bug11300: Interval Tree for Extent Lock HLD

Huang Wei

2006-12-13

1 Introduction.

The current HLD introduces a performance improvement in Lustre DLM extent lock management. The importance is determined by the size of modern clusters, linear lists of locks become a bottleneck with millions of clients. This improvement has a great impact on large scale systems.This document only present design issues for extent lock, another HLD (***10902 DLM Performance Improvement HLD***) will state other ldlm performance improvement design.

2 Requirements.

2.1 Use-Case requirements.

Parallel reading and writing to same files are much faster: when millions of clients concurrent read or write to one same file, extent-lock enqueue requests can be handled more effectively and faster.

2.2 Functional requirements.

Avoid O(N) lock operations: with current linear lists to link granted locks, either compat operation (ldlm_extent_compat_queue) or policy operation (ldlm_extent_internal_policy) are O(N) time complexity; if with correctly designed interval tree, time complexity of these operations can be decreased to O(LogN).

3 Summary of the solution.

The current implementation of the DLM lock management on the server (MDS, OST) side is based on a common lock list for all the granted locks and a common lock list for all the waiting locks on every resource. Whereas there are usually no

many waiting locks, they have a tendency to be granted finally, it may happen that amount of granted locks becomes huge, e.g. when all the clients get a PR lock on a file, etc.

The goal of the work is to optimize the following actions:

- check if a new lock conflicts with already granted ones;
- gather all the granted locks conflicting with a new one into the separate list of locks;
- find the maximum interval for a new extent lock not-conflicting with the granted locks;

To achieve these goals the proposed implementation uses the following properties:

- the lock has a mode, this is an integer value, and there is a map of mode conflicts.Grouping locks by their mode, lets us know if a group is conflicting with a new lock; For extent lock each group of one mode is organized into a interval tree, see below.
- extent locks are granted on the maximum not conflicting interval rather than on the requested interval. If conflict locks are not canceled, new self-compatible locks gets the same extent intervals. This tendency of self-compatible extent locks to have the same policy let us to form sub-groups of these locks. E.g. there can be many [0;eof] PR extent locks, grouping all of them into 1 sub-group allows us to perform only 1 check for a possible conflict with the new lock for the whole sub-group;
- extent lock represents an interval. Therefore, storing granted extent locks in interval tree will let us perform insert, delete and search operations in O(logN) time. Namely, for each mode, there is a separate extent interval tree to store locks of this mode. Roots of these interval trees are linked in resource's granted queue.

4 Definitions.

Self-compatible lock is a lock of a mode that do not conflict with other locks of the same mode: PR, CW, CR locks do not conflict to each other. It does not allow us to have 2 non-self-compatible locks with the same or overlapping policy.

Non-self-compatible lock is a lock of a mode that may conflict with other locks of the same mode: EX, PW locks may conflict to each other.

policy group For extent lock, it is a sub-group that all locks in the group have the same extent interval

interval tree is one kind of balanced binary tree, augmented from red-black tree. A interval is a pair [a, b] (a<b), In this design, we assume [a, b] is half open, that is, [a, b] represent a set {t∈R: a<= t <b}. New interval tree APIs will be implemented follow the book, while rb tree code can used that from the kernel. more detail of interval tree can be found in chap.14 of book <*introduce to algorithm 2nd*>.

nil[T] we use one nil[T] to represent all the NIL's-all leaves and the root's parent. nil[T] will be used in psuedo-code in logic section .

Fig.1 A normal interval tree

Fig.2 for non-self-compatible mode, no same or overlap intervals in granted queue

Fig.3 for self-compatible mode, interval of extent locks may be same or overlap to each other. for same interval, they are linked into a linear list, for overlap intervals, interval with same start is linked to the left child regardless its end is smaller or larger than current node.

5 Functional specification.

First, linear list are no longer used for granted extent lock, which are replaced by interval tree. so, basic list operations (add, delete, traverse, search) for locks need to be modified to that of interval tree, in turn all functions calling these operation need to be changed for these; Second, usage of @l_res_link for granted locks is changed, so all functions that want to use the @l_res_link to operate on locks need to be changed correctly.

5.1 ldlm lock/resource interfaces

1. **void ldlm_grant_lock(struct ldlm_lock *lock, struct list_head *work_list)**

   This method needs to find a proper place for the new lock in the grant list and inserts the lock: for extent lock, this function will change to call ldlm_grant_extent_lock()

2. void ldlm_grant_extent_lock(struct ldlm_lock *req, struct list_head *work_list)

   This method call ldlm_resource_add_lock_interval() to insert a lock's interval in the interval_tree.

3. void ldlm_lock_cancel(struct ldlm_lock *req)

   Cancels the lock by removing it from the granted lock list: for extent lock, this function will change to call ldlm_resource_unlink_lock_interval() to remove a lock.

4. void ldlm_resource_add_lock_interval(struct ldlm_resource *res, struct list_head *head, struct ldlm_lock *lock)

   This method add one lock into one proper lock mode interval tree with its root linked @head. This function is called when adding one extent lock into resource's granted queue.

5. void ldlm_resource_unlink_lock_interval(struct ldlm_lock *lock)

   This method unlink one lock from one proper lock mode interval tree. This function is called when unlinking one extent lock from resource's granted queue.

6. void ldlm_resource_dump(int level, struct ldlm_resource *res)

   This method is for debug use; This method iterate each lock from each queue of @res, for each lock, dump info of if. for the granted queue, no list traverse again, but the interval tree one replaced.

7. int ldlm_resource_foreach(struct ldlm_resource *res, ldlm_iterator_t iter, void* closure)

   This method iterate each lock from each queue of @res, for each lock, call @iter on it. for the granted queue, no list traverse again, but the interval tree one replaced.

8. static void cleanup_resource(struct ldlm_resource *res, struct list_head *q, int flags)

   This mehod cleanup locks linked by @q, if @q pointed to the granted queue of @res, the method need be modified to use interval tree traverse operation.

5.2 sever-side interfaces

1. static int ldlm_extent_compat_queue(struct list_head *queue, struct ldlm_lock *req, int *flags, ldlm_error_t *err, struct list_head *work_list)

   This method searches conflicts for the extent lock @req in the **@**queue list of extent locks. If **@**work_list is provided, all the conflict locks are gathered into this list, otherwise the @queue is walked through until the first conflict is found.

   Returns: 1 if no conflict found, 0 otherwise.

2. void ldlm_extent_internal_policy(struct list_head *queue, struct ldlm_lock *req, struct ldlm_extent *new_ex)

   Finds the maximum extent interval containing the given @req lock extent interval and not conflicting to other locks.

3. struct ldlm_extent *ldlm_extent_lock_policy(struct ldlm_lock *root, struct ldlm_lock *req)

   this method find the max non-conflict extent in the tree rooted by @root

4. int filter_intent_policy(struct ldlm_namespace *ns, struct ldlm_lock **lockp, void *req_cookie, ldlm_mode_t mode, int flags, void *data)

   Among other activities, this method finds the PW lock with the maximum extent start, and ask the client what is the proper object size.

5.3 client-side interfaces

1. static struct ldlm_lock *search_queue(struct list_head *queue, ldlm_mode_t mode,ldlm_policy_data_t *policy,struct ldlm_lock *old_lock, int flags)

   Finds a matched lock or NULL, match here is define as one existing lock with the same or wider range (for extent) and bits (for inodebits)

   returns a referenced lock or NULL

2. static struct ldlm_lock *search_queue_interval(struct list_head *queue, ldlm_mode_t mode,ldlm_policy_data_t *policy,struct ldlm_lock *old_lock, int flags)

   This method performs specific actions needed to search queue for extent lock.

3. __u64 ldlm_extent_shift_kms(struct ldlm_lock *lock, __u64 old_kms)

   When a lock is cancelled by a client, the KMS may undergo change if this is the "highest lock". This function returns the new KMS value. Caller must hold ns_lock already

4. static int ldlm_cli_cancel_unused_resource(struct ldlm_namespace *ns, struct ldlm_res_id res_id, int flags, void *opaque)

   Cancel all locks on a resource that have 0 readers/writers.

5. int ldlm_cli_join_lru(struct ldlm_namespace *ns, struct ldlm_res_id *res_id, int join)

   join/split resource locks to/from lru list

5.4 interval tree interface

series functions for new added interval data structure, these functions include:

1. int interval_entry(ptr, type, member)

   just like list_entry

2. int interval_iterate(interval_root *T, int (* iter)(struct ldlm_lock *, void *), void *data)

   for each node in the tree, call @iter on the related lock.

3. int interval_insert(interval_root *T, interval_node *x)

   adds the element x, whose interval field is assumed to contain an interval, to the interval tree T.

4. int interval_remove(interval_root *T, interval_node *x)

   removes the element x from the interval tree T.

5. int interval_search(interval_root *T, interval_node *i)

   This method returns a pointer to an element x in the interval tree T such that interval[x] overlaps interval[i], or the sentinel nil[T] if no such element is in the set.

6. interval_node* interval_expand(interval_root *T, interval_node *x)

   This method expand the endpoint of element x to max interval that will not conflict with other intervals in tree, and return the expanded interval. for lustre use, this function will expand an extent lock's extent range to max unconflict interval.

7. int interval_search_all(T, i, void (*func) (interval_node *x))

   This method find all intervals that overlap interval i, and call func to handle resulted intervals one by one. for lustre use, this fuction will find all conflict locks in the granted queue and add these locks to the ast work list.

8. interval_node* interval_next(interval_node *node)

   This method return the successor node to @node in the sorted order determined by an inorder tree walk.

9. interval_node* interval_prev(interval_node *node)

   This method return the predeccessor node to @node in the sorted order determined by an inorder tree walk.

6 Logic specification.

6.1 data struct

Skip lists and one interval tree are added into struct ldlm_lock to optimize the granted lock list handling: l_sl_mode, l_sl_policy skip lists, l_interval tree. l_sl_mode is not used for extent lock, same mode extent lock are linked by a interval tree (l_interval). l_sl_policy are used to gather same mode same interval extent locks into a policy group (so only one such lock's interval is in the interval tree, this lock's l_sl_policy point to the list head of the policy group, other lock's l_interval will be NULL).

\begin{lstlisting}

struct ldlm_lock{

....

struct list_head l_sl_mode; /* not used for extent lock */

struct list_head l_sl_policy; /* used to link same intervals, for self-compatible locks, many locks with same interval is possible */

struct interval_node l_interval;

};

\end{lstlisting}

\begin{lstlisting}

struct interval_node {

__u64 low; /* low endpoint of interval */

__u64 high; /* high endpoint of interval */

__u64 max_high; /* max_high = max(high, left->max_high, right->max_high) */

struct interval_node *parent;

int color;

#define INTERVAL_RED 0

#define INTERVAL_BLACK 1

struct interval_node *left;

struct interval_node *right;

}

\end{lstlisting}

Fig.4 new struct for ldlm_lock, the locks that are the root of interval trees are linked into res->granted_queue by l_res_link (it is used to link all granted locks before this design). l_sl_policy are used to organize and manage policy group. So, the struct of ldlm_resource can be kept unchanged.

6.2 ldlm lock/resource logic

1. void ldlm_grant_lock(struct ldlm_lock *req, struct list_head *work_list)

   - call a proper ldlm_grant_{plain, inodebits, extent, flock}_lock method.

2. void ldlm_grant_extent_lock(struct ldlm_lock *req, struct list_head *work_list)

   Walking through the lock list, performing the following actions:

   - for a lock mode that is not the @req lock mode, skip all the locks of that mode (just skill the interval tree).
   - for a lock mode that is equal to the @req lock mode, call ldlm_resource_add_lock_interval() insert this lock to the proper interval tree.

3. void ldlm_lock_cancel(struct ldlm_lock *req)

   - if flock lock, just remove from the granted list.
   - if @req is inodebite mode, *see <10902_hld.lyx>*
   - if @req is extent lock:
     - for a lock mode that is not the @req lock mode, skip all the locks of that mode (just skip the interval tree).
     - for a lock mode that is equal to the @req lock mode, call ldlm_resource_unlink_lock_interva remove this lock from the proper interval tree.

4. void ldlm_resource_add_lock_interval(struct ldlm_resource *res, struct list_head *head, struct ldlm_lock *lock)

   This method add one lock into one proper lock mode interval tree with its root linked by a list_head. This function is called when adding one extent lock into resource's granted queue.

5. void ldlm_resource_unlink_lock_interval(struct ldlm_lock *lock)

   This method unlink one lock from one proper lock mode interval tree with its root linked by a list. This function is called when unlinking one extent lock from resource's granted queue.

6. void ldlm_resource_dump(int level, struct ldlm_resource *res)

7. int ldlm_resource_foreach(struct ldlm_resource *res, ldlm_iterator_t iter, void* closure)

8. static void cleanup_resource(struct ldlm_resource *res, struct list_head *q, int flags)

   just as described in section 5.1, above three method need be modified to use interval_iterate() to dump, iter, cleanup.

6.3 server-side logic

1. static int ldlm_extent_compat_queue(struct list_head *queue, struct ldlm_lock *req, int *flags, ldlm_error_t *err, struct list_head *work_list)

   - **skip the locks with the compatible modes: if the mode of a @lock is compatible with the @req mode, skip all the locks in the whole interval tree and jump to @lock->l_res_link.next.**
   - if work_list is NULL, then call interval_search, otherwise to call interval_search_all to add all conflicting lock to the work list, if any conflicting lock's l_sl_policy is not NULL(it mean there is policy group in the tree, *see Fig.3*), then also add all elements in this policy group to the work list.

2. void ldlm_extent_internal_policy(struct list_head *queue, struct ldlm_lock *req, struct ldlm_extent *new_ex)

   - skip all the locks with the compatible mode: if the mode of a @lock is compatible with the @req mode, skip all the locks;
   - call ldlm_extent_lock_policy(req)(which in turn call interval_expand()) separately in the the incompatible lock mode trees to expand the req lock's extent range: conclude the max unconflict interval (max(low) and min(high)) from the outputs ;

3. struct ldlm_extent *ldlm_extent_lock_policy(struct ldlm_lock *root, struct ldlm_lock *req)

   call interval_expand to find the max non-conflict extent in the tree rooted by @root

4. int filter_intent_policy(struct ldlm_namespace *ns, struct ldlm_lock **lockp, void *req_cookie, ldlm_mode_t mode, int flags, void *data)

   - skip all the locks with not the PW modes: if a @lock is not the PW mode, skip all the locks in the whole tree and jump to @lock->l_res_link.next, if not NULL;
   - for the interval tree with the PW modes, compare root(tree)->max_high with reply_lvb->lvb_size.

6.4 client-side logic

1. static struct ldlm_lock *search_queue(struct list_head *queue, ldlm_mode_t mode,ldlm_policy_data_t *policy,struct ldlm_lock *old_lock, int flags)

   Finds a matched lock or NULL, match here is define as one existing lock with the same or wider range (for extent) and bits (for inodebits)

   returns a referenced lock or NULL. for extent lock, if it is granted queue, search_queue_interval will be called, otherwise see code as before.

2. static struct ldlm_lock *search_queue_interval(struct list_head *queue, ldlm_mode_t mode,ldlm_policy_data_t *policy,struct ldlm_lock *old_lock, int flags)

   This method performs specific actions needed to search queue for extent lock. approach of search same of wider range (interval) in interval_tree:

   definition:

   max_high[x] : x->max_high

   low[x]: x->low

   high[x]: x->high

   root[T] : root of subtree T

   left[x] : x->left

   righ[x] : x->right

   int[x]: interval of x

   nil[T] : we use one nil[T] to represent all the NIL's-all leaves and the root's parent.

   walk down from the root(T), assume x is current node, i is the interval of @old_lock,

   1.if low[x] <= low[i], tmp_end = max(high[x], max_high[left[x]]), if tmp_end >= high[i], it is to say matched lock found (return the lock with its end equal to tmp_end); else if tmp_end < high[i], x <- right[x], goto 1

   2.if low[x] > low[i], x <- left[x], goto 1

   3.if x is nil[T], i is to say no matched lock

3. __u64 ldlm_extent_shift_kms(struct ldlm_lock *lock, __u64 old_kms)

   with the interval tree (assume root is root[T]) that the @lock's interval are in, compare max_high[root[T]] with old_kms, return the lower one.

4. static int ldlm_cli_cancel_unused_resource(struct ldlm_namespace *ns, struct ldlm_res_id res_id, int flags, void *opaque)

   This method iterate each lock from granted queue of @res that have @res_id, no list traverse again, but the interval tree one replaced.

5. int ldlm_cli_join_lru(struct ldlm_namespace *ns, struct ldlm_res_id *res_id, int join)

   This method iterate each lock from granted queue of @res that have @res_id, no list traverse again, but the interval tree one replaced.

6.5 interval tree logic

1. interval_entry

   #define interval_entry(ptr, type, member) \

   ((type *)((char *)(ptr)-(unsigned long)(&((type *)0)->member)))

2. int interval_iterate(interval_root *T, int (* iter)(struct ldlm_lock *, void *), void *data)

   for each node in the tree, call @iter on the related lock. the algorithm is just the same that normal binary tree does, the complexity is O(N).

3. int interval_search(interval_root *T, interval_node *i)

   It can be performed on a O(logN) time. (see chap.14 of book <*introduce to algorithm 2nd*>)

   pseudo-code of interval_search:

\begin{lstlisting}

INTERVAL-SEARCH(T, i)

{

x <- root[T];

while x != nil[T] and i does not overlap int[x]

{

if left[x] != nil[T] and max_high[left[x]] >= low[i]

then x <- left[x];

else if (low[x] >= high[i])

return nil[T];

else x <- right[x];

}

return x;

}

\end{lstlisting}

1. interval_node* interval_expand(interval_root *T, interval_node *x)

   It can be performed on a O(logN) time. First of all, the algorithm written here are just for our extent lock situation: when call the function, the interval of @req will not conflict with that in the tree.

- find max(high that less than low(x)) -> new_low, recursion here just for describing the algorithm, no recursion in DLD later.
- find min(low that larger than high(x)) -> new_high

\begin{lstlisting}

/* this algorithm is only for ldlm extent lock situation:

* when call inteval_expand, no locks in the tree can be conflict to current lock

*/

FIND_MAX_HIGH_LESS_THAN_LOW(T, i)

{

x <- root[T];

if x == nil[T]

return 0;

if (low[x] < low[i]) {

if (max_high[x] <= low[i])

return max_high[x];

} else {

left_max = left[x]->max_high; /* left_max mush be less than low[i] */

right_max = FIND_MAX_HIGH_LESS_THAN_LOW(right[x]);

return max(left_max, right_max);

}

} else { /* low[x] > low[i] */

return FIND_MAX_HIGH_LESS_THAN_LOW(left[x]);

}

}

\end{lstlisting}

\begin{lstlisting}

FIND_MIN_LOW_LARGER_THAN_HIGH(T, i)

{

int result = 0;

x <- root[T];

while x != nil[T]

{

if low[x] == high[i] or max_high[x] < high[i]

break;

if low[x] > high[i]

result = low[x];

if(left[x] != nil[T])

x <- left[x];

else

return result;

else if low[x] < high[i]

x <- right[x];

}

return result;

}

\end{lstlisting}

1. int interval_search_all(T, i, void (*func) (interval_node *x))

   It can be performed on a O(min(n, klogN)) time, while k is the number of intervals conflicting with x. recursion here just for describing the algorithm, no recursion in DLD later.

   - y <− root(T)
   - while y != nil
   - if y overlap with i, func(y)
   - if left(y) != nil and max_high(left(y)) > low(i) –> interval_search_all(left(y), i, func);
   - if right(y) != nil and low(y) < high(i) –> interval_search_all(right(y), i, func)

2. interval_node* interval_next(interval_node *node)

3. interval_node* interval_prev(interval_node *node)

4. int interval_insert(interval_root *T, interval_node *x)

5. int interval_remove(interval_root *T, interval_node *x)

   For above methods, it can be performed on a O(logN) time. algorithm to insert and delete one element to or from interval tree is almost the same that red-black tree use (see chap.13 and chap.14 of book <*introduce to algorithm 2nd*>). pseudo-code and real code of red-black tree

insert and delete operation can be found in the book and in the kernel source code. in kernel source code path: <kernel>/lib/rb_tree.c and <kernel>/include/linux/rbtree.h, there are already functions or examples of rb_next, rb_prev, rb_insert_xxx(), rb_delete_xxx().

7 Use Cases

use case here only talk about those involved granted extent locks:

1. ldlm_lock_enqueue()
   - when doing file I/O, client call ldlm_cli_enqueue to enqueue a extent lock request
   - server receive the lock request, call ldlm_handle_enqueu to handle the requst
   - ldlm_lock_enqueue call ldlm_process_extent_lock to do extent lock specified process
     - ldlm_extent_compat_queue to determine if the lock is compatible with all locks on the granted queue and waiting queue
     - if lock can be granted, call ldlm_extent_policy to find the max interval can be locked and add the lock to the granted queue.
   - server delivery reply to client to indicate that: lock is granted or blocked; and client process the lock locally
2. ldlm_grant_lock()
   - on server side, if compat checking and extent internal policy successed, server call ldlm_grant_lock() to add lock into granted queue
   - ldlm_grant_lock call ldlm_resource_add_lock, in turn ldlm_resource_add_lock calls ldlm_resource_add_lock_interval to add lock to related interval tree.
   - ldlm_resource_add_lock_inerval call interval_insert() to accomplish the insert
   - after lock granted in serve side, server delivery replay to client; on client side, client call ldlm_grant_lock to do the same thing to granted local lock.
3. ldlm_lock_cancel()
   - lock cancel are caused by conflict lock request on server side
   - client check if lock can be canceled
   - client call ldlm_cli_cancel to send a lock cancel request

- server received request, server call ldlm_handle_cancel try to cancel lock at server side
- server call ldlm_lock_cancel, ldlm_lock_cancel in turn call ldlm_resource_unlink_lock to unlink lock from granted queue
- ldlm_resource_unlink_lock call ldlm_resource_unlink_lock_interval to remove lock from interval tree

4. ldlm_lock_convert()

   - client call ldlm_cli_convert to ask for lock converting
   - server received request, call ldlm_handle_convert to handle request
   - ldlm_handle_convert call ldlm_lock_convert to convert lock
   - ldlm_lock_convert first remove lock from granted queue, convert req mode to new mode,
   - then ldlm_lock_convert try to call ldlm_process_extent_lock to grant the lock .

5. filter_intent_policy()

   - client call ll_glimpse_size()
   - client call ldlm_cli_enqueue to enqueue a filter intent lock
   - server received request, server call ldlm_handle_enqueue to handle requst
   - ldlm_handle_enqueue call ldlm_lock_enqueue
   - ldlm_lock_enqueue call filter_intent_policy find the PW lock that extent start larger than lvb
   - filter_intent_policy call ldlm_server_glimpse_ast to get KMS from client
   - client call ll_glimpse_callback to send KMS to ost

6. ldlm_handle_cp_callback()

   * server granted lock on server side, delivery replay to client

   - client call ldlm_handle_cp_callback() do complete callback at local side
   - ldlm_handle_cp_callback call ldlm_grant_lock to grant lock locally

7. ldlm_lock_match()

   - before enqueue extent lock to server, client first call ldlm_lock_match to see if such lock is already existed local

- ldlm_lock_match call search_queue to search referenced lock in three lock queues
- search_queue walk through lock queue to find matched lock, for granted queue, search_queue call search_queue_interval() to find matched lock in interval tree.

8. ll_extent_lock_callback()

   - on client side, when client call ldlm_lock_cancel to cancel local lock
   - ldlm_lock_cancel call ll_extent_lock_callback
   - ll_extent_lock_callback call ldlm_extent_shift_kms to update KMS
   - ldlm_extent_shift_kms directly compare interval tree's max_high with old_kms, and return the lower one

9. ldlm_resource_foreach()

   - some functions call ldlm_resource_iterate to do some operation on each lock of resource
   - ldlm_resource_iterate call ldlm_resource_foreach
   - ldlm_resource_foreach traverse whole interval tree, and do callback on each lock

10. ldlm_replay_locks()

    - after client failure, ptlrpc_import_recovery_state_machine call ldlm_replay_locks to replay locks
    - ldlm_replay_locks call ldlm_namespace_foreach
    - ldlm_namespace_foreach call ldlm_resource_foreach and call ldlm_chain_lock_for_replay on each lock

11. ldlm_resource_dump()

    - in ldlm_resource_add_lock, ldlm_resource_dump is first called
    - ldlm_resource_dump traverse whole tree, call ldlm_lock_dump on each lock

12. cleanup_resource()

    - filter_cleanup call ldlm_namespace_free
    - ldlm_namespace_free call ldlm_namespace_cleanup
    - ldlm_namespace_cleanup call cleanup_resource on each resource
    - cleanup_resource traverse whole tree, cleanup extent lock on granted queue

13. ldlm_cli_cancel_unused_resource()

    - osc_cancel_unused call ldlm_cli_cancel_unused
    - ldlm_cli_cancel_unsued call ldlm_cli_cancel_unused_resource
    - ldlm_cli_cancel_unused_resource traverse whole tree, cancel each exent lock if it is not in used

14. ldlm_cli_join_lru()

    - osc_join_lru call ldlm_cli_join_lru
    - ldlm_cli_join_lru traverse whole tree, for each extent lock in granted queue add it to ns->ns_unused_list if it is unused.

8 performance

to compare list method and interval tree method: the list method is very simple and easy to implement; but contrast to current list approach, using interval tree to store granted extent lock can be much faster especially for large scale situations. performance compare between these two method include:

1. In the process of handling extent lock enqueue, previous list method need both O(N) cost during compat queue (see ldlm_extent_compat_queue()) and extent policy (ldlm_extent_internal_policy). but by using interval tree method, this process can only cose O(logN), if N is very huge, the performance improvement is prominent. i.e., if there are 1M locks current in the granted queue, list method need 1M times loop to compat or policy lock, while interval tree method need only 20 times.

2. for function filter_intent_policy() and ldlm_extent_shift_kms() (see section 5 and section 6), cost of previous method is also O(N), but with field max_high in interval tree node, these two functions can be O(1). for function search_queue, cost of interval tree is O(logN), list method is O(N).

3. to find all conflicting locks in granted queue (in ldlm_extent_compat_queue() with work_list not NULL), using interval tree will be O(min(klogN, N)), so if k is small, interval tree method is still between than list method, but if k is large performance for interval tree is about the same to that of list method.

4. there are also some function can not optimazed, such as ldlm_resource_for_each(), cleanup_resoure(), for these functions walk through the list (for list method) or traversing whole tree is obligatory. so we need a interface to traversing trees, though it is also O(N), but because of complexer struct than list, performance of traversing tree may be a little slower than walking throuh a list.

9 Locking.

All the lock list operations are performed under lr_lock held.

10 Recovery.

No recovery implications are involved.

11 API/Protocols.

No changes in api/protocols are involved.

12 Test plan.

For interval tree, tests inclulde:

- unit tests for the interval tree functions.
- time test functions, unit test demonstrate enqueue with 100,000 compatible intervals works quickly; get runtime for such enqueue with and without interval tree.
- use liblustre (with a different main function) as a user level API to test the functionality.

13 Alternatives

1. vitaly suggested to continue to use l_res_link for locks having the head in resource.lr_granted.

   since lock/resource management is relative low-level and independent module, correctly defined interval tree operations can completely subsitute list operations, and use l_res_link to link tree roots can still keep most interface unchanged.

2. vitaly suggested to drop @low, @high from interval_node because they are the same as lock->l_policy_data.l_extent.(start, end). and he also think interval node is added to every lock and it waste a lot of memory.

   because interval tree code are to be independent library code, so leave @low, @high in struct interval_node is prefer. for the later problem, vitaly suggests to allocate node only when needed and have: struct interval_node

*l_interval; and probably have the back pointer in the struct node, if needed: struct ldlm_lock *lock; This will be carefully thought of in next DLD phase.