LRU Resize DLD

Yury Umanets, Vitaly Fertman

10th April 2007

Contents

1 Introduction

This DLD is intended to fix the following issues:

1. Exhausted client LRU causes one lock cancel on each new lock taken by client. This doubles number of RPCs in some work loads what loads network and servers needlessly;
2. Automatically adjust number of locks in cluster.

To achieve this, we implement locks pool on server side which provides accounting and shrinking policies for all locks in cluster. Next sections describe API functions and details of pool implementation. Algorithm itself is described in HLD, read it first.

1.1 Glossary

Here is some glossary of terms used in this DLD.

L - locks limit, allowed amount of locks to be granted by server;

G - number of locks granted, we want to manage **G == L**;

T - period of time (in milliseconds) that we use for tracking any changes in number of locks granted by server or canceled by clients. In other words, **T** is step or algorithm tick;

K - correction factor for **SLV** (see below). Initial value of **K = 1**. Later, for each step it is calculated like this: **K = L - (G - GP)**;

GP - planned number of granted locks after next step. It is calculated this way: **GP = G + (L - G) / 10**. Its main point is to not allow grant locks so fast that they may exhaust **L** quickly during one or two **T**;

GR - grant rate, that is, number of locks granted for period **T**;

CR - cancel rate, that is, number of locks canceled for period **T**;

GS - grant speed, number **GR - CR** for period **T**;

LA - on client, for each lock in LRU - lock age (in seconds);

GSL - grant speed limit. If this is exceeded - emergency actions should take place as described;

SLV - server lock volume, the indication of current locking situation on server. Its meaning is similar to "flow" definition in statistical physics and means how many locks may be kept for particular amount of time. Server sends **SLV** to clients and clients use it to make decision whether some locks should be canceled. Initial **SLV** value is chosen to be big enough and not to cause lock cancels on client. For all next **T**-periods, it is calculated like

this: **SLV** = (**SLV** * **K**) / **L**. In the case of big load (when **GS** > **GSL**), **SLV** may be corrected some way. We propose simple division by 2 as an before-testing value;

LRU - current client LRU size;

CLV - client lock volume, the value calculated for each lock in client's LRU and compared with **SLV**. It is calculated this way: **CLV** = **F** * **LRU** * **LA.** Its meaning is: is this lock allowed to leave on client taking into account its age and LRU size. If we try to be coherent with "flow" paradigm like in **SLV** definition case, meaning of **CLV** is this: is this lock fit to allowed "flow" or already is outside of it and should be removed. See also below for details of what **F** is;

F - on client side, when calculating **CLV** we want to have one more factor for adjusting **CLV** for all locks and this way to cancel more or less locks than server requests to. See below for example of how **CLV** is calculated using **F**. For most of clients **F** = **1**. In some cases, however, when we want to achieve some effect like allow more/less locks stay on client, it may be 1.5 or 0.5.

2 Functional Specification

2.1 Client side

There is one main task on client side: check LRU, calculate **CLV** for every lock and compare it with last **SLV** from server. If lock **CLV** > **SLV** - remove this lock from LRU.

To implement this behavior on regular installment, we need special LRU resize thread on client. It should start in mount time and stop when lustre gets unmounted. It should check all namespaces on client and apply the policy as said above for each lock in each namespace. Read section 4 for implementation details.

2.2 Server side

The main task of ldlm pool on server side is to provide regular re-calculating of all values used in making decision of what **SLV** should be. This may be done by means of introducing special re-calculating thread, which will do that on time regular installment (every **T**). We chose this way as it provides as more regular calculations then doing it on behalf of ldlm activities (enqueue, cancel, etc.).

2.2.1 struct ldlm_pool

Structure *ldlm_pool* contains accounting related fields and defined as the following:

```
struct ldlm_pool {
        __u32                   pl_T;         /* Thread recalc period. */
        __u32                   pl_L;         /* Number of allowed locks. */
        atomic_t                pl_G;         /* Number of granted locks. */
        atomic_t                pl_GR;        /* Grant rate per T. */
        atomic_t                pl_CR;        /* Cancel rate per T. */
        atomic_t                pl_GS;        /* Grant speed (GR - CR) per T. */
        atomic_t                pl_GP;        /* Grant plan for next T. */
        __u32                   pl_GSL;       /* Grant speed limit. */
        atomic_t                pl_SLV;       /* Server lock volume. */
        struct ptlrpc_thread    pl_thread;    /* Thread control. */
        char                    pl_name[100]; /* Pool name. */
};
```

pl_T - refers to **T** from glossary in section 1.1;

pl_L - refers to **L** from glossary in section 1.1;

pl_G - refers to **G** from glossary in section 1.1;

pl_K - refers to **K** from glossary in section 1.1;

pl_GS - refers to **GS** from glossary in section 1.1;

pl_GR - refers to **GR** from glossary in section 1.1;

pl_CR - refers to **CS** from glossary in section 1.1;

pl_GSL - refers to **GSL** from glossary in section 1.1;

pl_SLV - refers to **SLV** from glossary in section 1.1;

pl_thread - pool re-calculating thread;

pl_name - pool name and pool thread name.

2.2.2 ldlm_pool_init()

Parameters

```
int
ldlm_pool_init(struct ldlm_pool *pl)
```

pl - pointer to allocated *struct ldlm_pool*.

Return value

Function returns zero on success and error code otherwise.

Description

This function initializes *struct ldlm_pool* which is the essence of LDLM pool, containing locks accounting and shrinking policies. Pool initializing includes setting default state for all counters and policy related value like lock volume, correcting factor, etc.

One more task for this function is to start special thread which is recalculating **SLV** on regular installment.

2.2.3 ldlm_pool_fini()

Parameters

```
void
ldlm_pool_fini(struct ldlm_pool *pl)
```

pl - pointer to allocated *struct ldlm_pool.*

Return value

No return value is needed.

Description

This function finalizes LDLM pool. As it does not free @pl memory it only may stop threads (if any) and zero out some fields of @pl. This function also stops pool thread which is re-calculating **SLV**.

2.2.4 ldlm_pool_add_lock()

Parameters

```
void
ldlm_pool_add_lock(struct ldlm_pool *pl, struct ldlm_lock *lock)
```

pl - pointer to allocated *struct ldlm_pool;*

lock - just granted ldlm lock.

Return value

No return value is needed.

Description

This function notifies pool @pl that @lock is just granted and may be taken into account;

2.2.5 ldlm_pool_del_lock()

Parameters

```
void
ldlm_pool_del_lock(struct ldlm_pool *pl, struct ldlm_lock *lock)
```

pl - pointer to allocated *struct ldlm_pool;*

lock - just canceled ldlm lock.

Return value

No return value is needed.

Description

This function notifies pool @pl that @lock is just canceled and may be taken into account;

3 Use Cases

Here are some use cases, which may easily be converted to test scripts for checking various SLV change scenarios.

3.1 Growing SLV case

As **SLV** is reflecting server's ability to handle more locks, it should be growing while number of granted locks per **T** (grant speed **GS**) is getting smaller. We would like to check this.

1. Mount one lustre client, create directory */mnt/lustre/1a*;
2. Find out **L** using proc interface */proc/fs/lustre/ldlm/server_pool/limit*. Create **L** files in */mnt/lustre/1a*;
3. Create **L** files using "touch" command. This should produce **L** locks, what may be checked via ldlm pool proc interface at */proc/fs/lustre/ldlm/server_pool/**. As number of locks increasing and reaching **L**, **SLV** is going down;
4. When **L** files are created, save **SLV** from */proc/fs/lustre/ldlm/server_pool/server_lock_volume*;
5. Remove files or umount lustre. This should not cause **SLV** drop, **SLV** should be growing, so that after files are removed, */proc/fs/lustre/ldlm/server_pool/server_lock_volume* should not be smaller than saved value.

3.2 Dropping SLV case

SLV should start going down when **G** is close to **GP**. This means that either pool is getting close to exhausting or grant speed (**GS**) is much higher than grant speed limit (**SGL**). We would like to check this case.

1. Mount one lustre client, create directory */mnt/lustre/1b*;
2. Find **L** via proc interface (*/proc/fs/lustre/ldlm/server_pool/limit*) and create **L** files using "touch" command in */mnt/lustre/1b*;
3. Save **SLV** from */proc/fs/lustre/ldlm/server_pool/server_lock_volume*;
4. Sleep for 5 *** T**, check */proc/fs/lustre/ldlm/server_pool/server_lock_volume*, it should be smaller than saved value, otherwise test is failed.

3.3 Stabilizing SLV case

When pool is exhausted and **SLV** is going down, this means, that client will cancel some amount of locks. While clients cancel locks, locks pressure is getting smaller and **SLV** should stop dropping at some moment and yet a bit later may start growing again. Let's check this.

1. Mount one lustre client and create directory */mnt/lustre/1c*;
2. Find **L** via proc interface (*/proc/fs/lustre/ldlm/server_pool/limit*) and create **L** files using "touch" command in */mnt/lustre/1c*;
3. Save **SLV** from */proc/fs/lustre/ldlm/server_pool/server_lock_volume*;
4. Sleep for 30 *** T**, check */proc/fs/lustre/ldlm/server_pool/server_lock_volume*, it should be same or greater than saved value.

4 Logic Specification

This section describes how some key functions may be implemented. In two words, main tasks are these:

4.1 Client side

- Check LRU on time basis and for every lock calculate its **CLV** as the following: **CLV = F * LA * LRU_SIZE**. See section 1.1 for details of **CLV** calculating components. Then compare it with last **SLV**. Locks with **CLV** greater than **SLV** should be canceled immediately;
- To provide reliable mechanism of delivering **SLV** to client, we want not to rely on enqueue or cancel RPCs which client issues in its work. Rather we want to use all other RPCs for delivering **SLV** to clients. Additionally, special thread on client should make sure, that **SLV** is not more than

some used delta (**D**) old. In the case **SLV** is older than **D** (this is the case when FS had no activities) client should issue empty cancel RPC to refresh **SLV**. See below for details of how **D** is calculated (section 4.1.1);

- Remove some amount of locks on memory pressure event. See below for details (section 4.1.2).

4.1.1 SLV requests frequency

There are possible scalability issues related to the fact that in big clusters clients may ask for SLV too often, so that, server and network will be overloaded with empty cancel RPCs. To solve this issue, we want use some interval to ask server for new SLV from a client. Apparently this interval should depend on number of locks kept by client.

Here is the formula:

```
D = L / LRU_SIZE * C
```

Where default value for **C** is proposed to be as following:

```
C = 1 / 10
```

Having **C** is useful due to the following:

1. We do not want to load network much even in the case that some client keeps big number of locks;
2. It is always useful to have additional tunable which may be changed via /proc so that behavior may be changed without re-compilation.

This implies, that client should know what the **L** for each server is. This means, that server should send to clients not only **SLV**, but also **L** in all/many RPCs. Sending it one time in connecting is not enough, as **L** may be changed on server via proc later.

4.1.2 Memory pressure

As was said above, we want to remove some number of locks from clients LRU in the memory pressure cases. This is absolutely required, because otherwise client may be overloaded by locks which are not longer limited by LRU size on client and server SLV may be big enough because it is calculated without taking into account client nodes RAM and other capabilities. And cluster may contain various clients. All this clearly shows, that we need to remove locks in OOM cases.

Memory pressure cases may be handled by means of registering LRU list (or locks cache) shrink-er by *set_shrinker()* method. This is special callback function, which gets control from Linux VM in the case of memory pressure and supposed to cancel some amount of locks. It works on principle of estimating

the cost (in disk seeks) of getting one cache object from its storage to the cache (that is to RAM). So that, when object cost is not high, VM tends to remove it from cache more often and better leave more expensive objects in their caches in other kernel subsystems.

In the light of what was said above, here is the question, how to estimate lock cost in disk seeks. We propose to estimate it as DEFAULT_SEEKS (number of seeks to get inode into memory). Rationale is the following:

1. Network cost is almost zero;
2. On server side, lock is mostly taken on behalf of some disk activities, like getting object attributes, this is almost the same as getting inode attributes in local filesystem.

LRU shrinker callback may be implemented into two different ways (see below). See section 4.1.3for example of shrinker callback implementation.

Enforcing F in CLV

We may enforce **F** for all namespaces and run *ldlm_cli_pool_recalc()* voluntary. So that, **CLV** for all locks will be higher than **SLV** from server and additional amount of locks (not only those according to **SLV**) will be canceled.

This way has advantage is that, locks are canceled natural way, that is, according to its age, so that, fresh locks will stay in tact. But this way is a bit complex. Some questions are left: how much should we change **F**? What to do with **F** at finish, return it to 1 again?

Simple removing

Alternatively, shrinker may be implemented without changing **F** and instead just remove 30% (adjustable number) of locks. This should be done with taking into account the age of locks, so that, to remove locks starting from most old ones.

This way is more simple but less clever and may cause some unexpected issues.

4.1.3 ldlm_cli_pool_shrink()

Shrinker is rather simple and looks like this:

```
static int
ldlm_cli_pool_shrink(int nr, unsigned int gfp_mask)
{
        struct ldlm_namespace *ns;
        int left, cached = 0;

        if (nr != 0 && !(gfp_mask & __GFP_FS))
                return -1;
```

```
        mutex_down(&ldlm_namespace_lock);
        list_for_each_entry(ns, &ldlm_namespace_list,
                            ns_list_chain)
        {
                cached += ns->ns_nr_unused;
        }

        if (nr == 0) {
                mutex_up(&ldlm_namespace_lock);
                return cached;
        }

        list_for_each_entry(ns, &ldlm_namespace_list,
                            ns_list_chain)
        {
                 /* Cancel the percent of @nr locks that
                  * belongs to this @ns. */
                 rc = ldlm_cancel_lru(ns, LDLM_ASYNC,
                                      ns->ns_nr_unused * nr / cached);
                 if (rc) {
                         LDLM_DEBUG(...);
                 }

                 left += ns->ns_nr_unused;
        }
        mutex_up(&ldlm_namespace_lock);
        return left;
}
```

4.1.4 ldlm_cli_pool_thread()

This is thread which checks client LRU from time to time and removes old locks from it.

```
static int ldlm_cli_pool_thread(void *arg)
{
        struct ldlm_pool *pl = (struct ldlm_pool *)arg;
        struct ptlrpc_thread *thread = ctl->thread;
        ENTRY;

        cfs_daemonize(pl->pl_name);

        /* Record that the thread is running */
        thread->t_flags = SVC_RUNNING;
        cfs_waitq_signal(&thread->t_ctl_waitq);
```

```
        while (1) {
                ldlm_cli_pool_recalc(pl);

                /* Wait until the next check time, or until we're
                 * stopped. */
                time_to_next_check = cfs_time_sub(cfs_time_add(this_check,
                    cfs_time_seconds(LUSTRE_CLI_LRU_T)), cfs_time_current());

                CDEBUG(D_INFO, "next check in "CFS_DURATION_T"
                       ("CFS_TIME_T")\n", time_to_next_check,
                       cfs_time_add(this_check,
                       cfs_time_seconds(LUSTRE_CLI_LRU_T)));

                if (time_to_next_check > 0) {
                        lwi = LWI_TIMEOUT(max_t(cfs_duration_t,
                            time_to_next_check, cfs_time_seconds(1)),
                            NULL, NULL);

                        l_wait_event(thread->t_ctl_waitq,
                                     thread->t_flags &
                                     (SVC_STOPPING|SVC_EVENT),
                                     &lwi);

                        if (thread->t_flags & SVC_STOPPING) {
                                thread->t_flags &= ~SVC_STOPPING;
                                break;
                        } else if (thread->t_flags & SVC_EVENT) {
                                thread->t_flags &= ~SVC_EVENT;
                        }
                }
        }

        thread->t_flags = SVC_STOPPED;
        cfs_waitq_signal(&thread->t_ctl_waitq);

        CDEBUG(D_NET, "client pool thread exiting, process %d\n",
               cfs_curproc_pid());

        RETURN(0);
}
```

4.1.5 ldlm_cli_pool_recalc()

This function performs actual CLV calculating and removing all old locks from LRU. It may be implemented like this:

```
#define LUSTRE_CLI_LRU_T 1

void
ldlm_cli_pool_recalc(struct ldlm_pool *pl) {
        CFS_LIST_HEAD(head);
        struct ldlm_namespace *ns;
        struct list_head *tmp;
        cfs_time_t time;
        int rc;
        ENTRY;

        /* Check all namespaces. */
        mutex_down(&ldlm_namespace_lock);
        list_for_each_entry(ns, &ldlm_namespace_list, ns_list_chain) {
                struct obd_export *exp = NULL;

                /* Check the update has happened within last
                 * LUSTRE_CLI_LRU_T. */
                time = cfs_time_sub(cfs_time_current(), ns->ns_slv_time);
                time = cfs_time_sub(time, cfs_time_seconds(LUSTRE_CLI_LRU_T));
                if (time < 0)
                        continue;

                /* Cancel aged locks. */
                rc = ldlm_cancel_lru(ns, LDLM_ASYNC);
                if (rc)
                        continue;

                spin_lock(&ns->ns_hash_lock);
                /* Find the export for this NS. */
                list_for_each(tmp, &ns->ns_unused_list) {
                        exp = list_entry(tmp, struct ldlm_lock,
                                         l_lru)->l_conn_export;
                        if (exp != NULL)
                                break;
                }
                spin_unlock(&ns->ns_hash_lock);

                /* Nothing to cancel within LUSTRE_CLI_LRU_T. Send an empty
                 * cancel to obtain up-to-date SLV. */
                rc = ldlm_cli_cancel_req(exp, &head, 0, 0);
                if (rc == 0) {
                        /* XXX: update ns_slv & ns_slv_time */
                }
        }
        mutex_up(&ldlm_namespace_lock);
```

```
}
```

4.2 Server side

- Re-calculate **K** and **SLV** every **T** seconds according to current **L**, **G** and **GS**;
- Send current **SLV** and **L** to all clients when they ask for using MD related RPCs;
- Cancel some amount of locks on memory pressure event. Memory pressure handler is registered with *set_shrinker()* function and follows its interface policies as for number of cached objects, etc. See example of memory pressure function for client side (section 4.1.3), on server it may looks very similar.

4.2.1 Pool logic

Server side ldlm pool component is working like the following. In server init time (regular start-up or start-up in recovery) it is being initialized by means of using *ldlm_pool_init()*. For granted and canceled locks accounting purposes it has two functions: *ldlm_pool_add_lock()* and *ldlm_pool_del_lock()*. Ldlm should call them in the following cases:

- *ldlm_pool_add_lock()* - when lock is granted or replied in recovery. This increases granted locks (**G**), grant rate (**GR**) and grant speed (**GS**);
- *ldlm_pool_del_lock()* - when lock is canceled. This function decreases **G**, cancel rate (**CR**) and **GS**.

So that, pool always has up-to-date information about how many locks are granted, canceled and what is grant speed in last **T**.

There is special pool thread *ldlm_srv_pool_thread()*, which is re-calculating **SLV** every **T** seconds according to current number of granted locks. Calculated **SLV** is always accessible for sending to clients. Re-calculating is actually done by *ldlm_srv_pool_recalc()* function which is called from *ldlm_srv_pool_thread()*.

In the time when server is going to tear down, it calls *ldlm_srv_pool_fini()*. Main task here is to stop re-calculating thread. For details of implementing all described functions, please look below.

4.2.2 ldlm_pool_init()

In this section, function *ldlm_pool_init()* is described in details. Its tasks are the following:

- Set default values for all fields in pool;
- Start calculating thread which is intended to do re-calculation for all values each **T** milliseconds.

```
/* One second for T. */
#define LDLM_POOL_DEF_T (1)

/* 100 ldlm locks for 1MB of RAM. */
#define LDLM_POOL_DEF_L (num_physpages >> (20 - PAGE_SHIFT)) * 100

/* 5% is limit for GS, and thus, this is default GSL. */
#define LDLM_POOL_DEF_GSL(L) ((L) * 5 / 100)

/* 5% of all locks is default GP. */
#define LDLM_POOL_DEF_GP(L) ((L) * 5 / 100)

/* Max age for locks on clients (10 hrs in seconds). */
#define LDLM_POOL_MAX_AGE   (36000)

static inline __u64 LDLM_POOL_SLV_MAX(__u32 L)
{
        /* Allow to have 100% of locks for 1 client for 10 hrs */
        __u64 lim = L *  LDLM_POOL_MAX_AGE/* 10 hrs in sec */
                 / 1/* client */;

        return lim;
}

static inline __u64 LDLM_POOL_SLV_MIN(__u32 L)
{
        return 1;
}

int ldlm_pool_init(struct ldlm_pool *pl, __u32 client)
{
        ENTRY;

        /* Nothing granted so far. */
        atomic_set(&pl->pl_G, 0);

        pl->pl_T = LDLM_POOL_DEF_T;

        atomic_set(&pl->pl_GP, 0);
        atomic_set(&pl->pl_GS, 0);
        atomic_set(&pl->pl_GR, 0);
        atomic_set(&pl->pl_CR, 0);
        pl->pl_L = LDLM_POOL_DEF_L;
        pl->pl_GSL = LDLM_POOL_DEF_GSL(LDLM_POOL_DEF_L);
        atomic_set(&pl->pl_GS, LDLM_POOL_DEF_GP(LDLM_POOL_DEF_L));
        atomic_set(&pl->pl_SLV, LDLM_POOL_DEF_SLV(LDLM_POOL_DEF_L));
```

```
        if (client == LDLM_NAMESPACE_SERVER) {
                strncpy(pl->pl_name, "server_pool", sizeof(pl->pl_name));
        } else {
                strncpy(pl->pl_name, "client_pool", sizeof(pl->pl_name));
        }

        pl->pl_client = client;
        pl->pl_proc_dir = lprocfs_register(pl->pl_name,
                                           ldlm_type_proc_dir,
                                           NULL, NULL);
        if (IS_ERR(pl->pl_proc_dir)) {
                CERROR("LProcFS failed in ldlm-pool-init\n");
                rc = PTR_ERR(pl->pl_proc_dir);
                RETURN (rc);
        }

        var_name[MAX_STRING_SIZE] = '\0';
        memset(pool_vars, 0, sizeof(pool_vars));
        pool_vars[0].name = var_name;

        if (client == LDLM_NAMESPACE_SERVER) {
                snprintf(var_name, MAX_STRING_SIZE, "server_lock_volume");
                pool_vars[0].data = &pl->pl_SLV;
                pool_vars[0].read_fptr = lprocfs_rd_atomic;
                lprocfs_add_vars(pl->pl_proc_dir, pool_vars, 0);

                /* Continue initialize lproc stuff here */

                ...
        }

        rc = ldlm_pool_thread_start(pl);
        if (rc)
                CERROR("Can't start pool thread, rc %d\n", rc);

        RETURN(rc);
}
```

4.2.3 ldlm_pool_fini()

In this section we show how *ldlm_pool_fini()* may be implemented.

```
void ldlm_pool_fini(struct ldlm_pool *pl)
{
        ENTRY;
```

```
        ldlm_pool_thread_stop(pl);
        if (pl->pl_proc_dir)
                lprocfs_remove(&pl->pl_proc_dir);
        EXIT;
}
```

4.2.4 ldlm_srv_pool_thread()

This is main function of ldlm pool. It re-calculates **SLV** and others for every **T**.

```
static int ldlm_srv_pool_thread(void *arg)
{
        struct ldlm_pool *pl = (struct ldlm_pool *)arg;
        struct ptlrpc_thread *thread = &pl->pl_thread;
        ENTRY;

        cfs_daemonize(pl->pl_name);

        /* Record that the thread is running */
        thread->t_flags = SVC_RUNNING;
        cfs_waitq_signal(&thread->t_ctl_waitq);

        while (1) {
                ldlm_srv_pool_recalc(pl);

                /* Wait until the next check time, or until we're
                 * stopped. */
                time_to_next_check = cfs_time_sub(cfs_time_add(this_check,
                    cfs_time_seconds(pl->pl_T)), cfs_time_current());

                CDEBUG(D_INFO, "next check in "CFS_DURATION_T
                       "("CFS_TIME_T")\n", time_to_next_check,
                       cfs_time_add(this_check, cfs_time_seconds(pl->pl_T)));

                if (time_to_next_check > 0) {
                        lwi = LWI_TIMEOUT(max_t(cfs_duration_t,
                            time_to_next_check, cfs_time_seconds(1)),
                            NULL, NULL);

                        l_wait_event(thread->t_ctl_waitq,
                                     thread->t_flags &
                                     (SVC_STOPPING|SVC_EVENT),
                                     &lwi);

                        if (thread->t_flags & SVC_STOPPING) {
```

```
                                thread->t_flags &= ~SVC_STOPPING;
                                break;
                        } else if (thread->t_flags & SVC_EVENT) {
                                thread->t_flags &= ~SVC_EVENT;
                        }
                }
        }

        thread->t_flags = SVC_STOPPED;
        cfs_waitq_signal(&thread->t_ctl_waitq);

        CDEBUG(D_NET, "pool thread exiting, process %d\n",
               cfs_curproc_pid());

        RETURN(0);
}
```

4.2.5 ldlm_srv_pool_recalc()

This function changes **SLV** every **T** according to current number of granted locks **G** and grant speed **GS**.

```
static void ldlm_srv_pool_recalc(struct ldlm_pool *pl)
{
        int k, l, g, ogp, gs, gsl;
        __u32 slv, ngp;
        __u64 slv_max;
        __u64 slv_min;
        ENTRY;

        l = pl->pl_L;
        gsl = pl->pl_GSL;

        g = atomic_read(&pl->pl_G);
        gs = atomic_read(&pl->pl_GS);
        ogp = atomic_read(&pl->pl_GP);

        /* Zero out grant/cancel rates and speed for this T. */
        atomic_set(&pl->pl_GR, 0);
        atomic_set(&pl->pl_CR, 0);
        atomic_set(&pl->pl_GS, 0);

        ngp = g + ((l - g) * 5) / 100;

        k = l - (g - ogp);
        if (k <= 0)
```

```
                k = 1;

        slv = (atomic_read(&pl->pl_SLV) * ((k * 100) / l));
        slv = div_round_up(slv, 100);
        slv_max = LDLM_POOL_SLV_MAX(pl->pl_L);
        slv_min = LDLM_POOL_SLV_MIN(pl->pl_L);

        if (slv > slv_max) {
                CDEBUG(D_NET, "Correcting SLV to allowed max %llu\n",
                       slv_max);
                slv = slv_max;
        } else if (slv < slv_min) {
                CDEBUG(D_NET, "Correcting SLV to allowed min %llu\n",
                       slv_min);
                slv = slv_min;
        } else {
                if (gs > gsl) {
                        CDEBUG(D_NET, "Locks loading is too high "
                               "(GS %d > GSL %u)\n", gs, gsl);
                        slv = slv / 2;
                }
        }

        atomic_set(&pl->pl_SLV, slv);
        atomic_set(&pl->pl_GP, ngp);

        EXIT;
}
```

As **SLV** tends to grow when ldlm is idle, we want to correct it every time when we calculate it. We do not allow it to exceed some max value *LDLM_POOL_SLV_MAX()*. As well, it should not be smaller than *LDLM_POOL_SLV_MIN()*.

LDLM_POOL_SLV_MAX() is chosen to have such value, that 100% of locks are allowed to be kept on one client for 10 hrs. And *LDLM_POOL_SLV_MIN()* is 1 to provide the correctness of **SLV** self growing according to its formula in glossary (Section 1.1).

One more correction is done when grant speed exceeds grant speed limit. If that happens, we know that way to many locks are granted in last T and want to ask clients in same time to cancel many locks, so that, pool balance is preserved. In this time, we divide **SLV** by factor 2, so that, we ask clients roughly speaking cancel all locks which are 2 times older than more recent lock.

4.2.6 ldlm_pool_add_lock()

This section shows how *ldlm_pool_add_lock()* re-calculates pool fields according to changed situation.

```
void ldlm_pool_add_lock(struct ldlm_pool *pl,
                        struct ldlm_lock *lock)
{
        ENTRY;
        atomic_inc(&pl->pl_G);
        atomic_inc(&pl->pl_GS);
        atomic_inc(&pl->pl_GR);
        EXIT;
}
```

4.2.7 ldlm_pool_del_lock()

This section shows how *ldlm_pool_del_lock()* re-calculates pool fields according to changed situation.

```
void ldlm_pool_del_lock(struct ldlm_pool *pl, struct ldlm_lock *lock)
{
        ENTRY;
        LASSERT(atomic_read(&pl->pl_G) > 0);
        atomic_dec(&pl->pl_G);
        atomic_dec(&pl->pl_GS);
        atomic_inc(&pl->pl_CR);
        EXIT;
}
```

As it can be seen, we use atomic values in pool and do not protect calculations in *ldlm_srv_pool_recalc()* with some lock. Rationale is the following:

- We do not protect changing **G**, **CR, GR** and **GS** in *ldlm_pool_add_lock()* and *ldlm_pool_del_lock()* and re-calculating in *ldlm_srv_pool_recalc()* because we want to preserve as much as possible of parallel work for the pool and to not create bottle neck;
- Having small deviation of **GS**, **CS** and **G** in the process of re-calculating is not so important here, it will not affect final **SLV** much and **SLV** may be not as precise as in drug store;
- Atomic type guarantees, that values will be sane all the time.

4.2.8 ldlm_pool_thread_start()

This function starts re-calculation thread. It may be implemented like this:

```
static int
ldlm_pool_thread_start(struct ldlm_pool *pl)
{
#ifdef __KERNEL__
```

```
        int (*pool_thread)(void *);
        struct l_wait_info lwi = { 0 };
        int rc;
        ENTRY;

        pool_thread = pl->pl_client == LDLM_NAMESPACE_SERVER ?
                ldlm_srv_pool_thread : ldlm_cli_pool_thread;

        cfs_waitq_init(&pl->pl_thread.t_ctl_waitq);

        /* CLONE_VM and CLONE_FILES just avoid a needless copy, because we
         * just drop the VM and FILES in ptlrpc_daemonize() right away. */
        rc = cfs_kernel_thread(pool_thread, pl, CLONE_VM | CLONE_FILES);
        if (rc < 0) {
                CERROR("Cannot start pool thread: %d\n", rc);
                RETURN(rc);
        }

        l_wait_event(pl->pl_thread.t_ctl_waitq,
                     pl->pl_thread.t_flags & SVC_RUNNING, &lwi);
        EXIT;
#endif
        return 0;
}
```

4.2.9 ldlm_pool_thread_stop()

And this function stops re-calculating thread.

```
static void
ldlm_pool_thread_stop(struct ldlm_pool *pl)
{
#ifdef __KERNEL__
        struct l_wait_info lwi = { 0 };
        ENTRY;

        pl->pl_thread.t_flags = SVC_STOPPING;
        cfs_waitq_signal(&pl->pl_thread.t_ctl_waitq);

        l_wait_event(pl->pl_thread.t_ctl_waitq,
                     (pl->pl_thread.t_flags & SVC_STOPPED), &lwi);
        EXIT;
#endif
}
```

5 State Specification

5.1 Recovery

The point of discussion here is how all this new schema will behave in recovery. Will it return to the state as it was just before recovery when recovery finishes and all locks are replied?

In this context, to preserve state as it was before recovery, we want to call *ldlm_pool_add_lock()* when lock is replied, so that, when recovery finishes, pool will have same **G** as before it.

In recovery, the following picture will take place:

- When server is up after failure, it will initialize pool and start re-calculation thread, all pool fields are set to initial values in this time and do not reflect real state of cluster. SLV is set to default value - maximal **SLV** here, this means that all clients will not be asked to cancel any locks in first **T** after recovery;
- When all locks are replied, we have the following picture:
  - **G** and **L** are the same as before recovery;
  - **GP** is close to **G**;
  - in process of replaying, **GS** is getting much higher than **GSL**, pool sees this and emergency actions take place on this - **SLV** is getting smaller. When **SLV** is so small that this may cause locks canceling, clients should not do this until recovery is finished;
  - when all is finished, **SLV** is close to the value as it was before recovery. It is possible that **SLV** may be lower than before recovery due to big **GS**. This is not big problem, it will stabilize in few **T**.

6 Environment

6.1 Wire changes

There are is type of wire change applied to many existing RPCs. We want to deliver SLV to clients with many/all MD related RPC, that is, link, unlink, cancel, etc. This means, that we will add additional field to these RPCs.

6.2 Protocol changes

No new RPCs are going to be involved. No RPCs order is changing.

6.3 Network Protocol Compatibility - New clients, Old Servers

New client signals to server, that it supports this feature and uses OBD_CONNECT_LRU_RESIZE flag for that. If server does not support it, it will not return this flag to client in connect time. Having this, client will not look for SLV in enqueue, cancel, whatever RPCs and will act some way like former client did. That is LRU is somewhat limited and locks are removed from it when it gets exhausted.

6.4 Network Protocol Compatibility - Old clients, New Servers

Old client signals to server, that it does not support OBD_CONNECT_LRU_RESIZE feature. Having this, new server will not send SLV to this client. Old client will manage its LRU in old fashioned way, that is, remove old locks from it when it gets exhausted.