# DLD of inodebits lock performance improvement

Bobi Jam

2006-12-19

## 1 Requirements

### 1.1 Use-Case requirements

Parallel adding, removing, opening, stating files from one directory are much faster.

### 1.2 Functional requirements

Avoid O(N^2) lock operations.

## 2 Functional Specification

### 2.1 Introduction

This DLD introduces a performance improvement in Lustre DLM plain and inodebits lock management. Because of modern cluster's size, linear list of locks becomes a bottleneck with millions of clients. This improvement is for handle of large scale usage.

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 on every resource. This improvement is to optimize the lock compatibility checking and searching conflicting locks on the granted list.

Skip lists are added into the structure *ldlm_lock* to optimize the granted lock list handling: mode, policy skip lists.

```
\begin{lstlisting}
struct ldlm_lock {
...
struct list_head l_sl_mode;
struct list_head l_sl_policy;
};
\end{lstlisting}
```

Refer to the figure above, when lock is not granted, its skip list pointers *l_sl_mode* and *l_sl_policy* are pointing to NULL, as lock *0* in the figure. When a lock is granted and inserted into a resource's granted list (*lr_granted*), some lock's skip list pointers will be changed to maintain some invariance, wherein this document, these invariance is called group integrity, which contains:

1. If a resource's granted list is empty, the resource's granted list_head (*lr_granted*) points to itself.

2. If a resource's granted list has lock(s) been granted, and if we start from resource's granted list_head (*lr_granted*), walking through all locks on the list by lock's *l_res_link.next* field, before we encounter the resource's granted list_head, and if lock m is encountered before lock n, then we say lock m is ahead of lock n. And there is only one lock in the list whose *l_res_link.next* points to the resource's granted list_head, this lock is called the tail of the granted list, as lock *n* in the figure .

3. Locks with the same request mode (*lock->l_req_mode*) are grouped, i.e. during the walk-through, before hitting the resource's granted list_head again, if a lock whose request mode is firstly encountered, all locks with the same request mode in the granted list will be met one by one after the firstly met lock, until a lock with different request mode or the resource's list_head is met. The lock of the specific request mode firstly met is called the head of the mode group (as lock *1* in the figure), the last met lock of the same request mode is called the tail of the mode group (as lock *5* in the figure). Those locks of the specific request mode which is neither the head nor the tail of the mode group are called in the middle of the mode group (as lock *2,3,4* in the figure). Some mode group may contain only one lock, and we call this kind of mode group as single element mode group (as lock *m* in the figure).

4. If the resource is a LDLM_IBITS resource, for all locks of each mode group, locks with the same bits policy *(lock->l_policy_data.l_inodebits.bits)* are grouped in the similar way as mode group does. I.e. during the walk-through in the mode group, before hitting a lock of different request mode or the resource's granted list_head, if a lock whose bits policy is firstly encountered, all locks with the same bits policy in the mode group will be met one by one after the firstly met lock, until a lock with different bits policy or different request mode is met. The lock of the specific bits policy firstly met is called the head of the policy group (as lock *2* in the figure), the last met lock of the same bits policy is called the tail of the policy group (as lock *3* in the figure). Those locks of the specific bits policy which is neither the head nor the tail of the policy group are called in the middle of the policy group (not showed in the figure). Some policy group may contain only one lock, and we call this kind of policy group as single element policy group (as lock *1,4,5* in the figure).

5. If the resource is a LDLM_PLAIN resource, all lock's *l_sl_policy* point to NULL. All lock's *l_sl_mode* point to NULL except for head lock's *l_sl_mode.next* and tail lock's *l_sl_mode.prev* of mode group which contain more than 1 member locks. The head points to the tail lock using head lock's *l_sl_mode.next* and the tail points to the head lock using tail lock's *l_sl_mode.prev*.

6. If the resource is a LDLM_IBITS resource, lock's *l_sl_mode* pointers will be arranged as describe in invariance item 5, and all lock's *l_sl_policy* point to NULL except for those head lock's *l_sl_policy.next* and tail lock's *l_sl_policy.prev* of policy group which contain at least 2 member locks. The head and tail locks of the same policy group in the same mode group will link together with their *l_sl_policy* pointers (using *l_sl_policy.next* for head pointing to tail and *l_sl_policy.prev* for tail pointing to head).

## 2.2 lock's insertion into/deletion from a resource's granted list

The skip list pointers in the *ldlm_lock* structure only make sense when the lock is granted, i.e. added into resource's granted list.

When a lock is created, it's skip list heads are initialized pointing to NULL. When a lock is granted, first need to determine where it should be inserted and then inserted into the resource's granted list. The position determination described as follows:

- If the resource is a LDLM_PLAIN resource, (1) the insert position will be before the head of the mode group whose request mode is the same as that of the lock to be inserted. (2) If such mode group can not be found, the lock would be appended after the tail of the granted list, becoming a single element mode group lock.

- If the resource is a LDLM_IBITS resource, first try to find the mode group whose request mode is the same as that of the lock to be inserted. (1) If such mode group can not be found, the lock will be appended after the tail of the granted list, becoming a single element mode/policy group; (2) if such mode group is found, keep searching in the mode group to find a policy group whose inodebits are the same as that of the lock to be inserted, if such policy group can not be found, the lock will be inserted before the head lock of the mode group, becoming a single element policy group and the head of the mode group; (3) if such policy group is found, the lock will be inserted before the head lock of the policy group.

As the lock inserted in the granted list, the group integrity maybe compromised, so we need adjust some lock's skip list pointers to maintain the group integrity.

- For LDLM_PLAIN resource. (1) If the insert position is before the head of a mode group which has at least 2 locks, the new lock will replace the

original mode group head lock, becomes the new mode group head lock, the new mode group head lock and the mode group tail lock will link together with their *l_sl_mode* pointers, the original mode group head lock's *l_sl_mode.next* will set to NULL. (2) If the insert position is before a single element mode group, the new lock and the single element mode group lock will link together with their *l_sl_mode* pointers, forming a mode group. (3) If the insert position is after the tail of the granted list, the new lock's skip lists need not change, keeping pointing to NULL as a single element mode group lock.

- For LDLM_IBITS resource. (1) If the insert position is after the tail of the granted list, the new lock's skip lists need not change, keeping pointing to NULL as a single element mode/policy group lock. (2) If the insert position is before the head lock of a mode group (two cases is here, one is the new lock joins a policy group and also becomes the head of the mode group; another is the new lock belongs to the mode group while no same policy group is already there). (a) In case 1, the new lock will take over the original mode group head lock's role (if it was not a single lock mode group), or link with the original lock forming a mode group (in the case the original lock is a single lock group); similar actions applies to policy skip list. (b) In case 2, mode skip list adjustment will abide by the description in (a), nothing about policy skip list needs to change. (3) If the insert position is before the head lock of a policy group which has at least 2 locks while not the head of mode group , the new lock will take over the original policy group head lock's role as the new policy group head lock . And if the insert position is before a single element policy group plus the new lock has the same policy bits, they will form a policy group.

When a lock is removed from the granted list, the skip list pointers of removed lock need to be restored pointing to NULL again, and the locks in the granted list affected needs adjust their skip list pointers to maintain the group integrity. Specifically:

- For LDLM_PLAIN resource. (1) If the lock to be removed was the head of a mode group, the next lock along the *l_res_link* list will be the mode group's new head lock, it and the mode group tail lock will link together with their *l_sl_mode* pointers; If the next lock happens to be the mode group tail lock, just initialize its *l_sl_mode* pointers to NULL; (2) If the lock to be removed was the tail of a mode group, the previous lock along the *l_res_link* list will be the mode group's new tail lock, it and the mode group head lock will link together with their *l_sl_mode* pointers; If the previous lock happens to be the mode group head lock, just initialize its *l_sl_mode* pointers to NULL; (3) Otherwise other lock's mode skip list pointers need not change.

- For LDLM_IBITS resource. Beside fix *l_sl_mode* pointers as described above, some lock's *l_sl_policy* need to change as follows: (1) If the lock

to be removed was the head of a policy group, the next lock along the *l_res_link* list will be the policy group's new head lock, it and the policy group tail lock will link together with their *l_sl_policy* pointers; If the next lock happens to be the policy group tail lock, just initialize its *l_sl_policy* pointers to NULL; (2) If the lock to be removed was the tail of a policy group, the previous lock along the *l_res_link* list will be the policy group's new tail lock, it and the policy group head lock will link together with their *l_sl_policy* pointers; If the previous lock happens to be the policy group head lock, just initialize its *l_sl_policy* pointers to NULL; (3) Otherwise other lock's policy skip list pointers need not change.

## 2.3 Functions to be added and modified

### 2.3.1 plain lock compatibility test

**Prototype:**

```
\lstinline|static inline int ldlm_plain_compat_queue(struct list_head *queue, struct ld
```

**Parameters:**

**queue [input]:** the queue to be searched, the improvement applies only on granted list;

**req [input]:** the lock whose mode compatibility is to be searched;

**work_list [input,output]:** the list gathering conflicting locks

**Return Values:** 1 if no conflict found which includes the @*req* is already in the list *@queue*, 0 otherwise.

**Description:**

This method searches conflicts for the plain lock *@req* in the *@queue* list of plain locks (using *lockmode_compat()* to test whether *@req*'s request mode is compatible with the lock's request mode being checked). If *@work_list* is provided, all the conflicting locks are gathered into this list, otherwise the *@queue* is walked through until the first conflicting lock is found.

### 2.3.2 inodebits lock compatibility test

**Prototype:**

```
\lstinline|static int ldlm_inodebits_compat_queue(struct list_head *queue, struct ldlm_
```

**Parameters:**

**queue [input]:** the queue to be searched, the improvement applies only on granted list;

**req [input]:** the lock whose mode and inodebits compatibility are to be searched;

**work_list [input,output]:** the list gathering conflicting locks

**Return Values:** 1 if no conflict found which includes the @*req* is already in the list *@queue*, 0 otherwise.

**Description:**

This method searches conflicts for the inodebits lock *@req* in the *@queue* list of inodebits locks (using *lockmode_compat()* to check mode compatibility and testing whether *@req*'s request bits are overlapped to lock's), if *@req*'s request mode is not compatible with lock's request mode while their bits are overlapped, then the lock is in conflict with *@req*. If *@work_list* is provided, all the conflicting locks are gathered into this list, otherwise the *@queue* is walked through until the first conflicting lock is found.

### 2.3.3 grant a lock

**Prototype:**

```
\lstinline|void ldlm_grant_lock(struct ldlm_lock *lock, struct list_head *work_list);|
```

**Parameters:**

**lock [input]:** the lock to be granted;

**work_list [input,output]:** the list gathering the lock

**Return Values:** none

**Description:**

For plain and inodebits locks call *ldlm_grant_lock_with_skiplist()* to grant the lock, otherwise call *ldlm_resource_add_lock()*.

### 2.3.4 grant a lock with skip list

**Prototype:**

```
\lstinline|static void ldlm_grant_lock_with_skiplist(struct ldlm_lock *lock);|
```

**Parameters:**

**lock [input]:** the lock to be granted;

**Return Values:** none

**Description:**

This method finds the proper position the lock should be inserted, then inserts it and adjusts relevant lock's skip list pointers.

### 2.3.5 search granted lock position

**Prototype:**

```
\lstinline|static int search_granted_lock(struct list_head *queue, struct ldlm_lock *re
```

**Parameters:**

**queue [input]:** the grant list where search acts on;

**req [input]:** the lock whose position to be located;

**lockp [output]:** the position where lock should be inserted before, or NULL indicating *@req* should be appended to *@queue*.

**Return Values:**

Bit-masks combination of following values indicating in which way the lock need to be inserted.

- LDLM_JOIN_NONE - nothing about skip list needs to be fixed;
- LDLM_MODE_JOIN_RIGHT - *@req* needs join right becoming the head of a mode group;
- LDLM_POLICY_JOIN_RIGHT - *@req* needs join right becoming the head of a policy group.

**Description:**

This method finds a position for insertion. Match is defined as the same lock mode and the same policy only for inodebits locks.

If the *@queue* is a LDLM_PLAIN resource's granted list, this method will search to find the head lock of the group whose request mode is the same as that of *@req*, and assign that lock to *@lockp*, returns LDLM_MODE_JOIN_RIGHT. If no such mode group could be found, *@lockp* returns NULL, meaning *@req* should be appended to the tail of the granted list, returns LDLM_JOIN_NONE.

If the *@queue* is a LDLM_IBITS resource's granted list, this method will firstly find the head lock of the mode group whose request mode is the same as that of *@req*, and further search all locks in the mode group to find the head lock of the policy group whose inodebits is the same as that of *@req*, and assign that lock to *@lockp*, returns LDLM_POLICY_JOIN_RIGHT; if the lock is also the head of a mode group or a single mode group lock, returns LDLM_POLICY_JOIN_RIGHT | LDLM_MODE_JOIN_RIGHT. If no such mode group can be found *@lockp* returns NULL and function returns LDLM_JOIN_NONE; If find such mode group while does not find such policy group, the head of the mode group will be assigned to *@lockp*, function returns LDLM_MODE_JOIN_RIGHT.

# 3 Use Cases

## 3.1 Enqueue a lock

When a lock is enqueued, *ldlm_lock_enqueue()* calls *ldlm_processing_policy_table[]* methods to process the lock.

## 3.2 Reprocess a queue

Reprocess locks on the converting and/or waiting list, it calls *ldlm_processing_policy_table[]* methods to check if some waiting locks are to be granted.

## 3.3 Grant a lock

Grants a lock. There are no conflicting locks by that time.

## 3.4 Handle completion callback

It calls *ldlm_grant_lock* to grant the lock on the client when the server confirms the lock is obtained.

## 3.5 Cancel a lock

It cancels a granted lock.

# 4 Logic Specification

## 4.1 Determining a lock's position

According the HLD, skip lists are added to *struct ldlm_lock* to optimize the granted lock list compatibility checking and searching conflicting locks.

The *l_sl_{mode|policy}* is used to link the head and the tail lock of a group of the same request mode/inodebits. And locks in the middle of the group have their *l_sl_{mode|policy}* point to NULL.

So there should be a way to judge whether a lock in the group is the head or tail of the group.

```
\begin{lstlisting}
#define LDLM_SL_HEAD(skip_list) ((skip_list)->next != NULL)
#define LDLM_SL_TAIL(skip_list) ((skip_list)->prev != NULL)
#define LDLM_SL_EMPTY(skip_list) ((skip_list)->next == NULL && (skip_list)->prev == NUL
\end{lstlisting}
```

## 4.2 Find lock compatibility from a list

\lstinline|int ldlm_plain_compat_queue(struct list_head *queue, struct ldlm_lock *req,

- if the *@queue* is a granted list, check the head of the first mode group, if it's mode is compatible with the *@req*, jump over to the tail of the mode group (via *l_sl_mode.next*) and check next group.
- if the *@work_list* exists, we need collect all incompatible locks on the *@queue*. And when the head lock of a group is found to be incompatible with the *@req*, all members in this mode group will be collected.

```
\begin{lstlisting}
static inline int ldlm_plain_compat_queue(struct
list_head *queue, struct ldlm_lock *req, struct
list_head *work_list)
{
list_head *tmp;
struct ldlm_lock *lock;
int compat = 1;
...
list_for_each(tmp, queue){
lock = list_entry(tmp, struct ldlm_lock, l_res_link);
if (lockmode_compat(lock->l_req_mode, req->l_req_mode))
{
/* jump to next mode group */
if (LDLM_SL_HEAD(&lock->l_sl_mode))
tmp = &list_entry(lock->l_sl_mode.next, struct
ldlm_lock, l_sl_mode)->l_res_link;
continue;
}
if (!work_list)
RETURN(0);
compat = 0;
if (lock->l_blocking_ast)
ldlm_add_ast_work_item(lock, req, work_list);
if (LDLM_SL_HEAD(&lock->l_sl_mode)) {
/* add all members of the mode group */
do {
tmp = lock->l_res_link.next;
lock = list_entry(tmp, struct ldlm_lock, l_res_link);
if (lock->l_blocking_ast)
ldlm_add_ast_work_item(lock, req, work_list);
} while (!LDLM_SL_TAIL(&lock->l_sl_mode));
```

```
}
}
RETURN(compat);
}
\end{lstlisting}
\lstinline|int ldlm_inodebits_compat_queue(struct list_head *queue, struct ldlm_lock *r
```

- if the *@queue* is a granted list, check the head of the first mode group, if it's mode is compatible with the *@req*, jump over to the tail of the mode group (via *l_sl_mode.next*) and check next mode group.
- if a incompatible mode group is found, then check policy group in the same way as processing the mode group.
- if the *@work_list* exists, we need collect all incompatible locks on the *@queue*. And when the head lock of a policy group is found to be incompatible with the *@req*, all members in the policy group will be collected.

```
\begin{lstlisting}
static int ldlm_inodebits_compat_queue(struct list_head
*queue, struct ldlm_lock *req, struct list_head
*work_list)
{
struct list_head *tmp, *tmp_tail;
struct ldlm_lock *lock;
int compat = 1;
ldlm_mode_t req_mode = req->l_req_mode;
_u64 req_bits = req->l_policy_data.l_inodebits.bits;
...
list_for_each(tmp, queue) {
lock = list_entry(tmp, struct ldlm_lock, l_res_link);
if (lockmode_compat(lock->l_req_mode, req->l_req_mode))
{
/* jump to next mode group */
if (LDLM_SL_HEAD(&lock->l_sl_mode))
tmp = &list_entry(lock->l_sl_mode.next, struct
ldlm_lock, l_sl_mode)->l_res_link;
continue;
}
tmp_tail = tmp;
if (LDLM_SL_HEAD(&lock->l_sl_mode))
tmp_tail = &list_entry(lock->l_sl_mode.next, struct
ldlm_lock, l_sl_mode)->l_res_link;
for (;;) {
```

```
/* locks whose bits overlapped are conflicting locks */
if (lock->l_policy_data.l_inodebits.bits & req_bits) {
/* found conflicting policy */
if (!work_list)
RETURN(0);
compat = 0;
if (lock->l_blocking_ast)
ldlm_add_ast_work_item(lock, req, work_list);
/* add all members of the policy group */
if (LDLM_SL_HEAD(&lock->l_sl_policy)) {
do {
tmp = lock->l_res_link.next;
lock = list_entry(tmp, struct ldlm_lock, l_res_link);
if (lock->l_blocking_ast)
ldlm_add_ast_work_item(lock, req, work_list);
} while(!LDLM_SL_TAIL(&lock->l_sl_policy));
}
} else {
/* jump to next policy group */
if (LDLM_SL_HEAD(&lock->l_sl_policy))
tmp = &list_entry(lock->l_sl_policy.next, struct
ldlm_lock, l_sl_policy)->l_res_link;
}
if (tmp == tmp_tail)
break;
else
tmp = tmp->next;
lock = list_entry(tmp, struct ldlm_lock, l_res_link);
} // for locks in a mode group
} // for each lock in the granted queue
RETURN(compat);
}
\end{lstlisting}
```

## 4.3 Search position a new granted lock should be inserted

```
\lstinline|static int search_granted_lock(struct list_head *queue, struct ldlm_lock *re
```

- Only defines plain lock and inodebits lock.
  - plain lock: finds the head lock of the *@req* mode group. If the to-be-checked lock's request mode differs from that of *@req*, jumps over all locks in the mode group until the *@queue* is met, meaning no such request mode locks are found in the list, assign NULL to *@lockp* indicating *@req*

should be appended to the queue; if the head lock of such mode group is found, assign the head lock to *@lockp*.

- inodebits lock: first search for the same request mode group the way as describe above in plain lock section. If no such same request mode group is found, assign NULL to *@lockp*; if a same mode group is found, search through the mode group by inodebits. If no same inodebits is found, assign the head lock of the mode group to *@lockp*, otherwise assign the head lock of the found policy group to *@lockp*.

```
\begin{lstlisting}
#define LDLM_JOIN_NONE 0
#define LDLM_MODE_JOIN_RIGHT 1
#define LDLM_MODE_JOIN_LEFT (1 << 1)
#define LDLM_POLICY_JOIN_RIGHT (1 << 2)
#define LDLM_POLICY_JOIN_LEFT (1 << 3)
int search_granted_lock(struct list_head *queue,
struct ldlm_lock *req, struct ldlm_lock **lockp)
{
struct list_head *tmp, *tmp_tail;
struct ldlm_lock *lock, *mode_head_lock;
__u64 req_bits =
req->l_policy_data.l_inodebits.bits;
int rc = LDLM_JOIN_NONE;
...
list_for_each(tmp, queue) {
lock = list_entry(tmp, struct ldlm_lock,
l_res_link);
if (lock->l_req_mode != req->l_req_mode) {
if (LDLM_SL_HEAD(&lock->l_sl_mode))
tmp = &list_entry(lock->l_sl_mode.next, struct
ldlm_lock, l_sl_mode)->l_res_link;
continue;
}
/* found the same mode group */
if (lock->l_resource->lr_type == LDLM_PLAIN) {
*lockp = lock;
return LDLM_MODE_JOIN_RIGHT;
}
```

```
if (lock->l_resource->lr_type == LDLM_IBITS) {
tmp_tail = tmp;
if (LDLM_SL_HEAD(&lock->l_sl_mode))
tmp_tail = &list_entry(lock->l_sl_mode.next, struct
ldlm_lock, l_sl_mode)->l_res_link;
mode_head_lock = lock;
for (;;) {
if (lock->l_policy_data.l_inodebits.bits ==
req_bits) {
/* lock of matched policy is found */
*lockp = lock;
rc |= LDLM_POLICY_JOIN_RIGHT;
/* the policy group head is also a mode group head
or a single mode group lock */
if (LDLM_SL_HEAD(&lock->l_sl_mode) || (tmp ==
tmp_tail && LDLM_SL_EMPTY(&lock->l_sl_mode)))
rc |= LDLM_MODE_JOIN_RIGHT;
return rc;
}
if (LDLM_SL_HEAD(&lock->l_sl_policy))
tmp = &list_entry(lock->l_sl_policy.next, struct
ldlm_lock, l_sl_policy)->l_res_link;
if (tmp == tmp_tail) /* reached the end of the mode
group */
break;
else /* next policy group */
tmp = tmp->next;
lock = list_entry(tmp, struct ldlm_lock,
l_res_link);
} /* for all locks in the matched mode group */
/* no matched policy group is found, insert before
the mode group head lock */
*lockp = mode_head_lock;
return LDLM_MODE_JOIN_RIGHT;
} // inodebits lock
} // for locks in queue
*lockp = NULL;
return LDLM_JOIN_NONE;
}
\end{lstlisting}
```

## 4.4 Grant a lock

\lstinline|void ldlm_grant_lock(struct ldlm_lock *lock, struct list_head *work_list)|

- If the resource type of the lock is plain lock or inodebits lock, call *ldlm_grant_lock_with_skiplist()* to grant the lock;
- Otherwise just add the lock to the resource.

```
\begin{lstlisting}
void ldlm_grant_lock(struct ldlm_lock *lock, struct
list_head *work_list)
{
struct ldlm_resource *res = lock->l_resource;
...
lock->l_granted_mode = lock->l_req_mode;
if (res->lr_type == LDLM_PLAIN || res->lr_type ==
LDLM_IBITS)
ldlm_grant_lock_with_skiplist(lock);
else
ldlm_resource_add_lock(res, &res->lr_granted, lock);
...
}
\end{lstlisting}
```

\lstinline|void ldlm_grant_lock_with_skiplist(struct ldlm_lock *lock)|

- call *search_granted_lock()* to find the position (assigned to *@lockp)* the lock should be inserted, *search_granted_lock()* also returns the way in which skip lists needs to change;
- insert the *@lock* before *@lockp* if *@lockp* is not NULL, otherwise *@lock* is appended to the tail of the granted list;
- adjust skip lists according to what *search_granted_lock()* returns.

```
\begin{lstlisting}
static void ldlm_grant_lock_with_skiplist(struct
ldlm_lock *lock)
{
int join = LDLM_JOIN_NONE;
struct ldlm_lock *lockp = NULL;
...
join =
search_granted_lock(&lock->l_resource->lr_granted, lock,
&lockp);
if (!lockp)
```

```
list_add_tail(&lock->l_res_link,
&lock->l_resource->lr_granted);
else
list_add_tail(&lock->l_res_link, &lockp->l_res_link);
/* fix skip lists */
if (join & LDLM_MODE_JOIN_RIGHT) {
if (LDLM_SL_EMPTY(&lockp->l_sl_mode)) {
lock->l_sl_mode.next = &lockp->l_sl_mode;
lockp->l_sl_mode.prev = &lock->l_sl_mode;
} else if (LDLM_SL_HEAD(&lockp->l_sl_mode)) {
lock->l_sl_mode.next = lockp->l_sl_mode.next;
lockp->l_sl_mode.next = NULL;
lock->l_sl_mode.next->prev = &lock->l_sl_mode;
}
}
if (join & LDLM_POLICY_JOIN_RIGHT) {
if (LDLM_SL_EMPTY(&lockp->l_sl_policy)) {
lock->l_sl_policy.next = &lockp->l_sl_policy;
lockp->l_sl_policy.prev = &lock->l_sl_policy;
} else if (LDLM_SL_HEAD(&lockp->l_sl_policy)) {
lock->l_sl_policy.next = lockp->l_sl_policy.next;
lockp->l_sl_policy.next = NULL;
lock->l_sl_policy.next->prev = &lock->l_sl_policy;
}
}
...
}
\end{lstlisting}
```

## 4.5 Cancel a lock

\lstinline|void ldlm_lock_cancel(struct ldlm_lock *req)|

- If to-be-canceled lock is the head of a mode group, set *@req->l_sl_mode* to the *@req->l_res_link.next* lock, if it happens to point to itself, NULL it; adjust *@req->l_sl_mode.next*'s lock's *l_sl_mode.prev* also.
- If to-be-canceled lock is the tail of a mode group, set *@req->l_sl_mode* to the *@req->l_sl_mode.prev* lock, if it happens to point to itself, NULL it; adjust *@req->l_sl_mode.prev*'s lock's *l_sl_mode.next* also.
- The similar thing happens with *l_sl_policy*.
- Remove *@req* from the granted list.

```
\begin{lstlisting}
void ldlm_lock_cancel(struct ldlm_lock *req)
{
struct ldlm_lock *lock;
...
if (LDLM_SL_HEAD(&req->l_sl_mode)) {
lock = list_entry(req->l_res_link.next, struct
ldlm_lock, l_res_link);
if (req->l_sl_mode.next == &lock->l_sl_mode) {
lock->l_sl_mode.prev = NULL;
} else {
lock->l_sl_mode.next = req->l_sl_mode.next;
lock->l_sl_mode.next->prev = &lock->l_sl_mode;
}
req->l_sl_mode.next = NULL;
} else if (LDLM_SL_TAIL(&req->l_sl_mode)) {
lock = list_entry(req->l_res_link.prev, struct
ldlm_lock, l_res_link);
if (req->l_sl_mode.prev == &lock->l_sl_mode) {
lock->l_sl_mode.next = NULL;
} else {
lock->l_sl_mode.prev = req->l_sl_mode.prev;
lock->l_sl_mode.prev->next = &lock->l_sl_mode;
}
req->l_sl_mode.prev = NULL;
}
if (LDLM_SL_HEAD(&req->l_sl_policy)) {
lock = list_entry(req->l_res_link.next, struct
ldlm_lock, l_res_link);
if (req->l_sl_policy.next == &lock->l_sl_policy) {
lock->l_sl_policy.prev = NULL;
} else {
lock->l_sl_policy.next = req->l_sl_policy.next;
lock->l_sl_policy.next->prev = &lock->l_sl_policy;
}
req->l_sl_policy.next = NULL;
} else if (LDLM_SL_TAIL(&req->l_sl_policy)) {
lock = list_entry(req->l_res_link.prev, struct
ldlm_lock, l_res_link);
if (req->l_sl_policy.prev == &lock->l_sl_policy) {
lock->l_sl_policy.next = NULL;
} else {
lock->l_sl_policy.prev = req->l_sl_policy.prev;
lock->l_sl_policy.prev->next = &lock->l_sl_policy;
}
req->l_sl_policy.prev = NULL;
```

```
}
ldlm_resource_unlink_lock(lock);
...
}
\end{lstlisting}
```

# 5 State Specification

## 5.1 Locking

All the lock list operations are performed under lr_lock held.

## 5.2 Recovery

No recovery implications are involved.