LRU Resize HLD (bug 2262)

Yury Umanets, Vitaly Fertman

15th March 2007

Contents

1 Introduction

There is the issue, that clients greedy with metadata, which are usually desktop related activities like a compilation, at some point always have their client side locks LRU exhausted, what causes the number of unpleasant effects. They are the following:

- Each new lock sent by server to client causes cancel RPC to be sent when client's LRU list is exhausted. This doubles number of RPCs what loads network and eats server's and client's CPU needlessly. This is going to be fixed by ELC works soon, but now it is not yet case and all installations require LRU size tuning;
- LRU size requires tinning according to working load, clients behavior, etc., what we would like to get rid of.

This HLD is aiming to do the following:

- To fix described issues by introducing a number of solutions for controlling client's LRU size or rather number of cached locks according to chosen policy;
- To control average number of locks in cluster given to clients by servers and have automatic lock LRU size adjustments.

2 Requirements

Implement such a mechanism which allows the following:

- Control average number of locks in cluster;
- Automatically adjust number of locks on all clients and thus, in cluster itself;
- Eliminate needless cancels due to limited LRU size as described in section 1. That is implement dynamic LRU on clients;
- Allow greedy clients work according to their working pattern, that is, do not force them use minimal number of locks if that does not contradict main invariant - number of locks is in cluster is limited according to chosen policy.

3 Functional specification

To meet requirements the following should be done:

1. Implement locks "pool" on server, its size should be based on servers RAM amount with generous limits;

2. Each lock in pool is "lock resource" which is exhausting each time as server sends locks to client and gets back to "pool" each time as lock is canceled by client or server itself;

3. Introduce an algorithm, that triggers a lock prune when "lock resource" get scarce. Remember that servers can export multiple targets. Lock prune should take into account, that there are "old" locks on some clients which we probably want to prune firstly;

4. On client, use the mechanisms introduced by Vitaly (bug 10589) to cancel more than one lock to send batch cancels from the clients to the servers;

5. On client, avoid situation when LRU size is exhausted and thus, makes client send cancel RPCs for old locks;

6. On client, make sure that "old" locks are canceled on memory pressure event.

4 Use cases

The following use cases are interested to check. To understand them better please read section 5.

4.1 Use cases

4.1.1 Enqueue

1. LDLM on client prepares to send an ENQUEUE RPC to the server, it checks LRU if some locks are redundant according to the current control info and cancel them. Canceled locks are bundled into ENQUEUE, RPC is sent;

2. LDLM on server updates its control info according to the came RPC;

3. LDLM on server prepares a reply and pack there the lock LRU control info too;

4. LDLM on client checks LRU according to new control info obtained and cancels redundant locks from LRU - bundled CANCEL RPC is sent to server.

4.1.2 Blocking AST

1. LDLM on server sends a blocking AST to the client with the packed pool control info;

2. LDLM on client handles LRU as in the above example.

4.2 Test cases

4.2.1 Average time spent for getattr of 1M files

This is main working pattern this HLD is aiming to fix. It means, that LRU on client should be quickly exhausted when we create big set of unique files and get their attributes. This test should show improvement in compare to former solution.

1. Mount lustre with disabled LRU sizing;
2. Create directory "1a" and "1b" in root;
3. Create 1M regular files (using mknod) in directory "1a" and measure time spent;
4. Get attributes of files in "1a";
5. Remount client with enabled LRU sizing;
6. Create 1M regular files (using mknod) in directory "1b" and measure time spent;
7. Get attributes of files in "1b";
8. Compare time spent in both cases, test fails if time spent to getattr files in directory "1b" is greater or equal to time spent for getattr files in directory "1a".

4.2.2 Getattr 1M files after recovery (failed MDT)

In recovery, all in-memory structures are lost and reconstructed later in time of locks resending. This test should check correctness of state reconstruction. For more details see section 6.2.1.

1. Mount lustre with enabled LRU sizing;
2. Create directory "2a" in root;
3. Create 1M regular files (using mknod) in directory "2a";
4. Get attributes of all files in "2a";
5. Save number locks in pool using proc;
6. Fail MDT, wait for recovery finish;
7. Check number of locks in pool. If it is different value than before recovery - test fails.

5 Logical specification

5.1 Overall algorithm

To achieve requirements we propose the following algorithm:

1. Server has locks limit **L** which is the maximal number of locks which may be issued by this server to clients;
2. For each period of time **T**, server calculates its current lock rate **SLR**. Lock rate depends on **L**, granted locks number **G**, grant speed **GS**, etc.;
3. For each **T**, server sends its **SLR** to clients as the indication of current situation;
4. Clients, for each lock in LRU, calculate client lock rate **CLR** to compare it with **last** received **SLR**. If current lock **CLR** is greater than **SLR**, client cancels this lock;
5. For all cases when **L** is getting close to be exhausted or **GS** is increasing more than predefined limit **GSL** (say 5% of pool) we want to react according to situating and signal clients that they should cancel more locks, because server is close to be in trouble. For implementing it, we have **K** which is adjusted on each step **T**. In the case of emergency we can adjust **K** such a way that **SLR** is getting much "stronger", what would cause clients to cancel more locks. See glossary for details of emergency case **K**. Alternatively others actions may be taken;
6. Server uses ping, enqueue and may be other RPCs for delivering **SLR** to clients.

5.2 Glossary

Before we start, we need some glossary to understand all clearly.

L - allowed amount of locks to be granted on one server. This is calculated on server abilities, RAM, may be something else. Proposed number is to have 10 ldlm locks per megabyte, later this value may be adjusted;

G - number of locks granted to clients, we want to keep **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;

K - correction factor for **SLR** (see below). Initial value of **K = 1**. Later, for each step **K = K * (1 - (G - GP) / L)**. In the case of emergency as described above, **K** may be calculated this way: **K = K * (1 - (GP - G) / L) ^ 2**;

GP - planned number of granted locks after next step. **GP = G + (L - G) / 10**;

GS - grant speed, that is, number of locks granted for period **T**;

CS - cancel speed, that is, number of locks canceled for period **T**;

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

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

SLR - server lock rate, the indication of current locking situation on server. Server sends **SLR** to clients and clients use it to make decision whether some locks should be canceled. Initial **SLR** value is chosen to be big enough and not to cause lock cancels on client and in same time not to be such a big to cause exhausting pool in first moments. It probably should be based on **L**. For all next steps (one step is **T**), **SLR = SLR * K**. There should be limit for SLR to not allow it be so bug that may cause issues;

LRU - current client LRU size;

CLR - client lock rate, the value calculated for each lock in client's LRU and compared with **SLR** from server. **CLR = LRU * LA**;

5.3 Implementation details

5.3.1 Server pool details

1. Server implements locks pool. Only this amount of locks (which is pool size) may be sent to clients and cached there. Pool contains locks from different name-spaces. Pool is kind of abstraction here, it does not contain locks their selves, it provides accounting for locks consumption, tracking all changes in **SLR**, **K**, **G**, etc. and policies for adjusting these numbers;

2. Pool does not implement any kind of list, because this requires additional amount of work for server about locking and adding/removing lock to/from list. Additionally, pool which is based on list would be bottle neck what we want to avoid;

3. Pool size **L** is based on server abilities such as RAM amount (see 5.2 for example of **L** definition). Lock in pool is valuable resource and its using we want to track;

4. Each time as server sends a lock to client - it consumes "lock resource" from pool, each time as client or server itself cancels a lock - "lock resource" is returned to pool. These changes are reflected to pool accounting;

5.3.2 Server side

1. Server communicates with clients and informs them of current lock rate using ping, enqueue (and may be others) RPCs. That is, for each **T**, server sends current **SLR** to all clients. Clients are to adjust their LRU lists according to current SLR as described in 5.1;

2. Server grants new lock in all cases, no matter if **L** is exhausted or not. However, there is hard limit **HL** which is intended to not allow to grant lock if server has **HL** exhausted. In this case, server wait until situation is changed and after that lock may be granted;

3. Server does not do anything about pool in the case of memory pressure, because pool consumes a little of memory;

4. Server exposes some of its pool tunable via proc and allows to adjust them. This may be **L, SLR**, etc., what is very handy in testing and pool activities control.

5.3.3 Client side

1. We **do not** want to limit client's LRU size. Client may have such a many locks from server as server allows it to have. As lock on client mostly means cached structure (for example, inode), this means that client will not ask such an amount of locks which are bigger than it may comfortably handle later, because its caches will not grow so much. As clients do not have LRU size limit - they do not do needless cancels when LRU is exhausted;

2. As a client cancels locks according to last **SLR** and calculates own **CLR** using **LA**, it may have more locks than other clients if these locks are "fresh" so that, their **LA** will not allow to cancel them because **CLR** calculated for them is smaller than last **SLR**;

3. Theoretically, one client (if it is only one in cluster) may consume whole server pool. In the case new clients connect in this time and start massive metadata operations, they will share server's pool and after some time they will consume roughly same amount of locks as others if they use same working pattern;

4. As already said, server allows some clients to have more locks if they want to. Client may need them because it does compilation or whatever else and we want to make all clients happy. If we have a lot of clients and they all consume such a many locks that server only does that takes emergency actions (like making **SLR** stronger) - this means, that one server of this particular configuration is not enough for such an amount of clients and their activities. We may track this situation and print warning to system log or console, kind of: *"Lustre: Cluster does not work optimally, upgrade MDS!"*. Same way we can even try to suggest amount of RAM;

5. Clients do not have anything like timeouts for locks or so. Instead, we cancel client's locks in the following cases:

   (a) **CLR** for particular lock is greater than last received **SLR**,

   (b) Memory pressure. This would simplify client part a lot (no timeouts for locks) and also would be in the main line for Linux behavior - in OOM cases all caches are asked to be shrink-ed and clients LRU list is going to be shrink-ed as any other FS caches.

6. Client exposes some of its LRU tunable via proc and allows to adjust them. This may be last **CLR**, LRU size, etc., what is very handy in testing and locking activities control.

5.3.4 Overall picture

Overall picture looks like this:

1. Number of locks is cluster depends on servers abilities and adjusted automatically;

2. Number of locks on clients depends on server abilities, number of clients, client's RAM and client's behavior and adjusted upon memory pressure and/or server request. If there is no memory pressure, good, the more cached locks on clients the better if it does not contradict to server's pool size policies;

3. Server limits number of locks in cluster which were issued by it. If we have 2 or 3 servers (for instance MDS ones), they will have own pools which are also calculated from their RAM amount. All clients would share total amount of locks which may be issued by all servers and probably would be happy. This would allow to work in same cluster servers with different amount of RAM and all they would still do it comfortable. The main point about this algorithm is that, servers do not issue more locks then they may serve later;

6 State management

6.1 Scalability & Performance

Scalability and performance should be better after introducing this change, as it minimizes number of RPCs and CPU using both on client and server. Read section 1 for more details.

6.2 Recovery changes

The following recovery aspects to be covered:

6.2.1 Failed MDT

In the case MDT is failed, all pool counters and in-memory structs are lost and after rebooting will come to default state until all locks from all clients are replied. In reply time, server should reconstruct the state of pool and proceed correctly after that.

6.2.2 Failed client

At the time when client fails, all in-memory structures are lost. Server cancels all locks of failed clients, that is, server states are preserved.

6.3 Wire format changes

As we are going to send **SLR** to clients with ping, enqueue and others, their format is going to change a bit. One more field will be added - **SLR**.

6.4 Protocol changes

No protocol changes are expected as there is no new RPCs or negotiation rules.

6.5 API changes

No API changes are expected. New API related to ldlm pool will be added. It is intended to be used by ldlm to notify pool of locks activities: grant, cancel, touch, etc. Will be described in DLD.