Write up for Quota on HEAD

Fan Yong

2008-08

1 Introduction

Quota feature has been supported in lustre 1.6.x release. Lustre 2.0 release will base on current HEAD which is original from CMD project without quota support. For the compatibility for former lustre release, we should add quota support for current HEAD. This document gives a brief description for how to support quota on current HEAD, mainly includes two parts:

- Port quota related code from b1_6 to HEAD with new MD stack on MDS server.
- Add security support for quota on HEAD, including MDS/OSS capability, remote user, and so on.

2 Nomenclature

- MDS/OSS capability: capability is a piece of non-fabricated data generated by **master service** (on MDS), passed to the client and presented by the client to **slave service** (on OSS and MDS), to authorize an action.
- Remote client (user): it means that the client is in different kerberos domain against server, and is regarded as untrusted. The contrary case is local client, which is in the same kerberos domain as server. A local client can claim to be remote one, but the inverted case is forbidden.

3 specification

3.1 porting quota from b1_6 to HEAD

Compare HEAD with lustre 1.6.x release, there are some architecture differences which affects the quota porting. (Note: security related features are not included, they will be discussed in the next section)

3.1.1 client side change

The most difference on client side is that LMV layer is introduced for CMD support. In lustre 1.6.x release, LLITE layer calls MDC layer methods directly. But in HEAD, it becomes LLITE=>LMV=>MDC. So all the quota related interfaces for original MDC layer should be processed in LMV layer firstly, then to MDC layer. On the other hand, LMV is the abstract layer to control different MDCs under layer, just like LOV layer against OSC layer. Its main work is to determine which MDC should transmit the quota operation request to, specified MDC or broadcast. (Although lustre 2.0 release will not support CMD).

1. lmv_quota_interface

```
quota_interface_t lmv_quota_interface = {
    .quota_ctl   = lmv_quota_ctl,
    .quota_check = lmv_quota_check,
};
```

2. lmv_quota_ctl

```
int lmv_quota_ctl(struct obd_device *unused, struct obd_export *exp, struct obd_qu
```

   For quota_ctl, LMV layer only sends request to quota master (lmv->tgts[0]), and quota master will process with quota slaves.

3. lmv_quota_check

```
int lmv_quota_check(struct obd_device *unused, struct obd_export *exp, struct obd_
```

   For quota_check, LMV layer sends request to all MDTs (all lmv->tgts, consider CMD case in future).

3.1.2 MDS side change

There are much of changes for HEAD on MDS side against lustre 1.6.x release. The greatest impact for quota are new MD stack and new current user identity.

3.1.2.1 new MD stack In HEAD, we use new MD stack of "MDT=>CMM=>MDD=>OSD" layers to replace the original single "MDS" layer for CMD supporting. But MDS layer does not disappear yet, it is used for maintaining the communication with OSS server, the stack trace some like "MDT=>CMM=>MDD=>MDS=>LOV=>OSC". All the quota related process in original MDS layer should be move to other layer(s). What we are facing is to determine which layer(s) they should be ported to.

In lustre 1.6.x release, all the quota control or management interfaces are implemented in LVFS layer, which call VFS methods to operate LDISKFS

quota files directly. They are common interfaces, and shared by MDS server and OSS server. These interfaces are organized by LQUOTA as LQUOTA interfaces, When server initializes, LQUOTA interfaces are registered to related obd_device as "obd_fsops". The stack trace for quota slave on MDS server is "MDS=>LQUOTA=>LVFS=>VFS".

In HEAD, we would like to reverse such LVFS and LQUOTA quota interfaces, and register them to MDS obd_device as "obd_fsops" also. But MDS is only used for communicating with OSS server, it does not know when to trigger the quota process. So we need to define the trigger points for quota process and select which layer(s) should be the entry to MDS for quota process. Currently, MDD is the unique entry for the new MD stack to MDS, we would like to keep the same stack framework for quota. Another reason for selecting MDD as the entry is that MDD is the real layer for implementing the file system (namespace), it accurately knows which file operation need to process quota and which one(s) need not, in spite of CMD case or not.

Quota process with new MD stack can be divided into three sorts: quota environment, quota control, and quota application.

1. quota environment (notify / setup / cleanup / recovery)

   It includes lquota_notify, lquota_setup, lquota_cleanup and lquota_recovery. These interfaces are used for processing quota running environment. They are triggered when system notify / setup / cleanup / recovery event in MDT layer, and need some system parameters from MDT. We defines some new md_device_operations for that:

   - md_device_operations

```
struct md_device_operations {
        ...
        struct md_quota_operations {
                int (*mqo_notify)(const struct lu_env *env,
                                  struct md_device *m);
                int (*mqo_setup)(const struct lu_env *env,
                                 struct md_device *m,
                                 void *data);
                int (*mqo_cleanup)(const struct lu_env *env,
                                   struct md_device *m);
                int (*mqo_recovery)(const struct lu_env *env,
                                    struct md_device *m);
                ...
        } mdo_quota;
};
```

   - cmm_md_ops

```
static struct md_device_operations cmm_md_ops = {
        ...
        .mdo_quota          = {
```

```
                .mqo_notify      = cmm_quota_notify,
                .mqo_setup       = cmm_quota_setup,
                .mqo_cleanup     = cmm_quota_cleanup,
                .mqo_recovery    = cmm_quota_recovery,
                ...
        }
};
```

- mdd_ops

```
struct md_device_operations mdd_ops = {
        ...
        .mdo_quota          = {
                .mqo_notify      = mdd_quota_notify,
                .mqo_setup       = mdd_quota_setup,
                .mqo_cleanup     = mdd_quota_cleanup,
                .mqo_recovery    = mdd_quota_recovery,
                ...
        }
};
```

The new stack trace for quota environment process is "MDT=>CMM=>MDD=>MDS=>LQUOTA=>L

(a) lquota_notify

- mdt_obd_notify

```
static int mdt_obd_notify(struct obd_device *host, struct obd_device *watche
                          enum obd_notify_event ev, void *data)
{
        ...
        case OBD_NOTIFY_CONFIG:
                mdt_allow_cli(mdt, (unsigned long)data);
                /* quota_type has been processed, we can now handle incoming
                next->md_ops->mdo_quota.mqo_notify(NULL, next);
                break;
       ...
}
```

- cmm_quota_notify

```
static int cmm_quota_notify(const struct lu_env *env, struct md_device *m)
{
        /* disable quota for CMD case temporary. */
        if (cmm_dev->cmm_tgt_count)
                RETURN(-EOPNOTSUPP);
        cmm_child_ops(cmm_dev)->mdo_quota.mqo_notify(env, cmm_dev->cmm_child
}
```

- mdd_quota_notify

```
int mdd_quota_notify(const struct lu_env *env, struct md_device *m)
{
        lquota_setinfo(mds_quota_interface_ref, obd, (void *)1);
}
```

- mds_quota_setinfo

```
static int mds_quota_setinfo(struct obd_device *obd, void *data)
{
        struct lustre_quota_ctxt *qctxt = &obd->u.obt.obt_qctxt;
        if (data != NULL)
                QUOTA_MASTER_READY(qctxt);
        else
                QUOTA_MASTER_UNREADY(qctxt);
}
```

(b) lquota_setup

- mdt_init0

```
static int mdt_init0(const struct lu_env *env, struct mdt_device *m,
                     struct lu_device_type *ldt, struct lustre_cfg *cfg)
{
        ...
        rc = mdt_fs_setup(env, m, obd);
        if (rc)
                GOTO(err_capa, rc);
        next = m->mdt_child;
        rc = next->md_ops->mdo_quota.mqo_setup(env, next, lmi->lmi_mnt);
        if (rc)
                GOTO(err_fs_cleanup, rc);
        ...
}
```

- cmm_quota_setup

```
static int cmm_quota_setup(const struct lu_env *env, struct md_device *m, vo
{
        /* disable quota for CMD case temporary. */
        if (cmm_dev->cmm_tgt_count)
                return(-EOPNOTSUPP);
        cmm_child_ops(cmm_dev)->mdo_quota.mqo_setup(env, cmm_dev->cmm_child,
}
```

- mdd_quota_setup

```
static int mdd_quota_setup(const struct lu_env *env, struct md_device *m, vo
```

```
{
        LASSERT(obd->obd_fsops != NULL);
        dt->dd_ops->dt_init_quota_ctxt(env, dt, (void *)obd, data);
        lquota_setup(mds_quota_interface_ref, obd);
}
```

In the new MD stack, we do not want md_device methods to refer to the linux-specific data structures, which makes lustre more portable for other platforms. For lquota_setup case, it needs to process some vfsmount related information. So we introduce new dt_device method to initialize quota in OSD layer.

- struct dt_device_operations

```
struct dt_device_operations {
        /**
         * Initialize quota context.
         */
        void (*dt_init_quota_ctxt)(const struct lu_env *env, struct dt_devic
        /**
         *  get transaction credits for given @op.
         */
        int (*dt_credit_get)(const struct lu_env *env, struct dt_device *dev
};
```

- osd_dt_ops

```
static struct dt_device_operations osd_dt_ops = {
        ...
        .dt_init_quota_ctxt= osd_init_quota_ctxt,
};
```

- osd_init_quota_ctxt

```
static void osd_init_quota_ctxt(const struct lu_env *env, struct dt_device *
                                struct dt_quota_ctxt *ctxt, void *data)
{
        struct obd_device *obd = (void *)ctxt;
        struct vfsmount *mnt = (struct vfsmount *)data;
        obd->u.obt.obt_sb = mnt->mnt_root->d_inode->i_sb;
        OBD_SET_CTXT_MAGIC(&obd->obd_lvfs_ctxt);
        obd->obd_lvfs_ctxt.pwdmnt = mnt;
        obd->obd_lvfs_ctxt.pwd = mnt->mnt_root;
        obd->obd_lvfs_ctxt.fs = get_ds();
}
```

(c) lquota_cleanup

- mdt_fini

```
static void mdt_fini(const struct lu_env *env, struct mdt_device *m)
```

```
{
        ...
        mdt_stop_ptlrpc_service(m);
        next->md_ops->mdo_quota.mqo_cleanup(env, next);
        mdt_fs_cleanup(env, m);
        ...
}
```

- cmm_quota_cleanup

```
static int cmm_quota_cleanup(const struct lu_env *env, struct md_device *m)
{
        /* disable quota for CMD case temporary. */
        if (cmm_dev->cmm_tgt_count)
                return(-EOPNOTSUPP);
        cmm_child_ops(cmm_dev)->mdo_quota.mqo_cleanup(env, cmm_dev->cmm_chil
}
```

- mdd_quota_cleanup

```
static int mdd_quota_cleanup(const struct lu_env *env, struct md_device *m)
{
        lquota_cleanup(mds_quota_interface_ref, obd);
        lquota_fs_cleanup(mds_quota_interface_ref, obd);
}
```

(d) lquota_recovery

- mdt_upcall

```
int mdt_upcall(const struct lu_env *env, struct md_device *md,
               enum md_upcall_event ev, void *data)
{
                ...
                case MD_LOV_SYNC:
                        ...
                        mdt_allow_cli(m, CONFIG_SYNC);
                        if (md->md_lu_dev.ld_obd->obd_recovering == 0)
                                next->md_ops->mdo_quota.mqo_recovery(env, ne
                        break;
                ...
}
```

- mdt_postrecov

```
int mdt_postrecov(const struct lu_env *env, struct mdt_device *mdt)
{
        ...
        rc = ld->ld_ops->ldo_recovery_complete(env, ld);
```

```
        next->md_ops->mdo_quota.mqo_recovery(env, next);
        ...
}
```

- cmm_quota_recovery

```
static int cmm_quota_recovery(const struct lu_env *env, struct md_device *m)
{
        /* disable quota for CMD case temporary. */
        if (cmm_dev->cmm_tgt_count)
                return(-EOPNOTSUPP);
        cmm_child_ops(cmm_dev)->mdo_quota.mqo_recovery(env, cmm_dev->cmm_ch
}
```

- mdd_quota_recovery

```
static int mdd_quota_recovery(const struct lu_env *env, struct md_device *m)
{
        lquota_recovery(mds_quota_interface_ref, obd);
}
```

2. quota control (check / on / off / set / get / invalidate)

   They are for quota processing RPC request from client to check / on / off / set / get / invalidate quota for (maybe specified) user (or group). MDS layer does not process RPC request from client side anymore. It is MDT's duty now. So the RPC handlers related quota process should be triggered in MDT layer.

   - mdt_mds_ops

```
static struct mdt_handler mdt_mds_ops[] = {
        ...
        DEF_MDT_HNDL_F(0,                       QUOTACHECK,   mdt_quotacheck_
        DEF_MDT_HNDL_F(0,                       QUOTACTL,     mdt_quotactl_ha
};
```

   The QUOTACTL RPC from client is ioctl style, it can be interpreted as different quota operations against different ioctl commands. We defines some new md_device_operations for transferring quota control from MDT to MDS, but we do not want to supply an ioctl style md_device_operations interface, on the contrary, we would like to split the ioctl interface into several md_device_operations interfaces, each of them only does definite quota process.

   - md_device_operations

```
struct md_device_operations {
        ...
        struct md_quota_operations {
```

```
...
int (*mqo_check)(const struct lu_env *env,
                 struct md_device *m,
                 struct obd_export *exp,
                 __u32 type);
int (*mqo_on)(const struct lu_env *env,
              struct md_device *m,
              __u32 type,
              __u32 id);
int (*mqo_off)(const struct lu_env *env,
               struct md_device *m,
               __u32 type,
               __u32 id);
int (*mqo_setinfo)(const struct lu_env *env,
                   struct md_device *m,
                   __u32 type,
                   __u32 id,
                   struct obd_dqinfo *dqinfo);
int (*mqo_getinfo)(const struct lu_env *env,
                   const struct md_device *m,
                   __u32 type,
                   __u32 id,
                   struct obd_dqinfo *dqinfo);
int (*mqo_setquota)(const struct lu_env *env,
                    struct md_device *m,
                    __u32 type,
                    __u32 id,
                    struct obd_dqblk *dqblk);
int (*mqo_getquota)(const struct lu_env *env,
                    const struct md_device *m,
                    __u32 type,
                    __u32 id,
                    struct obd_dqblk *dqblk);
int (*mqo_getoinfo)(const struct lu_env *env,
                    const struct md_device *m,
                    __u32 type,
                    __u32 id,
                    struct obd_dqinfo *dqinfo);
int (*mqo_getoquota)(const struct lu_env *env,
                     const struct md_device *m,
                     __u32 type,
                     __u32 id,
                     struct obd_dqblk *dqblk);
int (*mqo_invalidate)(const struct lu_env *env,
                      struct md_device *m,
                      __u32 type);
```

```
                int (*mqo_finvalidate)(const struct lu_env *env,
                                       struct md_device *m,
                                       __u32 type);

        } mdo_quota;
};
```

- cmm_md_ops

```
static struct md_device_operations cmm_md_ops = {
        ...
        .mdo_quota          = {
                ...
                .mqo_check       = cmm_quota_check,
                .mqo_on          = cmm_quota_on,
                .mqo_off         = cmm_quota_off,
                .mqo_setinfo     = cmm_quota_setinfo,
                .mqo_getinfo     = cmm_quota_getinfo,
                .mqo_setquota    = cmm_quota_setquota,
                .mqo_getquota    = cmm_quota_getquota,
                .mqo_getoinfo    = cmm_quota_getoinfo,
                .mqo_getoquota   = cmm_quota_getoquota,
                .mqo_invalidate  = cmm_quota_invalidate,
                .mqo_finvalidate = cmm_quota_finvalidate
        }
};
```

- mdd_ops

```
struct md_device_operations mdd_ops = {
        ...
        .mdo_quota          = {
                ...
                .mqo_check       = mdd_quota_check,
                .mqo_on          = mdd_quota_on,
                .mqo_off         = mdd_quota_off,
                .mqo_setinfo     = mdd_quota_setinfo,
                .mqo_getinfo     = mdd_quota_getinfo,
                .mqo_setquota    = mdd_quota_setquota,
                .mqo_getquota    = mdd_quota_getquota,
                .mqo_getoinfo    = mdd_quota_getoinfo,
                .mqo_getoquota   = mdd_quota_getoquota,
                .mqo_invalidate  = mdd_quota_invalidate,
                .mqo_finvalidate = mdd_quota_finvalidate
        }
};
```

The new stack trace for quota control process is "MDT=>CMM=>MDD=>MDS=>LQUOTA=>LVFS".

(a) quota check

- mdt_quotacheck_handle

```
static int mdt_quotacheck_handle(struct mdt_thread_info *info)
{
        oqctl = req_capsule_client_get(pill, &RMF_OBD_QUOTACTL);
        /* remote client has no permission for quotacheck */
        if (unlikely(exp->exp_connect_flags & OBD_CONNECT_RMT_CLIENT))
                return(-EPERM);
        next->md_ops->mdo_quota.mqo_check(info->mti_env, next, exp, oqctl->q
}
```

- cmm_quota_check

```
static int cmm_quota_check(const struct lu_env *env, struct md_device *m,
                           struct obd_export *exp, __u32 type)
{
        /* disable quota for CMD case temporary. */
        if (cmm_dev->cmm_tgt_count)
                RETURN(-EOPNOTSUPP);
        cmm_child_ops(cmm_dev)->mdo_quota.mqo_check(env, cmm_dev->cmm_child,
}
```

- mdd_quota_check

```
static int mdd_quota_check(const struct lu_env *env, struct md_device *m,
                           struct obd_export *exp, __u32 type)
{
        struct mdd_device *mdd = lu2mdd_dev(&m->md_lu_dev);
        struct obd_device *obd = mdd->mdd_obd_dev;
        struct obd_quotactl *oqctl = &mdd_env_info(env)->mti_oqctl;
        oqctl->qc_type = type;
        lquota_check(mds_quota_interface_ref, obd, exp, oqctl);
}
```

Compared with lustre 1.6 release, we add new parameter of "obd_export" for lquota_check. It is used for quota check callback to client. In lustre 1.6 release, MDS layer processes RPC request from client, so the MDS obd_device's export can be used for that directly. But in HEAD, MDS obd_device's export can be used for communication with client, so we transfer MDT obd_device's export for that.

(b) quota ctl

- mdt_quotactl_handle

```
static int mdt_quotactl_handle(struct mdt_thread_info *info)
{
```

```
        struct md_quota_operations *mqo = &next->md_ops->mdo_quota;
        id = oqctl->qc_id;
        if (info->mti_exp->exp_connect_flags & OBD_CONNECT_RMT_CLIENT) {
                if (unlikely(oqctl->qc_cmd != Q_GETQUOTA && oqctl->qc_cmd !=
                        return(-EPERM);
                if (oqctl->qc_cmd == Q_GETQUOTA) {
                        if (oqctl->qc_type == USRQUOTA)
                                id = lustre_idmap_lookup_uid(NULL, idmap, 0,
                        else if (oqctl->qc_type == GRPQUOTA)
                                id = lustre_idmap_lookup_gid(NULL, idmap, 0,
                }
        }
        switch (oqctl->qc_cmd) {
        case Q_QUOTAON:
                mqo->mqo_on(info->mti_env, next, oqctl->qc_type, id);
                break;
        case Q_QUOTAOFF:
                mqo->mqo_off(info->mti_env, next, oqctl->qc_type, id);
                break;
        case Q_SETINFO:
                mqo->mqo_setinfo(info->mti_env, next, oqctl->qc_type, id, &o
                break;
        case Q_GETINFO:
                mqo->mqo_getinfo(info->mti_env, next, oqctl->qc_type, id, &o
                break;
        case Q_SETQUOTA:
                mqo->mqo_setquota(info->mti_env, next, oqctl->qc_type, id, &
                break;
        case Q_GETQUOTA:
                mqo->mqo_getquota(info->mti_env, next, oqctl->qc_type, id, &
                break;
        case Q_GETOINFO:
                mqo->mqo_getoinfo(info->mti_env, next, oqctl->qc_type, id, &
                break;
        case Q_GETOQUOTA:
                mqo->mqo_getoquota(info->mti_env, next, oqctl->qc_type, id,
                break;
        case LUSTRE_Q_INVALIDATE:
                mqo->mqo_invalidate(info->mti_env, next, oqctl->qc_type);
                break;
        case LUSTRE_Q_FINVALIDATE:
                mqo->mqo_finvalidate(info->mti_env, next, oqctl->qc_type);
                break;
        }
        *repoqc = *oqctl;
}
```

- cmm_quota_on

```
static int cmm_quota_on(const struct lu_env *env, struct md_device *m,
                        __u32 type, __u32 id)
{
        /* disable quota for CMD case temporary. */
        if (cmm_dev->cmm_tgt_count)
                return(-EOPNOTSUPP);
        cmm_child_ops(cmm_dev)->mdo_quota.mqo_on(env, cmm_dev->cmm_child, ty
}
```

- mdd_quota_on

```
static int mdd_quota_on(const struct lu_env *env, struct md_device *m,
                        __u32 type, __u32 id)
{
        struct mdd_device *mdd = lu2mdd_dev(&m->md_lu_dev);
        struct obd_device *obd = mdd->mdd_obd_dev;
        struct obd_quotactl *oqctl = &mdd_env_info(env)->mti_oqctl;
        oqctl->qc_cmd = Q_QUOTAON;
        oqctl->qc_type = type;
        oqctl->qc_id = id;
        lquota_ctl(mds_quota_interface_ref, obd, oqctl);
}
```

Other quota control interfaces have the similar process as mqo_on.

3. quota application (quota_chkquota / quota_pending_commit / quota_adjust)

   It is used for quota slave on MDS server to manage quota limitation and usage which are related with some special file operations: create, unlink, rename and chown (chgrp). We can port these interfaces to MDT layer as other quota interfaces, but to MDD layer is more directly. MDD is the real layer for implementing the file system, it accurately knows the file operation intention, in spite of CMD case or not. But if port to MDT layer, except for more deep stack , we have to distinguish local operation or remote operation for CMD case in future. So we choice MDD layer to port these interfaces.

   - mdd_unlink

```
static int mdd_unlink(const struct lu_env *env, struct md_object *pobj,
                      struct md_object *cobj, const struct lu_name *lname,
                      struct md_attr *ma)
{
        ...
        rc = mdd_finish_unlink(env, mdd_cobj, ma, handle);
```

```
        if (ma->ma_valid & MA_INODE && ma->ma_attr.la_nlink == 0) {
                /* save uid/gid for quota acquire/release */
                qids[USRQUOTA] = ma->ma_attr.la_uid;
                qids[GRPQUOTA] = ma->ma_attr.la_gid;
                quota_opc = FSFILT_OP_UNLINK;
        }
        ...
out_trans:
        mdd_trans_stop(env, mdd, rc, handle);
        if (mds->mds_quota && quota_opc)
                /* trigger dqacq on the owner of child */
                lquota_adjust(mds_quota_interface_ref, obd, qids, qids, rc, quo
        return rc;
}
```

- mdd_rename_tgt

```
static int mdd_rename_tgt(const struct lu_env *env,
                          struct md_object *pobj, struct md_object *tobj,
                          const struct lu_fid *lf, const struct lu_name *lname,
                          struct md_attr *ma)
{
        ...
                rc = mdd_finish_unlink(env, mdd_tobj, ma, handle);
                if (rc)
                        GOTO(cleanup, rc);
                if (ma->ma_valid & MA_INODE && ma->ma_attr.la_nlink == 0) {
                        /* save uid/gid for quota acquire/release */
                        qids[USRQUOTA] = ma->ma_attr.la_uid;
                        qids[GRPQUOTA] = ma->ma_attr.la_gid;
                        quota_opc = FSFILT_OP_RENAME;
                }
        }
        ...
out_trans:
        mdd_trans_stop(env, mdd, rc, handle);
        if (mds->mds_quota && quota_opc)
                /* trigger dqacq on the owner of child */
                lquota_adjust(mds_quota_interface_ref, obd, qids, qids, rc, quo
        return rc;
}
```

- mdd_create

```
static int mdd_create(const struct lu_env *env, struct md_object *pobj, const s
                      struct md_object *child, struct md_op_spec *spec, struct
```

```
{
        ...
        rc = mdd_create_sanity_check(env, pobj, lname, ma, spec);
        if (rc)
                RETURN(rc);
        if (mds->mds_quota) {
                qids[USRQUOTA] = ma->ma_attr.la_uid;
                qids[GRPQUOTA] = ma->ma_attr.la_gid;
                /* we try to get enough quota to write here, and let
                 * ldiskfs decide if it is out of quota or not b=14783*/
                lquota_chkquota(mds_quota_interface_ref, obd,
                                qids[USRQUOTA], qids[GRPQUOTA], 1, &rec_pending
        }
        ...
out_pending:
        if (mds->mds_quota) {
                if (rec_pending)
                        lquota_pending_commit(mds_quota_interface_ref, obd, qid
                /* trigger dqacq on the owner of child */
                lquota_adjust(mds_quota_interface_ref, obd, qids, qids, rc, FSF
        }
        return rc;
}
```

- mdd_rename

```
static int mdd_rename(const struct lu_env *env, struct md_object *src_pobj, str
                      const struct lu_fid *lf, const struct lu_name *lsname, st
                      struct md_attr *ma)
{
        ...
                rc = mdd_finish_unlink(env, mdd_tobj, ma, handle);
                mdd_write_unlock(env, mdd_tobj);
                if (rc)
                        GOTO(cleanup, rc);
                if (ma->ma_valid & MA_INODE && ma->ma_attr.la_nlink == 0) {
                        /* save uid/gid for quota acquire/release */
                        qids[USRQUOTA] = ma->ma_attr.la_uid;
                        qids[GRPQUOTA] = ma->ma_attr.la_gid;
                        quota_opc = FSFILT_OP_RENAME;
                }
        }
        ...
cleanup_unlocked:
        mdd_trans_stop(env, mdd, rc, handle);
        if (mdd_sobj)
```

```
                mdd_object_put(env, mdd_sobj);
        if (mds->mds_quota && quota_opc)
                /* trigger dqacq on the owner of child */
                lquota_adjust(mds_quota_interface_ref, obd, qids, qids, rc, quo
        return rc;
}
```

- mdd_attr_set

```
static int mdd_attr_set(const struct lu_env *env, struct md_object *obj,
                        const struct md_attr *ma)
{
        ...
        rc = mdd_fix_attr(env, mdd_obj, la_copy, ma);
        if (rc)
                GOTO(cleanup, rc);
        if (mds->mds_quota && la_copy->la_valid & (LA_UID | LA_GID)) {
                struct lu_attr *la_tmp = &mdd_env_info(env)->mti_la;
                rc = mdd_la_get(env, mdd_obj, la_tmp, BYPASS_CAPA);
                if (rc)
                        GOTO(cleanup, rc);
                qnids[USRQUOTA] = la_copy->la_uid;
                qnids[GRPQUOTA] = la_copy->la_gid;
                qoids[USRQUOTA] = la_tmp->la_uid;
                qoids[GRPQUOTA] = la_tmp->la_gid;
        }
        ...
        /* trigger dqrel/dqacq for original owner and new owner */
        if (mds->mds_quota && la_copy->la_valid & (LA_UID | LA_GID))
                lquota_adjust(mds_quota_interface_ref, obd, qnids, qoids, rc, F
        RETURN(rc);
}
```

- mdd_ref_del

```
static int mdd_ref_del(const struct lu_env *env, struct md_object *obj,
                       struct md_attr *ma)
{
        ...
        rc = mdd_finish_unlink(env, mdd_obj, ma, handle);
        if (ma->ma_valid & MA_INODE && ma->ma_attr.la_nlink == 0) {
                /* save uid/gid for quota acquire/release */
                qids[USRQUOTA] = ma->ma_attr.la_uid;
                qids[GRPQUOTA] = ma->ma_attr.la_gid;
                quota_opc = FSFILT_OP_UNLINK;
        }
```

```
        ...
cleanup:
        mdd_write_unlock(env, mdd_obj);
        mdd_trans_stop(env, mdd, rc, handle);
        if (mds->mds_quota && quota_opc)
                /* trigger dqacq on the owner of child */
                lquota_adjust(mds_quota_interface_ref, obd, qids, qids, rc, quo
        return rc;
}
```

- mdd_object_create

```
static int mdd_object_create(const struct lu_env *env, struct md_object *obj,
                             const struct md_op_spec *spec, struct md_attr *ma)
{
        if (mds->mds_quota)
                /* we try to get enough quota to write here, and let
                 * ldiskfs decide if it is out of quota or not b=14783*/
                lquota_chkquota(mds_quota_interface_ref, obd, qids[USRQUOTA], q
        mdd_txn_param_build(env, mdd, MDD_TXN_OBJECT_CREATE_OP);
        ...
out_pending:
        if (mds->mds_quota) {
                if (rec_pending)
                        lquota_pending_commit(mds_quota_interface_ref, obd, qid
                /* trigger dqacq on the owner of child */
                lquota_adjust(mds_quota_interface_ref, obd, qids, qids, rc, FSF
        }
        return rc;
}
```

All the above discussion do not consider CMD support. Lustre 2.0 release will not support CMD. If consider quota support for CMD release in future, the first architecture choice to be made is that: single quota master for the whole system or each MDS server has one quota master.

3.1.2.2 new current user identity Quota is uid/gid based, when perform "create/unlink/rename/chown" file operations on MDS server, the lower LDISKFS checks/updates the quota limitation/usage with current user identity which is in current thread context. In lustre 1.6.x release, "push_ctxt()" and "pop_ctxt()" are used for initializing the current user identity in current thread context before the real file operations on MDS server. So all the MDS server stack layers (including the LDISKFS) can share the current user identity through current thread context. But for new MD stack in HEAD, we do not use current thread context to share current user identity anymore. It has been replaced with "md_ucred" structure transferred among different MD

stack layers as functions parameter. Unfortunately, "md_ucred" can not be used by LDISKFS now. But it is LDISKFS that really process quota when create/unlink/rename/chown. There are two directions to make LDISKFS work properly:

- Fix LDISKFS related interfaces (create/chown) to accept "md_ucred" as new parameter. But such interfaces are standard VFS interfaces, fixing them will cause more kernel patches. So it is bad choice.
- Perform some lightweight operations to initialize current thread context before calls such interfaces on MDS server. It just fixes the "fsuid/fsgid/cap_effective" which are related with quota operation, but "push_ctxt()" will fix "fs/pwdmnt/pwd/group_info" and some others also. So it is somehow lightweight "push_ctxt()". We prefer to do that.

Another issue to be considered is that: in lustre 1.6.x release, LDISKFS updates related quota usage when change file owner; but in HEAD, it is OSD's duty to update file attribute for chown, LDISKFS knows nothing about that, and can not update related quota usage for such case. So the OSD layer should update related quota usage by itself. "DQUOT_TRANSFER" is introduced for that.

1. osd_push_ctxt

```
static inline void osd_push_ctxt(const struct lu_env *env, struct osd_ctxt *save)

{
    struct md_ucred *uc = md_ucred(env);
    save->oc_uid = current->fsuid;
    save->oc_gid = current->fsgid;
    save->oc_cap = current->cap_effective;
    current->fsuid = uc->mu_fsuid;
    current->fsgid = uc->mu_fsgid;
    current->cap_effective = uc->mu_cap;
}
```

2. osd_pop_ctxt

```
static inline void osd_pop_ctxt(struct osd_ctxt *save)

{
    current->fsuid = save->oc_uid;
    current->fsgid = save->oc_gid;
    current->cap_effective = save->oc_cap;
}
```

3. osd_mkfile

```
static int osd_mkfile(struct osd_thread_info *info, struct osd_object *obj,

                      umode_t mode,
                      struct dt_allocation_hint *hint,
                      struct thandle *th)
{
        ...
        osd_push_ctxt(info->oti_env, &save);
        inode = ldiskfs_create_inode(oth->ot_handle, parent, mode);
        osd_pop_ctxt(&save);
        ...
}
```

4. osd_inode_setattr

```
static int osd_inode_setattr(const struct lu_env *env,

    struct inode *inode,
    const struct lu_attr *attr)
    {
        ...
        LASSERT(!(bits & LA_TYPE));/* Huh? You want too much. */
        if ((bits & LA_UID && attr->la_uid != inode->i_uid) ||
            (bits & LA_GID && attr->la_gid != inode->i_gid)) {
                struct osd_ctxt *save = &osd_oti_get(env)->oti_ctxt;
                struct iattr iattr;
                iattr.ia_valid = bits & (LA_UID | LA_GID);
                iattr.ia_uid = attr->la_uid;
                iattr.ia_gid = attr->la_gid;
                osd_push_ctxt(env, &save);
                DQUOT_TRANSFER(inode, &iattr) ? -EDQUOT : 0;
                osd_pop_ctxt(&save);
        }
        if (bits & LA_ATIME)
                inode->i_atime  = *osd_inode_time(env, inode, attr->la_atime);
        ...
    }
```

3.2 security support for quota on HEAD

Security is new feature of HEAD compared with lustre 1.6.x release, including GSS framework, MDS/OSS capability, remote client(user), and so on. Some of them are related with quota. Especially because lustre pre-create process.

When create on MDS, MDS (with OSC) will check whether the OSS objects have been pre-created or not. If yes, only creates MDS object and returns directly, otherwise, MDS sends RPC to OSS to create the OSS objects and pre-create some other objects for later creation. At that time, the OSS objects have "0" uid/gid with SUID/SGID mode. When client writes such OSS objects for the first time, it sends file owner information to OSS, then OSS set the real file owner which should be the same as MDS object.

3.2.1 remote client and quota

Remote client is regarded as untrusted. Generally, remote client and server use different user database, that means Tom@client-side and Tom@server-side are different user (maybe with different uids). So UID mapping is introduced (on MDS server) to make all the system to share the same user database. Currently, lustre UID mapping is dynamic, which means only logined user is mapped. On the other hand, the mapping is transparent to client, MDS only sends client-side uid/gid to remote client in RPC reply, and remote user only knows its client-side uid/gid, but does not know which server-side uid/gid are mapped to.

- Consider the pre-create, remote client should send the real file owner information to OSS to initialize the OSS objects. The real file owner are server-side uid/gid, not client-side ones, but remote client does not know that. So some new mechanism should be introduced for that. Four possible solutions:
  - MDS sends the real file owner information to OSS directly to initialize OSS objects when create. But it causes MDS to communicate with OSS for each create operation, and counteract the pre-create performance benefit. So it is bad choice.
  - MDS sends server-side file owner information back to client directly, and client transfers such information to OSS, OSS initializes OSS objects with that. It sound simple, but it breaks lustre UID mapping rule: UID mapping should be transparent to client, and it makes the former effort for such transparent mapping totally invalid. So not a good choice.
  - Perform UID mapping on OSS just like what has been done on MDS. When client sends client-side file owner information to service side, OSS maps them to server-side ones which are the real file owner. Sound reasonable, but here we can not guarantee that the real file owner are mapped, for lustre UID mapping is dynamic. If not mapped yet, "nobody" is used for client-side file owner information, but OSS can not perform such mapping. So some mechanism defects.
  - MDS sends the encrypted file owner information back to client, and client transfers such encrypted information to OSS, OSS decrypts

such file owner firstly, and then initializes OSS objects. It is relative better. So it is the current solution. There are many encrypt algorithms can be used for that, e.g. DES, 3DES, AES, and so on. We choose AES. It shares the same key (or part of the key) and key updating with MDS/OSS capability.

1. capa_encrypt_id

   ```
   int capa_encrypt_id(__u32 *d, __u32 *s, __u8 *key, int keylen)
   ```

   Wrapper of aes_encrypt to encrypt file owner information.
2. capa_decrypt_id

   ```
   int capa_decrypt_id(__u32 *d, __u32 *s, __u8 *key, int keylen)
   ```

   Wrapper of aes_decrypt to decrypt file owner information.

- Since remote user is untrusted, its behavior should be more controlled than trusted ones. Consider quota related operation, there are some limit:
  - Remote user can not "on / off / check / set" quota.
    Remote user has no permission to perform "lfs quotaon / lfs quotaoff / lfs quotacheck / lfs setquota", even if it is (remote) root user.
  - Remote user can query quota information only for its own uid / gid.
    Remote user can perform "lfs quota <-u username | -g groupname> filesystem", but the "username" / "groupname" must match its current euid / egid, otherwise the operation will be denied in spite of (remote) root user or not. The reason for "username" / "groupname" matching requirement is that such quota query operation should guarantee the specified user has been mapped, otherwise "nobody" information will be returned, and it is meaningless.
  - Remote user can query quota information only on the system level, but not on the node level.
    "lfs quota -v" is equal to "lfs quota" for remote user. That means remote user can not query quota information on specified nodes. The reason is that OSS does not support UID mapping as does on MDS now.
  - Remote user can not break through quota limitation even if it is (remote) root user.
    In lustre, there is no quota limitation for local root user. Local root user can write other user's file to any size in spite of the quota limitation for such user. But it is not the same situation for remote user, even if it is (remote) root user. Break through quota limitation permission flags ("OBD_BRW_NOQUOTA") is set by client, and transferred to OSS. A baleful remote client can set such flags arbitrarily, that makes quota limitation meaningless. On the other hand, client-side root user maybe mapped to non-root user on server-side, but

remote client does not know it, it sets "OBD_BRW_NOQUOTA" for client-side root as normal. So such flags for remote client is incredible, should be ignored by OSS anyway.

1. filter_commitrw_write

```
int filter_commitrw_write(struct obd_export *exp, struct obdo *oa,
                          int objcount, struct obd_ioobj *obj, int niocount,
                          struct niobuf_local *res, struct obd_trans_info *o
                          int rc)
{
        ...
                 /* if one page is a write-back page from client cache and
                  * not from direct_io, or it's written by root, then mark
                  * the whole io request as ignore quota request, remote
                  * client can not break through quota. */
                 if (exp_connect_rmtclient(exp))
                         flags &= ~OBD_BRW_NOQUOTA;
                 if ((flags & OBD_BRW_NOQUOTA) ||
                     (flags & (OBD_BRW_FROM_GRANT | OBD_BRW_SYNC)) ==
                      OBD_BRW_FROM_GRANT)
                         iobuf->dr_ignore_quota = 1;
        }
        push_ctxt(&saved, &obd->obd_lvfs_ctxt, NULL);
        ...
}
```

– Chown operation is controlled by admin for remote client

When changes file owner, the related block / file quota information will be updated. This can be used for breaking through quota by baleful remote client. So the remote root user's permission for chown should be controlled more carefully. Currently, system administrator can enable / disable such permission by configure the file "/etc/lustre/perm.conf" with flags "rmtown" / "normtown", the default mode is "normtown".

1. new_init_ucred

```
static int new_init_ucred(struct mdt_thread_info *info, ucred_init_type_t ty
                          void *buf)
{
        ...
        /* remove fs privilege for non-root user. */
        if (ucred->mu_fsuid)
                ucred->mu_cap = pud->pud_cap & ~CFS_CAP_FS_MASK;
        else
                ucred->mu_cap = pud->pud_cap;
```

```
                if (remote) {
                        ucred->mu_cap &= ~CFS_CAP_SYS_RESOURCE_MASK;
                        if (!(perm & CFS_RMTOWN_PERM))
                                ucred->mu_cap &= ~CFS_CAP_CHOWN_MASK;
                }
                ucred->mu_valid = UCRED_NEW;
                EXIT;
                ...
        }
```

3.2.2 MDS/OSS capability and quota

As mentioned above, OSS initializes objects with the file owner information from client when the first write. Here we point out one important issue: how can OSS believe the file owner information sent from client? Especially for remote client, a baleful one maybe send fake file owner information to cheat OSS to bypass quota limitation. So we need some mechanism to guarantee the file owner information can not be fabricated by client. The MDS/OSS capability can be used for that, which means MDS sends file owner information as part of OSS capability to client, then client transfers OSS capability to OSS, and then OSS verifies the validity of such OSS capability.

In fact, the risk of fabricating file owner information is not only for remote client, but also for local one. We should prevent any cases. But as known, assigning capability is time-consuming, maybe we can make a compromise between performance and reliability. So lustre provides four MDS/OSS capability security levels for different client(user) reliability environments.

- Level 0: disable MDS/OSS capability for all clients.
- Level 1: enable MDS/OSS capability on remote client.
- Level 2: enable MDS/OSS capability on selected client(s).
- Level 3: enable MDS/OSS capability on all clients.

For detail, please refer to <Lustre capability Security Level> https://bugzilla.lustre.org/show_bug.cgi?id=1556 attachment (id=17448).

For a client, if server disable MDS/OSS capability for it, then file owner information is packed in "obdo" as 1.6.x release does; otherwise, the real file owner information is in related OSS capability, it is encrypted mode for remote client, and unencrypted mode for local client.

1. lustre_capa

```
struct lustre_capa {
        struct lu_fid   lc_fid;         /* fid */
        __u64           lc_opc;         /* operations allowed */
        __u32           lc_uid;         /* file owner */
```

```
        __u32           lc_uid_pending; /* pending to 64 bits for encrypt */
        __u32           lc_gid;         /* file group */
        __u32           lc_gid_pending; /* pending to 64 bits for encrypt */
        __u32           lc_flags;       /* HMAC algorithm & flags */
        __u32           lc_keyid;       /* key# used for the capability */
        __u32           lc_timeout;     /* capa timeout value (sec) */
        __u32           lc_expiry;      /* expiry time (sec) */
        __u8            lc_hmac[CAPA_HMAC_MAX_LEN];   /* HMAC */
}__attribute__((packed));
```

2. osd_capa_get

```
static struct obd_capa *osd_capa_get(const struct lu_env *env,
                                     struct dt_object *dt,
                                     struct lustre_capa *old,
                                     __u64 opc)
{
        ...
        switch (ci->mc_auth) {
        case LC_ID_PLAIN:
                capa->lc_uid = obj->oo_inode->i_uid;
                capa->lc_gid = obj->oo_inode->i_gid;
                capa->lc_flags = LC_ID_PLAIN;
                break;
        case LC_ID_CONVERT: {
                __u32 d[4], s[4];
                s[0] = obj->oo_inode->i_uid;
                get_random_bytes(&(s[1]), sizeof(__u32));
                s[2] = obj->oo_inode->i_gid;
                get_random_bytes(&(s[3]), sizeof(__u32));
                capa_encrypt_id(d, s, key->lk_key, CAPA_HMAC_KEY_MAX_LEN);
                capa->lc_uid         = d[0];
                capa->lc_uid_pending = d[1];
                capa->lc_gid         = d[2];
                capa->lc_gid_pending = d[3];
                capa->lc_flags = LC_ID_CONVERT;
                break;
        }
        }
        capa->lc_fid = *fid;
        ...
}
```

3. filter_capa_fixoa

```
int filter_capa_fixoa(struct obd_export *exp, struct obdo *oa, __u64 mdsid,
                      struct lustre_capa *capa)
{
        if (capa_flags(capa) == LC_ID_CONVERT) {
                /* find OSS capability key, which is the same as uid/gid encrypt/d
                list_for_each_entry(k, &filter->fo_capa_keys, k_list) {
                        if (k->k_key.lk_mdsid == mdsid &&
                            k->k_key.lk_keyid == capa_keyid(capa))
                                found = 1;
                }
                if (found) {
                        __u32 d[4], s[4] = {capa_uid(capa), capa_uid_pending(capa)
                                            capa_gid(capa), capa_gid_pending(capa)
                        rc = capa_decrypt_id(d, s, k->k_key.lk_key, CAPA_HMAC_KEY_
                        oa->o_uid = d[0];
                        oa->o_gid = d[2];
                }
        }
}
```

4. filter_setattr

```
int filter_setattr(struct obd_export *exp, struct obd_info *oinfo,
                   struct obd_trans_info *oti)
{
        ...
        rc = filter_auth_capa(exp, NULL, obdo_mdsno(oa), capa, opc);
        if (rc)
                RETURN(rc);
        if (oa->o_valid & (OBD_MD_FLUID | OBD_MD_FLGID)) {
                filter_capa_fixoa(exp, oa, obdo_mdsno(oa), capa);
        }
        ...
}
```

5. filter_preprw_write

```
static int filter_preprw_write(int cmd, struct obd_export *exp, struct obdo *oa,
                               int objcount, struct obd_ioobj *obj,
                               int niocount, struct niobuf_remote *nb,
                               struct niobuf_local *res,
                               struct obd_trans_info *oti,
                               struct lustre_capa *capa)
{
```

```
        ...
        if (dentry->d_inode == NULL) {
                CERROR("%s: trying to BRW to non-existent file "LPU64"\n",
                       exp->exp_obd->obd_name, obj->ioo_id);
                GOTO(cleanup, rc = -ENOENT);
        }
        if (oa->o_valid & (OBD_MD_FLUID | OBD_MD_FLGID) &&
            dentry->d_inode->i_mode & (S_ISUID | S_ISGID)) {
                filter_capa_fixoa(exp, oa, obdo_mdsno(oa), capa);
        }
        ...
}
```

3.3 quota changes

3.3.1 condition compiling for different quota versions

In lustre 1.6.x release, there are many condition compiling for different quota versions which are for quota compatibility of former lustre release, e.g.

```
#if LUSTRE_VERSION_CODE < OBD_OCD_VERSION(1, 7, 0, 0)
extern void lustre_swab_qdata_old(struct qunit_data_old *d);
#else #warning "remove quota code above for format obsolete in new release"
#endif
#if LUSTRE_VERSION_CODE < OBD_OCD_VERSION(1, 9, 0, 0)
extern void lustre_swab_qdata_old2(struct qunit_data_old2 *d);
#else #warning "remove quota code above for format obsolete in new release"
#endif
```

In HEAD, we only support compiling lustre 2.0 release quota, which version is greater than 1.9.0.0, so all of these condition compiling have been dropped. For lustre 1.6.6 release and later versions, they use the same on wire and on disk quota structures or formats, have no compatibility issues.

3.3.2 64 bit quota support

lustre 1.6.x release supports both 32 bit quota and 64 bit quota. 64 bit quota is the new version, 32 bit quota is for the compatibility for the former lustre release. lustre 2.0 release only needs to be compatible with lustre 1.8 release which supports 64 bit quota. So we drop all the 32 bit quota support in lustre 2.0 release.

4 Focus for inspections

There is an known issue for quota when async write. The client can write data into its local cache and return success to user directly. After a while, the async

write are flushed to OSS with "OBD_BRW_FROM_GRANT" flag set. OSS can not refuse such write for the writer maybe already exit with success. Such case is normal for async write, in spite of local client or remote one. It can be used by baleful client to break through block quota limitation on OSS. Possible solutions are as following:

- Disable async write from remote client, all the write from remote client are converted to be sync write by force. But it will cause serious performance drop.
- Disable async write from remote client when it exceeds the quota limitation much (by some percentage). The issue is what percentage is suitable.
- Before async write, client should acquire related lock from OSS, when client applies such lock, the related quota should be considered. If there is not enough quota, then the client can not acquire such lock.

Seems the last solution is better, but it maybe cause much of change for lock and quota process, should be designed carefully. Since it is common issue for both b1_6 and HEAD, we would like to resolved it separately from this porting task.

5 Reference

1. <Quota For Lustre> cvs: doc/HLD/quota_hld.lyx
2. <Lustre capability Security Level> https://bugzilla.lustre.org/show_bug.cgi?id=15564 attachment (id=17448)