Programming Locking Applications

Kristin Thomas
IBM Corporation

kristint@us.ibm.com

Programming Locking Applications

by Kristin Thomas

Published August 2001

Copyright © 2001 by IBM Corporation. All rights reserved.

This document may be reproduced or distributed in any form without prior permission provided the copyright notice is retained on all copies. Modified versions of this document may be freely distributed, provided that they are clearly identified as such, and this copyright is included intact. This document is provided "AS IS," with no express or implied warranties. Use the information in this document at your own risk.

Special Notices

This publication/presentation was produced in the United States. IBM may not offer the products, programs, services or features discussed herein in other countries, and the information may be subject to change without notice. Consult your local IBM business contact for information on the products, programs, services, and features available in your area. Any reference to an IBM product, program, service, or feature is not intended to state or imply that only IBM's product, program, service, or feature may be used. Any functionally equivalent product, program, service, or feature that does not infringe on IBM's intellectual property rights may be used instead.

Questions on the capabilities of non-IBM products should be addressed to suppliers of those products. IBM may have patents or pending patent applications covering subject matter in this presentation. Furnishing this presentation does not give you any license to these patents. Send license inquiries, in writing, to IBM Director of Licensing, IBM Corporation, New Castle Drive, Armonk, NY 10504-1785 USA. All statements regarding IBM's future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only. Contact your local IBM office or IBM authorized reseller for the full text of a specific Statement of General Direction.

The information contained in this presentation has not been submitted to any formal IBM test and is distributed "AS IS." While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. The use of this information or the implementation of any techniques described herein is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. Customers attempting to adapt these techniques to their own environments do so at their own risk.

The information contained in this document represents the current views of IBM on the issues discussed as of the date of publication. IBM cannot guarantee the accuracy of any information presented after the date of publication.

Any performance data in this document was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements quoted in this book may have been made on development-level systems. There is no guarantee these measurements will be the same on generally-available systems. Some measurements quoted in this book may have been estimated through extrapolation. Actual results may vary. Users of this book should verify the applicable data for their specific environment.

A full list of U.S. trademarks owned by IBM may be found at http://www.ibm.com/legal/copy/trade.html. UNIX is a registered trademark of The Open Group in the United States and other countries. Linux is a trademark of Linus Torvalds. Other company, product, and service names may be trademarks or service marks of others.

Table of Contents

List of Tables

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. Locking Models

The Distributed Lock Manager provides two distinct locking models: the DLM locking model and the UNIX ® System V locking model.

The two locking models exist in separate namespaces and do not interact. Therefore, the Distributed Lock Manager manages simultaneous lock traffic of both types. A single application can use both types of locks.

1.2.1. 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.

1.2.2. UNIX Locking Model

The UNIX locking model supports UNIX System V region locking. Using the UNIX locking model, you can define regions of fine granularity within a resource. Locks in the UNIX locking model are either shared or exclusive.

For more information about the UNIX locking model, see Chapter 4.

1.3. Application Programming Interfaces

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. The API routines that implement the UNIX locking model are described in Chapter 4.

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

- `libdlm.so`: for user-space DLM client applications
- `libdlmk.o`: for kernel-space DLM client applications

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 daemon. 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 fallover occurs.

To increase the likelihood of local processing, the lock manager can also move a lock resource master to the node that is accessing the lock resource most frequently. This process is called lock resource master migration. Using these techniques, the lock manager attempts to increase lock throughput and reduce the network traffic overhead. Applications can also explicitly instruct the lock manager to process a lock locally.

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; the DLM will use only a single IP address at any one time for a node, but it supports switching among different IP addresses for each 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 been tested with:

- The open source Heartbeat environment, versions 0.4.8k and 0.4.9, from http://linux-ha.org
- The closed source clustering RSCT environment, from IBM

> **Note:** Currently, Heartbeat lacks a consistent membership protocol, so it does not provide sufficient consistency above two nodes. This limitation is being examined by IBM developers. For information on configuring the DLM and Heartbeat, see Chapter 6.

The FailSafe package from SGI has not yet been tested with the DLM, but, like Heartbeat, it is an open source environment, and it is supported for up to 16 nodes.

> **Note:** Users must install the entire FailSafe product, which may be significantly heavier than desired.

We will attempt to provide support for other, untested environments, but we cannot guarantee levels of function or performance for these environments.

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.

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:

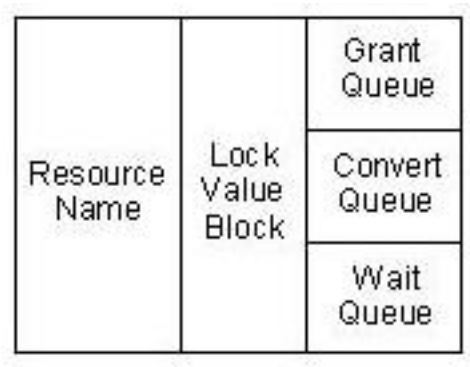
- Lock resources, including lock value blocks and lock queues
- Locks, including lock modes and lock states
- Deadlock, including transaction IDs and lock groups.

2.2. Lock Resources

A lock resource has the following components:

- A name, which is a string of no more than 31 characters
- A lock value block
- A set of lock queues

The following figure shows the components of a lock resource.



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 16-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

Mode	Requesting Process	Other Processes
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.

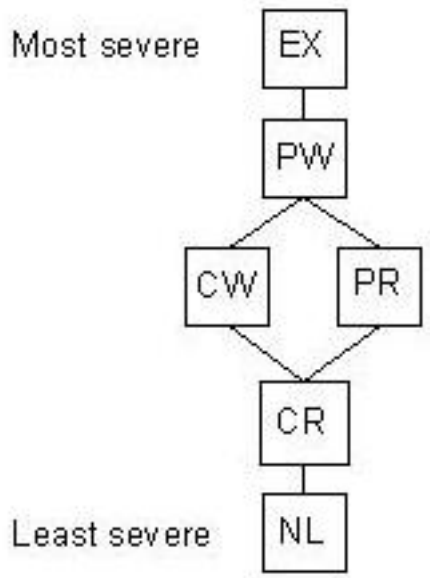
CW mode locks allow unprotected read and write operations. CW mode locks are compatible with NL mode locks, CR read mode locks, and other CW mode locks.

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.

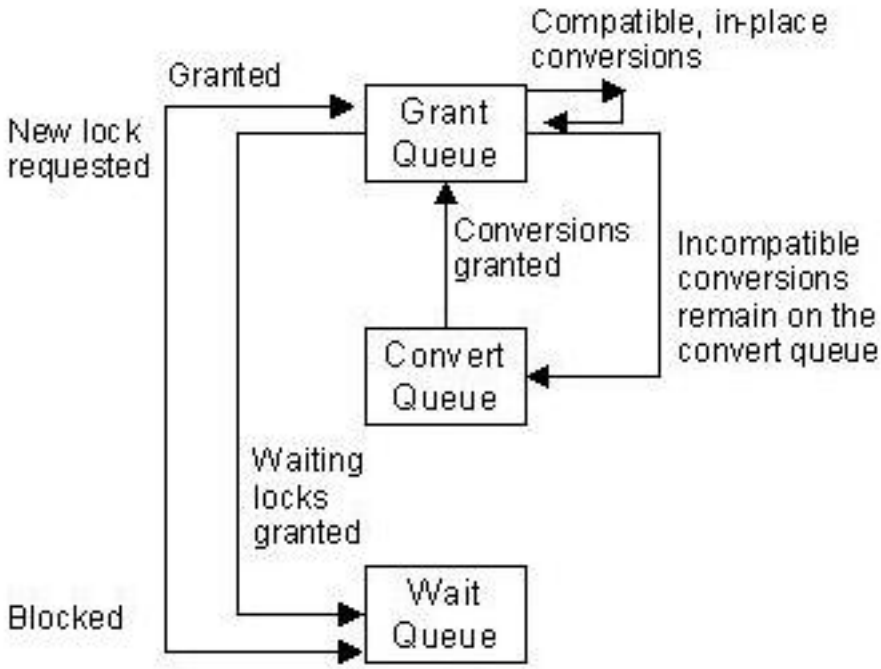
Converting

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.

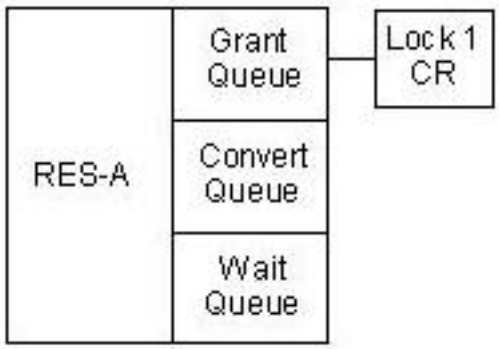


The following sections provide more information about each state. See Section 2.3.9 for a detailed example of the lock state transitions.

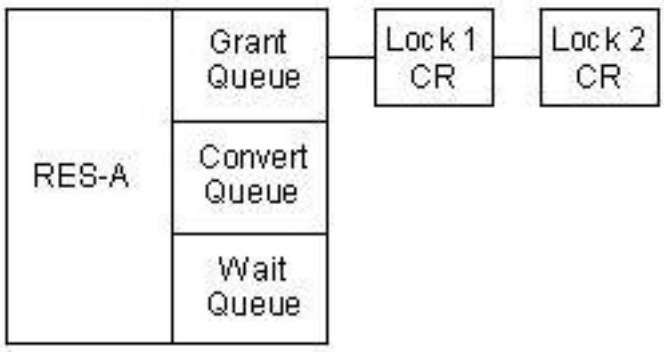
2.3.5. Granted

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 cut 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.



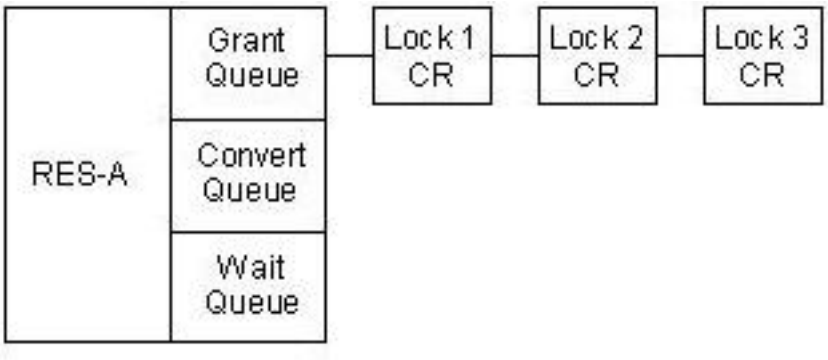
2.3.6. Leaving the Grant Queue

A lock can leave the grant queue only if the owner makes a request to release it or if the process holding the lock terminates. The lock manager releases all locks owned by the process that terminates. By using flags to the lock open routines, however, you can specify that you want the locks you create on a lock resource to remain after your process terminates. These locks are called orphan locks. If the orphan lock is on the grant queue, the lock manager leaves it there. If the orphan lock is on the convert queue, the lock manager puts it back on the grant queue at its old grant mode (its conversion request is canceled). If the orphaned lock is on the wait queue, the lock manager ignores its orphan-able state and removes it. For more information about creating orphan locks, see Section 3.3.9.

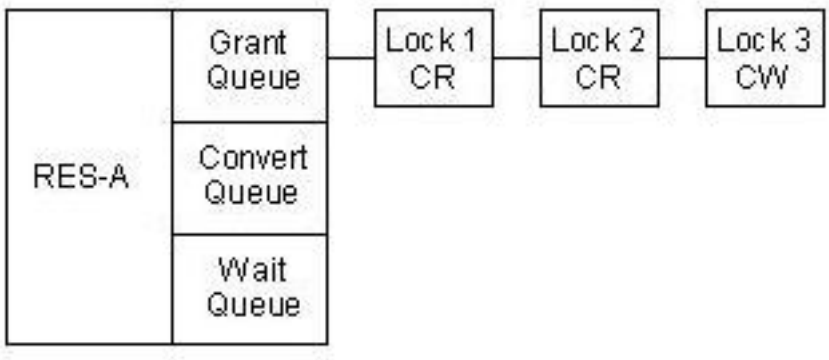
2.3.7. Converting

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.

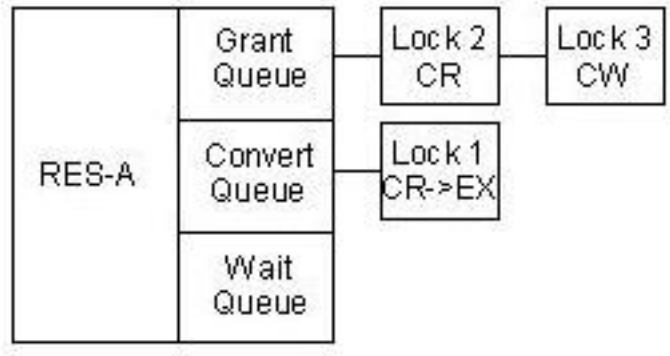


A lock conversion request that cannot be granted transitions into a converting state. The lock manager moves locks that are in converting state from the grant queue to the end of the convert queue. Locks that are in the converting state retain the lock mode they held in the grant queue.

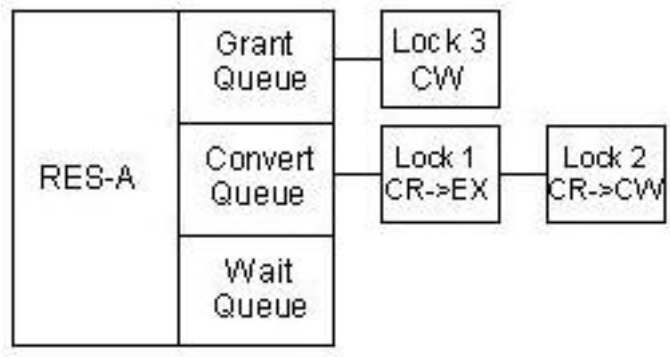
For example, using the previous lock scenario, if you try to convert Lock 1 from CR mode to the more restrictive EX

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:

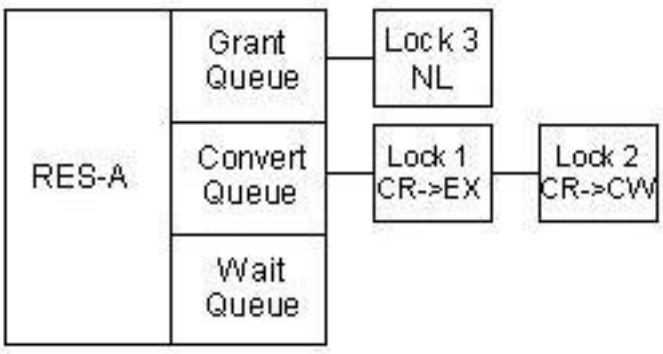
- The process that requested the lock terminates.
- The process that holds the lock cancels the conversion request. When a conversion request is canceled, the lock manager moves the lock back to the grant queue at its previously granted mode.

- The requested mode becomes compatible with the most restrictive granted lock, and all previously requested conversions have been granted or canceled.

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.

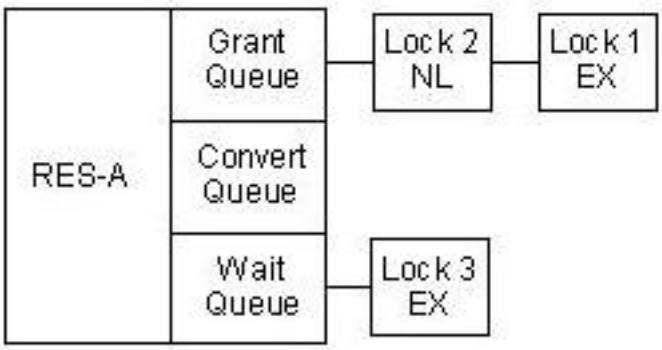
The previous example illustrates conversion deadlock. Even after the down-conversion of Lock 3 to NL mode, Lock 1 cannot be granted because it is blocked by Lock 2, also on the convert queue. Lock 1 cannot convert to EX mode because Lock 2 is still granted at CR mode, which is incompatible with EX mode.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

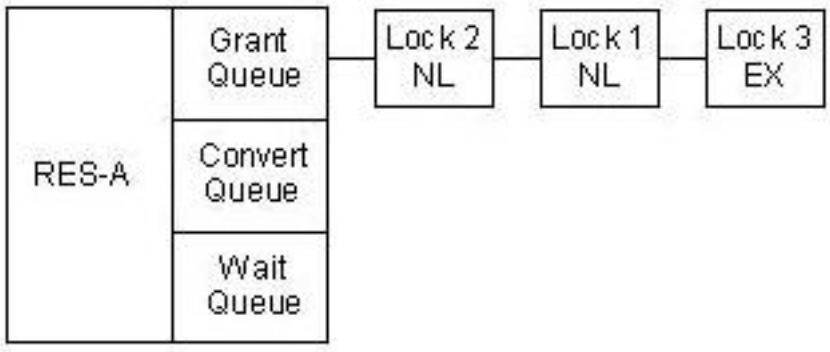


A lock can leave the wait queue if any of the following conditions are met:

- The process that requested the lock terminates.
- The requester cancels the blocked lock. When a blocked lock is canceled, 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.

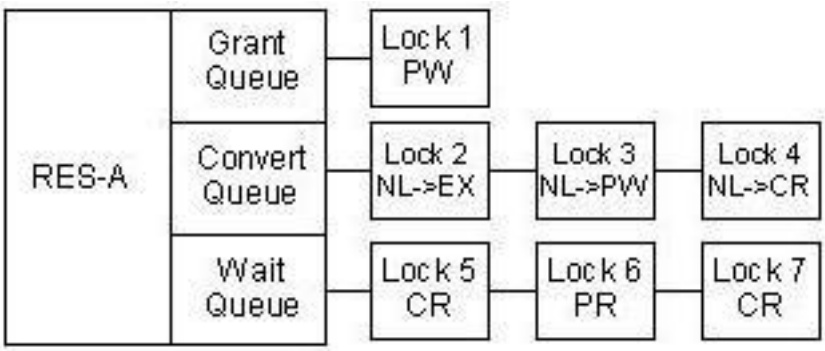
Thus, a lock request can become blocked only as the result of a lock request, but it can unblock as a result of the release or conversion of some other lock. (An exception is made in the case of deadlock. See Section 2.4.)

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.



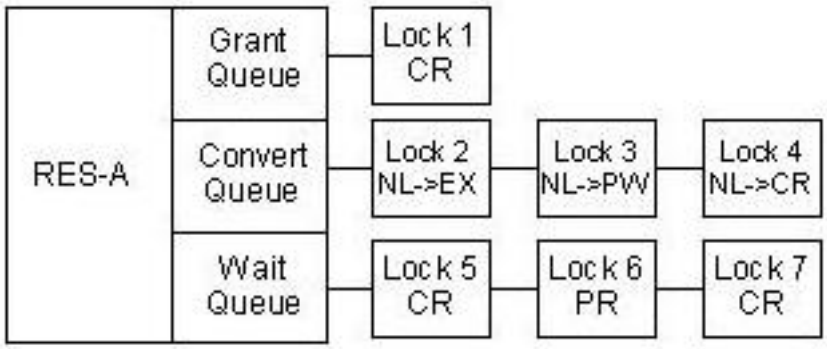
2.3.9. Interaction of Queues

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.

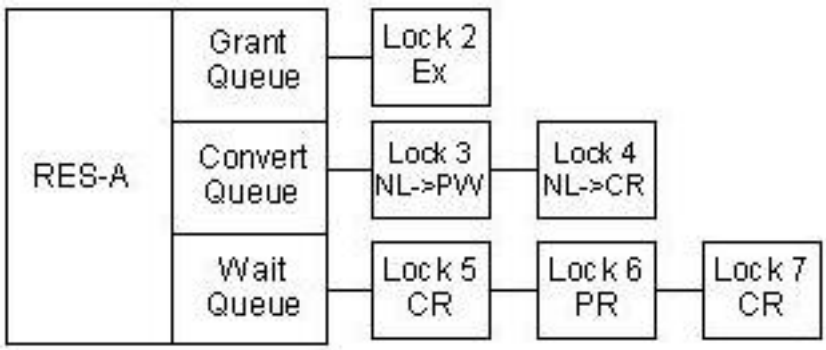


If you request a down-conversion of Lock 1 from PW to CR, the lock manager can grant the conversion request because a CR lock is compatible with the mode of the most restrictive currently granted lock. (Lock 1 itself is the most restrictive currently granted lock; a lock cannot block itself.) Note that 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.



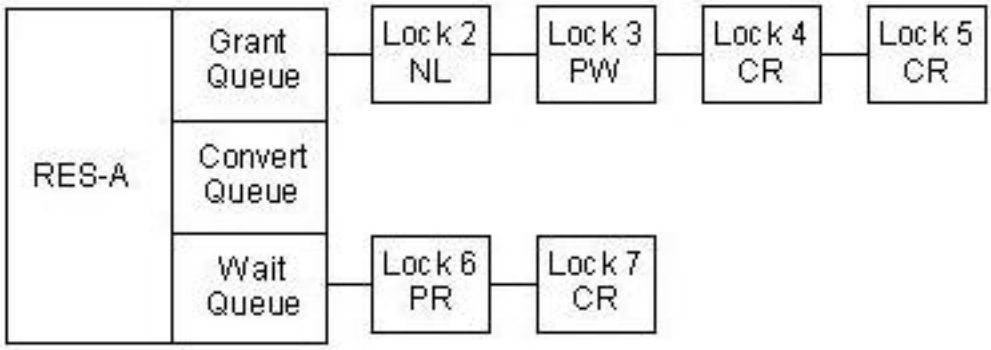
If you release Lock 1, the lock manager can grant Lock 2, the EX lock waiting at the head of the conversion queue. Because the mode requested by Lock 3 is not compatible with an EX mode lock, no other request on the convert queue can be granted. The following figure illustrates the lock resource's queues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



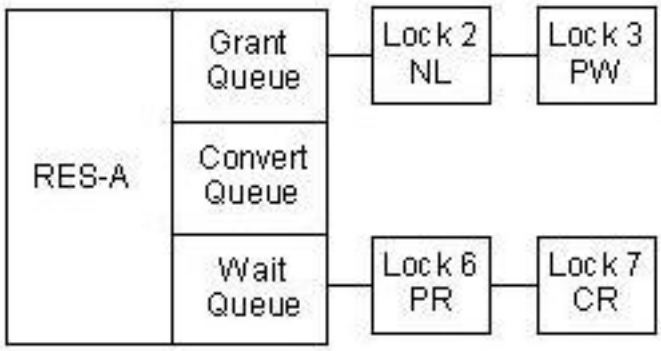
If you request a down-conversion of Lock 2 from EX to NL, the lock manager grants the conversion in-place because a NL lock is compatible with the mode of the most restrictive currently granted lock (EX). The lock manager checks the convert queue to see if the change allows any blocked conversion requests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

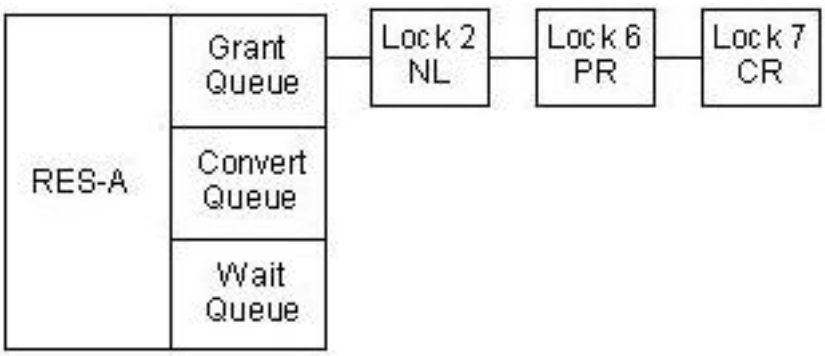
The following figure illustrates the lock resource's queues after the conversion operation of Lock 2.



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.

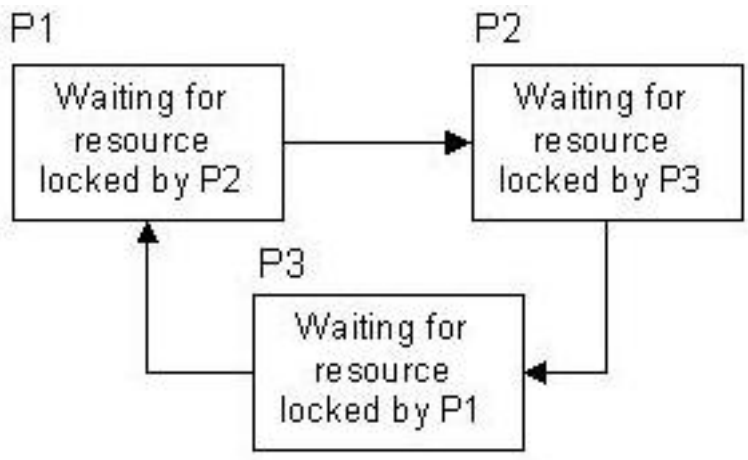


2.4. Deadlock

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.

2.4.5. Transaction IDs

By canceling 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.

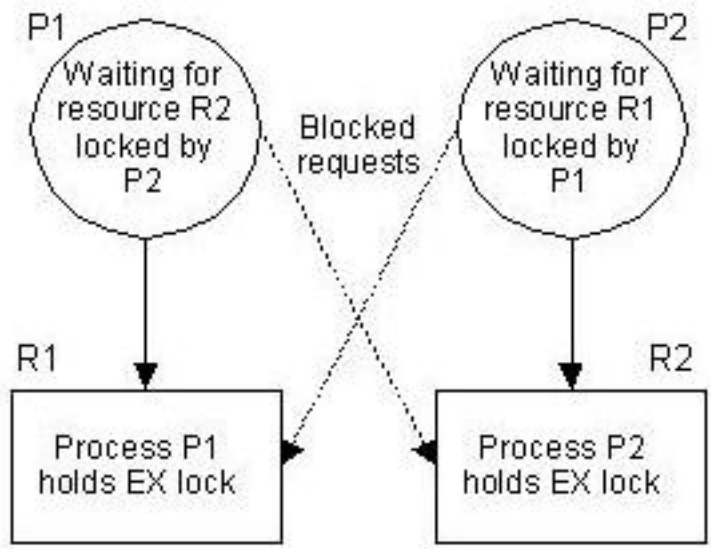
When determining whether a deadlock cycle exists, the lock manager normally assumes the process that created the lock owns the lock. By specifying a transaction ID (also called an XID or deadlock ID) as part of a lock request, a lock client can attribute ownership of a lock related to a particular task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

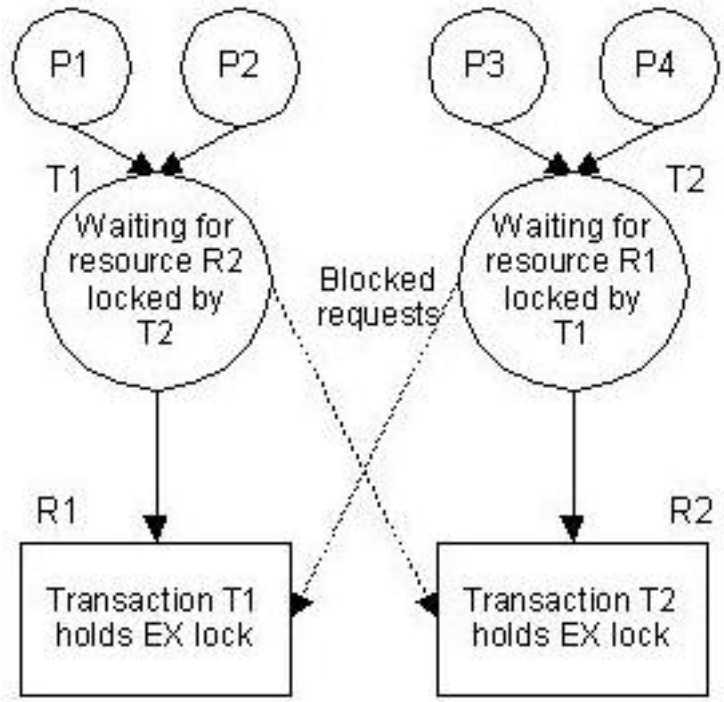
Consider the following example: Process P1 holds an exclusive lock on Resource R1 and requests an exclusive lock on Resource R2. Process P1 will not release the lock on Resource R1 until the lock manager grants the lock on Resource R2. Process P2, meanwhile, holds an exclusive lock on Resource R2 and 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 canceling 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.

2.4.6. Lock Groups

A lock group joins related lock client processes into a single entity. A lock client may create a new lock group or join an existing group. A lock client may belong to, at most, one lock group. Once a client belongs to a group, the group owns all subsequent locks created by that process. Therefore, any process in the group may manipulate group-owned locks.

Alternatively, a process belonging to a lock group can pass the LKM_PROC_OWNED flag to a lock open routine to indicate that this lock is owned by the process, not by the group. Other processes belonging to the group may not manipulate this lock.

The lock manager does not purge a lock owned by a group until all processes belonging to the group have exited or all processes have detached from the group.

A lock group may not span cluster nodes. The lock manager only acknowledges a group ID on the node on which it was created. Therefore, a lock client on one node cannot join a group that was created on a different node.

A process that has left a group can no longer manipulate locks owned by that group, including locks it created while it belonged to the group. If a process is the last group member to leave a group, the locks owned by the group are purged and the group no longer exists. A process is implicitly removed from a group when it terminates.

Lock groups affect deadlock detection in the same way as transaction IDs. Locks requested by a group member without specifying a transaction ID are owned by the group. In this type of situation, the group is the owning entity when determining if deadlock exists.

> **Note:** Since group deadlock can occur more frequently than transaction ID deadlock, you should use transaction IDs when using lock groups. Transaction IDs override group or process ownership.

Chapter 3. Using DLM Locking Model API Routines

This chapter describes how to use the DLM locking model API routines in a distributed Linux application. Chapter 7 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 `dlmlock` or `dlmlockx` 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 `ASTpoll` 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 <dlm/dlm.h>
```

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

If your application uses the dlm_scnop routine, you must also include the following include directive:

```
de <cluster/scn.h>
```

If you are compiling a kernel-space DLM client, you need to compile your source against the kernel include files. You need to at least include the following compile-time flags as part of your build to ensure that you properly reference the DLM include files:

```
-D__linux__ -DGNU_SOURCE -D__KERNEL__ -DMODULE
```

> **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 clients and a loadable kernel module that supports kernel-space DLM clients.

3.2.2.1. Linking User-space Clients to the DLM

Specify the following libraries when you invoke the linkage editor for a single-threaded application:

```
-ldlm -lclstr
```

The `libdlm.a` library contains the routines that support the Distributed Lock Manager. The `libclstr.a` library contains the routines that support the Cluster Manager.

3.2.2.2. Linking Kernel-space Clients to the DLM

Once the DLM has been loaded and built on your system (for instructions for doing this, see Chapter 6), the module `libdlmk.o` will be placed in the directory **/lib/modules/<kernel_level>/dlm** where **<kernel_level>** is the level of the kernel against which the code was built (for example, 2.4.5) and is accessible using the **uname -r** command.

You will need to use the **depmod** command against your module to establish its dependencies against the DLM kernel API module. When you load your module, the DLM kernel API module will also be loaded.

Note:

- The core DLM must be loaded and started prior to running the **depmod** command.
- The DLM currently does NOT support kernel-space callers that are built into the kernel. Kernel-space users of the DLM must themselves be loadable modules.

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 following:

- The length of time you want to wait for a blocked request to be granted (timeout value)
- The value of the lock value block associated with a lock resource

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/dlm/dlm.h` include file, has the following structure:

```
struct lockstatus {
      dlm_stats_t       status;
      int               lockid;
      char              value[MAXLOCKVAL];
      unsigned int      timeout;
};
```

The following list describes each field:

status

Contains the status code returned by the lock manager. The status codes are defined in the `/usr/include/dlm/dlm.h` include file.

lockid

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

value

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 MAXLOCKVAL which is defined in the `/usr/include/dlm/dlm.h` header file as 16 bytes.

timeout

Specifies the amount of time the application allows for a lock request to be granted. This value is only used if the LKM_TIMEOUT flag is also set in the request.

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 lock open 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:

`dlmlock`

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

`dlmlockx`

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 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 LKM_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 ASTpoll 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 ASTpoll routine.
- A pointer to arguments you want passed to either the AST routine or the blocking AST routine.
- For calls to the `dlmlockx` routine, a pointer to an eight-byte transaction ID that indicates the lock is owned by a transaction or group.

The following example uses the `dlmlock` routine to request a CR mode lock on a lock resource named RES-A.

```
#include <dlm/dlm.h>

dlm_stats_t status;
struct lockstatus lksb;        /* lock status block */
extern void ast_func();
.
.
.
status = dlmlock( LKM_CRMODE,   /* mode  */
                      &lksb,   /* addr of lock status block */
                  LKM_VALBLK,   /* flags  */
                     "RES-A",   /* name  */
                           5,   /* namelen  */
                    ast_func,   /* ast routine triggered */
                           0,   /* astargs */
                           0);  /* bast  */
if ( status != DLM_NORMAL )
{
      dlm_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 **lockid** field of the lock status block. (You

specify the address of the lock status block as an argument to the `dlmlock` and `dlmlockx` 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 7 for a complete list of all possible status codes returned by the `dlmlock` and `dlmlockx` routines.

3.3.1.1. Obtaining the Status of a Lock Request Synchronously

Using the asynchronous lock routines, lock clients can request a synchronous return of your lock request by specifying the LKM_SYNCSTS flag. Kernel-space clients can obtain synchronous lock returns only using this method, they cannot call the purely synchronous interfaces (e.g., dlmlock_sync). When this flag is specified, the lock manager returns status synchronously if the following conditions are satisfied:

- The request can be granted immediately; that is, it is not blocked by the mode of an existing lock.
- The master copy of the lock resource resides on the same node as the requesting process.

When the lock manager returns synchronously, the `dlmlock` or `dlmlockx` routines return the status code DLM_SYNC, indicating success, instead of the DLM_NORMAL status code and the lock manager does not queue an AST routine for execution.

If the lock manager cannot grant the request immediately or if the lock resource is not mastered on the same node as the requesting process, the lock manager returns the DLM_NORMAL status code and returns the status of the lock request asynchronously.

> **Note:** To guarantee that you obtain a synchronous return from your lock request as a user-space client, use the `dlmlock_sync` or `dlmlockx_sync` routines, described in the next section.

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. Only user-space clients can call the synchronous lock routines. 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:

`dlmlock_sync`

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

`dlmlockx_sync`

Requests a lock and waits for a return or converts an existing lock to a different mode and specifies a transaction ID for that lock.

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 LKM_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 `ASTpoll` routine.
- A pointer to arguments you want passed to the blocking AST routine.
- For calls to the `dlmlockx_sync` routine, a pointer to an eight-byte transaction ID that indicates the lock is owned by a transaction or group.

3.3.3. Triggering AST Routines

This section describes AST handling for user-space and kernel space clients.

3.3.3.1. AST Handling for User-space Clients

To trigger the execution of AST routines (both regular and blocking), your application must call the ASTpoll routine. The lock manager can send a signal to your application when it has AST routines queued for execution. To use this signal mechanism, your application must:

- Use the `dlm_setnotify` routine to specify the signal you want the lock manager to use to notify your application. Use SIGUSR1 and SIGUSR2. Other signals can be used, but this can interfere with their normal use by the system to signal exceptional conditions to the application.
- Create a routine in your application that will handle the signal when your application receives it. Typically, applications call the `ASTpoll` routine from within this signal handling routine. Use the `signal` routine or the

`sigaction` routine to associate the execution of this routine with the reception of the signal. For more information about using the `signal` or `sigaction` routines, see their man page. Note that you must set up the signal handling routine before each call to the `dlmlock` routine.

For each lock or conversion request granted by the lock manager, only one blocking AST will be sent to the process that owns the lock in situations where the lock is blocking another request. It is expected that if a client specifies a blocking AST function for a lock, the client will take some action in response to the blocking AST. An expected response would be to either convert or unlock the lock. The lock manager will not send another blocking AST for this lock until after the client that owns the lock has taken one of these actions.

For an example of how to use these routines in an application, see Section 3.3.5.

3.3.3.2. AST Handling for Kernel-space Clients

Unlike user-space clients, the DLM can directly execute callback routines when asynchronous lock requests are satisfied (AST delivery), or in the case a kernel-space client is holding that blocks another lock request (BAST delivery). This difference means that the kernel-space clients do not go through the same procedure as the user-space clients.

To establish the AST/BAST handling callback routines, a kernel space client continues to provide routine pointers to routines that will handle the AST/BAST notifications on the appropriate lock request calls (for example, `dlmlock`). However, kernel space clients do not call `setnotify` and do not establish a signal handler to deal with signals from the DLM. They can, however, setup a signal handler for their own purposes.

When a kernel-space client's lock request is satisfied, whether successfully (lock granted) or unsuccessfully (for example, request timed out) the client will be notified via an AST, as with user-space clients. the DLM will directly execute the AST handler specified on the lock request, passing in the appropriate data.

This will be done under the following conditions:

- The AST routine will be executed using an arbitrary kernel thread, but this thread will not be the client's thread that submitted the lock request.
- There is no guarantee that the client's thread submitting the lock request will return to the client's code before the AST routine is executed.
- The client's AST handler should attempt to complete its work as quickly as possible, it should not block or wait on any events. If it needs to do so, it should queue work in a fashion appropriate to the client and the client's code should deal complete its work on its own thread(s).
- If the client is submitting multiple simultaneous lock requests, it is arbitrary in what order the DLM will handle them and return status.

Similarly, for BAST handling, the DLM will directly execute at any time the BAST routine specified for a kernel-space client's lock. Since a BAST is executed only when the client already holds the lock, a BAST routine

will always be executed after the AST routine for that lock. As with the AST handler, the BAST handler should not block or wait, and should queue work if necessary for the client to handle on its own threads.

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 parameter to the `dlmlock` and `dlmlockx` routines or the bastarg parameter to the `dlmlock_sync` and `dlmlockx_sync 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 `dlmlock` 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 the signal handling mechanism to obtain the status of the request. The example also illustrates how to use the astarg parameter to track lock requests.

```
#include <dlm/dlm.h>
#include <stdio.h>
#include <signal.h>
pid_t getpid(); /* needed for ASTpoll routine */
struct lockstatus lksb[12]; /* array of lock status blocks */
int which_lock;  /* index into array of lock status blocks */
void  ast_func();  /* AST routine  */
void  sig_func();  /* signal handling routine */
dlm_stats_t status;
int stat;
char *msg; /* for printable status code */
main(argc, argv)
        int argc;
        char *argv[];
{
        int astarg = 0;  /* astarg parameter */
        status = dlm_setnotify( SIGUSR1, NULL );
        if( status != DLM_NORMAL )
        {
```

```
                dlm_perror("dlm_setnotify");
        }
        stat = signal( SIGUSR1, sig_func );
        if( stat != 0 )
        {
                perror("signal");
        }
        which_lock = 0;
        astarg = which_lock;

        status = dlmlock( LKM_CRMODE,     /* mode  */
                     &lksb[which_lock],   /* lock status block */
                      LKM_VALBLK,
                           "RES-A",       /* name  */
                                   5,     /* namelen  */

                           ast_func,      /* ast routine */
                             &astarg,     /* astargs */
                                   0);    /* bast  */
        if ( status != DLM_NORMAL )
        {
                dlm_perror( "dlmlock");
        }
}
/* Signal handling routine; calls ASTpoll to trigger AST routine  */

void sig_func()
{
        ASTpoll( getpid(), 0 );
}

/* Routine that is triggered by ASTpoll.  */

void ast_func(astarg)
int *astarg;
{
        msg = dlm_errmsg( lksb[*astarg].status );
        printf("status= %s; astarg passed = %d",msg,*astarg);
}
```

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 LKM_NOQUEUE flag as an argument to the lock routine.

If your lock request cannot be granted immediately, the lock open routine returns the status DLM_NORMAL and the AST is queued with the status DLM_NOTQUEUED 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 canceled, 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** field of the lock status block that is passed as an argument to the lock open routine.

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

```
#include <dlm/dlm.h>

dlm_stats_t status;
struct lockstatus lksb; /* lock status block */
extern void ast_func();

lksb.timeout = 500; /* 5 seconds */

status = dlmlock( LKM_CRMODE,   /* mode  */
                      &lksb,   /* lock status block */
                 LKM_TIMEOUT,   /* flags  */
                     "RES-A",   /* name  */
                           5,   /* namelen  */
                    ast_func,   /* routine to trigger ast */
                           0,   /* astargs */
                           0);  /* bast  */
if ( status != DLM_NORMAL )
{
        dlm_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 LKM_NODLCKWT flag with the lock open 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 LKM_ORPHAN flag set. Locks originally requested with the LKM_ORPHAN 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 LKM_ORPHAN flag is set (whether by the initial lock request or by a subsequent conversion), that flag remains set for the duration of that lock.

3.3.10. Requesting Local Locks

Lock clients can achieve enhanced locking performance when obtaining short-lived locks against equally short-lived lock resources by specifying the LKM_LOCAL flag with the lock open routine. This flag directs the lock manager to skip the lock resource directory look up it would normally perform as part of lock request processing and master the lock resource on the local node. For standard lock requests, the lock manager checks its lock resource directory to find out on which node the lock resource is mastered. Because the lock resource directory is spread among all cluster nodes, the directory look up step may require communication with a remote node. By bypassing the directory look up, the lock manager reduces the network overhead associated with the lock request, improving performance.

3.3.11. Guidelines for Use

Use caution when creating local lock resources. While eliminating the lock resource directory look up can improve performance, it allows applications to create multiple masters of a lock resource. Lock resources created using the LKM_LOCAL flag are not included in the lock manager's lock resource directory but exist in the same namespace as global lock resources. Duplicate lock resource masters can compromise the integrity of the locking scheme and can cause data corruption.

To use local lock resources effectively and safely, make sure that the lock resource you are creating does not already exist in the cluster. If you are certain that the lock resource you want to master locally is unique, then acquire the local lock, accomplish the task, and release the lock as quickly as possible. If the lock is held briefly, it is unlikely that another client will need to lock the same resource. If contention is likely, do not use local locks.

3.3.12. Acquiring Additional Locks on a Local Lock Resource

If your application must acquire additional locks on a local lock resource, specify the LKM_FINDLOCAL flag with the lock open routine when requesting these locks. When this flag is specified, the lock manager queries each node to determine on which node the lock resource is mastered. The lock manager does not check its lock resource directory because local lock resources will not have an entry.

If the lock manager finds the lock resource, it processes the lock request, adding the lock to one of the lock resource's queues, depending on mode compatibility. If the lock manager does not find the lock resource, it creates a local lock resource on the initiating node, granting the lock.

Because lock requests that use the LKM_FINDLOCAL flag require a query to be processed by all active nodes in the cluster, they take longer to process than requests using the LKM_LOCAL flag or even normal lock requests. You should only use the LKM_FINDLOCAL flag to obtain a lock against a lock resource that you know was created using the LKM_LOCAL flag. The lock manager processes the request against the lock resource whether it's global or local; local lock resources and global lock resources share the same namespace. However, the cost of the extra overhead incurred by using the LKM_FINDLOCAL flag is wasted when the lock resource is global.

> **Note:** Use the LKM_FINDLOCAL flag with caution. Even though the lock manager checks all cluster nodes for the local lock before creating a new lock resource, the potential still exists for creating duplicate lock resource masters. For example, if two lock clients running on different nodes initiate LKM_FINDLOCAL lock requests simultaneously, their searches for the local lock resource may both complete without finding the lock resource because of timing considerations. Then each node may proceed to create local masters of the same lock resource.

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 `dlmunlock_sync` 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 LKM_CANCEL flag.
> - If you want to modify the lock value block, specify the LKM_VALBLK 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. In addition, the lock manager sets the status in the lock status block from the original conversion request to DLM_CANCEL and queues for execution the AST routine associated with the request.

Wait queue

The lock manager removes the lock from the wait queue. In addition, the lock manager sets the status in the lock status block from the original request to DLM_ABORT and queues the AST routine associated with the lock for execution.

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 <dlm/dlm.h>

dlm_stats_t status;

struct lockstatus lksb[MAXLOCKS]; /* lock status block */
int index=0;
.
.
.
status = dlmunlock_sync(lksb[index].lockid, 0, 0);
if (status != DLM_NORMAL)
{
        dlm_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 orphaned locks.

3.6. Manipulating the Lock Value Block

Every lock resource includes a fixed number of bytes of storage, called a lock value block (LVB), that applications can use to store data. The DLM can be built in one of two modes:

- Strict VMS compatibility, where the LVB is 16 bytes
- Default, where the LVB is 32 bytes

Within a cluster all DLM instances must be the same. Results are undefined, but definitely not good, if you mix them within a cluster.

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 `dlmunlock` or the `dlmunlock_sync` 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 LKM_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 <dlm/dlm.h>

struct lockstatus lksb;

char valblk[16];

strcpy(valblk,"my lvb");

status = dlmunlock_sync( lksb.lockid,   /* mode  */
                        &valblk,   /* lock value block */
                     LKM_VALBLK);  /* flags  */

if ( status != DLM_NORMAL )
{
         dlm_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 LKM_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 <dlm/dlm.h>

struct lockstatus lksb;

char valblk[16];

strcpy(valblk, "my lvb");

lksb.valblk = &valblk;

status = dlmlock( LKM_CRMODE,   /* mode  */
                       &lksb,   /* lock status block */
    LKM_CONVERT | LKM_VALBLK,   /* flags  */
```

```
                      "RES-A",   /* name  */
                            5,   /* namelen  */
                     ast_func,   /* routine to trigger ast */
                            0,   /* astargs */
                            0 );
if ( status != DLM_NORMAL )
{
         dlm_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 LKM_INVVALBLK flag when releasing a lock using the dlmunlock 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.

```
#include <dlm/dlm.h>

struct lockstatus lksb;


status = dlmlock( LKM_CRMODE,   /* mode  */
                       &lksb,   /* lock status block */
 LKM_CONVERT | LKM_INVVALBLK,   /* flags  */
                      "RES-A",   /* name  */
                            5,   /* namelen  */
                     ast_func,   /* routine to trigger ast */
                            0,   /* astargs */
                            0 );
if ( status != DLM_NORMAL )
{
         dlm_perror( "dlmlock");
```

```
}
```

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.

3.6.4.2. Implementing Cluster-Global Counters

One specialized use of lock value blocks is to implement a cluster-global counter, called a System Commit Number (SCN). Databases can use the SCN to provide unique identifying numbers to database transactions; these numbers help track database transactions. To facilitate the implementation of such a counter, the lock manager includes a routine, called the dlm_scnop routine, that allows you to manipulate the LVB associated with a lock resource directly.

Using the standard DLM locking model interface, you would need two separate lock operations to manipulate an SCN: one operation to acquire an exclusive lock on the lock resource and another lock request to modify the LVB (by down-converting the lock to a less restrictive mode). Using the `dlm_scnop` routine, you can modify the value of the SCN without making any calls to the lock routines, avoiding the overhead incurred by a lock request. (You must make one call to the one of the lock open routines to acquire a NL lock on the lock resource that stores the SCN. You can use any lock resource to store the SCN.)

> **Note:** Do not use the `dlm_scnop` routine to modify the LVB associated with lock resources other than the lock resource used to store the SCN. Bypassing the standard lock interface could compromise the integrity of your application's locking scheme.

As with the LVB associated with any lock resource, the SCN is marked invalid if a node fails. If the `dlm_scnop` routine retrieves or attempts to change the value of an SCN marked invalid, it returns the status DLM_VALNOTVALID. To reset an invalidated SCN, call the `dlm_scnop` routine specifying the `SCN_SET` operation.

3.7. Handling Returned Status Codes

The global variable **dlm_errno** is declared as the enumerated type dlm_stats_t, which is defined in the /usr/include/dlm/dlm.h include file. Specify it in your application as follows:

```
dlm_stats_t dlm_errno;
```

The enumerated type dlm_stats_t is made up of all the status codes returned by the lock manager API routines, both the DLM locking model and UNIX locking model routines. As with the standard Linux global variable `errno`, the value of `dlm_errno` is set by the last lock operation.

To facilitate the printing of error status messages, the DLM API includes the following routines:

`dlm_perror`

> Writes a message you specify to standard error. Appended to the message is the status code returned by the last DLM API routine to execute. The `dlm_perror` routine obtains the value of the status code from the global variable `dlm_errno`.

`dlm_errmsg`

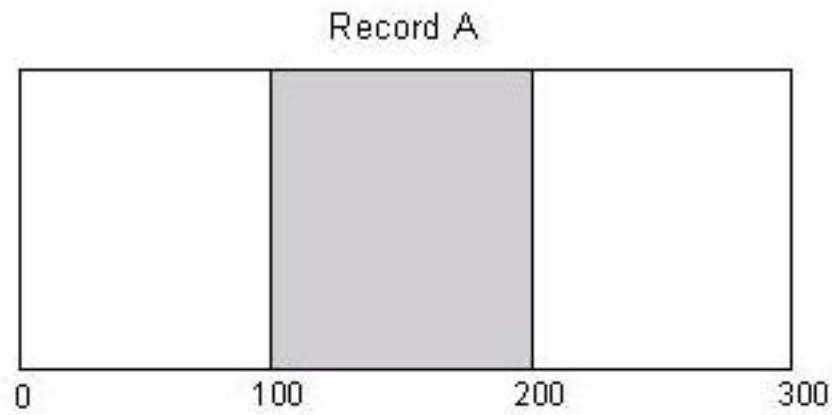
> Returns a pointer to a printable version of the DLM API status code. The status codes that make up the `dlm_stats_t` enumerated type are constants, not printable character strings. This routine is useful for applications that format their own status return messages (instead of using the `dlm_perror` routine).

Chapter 4. UNIX Locking Model

This chapter presents the concepts you need to understand how to use UNIX locks effectively in an application. Chapter 5 describes how to use the UNIX locking model API routines to implement locking in an application.

4.1. Overview

UNIX System V locks support the concept of lock regions. An application first registers, or creates, a lock resource with the lock manager by giving the lock resource a name. Then, when it wants to lock this lock resource, the application specifies a range of locations that should be locked and links this region to the lock resource name. For example, an application could create a resource called "Record-A" and then lock locations 100 through 200 in Record A. The following figure shows a UNIX locking region.



The lock manager does not maintain distinct lock objects. Rather, it keeps a database of which regions of the resource are locked. The lock manager does not keep locks separate. Instead, it coalesces overlapping locks of the same mode. If an application has an exclusive lock on a region from 0 to 10 and then obtains another exclusive lock on the region from 11 to 20, do not assume that two locks exist. Rather, consider the region from 0 to 20 locked.

Likewise, a request to unlock range 0 to 100 unlocks all the regions within those bounds that are currently locked by the client requesting the unlock.

For example, assume that a client has two locks. The first is a shared lock from 0 to 25 and the second is a shared lock from 50 to 75. A request to unlock the region from 0 to 75 would release both locks.

4.2. Lock Modes

A lock mode indicates whether a process wants to share access to a region with other processes or whether it wants to prevent other processes from accessing that region while it holds the lock. A lock request always specifies a lock mode as part of that request. A UNIX lock can either be shared or exclusive.

4.2.1. Shared

A shared lock is the traditional read lock. Multiple applications can simultaneously request shared locks on the same region.

4.2.2. Exclusive

An exclusive lock is the traditional write lock. If an application wants to prevent any other application from accessing a lock resource, it can request an exclusive lock. Only one application at a time can possess an exclusive lock on a region.

A request for an exclusive lock blocks if another application has a current lock on the specified region.

Once the lock manager grants an exclusive lock, all successive lock requests on that region fail or block until the exclusive lock is released.

4.3. Lock States

A lock state indicates the current status of a lock request. A UNIX lock request is either GRANTED or BLOCKED.

4.3.1. GRANTED

An application has acquired a lock on the desired region at the desired lock mode.

4.3.2. BLOCKED

An application is unable to acquire a lock on the requested region at the requested mode, because a conflicting lock is currently granted on that region. A blocked lock cannot be granted until the conflicting lock is released or downgraded to a compatible mode. For example, an exclusive lock blocks all other lock requests. A shared lock does not block a request for a shared lock but does block a request for an exclusive lock.

A client's own locks are transparent in that the locks the client has previously requested will not block the client's current request. Instead, the old locks are discarded. When a client requests a lock, the lock manager releases any existing locks held by that client that are overlaid by the new request, regardless of the mode of those locks.

For example, assume that a client has an exclusive lock on a region from 50 to 75. That same client requests an exclusive lock on the region from 0 to 100. The lock manager releases the lock on region 50 to 75, and grants the lock on region 0 to 100. Had a different client requested the lock on 0 to 100, that request would have been blocked, waiting for the exclusive lock on 50 to 75 to be released.

Chapter 5. Using UNIX Locking Model API Routines

This chapter describes how to use UNIX locking model API routines in a distributed Linux application. Chapter 7 provides reference information on the routines discussed in this chapter.

5.1. Overview

Use the UNIX locking services by making requests from an application. You can:

- Register (create) lock resources
- Acquire locks on the lock resources you create
- Release locks held on a lock resource
- Handle returned status codes
- Purge all the locks held by a particular client, if necessary

To perform these tasks, applications use the routines in the UNIX locking model API to make requests to the lock manager. For example, to request a lock on a lock resource, an application would use the `dlmregionlock` routine.

The following sections describe how to perform these primary locking tasks, including any ancillary tasks that may be required.

5.2. Prerequisites

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

5.2.1. Header Files

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

```
#include <cluster/dlm.h>
```

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

To use the `dlmregionlock` routine, you must also include the following system include file:

```
#include <sys/file.h>
```

5.2.2. Library Files

The distributed Linux software includes separate libraries for multi-threaded and for single-threaded applications. Be sure to link with the appropriate library for your application.

5.2.2.1. Single-threaded Applications

Specify the following libraries when you invoke the linkage editor for a single-threaded application:

```
-ldlm -lclstr
```

The `libdlm.a` library contains the routines that support the Distributed Lock Manager. The `libclstr.a` library contains the routines that support the Cluster Manager.

If your application uses the services of the Cluster Information Program (Clinic), you must also include the `libcl.a library`.

5.2.2.2. Multi-threaded Applications

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

```
-ldlm_r -lclstr_r
```

The `libdlm_r.a` library contains the routines that support the Distributed Lock Manager. The `libclstr_r.a` library contains the routines that support the Cluster Manager.

5.2.3. Data Structure

The UNIX locking model API uses a data structure, called the lock resource handle, defined in the `/usr/include/cluster/dlm.h` include file.

The `dlmregister` routine returns a resource handle to the application. A resource handle is a union data type that has the following format:

```
union dlm_rh {
      unsigned long rh;
      struct {
      unsigned char site;
      unsigned char type;
      unsigned short cookie;
      } handle;
};
```

5.3. Registering a Lock Resource

A lock resource is a range of locations you can lock. A lock resource can represent any entity, such as a file, a data structure, a database, or an executable routine. In fact, a lock resource is nothing more than a name. The name does not have to correspond to an actual object.

5.3.1. `dlmregister` Routine

Before locking a lock resource, you must first register, or create, that lock resource. You register a lock resource by calling the `dlmregister` routine with the name of the lock resource you want to lock. The lock resource name is a null-terminated string of no more than 255 ASCII characters.

5.3.1.1. Resource Handles

The `dlmregister` routine returns a lock resource handle to the calling routine. A lock resource handle is a 32-bit integer that describes a lock resource. The lock manager uses lock resource handles to efficiently look up lock resources.

You must use this resource handle in any subsequent lock and unlock requests that refer to that lock resource.

5.4. Locking a Lock Resource

Use the `dlmregionlock` routine to lock a lock resource region. Supply the following information with the `dlmregionlock` routine:

- The resource handle returned from an earlier call to `dlmregister` that registered the resource

- The lower bound of the region you want to lock
- A length that indicates the extent of that region
- A flag that indicates the type of lock: either shared or exclusive

If there are currently no locks on the lock resource or if the requested mode is compatible with the modes of the current locks, the lock manager grants the lock and returns immediately with a status of DLM_NORMAL.

If the requested mode is incompatible with the mode of a current lock, the lock manager marks the request as blocked and does not return until the lock is granted. Using a flag to the `dlmregionlock` routine, you can mark a lock request, either shared or exclusive, as non-blocking. This indicates that the request should return with an error status if it cannot be granted immediately.

5.5. Unlocking a Resource

Use the `dlmregionlock` routine to unlock a region. Any regions currently locked by the application making the request that overlap the region specified in the unlock request are released.

Supply the following information with the `dlmregionlock` routine:

- The resource handle
- The lower bound of the region you want to unlock
- A length that indicates the extent of that region
- A flag that indicates that this is an unlock request

The lock manager releases the lock and returns with a status of DLM_NORMAL.

5.6. Handling Returned Status Codes

The global variable `dlm_errno` is declared as the enumerated type `dlm_stats_t`, which is defined in the `/usr/include/cluster/dlm.h` include file. Specify it in your application as follows:

```
dlm_stats_t dlm_errno;
```

The enumerated type `dlm_stats_t` is made up of all the status codes returned by the lock manager API routines, both the DLM locking model and UNIX locking model routines. As with the standard Linux global variable `errno`, the value of `dlm_errno` is set by the last lock operation.

To facilitate the printing of error status messages, the UNIX locks API includes the following routines:

`dlm_perror`

Writes a message you specify to standard error. Appended to the message is the status code returned by the last UNIX API routine to execute. The `dlm_perror` routine obtains the value of the status code from the global variable `dlm_errno`.

`dlm_errmsg`

Returns a pointer to a printable version of the UNIX API status code. The status codes that make up the `dlm_stats_t` enumerated type are constants, not printable character strings. This routine is useful for applications that format their own status return messages (instead of using the `dlm_perror` routine).

5.7. Purging Locks

The Distributed Lock Manager `dlm_purge` routine can be used to facilitate releasing UNIX locks. The `dlm_purge` routine releases all locks owned by a particular process, identified by its process ID.

When a client process terminates while holding one or more locks, the lock manager purges any locks held by that client process.

Chapter 6. Configuring the Distributed Lock Manager

This chapter provides a high level description of the process to download, build, and run the DLM. Refer to the earlier chapters and chapter 7 about using the DLM as a client application.

6.1. Downloads

This section describes how to download the Distributed Lock Manager and its supporting applications.

6.1.1. Distributed Lock Manager

Currently, the DLM code is only available as compressed tar archives from the DLM Web site on the IBM developerWorks Open Source Projects Web site at http://oss.software.ibm.com/developerworks/projects/dlm/

You can install the tar archive into any directory. Uncompress and untar it.

6.1.2. Heartbeat

You must have the Heartbeat package (from http://www.linux-ha.org/) installed on your system in order to use or build the DLM. Simply installing the binaries and the libraries is not enough. The header files must also be available. The DLM has successfully been built and tested against versions 0.4.8i and 0.4.8k. Other 0.4.8 and 0.4.9 versions should also work, but have not been tested.

Please refer to the instructions with Heartbeat for more information about installing and running it.

6.2. Building the DLM

You can build the DLM in two different ways: in the "source" tree or in a "build" tree. It is strongly recommended that you build using a build tree, but both procedures are described in this section.

6.2.1. Building in a Source Tree

To build the DLM in a source tree follow this procedure.

1. Type the following command:

   cd dlm/source

   CLUSTMGR=heartbeat UP=up HEARTBEAT_PATH=<path to heartbeat source> make dep

   <path to heartbeat source> should point to the location where you installed the Heartbeat code. It should point to the base directory, not the directory containing the Heartbeat include files. For example:

   /home/wombat/linux/heartbeat/heartbeat-0.4.9/

   This command will build the .depend files in each subdirectory. These files are used by **make** to control when files are re-built. If you are building for SMP, you must omit the "UP=up" statement. Additionally, your kernel sources must be configured for SMP. In the near future the build scripts will be modified, so the DLM can be built against different kernel sources on the same machine.

   > **Note:** If all the header files from Heartbeat were installed (both the ones in the heartbeat and stonith subdirectories) in system include directories (i.e., /usr/include), the "HEARTBEAT_PATH=<path to heartbeat source>" statement can also be omitted.

2. Type the following command:

   UP=up HEARTBEAT_PATH=<path to heartbeat source> make all

   This command will build all of the libraries, daemons, and kernel modules.

6.2.2. Building in a Build Tree

To help keep management of the files cleaner, it is recommended that you build the DLM object files and modules into a build tree separate from the source tree. Building the files here will allow you to have multiple build trees. You can use the multiple trees for SMP and UP versions or for building the DLM against different kernel releases.

To build the DLM in the build tree follow this procedure.

1. Type the following command:

   cd <path to DLM source>/source

src_path=‘pwd‘

2. Change into the top-level source directory, then set the environment variable `src_path` to that path.
3. Type the following command:

   BUILD_DIR=<path to build tree> make build_dir

   This will create the build directory at your desired location.

4. Type the following command:

   cd <path to build tree>

   **SOURCE_PATH=$src_path CLUSTMGR=heartbeat UP=up **

   HEARTBEAT_PATH=<path to heartbeat source> make dep

   **SOURCE_PATH=$src_path CLUSTMGR=heartbeat UP=up **

   HEARTBEAT_PATH=<path to heartbeat source> make all

   As in the previous case, you need to set the heartbeat source path, and then first build dependencies and then the actual object files and modules.

6.2.3. Modules

Once built, you should have a number of object (.o) files and also a number of modules, an executable, and a shared library:

```
<build directory path>/libdlm/libdlm.so
<build directory path>/libdlm/libdlm.so.0
<build directory path>/libdlm/libdlm.so.0.x
<build directory path>/libdlmk/libdlmk.o
<build directory path>/dlmdu/dlmdu
<build directory path>/dlmdk/dlmdk.base.o
<build directory path>/dlmdk/dlmdk.core.o
<build directory path>/dlmcccp/cccp.o
```

6.3. Installing the Distributed Lock Manager

After building the object files, the following command must be executed as root:

make install

This will install all of the kernel modules in the correct place under **/lib/modules**, and it will also execute **depmod** to establish the proper module dependencies.

If the modules are to be installed on a different system, you will need to manually copy `dlmdk/dlmdk.core.o`, `dlmdk/dlmdk.base.o`, `dlmcccp/cccp.o`, and `libdlmk/libdlmk.o` to **/lib/modules/<kernel version>/dlm** on that system. You will also need to manually run **depmod**.

6.4. Configuring and Running the Distributed Lock Manager

Heartbeat must be properly configured on the system where you will configure and run the DLM. See the Heartbeat documentation for information on installing and configuring it. Once Heartbeat is installed and configured, complete the following steps to configure and run the DLM.

1. Create a configuration file for the DLM's user-mode daemon (`dlmdu`). Currently, the DLM has only been tested (on Linux) on a one and two node clusters. Given two machines, cluster1 with an IP address of 192.168.1.1 and cluster2 with an IP address of 192.168.1.2, your configuration file might look like this:

   ```
   NODECOUNT 2
   1 cluster1 192.168.1.1
   1 cluster2 192.168.1.2
   DLMNAME haDLM
   DLMMAJOR 250
   DLMCMGR heartbeat
   DLMADMIN admin 0
   DLMLOCKS locks 1
   ```

2. Name the configuration file `<build directory>/dlmdu/dlmconf.cfg`, and run the following command as root:

   ./dlmdu -C dlmconf.cfg -d ALL

   > **Note:** You can also place **dlmdu** in any directory you wish and provide an absolute path name to the configuration file. You can also name the configuration file anything you want. However, if you change any of these things, remember to specify the proper path and name on the **dlmdu** command line.

You should see output similar to the following:

```
DLMK Major/Minor(0/1/0x1) Time(11:42:55) Date(Mar  8 2001)
ALL debugging enabled
Opening configuration file [dlmconf.cfg]
Opened configuration file.
Linenum: 1  Line: NODECOUNT 2
Linenum: 2  Line: 1 cluster1 192.168.1.1
Linenum: 3  Line: 2 cluster2 192.168.1.2
Linenum: 4  Line: DLMNAME haDLM
Linenum: 5  Line: DLMMAJOR 250
Linenum: 6  Line: DLMCMGR heartbeat
Linenum: 7  Line: DLMADMIN admin 0
Linenum: 8  Line: DLMLOCKS locks 1
Attempt to load DLM module, cmd [modprobe -a -k haDLM haDLM_major_number=250 haDLM_name=haDLM haD
Loaded DLM module!
Opened [/dev/haDLM/admin], filedes [3]
1st thread, pid [2576]
2nd thread, pid/tid [2580/1026]
Created thread, id [1026]
Wrote code/size[0/12], blocks so far [1]
No work to do in handling event queue.
Node have [1] nodeq elements queued
Local node is [cluster1]
compare [cluster1] and [cluster1]
found number/name[1/cluster1/192.168.1.1]
Node have [1] nodeq elements queued
Node have [2] nodeq elements queued
compare [cluster1] and [cluster1]
found number/name[1/cluster1/192.168.1.1]
Wrote code/size[2/12], blocks so far [2]
```

There will also be two new files in `/proc`: `/proc/haDLM` and `/proc/cccp`. Until some clients are written, that is all that will be in this directory.

6.5. The Role of dlmdu

The **dlmdu** executable has the following roles:

- Interface with the cluster manager software (e.g., Heartbeat)

- Receive events, node up/down, and pass these events to the core DLM kernel modules
- Establish the DLM /dev entries
- Load the DLM kernel modules that provide the real DLM function
- Stop the DLM and unload the modules when **dlmdu** is stopped

In the future, a wider set of supported cluster manager software packages that interact with the DLM will be provided. Part of the configuration process is to specify which cluster manager is being used.

6.5.1. Command Line Options for dlmdu

In the future, there will be support for additional command line options that will be used to tune the DLM behavior. These options are not documented right now because they are not currently effective. The currently supported options are:

- **-C <configuration file>**: The path and file name of the configuration file to use.
- **-d <debug options>**: The list of debug options to print into the log file.

> **Note:** This list is not documented because it is recommended that you use "ALL" until you are experienced with the DLM or until the DLM is completely stable. You can see the various options in `source/dlmdu/dlm_cmdline.c:parse_debug()`, but you'll also have to read the code in `source/dlmdk` and `source/dlmcccp` to understand the options.

6.6. Using the DLM

At this point, you should be able to execute your clients against the DLM. Assuming you correctly referenced the appropriate DLM API (user-space or kernel-space) based on your client, you should be able to execute (or load and execute for kernel space) your client.

6.6.1. Sample Clients

As part of the distributed code, we provide a number of sample and test client programs. These are provided as-is, without much documentation except the code, refer to the test/ directory and explore the files there.

Chapter 7. Lock Manager API Routines

This chapter provides reference information on the C language routines used to implement locking in a distributed Linux application.

7.1. Lock Manager Routines

The following list summarizes the lock manager API routines, grouped by locking model. The routines appear in one alphabetical list in this chapter.

7.1.1. DLM Locking Model-specific Routines

`ASTpoll`

Triggers the execution of pending AST routines.

`dlmlock`

Makes an asynchronous (non-blocking) request to acquire or convert a lock on a lock resource.

`dlmlockx`

Makes an asynchronous (non-blocking) request to acquire or convert a lock on a lock resource, and specifies a transaction ID for that lock.

`dlmlock_sync`

Acquires or converts a lock on a lock resource and obtains a synchronous return.

`dlmlockx_sync`

Acquires or converts a lock on a lock resource, specifies a transaction ID for that lock, and obtains a synchronous return.

`dlmunlock`

Makes an asynchronous (non-blocking) request to release a lock on a lock resource or cancel a lock request that is in blocked or converting state.

`dlmunlock_sync`

Releases a lock on a lock resource or cancels a lock request that is in blocked or converting state.

`dlm_scnop`

Manipulates the SCN, a specialized use of the lock value block associated with a lock resource.

`dlm_setnotify`

Specifies which signal the lock manager should use to notify your application of a pending AST.

7.1.2. UNIX Locking Model-specific Routines

`dlmregister`

Registers a lock resource.

`dlmregionlock`

Acquires a lock on a lock resource or releases a lock on a lock resource.

7.1.3. Routines Common to Both Locking Models

`dlm_errmsg`

Returns a pointer to a printable version of the DLM API status code for single-threaded applications.

`dlm_getglobparams`

Obtains the value of the global lock manager parameters.

`dlm_getresparams`

Returns the value of a lock resource’s stickiness attribute.

`dlm_getstats`

Obtains statistics on resource usage.

`dlm_grp_attach`

Attaches a lock client to an existing lock group.

`dlm_grp_create`

Creates a new lock group and associates the lock client process with the group.

`dlm_grp_detach`

Removes a lock client process from an existing lock group.

`dlm_perror`

Writes a message you specify to standard error.

`dlm_purge`

Releases all locks owned by a particular client, identified by its process ID.

`dlm_setglobparams`

Sets the value of the global lock manager parameters, including the evaluation threshold and the decay rate.

`dlm_setresparams`

Sets the value of the lock resource's stickiness attribute.

7.2. Routine Index

The following sections provide reference information about the routines.

7.2.1. `ASTpoll`

7.2.1.1. Syntax

```
int ASTpoll(pid, tid);
int pid;
int tid;
```

7.2.1.2. Description

Use the `ASTpoll` routine to trigger any pending ASTs resulting from the completion of previous `dlmlock` or `dlmlockx` routine requests or the delivery of blocking ASTs.

> **Note:** This routine can be called only by user-space clients of the DLM. Kernel-space clients cannot call it.

7.2.1.3. Parameters

pid

> This argument indicates the process ID of the application having outstanding ASTs or blocking ASTs. This process should be the same process that queued the initial lock requests.

tid

> This argument indicates the thread ID of the thread that queued the ASTs.
>
> **Note:** Thread support is not fully implemented in this version of the lock manager. Therefore, you must always specify zero for the thread ID.

7.2.1.4. Status Codes

The `ASTpoll` routine returns the number of ASTs successfully invoked. `ASTpoll` returns 0 if there is a shared memory error or if there is no client record.

7.2.1.5. Example

For an example of the ASTPoll routine, see Section 3.3.5.

7.2.2. `dlmlock`

7.2.2.1. Syntax

```
dlm_stats_t dlmlock(mode, lksb, flags, name, namelen, ast, astargs, bast);
int mode;
struct lockstatus *lksb;
int flags;
void *name;
unsigned int namelen;
void (*ast)();
void *astargs;
void (*bast)();
```

7.2.2.2. Description

Use the `dlmlock` routine to make an asynchronous (non-blocking) request to acquire or convert a lock on a lock resource. If the lock resource does not exist, the lock manager creates it.

The various lock modes specify different degrees of access to a lock resource. You specify this mode as a part of the request. These lock modes are described in Section 7.2.2.3.

To convert an existing lock to a different mode, you must specify the LKM_CONVERT flag. You can also control other aspects of lock manager behavior by specifying flags as part of your request. For more information about the flags supported, see the listing of flags in Section 7.2.2.3.

The lock manager returns status in two locations: the status value returned by the `dlmlock` routine and the status field of the lock status block. The status value returned by the `dlmlock` routine indicates whether the request was accepted by the lock manager. The DLM_NORMAL status value indicates your request was successfully queued. If your request cannot be queued because of syntax problems or invalid arguments, your request is aborted and the dlmlock routine returns an error status code. See Section 7.2.2.4 for a list of these status values.

A success status from the `dlmlock` routine does not indicate that your request has been granted. The lock manager reports whether your request was granted (or denied, canceled, or aborted) asynchronously by queuing for execution the AST routine you specified as an argument. When the AST routine executes, the lock manager returns the status of the request (whether it was granted, denied, canceled or aborted) in the status field of the lock status block. The DLM_NORMAL status value indicates your request was granted. See Section 7.2.2.5 for a list of other possible status values. (For information about the composition of the lock status block, see Section 3.2.3.)

If your request is queued, the lock manager returns a lock ID, a token that identifies the lock, in the lock status block. Note that this field in the lock status block is valid *before* the asynchronous return reporting on lock status. All subsequent requests concerning the lock, such as a cancellation request, must identify the lock by its lock ID.

You can also specify an additional AST routine, called a blocking AST routine, that the lock manager queues for execution when a lock your application holds on a lock resource is blocking another lock request.

7.2.2.3. Parameters

7.2.2.3.1. mode

The requested lock mode, required for both lock requests and conversion requests. The modes supported are listed below in order of severity, from least restrictive to most restrictive:

LKM_NLMODE

Does not grant the requesting process any access to the resource, but indicates future interest in the resource. This mode acts as a placeholder for later conversion requests.

LKM_CRMODE

Allows the requesting process to read from a resource, and allows other processes simultaneous read or write access to the same resource. This mode allows an unprotected read operation.

LKM_CWMODE

Allows the requesting process to read or write to a resource while other processes simultaneously read or write to the same resource. This mode allows an unprotected write operation.

LKM_PRMODE

Allows the requesting process to read from a resource while other processes simultaneously read from the same resource. No processes can write to the resource while the requesting process holds the lock. This mode is an example of a shared lock.

LKM_PWMODE

Allows the requesting process to read or write to a resource, and allows other processes that have concurrent read access to read from the resource. This mode is an example of an update lock.

LKM_EXMODE

Allows the requesting process to read or write to a resource while it prevents any other process from accessing that resource.

7.2.2.3.2. lksb

A pointer to the lock status block (`struct lockstatus`). Use this data structure to specify the contents of the lock value block and the timeout value for the request. For a lock conversion request, you must also use this structure to specify the lock ID of the lock. The lock manager writes the status of the lock request and the lock ID assigned to the request in the lock status block. For more information about this structure, see Section 3.2.3.

7.2.2.3.3. flags

The lock request takes various flags that modify its behavior. The flags supported are listed alphabetically as follows:

LKM_CONVERT

Indicates a lock conversion request.

LKM_FINDLOCAL

Used to acquire a lock on an existing local lock resource; that is, a lock resource created by a previous lock request that specified the LKM_LOCAL flag. The lock manager queries each cluster node, looking for the location of the local lock resource. If the lock manager cannot find the lock resource master on any cluster node, it creates a new local lock resource. For more information, see Section 3.3.10.

Subsequent requests to manipulate the lock require only the lock ID (not the LKM_FINDLOCAL flag). If the lock request includes the LKM_CONVERT flag (that is, the request is a conversion), the lock manager ignores the LKM_FINDLOCAL flag.

> **Note:** A lock request that specifies the LKM_FINDLOCAL flag takes longer to complete than a lock request that specifies the LKM_LOCAL flag, or even a standard lock request. Use this flag only when you are certain the lock resource specified was created with the LKM_LOCAL flag.

LKM_INVVALBLK

Allows clients to invalidate the lock value block associated with the lock resource. If the lock on the lock resource is not a PW or EX mode lock, the flag is ignored. (You must have a PW or EX mode lock on a lock resource to modify a lock value block.)

LKM_LOCAL

Specifies that the lock manager bypass the lock resource directory look up that it normally performs and create the lock resource master on the local node. The lock resource should not already exist anywhere in the cluster. Subsequent requests to manipulate this lock require only the lock ID (and not the LKM_LOCAL flag). If the

lock request includes the LKM_CONVERT flag (that is, the request is a conversion), the lock manager ignores the LKM_LOCAL flag.

> **Note:** When you specify the LKM_LOCAL flag, the lock manager does not check the lock resource directory to determine if the lock resource is already mastered on a cluster node, as it does for standard lock requests. Consequently, using this flag, you can create duplicate masters of lock resources, which can compromise lock integrity and result in data corruption.
>
> The LKM_LOCAL flag should only be used to acquire short-lived locks on short-lived lock resources. If your application must acquire additional locks on a local lock resource, you must specify the LKM_FINDLOCAL flag when requesting the lock. For more information about local lock resource, see Section 3.3.10.

LKM_NODLCKWT

Directs the lock manager to exclude this lock request from consideration when it periodically performs deadlock detection processing.

LKM_NOQUEUE

Requests that the lock manager not put the lock request on the wait queue if it cannot be immediately granted. The lock manager returns the status DLM_NOTQUEUED in the lock status block.

LKM_ORPHAN

Requests that the lock manager not purge this lock if the application fails. Use this flag with great care and only if you have a transaction recovery process that will eventually remove the orphaned locks.

LKM_PROC_OWNED

Directs the lock manager to exclude this lock from the lock group. This lock is owned by the process and not by the group. Other clients belonging to the group may not manipulate this lock.

LKM_SNGLDLCK

Requests that the lock manager check this lock request for self-client deadlock.

> **Note:** The LKM_SNGLDLCK flag is obsolete but is retained for backwards compatibility. The lock manager now checks for self-client deadlock by default.

LKM_SYNCSTS

Requests that the lock request return synchronously, if possible. If the lock manager can grant the request, the `dlmlock` routine returns the status DLM_SYNC, instead of DLM_NORMAL, and there is no asynchronous return. If the lock manager cannot grant the request synchronously, the `dlmlock` routine returns DLM_NORMAL and the lock manager queues the request as it would any other request.

LKM_TIMEOUT

Requests that the lock manager cancel the lock request or lock conversion request if the request cannot be granted within the time limit specified in the timeout field of the lock status block. If the time limit expires, the lock manager cancels the operation and queues the AST routine which returns the status value DLM_TIMEOUT. You specify the timeout value in units of hundredths of a second (0.01). For example, a timeout value of 500 specifies five seconds.

LKM_VALBLK

Requests that the lock manager return the current contents of the lock value block in the lock status block. When this flag is specified in a lock conversion request that is down-converting a lock from EX or PW mode to a less restrictive mode, the lock manager assigns the value specified in the lock status block to the lock value block of the lock resource. The lock value block is a 16-byte array containing application-specific information. This information is user-defined and interpreted by the application. For more information about the lock value block, see Section 3.6

7.2.2.3.4. name

The name of the requested lock resource. A resource name can contain binary data.

7.2.2.3.5. namelen

The length of the lock resource name provided in the name parameter. A resource name cannot exceed 31 characters.

7.2.2.3.6. ast

The address of a function the lock manager queues for execution when it finishes processing the lock request. Your application triggers the execution of this routine by calling the ASTpoll routine.

7.2.2.3.7. astargs

An argument that is passed to the routine specified by the ast or bast argument when that function is invoked. Typically used to pass a value that uniquely identifies the lock request when the status is returned. For example:

```
void ast_func(void *astargs);
```

7.2.2.3.8. bast (blocking AST)

The address of a function invoked if the requested lock is granted and later blocks another lock request. The blocking AST routine is called with two arguments: the astargs argument previously specified, and the requested mode that caused the queuing of the blocking AST routine. For example:

```
void bast_func(void *astargs, int mode);
```

7.2.2.4. Status Codes

The status codes returned are listed alphabetically as follows:

DLM_BADARGS

One of the following:

- The request included the LKM_VALBLK flag but passed a NULL pointer to the lock status block.
- The request passed a NULL pointer to the lock status block.
- The request included an invalid flag.
- The request passed a NULL lock resource name pointer, but did not include the LKM_CONVERT flag. (A conversion request requires a valid lock ID, but does not require a valid lock resource name.)

DLM_BADPARAM

The lock mode specified is not a valid lock mode.

DLM_DENIED_NOASTS

No more ASTs are available.

DLM_IVBUFLEN

The `namelen` (name length) was less than 1 or greater than 31 characters.

DLM_LOCKID

The lock ID is invalid.

DLM_NOLOCKMGR

A request timed out waiting for a response from the lock manager. If the lock manager is restarted while it is being used by a client, the next lock request returns this status.

DLM_NORMAL

The lock request completed successfully.

DLM_REJECTED

The lock manager does not recognize the client. This occurs if a lock manager is restarted during a lock session.

DLM_SYNC

The lock request included the LKM_SYNCSTS flag and the lock manager was able to grant it synchronously.

DLM_SYSERR

A data format error occurred, indicating a problem with the network facilities or an internal error with the lock manager.

7.2.2.5. Status Codes Returned in the Lock Status Block

The status codes returned in the lock status block are listed alphabetically as follows:

DLM_ABORT

A waiting lock was canceled by a `dlmunlock_sync` call with the LKM_CANCEL flag set.

DLM_CANCEL

A blocked conversion was canceled by a `dlmunlock_sync` call with the LKM_CANCEL flag set. The lock retains its original granted mode.

DLM_CVTUNGRANT

The request attempted to convert a lock that was blocked in the WAIT state.

DLM_DEADLOCK

The lock manager canceled this request to prevent deadlock from occurring.

DLM_DENIED

The request attempted to convert a lock that was already blocked on a conversion request.

DLM_DENIED_NOLOCKS

Either no more locks or no more resources are available. For information about the lock manager allocates locks and lock resources, see Chapter 6.

DLM_NORMAL

The lock request completed successfully.

DLM_NOTQUEUED

The request included the LKM_NOQUEUE flag and could not be satisfied immediately.

DLM_TIMEOUT

The timeout expired before this request was able to complete.

DLM_VALNOTVALID

The lock request, which included a request for the lock value block, completed successfully; however, the lock value block is not valid. This indicates that a client terminated while holding a lock on the lock resource at the LKM_EXMODE or LKM_PWMODE mode or that a client invalidated the lock value block by specifying the LKM_INVVALBLK flag with the lock routines.

7.2.2.6. Example

For an example of using this routine, see Section 3.3.5

7.2.3. `dlmlockx`

7.2.3.1. Syntax

```
dlm_stats_t dlmlockx(mode, lksb, flags, name, namelen, ast, astargs, bast, xid);
int mode;
struct lockstatus *lksb;
int flags;
void *name;
unsigned int namelen;
void (*ast)();
void *astargs;
void (*bast)();
```

```
dlm_xid_t *xid;
```

7.2.3.2. Description

Use the `dlmlockx` routine to make an asynchronous (non-blocking) request to acquire or convert a lock on a lock resource, and specify a transaction ID for that lock. The `dlmlockx` routine performs the same function as the `dlmlock` routine. See Section 7.2.2 for a description of the base functionality.

Additionally, the `dlmlockx` routine accepts a transaction ID (also called an XID or deadlock ID) as a parameter. Normally, the lock manager assumes the process that created the lock owns the lock when determining whether a deadlock cycle exists. By specifying a transaction ID, a lock client can attribute the ownership of a lock to a transaction rather than to a process. For deadlock detection, therefore, a transaction replaces a process or group as the owner of a lock.

You must specify a transaction ID when calling the `dlmlockx` routine. The transaction ID should either point to an eight-byte XID value or be NULL. Also, you must also set the LKM_XID_CONFLICT flag when calling the `dlmlockx` routine. This flag will eventually control functionality not included in this release.

The lock manager uses the transaction ID parameter only when creating a lock; it ignores this flag when converting a lock.

A transaction ID does not span nodes. Therefore, the lock manager considers equivalent transaction IDs on different nodes to be different transaction IDs.

7.2.3.3. Parameters

7.2.3.3.1. mode

The requested lock mode, required for both lock requests and conversion requests. The modes supported are listed below in order of severity, from least restrictive to most restrictive:

LKM_NLMODE

> Does not grant the requesting process any access to the resource, but indicates future interest in the resource. This mode acts as a placeholder for later conversion requests.

LKM_CRMODE

> Allows the requesting process to read from a resource, and allows other processes simultaneous read or write access to the same resource. This mode allows an unprotected read operation.

LKM_CWMODE

Allows the requesting process to read or write to a resource while other processes simultaneously read or write to the same resource. This mode allows an unprotected write operation.

LKM_PRMODE

Allows the requesting process to read from a resource while other processes simultaneously read from the same resource. No processes can write to the resource while the requesting process holds the lock. This mode is an example of a shared lock.

LKM_PWMODE

Allows the requesting process to read or write to a resource, and allows other processes that have concurrent read access to read from the resource. This mode is an example of an update lock.

LKM_EXMODE

Allows the requesting process to read or write to a resource while it prevents any other process from accessing that resource.

7.2.3.3.2. lksb

A pointer to the lock status block (`struct lockstatus`). Use this data structure to specify the contents of the lock value block and the timeout value for the request. For a lock conversion request, you must also use this structure to specify the lock ID of the lock. The lock manager writes the status of the lock request and the lock ID assigned to the request in the lock status block. For more information about this structure, see Section 3.2.3.

7.2.3.3.3. flags

The lock request takes various flags that modify its behavior. The flags supported are listed alphabetically as follows:

LKM_CONVERT

Indicates a lock conversion request.

LKM_FINDLOCAL

Used to acquire a lock on an existing local lock resource; that is, a lock resource created by a previous lock request that specified the LKM_LOCAL flag. The lock manager queries each cluster node, looking for the location of the local lock resource. If the lock manager cannot find the lock resource master on any cluster node, it creates a new local lock resource. For more information, see Section 3.3.10.

Subsequent requests to manipulate the lock require only the lock ID (not the LKM_FINDLOCAL flag). If the lock request includes the LKM_CONVERT flag (that is, the request is a conversion), the lock manager ignores the LKM_FINDLOCAL flag.

> **Note:** A lock request that specifies the LKM_FINDLOCAL flag takes longer to complete than a lock request that specifies the LKM_LOCAL flag, or even a standard lock request. Use this flag only when you are certain the lock resource specified was created with the LKM_LOCAL flag.

LKM_INVVALBLK

Allows clients to invalidate the lock value block associated with the lock resource. If the lock on the lock resource is not a PW or EX mode lock, the flag is ignored. (You must have a PW or EX mode lock on a lock resource to modify a lock value block.)

LKM_LOCAL

Specifies that the lock manager bypass the lock resource directory look up that it normally performs and create the lock resource master on the local node. The lock resource should not already exist anywhere in the cluster. Subsequent requests to manipulate this lock require only the lock ID (and not the LKM_LOCAL flag). If the lock request includes the LKM_CONVERT flag (that is, the request is a conversion), the lock manager ignores the LKM_LOCAL flag.

> **Note:** When you specify the LKM_LOCAL flag, the lock manager does not check the lock resource directory to determine if the lock resource is already mastered on a cluster node, as it does for standard lock requests. Consequently, using this flag, you can create duplicate masters of lock resources, which can compromise lock integrity and result in data corruption.
>
> The LKM_LOCAL flag should only be used to acquire short-lived locks on short-lived lock resources. If your application must acquire additional locks on a local lock resource, you must specify the LKM_FINDLOCAL flag when requesting the lock. For more information about local lock resource, see Section 3.3.10.

LKM_NODLCKWT

Directs the lock manager to exclude this lock request from consideration when it periodically performs deadlock detection processing.

LKM_NOQUEUE

Requests that the lock manager not put the lock request on the wait queue if it cannot be immediately granted. The lock manager returns the status DLM_NOTQUEUED in the lock status block.

LKM_ORPHAN

Requests that the lock manager not purge this lock if the application fails. Use this flag with great care and only if you have a transaction recovery process that will eventually remove the orphaned locks.

LKM_PROC_OWNED

Directs the lock manager to exclude this lock from the lock group. This lock is owned by the process and not by the group. Other clients belonging to the group may not manipulate this lock.

LKM_SNGLDLCK

Requests that the lock manager check this lock request for self-client deadlock.

Note: The LKM_SNGLDLCK flag is obsolete but is retained for backwards compatibility. The lock manager now checks for self-client deadlock by default.

LKM_SYNCSTS

Requests that the lock request return synchronously, if possible. If the lock manager can grant the request, the `dlmlock` routine returns the status DLM_SYNC, instead of DLM_NORMAL, and there is no asynchronous return. If the lock manager cannot grant the request synchronously, the `dlmlock` routine returns DLM_NORMAL and the lock manager queues the request as it would any other request.

LKM_TIMEOUT

Requests that the lock manager cancel the lock request or lock conversion request if the request cannot be granted within the time limit specified in the timeout field of the lock status block. If the time limit expires, the lock manager cancels the operation and queues the AST routine which returns the status value DLM_TIMEOUT. You specify the timeout value in units of hundredths of a second (0.01). For example, a timeout value of 500 specifies five seconds.

LKM_VALBLK

Requests that the lock manager return the current contents of the lock value block in the lock status block. When this flag is specified in a lock conversion request that is down-converting a lock from EX or PW mode to a less restrictive mode, the lock manager assigns the value specified in the lock status block to the lock value block of the lock resource. The lock value block is a 16-byte array containing application-specific information.

This information is user-defined and interpreted by the application. For more information about the lock value block, see Section 3.6.

LKM_XID_CONFLICT

Requests that transaction IDs are used only for deadlock detection. Currently, you must set this flag. The lock manager returns an error if this flag is not set.

7.2.3.3.4. name

The name of the requested lock resource. A resource name can contain binary data.

7.2.3.3.5. namelen

The length of the lock resource name provided in the name parameter. A resource name cannot exceed 31 characters.

7.2.3.3.6. ast

The address of a function the lock manager queues for execution when it finishes processing the lock request. Your application triggers the execution of this routine by calling the ASTpoll routine.

7.2.3.3.7. astargs

An argument that is passed to the routine specified by the ast or bast argument when that function is invoked. Typically used to pass a value that uniquely identifies the lock request when the status is returned. For example:

```
void ast_func(void *astargs);
```

7.2.3.3.8. bast (blocking AST)

The address of a function invoked if the requested lock is granted and later blocks another lock request. The blocking AST routine is called with two arguments: the astargs argument previously specified, and the requested mode that caused the queuing of the blocking AST routine. For example:

```
void bast_func(void *astargs, int mode);
```

7.2.3.3.9. xid

A pointer to an eight-byte transaction ID or NULL. A NULL value indicates the lock will be owned by a group or process.

7.2.3.4. Status Codes

The status codes returned are listed alphabetically as follows:

DLM_BADARGS

One of the following:

- The request included the LKM_VALBLK flag but passed a NULL pointer to the lock status block.
- The request passed a NULL pointer to the lock status block.
- The request included an invalid flag.
- The request passed a NULL lock resource name pointer, but did not include the LKM_CONVERT flag. (A conversion request requires a valid lock ID, but does not require a valid lock resource name.)

DLM_BADPARAM

The lock mode specified is not a valid lock mode.

DLM_DENIED_NOASTS

No more ASTs are available.

DLM_IVBUFLEN

The `namelen` (name length) was less than 1 or greater than 31 characters.

DLM_LOCKID

The lock ID is invalid.

DLM_NOLOCKMGR

A request timed out waiting for a response from the lock manager. If the lock manager is restarted while it is being used by a client, the next lock request returns this status.

DLM_NORMAL

The lock request completed successfully.

DLM_REJECTED

The lock manager does not recognize the client. This occurs if a lock manager is restarted during a lock session.

DLM_SYNC

The lock request included the LKM_SYNCSTS flag and the lock manager was able to grant it synchronously.

DLM_SYSERR

A data format error occurred, indicating a problem with the network facilities or an internal error with the lock manager.

7.2.3.5. Status Codes Returned in the Lock Status Block

The status codes returned in the lock status block are listed alphabetically as follows:

DLM_ABORT

A waiting lock was canceled by a `dlmunlock_sync call` with the LKM_CANCEL flag set.

DLM_CANCEL

A blocked conversion was canceled by a `dlmunlock_sync` call with the LKM_CANCEL flag set. The lock retains its original granted mode.

DLM_CVTUNGRANT

The request attempted to convert a lock that was blocked in the WAIT state.

DLM_DEADLOCK

The lock manager canceled this request to prevent deadlock from occurring.

DLM_DENIED

The request attempted to convert a lock that was already blocked on a conversion request.

DLM_DENIED_NOLOCKS

Either no more locks or no more resources are available. For information about the lock manager allocates locks and lock resources, see Chapter 6

DLM_NORMAL

The lock request completed successfully.

DLM_NOTQUEUED

The request included the LKM_NOQUEUE flag and could not be satisfied immediately.

DLM_TIMEOUT

The timeout expired before this request was able to complete.

DLM_VALNOTVALID

The lock request, which included a request for the lock value block, completed successfully; however, the lock value block is not valid. This indicates that a client terminated while holding a lock on the lock resource at the LKM_EXMODE or LKM_PWMODE mode or that a client invalidated the lock value block by specifying the LKM_INVVALBLK flag with the lock routines.

7.2.3.6. Example

```
dlm_stats = status;

status = dlmlockx(LKM_CRMODE,   /* mode  */
           &lksb[which_lock],   /* lock status block */
                  LKM_VALBLK,
                     "RES-A",   /* name  */
                           5,   /* namelen  */
                    ast_func,   /* ast routine */
                     &astarg,   /* astargs */
                           0,   /* bast  */
                       &xid);   /* transaction id */
```

7.2.4. `dlmlock_sync`

7.2.4.1. Syntax

```
dlm_stats_t dlmlock_sync(mode, lksb, flags, name, namelen, bastargs, bast);
int mode;
struct lockstatus lksb;
int flags;
void *name;
unsigned int namelen;
void *bastargs;
```

```
void (*bast)(void *);
```

7.2.4.2. Description

Use the `dlmlock_sync` routine to acquire or convert a lock on a lock resource and obtain a synchronous return. If the lock resource does not exist, the lock manager creates it.

A synchronous request performs the same function as an asynchronous request, but does not return control to the calling process until the request is resolved. The calling process does not have to poll for an AST; it simply waits until the request returns.

Since the `dlmlock_sync` routine does not use an AST to signal completion, it does not require a pointer to an ast function as an argument.

The various lock modes specify different degrees of access to a lock resource. You specify this mode as a part of the request. These lock modes are described in Section 7.2.4.3.

To convert an existing lock to a different mode, you must specify the LKM_CONVERT flag. You can also control other aspects of lock manager behavior by specifying flags as part of your request. For more information about the flags supported, see the listing of flags in Section 7.2.4.3.

The lock manager returns status in two locations: the status value returned by the dlmlock_sync routine and the status field of the lock status block. The status value returned by the `dlmlock_sync` routine indicates whether the request was accepted by the lock manager. The DLM_NORMAL status value indicates your request was successfully queued. If your request cannot be queued because of syntax problems or invalid arguments, your request is aborted and the dlmlock_sync routine returns an error status code. See Section 7.2.4.4for a list of these status values.

The lock manager returns the status of the request (whether it was granted, denied, canceled or aborted) in the status field of the lock status block. The DLM_NORMAL status value indicates your request was granted. See Section 7.2.4.5 for a list of other possible status values. (For information about the composition of the lock status block, see Section 3.2.3.)

> **Note:** This routine can be called only by user-space clients of the DLM. Kernel-space clients cannot call it.

7.2.4.3. Parameters

7.2.4.3.1. mode

The requested lock mode, required for both lock requests and conversion requests. The modes supported are listed

below in order of severity, from least restrictive to most restrictive:

LKM_NLMODE

Does not grant the requesting process any access to the resource, but indicates future interest in the resource. This mode acts as a placeholder for later conversion requests.

LKM_CRMODE

Allows the requesting process to read from a resource, and allows other processes simultaneous read or write access to the same resource. This mode allows an unprotected read operation.

LKM_CWMODE

Allows the requesting process to read or write to a resource while other processes simultaneously read or write to the same resource. This mode allows an unprotected write operation.

LKM_PRMODE

Allows the requesting process to read from a resource while other processes simultaneously read from the same resource. No processes can write to the resource while the requesting process holds the lock. This mode is an example of a shared lock.

LKM_PWMODE

Allows the requesting process to read or write to a resource, and allows other processes that have concurrent read access to read from the resource. This mode is an example of an update lock.

LKM_EXMODE

Allows the requesting process to read or write to a resource while it prevents any other process from accessing that resource.

7.2.4.3.2. lksb

A pointer to the lock status block (`struct lockstatus`). Use this data structure to specify the contents of the lock value block and the timeout value for the request. For a lock conversion request, you must also use this structure to specify the lock ID of the lock. The lock manager writes the status of the lock request and the lock ID assigned to the request in the lock status block. For more information about this structure, see Section 3.2.3.

7.2.4.3.3. flags

The lock request takes various flags that modify its behavior. The flags supported are listed alphabetically as follows:

LKM_CONVERT

Indicates a lock conversion request.

LKM_FINDLOCAL

Used to acquire a lock on an existing local lock resource; that is, a lock resource created by a previous lock request that specified the LKM_LOCAL flag. The lock manager queries each cluster node, looking for the location of the local lock resource. If the lock manager cannot find the lock resource master on any cluster node, it creates a new local lock resource. For more information, see Section 3.3.10.

Subsequent requests to manipulate the lock require only the lock ID (not the LKM_FINDLOCAL flag). If the lock request includes the LKM_CONVERT flag (that is, the request is a conversion), the lock manager ignores the LKM_FINDLOCAL flag.

Note: A lock request that specifies the LKM_FINDLOCAL flag takes longer to complete than a lock request that specifies the LKM_LOCAL flag, or even a standard lock request. Use this flag only when you are certain the lock resource specified was created with the LKM_LOCAL flag.

LKM_INVVALBLK

Allows clients to invalidate the lock value block associated with the lock resource. If the lock on the lock resource is not a PW or EX mode lock, the flag is ignored. (You must have a PW or EX mode lock on a lock resource to modify a lock value block.)

LKM_LOCAL

Specifies that the lock manager bypass the lock resource directory look up that it normally performs and create the lock resource master on the local node. The lock resource should not already exist anywhere in the cluster. Subsequent requests to manipulate this lock require only the lock ID (and not the LKM_LOCAL flag). If the lock request includes the LKM_CONVERT flag (that is, the request is a conversion), the lock manager ignores the LKM_LOCAL flag.

Note: When you specify the LKM_LOCAL flag, the lock manager does not check the lock resource directory to determine if the lock resource is already mastered on a cluster node, as it does for standard lock requests. Consequently, using this flag, you can create duplicate masters of lock resources, which can compromise lock integrity and result in data corruption.

The LKM_LOCAL flag should only be used to acquire short-lived locks on short-lived lock resources. If your application must acquire additional locks on a local lock resource, you must specify the LKM_FINDLOCAL flag when requesting the lock. For more information about local lock resource, see Section 3.3.10.

LKM_NODLCKWT

Directs the lock manager to exclude this lock request from consideration when it periodically performs deadlock detection processing.

LKM_NOQUEUE

Requests that the lock manager not put the lock request on the wait queue if it cannot be immediately granted. The lock manager returns the status DLM_NOTQUEUED in the lock status block.

LKM_ORPHAN

Requests that the lock manager not purge this lock if the application fails. Use this flag with great care and only if you have a transaction recovery process that will eventually remove the orphaned locks.

LKM_PROC_OWNED

Directs the lock manager to exclude this lock from the lock group. This lock is owned by the process and not by the group. Other clients belonging to the group may not manipulate this lock.

LKM_SNGLDLCK

Requests that the lock manager check this lock request for self-client deadlock.

Note: The LKM_SNGLDLCK flag is obsolete but is retained for backwards compatibility. The lock manager now checks for self-client deadlock by default.

LKM_TIMEOUT

Requests that the lock manager cancel the lock request or lock conversion request if the request cannot be granted within the time limit specified in the timeout field of the lock status block. If the time limit expires, the lock manager cancels the operation and queues the AST routine which returns the status value DLM_TIMEOUT. You specify the timeout value in units of hundredths of a second (0.01). For example, a timeout value of 500 specifies five seconds.

LKM_VALBLK

Requests that the lock manager return the current contents of the lock value block in the lock status block. When this flag is specified in a lock conversion request that is down-converting a lock from EX or PW mode to a less restrictive mode, the lock manager assigns the value specified in the lock status block to the lock value block of the lock resource. The lock value block is a 16-byte array containing application-specific information.

This information is user-defined and interpreted by the application. For more information about the lock value block, see Section 3.6

7.2.4.3.4. name

The name of the requested lock resource. A resource name can contain binary data.

7.2.4.3.5. namelen

The length of the lock resource name provided in the name parameter. A resource name cannot exceed 31 characters.

7.2.4.3.6. bastargs

An argument that is passed to the routine specified by the bast argument when that function is invoked. Typically used to pass a value that uniquely identifies the lock request when the lock is blocking another lock request. For example:

```
void bast_func(void *bastargs, int mode);
```

7.2.4.3.7. bast (blocking AST)

The address of a function invoked if the requested lock is granted and later blocks another lock request. The blocking AST routine is called with two arguments: the bastargs argument previously specified, and the requested mode that caused the queuing of the blocking AST routine. For example:

```
void bast_func(void *bastargs, int mode);
```

7.2.4.4. Status Codes

The status codes returned are listed alphabetically as follows:

DLM_BADARGS

One of the following:

- The request included the LKM_VALBLK flag but passed a NULL pointer to the lock status block.
- The request passed a NULL pointer to the lock status block.
- The request included an invalid flag.

- The request passed a NULL lock resource name pointer, but did not include the LKM_CONVERT flag. (A conversion request requires a valid lock ID, but does not require a valid lock resource name.)

DLM_BADPARAM

The lock mode specified is not a valid lock mode.

DLM_IVBUFLEN

The `namelen` (name length) was less than 1 or greater than 31 characters.

DLM_LOCKID

The lock ID is invalid.

DLM_NOLOCKMGR

A request timed out waiting for a response from the lock manager. If the lock manager is restarted while it is being used by a client, the next lock request returns this status.

DLM_NORMAL

The lock request completed successfully.

7.2.4.5. Status Codes Returned in the Lock Status Block

The status codes returned in the lock status block are listed alphabetically as follows:

DLM_CVTUNGRANT

The request attempted to convert a lock that was blocked in the WAIT state.

DLM_DEADLOCK

The lock manager canceled this request to prevent deadlock from occurring.

DLM_DENIED

The request attempted to convert a lock that was already blocked on a conversion request.

DLM_DENIED_NOLOCKS

Either no more locks or no more resources are available. For information about the lock manager allocates locks and lock resources, see Chapter 6.

DLM_NORMAL

The lock request completed successfully.

DLM_NOTQUEUED

The request included the LKM_NOQUEUE flag and could not be satisfied immediately.

DLM_TIMEOUT

The timeout expired before this request was able to complete.

DLM_VALNOTVALID

The lock request, which included a request for the lock value block, completed successfully; however, the lock value block is not valid. This indicates that a client terminated while holding a lock on the lock resource at the LKM_EXMODE or LKM_PWMODE mode or that a client invalidated the lock value block by specifying the LKM_INVVALBLK flag with the lock routines.

7.2.4.6. Example

```
dlm_stats = status;

status = dlmlock_sync(LKM_CRMODE,   /* mode  */
               &lksb[which_lock],   /* lock status block */
                      LKM_VALBLK,
                         "RES-A",   /* name  */
                               5,   /* namelen  */
                        &bastargs,   /* bastargs */
                        bast_func);  /* bast routine */
```

7.2.5. `dlmlockx_sync`

7.2.5.1. Syntax

```
dlm_stats_t dlmlockx_sync(mode, lksb, flags, name, namelen, bastargs, bast, xid);
int mode;
struct lockstatus lksb;
```

```
int flags;
void *name;
unsigned int namelen;
void *bastargs;
void (*bast)(void *);
dlm_xid_t *xid;
```

7.2.5.2. Description

Use the `dlmlockx_sync` routine to acquire or convert a lock on a lock resource, specify a transaction ID for that lock, and obtain a synchronous return. The `dlmlockx_sync` routine performs the same function as the `dlmlock_sync` routine. See Section 7.2.4 for a description of the base functionality of the `dlmlock_sync` routine.

Additionally, the `dlmlockx_sync` routine accepts a transaction ID (also called an XID or deadlock ID) as a parameter. Normally, the lock manager assumes the process that created the lock owns the lock when determining whether a deadlock cycle exists. By specifying a transaction ID, a lock client can attribute the ownership of a lock to a transaction rather than to a process. For deadlock detection, therefore, a transaction replaces a process or group as the owner of a lock.

You must specify a transaction ID when calling the `dlmlockx_sync` routine. The transaction ID should either point to an eight-byte XID value or be NULL. Also, you must also set the LKM_XID_CONFLICT flag when calling the `dlmlockx_sync` routine. This flag will eventually control functionality not included in this release. The lock manager uses the transaction ID parameter only when creating a lock; it ignores this flag when converting a lock.

A transaction ID does not span nodes. Therefore, the lock manager considers equivalent transaction IDs on different nodes to be different transaction IDs.

> **Note:** This routine can be called only by user-space clients of the DLM. Kernel-space clients cannot call it.

7.2.5.3. Parameters

7.2.5.3.1. mode

The requested lock mode, required for both lock requests and conversion requests. The modes supported are listed below in order of severity, from least restrictive to most restrictive:

LKM_NLMODE

Does not grant the requesting process any access to the resource, but indicates future interest in the resource. This mode acts as a placeholder for later conversion requests.

LKM_CRMODE

Allows the requesting process to read from a resource, and allows other processes simultaneous read or write access to the same resource. This mode allows an unprotected read operation.

LKM_CWMODE

Allows the requesting process to read or write to a resource while other processes simultaneously read or write to the same resource. This mode allows an unprotected write operation.

LKM_PRMODE

Allows the requesting process to read from a resource while other processes simultaneously read from the same resource. No processes can write to the resource while the requesting process holds the lock. This mode is an example of a shared lock.

LKM_PWMODE

Allows the requesting process to read or write to a resource, and allows other processes that have concurrent read access to read from the resource. This mode is an example of an update lock.

LKM_EXMODE

Allows the requesting process to read or write to a resource while it prevents any other process from accessing that resource.

7.2.5.3.2. lksb

A pointer to the lock status block (`struct lockstatus`). Use this data structure to specify the contents of the lock value block and the timeout value for the request. For a lock conversion request, you must also use this structure to specify the lock ID of the lock. The lock manager writes the status of the lock request and the lock ID assigned to the request in the lock status block. For more information about this structure, see Section 3.2.3.

7.2.5.3.3. flags

The lock request takes various flags that modify its behavior. The flags supported are listed alphabetically as follows:

LKM_CONVERT

Indicates a lock conversion request.

LKM_FINDLOCAL

Used to acquire a lock on an existing local lock resource; that is, a lock resource created by a previous lock request that specified the LKM_LOCAL flag. The lock manager queries each cluster node, looking for the location of the local lock resource. If the lock manager cannot find the lock resource master on any cluster node, it creates a new local lock resource. For more information, see Section 3.3.10.

Subsequent requests to manipulate the lock require only the lock ID (not the LKM_FINDLOCAL flag). If the lock request includes the LKM_CONVERT flag (that is, the request is a conversion), the lock manager ignores the LKM_FINDLOCAL flag.

Note: A lock request that specifies the LKM_FINDLOCAL flag takes longer to complete than a lock request that specifies the LKM_LOCAL flag, or even a standard lock request. Use this flag only when you are certain the lock resource specified was created with the LKM_LOCAL flag.

LKM_INVVALBLK

Allows clients to invalidate the lock value block associated with the lock resource. If the lock on the lock resource is not a PW or EX mode lock, the flag is ignored. (You must have a PW or EX mode lock on a lock resource to modify a lock value block.)

LKM_LOCAL

Specifies that the lock manager bypass the lock resource directory look up that it normally performs and create the lock resource master on the local node. The lock resource should not already exist anywhere in the cluster. Subsequent requests to manipulate this lock require only the lock ID (and not the LKM_LOCAL flag). If the lock request includes the LKM_CONVERT flag (that is, the request is a conversion), the lock manager ignores the LKM_LOCAL flag.

Note: When you specify the LKM_LOCAL flag, the lock manager does not check the lock resource directory to determine if the lock resource is already mastered on a cluster node, as it does for standard lock requests. Consequently, using this flag, you can create duplicate masters of lock resources, which can compromise lock integrity and result in data corruption.

The LKM_LOCAL flag should only be used to acquire short-lived locks on short-lived lock resources. If your application must acquire additional locks on a local lock resource, you must specify the LKM_FINDLOCAL flag when requesting the lock. For more information about local lock resource, see Section 3.3.10.

LKM_NODLCKWT

Directs the lock manager to exclude this lock request from consideration when it periodically performs deadlock detection processing.

LKM_NOQUEUE

Requests that the lock manager not put the lock request on the wait queue if it cannot be immediately granted. The lock manager returns the status DLM_NOTQUEUED in the lock status block.

LKM_ORPHAN

Requests that the lock manager not purge this lock if the application fails. Use this flag with great care and only if you have a transaction recovery process that will eventually remove the orphaned locks.

LKM_PROC_OWNED

Directs the lock manager to exclude this lock from the lock group. This lock is owned by the process and not by the group. Other clients belonging to the group may not manipulate this lock.

LKM_SNGLDLCK

Requests that the lock manager check this lock request for self-client deadlock.

Note: The LKM_SNGLDLCK flag is obsolete but is retained for backwards compatibility. The lock manager now checks for self-client deadlock by default.

LKM_TIMEOUT

Requests that the lock manager cancel the lock request or lock conversion request if the request cannot be granted within the time limit specified in the timeout field of the lock status block. If the time limit expires, the lock manager cancels the operation and queues the AST routine which returns the status value DLM_TIMEOUT. You specify the timeout value in units of hundredths of a second (0.01). For example, a timeout value of 500 specifies five seconds.

LKM_VALBLK

Requests that the lock manager return the current contents of the lock value block in the lock status block. When this flag is specified in a lock conversion request that is down-converting a lock from EX or PW mode to a less restrictive mode, the lock manager assigns the value specified in the lock status block to the lock value block of the lock resource. The lock value block is a 16-byte array containing application-specific information. This information is user-defined and interpreted by the application. For more information about the lock value block, see Section 3.6

7.2.5.3.4. name

The name of the requested lock resource. A resource name can contain binary data.

7.2.5.3.5. namelen

The length of the lock resource name provided in the name parameter. A resource name cannot exceed 31 characters.

7.2.5.3.6. bastargs

An argument that is passed to the routine specified by the bast argument when that function is invoked. Typically used to pass a value that uniquely identifies the lock request when the lock is blocking another lock request. For example:

```
void bast_func(void *bastargs, int mode);
```

7.2.5.3.7. bast (blocking AST)

The address of a function invoked if the requested lock is granted and later blocks another lock request. The blocking AST routine is called with two arguments: the bastargs argument previously specified, and the requested mode that caused the queuing of the blocking AST routine. For example:

```
void bast_func(void *bastargs, int mode);
```

7.2.5.4. Status Codes

The status codes returned are listed alphabetically as follows:

DLM_BADARGS

One of the following:

- The request included the LKM_VALBLK flag but passed a NULL pointer to the lock status block.
- The request passed a NULL pointer to the lock status block.
- The request included an invalid flag.
- The request passed a NULL lock resource name pointer, but did not include the LKM_CONVERT flag. (A conversion request requires a valid lock ID, but does not require a valid lock resource name.)

DLM_BADPARAM

The lock mode specified is not a valid lock mode.

DLM_IVBUFLEN

The `namelen` (name length) was less than 1 or greater than 31 characters.

DLM_LOCKID

The lock ID is invalid.

DLM_NOLOCKMGR

A request timed out waiting for a response from the lock manager. If the lock manager is restarted while it is being used by a client, the next lock request returns this status.

DLM_NORMAL

The lock request completed successfully.

7.2.5.5. Status Codes Returned in the Lock Status Block

The status codes returned in the lock status block are listed alphabetically as follows:

DLM_CVTUNGRANT

The request attempted to convert a lock that was blocked in the WAIT state.

DLM_DEADLOCK

The lock manager canceled this request to prevent deadlock from occurring.

DLM_DENIED

The request attempted to convert a lock that was already blocked on a conversion request.

DLM_DENIED_NOLOCKS

Either no more locks or no more resources are available. For information about the lock manager allocates locks and lock resources, see Chapter 6.

DLM_NORMAL

The lock request completed successfully.

DLM_NOTQUEUED

The request included the LKM_NOQUEUE flag and could not be satisfied immediately.

DLM_TIMEOUT

The timeout expired before this request was able to complete.

DLM_VALNOTVALID

The lock request, which included a request for the lock value block, completed successfully; however, the lock value block is not valid. This indicates that a client terminated while holding a lock on the lock resource at the LKM_EXMODE or LKM_PWMODE mode or that a client invalidated the lock value block by specifying the LKM_INVVALBLK flag with the lock routines.

7.2.5.6. Example

```
dlm_stats = status;

status = dlmlockx_sync(LKM_CRMODE,   /* mode  */
                &lksb[which_lock],   /* lock status block */
                       LKM_VALBLK,
                          "RES-A",   /* name  */
                                5,   /* namelen  */
                         &bastargs,   /* bastargs */
                         bast_func,   /* bast routine  */
                             &xid);   /* transaction id */
```

7.2.6. `dlmregister`

7.2.6.1. Syntax

```
union dlm_r dlmregister(name);
union dlm_rh dlmregister(name)
char *name;
```

7.2.6.2. Description

Before requesting a lock on a UNIX lock resource, you must register the lock resource-the object against which all locking occurs. Use the `dlmregister` routine to register (create) a lock resource. A lock resource remains in existence until the last process to have it registered exits.

7.2.6.3. Parameters

7.2.6.3.1. name

The name of the lock resource being registered. A lock resource name is a NULL-terminated string. A lock resource name can contain up to 255 bytes. This limit is defined by the value of the MAXRESOURCELEN constant in the `/usr/include/cluster/dlm.h` header file.

7.2.6.4. Return Values

After a lock resource has been registered successfully, the lock manager returns a lock resource handle to the calling program. A lock resource handle is defined by the `union dlm_rh` data structure. The lock resource handle is a token which must be passed to all subsequent lock requests.

If an error occurs, NULL is returned instead of a valid lock resource handle. In this case, the `dlm_errno` global variable contains the status code associated with the error.

7.2.6.5. Status Codes

DLM_IVBUKLEN

The request specified a lock resource name that was either less than one or greater than MAXRESOURCELEN.

DLM_MAXHANDLES

The system limit for resource handles for an application has been reached.

DLM_NOLOCKMGR

The lock manager daemon is not running. If the lock manager is restarted while it is being used by a client, the next lock request returns this status.

DLM_NORMAL

The register request completed successfully.

7.2.6.6. Example

```
union dlm_rh reshandle;

/* create a resource handle against which to lock */
reshandle = dlmregister("A Lock");
if (reshandle.rh == 0) {
        dlm_perror("Can’t register lock");
        exit(1);
}
```

7.2.7. `dlmregionlock`

7.2.7.1. Syntax

```
dlm_stats_t dlmregionlock(rh, offset, length, flags);
union dlm_rh rh;
unsigned long offset;
unsigned long length;
unsigned long flags;
```

7.2.7.2. Description

Use the `dlmregionlock` routine to acquire and release locks. You indicate you want to release a lock by setting the LOCK_UN flag.

7.2.7.3. Parameters

7.2.7.3.1. rh

A valid lock resource handle returned by an earlier call to the `dlmregister` routine.

7.2.7.3.2. offset

The lower bound of the region that the request should affect.

7.2.7.3.3. length

The length of the region starting at offset.

7.2.7.3.4. flags

A bitmask of various options, described in the following list. If you specify both the LOCK_EX and LOCK_SH flags, the LOCK_EX flag is honored.

LOCK_SH

A shared lock (read) is being requested. Multiple applications can simultaneously request shared locks, but no exclusive locks are granted while any shared locks are held on a specified region of the resource by any application other than the requesting application.

LOCK_EX

An exclusive lock (write) is being requested. Only one application can possess a write lock on a resource at any given time. A request for an exclusive lock fails if any locks are currently held on the specified region of the resource by any applications other than the requesting application.

LOCK_NB

Normally, if a lock request cannot be immediately granted because it is incompatible with existing locks, the requesting application will suspend (block) until the request can be completed. An application specifies the LOCK_NB option to indicate that this request is non-blocking. If the request would suspend, an error is returned instead. An application never blocks against locks that it holds. An application never blocks on an unlock request.

LOCK_UN

This flag specifies that the indicated resource region should be unlocked. Any regions currently locked by the requesting application that overlap the region specified in the unlock request are released.

7.2.7.4. Status Codes

DLM_BADARGS

The request specified an unsupported flag. The supported flags are LOCK_EX, LOCK_UN, LOCK_SH, and LOCK_NB.

DLM_DENIED

The request would block and had set the LOCK_NB flag, or it attempted to unlock a region that was not previously locked.

DLM_IVRESHANDLE

The resource handle is invalid.

DLM_NOLOCKMGR

The lock manager daemon is not running. If the lock manager is restarted while it is being used by a client, the next lock request returns this status.

DLM_NORMAL

The lock request completed successfully.

7.2.7.5. Example

```
union dlm_rh reshandle;
unsigned long offset;
unsigned long len;
dlm_stats_t status;

/* acquire an exclusive lock on region from byte 0 to 9 */
status = dlmregionlock(reshandle, offset, length, LOCK_EX);
if (status != DLM_NORMAL) {
        dlm_perror("Can’t acquire lock");
        exit(1);
}

/*  processing occurs here  */

/* release lock */
status = dlmregionlock(reshandle, offset, length, LOCK_UN);
if (status != DLM_NORMAL) {
        dlm_perror("Unlock failed");
        exit(1);
}
```

7.2.8. dlmunlock_sync

7.2.8.1. Syntax

```
dlm_stats_t dlmunlock_sync(lockid, valueblock, flags);
int lockid;
char *valueblock;
int flags;
```

7.2.8.2. Description

Use the dlmunlock_sync routine to make a synchronous (blocking) request to:

- Release a lock.
- Cancel a blocked lock request on the wait queue.
- Cancel a blocked conversion request on the convert queue.
- Invalidate a lock value block when releasing a lock held in PW or EX mode.

Note that the dlmunlock_sync routine always operates synchronously. There is no AST mechanism available. However, the release or cancellation of a lock can cause the queuing of AST routines associated with locks when they change state.

> **Note:** This routine can be called only by user-space clients of the DLM. Kernel-space clients cannot call it.

7.2.8.3. Parameters

7.2.8.3.1. lockid

A valid lock ID returned from a previous call to the dlmlock routine.

7.2.8.3.2. valueblock

The lock value block is a 16-byte structure containing information about the lock resource. This information is user-defined and interpreted by the application. It is not used by the lock manager.

The lock manager updates the contents of the lock value block associated with the lock name using the value contained in valueblock if the conditions are true:

- The request is an unlock request (the LKM_CANCEL flag is not included).
- The current granted mode of the lock is either EX or PW.
- The LKM_VALBLK flag was included.

7.2.8.3.3. flags

A bitmask of various options. The flags are as follows:

LKM_CANCEL

When you set the LKM_CANCEL flag, the `dlmunlock_sync` request:

- Cancels a request that is blocked and on the wait queue.
- Cancels a conversion request. The lock retains its original mode. If a conversion request was already granted, the lock manager returns a status of DLM_CANCELGRANT.

LKM_FORCE

Directs the lock manager to release a lock regardless of its current state. If the specified lock has been granted, the lock manager releases the lock. If the specified lock is waiting to convert from one state to another, the lock manager cancels the pending conversion and then releases the lock. If the specified lock has not been granted, the lock manager cancels the open request. If the force operation involves the canceling of a pending request, the appropriate AST will be queued indicating that the request was canceled.

If an unlock request includes both the LKM_CANCEL and LKM_FORCE flags, the lock manager ignores the LKM_FORCE flag.

If the LKM_FORCE flag is included in a lock request other than `dlmunlock_sync` or `dlmunlock`, it is ignored.

LKM_IVVALBLK

Allows clients to invalidate the lock value block associated with the lock resource. If the lock on the lock resource is not a PW or EX mode lock, the flag is ignored.

LKMVALBLK

Sets the lock value block from valueblock if the modes are appropriate. See the description of the valueblock argument. This flag is ignored if LKM_CANCEL is set.

7.2.8.4. Status Codes

The status codes returned are listed alphabetically as follows:

DLM_BADARGS

The request included the LKM_VALBLK flag, but passed a NULL pointer.

DLM_CANCELGRANT

The request attempted to cancel a conversion (by including the LKM_CANCEL flag), but the request was already granted.

DLM_DENIED

The request attempted to cancel a conversion, but the specified lock is not in a granted state and neither the LKM_CANCEL nor the LKM_FORCE flag was included in the unlock request.

DLM_IVLOCKID

The lock ID is invalid.

DLM_NOLOCKMGR

The lock manager daemon is not running. If the lock manager is restarted while it is being used by a client, the next lock request returns this status.

DLM_NORMAL

The unlock request completed successfully.

7.2.8.5. Example

For an example of using this routine, see Section 3.4.

7.2.9. dlmunlock

7.2.9.1. Syntax

```
dlm_stats_t dlmunlock(lockid, valueblock, flags, unlockast, unblockastargs, extrap);
int lockid;
char *valueblock;
int flags;
void (unlockast) ();
void *unlockastargs;
void *extrap;
```

7.2.9.2. Description

Use the `dlmunlock` routine to make an asynchronous (non-blocking) request to:

- Release a lock.
- Cancel a blocked lock request on the wait queue.
- Cancel a blocked conversion request on the convert queue.
- Invalidate a lock value block when releasing a lock held in PW or EX mode.

The lock manager returns status in two locations: the status value returned by the `dlmunlock` routine and the lstat argument passed to the `unlockast` function. The status value returned by the `dlmunlock` routine indicates whether the unlock request was accepted by the lock manager. If the unlock request cannot be accepted because of syntax problems or invalid arguments, it is rejected, and the `dlmunlock` routine returns an error status code. See Section 7.2.9.4 for a list of the status values.

A success status from the `dlmunlock` routine does not indicate that the unlock has been completed. The lock manager reports asynchronously whether the unlock was completed, denied, canceled, or aborted by queuing for execution the `unlockast` function specified as an argument to the `dlmunlock` routine.

The `unlockast` function has the following declaration:

```
void (*unlockast) (void *unlockastargs, dlm_stats_t lstat,
              void *extrap)
```

The unlockastargs and extrap values passed to the `unlockast` routine are the same values passed to the call to the `dlmunlock` routine. The lstat value will be the status value of the unlock completion request.

Note: The lstat value is the same as the value returned from the synchronous `dlmunlock_sync` routine.

An application triggers the `unlockast` routine by calling the `ASTpoll` routine. See Section 7.2.9.5 for a list of other possible status values.

7.2.9.3. Parameters

7.2.9.3.1. lockid

A valid lock ID returned from a previous call to the dlmlock routine.

7.2.9.3.2. valueblock

The lock value block is a 16-byte structure containing information about the lock resource. This information is user-defined and interpreted by the application. It is not used by the lock manager.

The lock manager updates the contents of the lock value block associated with the lock name using the value contained in valueblock if the conditions are true:

- The request is an unlock request (the LKM_CANCEL flag is not included).
- The current granted mode of the lock is either EX or PW.
- The LKM_VALBLK flag was included.

7.2.9.3.3. flags

A bitmask of various options. The flags are as follows:

LKM_CANCEL

When you set the LKM_CANCEL flag, the `dlmunlock_sync` request:

- Cancels a request that is blocked and on the wait queue.
- Cancels a conversion request. The lock retains its original mode. If a conversion request was already granted, the lock manager returns a status of DLM_CANCELGRANT.

LKM_FORCE

Directs the lock manager to release a lock regardless of its current state. If the specified lock has been granted, the lock manager releases the lock. If the specified lock is waiting to convert from one state to another, the lock manager cancels the pending conversion and then releases the lock. If the specified lock has not been granted, the lock manager cancels the open request. If the force operation involves the canceling of a pending request, the appropriate AST will be queued indicating that the request was canceled.

If an unlock request includes both the LKM_CANCEL and LKM_FORCE flags, the lock manager ignores the LKM_FORCE flag.

If the LKM_FORCE flag is included in a lock request other than `dlmunlock_sync` or `dlmunlock`, it is ignored.

LKM_IVVALBLK

Allows clients to invalidate the lock value block associated with the lock resource. If the lock on the lock resource is not a PW or EX mode lock, the flag is ignored.

LKMVALBLK

Sets the lock value block from valueblock if the modes are appropriate. See the description of the valueblock argument. This flag is ignored if LKM_CANCEL is set.

7.2.9.3.4. unlockast

The address of a function that is queued for execution by the lock manager when it finishes processing the unlock request.

7.2.9.3.5. unlockastargs

An argument passed to the routine specified by `unlockast`.

7.2.9.3.6. extrap

An extra context pointer passed to the routine specified by `unlockast`.

7.2.9.4. Status Codes

The status codes returned are listed alphabetically as follows:

DLM_BADARGS

The request included the LKM_VALBLK flag, but passed a NULL pointer.

DLM_DENIED

The request attempted to cancel a conversion, but the specified lock is not in a granted state and neither the LKM_CANCEL nor the LKM_FORCE flag was included in the unlock request.

DLM_IVLOCKID

The lock ID is invalid.

DLM_NOLOCKMGR

The lock manager daemon is not running. If the lock manager is restarted while it is being used by a client, the next lock request returns this status.

DLM_NORMAL

The unlock request completed successfully.

7.2.9.5. Status Codes Returned in the AST

DLM_CANCELGRANT

The request attempted to cancel a conversion (by including the LKM_CANCEL flag), but the request was already granted.

DLM_DENIED

The request attempted to cancel a conversion, but the specified lock is not in a granted state and neither the LKM_CANCEL nor the LKM_FORCE flag was included in the unlock request.

DLM_NORMAL

The unlock request completed successfully.

7.2.10. dlm_errmsg

7.2.10.1. Syntax

```
char *dlm_errmsg(status);
dlm_stats_t status;
```

7.2.10.2. Description

The dlm_errmsg routine takes a status code returned by the lock manager and returns a pointer to a printable version of the status code. The status codes that make up the dlm_stats_t enumerated type are constants, not printable character strings.

7.2.10.3. Parameters

7.2.10.3.1. status

A DLM API status code.

7.2.10.4. Returns

A NULL-terminated character string. For example, if the status code returned is DLM_NORMAL, the string returned is "DLM_NORMAL."

If the status parameter you specify as an argument is not a valid lock manager status code, the dlm_errmsg routine returns the string "Invalid status."

7.2.10.5. Example

The following code fragment uses the dlm_errmsg routine to convert the status code returned by the dlm_setnotify routine to a printable string. The string is then passed as an argument to the printf routine.

```
#include <cluster/dlm.h>

char *msg;
```

```
dlm_stats_t status;
.
.
.
status = dlm_setnotify( SIGUSR1, NULL );
if ( status != DLM_NORMAL )
{
    msg = dlm_errmsg(status);
    printf("dlm_setnotify returns %s",msg);
}
```

If the routine failed because the arguments passed were invalid, the following message would be printed to stderr:

```
dlm_setnotify returns DLM_BADARGS
```

7.2.11. `dlm_getresparams`

7.2.11.1. Syntax

```
dlm_stats_t dlm_getresparams(res_name, namelen, res_type, params);
char *res_name;
short namelen;
short res_type;
dlm_resparams_t *params;
```

7.2.11.2. Description

The `dlm_getresparams` routine returns the value of a lock resource's stickiness attribute.

7.2.11.3. Parameters

7.2.11.3.1. res_name

A NULL-terminated character string specifying the name of the lock resource.

7.2.11.3.2. namelen

The number of characters in the name.

7.2.11.3.3. res_type

Specifies the type of lock resource. For DLM lock resources, specify the constant DLM_RES_VMS. For UNIX lock resources, specify the constant DLM_RES_UNIX.

7.2.11.3.4. params

Address of the dlm_resparams_t structure that contains the value of the lock resource stickiness attribute.

7.2.11.4. Status Codes

The following is an alphabetical list of status values returned by the `dlm_getresparams` routine.

DLM_BADARGS

One of the following:

- The request specified an invalid resource type.
- The length of the lock resource name exceeds the limit.
- The pointer to the dlm_resparams_t structure is invalid.

DLM_BADRESOURCE

The lock resource specified is invalid.

DLM_NORMAL

The operation completed successfully.

7.2.11.5. Example

In the following code fragment, the application retrieves the value of the stickiness attribute.

```
#include <cluster/dlm.h>
dlm_resparams_t  resparams;
```

```
dlm_status_t status;

#define NAMELEN 7
.
.
.
status = dlm_getresparams("my_lock",  /* name of lock resource */
                          NAMELEN,  /* length of name        */
                      DLM_RES_VMS,  /* resource type         */
                      &resparams);  /* address of resparms   */
                                    /*     structure         */
if(status != DLM_NORMAL)
{
dlm_perror("Can’t read stickiness value");
exit(1);
}
```

7.2.12. `dlm_grp_attach`

7.2.12.1. Syntax

```
dlm_stats_t dlm_grp_attach(gid, flags);
int gid;
int flags;
```

7.2.12.2. Description

Use the `dlm_grp_attach` routine to attach a lock client to an existing lock group. Use the group ID returned by the `dlm_grp_create` routine to specify the group. A client may belong to only one group.

7.2.12.3. Parameters

7.2.12.3.1. gid

The group ID returned by the `dlm_grp_create` routine.

7.2.12.3.2. flags

None.

7.2.12.4. Status Codes

The status codes are listed below:

DLM_DENIED

The process was already attached to a lock group.

DLM_IVGROUPID

The request specified an invalid lock group.

DLM_NORMAL

The request completed successfully.

7.2.12.5. Example

```
int groupid = 0x1010000;
int ret;

/* Attach the current process to an existing group with id 0x1010000 */
ret = dlm_grp_attach(groupid, 0);
if (ret == DLM_NORMAL) {
printf("Successfully attached to group %d", groupid);
}
```

7.2.13. `dlm_grp_create`

7.2.13.1. Syntax

```
dlm_stats_t dlm_grp_create(gid, flags);
int *gid;
int flags;
```

7.2.13.2. Description

Use the `dlm_grp_create` routine to create a new lock group and associate the client with this group. The `dlm_grp_create` routine returns a group ID.

A lock group joins related lock client processes into a single entity. A lock client may create a new lock group or join an existing group. A lock client may belong to, at most, one lock group. Once a client belongs to a group, the group owns all subsequent locks created by that process. Any process in a group may manipulate locks owned by that group.

Alternatively, a process belonging to a lock group can pass the LKM_PROC_OWNED flag to either the `dlmlock` or `dlmlock_sync` routine to indicate that this lock is owned by the process, not by the group. Other processes belonging to the group may not manipulate this lock.

The lock manager does not purge a lock owned by a group until all processes belonging to the group have exited. The lock manager also purges if all group processes detach.

A lock group may not span cluster nodes. The lock manager only acknowledges a group ID on the node on which it was created. Therefore, a lock client on one node cannot join a group was created on a different node.

7.2.13.3. Parameters

7.2.13.3.1. gid

Pointer to location to store the group ID.

7.2.13.3.2. flags

None.

7.2.13.4. Status Codes

DLM_DENIED

The process was already attached to a group.

DLM_NORMAL

The request completed successfully.

7.2.13.5. Example

```
int groupid;
int ret;

/* Create lock group and associate process with group */
ret =  dlm_grp_create(&groupid, 0);
if (ret == DLM_NORMAL) {
     printf("Group created. Group id is %d", groupid);
}
```

7.2.14. `dlm_grp_detach`

7.2.14.1. Syntax

```
dlm_stats_t dlm_grpdetach(flags);
```

7.2.14.2. Description

Use the `dlm_grp_detach` routine to remove a process from a lock group. A process that has left a group can no longer manipulate locks owned by that group, including locks it created while belonging to the group. If a process is the last group member to leave a group, the locks owned by the group are purged and the group no longer exists. A client is implicitly removed from a group when it terminates.

7.2.14.3. Parameters

7.2.14.3.1. flags

Should be set to zero for now.

7.2.14.4. Status Codes

The status codes are listed below:

DLM_DENIED

The process was not attached to a group.

DLM_NORMAL

The request completed successfully.

7.2.14.5. Example

```
int ret;

/* Detach the current process from lock group */

ret = dlm_grp_detach(0);

if (ret == DLM_NORMAL) {
     printf("Successfully detach from lock group.");
}
```

7.2.15. `dlm_perror`

7.2.15.1. Syntax

```
void dlm_perror(message);
char *message;
```

7.2.15.2. Description

The `dlm_perror` routine allows an application to write a message to standard error that indicates why a lock request failed. The `dlm_perror` routine consults the `dlm_errno` global variable to determine the status of the last lock

request. The `dlm_perror` routine appends the supplied message with a colon and a printable version of the status code.

7.2.15.3. Parameters

7.2.15.3.1. message

A NULL-terminated character string.

7.2.15.4. Example

The following code fragment uses the `dlm_perror` return to print an error message if the `dlm_setnotify` routine fails. The application includes the message string "dlm_setnotify fails" for the `dlm_perror` routine to print along with the status code return by the routine.

```
#include <cluster/dlm.h>

dlm_stats_t status;
.
.
.
status = dlm_setnotify( SIGUSR1, NULL );
if ( status != DLM_NORMAL )
{
    dlm_perror("dlm_setnotify fails");
}
```

If the routine failed because the arguments passed were invalid, the following message would be printed to stderr, "dlm_setnotify fails: DLM_BADARGS."

7.2.16. `dlm_purge`

7.2.16.1. Syntax

```
dlm_stats_t dlm_purge(node_id, pid, flags);
int node_id;
```

```
int pid;
int flags;
```

7.2.16.2. Description

The `dlm_purge` routine allows a client application to purge locks in two different situations:

- Purging all the locks owned by that client. To purge its own locks, a client must call the `dlm_purge` routine with its own node ID and process ID (pid) value specified.
- Removing orphaned locks that have been left behind by clients that have terminated.

The `dlm_purge` function cannot be used to purge the locks of an active client other than the one calling the function.

7.2.16.3. Parameters

7.2.16.3.1. node_id

The ID of the node on which the locks were originated.

7.2.16.3.2. pid

This argument indicates the process ID (PID) of the application owning the locks. If you specify a PID of 0, the lock manager purges all orphaned locks for the specified node.

7.2.16.3.3. flags

None.

7.2.16.4. Status Codes

The following is an alphabetical list of all the status codes returned by the dlm_purge routine:

DLM_BADARGS

The request specified an invalid node ID.

DLM_NOLOCKMGR

The lock manager daemon is not running. If the lock manager is restarted while it is being used by a client, the next lock request returns this status.

DLM_NORMAL

The purge request completed successfully.

7.2.16.5. Example

In the following example, all the locks associated with the process are released. The example assumes that the node ID, whether local or remote, has already been obtained.

```
#include <cluster/dlm.h>
int nodeid;
.
.
.
status = dlm_purge(nodeid, get_pid(), 0 );
if( status != DLM_NORMAL )
   dlm_perror("dlm_purge");
```

7.2.17. `dlm_scnop`

7.2.17.1. Syntax

```
dlm_stats_t dlm_scnop(lockid, op_type, bit_len, in_lvb, out_lvb);
int lockid;
scn_op_t op_type;
short bit_len;
char *in_lvb;
char *out_lvb;
```

7.2.17.2. Description

The `dlm_scnop` routine manipulates a cluster-global counter called the System Commit Number (SCN). Using this routine, you can perform any of the following operations on the SCN:

- Obtain the current value of the SCN.
- Increment the current value of the SCN.
- Assign a value to the SCN.
- Assign a value to the SCN if the current value of the SCN is less than a specified value.

If the `dlm_scnop` routine returns successfully, the SCN operation is complete.

The SCN operations performed by the `dlm_scnop` routine are atomic. Accessing the SCN concurrently from different nodes or processes will not corrupt the SCN.

7.2.17.3. Parameters

7.2.17.3.1. lockid

The lock ID of the lock granted against the lock resource in which the SCN is stored, returned from a previous call to a lock open routine. You can use any lock resource to store an SCN. The counter is stored in the lock value block (LVB) of this lock resource. If there are locks on this lock resource at modes other than NL, the `dlm_scnop` routine returns the status DLM_BLOCKED.

7.2.17.3.2. op_type

The requested SCN operation. The operations are defined as follows:

SCN_CUR

Obtain the current value of the SCN. The SCN value is returned in the `out_lvb`. parameter.

SCN_INC

Increment the SCN and return the new value of the SCN. The SCN value is returned in the `out_lvb` parameter.

SCN_ADD

Add a specified value to the SCN. You specify the value to be added to the SCN in the `in_lvb` parameter. The new value of the SCN is returned in the `out_lvb` parameter.

SCN_ADJ

Set the value of the SCN to the value specified in the in_lvb parameter, if the current value of the SCN is less than the value specified. The current value of the SCN is returned in the `out_lvb` parameter.

SCN_SET

Set the value of the SCN to the value specified in the in_lvb parameter. The current value of the SCN is returned in the `out_lvb` parameter.

7.2.17.3.3. bit_len

The number of bits used to represent the range of values of the SCN. You may specify any value between 1 and 128. The maximum value of an SCN is $2^{bit\_len - 1}$. Any SCN value you specify that exceeds this maximum is ignored or zeroed. If you specify a value for this parameter, you must always specify the same value to ensure predictable results.

7.2.17.3.4. in_lvb

Pointer to the input value of the SCN.

7.2.17.3.5. out_lvb

The address into which the lock manager writes the SCN that results from the operation.

7.2.17.4. Status Codes

The following is an alphabetized list of status values returned by the `dlm_scnop` routine.

DLM_BADARGS

An argument to the `dlm_scnop` is incorrect. For example, an invalid operation type was specified.

DLM_BLOCKED

There are non-NULL locks held against the lock resource containing the SCN.

DLM_IVLOCKID

The value specified in the lockid parameter is not a valid lock ID.

DLM_NORMAL

The SCN operation completed successfully.

DLM_NOLOCKMGR

The Lock Manager is not running.

DLM_VALNOTVALID

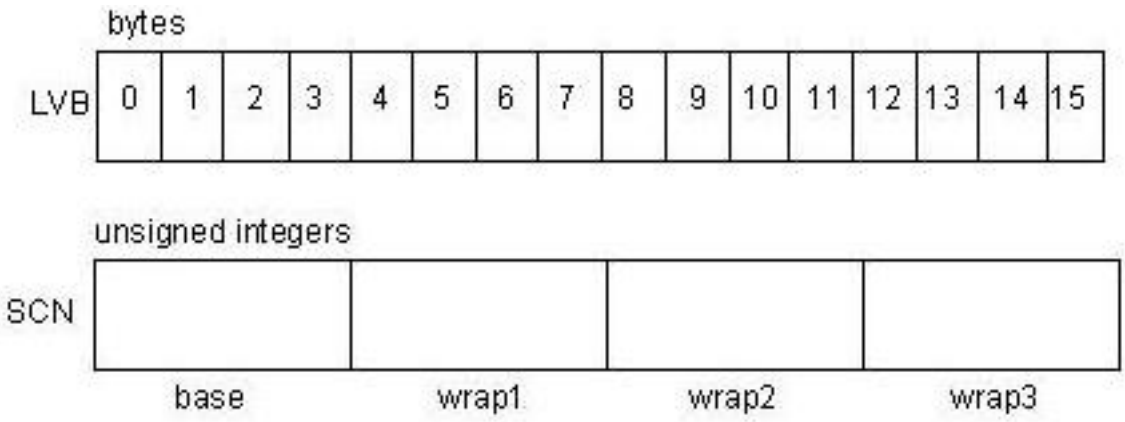
The Lock Value Block (LVB) in which the SCN is stored is marked invalid.

7.2.17.5. System Commit Number

The SCN is stored in the LVB associated with a lock resource. An LVB is an array of 16 bytes. The lock manager represents an SCN value of up to 128 bits by using four unsigned integers. These integers are the four fields contained in the scn_t structure, defined in the `/usr/include/cluster/scn.h` include file as follows:

```
typedef struct scn {
      unsigned int base;
      unsigned int wrap1;
      unsigned int wrap2;
      unsigned int wrap3;
} scn_t;
```

The following figure illustrates how the scn_t structure overlays the bytes in the LVB:

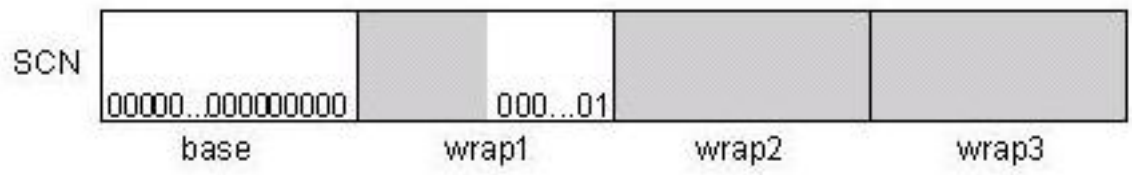


You define the range of the SCN counter by specifying, in the `bit_len` parameter passed to the `dlm_scnop` routine, how many bits are used to represent its value. The value of the `bit_len` parameter controls which bits in the four fields are used.

If you specify a `bit_len` value of 32 or less, the lock manager uses only the base field of the structure. If you increment the SCN value past the maximum value (defined as 2^{bit_len-1}), the base field in the structure wraps back to zero.

If you specify a `bit_len`. value of 64 or less, the lock manager uses the base and the wrap1 fields in the structure. The integer value in the base field overflows into the wrap1 field. The value of the SCN should be interpreted by concatenating the integer fields, and not by adding them. The entire value will wrap back to zero when it is incremented past the maximum value, determined by the `bit_len` parameter.

For example, if the wrap1 field is equal to 1 and the base is 0, then the value of the SCN is 232 or 4294967296 because the first bit of the wrap1 field is the 33rd bit of the SCN. The following figure illustrates this SCN value. The `bit_len` is set to 48. The unused bits are covered with gray.



For `bit_len` values greater than 64, the lock manager uses the wrap2 and wrap3 fields in the structure, as necessary.

7.2.17.6. Example

The following example sets the SCN to 100,000, if the current value of the SCN is less than 100,000.

```
#include <cluster/dlm.h>
#include <cluster/scn.h>    /* SCN definitions  */

struct lockstatus    lksb;
dlm_stats_t        status;
scn_t              in_scn;
scn_t             out_scn;

in_scn.base = 100000;

/* Set SCN value if less than in_scn */

status = dlm_scnop( lksb.lockid, /* Lock on SCN lock resource */
                       SCN_ADJ,  /* SCN operation
                            32,  /* bit length             */
                       &in_scn,  /* incoming SCN           */
                      &out_scn); /* returned SCN           */

if (scn_status != DLM_NORMAL)
```

```
{
      dlm_perror("Can’t get SCN.");
}
```

7.2.18. `dlm_setnotify`

7.2.18.1. Syntax

```
dlm_stats_t dlm_setnotify(signo, oldsigp);
int signo;
int *oldsigp;
```

7.2.18.2. Description

The `dlm_setnotify` routine allows a lock client to specify a signal to be delivered whenever an AST is pending.

> **Note:** This routine can be called only by user-space clients of the DLM. Kernel-space clients cannot call it.

7.2.18.3. Parameters

7.2.18.3.1. signo

This argument indicates the signal to be delivered. Specifying SIG_DFL indicates that no signal is desired and cancels any previously specified signal.

7.2.18.3.2. oldsigp

If non-NULL, this argument specifies a location that should receive the number of the existing notify signal.

7.2.18.4. Status Codes

The following is an alphabetical list of status values returned by the `dlm_setnotify` routine.

DLM_BADARGS

The specified signal is out of range. That is, the signal has a value less than zero, or greater than or equal to SIGMAX as defined in `<sys/signal.h>`.

DLM_NORMAL

The request completed successfully.

7.2.18.5. Example

For an example of using this routine, see Section 3.3.5.

7.2.19. `dlm_setresparams`

7.2.19.1. Syntax

```
dlm_stats_t dlm_setresparams(res_name, namelen, res_type, params);
```

7.2.19.2. Description

The `dlm_setresparams` routine sets the value of a lock resource's stickiness attribute.

7.2.19.2.1. res_name

The name of the lock resource.

7.2.19.2.2. namelen

The number of characters in the resource name.

7.2.19.2.3. res_type

Specifies the type of lock resource. For DLM lock resources, specify the constant DLM_RES_VMS. For UNIX lock resources, specify the constant DLM_RES_UNIX.

7.2.19.2.4. params

Address of the dlm_resparams_t structure in which you specify the value of the stickiness attribute.

7.2.19.3. Status Codes

The following is an alphabetical list of status values returned by the `dlm_setresparams` routine.

DLM_BADARGS

One of the following:

- The request specified an invalid resource type.
- The length of the resource name exceeds the limit.
- The pointer to the dlm_resparams_t structure is invalid.

DLM_BADRESOURCE

The lock resource specified is invalid.

DLM_NORMAL

The operation completed successfully.

7.2.19.4. Example

In the following code fragment, the application sets the value of the stickiness attribute.

```
#include <cluster/dlm.h>

dlm_stats_t status;

#define  NAMELEN 7

dlm_resparams_t resparams;
```

```
.
.
.
resparams.cr_valid = DLM_RES_STICKINESS;
resparams.cr_stickiness = 50;


status = dlm_setresparams( "my_lock",/* name of lock resource
                           NAMELEN,  /* length of name         */
                       DLM_RES_VMS,  /* resource type          */
                        &resparams); /* address of lock        */
                                     /* resource structure     */
if (status != DLM_NORMAL)
{
      dlm_perror("Lock resource name invalid.");
}
```