DLD Porting of Early Lock Cancellation to CMD.

Vitaly Fertman

2007-03-30

1 Introduction.

The current DLD describes changes that need to be made in the Early Lock Cancellation mechanism produced in the bug 10589 to have it working in CMD environment.

The Early Lock Cancellation is an ldlm optimization for some lustre filesystem operations that decreases the amount of synchronous RPC between clients and servers. Thus the unlink system call may involve CB_BLOCKING and CB_CANCELING RPCs for every lock clients have on this file. Canceling these locks on the client side before sending unlink to servers and batching these cancels with the unlink RPC will save 2 RPC per lock for the client that sends unlink. There are other filesystem operations that benefit from such an improvement like rename and create.

The description of Early Lock Cancellation internals is out of scope of this document, it focuses instead on the porting itself.

2 Requirements.

To understand the problem scope better the Early Lock Cancellation requirements are also provided.

2.1 Early Lock Cancellation requirements.

- The client cancels all the conflict locks for the current fs operation on the client side beforehand.
- The client cancels only those conflict locks that are unused. Other locks are canceled later in the usual way at the server blocking request.

- The canceling of these locks must be done locally on the client without sending an RPC to a server.
- The client must have an unified mechanism of batching canceled locks into a special RPC buffer for an arbitrary RPC.
- Receiving an RPC with such a batched cancel buffer, a server (MDS, OST) must cancel these locks before proceeding with the main fs operation.
- The client must bundle lock cancels with RPCs whenever possible, the following fs operation are involved currently: unlink, rename, create, setattr, link, destroy.
- The client also cancels an unused lock and all the aged locks when a lock is enqueued and lru list is full.

Note: The last requirement was defined and implemented for the Early Lock Cancellation task, but can be re-worked later in the LRU-resize task.

2.2 Early Lock Cancellation porting requirements.

- The client bundle lock cancels to the RPC to the proper server in the metadata cluster. If there are locks conflicting to the metadata operation but from other servers, lock cancels are not bundled.
- The client sends additional asynchronous batched cancel RPC to servers before the main metadata operation RPC, if it has locks, conflicting to the current metadata operation, from other servers, but there are no RPC to bundle these locks with yet.

3 Functional specification.

The additional requirements for the porting talk about matching conflicting locks on the client with the servers we send RPC to and bundling them with the proper RPC or sending additional cancel RPC. To meet these requirements the modules that need their logic to be changed are LMV and MDC. Other changes belongs to different method names or data/format descriptions.

3.1 Data definitions.

1. Flags to let MDC know if conflicting locks should be searched on MDC's export for the appropriate fid in md_op_data. The mdt_ioepoch_flags enum is enhanced and renamed:

```
enum md_op_flags {
    ...
    /* Cancel locks on the said fid. */
    MF_MDC_CANCEL_FID1 = (1 << 3),
    MF_MDC_CANCEL_FID2 = (1 << 4),
    MF_MDC_CANCEL_FID3 = (1 << 5),
    MF_MDC_CANCEL_FID4 = (1 << 6),
};
```

2. Mask for the SOM flags to distinguish them from the above cancel flags.

   This mask is used in mdc_*_pack methods when md_op_flags are sent to server.

```
#define MF_SOM_FLAGS (MF_SOM_CHANGE | MF_EPOCH_OPEN | MF_EPOCH_CLOSE)
```

3. Lock flag to pass into ldlm code along with other flags, it makes cancel request asynchronous.

```
#define LDLM_FL_ASYNC 0x20000000
```

3.2 LDLM.

1. int ldlm_cli_cancel_unused_resource(struct ldlm_namespace *ns, const struct ldlm_res_id *res_id, ldlm_policy_data_t *policy, int mode, int flags, void *opaque)

   New 2 parameters @policy and @mode are added to cancel only matched locks on the resource instead of all unused locks.

```
@policy keeps the policy of locks we are interested in. Locks of other policies are
@mode defines the mode of the new lock that will be taken later, all the locks conf
```

3.3 MD.

1. static inline int md_cancel_unused(struct obd_export *exp, const struct lu_fid *fid, ldlm_policy_data_t *policy, ldlm_mode_t mode, int flags, void *opaque)

   New 2 parameters @policy, and @mode are added to cancel only matched locks on the ldlm resource. Their semantic is the same as described above in ldlm_cli_cancel_unused_resource().

3.4 LMV.

1. static int lmv_early_cancel(struct lmv_obd *lmv, struct obd_export *tgt_exp, struct obd_export *fid_exp, struct md_op_data *op_data, ldlm_mode_t mode, int bits, int flag)

   Check if we need to cancel locks in the same namespace as we send RPC to, if so mark this fid in md_op_data flags as to be handled later in MDC. Otherwise, find all the conflict locks for that export and given fid, cancel them locally and send asynchronous batched cancel RPC.

   ```
   @tgt_exp is the export the metadata request is sent to.
   @fid_exp is the export the cancel should be sent to for the current fid. if @fid_ex
   @op_data keeps the current fid, which is pointed through @flag.
   @flag points the current fid in op_data and is set to @op_data->op_flags if exports
   @mode, @bits -- inodebits lock match parameters.
   ```

   Return value: an error occurred.

4 Use case scenarios.

4.1 ll_unlink_generic().

1. LLITE calls lmv_unlink() to forward a file unlink request to the proper MDS.
2. LMV checks if the client has conflicting locks from other servers than the unlink RPC will be sent to. If there are some, it calls mdc_cancel_unused() to cancel these locks before the unlink RPC.
3. MDC calls ldlm_cli_cancel_unused_resource() to find and cancel locks on the proper resource and then send batched cancel RPC asynchronously.
4. LMV proceeds with the metadata operation, calls mdc_unlink() which in its turn checks for client conflicting locks from this server, cancel them locally and bundle lock handles with the unlink RPC.

4.2 ll_rmdir_generic().

1. LLITE calls lmv_unlink(), the following is the same as in ll_unlink_generic().

4.3 ll_rename_generic().

1. LLITE calls lmv_rename() to rename a file on MDS.
2. LMV checks if the client has conflicting locks from other servers than the rename RPC will be sent to. If there are some, it calls mdc_cancel_unused() to cancel these locks before the rename RPC.
3. MDC calls ldlm_cli_cancel_unused_resource() to find and cancel locks on the proper resource and and send batched cancel RPC asynchronously.
4. LMV proceeds with the metadata operation, calls mdc_rename() which in its turn checks for client conflicting locks from this server, cancel them locally and bundle lock handles with the rename RPC.

4.4 ll_link_generic().

1. LLITE calls lmv_link() to link a file on MDS.
2. LMV checks if the client has conflicting locks from other servers than the link RPC will be sent to. If there are some, it calls mdc_cancel_unused() to cancel these locks before the link RPC.
3. MDC calls ldlm_cli_cancel_unused_resource() to find and cancel locks on the proper resource and and send batched cancel RPC asynchronously.
4. LMV proceeds with the metadata operation, calls mdc_link() which in its turn checks for client conflicting locks from this server, cancel them locally and bundle lock handles with the link RPC.

5 Logic specification.

5.1 LDLM.

1. int ldlm_cli_cancel_unused_resource(struct ldlm_namespace *ns, const struct ldlm_res_id *res_id, ldlm_policy_data_t *policy, int mode, int flags, void *opaque)

   New 2 parameters @policy, and @mode are added to cancel only matched locks on the resource instead of all unused locks. Passing them down to ldlm_cancel_resource_local().

```
{
    ...
    count = ldlm_cancel_resource_local(res, &cancels, policy,
                                       mode, 0, flags, opaque);
    ...
}
```

2. int ldlm_cli_cancel_unused(struct ldlm_namespace *ns, const struct ldlm_res_id *res_id, int flags, void *opaque)

   Ask resource to cancel all unused locks, pass there NULL and LCK_MINMODE as policy and lock mode correspondingly for them.

```
{
    ...
    rc = ldlm_cli_cancel_unused_resource(ns, res_id, NULL,
                                         LCK_MINMODE, flags,
                                         opaque);
    ...
}
```

3. int ldlm_cli_cancel_req(struct obd_export *exp, struct list_head *head, int count, int flags)

   Handle the given @flags – if LDLM_FL_ASYNC flag is set, send asynchronous RPC.

```
if (flags & LDLM_FL_ASYNC)
    ptlrpcd_add_req(req);
else
    rc = ptlrpc_queue_wait(req);
```

5.2 MDC.

1. static inline int mdc_cancel_unused(struct obd_export *exp, const struct lu_fid *fid, ldlm_policy_data_t *policy, ldlm_mode_t mode, int flags, void *opaque)

   New 2 parameters @policy, and @mode are added to cancel only matched locks on the resource. Call ldlm_cli_cancel_unused_resource() directly giving it new parameters.

```
{
    ...
    rc = ldlm_cli_cancel_unused_resource(obd->obd_namespace,
                                         &res_id, policy, mode,
                                         flags, opaque);
    ...
}
```

2. int mdc_unlink(struct obd_export *exp, struct md_op_data *op_data, struct ptlrpc_request **request)

   Instead of searching conflict locks for fid3 (child's fid), check for the proper flag first. If there is no flag set, locks are on another export from another namespace and it was handled above in lmv.

```
{
    ...
    if ((op_data->op_flags & MF_MDC_CANCEL_FID3) &&
        (fid_is_sane(&op_data->op_fid3)))
        count += mdc_resource_get_unused(exp, &op_data->op_fid3,
                                         &cancels, LCK_EX,
                                         MDS_INODELOCK_FULL);
    ...
}
```

There is no need for an extra check for fid1, as it is of the same export as the main metadata operation.

3. int mdc_link(struct obd_export *exp, struct md_op_data *op_data, struct ptlrpc_request **request)

   The same as in mdc_unlink for fid1 (child's fid).

4. int mdc_rename(struct obd_export *exp, struct md_op_data *op_data, const char *old, int oldlen, const char *new, int newlen, struct ptlrpc_request **request)

   The same as in mdc_unlink for fid2 (target parent), fid3 (source child), fid4 (target child).

5.3 LMV.

1. Get the proper fid in md_op_data by a md_op_flags flag. LMV local purpose macro.

```
#define md_op_data_fid(op_data, fl)                     \
        (fl == MF_MDC_CANCEL_FID1 ? &op_data->op_fid1 : \
         fl == MF_MDC_CANCEL_FID2 ? &op_data->op_fid2 : \
         fl == MF_MDC_CANCEL_FID3 ? &op_data->op_fid3 : \
         fl == MF_MDC_CANCEL_FID4 ? &op_data->op_fid4 : \
         NULL)
```

2. static int lmv_early_cancel(struct lmv_obd *lmv, struct obd_export *tgt_exp, struct obd_export *fid_exp, struct md_op_data *op_data, ldlm_mode_t mode, int bits, int flag)

   Check if we need to cancel locks in the same namespace as we send RPC to, if so mark this fid in md_op_data flags as to be handled later in MDC. Otherwise, cancel all the conflict locks for that export and given fid.

```
static int lmv_early_cancel(struct lmv_obd *lmv,
                            struct obd_export *tgt_exp,
                            struct obd_export *fid_exp,
```

```
                                struct md_op_data *op_data,
                                ldlm_mode_t mode, int bits, int flag)
{
    struct lu_fid *fid = md_op_data_fid(op_data, flag);
    ...
    if (!fid_is_sane(fid))
            RETURN(0);
    if (fid_exp == NULL)
            fid_exp = lmv_find_export(lmv, fid);
    if (tgt_exp == fid_exp) {
            op_data->op_flags |= flag;
            RETURN(0);
    }
    policy.l_inodebits.bits = bits;
    rc = md_cancel_unused(fid_exp, fid, &policy,
                          mode, LDLM_FL_ASYNC, NULL);
    RETURN(rc);
}
```

3. static int lmv_link(struct obd_export *exp, struct md_op_data *op_data, struct ptlrpc_request **request)

   Before mdc_link(), check if child (fid1) is of the same export as parent (fid2). Otherwise, send asynchronous cancel RPC with found conflicting locks on that export: UPDATE lock. Set the proper flags to search the conflicting locks on MDC.

```
if (op_data->op_namelen) {
    op_data->op_flags |= MF_MDC_CANCEL_FID2;
    /* Cancel unused update locks on child (fid1). */
    rc = lmv_early_cancel(lmv, tgt_exp, NULL, op_data, LCK_EX,
                          MDS_INODELOCK_UPDATE,
                          MF_MDC_CANCEL_FID1);
}
```

4. static int lmv_rename(struct obd_export *exp, struct md_op_data *op_data, const char *old, int oldlen, const char *new, int newlen, struct ptlrpc_request **request)

   Before mdc_rename(), check if target parent (fid2), target child (fid4) are of the same exports as the source parent (fid1). Otherwise, send asynchronous cancel RPC with found conflicting locks on those exports: UPDATE lock for target parent, FULL lock for target child. Set the proper flags to search the conflicting locks on MDC.

```
if (oldlen) {
    /* LOOKUP lock on src child (fid3) should also be
     * canceled for parent src_exp in mdc_rename. */
```

```
        op_data->op_flags |= MF_MDC_CANCEL_FID1 | MF_MDC_CANCEL_FID3;
        /* Cancel UPDATE locks on tgt parent (fid2). */
        rc = lmv_early_cancel(lmv, src_exp, NULL, op_data,
                              LCK_EX, MDS_INODELOCK_UPDATE,
                              MF_MDC_CANCEL_FID2);
        /* Cancel LOOKUP locks on tgt child (fid4) for parent tgt_exp */
        if (rc == 0)
            rc = lmv_early_cancel(lmv, src_exp, tgt_exp, op_data,
                                  LCK_EX, MDS_INODELOCK_LOOKUP,
                                  MF_MDC_CANCEL_FID4);
        /* Cancel all the locks on tgt child (fid4). */
        if (rc == 0)
            rc = lmv_early_cancel(lmv, src_exp, NULL, op_data, LCK_EX,
                                  MDS_INODELOCK_FULL,
                                  MF_MDC_CANCEL_FID4);
    }
```

Note: The case when target child is a striped dir is not supported. Only the master stripe has all locks canceled early.

5. static int lmv_unlink(struct obd_export *exp, struct md_op_data *op_data, struct ptlrpc_request **request)

   Before mdc_unlink(), check if child (fid3)is of the same export as the parent (fid1). Otherwise, send asynchronous cancel RPC with found conflicting locks on that export: FULL locks. Set the proper flags to search the conflicting locks on MDC.

```
    if (op_data->op_namelen) {
        /* LOOKUP lock for child (fid3) should also be
         * canceled on parent tgt_exp in mdc_unlink(). */
        op_data->op_flags |= MF_MDC_CANCEL_FID1 | MF_MDC_CANCEL_FID3;
        /* Cancel FULL locks on child (fid3). */
        rc = lmv_early_cancel(lmv, tgt_exp, NULL, op_data, LCK_EX,
                              MDS_INODELOCK_FULL, MF_MDC_CANCEL_FID3);
    }
```

   Note: the case when child is a striped dir is not supported. Only the master stripe has all locks canceled early.

5.4 MDT.

1. static inline int mdt_dlmreq_unpack(struct mdt_thread_info *info)

   It checks that DLM_REQ buffer presents in RPC and unpack it.

```
    if (req_capsule_get_size(pill, &RMF_DLM_REQ, RCL_CLIENT)) {
```

```
        info->mti_dlm_req = req_capsule_client_get(pill, &RMF_DLM_REQ);
        if (info->mti_dlm_req == NULL)
            RETURN(-EFAULT);
    }
    RETURN(0);
```

2. static int mdt_setattr_unpack(struct mdt_thread_info *info)

   As it is possible that EADATA and LOGCOOKIES are not sent but there are some locks to be sent in DLM_REQ buffer, EADATA and LOGCOOKIES will present but will be of zero length. Therefore, the buffer presents if it exists in RPC and its length is not zero. It calls mdt_dlmreq_unpack to unpack batched cancel data.

```
    if ((ma->ma_lmm_size =
         req_capsule_get_size(pill, &RMF_EADATA, RCL_CLIENT))) {
        ma->ma_lmm = req_capsule_client_get(pill, &RMF_EADATA);
        ma->ma_valid |= MA_LOV;
    }
    if ((ma->ma_cookie_size =
        req_capsule_get_size(pill, &RMF_LOGCOOKIES, RCL_CLIENT))) {
        ma->ma_cookie = req_capsule_client_get(pill, &RMF_LOGCOOKIES);
        ma->ma_valid |= MA_COOKIE;
    }
    rc = mdt_dlmreq_unpack(info);
```

3. static int mdt_create_unpack(struct mdt_thread_info *info)

   The same as in mdt_setattr_unpack() case, SYMTGT buffer is unpacked relying on the bugger size. It calls mdt_dlmreq_unpack to unpack batched cancel data.

```
    if (req_capsule_get_size(pill, &RMF_SYMTGT, RCL_CLIENT)) {
        tgt = req_capsule_client_get(pill, &RMF_SYMTGT);
        sp->u.sp_symname = tgt;
    }
    ...
    rc = mdt_dlmreq_unpack(info);
```

4. static int mdt_link_unpack(struct mdt_thread_info *info)
5. static int mdt_unlink_unpack(struct mdt_thread_info *info)
6. static int mdt_rename_unpack(struct mdt_thread_info *info)

   These methods also call mdt_dlmreq_unpack as shown above.

6 Protocol, APIs, disk format.

No more changes added. However, the wire format change is defined differently.

1. mds_reint_create_*_client[]

   The new buffer is added to the end, as there are create formats with different amount of buffers, the result create format with DLM_REQ will always have all the previous buffers, some of which may be zero length. They are:

```
static const struct req_msg_field *mds_reint_create_rmt_acl_client[]={
    &RMF_PTLRPC_BODY,
    &RMF_REC_CREATE,
    &RMF_CAPA1,
    &RMF_NAME,
    &RMF_EADATA,
    &RMF_DLM_REQ
};
static const struct req_msg_field *mds_reint_create_sym_client[] = {
    &RMF_PTLRPC_BODY,
    &RMF_REC_CREATE,
    &RMF_CAPA1,
    &RMF_NAME,
    &RMF_SYMTGT,
    &RMF_DLM_REQ
};
```

2. mds_reint_unlink_client[], mds_reint_link_client[], mds_reint_rename_client[], mds_reint_setattr_client[]

   These formats get new RMF_DLM_REQ buffer at the end of their buffer lists, the same as shown above in mds_reint_create_rmt_acl_client[].

7 Scalability and performance.

The Early Lock Cancel mechanism introduces a LDLM improvement that may reduce the amount of RPC for particular workloads. If a client is performing a filesystem operation (e.g. unlink) and has some locks conflicting with this operation, early lock cancellation and batching all the canceled locks into the LDLM request to the server saves 2 RPC per lock – blocking and canceling RPC.

However it also may happen that the performance degrades in some cases. If a lock is canceled locally beforehand and this lock is immediately needed by

another thread, another enqueue RPC for the same lock is sent to the server and if this enqueue RPC reaches the server before the original RPC with the batched cancels, the secondary lock is obtained on the client. This lock will be canceled in the usual order, therefore instead of 2 RPC speedup there will be the 1 enqueue RPC slowdown.

It is possible that Early Lock Cancel will be slower in CMD environment, in case when asynchronous RPC sent to other servers do not get in time due to some (network) reasons and MDS will have to wait for blocking RPC (which will be sent to client and client will return ESTALE as the locks are already canceled there). Although it is likely to be a rare case, because the MDS we send the main RPC will also have to communicate with that MDS we send asynchronous RPC to.

8 Test plan.

1. Unlink of cross-referenced file.
   (a) create a directory and a cross-referenced file in it.
   (b) stat the directory and the file – obtain LOOKUP,UPDATE locks on them.
   (c) unlink the file.
   (d) check there is no blocking RPC sent from the server that holds the file during unlink.
2. Link cross referenced file, rename one cross-referenced file to another cross reference file.
   (a) Similar to the previous test, if these redundant tests are needed.

In addition to the previous Early Lock Cancel work, the following tests should be implemented:

1. Unlink a file, destroy its objects.
   (a) create a file, write to it.
   (b) stat parent directory and the file – obtain LOOKUP,UPDATE locks on them.
   (c) unlink the file.
   (d) check there is no cancel RPC sent from client to MDS or OST during unlink.
2. Rename a file.

(a) create 2 directories and a file in each.

(b) stat both directories and both files – obtain LOOKUP, UPDATE locks on them.

(c) mv one file to the other file.

(d) check there is no cancel RPC sent from client during rename.

3. Create a directory.

(a) create a directory.

(b) stat the directory – obtain LOOKUP, UPDATE locks on it.

(c) mkdir another directory in it.

(d) check there is no cancel RPC sent from client during the child mkdir.

4. Link a file.

(a) create 2 directories and a file in one of them.

(b) stat directories and the file – obtain LOOKUP,UPDATE locks on them.

(c) hard link the file from the other directory.

(d) check there is no cancel RPC sent from client during link.

5. Setattr a file.

(a) create a file.

(b) stat it – obtain LOOKUP, UPDATE locks on it.

(c) chmod the file

(d) check there is no cancel RPC sent from client during chmod.

6. Create a file.

(a) create a directory.

(b) stat the directory – obtain LOOKUP, UPDATE locks on it.

(c) create a file in it.

(d) check there is no cancel RPC sent from client during create.

9 Recovery.

No recovery implications.

10 Alternatives.