name, uc_context,
                                                handle);
}
static int mdd_attr_set(struct md_object *obj, void *buf, int buf_len,
                        const char *name, struct context *uctxt)
{
        struct mdd_device *mdd = mdo2mdd(obj);
        void *handle = NULL;
        int  rc;
        handle = mdd_trans_start(mdd, mdo2mddo(obj));
        if (!handle)
                 RETURN(-ENOMEM);
        rc = __mdd_attr_set(mdd, mdo2mddo(obj), buf, buf_len, name, uctxt,
                            handle);
        mdd_trans_stop(mdd, handle);
        RETURN(rc);
}
```

6) Index insert/delete

In these API, mdd will call dt index methods to insert/delete index in MDD. For example, index delete API prototype are

```
static int __mdd_index_delete(const struct lu_context *ctxt,
                              struct mdd_device *mdd, struct mdd_object *pobj,
                              struct mdd_object *obj, const char *name,
                              struct thandle *handle)
{
    int rc;
    struct dt_object *next = mdd_object_child(pobj);
    ENTRY;
    mdd_lock2(ctxt, pobj, obj);
    rc = next->do_index_ops->dio_delete(ctxt, next, (struct dt_key *)name, handle);
    mdd_unlock2(ctxt, pobj, obj);
    RETURN(rc);
}
static int mdd_index_delete(const struct lu_context *ctxt, struct md_object *pobj,
                            struct md_object *obj, const char *name)
{
    struct mdd_object *mdd_pobj = mdo2mddo(pobj);
    struct mdd_object *mdd_obj = mdo2mddo(obj);
    struct mdd_device *mdd = mdo2mdd(obj);
    struct thandle *handle;
    int rc;
    ENTRY;

    mdd_txn_param_build(ctxt, &MDD_TXN_INDEX_DELETE);
    handle = mdd_trans_start(ctxt, mdd);
    if (IS_ERR(handle))
          RETURN(PTR_ERR(handle));
    rc = __mdd_index_delete(ctxt, mdd, mdd_pobj, mdd_obj, name, handle);
    mdd_trans_stop(ctxt, mdd, handle);
    RETURN(rc);
}
```

6 State Specification

6.1 Objects life cycle.

According to the new MDS layer, the object is lookup and created in MDT layer before calling API of the lower layer, where each layer objects are created and linked to the top list of the object, and the refcount of the object will be set to 1. Then in lu_object_put, the refcount of the object is decreased. If the refcount reach zero, the release method of each layer will be called. In MDD

release API, the nlink will be checked too. If the nlink is zero, the object should be destroyed.

```
int __mdd_object_destroy(const struct lu_context *ctxt, struct mdd_object *obj,
                                 struct thandle *handle)
         {
                 struct dt_object  *next = mdd_object_child(obj);
                 int rc = 0;
                 ENTRY;
                 if (mdd_object_exists(ctxt, &obj->mod_obj.mo_lu))
                        rc = next->do_ops->do_object_destroy(ctxt, next, handle);
                 LASSERT(ergo(rc == 0, !mdd_object_exists(ctxt, &obj->mod_obj.mo_lu)));
                 RETURN(rc);
        }
        int mdd_object_destroy(const struct lu_context *ctxt, struct md_object *obj)
         {
               struct mdd_device *mdd = mdo2mdd(obj);
               struct mdd_object *mdd_obj = mdo2mddo(obj);
               struct thandle *handle;
               int rc ;
               ENTRY;

               mdd_txn_param_build(ctxt, &MDD_TXN_OBJECT_DESTROY);
               handle = mdd_trans_start(ctxt, mdd);
               if (IS_ERR(handle))
                    RETURN(PTR_ERR(handle));
               mdd_lock(ctxt, mdd_obj, DT_WRITE_LOCK);
               if (open_orphan(mdd_obj))
                         rc = mdd_add_orphan(mdd, mdd_obj, handle);
               else {
                         rc = __mdd_object_destroy(ctxt, mdd_obj, handle);
                         if (rc == 0)
                               rc = mdd_add_unlink_log(mdd, mdd_obj, handle);
               }
               mdd_unlock(ctxt, mdd_obj, DT_WRITE_LOCK);
               mdd_trans_stop(ctxt, mdd, handle);
               RETURN(rc);
         }

         void mdd_object_release(struct lu_object *o)
         {
                struct mdd_device *mdd = lu2mdd_dev(o->lo_dev);
                struct mdd_object *obj = mdd_obj(o);
                int rc = 0;
```

```
                rc = mdd_attr_get(&obj->mod_obj, &nlink, sizeof(nlink),
                                  "NLINK", NULL);
                if (!rc)
                        RETURN(-EINVAL);
                if (nlink == 0)
                             rc = mdd_object_destroy(&obj->mod_obj);

                RETURN(rc);
        }
```

In mdd_object_destroy methods, where the open refcount will be checked. If it is not zero, the object will be added to the orphan list. If it is zero, the objects will be destroyed. and the unlink log record will be added in unlink log (which will be discussed in other DLD).

6.2 MDD objects Locking

MDD objects locking should be implemented with the support of dt locking method. In OSD object, we define an oo_iem semaphore, which will be used to protect the object.

```
static void mdd_lock(const struct lu_context *ctxt,
                     struct mdd_object *obj, enum dt_lock_mode mode)
{
        struct dt_object  *next = mdd_object_child(obj);
        next->do_ops->do_object_lock(ctxt, next, mode);
}
static void mdd_unlock(const struct lu_context *ctxt,
                       struct mdd_object *obj, enum dt_lock_mode mode)
{
       struct dt_object  *next = mdd_object_child(obj);
       next->do_ops->do_object_unlock(ctxt, next, mode);
}
```