Object handling in uOSS DLD

Mikhail Pershin

2007-12-06

1 Requirements

Implement object management like precreation, create-on-write and destroy in new uOSS obdfilter code.

2 Functional specification

2.1 General approach

uOSS is working in *compatible mode* - the MDS takes object id from pool of precreated object IDs. OSS doesn't create object during precreate but manage the last_id only. CROW is used to create objects during write operations.

2.2 IDIF

For compatibility with FID-based API and future use the IDIF is defined. IDIF is the object id and group packed to FID. This is only needed for using struct lu_fid in API and is temporary thing until client will support FIDs for OST objects.

Several methods are used to pack/unpack IDIF. Strictly speaking we don't use group greater than FILTER_GROUP_MDS0 because there is no CMD demands for uOSS and when CMD will become production the FIDs will be used, so group may be not used for IDIF building.

2.2.1 lu_idif_build()

Prototype: `void lu_idif_build(obd_id id, obd_group group, struct lu_fid *fid);`

Description: build IDIF from id and group and save it into *fid.

2.2.2 lu_idif_id()

Prototype: `obd_id idif_id(struct lu_fid *fid);`

Description: unpack object ID from IDIF

2.2.3 lu_idif_group()

Prototype: `obd_group idif_id(struct lu_fid *fid);`

Description: unpack object group from IDIF

2.2.4 lu_idif_resid()

IDIF is the FID and the function `fid_build_reg_res_name(const struct lu_fid *f, struct ldlm_res_id *name)` could be be used for that but we can't do that until client will do the same. Therefore we need another resource name creation function:

Prototype: `lu_idif_resid(const struct lu_fid *f, struct ldlm_res_id *name)`

Description: create ldlm_res_id in the same form as client.

2.3 Reservation of object IDs

2.4 Create on write

2.4.1 Create object on write

Prototype: `struct filter_object *filter_object_find_or_create(const struct lu_env *env, struct filter_device *ofd, const struct lu_fid *fid, struct lu_attr *attr);`

Description: function creates object if it is not yet created.

2.4.2 Accessing not yet created object

Several functions can access object that is not yet created. In that case we should handle such cases like the object exists with 0 size and answer with correct result instead of -ENOENT. The affected functions are the following:

`filter_getattr()`

`filter_preprw_read()`

`filter_commitrw_read()`

2.5 Object destroy

Object destroy can be invoked from `filter_destroy()` or from `filter_create()` with `OBD_FL_DELORPHAN` flag. Both should use IDIF of object to destroy it so the new function is defined for that:

Prototype: `filter_destroy_by_fid(struct lu_env *env, struct filter_device *ofd, struct lu_fid *fid);`

Description: take object by fid and call destroy on it. Take care about ldlm part too.

3 Use cases

3.1 Object ID reservation

- *filter_create() -> filter_last_id_update() -> filter_write_last_objid()*

  Filter set new last_id value during both precreate and recreate of objects

3.2 Create on write cases

- *filter_setattr(id)/filter_punch(id) -> lu_idif_build(id) -> filter_object_find_or_create(idif)*

  The new object is created during setattr/punch.

- *filter_preprw_write(id) -> lu_idif_build(id) -> filter_object_find_or_create(idif)*

  During `preprw_write()` the object is created.

- *filter_preprw_read(id) -> lu_idif_build(id) -> filter_object_find(idif)*

  The filter_object_find() may return filter_object that doesn't exist on disk yet. For such case the empty object is simulated.

- *filter_getattr()*

  These methods can found that object is not yet created, but should simulate empty object.

3.3 Destroy

- *filter_destroy(id) -> lu_idif_build(id) -> filter_destory_by_fid(idif)*

  Destroy the object by given ID. Should care about llog_cancel cookie.

3.4 Orphan destroy

- *filter_create([flags OBD_FL_DELORPHAN]) -> filter_destroy_by_id(idif)*

  MDS require orphans to be destroyed. These objects were created on OSS but lost on MDS and should be deleted. Llog cookies are not needed.

4 Logic specification

4.1 IDIF management

```
int lu_idif_build(obd_id id, obd_group gr, struct lu_fid *fid)
{
        LASSERT((id >> 48) == 0);
        fid->f_seq = (1 << 33) | (id >> 32);
        fid->f_oid = (__u32)(id & 0xffffffff);
        fid->f_ver = gr;
}
static inline obd_id lu_idif_id(const struct lu_fid *fid)
{
        return ((fid->f_seq & 0xffff) << 32) | fid->f_oid;
}
static inline obd_group lu_idif_group(struct lu_fid *fid)
{
        return fid->f_ver;
}
static inline struct ldlm_res_id * lu_idif_resid(const struct lu_fid *f,
                                                 struct ldlm_res_id *name)
{
        name->name[LUSTRE_RES_ID_SEQ_OFF] = lu_idif_id(f);
        name->name[LUSTRE_RES_ID_OID_OFF] = 0;
        name->name[LUSTRE_RES_ID_VER_OFF] = lu_idif_group(f);
        name->name[LUSTRE_RES_ID_HSH_OFF] = 0;
        return name;
}
```

4.2 Last_id management

4.2.1 IDs reservation

```
int filter_create(struct lu_env *env, struct obd_export *exp, struct obdo *oa,
                         struct lov_stripe_md **ea, struct obd_trans_info *oti)
{
```

```
struct filter_export_data *fed = &exp->exp_filter_data;
struct obd_device *obd = exp->exp_obd;
struct filter_device *ofd = filter_dev(obd->obd_lu_dev);
int rc = 0, diff;
ENTRY;
LASSERT(ea == NULL);
CDEBUG(D_INFO, "filter_create(od->o_gr="LPU64",od->o_id="LPU64")\n",
      oa->o_gr, oa->o_id);
if ((oa->o_valid & OBD_MD_FLFLAGS) &&
    (oa->o_flags & OBD_FL_RECREATE_OBJS)) {
        if (oa->o_id > filter_last_id(ofd, oa->o_gr)) {
                CERROR("recreate objid "LPU64" > last id "LPU64"\n",
                       oa->o_id, filter_last_id(ofd, oa->o_gr));
                RETURN(-EINVAL);
        }
        /* do nothing because we create objects during first write */
} else if ((oa->o_valid & OBD_MD_FLFLAGS) &&
           (oa->o_flags & OBD_FL_DELORPHAN)){
        /* destroy orphans */
        if (oti->oti_conn_cnt < exp->exp_conn_cnt) {
                CERROR("%s: dropping old orphan cleanup request\n",
                       obd->obd_name);
                RETURN(0);
        }
        /* This causes inflight precreates to abort and drop lock */
        set_bit(group, &ofd->ofd_destroys_in_progress);
        mutex_down(&ofd->ofd_create_locks[group]);
        if (!test_bit(group, &ofd->ofd_destroys_in_progress)) {
                CERROR("%s:["LPU64"] destroys_in_progress already cleared\n",
                       exp->exp_obd->obd_name, group);
                GOTO(out, rc = 0);
        }
        diff = oa->o_id - filter_last_id(ofd, group);
        CDEBUG(D_HA, "filter_last_id() = "LPU64" -> diff = %d\n",
              filter_last_id(ofd, group), diff);
        if (-diff > OST_MAX_PRECREATE) {
                /* FIXME: should reset precreate_next_id on MDS */
                rc = 0;
        } else if (diff < 0) {
                rc = filter_orphans_destroy(env, exp, oa);
        } else {
                /* XXX: Used by MDS for the first time! */
                clear_bit(group, &ofd->ofd_destroys_in_progress);
        }
} else {
        mutex_down(&ofd->ofd_create_locks[group]);
```

```
                if (oti->oti_conn_cnt < exp->exp_conn_cnt) {
                        CERROR("%s: dropping old precreate request\n",
                               obd->obd_name);
                        GOTO(out, rc = 0);
                }
                /* only precreate if group == 0 and o_id is specfied */
                if (group < FILTER_GROUP_MDS0 || oa->o_id == 0)
                        diff = 1; /* shouldn't we create this right now? */
                else
                        diff = oa->o_id - filter_last_id(ofd, group);
        }
        if (diff > 0) {
                rc = filter_last_id_update(env, ofd, group, &diff);
                oa->o_id = filter_last_id(ofd, group);
                LASSERT(oa->o_gr == group);
                oa->o_valid = OBD_MD_FLID | OBD_MD_FLGROUP;
        }
out:
        mutex_up(&ofd->ofd_create_locks[group]);
        return rc;
}
static int filter_last_id_update(struct lu_env *env, struct filter_device *ofd,
                                 obd_gr group, int *num)
{
        struct obd_device *obd = filter_obd(ofd);
        int err = 0, rc = 0, recreate_obj = 0, i;
        obd_id last_id = filter_last_id(ofd, group);
        obd_id next_id = last_id + *num;
        ENTRY;
        /* TODO: check we have free space. Need DMU support */
        CDEBUG(D_HA, "%s: reserve %d objects in group "LPU64" at "LPU64"\n",
               obd->obd_name, *num, group, filter_last_id(ofd, group));
        LASSERT(next_id > last_id);
        filter_set_last_id(ofd, next_id, group);
        err = filter_write_last_objid(env, ofd, group, 0);
        if (err)
                CERROR("unable to write lastobjid but file created\n");
        CDEBUG(D_HA, "%s: reserved %d objects for group "LPU64": "LPU64"\n",
               obd->obd_name, *num, group, filter_last_id(ofd, group));
        RETURN(rc);
}
```

4.2.2 Orphans destroy

```
static int filter_orphans_destroy(struct lu_env *env, struct filter_device *ofd,
                                  struct obdo *oa)
{
        struct lu_fid fid;
        obd_id last, id;
        int rc;
        ENTRY;
        LASSERT(oa);
        LASSERT(oa->o_gr != 0);
        LASSERT(oa->o_valid & OBD_MD_FLGROUP);
        LASSERT(mutex_try_down(&ofd->ofd_create_locks[oa->o_gr]) != 0);
        if (!test_bit(oa->o_gr, &ofd->ofd_destroys_in_progress)) {
                CERROR("%s:["LPU64"] destroys_in_progress already cleared\n",
                       exp->exp_obd->obd_name, oa->o_gr);
                RETURN(0);
        }
        last = filter_last_id(ofd, oa->o_gr);
        CWARN("%s: deleting orphan objects from "LPU64" to "LPU64"\n",
               exp->exp_obd->obd_name, oa->o_id + 1, last);

        for (id = last; id > oa->o_id; id--) {
                lu_idif_build(id, oa->o_gr, &fid);
                rc = filter_destroy_by_fid(env, ofd, &fid);
                if (rc && rc != -ENOENT) /* this is pretty fatal... */
                        CEMERG("error destroying precreate objid "LPU64": %d\n",
                               id, rc);
                filter_set_last_id(ofd, id - 1, oa->o_gr);
                /* update last_id on disk periodically so that if we restart
                 * we don't need to re-scan all of the just-deleted objects. */
                if ((id & 511) == 0)
                        filter_write_last_objid(env, ofd, oa->o_gr, 0);
        }
        CDEBUG(D_HA, "%s: after destroy: set last_objids["LPU64"] = "LPU64"\n",
               exp->exp_obd->obd_name, oa->o_gr, oa->o_id);
        rc = filter_write_last_objid(env, ofd, oa->o_gr, 1);
        clear_bit(oa->o_gr, &ofd->ofd_destroys_in_progress);
        RETURN(rc);
}
```

4.3 CROW

4.3.1 Read/write

```
int filter_preprw(struct lu_env *env, int cmd, struct obd_export *exp,
                  struct obdo *oa, int objcount, struct obd_ioobj *obj,
                  int niocount, struct niobuf_remote *nb,
                  int *nr_local, struct niobuf_local *res,
                  struct obd_trans_info *oti, struct lustre_capa *capa)
{
        struct filter_device *ofd = filter_exp(exp);
        struct filter_object *fo;
        struct lu_attr attr;
        struct lu_fid fid;
        LASSERT(objcount == 1);
        LASSERT(obj->ioo_bufcnt > 0);
        lu_idif_build(obj->ioo_id, obj->ioo_gr, &fid);
        if (cmd == OBD_BRW_WRITE) {
                rc = filter_auth_capa(exp, NULL, obdo_mdsno(oa), capa,
                                      CAPA_OPC_OSS_WRITE);
                if (rc == 0) {
                        attr.la_valid = LA_MODE;
                        attr.la_mode = S_IFREG | 0666;
                        rc = filter_preprw_write(env, ofd, &fid, &attr, niocount,
                                                 nb, nr_local, res);
        } else if (cmd == OBD_BRW_READ)
                rc = filter_auth_capa(exp, NULL, obdo_mdsno(oa), capa,
                                      CAPA_OPC_OSS_READ);
                if (rc == 0) {
                        rc = filter_preprw_read(env, ofd, &fid, &attr, niocount,
                                                nb, nr_local, res);
                        obdo_from_la(oa, &attr, LA_ATIME);
                }
        } else {
                LBUG();
                rc = -EPROTO;
        }
        return rc;
}
static int
filter_preprw_write(struct lu_env *env, struct filter_device *ofd,
                    struct lu_fid *fid, struct lu_attr *la,
                    int niocount, struct niobuf_remote *nb,
                    int *nr_local, struct niobuf_local *res)
{
        struct filter_object *fo;
        int i, j, rc = 0;
        ENTRY;
        LASSERT(env != NULL);
        fo = filter_object_find_or_create(env, ofd, fid, la);
        if (IS_ERR(fo))
                RETURN(PTR_ERR(fo));
        LASSERT(fo != NULL);
        LASSERT(filter_object_exists(fo));
        /* parse remote buffers to local buffers and prepare the latter */
        for (i = 0, j = 0; i < niocount; i++) {
                rc = filter_bufs_get(env, fo, nb + i, res + j);
                LASSERT(rc > 0);
                LASSERT(rc < PTLRPC_MAX_BRW_PAGES);
                /* correct index for local buffers to continue with */
                j += rc;
                LASSERT(j <= PTLRPC_MAX_BRW_PAGES);
        }
        *nr_local = j;
        LASSERT(*nr_local > 0 && *nr_local <= PTLRPC_MAX_BRW_PAGES);
        rc = filter_bufs_write_prep(env, fo, res, *nr_local);
        filter_object_put(env, fo);
        RETURN(rc);
}
static int
filter_preprw_read(struct lu_env *env, struct filter_device *ofd,
                   struct lu_fid *fid, struct lu_attr *la,
                   int niocount, struct niobuf_remote *nb,
                   int *nr_local, struct niobuf_local *res)
{
        struct filter_object *fo;
        int i, j, rc;
        LASSERT(env != NULL);
        fo = filter_object_find(env, ofd, fid);
        if (IS_ERR(fo))
                RETURN(PTR_ERR(fo));
        LASSERT(fo != NULL);
        if (filter_object_exists(fo)) {
                /* parse remote buffers to local buffers
                   and prepare the latter */
                for (i = 0, j = 0; i < niocount; i++) {
                        rc = filter_bufs_get(env, fo, nb + i, res + j);
                        LASSERT(rc > 0);
                        LASSERT(rc < PTLRPC_MAX_BRW_PAGES);
                        /* correct index for local buffers to continue with */
                        j += rc;
                        LASSERT(j <= PTLRPC_MAX_BRW_PAGES);
                }
                *nr_local = j;
                LASSERT(*nr_local > 0 && *nr_local <= PTLRPC_MAX_BRW_PAGES);
                rc = dt_attr_get(env, filter_object_child(fo), la, BYPASS_CAPA);
                LASSERT(rc == 0);
                rc = filter_bufs_read_prep(env, fo, res, *nr_local);
        } else {
                for (i = 0, j = 0; i < niocount; i++) {
                        res[j].file_offset = nb[i].offset;
                        res[j].page_offset = 0;
                        res[j].len = 0;
                        res[j].page = NULL;
                        j++;
                        LASSERT(j <= PTLRPC_MAX_BRW_PAGES);
                }
                *nr_local = j;
                attr.la_size = 0;
                attr.la_atime = 0;
                attr.la_ctime = 0;
                attr.la_mtime = 0;
        }
        filter_object_put(env, fo);
        RETURN(rc);
}
```

4.3.2 Setattr

```
int filter_setattr(const struct lu_env *env, struct obd_export *exp,
                   struct obd_info *oinfo, struct obd_trans_info *oti)
{
        struct filter_device *ofd = filter_exp(exp);
        struct ldlm_namespace *ns = ofd->ofd_namespace;
        struct lu_fid fid;
        struct filter_object *fo;
        struct lu_attr attr;
        ...
        int rc = 0;
        lu_idif_build(oinfo->oi_oa->o_id, oinfo->oi_oa->o_gr, &fid);
        ...
        attr.la_valid = LA_MODE;
        attr.la_mode = S_IFREG | 0666;
        fo = filter_object_find_or_create(env, ofd, &fid, &attr);
        if (IS_ERR(fo)) {
                CERROR("can't find object %lu:%llu\n", fid.f_oid, fid.f_seq);
                RETURN(PTR_ERR(fo));
        }
        rc = filter_attr_set(env, fo, &attr, ...);
        if (rc)
                GOTO(out_unlock, rc);
        }
        idif2resid(&fid, &res_id);
        res = ldlm_resource_get(ns, NULL, &res_id, LDLM_EXTENT, 0);
        if (res != NULL) {
                ns->ns_lvbo->lvbo_update(res, NULL, 0, 0);
                ldlm_resource_putref(res);
        }
        ...
out_unlock:
        filter_object_put(env, fo);
        return rc;
}
```

4.3.3 Getattr

```
static int filter_getattr(struct obd_export *exp, struct obd_info *oinfo)
{
        struct filter_device *ofd = filter_exp(exp);
        struct filter_object *fo;
        struct obd_device *obd = filter_obd(ofd);
        struct lu_fid fid;
        struct lu_attr attr;
        int rc = 0;
        ENTRY;
        rc = filter_auth_capa(exp, NULL, oinfo_mdsno(oinfo),
                              oinfo_capa(oinfo), CAPA_OPC_META_READ);
        if (rc)
                RETURN(rc);
        lu_idif_build(oinfo->oi_oa->o_id, oinfo->oi_oa->o_gr, &fid);
        fo = filter_object_find(env, ofd, &fid);
        if (IS_ERR(fo))
                RETURN(PTR_ERR(fo));
        LASSERT(fo != NULL);
        /* CROW allow object to don't exist */
        if (filter_object_exists(fo)) {
                rc = dt_attr_get(env, filter_object_child(fo), &attr, BYPASS_CAPA);
        } else {
                attr.la_size = 0;
                attr.la_atime = 0;
                attr.la_ctime = 0;
                attr.la_mtime = 0;
        }
        oinfo->oi_oa->o_valid = OBD_MD_FLID;
        obdo_from_la(oinfo->oi_oa, &attr, flags);
        filter_object_put(env, fo);
        RETURN(rc);
}
```

4.3.4 Punch

```
int filter_punch(const struct lu_env *env, struct obd_export *exp,
                 struct obd_info *oinfo, struct obd_trans_info *oti,
                 struct ptlrpc_request_set *rqset)
{
        struct ldlm_res_id res_id;
        struct filter_device *ofd = filter_exp(exp);
        struct ldlm_namespace *ns = ofd->ofd_namespace;
        struct ldlm_resource *res;
        struct lu_attr attr;
        struct lu_fid fid;
        struct filter_object *fo;
        int rc = 0;
        lu_idif_build(oinfo->oi_oa->o_id, oinfo->oi_oa->o_gr, &fid);

        attr.la_valid = LA_MODE;
        attr.la_mode = S_IFREG | 0666;
        fo = filter_object_find_or_create(env, ofd, &fid, &attr);
        if (IS_ERR(fo)) {
                CERROR("can't find object %lu:%llu\n", fid.f_oid, fid.f_seq);
                RETURN(PTR_ERR(fo));
        }
        ...
        filter_object_punch(env, fo, ...);
        idif2resid(&idif, &res_id);
        res = ldlm_resource_get(ns, NULL, &res_id, LDLM_EXTENT, 0);
        if (res != NULL) {
                ns->ns_lvbo->lvbo_update(res, NULL, 0, 0);
                ldlm_resource_putref(res);
        }
        filter_object_put(env, fo);
        RETURN(rc);
 }
```

4.4 Object destroy

```
int filter_destroy(struct lu_env *env, struct obd_export *exp,
                   struct obdo *oa, struct lov_stripe_md *md,
                   struct obd_trans_info *oti, struct obd_export *md_exp)
{
        struct filter_device *ofd = filter_exp(exp);
        struct llog_cookie *fcc = NULL;
        struct lu_fid fid;
        int rc = 0;
        ENTRY;
        if (!(oa->o_valid & OBD_MD_FLGROUP))
                oa->o_gr = 0;
        lu_idif_build(oa->o_id, oa->o_gr, &fid);
        rc = filter_destroy_by_fid(env, ofd, &fid);
        if (rc == -ENOENT) {
                CDEBUG(D_INODE, "destroying non-existent object "LPU64"\n",
                       oa->o_id);
                /* If object already gone, cancel cookie right now */
                if (oa->o_valid & OBD_MD_FLCOOKIE) {
                        struct llog_ctxt *ctxt;
                        fcc = obdo_logcookie(oa);
                        ctxt = llog_get_context(obd, fcc->lgc_subsys + 1);
                        llog_cancel(ctxt, NULL, 1, fcc, 0);
                        fcc = NULL; /* we didn't allocate fcc, don't free it */
                }
        } else {
                /* XXX: no commit callbacks from DMU yet, so cancel cookie immediately
                if (oa->o_valid & OBD_MD_FLCOOKIE) {
                        struct llog_ctxt *ctxt;
                        fcc = obdo_logcookie(oa);
                        ctxt = llog_get_context(obd, fcc->lgc_subsys + 1);
                        llog_cancel(ctxt, NULL, 1, fcc, 0);
                        fcc = NULL; /* we didn't allocate fcc, don't free it */
                }
        }
        RETURN(rc);
}
int filter_destroy_by_fid(struct lu_env *env, struct filter_device *ofd,
                          struct lu_fid *fid)
{
        struct filter_object *fo;
        struct obd_device *obd = exp->exp_obd;
        fo = filter_object_find(env, ofd, fid);
        if (IS_ERR(fo))
```

```
                RETURN(PTR_ERR(fo));
        LASSERT(fo != NULL);
        if (!filter_object_exists(fo))
                GOTO(cleanup, rc = -ENOENT);
        /* XXX: should we check return code below? */
        filter_prepare_destroy(ofd, fid);
        rc = filter_object_destroy(env, fo);
cleanup:
        filter_object_put(env, fo);
        RETURN(rc);
}
static int filter_prepare_destroy(struct filter_device *ofd, const struct lu_fid *fid)
{
        struct lustre_handle lockh;
        int flags = LDLM_AST_DISCARD_DATA, rc;
        struct ldlm_res_id res_id;
        ldlm_policy_data_t policy = { .l_extent = { 0, OBD_OBJECT_EOF } };
        ENTRY;
        /* Tell the clients that the object is gone now and that they should
         * throw away any cached pages. */
        fid_build_reg_res_name(fid, &res_id);
        rc = ldlm_cli_enqueue_local(ofd->ofd_namespace, &res_id, LDLM_EXTENT,
                                    &policy, LCK_PW, &flags, ldlm_blocking_ast,
                                    ldlm_completion_ast, NULL, NULL, 0, NULL,
                                    &lockh);
        /* We only care about the side-effects, just drop the lock. */
        if (rc == ELDLM_OK)
                ldlm_lock_decref(&lockh, LCK_PW);
        RETURN(rc);
}
int filter_object_destroy(struct lu_env *env, struct filter_object *fo)
{
        struct thandle *th;
        int rc = 0;
        ENTRY;
        th = filter_trans_start(env, ofd, 1); /* XXX: real credits */
        LASSERT(th != NULL);
        dt_ref_del(env, filter_object_child(fo), th);
        filter_trans_stop(env, ofd, th);
        RETURN(rc);
}
```