Programming Locking Applications

Patrick Caulfield
Red Hat Inc

pcaulfie@redhat.com

Based on:

Programming Locking Applications

by Kristin Thomas
kristint@us.ibm.com
Published August 2001

Revision History

0.1	5th October 2007	Cosmetic fixes & tidying
0.2	31st October 2007	Did the table of contents, added a bit on devices
0.3	7th December 2007	Fixed typo /proc/misc/* should be /dev/misc/*

Table of Contents

Chapter 1. Distributed Lock Manager

This chapter introduces the Distributed Lock Manager.

1.1. An Overview of the Distributed Lock Manager

The Distributed Lock Manager (DLM) provides advisory locking services that allow concurrent applications running on multiple nodes in a Linux cluster to coordinate their use of shared resources. Cooperating applications running on different nodes in a Linux cluster can share common resources without corrupting those resources. The shared resources are not corrupted because the lock manager synchronizes (and if necessary, serializes) access to them.

> **Note:** All locks are advisory, that is, voluntary. The system does not enforce locking. Instead, applications running on the cluster must cooperate for locking to work. An application that wants to use a shared resource is responsible for first obtaining a lock on that resource before attempting to access it.

Applications that can benefit from using the Distributed Lock Manager are transaction-oriented, such as a database or a resource controller or manager.

1.2. DLM Locking Model

The DLM locking model provides a rich set of locking modes and both synchronous and asynchronous execution.

The DLM locking model supports:

- Six locking modes that increasingly restrict access to a resource
- The promotion and demotion of locks through conversion
- Synchronous completion of lock requests
- Asynchronous completion through asynchronous system trap (AST) emulation
- Global data through lock value blocks

For more information about the DLM locking model, see Chapter 2.

1.3. Application Programming Interface, introduction

The Distributed Lock Manager supports an application programming interface (API), a collection of C language routines, that allow you to acquire, manipulate, and release locks. This API presents a high-level interface that you can use to implement locking in an application. The API routines that implement the DLM locking model are described in Chapter 7.

Red Hat Linux includes two versions of the lock manager API libraries. They are:

- libdlm.so: for user-space DLM client applications that use pthreads
- libdlm_lt.so: for user-space DLM client applications that do not use pthreads

This document will only discuss the user-space use of the DLM using these libraries.

1.4. Distributed Lock Manager Architecture

The lock manager defines a lock resource as the lockable entity. The lock manager creates a lock resource the first time an application requests a lock against it. A single lock resource can have one or many locks associated with it. A lock is always associated with one lock resource.

The lock manager provides a single, unified lock image shared among all nodes in the cluster. Each node runs a copy of the lock manager kernel daemons. These lock manager daemons communicate with each other to maintain a cluster-wide database of lock resources and the locks held on these lock resources.

Within this cluster-wide database, the lock manager maintains one master copy of each lock resource. This master copy can reside on any cluster node. Initially, the master copy resides on the node on which the lock request originated. The lock manager maintains a cluster-wide directory of the locations of the master copy of all the lock resources within the cluster. The lock manager attempts to evenly divide the contents of this directory across all cluster nodes. When an application requests a lock on a lock resource, the lock manager first determines which node holds the directory entry and then, reads the directory entry to find out which node holds the master copy of the lock resource.

By allowing all nodes to maintain the master copy of lock resources, instead of having one primary lock manager in a cluster, the lock manager can reduce network traffic in cases when the lock request can be handled on the local node. Handling the requests on the local node also avoids the potential bottleneck resulting from having one primary lock manager and reduces the time required to reconstruct the lock database when a failover occurs. Using these techniques, the lock manager attempts to increase lock throughput and reduce the network traffic overhead.

When a node fails, the lock managers running on the surviving cluster nodes release the locks held by the failed node. The lock manager then processes lock requests from surviving nodes that were previously blocked by locks owned by the failed node. In addition, the other nodes re-master locks that were mastered on the failed node.

1.4.1. The DLM and Different Cluster Infrastructures

The DLM provides its own mechanisms to support its locking features, such as inter-node communication to manage lock traffic and recovery protocols to re-master locks after a node failure or to migrate locks when a node joins the cluster. However, the DLM does not provide mechanisms to actually manage the cluster itself.

Therefore the DLM expects to operate in a cluster in conjunction with another cluster infrastructure environment that provides the following minimum requirements:

- Node liveness: The node is a node part of a cluster.
- Consistent view of membership: All nodes agree on cluster membership.
- IP address liveness: An IP address to use to communicate with the DLM on a node; Normally the DLM use TCP/IP for inter-node communications which restricts it to a single IP address per node (though this can be made more redundant using the bonding driver). The DLM can be configured to use SCTP as it's inter-node transport which allows multiple IP addresses per node.

The DLM works with any cluster infrastructure environments that provide the minimum requirements listed above. The choice of an open source or closed source environment is up to the user. However, the DLM's main limitation is the amount of testing performed with different environments. Currently, the DLM has only been tested with Red Hat's cluster manager.

Chapter 2. DLM Locking Model

This chapter presents concepts that will help you use DLM locks effectively in an application. Chapter 3 describes how to use the DLM locking model API routines to implement locking in an application, and chapter 4 presents the API as a whole.

2.1. Overview

In the DLM locking model, a lock resource is the lockable entity. An application acquires a lock on a lock resource. A one-to-many relationship exists between lock resources and locks: a single lock resource can have multiple locks associated with it.

A lock resource can correspond to an actual object, such as a file, a data structure, a database, or an executable routine, but it does not have to correspond to one of these things. The object you associate with a lock resource determines the granularity of the lock. For example, locking an entire database is considered locking at coarse granularity. Locking each item in a database is considered locking at fine granularity.

The following sections provide more information about:

- Lock resources, including lock value blocks and lock queues
- Locks, including lock modes and lock states
- Deadlock, including transaction IDs and lock groups.

2.2. Lock Resources

A lock resource has the following components:

- A name, which is a string of no more than 64 characters
- A lock value block
- A set of lock queues

The following figure shows the components of a lock resource.

<table>
<tr><td rowspan="3">Resource Name</td><td rowspan="3">Lock Value Block</td><td>Grant Queue</td></tr>
<tr><td>Convert Queue</td></tr>
<tr><td>Wait Queue</td></tr>
</table>

The lock manager creates a lock resource in response to the first request for a lock on that lock resource. The lock manager destroys the internal data structures for that lock resource when the last lock held on the lock resource is released.

2.2.1. Lock Value Block

The lock value block (LVB) is a 32-byte character array associated with a lock resource that applications can use to store data. This data is application-specific; the lock manager does not make any direct use of this data. The lock manager allocates space for the LVB when it creates the lock resource. When the lock manager destroys the lock resource, any information stored in the lock value block is also destroyed.

See Chapter 3 for information about using the lock value block.

2.2.2. Lock Resource Queues

Each lock resource has three queues associated with it, one for each possible lock state.

Grant Queue

> The grant queue contains all locks granted by the lock manager on the lock resource, except those locks converting to a mode incompatible with the mode of a granted lock. The lock manager maintains the grant queue as a queue; however, the order of the locks on the queue does not affect processing.

Convert Queue

> The convert queue contains all granted locks that have attempted to convert to a mode incompatible with the mode of the most restrictive currently granted lock. The locks on the convert queue are still granted at the same mode as before the conversion request. The lock manager processes the locks on the convert queue in "first-in, first-out" (FIFO) order. The lock at the head of the queue must be granted before any other locks on the queue can be granted.

Wait Queue

> The wait queue contains all new lock requests not yet granted because their mode is incompatible with the mode of the most restrictive currently granted lock. The lock manager processes the locks on the wait queue in FIFO order.

For more information about the relationship of these lock queues, see Section 2.3.4.

2.3. Locks

In the DLM locking model, you can request a lock from the lock manager on any lock resource. Locks have the following properties:

- A mode that defines the degree of protection provided by the lock
- A state that indicates whether the lock is currently granted, converting, or waiting

2.3.1. Lock Modes

A lock mode indicates whether a process shares access to a lock resource with other processes or whether it prevents other processes from accessing that lock resource while it holds the lock. A lock request specifies a lock mode.

> **Note:** The Distributed Lock Manager does not force a process to respect a lock. Processes must agree to cooperate. They must voluntarily check for locks before accessing a resource and, if a lock incompatible with a request exists, wait for that lock to be released or converted to a compatible mode.

2.3.2. Lock Mode Severity

The lock manager supports six lock modes that range in the severity of their restriction. The following table lists the modes, in order from least severe to most severe, with the types of access associated with each mode.

Table 2-1. Lock Modes

Mode	Requesting Process	Other Processes
Null (NL)	No access	Read or write access
Concurrent Read (CR)	Read access only	Read or write access
Concurrent Write (CW)	Read or write access	Read or write access
Protected Read (PR)	Read access only	Read access only
Protected Write (PW)	Read or write access	Read access only
Exclusive (EX)	Read or write access	No access

Within an application, you can determine which mode is more severe by making a simple arithmetic comparison. Modes that are more severe are arithmetically greater than modes that are less severe.

2.3.3. Lock Mode Compatibility

Lock mode compatibility determines whether two locks can be granted simultaneously on a particular lock resource. Because of their restriction, certain lock combinations are compatible and certain other lock combinations are incompatible.

For example, because an EX lock does not allow any other user to access the lock resource, it is incompatible with locks of any other mode (except NL locks, which do not grant the holder any privileges). Because a CR lock is less restrictive, however, it is compatible with any other lock mode, except EX.

The following table presents a mode compatibility matrix.

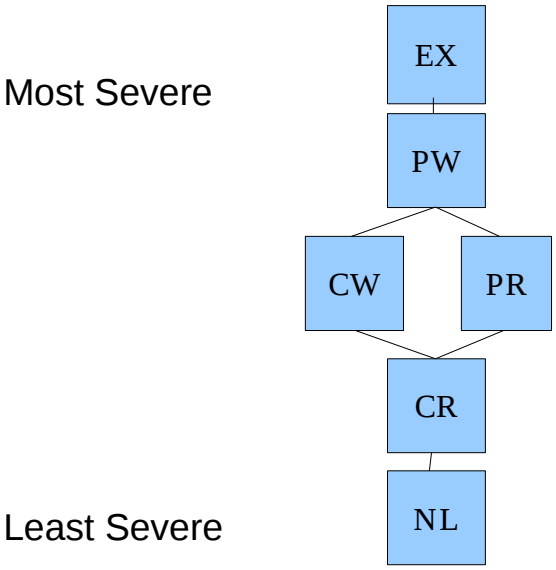
Table 2-2. Lock Mode Compatibility

Requested Lock	Currently Granted Lock					
	NL	**CR**	**CW**	**PR**	**PW**	**EX**
NL	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**
CR	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**No**
CW	**Yes**	**Yes**	**Yes**	**No**	**No**	**No**
PR	**Yes**	**Yes**	**No**	**Yes**	**No**	**No**
PW	**Yes**	**Yes**	**No**	**No**	**No**	**No**
EX	**Yes**	**No**	**No**	**No**	**No**	**No**

NL mode locks grant no privileges to the lock holder. NL mode locks are compatible with locks of any other mode. Applications typically use NL mode locks as place holders for later conversion requests.
CR mode locks allow unprotected read operations. The read operations are unprotected because other processes can read or write the lock resource while the holder of a CR lock is reading the lock resource. CR mode locks are compatible with every other mode lock except EX mode.
CW mode locks allow unprotected read and write operations. CW mode locks are compatible with NL mode locks, CR read mode locks, and other CW mode locks.
PR mode locks allow a lock client to read from a lock resource knowing that no other process can write to the lock resource while it holds the lock. PR mode locks are compatible with NL mode locks, CR mode locks,

and other PR mode locks. PR mode locks are an example of a traditional shared lock.
PW mode locks allow a lock client to read or write to a lock resource, knowing that no other process can write to the lock resource. PW mode locks are compatible with NL mode locks and CR mode locks. Other processes that hold CR mode locks on the lock resource can read it while a lock client holds a PW lock on a lock resource. A PW lock is an example of a traditional update lock.
EX mode locks allow a lock client to read or write a lock resource without allowing access to any other mode lock (except NL). An EX lock is an example of a traditional exclusive lock.

The following figure shows the modes in descending order from most to least severe. Note that, because CW and PR modes are both compatible with three modes, they provide the same level of severity.



2.3.4. Lock States

A lock state indicates the current status of a lock request. A lock is always in one of three states:

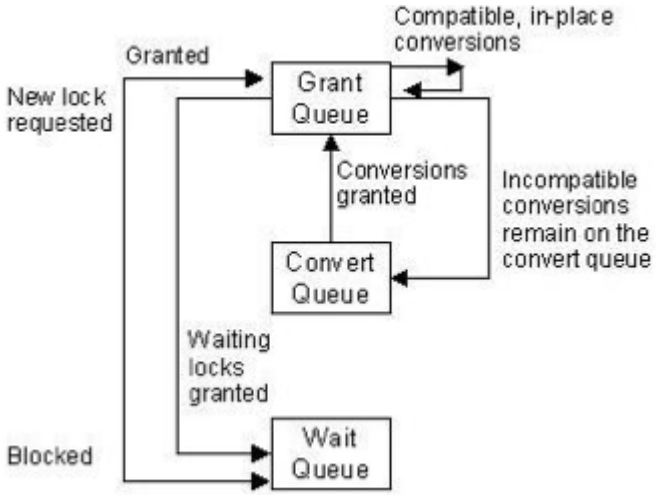
Granted
: The lock request succeeded and attained the requested mode.

Converting
: A client attempted to change the lock mode and the new mode is incompatible with an existing lock.

Blocked
: The request for a new lock could not be granted because conflicting locks exist.

A lock's state is determined by its requested mode and the modes of the other locks on the same resource. The following figure shows all the possible lock state transitions.

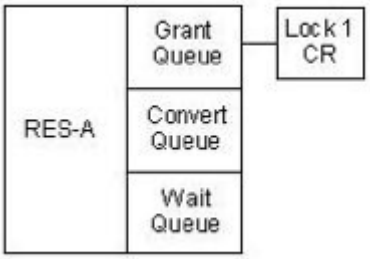
The following sections provide more information about each state. See Section 2.3.9 for a detailed example of the lock state transitions.



2.3.5. Granted

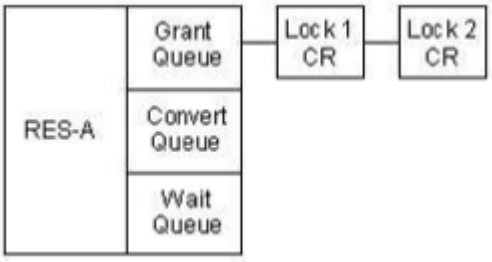
A lock request that attains its requested mode is granted. The lock manager grants a lock if there are currently no locks on the specified lock resource, or if the requested mode is compatible with the mode of the most restrictive, currently granted lock, and the convert queue is empty. The lock manager adds locks in the granted state to the lock resource's grant queue.

For example, if you request a CR mode lock on a lock resource, named RES-A, and there are no other locks, the lock manager grants your request and adds your lock to the lock resource's grant queue. The following figure illustrates the lock resource's queues after this lock operation.



If the lock manager receives another request for a lock on RES-A at mode CR, it grants the request because the mode is compatible with the currently granted lock. The lock manager adds this lock to the lock resource's grant queue.

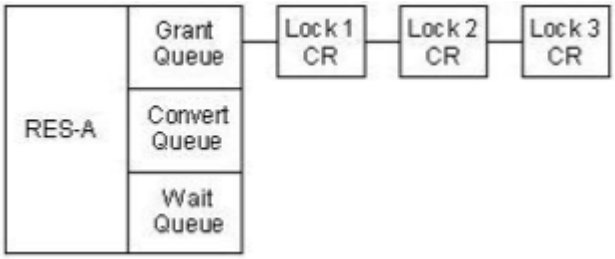
The figure below illustrates the lock resource's queues after these operations.



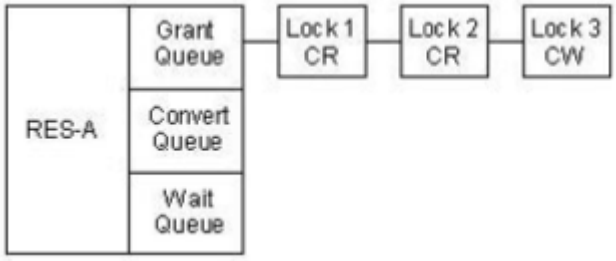
2.3.7. Converting

A lock conversion request changes the mode at which a lock is held. The conversion can promote a lock from a less restrictive to a more restrictive mode, called an up-conversion, or demote a lock from a more restrictive to a less restrictive mode, called a down-conversion. For example, a request to convert the mode of a lock from NL to EX is an up-conversion. Only granted locks can be converted. It is not possible to convert a lock already in the process of converting or a request that is blocked on the wait queue.

The lock manager grants up-conversion requests if the requested mode is compatible with the mode of the most restrictive, currently granted lock, and there are no blocked lock conversion requests waiting on the convert queue. To illustrate, consider the following lock resource with three granted locks, all at CR mode.



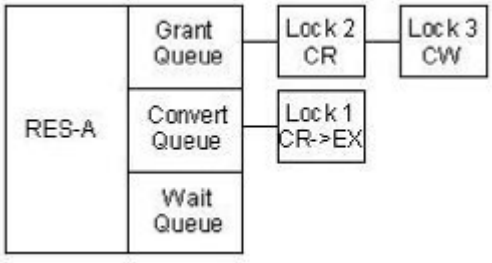
If you request a conversion of Lock 3 from CR mode to CW mode, the lock manager can grant the request because CW mode is compatible with CR mode, and there are no lock conversion requests on the convert queue. The following illustrates the state of the lock queues after this request.



A lock conversion request that cannot be granted transitions into a converting state. The lock manager moves locks that are in converting state from the grant queue to the end of the convert queue. Locks that are in the converting state retain the lock mode they held in the grant queue.

For example, using the previous lock scenario, if you try to convert Lock 1 from CR mode to the more restrictive EX mode, the lock manager cannot grant the request because EX mode is not compatible with the mode of the most restrictive granted lock (CW). The lock manager moves Lock 1 from the grant queue to the convert queue.

The following figure illustrates the lock resource’s queues after the conversion request.



Once there is a lock on the convert queue, all subsequent up-conversion requests get moved to the convert queue, even if the requested mode is compatible with the most restrictive granted lock. For example, using the preceding lock scenario, a request to convert Lock 2 from CR to CW could not be performed because the conversion of Lock 1 is waiting on the convert queue, even though CW mode is compatible with the mode of the most restrictive currently granted lock. The lock manager moves the lock to the end of the convert queue. The following illustrates the state of the lock resource queues after this conversion request.

2.3.7.1. Leaving the Converting State

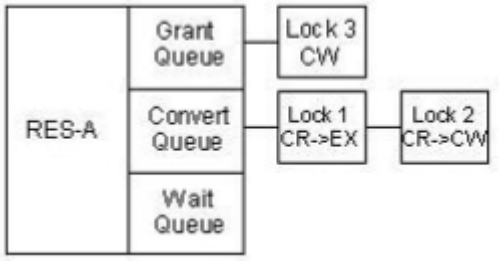
A lock can leave the converting state if any of the following conditions are met:

- The process that requested the lock terminates.
- The process that holds the lock cancels the conversion request. When a conversion request is cancelled, the lock manager moves the lock back to the grant queue at its previously granted mode.
- The requested mode becomes compatible with the most restrictive granted lock, and all previously requested conversions have been granted or cancelled.

2.3.7.2. In-Place Conversions

The lock manager grants all down-conversion requests in-place; that is, the lock is converted to the new mode without being moved to the convert queue, even if there are other lock requests on the convert queue. The lock manager grants all down-conversions because they are compatible with the most restrictive locks on the grant queue (the lock was already granted at a more restrictive mode). For example, given the preceding lock scenario, if you requested a down-conversion of Lock 3 from CW to NL, the lock manager would grant the conversion in-place.

The following illustrates the state of the locks after this conversion.



2.3.7.3. Conversion Deadlock

Because the lock manager processes the convert queue in FIFO order, the conversion of the lock at the head of the convert queue must occur before any other conversions on the convert queue can be granted. Occasionally, the lock at the head of the convert queue can be blocked by one of the other lock conversion requests on the convert queue. The lock conversion requests on the convert queue are all blocked by the lock at the head of convert queue. Thus, a deadlock cycle is created.

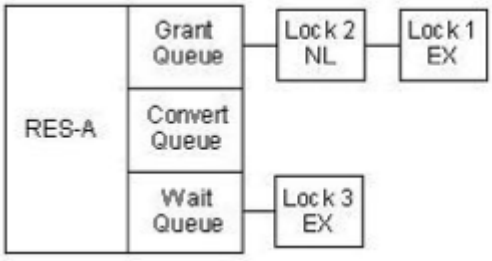
The previous example illustrates conversion deadlock. Even after the down-conversion of Lock 3 to NL mode, Lock 1 cannot be granted because it is blocked by Lock 2, also on the convert queue. Lock 1 cannot convert to EX mode because Lock 2 is still granted at CR mode, which is incompatible with EX mode. Thus, Lock 1 is blocked by Lock 2 and Lock 2 is blocked by Lock 1. For more information about conversion deadlock, see Section 2.4.2

2.3.8. Blocked

If you request a lock and the mode is incompatible with the most restrictive granted lock, your request is blocked. The lock manager adds the blocked lock request to the lock resource's wait queue. (You can choose to have the lock manager abort a request that cannot be immediately granted instead of putting it on the wait queue. For more information, see Section 3.3.6.

Continuing the previous example, if you request a new EX lock on the same lock resource (Lock 3), the lock manager cannot grant your request because EX is not compatible with the most restrictive mode of a currently granted lock (Lock 1 at EX mode). The lock manager adds this lock request to the end of the lock resource's wait queue.

The figure below illustrates the lock resource's queues after this request.

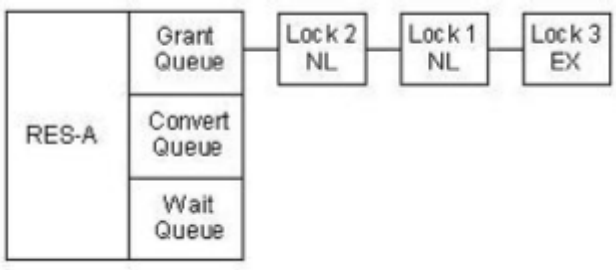


A lock can leave the wait queue if any of the following conditions are met:

- The process that requested the lock terminates.
- The requester cancels the blocked lock. When a blocked lock is cancelled, the lock manager removes it from the wait queue.
- The lock request becomes compatible with the mode of the most restrictive lock currently granted on the lock resource, and there are no converting locks or blocked locks queued ahead of the lock request. The lock manager processes the wait queue in FIFO order, after processing the convert queue. No blocked request can be unblocked by the release of a granted lock, regardless of the compatibility of its mode, until all blocked requests on the convert queue and all blocked requests ahead of it on the wait queue have been granted.

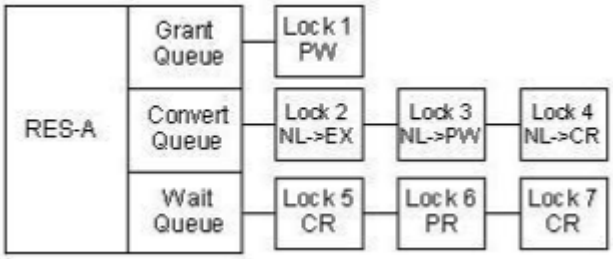
Thus, a lock request can become blocked only as the result of a lock request, but it can unblock as a result of the release or conversion of some other lock. (An exception is made in the case of deadlock. See Section 2.4.) Continuing the previous example, if you convert Lock 1 from EX to NL, the lock manager can grant the blocked request because EX is compatible with NL mode locks. The lock manager moves Lock 3 from the wait queue to the grant queue.

The following figure illustrates the lock resource's queues after the conversion of Lock 1.



2.3.9. Interaction of Queues

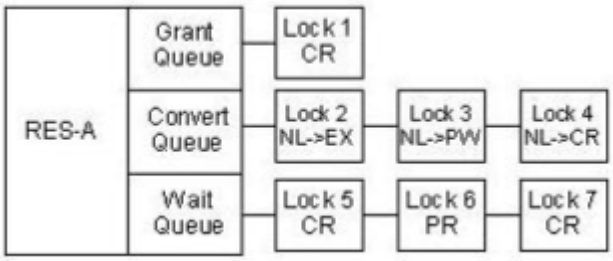
To illustrate how the lock manager processes a lock resource's grant, convert, and wait queues, consider the lock scenario illustrated in the following figure. This example has one lock on the grant queue, three lock conversion requests blocked on the convert queue, and three lock requests blocked on the wait queue.



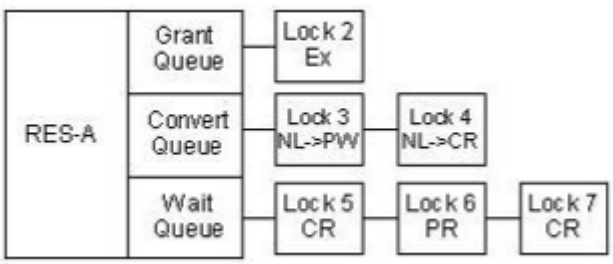
If you request a down-conversion of Lock 1 from PW to CR, the lock manager can grant the conversion request because a CR lock is compatible with the mode of the most restrictive currently granted lock. (Lock 1 itself is the most restrictive currently granted lock; a lock cannot block itself.) Note that the lock manager performs an in-place conversion of Lock 1, without adding it to the end of the convert queue.

After granting the conversion, the lock manager checks if the change allows any blocked conversions to be granted, starting at the head of the convert queue. Because CR and EX are not compatible, Lock 2 cannot be unblocked. Because the lock manager processes the convert queue in FIFO order, no other locks on the convert queue can be granted, even though their requested modes are compatible with CR. Because there are conversions still blocked on the convert queue, the blocked locks on the wait queue can not be processed either.

The following figure illustrates the lock resource's queues after the Lock 1 conversion request is completed.



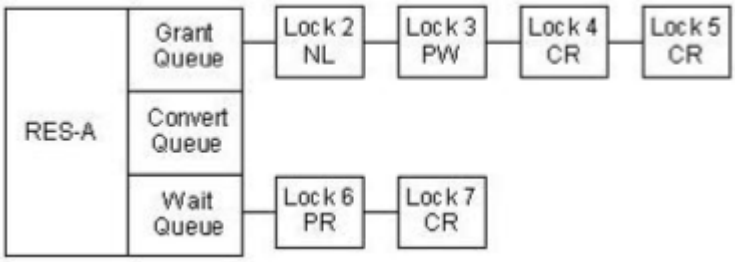
If you release Lock 1, the lock manager can grant Lock 2, the EX lock waiting at the head of the conversion queue. Because the mode requested by Lock 3 is not compatible with an EX mode lock, no other request on the convert queue can be granted. The following figure illustrates the lock resource's queues after the lock conversion operation.



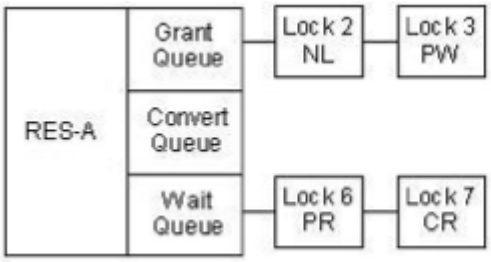
If you request a down-conversion of Lock 2 from EX to NL, the lock manager grants the conversion in-place because a NL lock is compatible with the mode of the most restrictive currently granted lock (EX). The lock manager checks the convert queue to see if the change allows any blocked conversion requests to be granted. The lock manager can grant Lock 3 and Lock 4 on the convert queue because PW and CR are compatible.

In addition, because there are no locks left on the convert queue, the lock manager can process the locks blocked on the wait queue. The lock manager can grant the lock at the head of the wait queue, Lock 5, because a CR lock is compatible with the most restrictive currently granted lock. The lock manager cannot grant Lock 6, however, because PR is incompatible. Because the lock manager processes the wait queue in FIFO order, Lock 7 cannot be granted, even though it is compatible with the most restrictive currently granted lock.

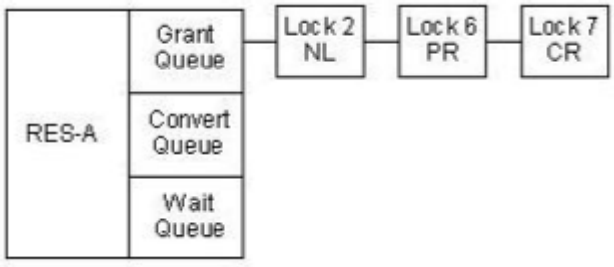
The following figure illustrates the lock resource's queues after the conversion operation of Lock 2.



If you release Lock 4 and Lock 5, the lock manager cannot unblock the locks on the wait queue because the mode of Lock 6 is still not compatible with the most restrictive currently granted lock.



If you release Lock 3, the lock manager can grant Lock 6 at the head of the wait queue and lock 7 because their modes are compatible.



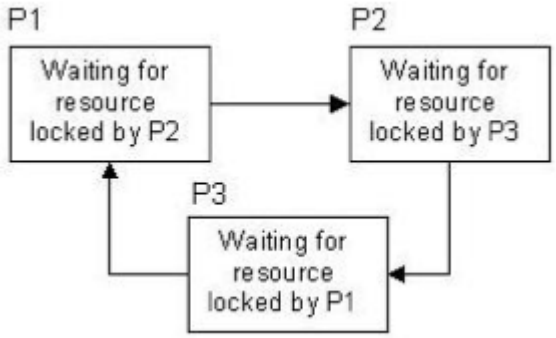
2.4. Deadlock

The lock manager faces deadlock when two or more lock requests are blocking each other with incompatible modes for lock requests. Three types of deadlock can occur: normal deadlock, conversion deadlock, and self-client deadlock.

2.4.1. Normal Deadlock

Normal deadlock occurs when two or more processes are blocking each other in a cycle of granted and blocked lock requests. For example, say Process P1 has a lock on Resource R1 and is blocked waiting for a lock on Resource R2 held by Process P2. Process P2 has a lock on Resource R2 and is blocked waiting for a lock on Resource R3 held by Process P3, and Process P3 has a lock on resource R3 and is blocked waiting for a lock on Resource R1 held by Process P1.

This scenario is illustrated in the following figure.



2.4.2. Conversion Deadlock

Conversion deadlock occurs when the requested mode of the lock at the head of the convert queue is incompatible with the granted mode of some other lock also on the convert queue. The first lock cannot convert because its requested mode is incompatible with a currently granted lock. The other lock cannot convert because the convert queue is strictly FIFO.

2.4.3. Self-Client Deadlock

Self-client deadlock occurs when a single client requests a lock on a lock resource on which it already holds a lock and its first lock blocks the second request. For example, if Process P1 requests a lock on a lock resource on which it already holds a lock, the second lock may be blocked.

2.4.4. Deadlock Detection

The lock manager periodically checks for all types of deadlock by following chains of blocked locks and the locks blocking them. If the lock manager detects a cycle of locks that indicate deadlock (that is, if the same process occurs more than once in a chain), it denies the request that has been blocked the longest. The lock manager sets the status field in the lock status block associated with this lock request to DLM_DEADLOCK and queues for execution of the AST routine associated with the request. (For more information about how the lock manager returns the status of lock requests, see Section 3.3.1.1.)

> **Note:** The lock manager does not arbitrate among lock client applications to resolve a deadlock condition. The lock manager simply cancels one of the requests causing deadlock and notifies the client. The lock client applications, when they receive a return value indicating deadlock, must decide how to handle the deadlock. In most cases, releasing existing locks and then re-acquiring them should eliminate the deadlock condition.
>
> **Note:** deadlock detection is only implemented from Red Hat Linux 5.2 onwards

2.4.5. Transaction IDs

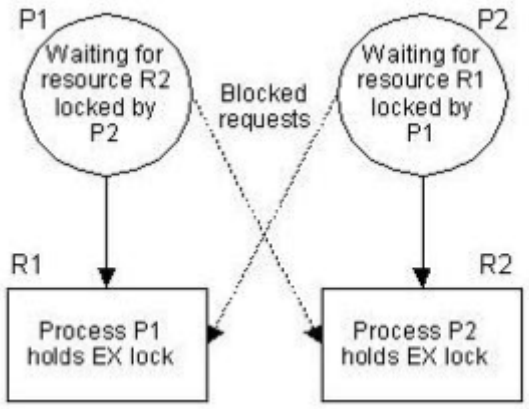
By cancelling one of the requests causing a deadlock, the lock manager prevents clients contending for the same lock resources from blocking each other indefinitely. Additionally, the lock manager supports transaction IDs, a mechanism clients can use to improve application throughput by diminishing the impact of deadlock when it does occur.

When determining whether a deadlock cycle exists, the lock manager normally assumes the process that created the lock owns the lock. By specifying a transaction ID (also called an XID or deadlock ID) as part of a lock request, a lock client can attribute ownership of a lock related to a particular task to a "transaction" rather than to itself. For deadlock detection, therefore, a transaction replaces a process or group as the owner of a lock.

Furthermore, transaction IDs allow different clients to request locks on the same transaction. A unique transaction ID should be associated with each transaction (task). Since transaction IDs do not span nodes, the lock manager considers equivalent transaction IDs on different nodes to be different transaction IDs.

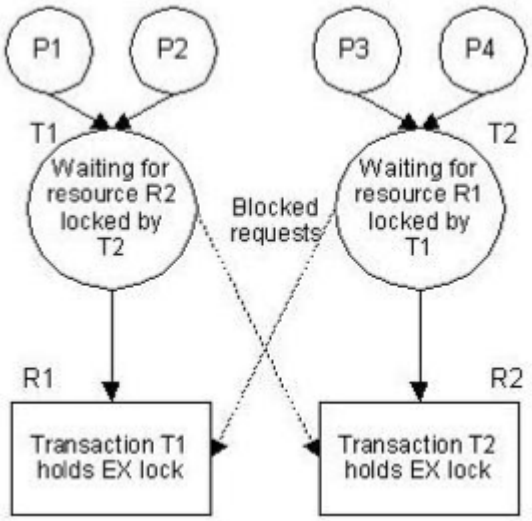
Transaction IDs are beneficial when multiple client processes request locks on a common transaction and each process works on multiple tasks.

Consider the following example: Process P1 holds an exclusive lock on Resource R1 and requests an exclusive lock on Resource R2. Process P1 will not release the lock on Resource R1 until the lock manager grants the lock on Resource R2. Process P2, meanwhile, holds an exclusive lock on Resource R2 and requests an exclusive lock on Resource R1. Process R2 will not release the lock on Resource R2 until the lock manager grants the lock on Resource R1. This is illustrated in the following figure. The dotted lines indicate blocked requests.



The processes in this example are deadlocked. Each process is blocking the other and neither is able to do any work. To break this deadlock, the lock manager cancels one of the blocked requests and notifies the requesting client by returning a deadlock status.

Using transaction IDs would allow the processes to work on different tasks even though they are blocked on a particular transaction. To expand on the previous example: Process P1 holds an exclusive lock on Resource R1 that it asked for using transaction ID T1. Process P2, which is also working on task T1, requests an exclusive lock on Resource R2. Process P3, however, holds an exclusive lock on R2 that it asked for using transaction ID T2. Process P4, also working on task T2, requests an exclusive lock on Resource R1. This is illustrated in the following figure.



Again, deadlock occurs. Task T1 is blocked by task T2 and task T2 is blocked by task T1. No work will be done on these transactions until the lock manager breaks the deadlock by cancelling one of the requests. The transactions are blocked, but not necessarily the lock client processes. If the lock clients are concurrently working on other tasks, they can continue to work on these tasks. When the lock manager detects the deadlock and cancels one of the requests causing the deadlock, the lock client applications can once again resume work on these transactions.

Note: Transaction IDs are only available from RHEL 5.1 onwards

Chapter 3. Using DLM API Routines

This chapter describes how to use the DLM locking API routines in a distributed Linux application. Chapter 4 provides reference information on the routines discussed in this chapter.

3.1. Overview

The three primary programming tasks you must perform to implement locking in an application are:

- Acquiring locks on a lock resources
- Converting existing locks to different modes
- Releasing locks

To perform these tasks, applications use the routines in the DLM locking model API to make requests to the lock manager. For example, to make an asynchronous request for a lock on a lock resource, an application would use either the dlm_lock or dlm_lock_wait routine.

The DLM locking model API also includes routines that help applications perform ancillary tasks related to manipulating locks. For example, the DLM locking model API includes the dlm_dispatch routine that applications must use to receive the asynchronous notification of the status of their request. The following sections describe how to perform these primary locking tasks, including any ancillary tasks that may be required.

3.2. Prerequisites

This section describes the header files you must include in your application to use the DLM locking model API routines, the libraries with which you must link your application, and the primary data structure applications you must use to implement locking.

3.2.1. Header Files

To use the DLM locking model API routines, you must specify the following include directive:

```
#include <libdlm.h>
```

The /usr/include/libdlm.h file defines the constants, data structures, status codes, and flags used by the DLM locking model API.

> **Note:** Your client code may require additional flags, be sure to include those flags necessary for your code to build.

3.2.2. Library Files

The distributed Linux software includes a library for linking against user-space DLM. For threaded applications this is called libdlm.so, for non-threaded applications it is called libdlm_lt.so

3.2.2.1. Linking Clients to the DLM

Specify the following libraries when you invoke the linkage editor for a multi-threaded application:
`-ldlm -lpthread`

The libdlm.so library contains the routines that support the Distributed Lock Manager.

3.2.3. Data Structure

The DLM locking model API includes a data structure, called the lock status block, that your application can use to specify the value of the lock value block associated with a lock resource (if required)

In addition, the lock manager uses the lock status block to return the following information to your application:

- The status of the request
- The lock ID the lock manager has assigned to the lock request
- The value stored in the lock value block

The lock status block, defined in the /usr/include/libdlm.h include file, has the following structure:

```
struct dlm_lksb {
        int sb_status;
        uint32_t sb_lkid;
        char sb_flags;
        char *sb_lvbptr;
};
```

The following list describes each field:

sb_status
: Contains the status code returned by the lock manager. The status codes are normal errno codes, plus some extras defined in the /usr/include/libdlm.h include file. See the API reference for more detail for specific status codes for each call.

sb_lkid
: Contains the lock ID that the lock manager assigned to this lock request.

sb_lvbptr
: Determines the lock value block, an array that applications can use to store application-specific data. The size of the array is specified by the value of the constant DLM_LVB_LEN which is defined in the /usr/include/libdlm.h header file as 32 bytes.

sb_flags
: Flags returned from the lock operation providing further information about a successful call.

3.3. Acquiring or Converting a Lock on a Lock Resource

To acquire a lock on a lock resource, or convert an existing lock to a different mode, you make a request to the lock manager using one of the locking routines described in the following sections. If the lock resource does not exist, the lock manager creates it.

The Distributed Lock Manager supports both asynchronous and synchronous lock routines.

3.3.1. Requesting Locks Asynchronously

An asynchronous lock routine queues the request and then immediately returns control to the lock client

making the call. The status code indicates whether the request was queued successfully. The lock client can perform other operations while it waits for the lock manager to resolve the request. When the lock manager resolves the request, it queues the AST routine specified by the request for execution. The lock process must then trigger the execution of this AST routine.

The routines that request a lock asynchronously are:

dlm_lock
: Makes an asynchronous (non-blocking) request for a lock on a lock resource or converts an existing lock to a different mode.

dlm_ls_lock
: Makes an asynchronous (non-blocking) request for a lock on a lock resource in a specified lockspace or converts an existing lock to a different mode.

dlm_ls_lockx
: Makes an asynchronous (non-blocking) request for a lock on a lock resource or converts an existing lock to a different mode, and specifies a transaction ID and/or timeout for that lock.

When requesting an asynchronous lock, you supply the following information:

- The name of the lock resource, along with the length of the name.
- The requested mode of the lock.
- A pointer to a lock status block; for a conversion request, the lock status block must contain a valid lock ID.
- Flags that determine characteristics of the lock operation. For a conversion request, you must specify the LKF_CONVERT flag.
- A pointer to an AST routine that the lock manager queues for execution when it grants (or denies, aborts, or cancels) your lock request. Your application triggers the execution of this routine by calling the dlm_dispatch routine.
- Optionally, a pointer to a blocking AST routine that the lock manager queues for execution when the lock is blocking another lock request. Your application triggers the execution of this routine by calling the dlm_dispatch routine.
- A pointer to arguments you want passed to either the AST routine and/or the blocking AST routine.

The following example uses the dlm_lock routine to request a CR mode lock on a lock resource named RES-A. This resource will be in the default lockspace.

```
#include <libdlm.h>
int status;
struct dlm_lksb lksb; /* lock status block */
extern void ast_func();
.
.
.
     status = dlm_lock( LKM_CRMODE, /* mode */
                        &lksb, /* addr of lock status block */
                        LKF_VALBLK, /* flags */
                        "RES-A", /* name */
                        5, /* namelen */
                        0, /* Parent    not used */
                        ast_func, /* ast routine triggered */
                        0, /* astargs */
                        NULL, /* */
                        NULL); /* Range */
     if ( status != 0 )
     {
          perror( "dlmlock");
```

```
}
```

When the lock manager accepts your lock request, it passes a token back to your application, called a **lock ID**, that uniquely identifies your lock. The lock manager writes the lock ID in the **sb_lkid** field of the lock status block. (You specify the address of the lock status block as an argument to the dlm_lock and dlm_ls_lock routines.) All subsequent requests concerning that lock, such as conversion requests, must use the lock ID to identify the lock. When your application triggers the execution of the AST routine, the lock manager writes the status of your request in the status field of the lock status block. See Chapter 4 for a complete list of all possible status codes returned by the dlm_lock and dlm_ls_lock routines.

3.3.2. Requesting Locks Synchronously

A synchronous lock routine performs the same function as an asynchronous lock routine, but does not return control to the calling process until the request is resolved. A synchronous lock routine queues the request and then places the calling process into a wait state until the lock manager resolves the request. A process making a synchronous lock request does not have to poll for an AST; it simply waits until the request returns.
The routines that request a lock synchronously are:

dlm_lock_wait
: Requests a lock and waits for a return or converts an existing lock to a different mode.

dlm_ls_lock_wait
: Requests a lock and waits for a return or converts an existing lock to a different mode in a specified lockspace.

When requesting a synchronous lock, you supply the following information:

- The name of the lock resource, along with the length of the name.
- The requested mode of the lock.
- A pointer to a lock status block. For a conversion request, the lock status block must contain a valid lock ID.
- Flags that determine characteristics of the lock operation. For a conversion request, you must specify the LKF_CONVERT flag.
- Optionally, a pointer to a blocking AST routine that the lock manager queues for execution when the lock is blocking another lock request. Your application triggers the execution of this routine by calling the dlm_dispatch routine.
- A pointer to arguments you want passed to the blocking AST routine.

3.3.3. Triggering AST Routines

This section describes AST handling for user-space clients.

To trigger the execution of AST routines (both regular and blocking), your application must call the dlm_dispatch routine.

dlm_dispatch is usually called from dlm_pthread_init, this starts a thread dedicated to running AST routines. If you have an open lockspace then dlm_ls_pthread_init can be called for each lockspace. See Chapter 4 for more information on lockspaces.

3.3.4. Keeping Track of Lock Requests

To keep track of the lock requests your application makes, which may be granted in a different sequence than they were requested, assign each request a unique identifier using the astarg and/or bastarg parameters to the dlm_lock and dlm_ls_lock routines. When you trigger an AST routine, this argument identifies which request is associated with this return.

For example, the value passed in the astarg parameter to the dlm_lock routine could be an index into an array of lock status blocks. Each time your application makes a lock or conversion request, it would use another lock status block from the array by incrementing this index. The index value would then be passed as the value of the astarg parameter to the dlmlock routine. When the request returns, the argument passed to the AST routine identifies which lock status block in the array is associated with the returned value.

3.3.5. Sample Locking Application

The following example illustrates how to make a lock request and use pthreads and AST routines. The example also illustrates how to use the astarg parameter to track lock requests.

It basically implements simple synchronous interface to the DLM routines. You don't need to use this because dlm_lock_wait is provided, but this example does illustrate many of the points made so far.

```
#include <sys/types.h>
#include <errno.h>
#include <string.h>
#include <inttypes.h>
#include <libdlm.h>
#include <pthread.h>

struct lock_wait {
        pthread_cond_t cond;
        pthread_mutex_t mutex;
        struct dlm_lksb lksb;
};

static void sync_ast_routine(void *arg)
{
        struct lock_wait *lwait = arg;

        pthread_mutex_lock(&lwait->mutex);
        pthread_cond_signal(&lwait->cond);
        pthread_mutex_unlock(&lwait->mutex);
}

static int sync_lock(const char *resource, int mode, int flags, int *lockid)
{
        int status;
        struct lock_wait lwait;

        if (!lockid) {
                errno = EINVAL;
                return -1;
        }

        /* Conversions need the lockid in the LKSB */
        if (flags & LKF_CONVERT)
                lwait.lksb.sb_lkid = *lockid;

        pthread_cond_init(&lwait.cond, NULL);
        pthread_mutex_init(&lwait.mutex, NULL);
        pthread_mutex_lock(&lwait.mutex);
```

```
        status = dlm_lock(mode,
                          &lwait.lksb,
                          flags,
                          resource,
                          strlen(resource),
                          0, sync_ast_routine, &lwait, NULL, NULL);
        if (status)
                return status;

        /* Wait for it to complete */
        pthread_cond_wait(&lwait.cond, &lwait.mutex);
        pthread_mutex_unlock(&lwait.mutex);

        *lockid = lwait.lksb.sb_lkid;

        errno = lwait.lksb.sb_status;
        if (lwait.lksb.sb_status)
                return -1;
        else
                return 0;
}
```

3.3.6 Avoiding the Wait Queue

If the lock manager cannot grant your lock request, it adds your request to the end of the wait queue, along with all other blocked lock requests on the lock resource. You can specify that the lock manager not queue your request if it cannot be granted immediately by specifying the LKF_NOQUEUE flag as an argument to the lock routine.

If your lock request cannot be granted immediately, the lock open routine returns the status 0 and the AST is queued with the status EAGAIN in the status field of the lock status block.

3.3.7. Specifying a Timeout Value for a Lock Request

Blocked locks remain on the wait queue until they are granted (or cancelled, denied, or aborted). You can specify to the lock manager that you only want your request to remain on the wait queue for a certain time period. You specify this value in the **timeout** parameter to dlm_ls_lockx()

In the following example, the lock request specifies a timeout value of five seconds. (The value is specified in hundredths of seconds.)

```
#include <libdlm.h>

int status;
struct dlm_lksb lksb; /* lock status block */
extern void ast_func();

status = dlm_ls_lockx(lockspace, /* Previously opened */
                      LKM_CRMODE, /* mode */
                      &lksb, /* lock status block */
                      0, /* flags */
                      "RES-A", /* name */
                      5, /* namelen */
                      ast_func, /* routine to trigger ast */
                      0, /* astargs */
                      0, /* bast */
                      0, /* XID */
                      500); /* Timeout */
        if ( status != DLM_NORMAL )
        {
            perror( "dlmlock");
```

```
}
```

3.3.8. Excluding a Lock Request from Deadlock Detection Processing

To exclude a lock request from the lock manager's deadlock detection processing, specify the LKF_NODLCKWT flag with the dlm_lock routine.

3.3.9. Requesting Persistent Locks

When a client terminates while holding one or more locks, the lock manager purges any locks that do not have the LKF_PERSISTENT flag set. Locks originally requested with the LKF_PERSISTENT flag set remain after a client terminates. Applications use orphan locks to prevent other lock clients from accessing a lock resource until any clean up made necessary by the termination has been performed. Once the LKF_PERSISTENT flag is set (whether by the initial lock request or by a subsequent conversion), that flag remains set for the duration of that lock.

3.4. Releasing a Lock on a Lock Resource

To release an existing lock or cancel a lock request blocked on the convert queue or wait queue, you must use the dlm_unlock or dlm_ls_unlock routine. When releasing a lock, you supply the following information:

- A valid lock ID or a pointer to a lock value block
- Flags

  Note: The flag you specify depends on the type of operation you are requesting. The following options are available:
  - If you want to cancel a lock request or a conversion request that is blocked, specify the LKF_CANCEL flag
  - If you want to modify the lock value block, specify the LKF_VALBLK flag.
  - If you want to invalidate the lock value block, specify the LKF_IVVALBLK flag.

When you release or cancel a lock on a lock resource, the lock manager performs the following processing, depending on which queue the lock was located:

Grant queue
: If you release a granted lock, the lock manager removes the lock from the grant queue.

Convert queue
: If you cancel a conversion request, the lock manager puts the lock back on the grant queue at its old grant mode.

Wait queue
: The lock manager removes the lock from the wait queue.

Note that cancelling a waiting or converting lock request will also cancel the AST routine that the request is waiting for. So if you are using synchronous lock requests you *must* use an asynchronous cancel request so that the original lock request gets the AST with ECANCEL status – this happens because the astaddr is cannot be changed in a dlm_unlock call. You can change the astarg if you want, but the status of ECANCEL is usually enough to identify what has happened. If you call dlm_unlock_wait() to cancel a synchronous lock request, the original dlm_lock_wait call will never return because dlm_unlock_wait() hijacks the ast routine.

The following example releases a lock, identified by its lock ID. The example illustrates a typical way

applications use an array of lock status blocks to keep track of the locks they acquire. The application uses the astarg parameter to assign a number that identifies each lock. The astarg parameter is an index into the array.

```
#include <libdlm.h>
int status;
struct dlm_lksb lksb[MAXLOCKS]; /* lock status block */
int index=0;
.
.
.
status = dlm_unlock(lksb[index].lockid, 0, 0);
if (status != 0)
{
     perror("Unlock failed");
}
```

3.5. Purging Locks

The DLM API includes the dlm_purge routine to facilitate releasing locks. The dlm_purge routine releases all locks owned by a particular client, identified by its process ID. When you specify a process ID of 0, all orphaned locks for the specified node ID are released.

> **Note:** Locks owned by LIVE clients can only be purged by the owner of the lock. Otherwise, dlm_purge only
> affects or orphaned locks.

> **Note:** dlm_purge is not currently implemented.

3.6. Manipulating the Lock Value Block

Every lock resource can include a fixed number of bytes of storage, called a lock value block (LVB), that applications can use to store data.

You cannot assign a value to an Lock Value Block (LVB) when you acquire a lock on a lock resource; you can only read its current value. To modify the contents of the LVB, you must hold an EX lock or a PW lock on a lock resource. You can assign a value to an LVB when:

- Releasing the EX or PW mode lock
- Down-converting the EX or PW mode lock to a less restrictive mode

The following sections describe how to modify an LVB using these methods.

3.6.1. Setting an LVB When Releasing an EX or PW Lock

You can modify a lock value block when you release an EX or PW lock by using the dlm_unlock or the dlm_unlock_wait routine. You specify a pointer to the value you want assigned to the lock value block as an argument to the routine. You must also set the LKF_VALBLK flag.

> **Note:** There must be another lock on the lock resource. If you release the last lock on a lock resource, the lock manager destroys the lock resource and the LVB associated with it.

The following example illustrates how to set an LVB. The example assumes that the process holds an EX lock on the lock resource.

```
#include <libdlm.h>

struct dlm_lksb lksb;
char valblk[32];

strcpy(valblk,"my lvb");

status = dlm_unlock_wait(lksb.lockid, /* mode */
                         &valblk, /* lock value block */
                         LKF_VALBLK); /* flags */
         if ( status != 0 )
         {
             perror("dlmlock");
         }
```

3.6.2. Setting an LVB When Converting an EX or PW Lock

You can modify a lock value block when down-converting an EX or PW mode lock to a less restrictive mode using one of the lock open routines. You specify a pointer to the value you want assigned to the lock value block in the lock status block passed in as a part of the request. (This pointer must be valid when the LKF_VALBLK flag is set.) The following example illustrates how to set an LVB when down-converting an EX mode lock on a lock resource.

```
#include <libdlm.h>

struct dlm_lksb lksb;
char valblk[16];

strcpy(valblk, "my lvb");
lksb.valblk = &valblk;

status = dlm_lock_wait(LKM_CRMODE, /* mode */
                       &lksb, /* lock status block */
                       LKF_CONVERT | LKF_VALBLK, /* flags */
                       "RES-A", /* name */
                       5, /* namelen */
                       0, /* parent, unused */
                       bast_func, /* routine for blocking ast */
                       0, /* astargs */
                       0 ); /* range, unused */
         if ( status != 0 )
         {
              perror( "dlmlock");
         }
```

3.6.3. Invalidating a Lock Value Block

If a client holding an EX or PW mode lock on a lock resource terminates abruptly, the lock manager sets a flag to notify other clients holding locks on the lock resource that the contents of the LVB are no longer reliable. This LVB is considered invalid. An LVB is valid when the lock manager first creates the lock resource, in response to the first lock request, before any client can assign a value to the LVB. An application may want to deliberately invalidate an LVB. For example, you can invalidate an LVB to ensure that other lock holders on a lock resource reset the value of the LVB.

To invalidate an LVB, specify the LKF_IVVALBLK flag when releasing a lock using the dlm_unlock routine or when down-converting a lock to a less restrictive mode using one of the lock open routines. Your application must hold an EX mode or PW mode lock on the lock resource to invalidate the LVB. If you hold a less restrictive lock (lower than PW mode), your request is ignored.

The following example illustrates how to invalidate an LVB when down converting an EX mode lock on a

lock resource.

```
status = dlm_lock_wait(LKM_CRMODE, /* mode */
                       &lksb, /* lock status block */
                       LKF_CONVERT | LKF_IVVALBLK, /* flags */
                       "RES-A", /* name */
                       5, /* namelen */
                       0, /* parent, unused */
                       bast_func, /* routine for blocking ast */
                       0, /* astargs */
                       0 ); /* range, unused */
         if ( status != DLM_NORMAL )
         {
              perror( "dlmlock");
         }
```

If the lock value block is invalid, then any attempt to read it will set the flag DLM_SBF_VALNOTVALID in the lock status block member: sb_flags.

3.6.4. Using Lock Value Blocks

The purpose of the lock value block is to provide a client application with a small amount of state information that is guaranteed to be consistent throughout the cluster. Applications can use the storage provided by the LVB for any purpose.

3.6.4.1. Implementing a Local Disk Cache

An application can use a lock value block to implement local disk caches across a number of different nodes that share access to a common disk. In a local cache scheme, each node maintains a copy of the disk blocks in local memory to speed access to the data on the common disk. To make sure that each system always accesses the most up-to-date copy of the disk block in its cache, an application acquires a lock on each disk block in the cache.

When the application references a disk block from the cache, it acquires the lock associated with that block and it keeps a record of the current value of the lock value block. When an application modifies the disk block, it changes the value in the lock value block. The next time the application accesses the disk block, it reads the value of the lock value block and compares it to the value that it stored previously. If the values differ, the application knows the disk block has been modified, that the copy of the disk block it has in its cache is invalid, and that it must read the up-to-date contents of the disk block from disk.

3.7. Handling Returned Status Codes

The dlm routines return normal error codes in either errno (if the lock operation fails immediately) or in the lock status block (if it fails later on in processing). If the API call returns 0, then an AST will be delivered to the program, with further status information in the lock status block sb_status field.

In addition to normal errno values, there are two DLM-specific error codes that are returned in the lock status block sb_status field:

ECANCEL
: Returned to a lock request or conversion request that has been cancelled by a call to dlm_unlock with the LKF_CANCEL flag.

EUNLOCK

Returned after a successful unlock operation.

You might also see EINPROG in a lock status block sb_status field, this indicates that the lock operation is currently being processed by the lock manager. If you see this in the lksb then the lock ID is also valid and can be used to cancel the lock operation if necessary.

3.8. Lockspaces

The DLM allows locks to be partitioned into "lockspaces", and these can be manipulated by userspace calls. It is possible (though not recommended) for an application to have multiple lockspaces open at one time in the same application.

The purpose of lockspaces is to provide a private namespace for locks that are part of a single application. For example GFS uses a lockspace for each mounted GFS filesystem, so that an open file on one filesystem does not affect a file of the same name or inode on another filesystem. Similarly a database application might use a lockspace for each distinct database in a cluster.

Lockspaces are identified by name and are cluster-wide. A lockspace named “myLS” is the same lockspace on all nodes in the cluster and locks will contend for resources the same as if they were on the same system. Lockspace names are case-sensitive so “MyLS” is a distinct lockspace to “myLS”.

The DLM provides a default lockspace for applications that do a small amount of locking and can be sure that their lock names will not contend with other applications in the cluster that are also using that lockspace. This lockspace is called “default” and is always available. If it does not exist it will be created when it is first accessed.

For applications that do need their own lockspace they should create it using dlm_create_lockspace() or open it using dlm_open_lockspace(), dlm_create_lockspace() also opens it. Note that you need to have root privileges to create a new lockspace but you do not need them to open it. Access to lockspaces is governed by the permissions on the file /dev/misc/dlm_<lockspace> device files.

After opening or creating a lockspace you will have a dlm_lshandle_t which should be passed to the dlm_ls_* routines as the first parameter to indicate that the locks you are manipulating are contained in the lockspace you have just created/opened. You should also use the dlm_ls_pthread_init() routine to create a thread to service ASTs for that lockspace.

If you use multiple lockspace in a single program, you should call dlm_ls_pthread_init for each lockspace.

3.9. pthreads

So far, this document assumes that locking application will use pthreads to manager the ASTs (callbacks) returned from the lock manager. This is by far the most convenient method of using the lock manager but it is possible to write an application that does not use pthreads if you need to. The library libdlm_lt.so provides a subset of DLM calls and routines to allow you to do your own AST handling.

The userland locking library uses a file descriptor to communicate with the in-kernel lock manager. This file descriptor can be returned by calling dlm_get_fd() to get the file descriptor associated with the default lockspace or dlm_ls_get_fd() to get the file descriptor associated with a lockspace created or opened by the application.

This file descriptor should be passed into poll() or select() like any other file descriptor. When it becomes active you should then call dlm_dispatch(fd) to process the callbacks associated with it.

The following example shows how to wait for a lock request to complete. It assumes that the AST routine sets the “ast_called” variable to 1 when it is called.

```
/* Using poll(2) to wait for and dispatch ASTs */
static int poll_for_ast()
{
    struct pollfd pfd;

    pfd.fd = dlm_get_fd();
    pfd.events = POLLIN;
    while (!ast_called)
    {
        if (poll(&pfd, 1, 0) < 0)
        {
            perror("poll");
            return -1;
        }
        dlm_dispatch(pfd.fd);
    }
    ast_called = 0;
    return 0;
}
```

If you elect not to use pthreads, then you can only use the asynchronous DLM calls, ie those not ending in _wait().

3.10. Lockspace Devices

As mentioned above, lockspaces are accessed via devices in /proc/misc.

When a lockspace is created the kernel part of the DLM creates a misc device for communication with that lockspace and userspace programs. Udev is then responsible for creating the actual device file in /dev/misc. Once this has been done, libdlm sets the protection on the device to that requested by the dlm_create_lockspace() call. The file will be called /dev/misc/dlm_<lsname> so eg: a call to

dlm_create_lockspace(“myls”, 0666)

will create a file called /dev/misc/dlm_myls with protection mode 0666 (depending on umask). This file will be removed when the lockspace is released using dlm_release_lockspace.

It is important to realise that the device protection (including ACLs or SELinux if appropriate) is the *only* way that access to locks in a lockspace is controlled. Once a process has managed to open a lockspace device then it can manipulate all locks in the lockspace, regardless of where or how the locks were created. This includes lockspaces that would normally exist only in the kernel (eg for GFS).

In addition, there is a file /dev/misc/dlm-control which is the device through which lockspace manipulation is carried out. The default protection for this file is 0666 to allow unprivileged users to create the default lockspace. However, creation of all other lockspaces requires root privilege.

Chapter 4. User-space API Reference

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdint.h>
#include <libdlm.h>

cc -D_REENTRANT prog.c  -ldlm -lpthread

cc prog.c  -ldlm_lt
```

There are basically two interfaces to libdlm. The first is the "dead simple" one that has limited functionality and assumes that the application is linked with pthreads. The second is the full-featured DLM interface that looks identical to the kernel interface.

There are also two libdlm libraries, one of which uses pthreads (libdlm) and one of which does not (libdlm_lt) . Both provide the same interface to the locking system, but the pthread version has some additional calls do to with threading. In addition, the 'simple' interface (detailed below) is only available in the pthread (libdlm) library because it uses pthreads to gain its simplicity.

See the source CVS or tarball directory **dlm/tests/usertest** for examples of use of both these APIs.

4.1. The simple API

This provides two API calls, lock_resource() and unlock_resource(). Both of these calls block until the lock operation has completed - using a worker thread to deal with the callbacks that come from the kernel.

`int lock_resource(const char *resource, int mode, int flags, int *lockid);`

This function locks a named (NUL-terminated) resource and returns the lockid if successful.

mode may be any of

LKM_NLMODE LKM_CRMODE LKM_CWMODE LKM_PRMODE LKM_PWMODE LKM_EXMODE

f**lags** may be any combination of

LKF_NOQUEUE	Don't wait if the lock cannot be granted immediately, will return EAGAIN if this is so.
LKF_CONVERT	Convert lock to new mode. *lockid must be valid, resource name is ignored.
LKF_QUECVT	Add conversion to the back of the convert queue - only valid for some convert operations
LKF_PERSISTENT	Don't automatically unlock this lock when the process exits (must be root).

Return codes:
0 is returned if the call completed successfully. If not, -1 is returned and errno is set to one of the following:

EINVAL	An invalid parameter was passed to the call (eg bad lock mode or flag)
ENOMEM	A (kernel) memory allocation failed
EAGAIN	LKF_NOQUEUE was requested and the lock could not be granted
EBUSY	The lock is currently being locked or converted
EFAULT	The userland buffer could not be read/written by the kernel (this indicates a library problem)

`int unlock_resource(int lockid);`

Unlocks the resource.

Return codes:
0 is returned if the call completed successfully. If not, -1 is returned and errno is set to one of the following:

EINVAL	An invalid parameter was passed to the call (eg bad lock mode or flag)
EINPROGRESS	The lock is already being unlocked
EBUSY	The lock is currently being locked or converted

Example

```
int lockid;
status = lock_resource(“mylock”, LKM_EXMODE, 0, &lockid);
if (status == 0)
      status = unlock_resource(lockid);
```

4.2. The Full API

This interface is identical to the kernel interface with the exception of the lockspace argument. All userland locks sit in the same lockspace by default.

libdlm can be used in pthread or non-pthread applications. For pthread applications simply call the following function before doing any lock operations. If you're using pthreads, remember to define _REENTRANT at the top of the program or using -D_REENTRANT on the compile line.

`int dlm_pthread_init()`

Creates a thread to receive all lock ASTs. The AST callback function for lock operations will be called in the context of this thread. If there is a potential for local resource access conflicts you must provide your own pthread-based locking in the AST routine.

`int dlm_pthread_cleanup()`

Cleans up the default lockspace threads after use. Normally you don't need to call this, but if the locking code is in a dynamically loadable shared library this will probably be necessary.

For non-pthread based applications the DLM provides a file descriptor that the program can feed into poll/select. If activity is detected on that FD then a dispatch function should be called.

int dlm_get_fd()

Returns a file-descriptor for the DLM suitable for passing in to poll() or select().

int dlm_dispatch(int fd)

Reads from the DLM and calls any AST routines that may be needed. This routine runs in the context of the caller so no extra locking is needed to protect local resources.

```
int dlm_lock(uint32_t mode,
             struct dlm_lksb *lksb,
             uint32_t flags,
             const void *name,
             unsigned int namelen,
             uint32_t parent,
             void (*astaddr) (void *astarg),
             void *astarg,
             void (*bastaddr) (void *astarg),
             struct dlm_range *range);


      struct dlm_lksb {
              int      sb_status; /* Final status of lock operation */
              uint32_t sb_lkid;   /* ID of lock. Returned from dlm_lock()
                                     on first use. Used as input to
                                     dlm_lock() for a conversion */
              char     sb_flags;  /* Completion flags, see below */
              char     sb_lvbptr; /* Optional pointer to lock value block */
      };

      struct dlm_range {
              uint64_t ra_start;
              uint64_t ra_end;
      };
```

mode	lock mode:	
	LKM_NLMODE	NULL Lock
	LKM_CRMODE	Concurrent read
	LKM_CWMODE	Concurrent write
	LKM_PRMODE	Protected read
	LKM_PWMODE	Protected write
	LKM_EXMODE	Exclusive
flags	LKF_NOQUEUE	Don't queue the lock. If it cannot be granted return EAGAIN
	LKF_CONVERT	Convert an existing lock
	LKF_VALBLK	Lock has a value block
	LKF_QUECVT	Put conversion to the back of the queue
	LKF_EXPEDITE	Grant a NL lock immediately regardless of other locks on the conversion queue
	LKF_PERSISTENT	Specifies a lock that will not be unlocked when the process exits.
	LKF_CONVDEADLK	Enable conversion deadlock
	LKF_NODLCKWT	Do not consider this lock when trying to detect deadlock conditions
	LKF_NODLCKBLK	Do not consider this lock as blocking other locks when trying to detect deadlock conditions.

LKF_NOQUEUEBAST	Send blocking ASTs even for NOQUEUE operations
LKF_HEADQUE	Add locks to the head of the convert or waiting queue
LKF_NOORDER	Avoid the VMS rules on grant order when using range locks

lksb Lock status block.
This structure contains the returned lock ID, the actual status of the lock operation (all lock ops are asynchronous) and the value block if LKF_VALBLK is set. It is up to the application to allocate space for the LVB if it is required.

name Name of the lock. Can be binary, max 64 bytes in length. Ignored for lock conversions.

namelen Length of the above name. Ignored for lock conversions.

parent ID of parent lock or NULL if this is a top-level lock (ignored in RHEL5)

ast Address of AST routine to be called when the lock operation completes.
The final completion status of the lock will be in the lksb. the AST routine must not be NULL.

astarg Argument to pass to the AST routine (most people pass the lksb in here but it can be anything you like.)

bast Blocking AST routine. Address of a function to call if this lock is blocking another. The function will be called with astarg as the parameter.

range An optional structure of two uint64_t that indicate the range of the lock. Locks with overlapping ranges will be granted only if the lock modes are compatible. locks with non-overlapping ranges (on the same resource) do not conflict. A lock with no range is assumed to have a range encompassing the largest possible range. ie. 0-0xFFFFFFFFFFFFFFFF. Note that it is more efficient to specify no range than to specify the full range above.
Lock ranges are only supported in RHEL4.

dlm_lock and its variants acquire and convert locks in the DLM.

dlm_lock operations are asynchronous. If the call to dlm_lock returns an error then the operation has failed and the AST routine will not be called. If dlm_lock returns 0 it is still possible that the lock operation will fail. The AST routine will be called when the locking is complete or has failed and the status is returned in the lksb.

For conversion operations the name and namelen are ignored and the lock ID in the LKSB is used to identify the lock to be converted.

If a lock value block is specified then in general, a grant or a conversion to an equal-level or higher-level lock mode reads the lock value from the resource into the caller's lock value block. When a lock conversion from EX or PW to an equal-level or lower-level lock mode occurs, the contents of the caller's lock value block are written into the resource. If the LVB is invalidated the lksb.sb_flags member will be set to DLM_SBF_VALNOTVALID. Lock values blocks are always 32 bytes long.

If the AST routines or parameter are passed to a conversion operation then they will overwrite those

values that were passed to a previous dlm_lock call.

Return codes:
0 is returned if the call completed successfully. If not, -1 is returned and errno is set to one of the following:

EINVAL	An invalid parameter was passed to the call (eg bad lock mode or flag)
ENOMEM	A (kernel) memory allocation failed
EAGAIN	LKF_NOQUEUE was requested and the lock could not be granted
EBUSY	The lock is currently being locked or converted
EFAULT	The userland buffer could not be read/written by the kernel
EDEADLOCK	The lock operation is causing a deadlock and has been cancelled. If this was a conversion then the lock is reverted to its previously granted state. If it was a new lock then it has not been granted. (NB Only conversion deadlocks are detected in RHEL4)

If an error is returned in the AST, then lksb.sb_status is set to the one of the above values instead of zero.

```
int dlm_lock_wait(uint32_t mode,
                  struct dlm_lksb *lksb,
                  uint32_t flags,
                  const void *name,
                  unsigned int namelen,
                  uint32_t parent,
                  void *bastarg,
                  void (*bastaddr) (void *bastarg),
                  struct dlm_range *range);
```

As above except that the call will block until the lock is granted or has failed. The return from the function is the final status of the lock request (ie that was returned in the lksb after the AST routine was called if the lock operation suceeded).

```
int dlm_unlock(uint32_t lkid,
               uint32_t flags,
               struct dlm_lksb *lksb,
               void *astarg)
```

lkid Lock ID as returned in the lksb

flags flags affecting the unlock operation:

LKF_CANCEL	CANCEL a pending lock or conversion. This returns the lock to it's previously granted mode (in case of a conversion) or unlocks it (in case of a waiting lock).
LKF_IVVALBLK	Invalidate value block

lksb LKSB to return status and value block information.

astarg New parameter to be passed to the completion AST.

The completion AST routine is the last completion AST routine specified in a dlm_lock call. If dlm_lock_wait() was the last routine to issue a lock, dlm_unlock() should be used to release the lock. If dlm_lock() was the last routine to issue a lock then either dlm_unlock() or dlm_unlock_wait() may be called.

Unlocks are also asynchronous. The AST routine is called when the resource is successfully unlocked (see below).

Return codes:
0 is returned if the call completed successfully. If not, -1 is returned and errno is set to one of the following:

EINVAL	An invalid parameter was passed to the call (eg bad lock mode or flag)
EINPROGRESS	The lock is already being unlocked
EBUSY	The lock is currently being locked or converted
ENOTEMPTY	An attempt to made to unlock a parent lock that still has child locks.
ECANCEL	A lock conversion was successfully cancelled
EUNLOCK	An unlock operation completed successfully (sb_status only)
EFAULT	The userland buffer could not be read/written by the kernel

If an error is returned in the AST, then lksb.sb_status is set to the one of the above numbers instead of zero.

```
int dlm_unlock_wait(uint32_t lkid,
                    uint32 flags,
                    struct dlm_lksb *lksb)
```

As above but returns when the unlock operation is complete either because it finished or because an error was detected. In the case where the unlock operation was successful no error is returned.

The return value of the function call is the status of issuing the unlock operation. The status of the unlock operation itself is in the lock status block. Both of these must be checked to verify that the unlock has completed successfully.

4.3. Lock Query Operations

The dlm query API is a method of getting information about other locks that are held (or pending) on a resource that you already hold a lock for. If you don't have a lock on a known resource then a getting a NL lock on it will allow you to query it without affecting ether locking operations (much). It is not a method of getting a list of resources in a lockspace, use dlm_tool or similar for that purpose.

Once you have a lock ID, you can use these calls to find out (eg) which locks are waiting to be granted, or which locks are blocking your lock, including which cluster node and PID of the process (if relevant). You can also get general information about the lock such as the resource name (if you've forgotten it!), LVB state and value, and the node where it is mastered.

The main structure passed into dlm_query() is the dlm_queryinfo struct (see below). The main constituents of this are two pointers, one to a gdlm_resinfo structure which will be filled in with information about the resource, and an array of gdlm_lockinfo structures which contains information about all the locks that match the requested criteria; or, at least, as many as will fit into the allocated space.

Note: The Lock query API is only currently available on RHEL 4

```
int dlm_query(struct dlm_lksb *lksb,
```

```
            int query,
            struct dlm_queryinfo *qinfo,
            void (*ast_routine(void *astarg)),
            void *astarg);
```

The operation is asynchronous, the ultimate status and data will be returned into the dlm_query_info structure which should be checked when the ast_routine is called. The lksb must contain a valid lock ID in sb_lkid which is used to identify the resource to be queried, status will be returned in sb_status; As with the locking calls an AST routine will be called when the query completes if the call itself returns 0. See below for full details of the query call.

```
int dlm_query_wait(struct dlm_lksb *lksb,
                   int query,
                   struct dlm_queryinfo *qinfo)
```

Same as dlm_query() except that it waits for the operation to complete. When the operation is complete the status will be in the lksb. Both the return value from the function call and the condition code in the lksb must be evaluated.

If the provided lock list is too short to hold all the locks, then sb_status in the lksb will contain E2BIG but the list will be filled in as far as possible. Either gqi_lockinfo or gqi_resinfo may be specified as NULL if that information is not required.

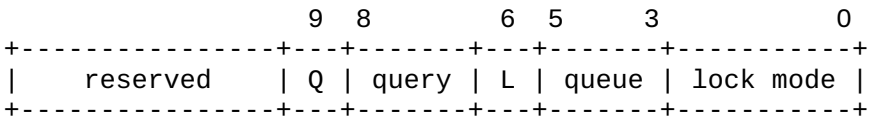
```
/* Structures passed into and out of the query */

struct gdlm_lockinfo
{
  int   lki_lkid;          /* Lock ID on originating node */
  int   lki_parent;
  int   lki_node;          /* Originating node (not master) */
  int   lki_ownpid;        /* Owner pid on the originating node */
  uint8 lki_state;         /* Queue the lock is on */
  int8  lki_grmode         /* Granted mode */
  int8  lki_rqmode;        /* Requested mode */
  struct dlm_range lki_grrange /* Granted range, if applicable */
  struct dlm_range lki_rqrange /* Requested range, if applicable */
};

struct gdlm_resinfo
{
  int  rsi_length;
  int  rsi_grantcount; /* No. of nodes on grant queue */
  int  rsi_convcount;  /* No. of nodes on convert queue */
  int  rsi_waitcount;  /* No. of nodes on wait queue */
  int  rsi_masternode; /* Master node for this resource */
  char rsi_name[DLM_RESNAME_MAXLEN]; /* Resource name */
  char rsi_valblk[DLM_LVB_LEN];    /* Master's LVB contents, if applicable */
};

struct dlm_queryinfo
{
  struct dlm_resinfo  *gqi_resinfo; /* Points to a single resinfo struct */
  struct dlm_lockinfo *gqi_lockinfo;/* This points to an array of structs */
  int                  gqi_locksize;  /* input */
  int                  gqi_lockcount; /* output */
};
```

The query is made up of several blocks of bits as follows:



lock mode is a normal DLM lock mode or DLM_LOCK_THIS to use the mode of the lock in sb_lkid.

queue is a bitmap of
DLM_QUERY_QUEUE_WAIT
DLM_QUERY_QUEUE_CONVERT
DLM_QUERY_QUEUE_GRANT

or one of the two shorthands:
DLM_QUERY_QUEUE_GRANTED (for WAIT+GRANT)
DLM_QUERY_QUEUE_ALL (for all queues)

L is an optional flag DLM_QUERY_LOCAL which specifies that a remote access should not happen. Only lock information that can be gleaned from the local node will be returned so be aware that it may not be complete. It's useful for getting the master node ID or the resource name perhaps.

The **query** is one of the following:
DLM_QUERY_LOCKS_HIGHER
DLM_QUERY_LOCKS_LOWER
DLM_QUERY_LOCKS_EQUAL
DLM_QUERY_LOCKS_BLOCKING
DLM_QUERY_LOCKS_NOTBLOCK
DLM_QUERY_LOCKS_ALL

which specifies which locks to look for by mode, either the lockmode is lower, equal or higher to the mode at the bottom of the query. DLM_QUERY_ALL will return all locks on the resource.

DLM_QUERY_LOCKS_BLOCKING returns only locks that are blocking the current lock. The lock must not be waiting for grant or conversion for this to be a valid query, the other flags are ignored.

DLM_QUERY_LOCKS_NOTBLOCKING returns only locks that are granted but NOT blocking the current lock.

Q specifies which lock queue to compare. By default the granted queue is used. If the flags DLM_QUERY_RQMODE is set then the requested mode will be used instead.

The "usual" way to call dlm_query is to put the address of the dlm_queryinfo struct into lksb.sb_lvbptr and pass the lksb as the AST param, that way all the information is available to you in the AST routine.

Return codes:
0 is returned if the call completed successfully. If not, -1 is returned and errno is set to one of the following:

EINVAL	An invalid parameter was passed to the call (eg bad lock mode or flag)
ENOMEM	A (kernel) memory allocation failed
ENOBUFS	A (kernel) communications buffer allocation failed

E2BIG	The ggi_lockinfo array is too small to hold all the results.
EFAULT	The userland buffer could not be read/written by the kernel

If an error is returned in the AST, then lksb.sb_status is set to the one of the above numbers instead of zero.

Example:

This example display all other locks that are held against the resource up to a maximum of 100.

```
#define MAX_QUERY_LOCKS 100

int status;
struct dlm_lksb tmplksb;
struct dlm_queryinfo qinfo;
struct dlm_resinfo resinfo;

lksb.sb_lkid = <lockid>;
qinfo.gqi_resinfo = &resinfo;
qinfo.gqi_lockinfo = malloc(sizeof(struct dlm_lockinfo) * MAX_QUERY_LOCKS);
qinfo.gqi_locksize = MAX_QUERY_LOCKS;

status = dlm_query_wait(&tmplksb,
                        DLM_QUERY_QUEUE_ALL | DLM_QUERY_LOCKS_ALL,
                        &qinfo);
if (status)
    perror("Query failed");

/* Dump resource info */
printf("lockinfo: status     = %d\n", lksb->sb_status);
printf("lockinfo: resource   = '%s'\n", qi->gqi_resinfo->rsi_name);
printf("lockinfo: grantcount = %d\n", qi->gqi_resinfo->rsi_grantcount);
printf("lockinfo: convcount  = %d\n", qi->gqi_resinfo->rsi_convcount);
printf("lockinfo: waitcount  = %d\n", qi->gqi_resinfo->rsi_waitcount);
printf("lockinfo: masternode = %d\n", qi->gqi_resinfo->rsi_masternode);
printf("\n");

/* Dump all the locks */
for (i = 0; i < qi->gqi_lockcount; i++)
{
    struct dlm_lockinfo *li = &qi->gqi_lockinfo[i];

    printf("lockinfo: lock: lkid        = %x\n", li->lki_lkid);
    printf("lockinfo: lock: master lkid = %x\n", li->lki_mstlkid);
    printf("lockinfo: lock: parent lkid = %x\n", li->lki_parent);
    printf("lockinfo: lock: node        = %d\n", li->lki_node);
    printf("lockinfo: lock: pid         = %d\n", li->lki_ownpid);
    printf("lockinfo: lock: state       = %d\n", li->lki_state);
    printf("lockinfo: lock: grmode      = %d\n", li->lki_grmode);
    printf("lockinfo: lock: rqmode      = %d\n", li->lki_rqmode);
    printf("\n");
}
```

4.4. Lockspace Operations

The DLM allows locks to be partitioned into "lockspaces", and these can be manipulated by userspace calls. It is possible (though not recommended) for an application to have multiple lockspaces open at one time.

All the above calls work on the "default" lockspace, which should be fine for most users. The calls below

allow you to isolate your application from all others running in the cluster. Remember, lockspaces are a cluster-wide resource, so if you create a lockspace called "myls" it will share locks with a lockspace called "myls" on all nodes.

`dlm_lshandle_t dlm_create_lockspace(const char *name, mode_t mode);`

This creates a lockspace called <name> and the mode of the file user to access it will be <mode> (subject to umask as usual). The lockspace must not already exist on this node, if it does -1 will be returned and errno will be set to EEXIST. If you really want to use this lockspace you can then user dlm_open_lockspace() below. The name is the name of a misc device that will be created in /dev/misc.

On success a handle to the lockspace is returned, which can be used to pass into subsequent dlm_ls_lock/unlock calls. Make no assumptions as to the content of this handle as it's content may change in future.

The caller must have CAP_SYSADMIN privileges to do this operation.

Return codes:
0 is returned if the call completed successfully. If not, -1 is returned and errno is set to one of the following:

EINVAL	An invalid parameter was passed to the call
ENOMEM	A (kernel) memory allocation failed
EEXIST	The lockspace already exists
EPERM	Process does not have capability to create lockspaces
ENOSYS	A fatal error occurred initialising the DLM

Any error returned by the open() system call

`int dlm_release_lockspace(const char *name, dlm_lshandle_t lockspace, int force)`

Deletes a lockspace. If the lockspace still has active locks then -1 will be returned and errno set to EBUSY. Both the lockspace handle /and/ the name must be specified. This call also closes the lockspace and stops the thread associated with the lockspace, if any.

Note that other nodes in the cluster may still have locks open on this lockspace. This call only removes the lockspace from the current node. If the force flag is set then the lockspace will be removed even if another user on this node has active locks in it. Existing users will NOT be notified if you do this, so be careful.

The caller must have CAP_SYSADMIN privileges to do this operation.

Return codes:
0 is returned if the call completed successfully. If not, -1 is returned and errno is set to one of the following:

EINVAL	An invalid parameter was passed to the call
EPERM	Process does not have capability to release lockspaces
EBUSY	The lockspace could not be freed because it still contains locks and force was not set.

`dlm_lshandle_t dlm_open_lockspace(const char *name)`

Opens an already existing lockspace and returns a handle to it.

Return codes:
0 is returned if the call completed successfully. If not, -1 is returned and errno is set to an error returned by the open() system call

```
int dlm_close_lockspace(dlm_lshandle_t lockspace)
```

Close the lockspace. Any locks held by this process will be freed. If a thread is associated with this lockspace then it will be stopped.

Return codes:
0 is returned if the call completed successfully. If not, -1 is returned and errno is set to one of the following:

EINVAL lockspace was not a valid lockspace handle

```
int dlm_ls_get_fd(dlm_lshandle_t lockspace)
```

Returns the file descriptor associated with the lockspace so that the user call use it as input to poll/select.

```
int dlm_ls_pthread_init(dlm_lshandle_t lockspace)
```

Initialise threaded environment for this lockspace, similar to dlm_pthread_init() above.

```
int dlm_ls_lock(dlm_lshandle_t lockspace,
            int mode,
            struct dlm_lksb *lksb,
            uint32_t flags,
            void *name,
            unsigned int namelen,
            uint32_t parent,
            void (*ast) (void *astarg),
            void *astarg,
            void (*bast) (void *astarg),
            struct dlm_range *range)
```

Same as dlm_lock() above but takes a lockspace argument.

```
int dlm_ls_lock_wait(dlm_lshandle_t lockspace,
                int mode,
                struct dlm_lksb *lksb,
                uint32_t flags,
                void *name,
                unsigned int namelen,
                uint32_t parent,
                void *bastarg,
                void (*bast) (void *bastarg),
                struct dlm_range *range)
```

Same as dlm_lock_wait() above but takes a lockspace argument.

```
int dlm_ls_unlock(dlm_lshandle_t lockspace,
                  uint32_t lkid,
                  uint32_t flags,
                  struct dlm_lksb *lksb,
                  void *astarg)
```

Same as dlm_unlock above but takes a lockspace argument.

```
int dlm_ls_unlock_wait(dlm_lshandle_t lockspace,
                       uint32_t lkid,
                       uint32_t flags,
                       struct dlm_lksb *lksb)
```

Same as dlm_unlock_wait above but takes a lockspace argument.

```
int dlm_ls_query(dlm_lshandle_t lockspace,
                 struct dlm_lksb *lksb,
                 int query,
                 struct dlm_queryinfo *qinfo,
                 void (*ast_routine(void *astarg)),
                 void *astarg);
```

Same as dlm_query above but takes a lockspace argument.

```
int dlm_ls_query_wait(dlm_lshandle_t lockspace,
                      struct dlm_lksb *lksb,
                      int query,
                      struct dlm_queryinfo *qinfo)
```

Same as dlm_query_wait above but takes a lockspace argument.

One further point about lockspace operations is that there is no locking around the creating/destruction of lockspaces in the library when when using pthreads, so it is up to the application to only call dlm_*_lockspace when it is sure that no other locking operations are likely to be happening within that process.