Bull

HACMP 4.4
Programming Locking Applications

AIX

ORDER REFERENCE
86 A2 59KX 02

Bull

HACMP 4.4

Programming Locking Applications

AIX

Software

August 2000

BULL CEDOC
357 AVENUE PATTON
B.P.20845
49008 ANGERS CEDEX 01
FRANCE

ORDER REFERENCE
86 A2 59KX 02

Trademarks and Acknowledgements

Year 2000

The product documented in this manual is Year 2000 Ready.

Contents

About This Guide

This book describes the Cluster Lock Manager (CLM) application programming interface (API) supplied with the High Availability Cluster Multi-Processing for AIX, Version 4.4 (HACMP for AIX) software. The lock manager supports two APIs: the CLM Locking API and the UNIX Locking API.

Who Should Use This Book

This guide is intended for application developers who want to write highly available applications for an HACMP for AIX environment. Readers of this guide should understand the C programming language and database concepts.

How to Use This Book

Overview of Contents

This book provides both conceptual and reference information. The book has the following chapters.

- Chapter 1, Cluster Lock Manager, introduces the Cluster Lock Manager.
- Chapter 2, CLM Locking Model, describes the CLM locking model.
- Chapter 3, Using CLM Locking Model API Routines, describes how to use the CLM locking model API routines in an HACMP for AIX application.
- Chapter 4, UNIX Locking Model, describes the Cluster Lock Manager's implementation of UNIX System V locks.
- Chapter 5, Using UNIX Locking Model API Routines, describes how to use the UNIX locking model API routines in an HACMP for AIX application.
- Chapter 6, Tuning the Cluster Lock Manager, describes tuning lock manager behavior to optimize lock throughput and obtain statistics about lock resource usage.
- Chapter 7, Lock Manager API Routines, provides reference information on the C language routines used to implement locking in an HACMP for AIX application.

Highlighting

The following highlighting conventions are used in this book:

Italics	Identifies new terms or concepts.
Bold	Identifies routines, commands, keywords, files, directories, menu items, and other items whose actual names are predefined by the system.
`Monospace`	Identifies examples of specific data values, examples of text similar to what you might see displayed, examples of program code similar to what you might write as a programmer, messages from the system, or information that you should actually type.

Related Publications

The following books provide additional information about HACMP for AIX:

- *Release Notes* in **/usr/lpp/cluster/doc/release_notes** describe hardware and software requirements
- *HACMP for AIX, Version 4.4: Concepts and Facilities*, order number 86 A2 54KX 02
- *HACMP for AIX, Version 4.4: Planning Guide*, order number 86 A2 55KX 02
- *HACMP for AIX, Version 4.4: Installation Guide*, order number 86 A2 56KX 02
- *HACMP for AIX, Version 4.4: Administration Guide*, order number 86 A2 57KX 02
- *HACMP for AIX, Version 4.4: Troubleshooting Guide*, order number 86 A2 58KX 02
- *HACMP for AIX, Version 4.4: Programming Client Applications*, order number 86 A2 60KX 02
- *HACMP for AIX, Version 4.4: Master Index and Glossary*, order number 86 A2 65KX 02
- *HACMP for AIX, Version 4.4: Enhanced Scalability Installation and Administration Guide, Volumes I and II*, order numbers 86 A2 62KX 02 and 86 A2 89KX 01

ISO 9000

ISO 9000 registered quality systems were used in the development and manufacturing of this product.

Ordering Publications

To order additional copies of this guide, use order number 86 A2 59KX 02.

Chapter 1 Cluster Lock Manager

This chapter introduces the HACMP for AIX Cluster Lock Manager.

An Overview of the HACMP Cluster Lock Manager

The Cluster Lock Manager provides advisory locking services that allow concurrent applications running on multiple nodes in an HACMP cluster to coordinate their use of shared resources.

Cooperating applications running on different nodes in an HACMP 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 Cluster Lock Manager are transaction-oriented, such as a database or a resource controller or manager.

Locking Models

The Cluster Lock Manager provides two distinct locking models: the CLM locking model and the UNIX System V locking model.

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

CLM Locking Model

The CLM locking model provides a rich set of locking modes and both synchronous and asynchronous execution. The CLM 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 CLM locking model, see Chapter 2, CLM Locking Model.

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 a more information about the UNIX locking model, see Chapter 4, UNIX Locking Model.

Application Programming Interfaces

The Cluster 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 CLM locking model are described in Chapter 3, Using CLM Locking Model API Routines, of this manual. The API routines that implement the UNIX locking model are described in Chapter 5, Using UNIX Locking Model API Routines.

HACMP for AIX includes two versions of the lock manager API libraries: one for single-threaded (non-threaded) applications (**libclm.a**) and one for multi-threaded applications (**libclm_r.a**).

Cluster 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. This 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 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.

Support for HC Daemon

The HC daemon provides support needed to run Oracle's Distributed Fault Tolerant Lock Manager as part of Oracle Parallel Server. HACMP reports and reacts to the presence or loss of a processor in the cluster. The HC daemon extends this functionality to process membership information. It allows peer processes in a cluster, such as the instances of the Oracle Fault Tolerant Lock Manager on each node, to be informed of the presence or loss of other instances across the cluster.

The HC daemon accepts a socket connection from a client process. It keeps all clients informed of process membership, broadcasting messages regarding the addition or loss of a client to all clients across the cluster. The HC daemon regularly exchanges heartbeat messages with its client. This allows it to detect the loss of a client for reasons other than loss of a node.

Installing the HC daemon puts the following entry in the **/etc/inittab** file:

```
hc:2:respawn:/usr/lpp/csd/bin/hacmp_hc_start # start hc daemon
```

This entry directs init to run the shell script **/usr/lpp/csd/bin/hacmp_hc_start** at system initialization time, and re-run it should the HC daemon ever exit.

This shell script constructs the file **/usr/lpp/csd/bin/machines.1st**, which the HC daemon uses to determine the TCP/IP addresses needed to communicate with other instances in the cluster.

In the event that a node joins or leaves the cluster, the HACMP cluster manager runs shell scripts that react to these events. These shell scripts call other scripts associated with the HC daemon and which pass updated cluster membership to the HC daemon. The HC daemons in turn inform their clients and any instances of the Oracle Distributed Fault Tolerant Lock Manager.

```
Event                 Shell Script

Node Up               /usr/lpp/csd/bin/hacmp_vsd_up1
Node up complete      /usr/lpp/csd/bin/hacmp_vsd_up2
Node down             /usr/lpp/csd/bin/hacmp_vsd_down1
node down complete    /usr/lpp/csd/bin/hacmp_vsd_down2
```

Restrictions

Important restrictions on the use of the HC daemon:

- The HC daemon is supported only on nodes that are part of a concurrent resource group.
- Each node on which the HC daemon is installed must have a service interface defined.
- The HC daemon should not be loaded on nodes which are part of an SP system on which the Recoverable Virtual System Disk (RVSD) is installed. The RVSD facility provides its own version of the HC daemon.

See the applicable Oracle documentation for information on which levels of Oracle Parallel Server provide Oracle's Distributed Fault Tolerant Lock Manager.

Chapter 2 CLM Locking Model

This chapter presents the concepts you need to understand to use CLM locks effectively in an application. Chapter 3, Using CLM Locking Model API Routines, describes how to use the CLM locking model API routines to implement locking in an application.

Overview

In the CLM 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; however, it does not have to. 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:

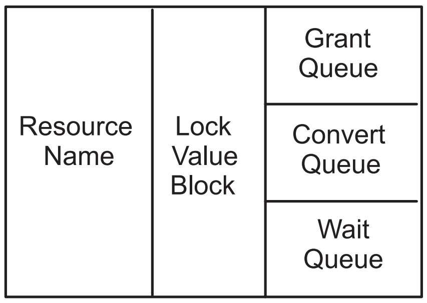
- Lock resources, including lock value blocks and lock queues
- Locks, including lock modes and lock states
- Deadlock, including transaction IDs and lock groups.

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 illustrates a lock resource.



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.

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, Using CLM Locking Model API Routines, for information about using the lock value block.

Lock Resource Queues

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

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	Contains all granted locks that have subsequently 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	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 Lock States on page 2-5.

Locks

In the CLM 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

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 Cluster 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.

Lock Mode Severity

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

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

Lock Mode Compatibility

Lock mode compatibility determines whether two locks can be granted simultaneously on a particular lock resource. Because of their restrictiveness, 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 at 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.

This table presents a mode compatibility matrix.

Mode of Requested Lock	Mode of 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

Lock Mode Compatibility

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 placeholders 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.

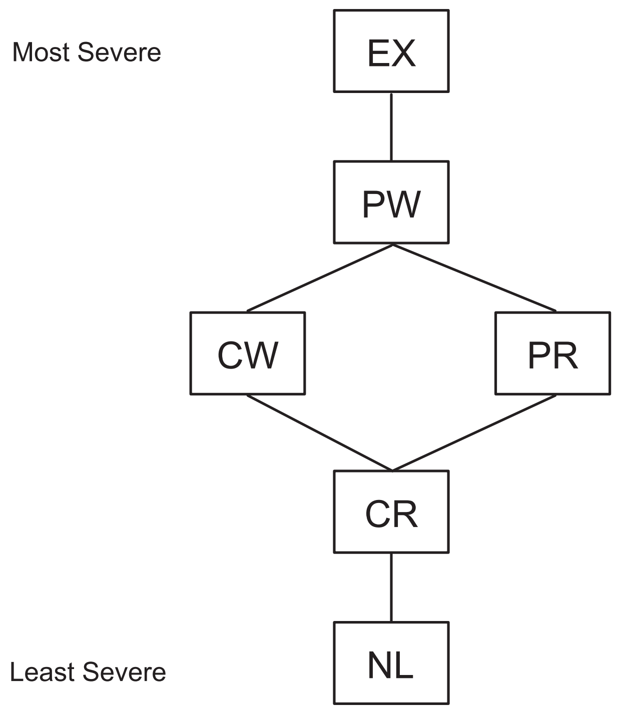
CW mode locks allow unprotected read and write operations. CW mode locks are compatible with NL mode locks, CR read mode locks, and other CW mode locks.

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.



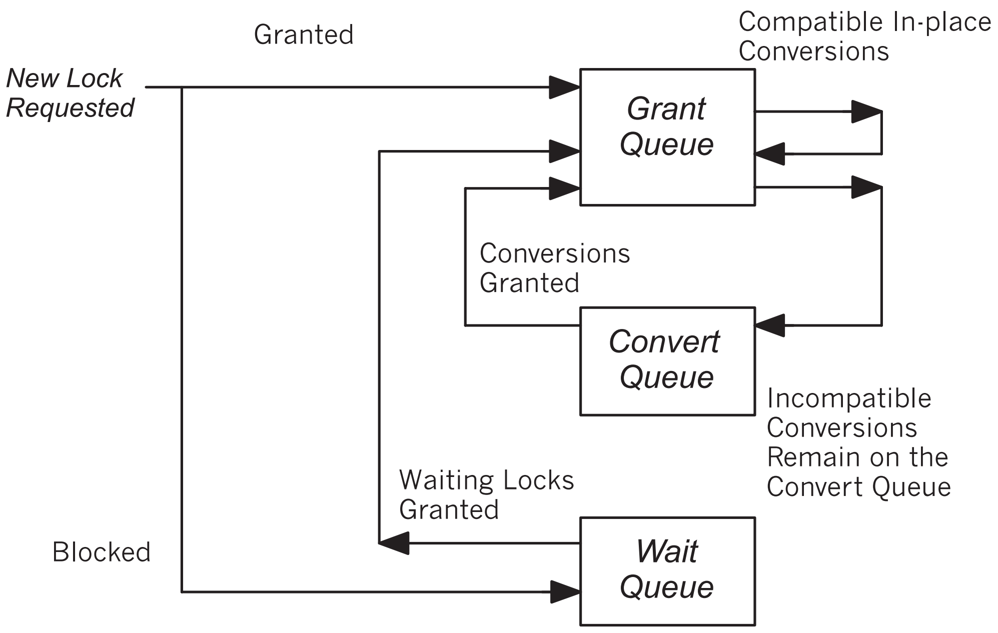
Lock Mode Severity

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.



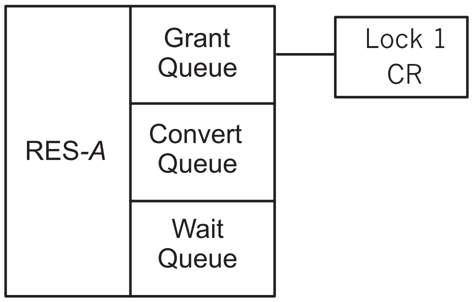
Lock Queues

The following sections provide more information about each state. See Interaction of Queues on page 2-12 for a detailed example of the lock state transitions.

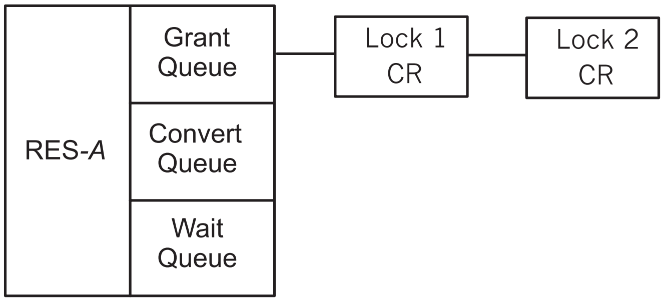
Granted

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.



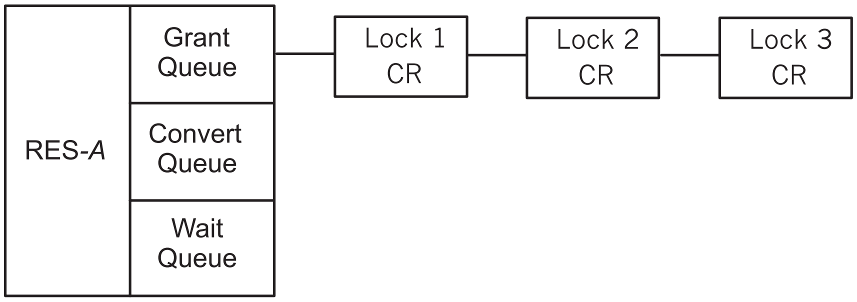
Leaving the Grant Queue

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 cancelled). If the orphaned lock is on the wait queue, the lock manager ignores its orphanable state and removes it. For more information about creating orphan locks, see Requesting Persistent Locks on page 3-8.

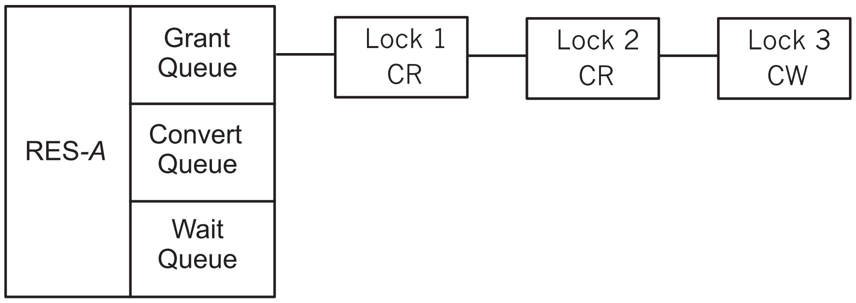
Converting

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 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.



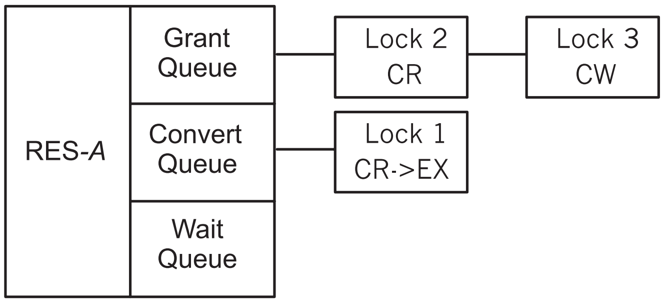
If you request a conversion of Lock 3 from CR mode to CW mode, the lock manager can grant the request because CW mode is compatible with CR mode and there are no lock conversion requests on the convert queue. The following illustrates the state of the lock queues after this request.



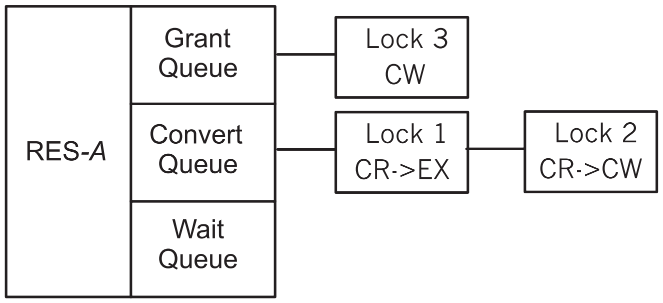
A lock conversion request that cannot be granted transitions into *converting* state. The lock manager moves locks in converting state from the grant queue to the end of the convert queue. Locks in the converting state retain the lock mode that they held on the grant queue.

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 Lock1 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.



Leaving the Converting State

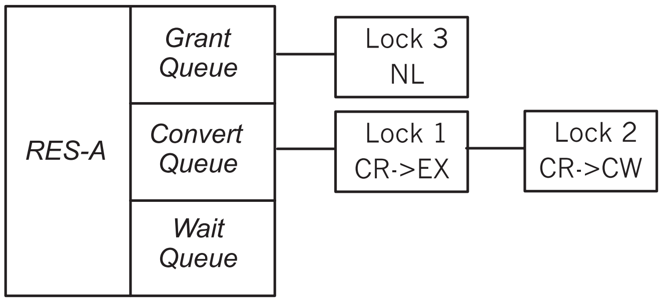
A lock can leave the converting state if any of the following conditions are met:

- 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.

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.



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.

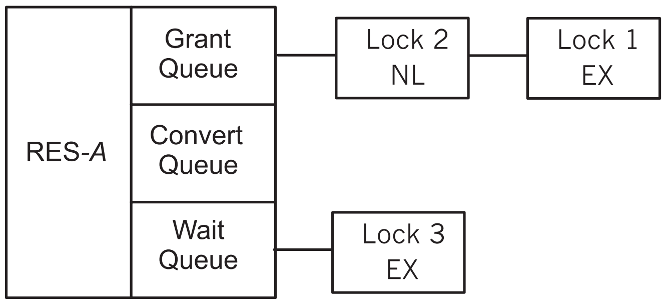
The previous example illustrates conversion deadlock. Even after the down-conversion of Lock 3 to NL mode, Lock 1 cannot be granted because it is blocked by Lock 2, also on the convert queue. Lock 1 cannot convert to EX mode because Lock 2 is still granted at CR mode, which is incompatible with EX mode. Thus, Lock 1 is blocked by Lock 2 and Lock 2 is blocked by Lock 1. For more information about conversion deadlock, see Conversion Deadlock on page 2-15.

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 Avoiding the Wait Queue on page 3-7.)

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 grant lock (Lock 1 at EX mode). The lock manager adds this lock request to the end of the lock resource’s wait queue.

This figure illustrates the lock resource’s queues after this request.

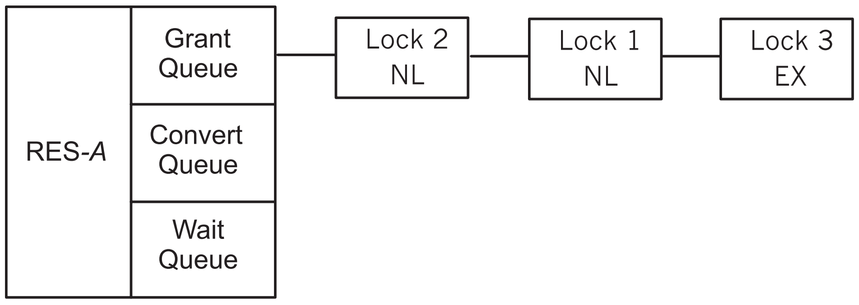


A lock can leave the wait queue if any of the following conditions are met:

- 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.

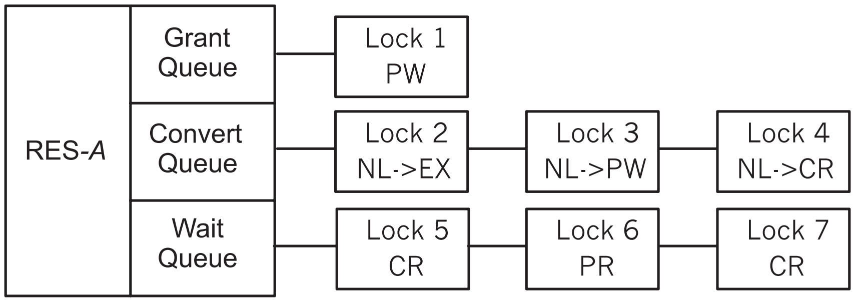
Thus, a lock request can become blocked only as the result of a lock request, but it can unblock as a result of the release or conversion of some other lock. (An exception is made in the case of deadlock. See Deadlock on page 2-14.)

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.



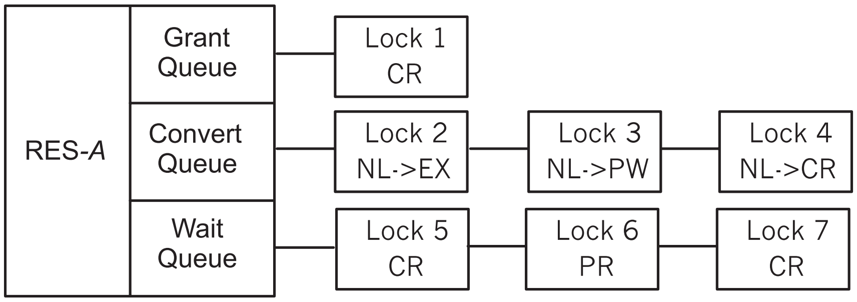
Interaction of Queues

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.

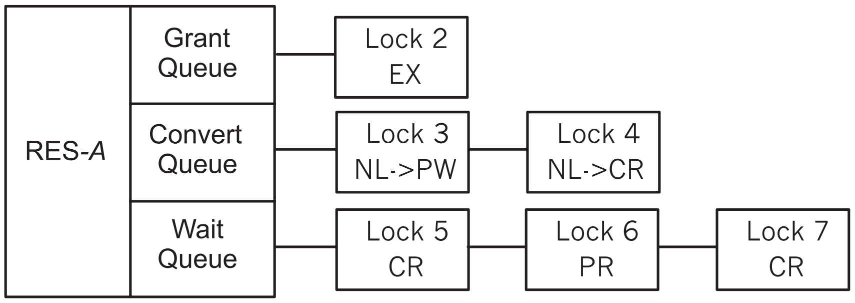


If you request a down-conversion of Lock 1 from PW to CR, the lock manager can grant the conversion request because a CR lock is compatible with the mode of the most restrictive currently granted lock. (Lock 1 itself is the most restrictive currently granted lock; a lock cannot block itself.) Note that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



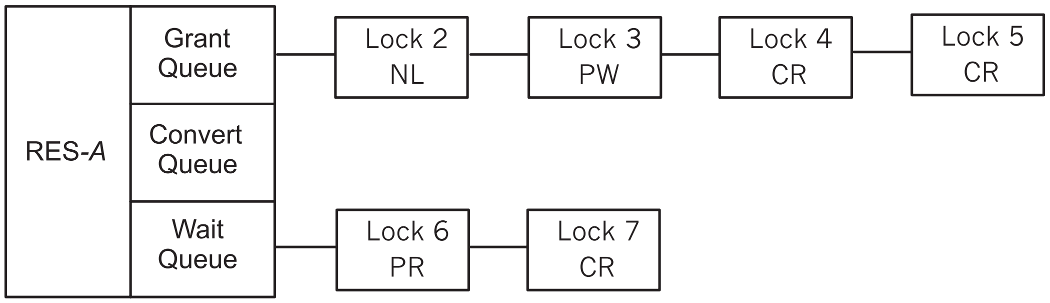
If you release Lock 1, the lock manager can grant Lock 2, the EX lock waiting at the head of the conversion queue. Because the mode requested by Lock 3 is not compatible with an EX mode lock, no other request on the convert queue can be granted. The following figure illustrates the lock resource's queues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



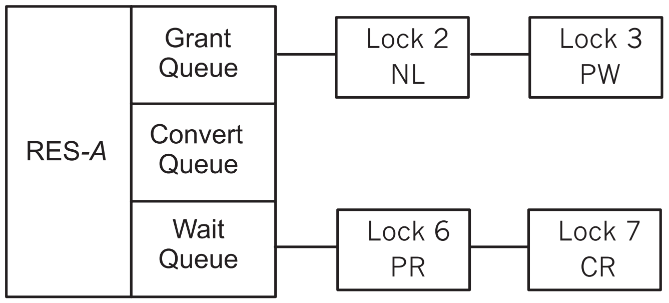
If you request a down-conversion of Lock 2 from EX to NL, the lock manager grants the conversion in-place because a NL lock is compatible with the mode of the most restrictive currently granted lock (EX). The lock manager checks the convert queue to see if the change allows any blocked conversion requests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

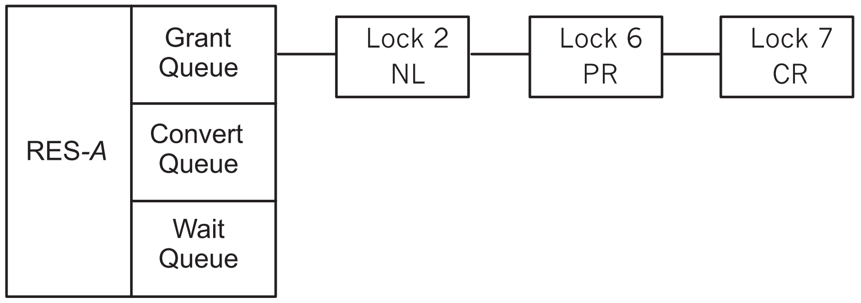
The following figure illustrates the lock resource's queues after the conversion operation of Lock 2.



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.

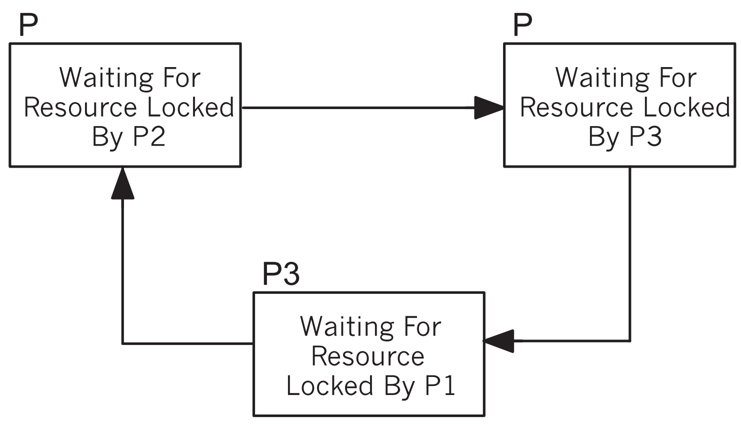


Deadlock

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.

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 is illustrated in the following figure.


P
Waiting For Resource Locked By P2
P
Waiting For Resource Locked By P3
P3
Waiting For Resource Locked By P1

Deadlock

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.

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.

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 CLM_DEADLOCK and queues for execution the AST routine associated with the request. (For more information about how the lock manager returns the status of lock requests, see Obtaining the Status of a Lock Request Synchronously on page 3-4.)

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 reacquiring them should eliminate the deadlock condition.

Transaction IDs

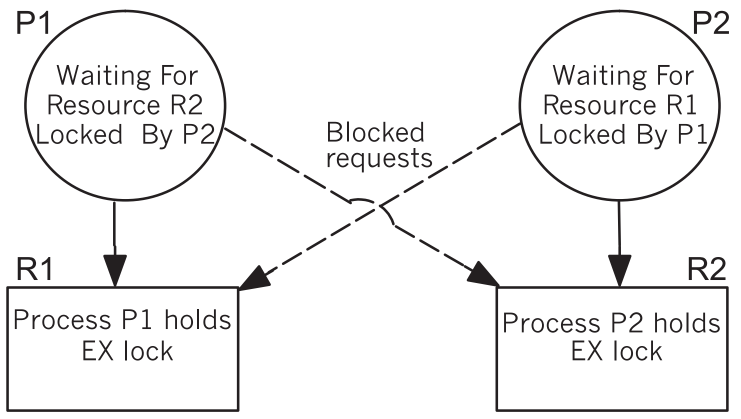
By canceling one of the requests causing a deadlock, the lock manager prevents clients contending for the same lock resources from blocking each other indefinitely. Additionally, the lock manager supports transaction IDs, a mechanism clients can use to improve application throughput by diminishing the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

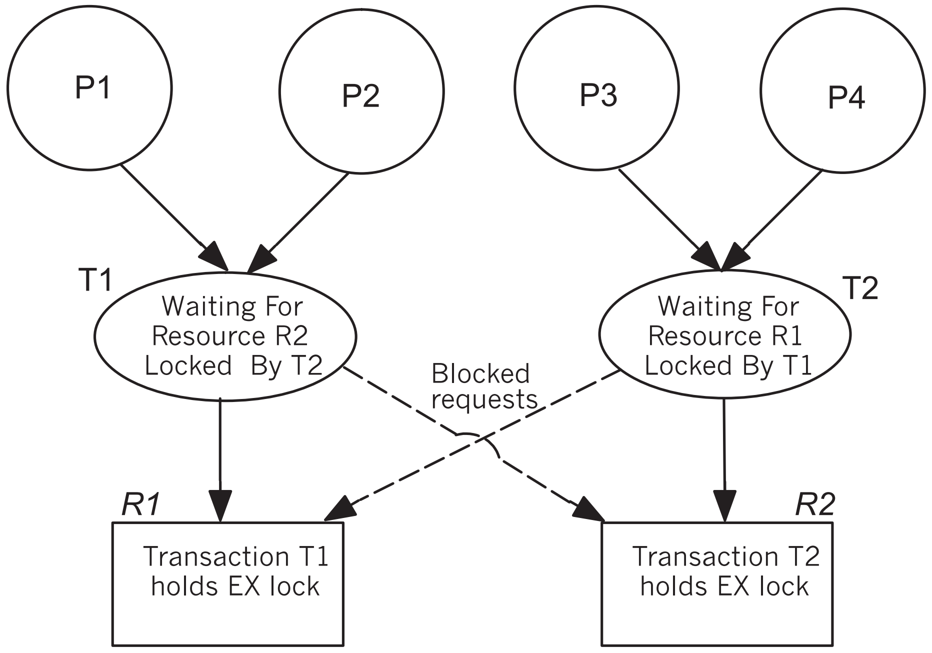
Consider the following example: Process P1 holds an exclusive lock on Resource R1 and requests an exclusive lock on Resource R2. Process P1 will not release the lock on Resource R1 until the lock manager grants the lock on Resource R2. Process P2, meanwhile, holds an exclusive lock on Resource R2 and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


P1
Waiting For Resource R2 Locked By P2
P2
Waiting For Resource R1 Locked By P1
Blocked requests
R1
Process P1 holds EX lock
R2
Process P2 holds EX lock

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 example above:

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.


P1
P2
P3
P4
T1
Waiting For Resource R2 Locked By T2
T2
Waiting For Resource R1 Locked By T1
Blocked requests
R1
Transaction T1 holds EX lock
R2
Transaction T2 holds EX lock

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

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 CLM Locking Model API Routines

This chapter describes how to use the CLM locking model API routines in an HACMP for AIX application. Chapter 7, Lock Manager API Routines, provides reference information on the routines discussed in this chapter.

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 CLM 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 **clmlock** or **clmlockx** routine.

The CLM locking model API also includes routines that help applications perform ancillary tasks related to manipulating locks. For example, the CLM 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.

Prerequisites

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

Header Files

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

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

The **/usr/include/cluster/clm.h** file defines the constants, data structures, status codes, and flags used by the CLM locking model API.

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

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

The **/usr/include/cluster/scn.h** file defines the constants, data structures, and status codes used by the **clm_scnop** routine.

Library Files

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

Single-threaded Applications

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

```
-lclm -lclstr
```

The **libclm.a** library contains the routines that support the Cluster 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 (Clinfo), you must also include the **libcl.a** library.

Multi-threaded Applications

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

```
-lclm_r -lclstr_r
```

The **libclm_r.a** library contains the routines that support the Cluster Lock Manager. The **libclstr_r.a** library contains the routines that support the Cluster Manager.

If your application uses the services of Clinfo, you must also include the **libcl_r.a** library.

Data Structure

The CLM 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/cluster/clm.h** include file, has the following structure:

```
struct lockstatus {
         clm_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/cluster/clm.h** include file.
lockid	Contains the lock ID the lock manager assigned to this lock request.
value	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/cluster/clm.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.

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 Cluster Lock Manager supports both asynchronous and synchronous lock routines.

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:

clmlock	Makes an asynchronous (non-blocking) request for a lock on a lock resource or converts an existing lock to a different mode.
clmlockx	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 **clmlockx** 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 **clmlock** routine to request a CR mode lock on a lock resource named RES-A.

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

clm_stats_t status;
struct lockstatus lksb;        /* lock status block */
extern void ast_func();
.
.
.
status = clmlock( 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 != CLM_NORMAL )
{
    clm_perror( "clmlock");
}
```

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 **clmlock** and **clmlockx** 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 3, Using CLM Locking Model API Routines, for a complete list of all possible status codes returned by the **clmlock** and **clmlockx** routines.

Obtaining the Status of a Lock Request Synchronously

Using the asynchronous lock routines, you can request a synchronous return of your lock request by specifying the LKM_SYNCSTS flag. 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 **clmlock** or **clmlockx** routines return the status code CLM_SYNC, indicating success, instead of the CLM_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 CLM_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, use the **clmlock_sync** or **clmlockx_sync** routines, described in the next section, Requesting Locks Synchronously.

Requesting Locks Synchronously

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

The routines that request a lock synchronously are:

clmlock_sync	Requests a lock and waits for a return or converts an existing lock to a different mode.
clmlockx_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 **clmlockx_sync** routine, a pointer to an eight-byte transaction ID that indicates the lock is owned by a transaction or group

Triggering AST Routines

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:

1. Use the **clm_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 AIX.
2. 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 **clmlock** 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 Sample Locking Application on page 3-6 (below).

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 **clmlock** and **clmlockx** routines or the bastarg parameter to the **clmlock_sync** and **clmlockx_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 **clmlock** 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 **clmlock** 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.

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 <cluster/clm.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 */
```

```
clm_stats_t status;
int stat;
char *msg; /* for printable status code */
main(argc, argv)
    int argc;
    char *argv[];
{
    int astarg = 0;  /* astarg parameter */
    status = clm_setnotify( SIGUSR1, NULL );
    if( status != CLM_NORMAL )
    {
         clm_perror("clm_setnotify");
    }

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

    status = clmlock( 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 != CLM_NORMAL )
    {
        clm_perror( "clmlock");
    }
}

/* 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 = clm_errmsg( lksb[*astarg].status );
   printf("status= %s; astarg passed = %d",msg,*astarg);
}
```

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 CLM_NORMAL and the AST is queued with the status CLM_NOTQUEUED in the status field of the lock status block.

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 <cluster/clm.h>

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

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

status = clmlock( 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 != CLM_NORMAL )
{
    clm_perror( "clmlock");
}
```

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.

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.

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 lookup 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 lookup step may require communication with a remote node. By bypassing the directory lookup, the lock manager reduces the network overhead associated with the lock request, improving performance.

Guidelines for Use

Use caution when creating local lock resources. While eliminating the lock resource directory lookup 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.

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.

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 **clmunlock** routine. When releasing a lock, you supply the following information:

- A valid lock ID
- Optionally, a pointer to a lock value block
- Flags

  The flag you specify depends on the type of operation you are requesting. The following summarizes the options 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 **CLM_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 **CLM_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 <cluster/clm.h>

clm_stats_t status;

struct lockstatus lksb[MAXLOCKS]; /* lock status block */
int index=0;
.
.
.
status = clmunlock(lksb[index].lockid, 0, 0);
if (status != CLM_NORMAL)
{
     clm_perror("Unlock failed");
}
```

Purging Locks

The CLM API includes the **clm_purge** routine to facilitate releasing locks. The **clm_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, **clm_purge** only affects orphaned locks.

Manipulating the Lock Value Block

Every lock resource includes 16 bytes of storage, called a lock value block, that applications can use to store data. You cannot assign a value to an 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.

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 **clmunlock** 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 <cluster/clm.h>

struct lockstatus lksb;

char valblk[16];

strcpy(valblk,"my lvb");

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

if ( status != CLM_NORMAL )
{
    clm_perror("clmlock");
}

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 <cluster/clm.h>

struct lockstatus lksb;

char valblk[16];

strcpy(valblk, "my lvb");

lksb.valblk = &valblk;

status = clmlock( 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 != CLM_NORMAL )
{
    clm_perror( "clmlock");
}
```

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 **clmunlock** 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 <cluster/clm.h>

struct lockstatus lksb;


status = clmlock( 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 != CLM_NORMAL )
{
    clm_perror( "clmlock");
}
```

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.

Implementing a Local Disk Cache

For example, 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.

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 **clm_scnop** routine, that allows you to manipulate the LVB associated with a lock resource directly.

Using the standard CLM 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 **clm_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 **clm_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 **clm_scnop** routine retrieves or attempts to change the value of an SCN marked invalid, it returns the status **CLM_VALNOTVALID**. To reset an invalidated SCN, call the **clm_scnop** routine specifying the **SCN_SET** operation.

Handling Returned Status Codes

The global variable **clm_errno** is declared as the enumerated type **clm_stats_t**, which is defined in the **/usr/include/cluster/clm.h** include file. Specify it in your application as follows:

```
clm_stats_t clm_errno;
```

The enumerated type **clm_stats_t** is made up of all the status codes returned by the lock manager API routines, both the CLM locking model and UNIX locking model routines. As with the standard AIX global variable **errno**, the value of **clm_errno** is set by the last lock operation.

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

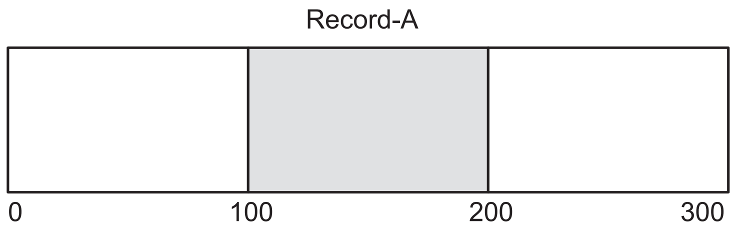
clm_perror	Writes a message you specify to standard error. Appended to the message is the status code returned by the last CLM API routine to execute. The **clm_perror** routine obtains the value of the status code from the global variable **clm_errno**.
clm_errmsg	Returns a pointer to a printable version of the CLM API status code. The status codes that make up the **clm_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 **clm_perror** routine).

Chapter 4 UNIX Locking Model

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

Lock Regions

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.



UNIX Lock 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.

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.

Shared

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

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.

Lock States

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

Granted

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

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 an HACMP for AIX application. Chapter 7, Lock Manager API Routines, provides reference information on the routines discussed in this chapter.

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 **clmregionlock** routine.

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

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.

Header Files

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

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

The **/usr/include/cluster/clm.h** file defines the constants, data structures, status codes, and flags used by the CLM locking model API.

To use the **clmregionlock** routine, you must also include the following system include file:

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

Library Files

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

Single-threaded Applications

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

```
-lclm -lclstr
```

The **libclm.a** library contains the routines that support the Cluster 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 (Clinfo), you must also include the **libcl.a** library.

Multi-threaded Applications

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

```
-lclm_r -lclstr_r
```

The **libclm_r.a** library contains the routines that support the Cluster Lock Manager. The **libclstr_r.a** library contains the routines that support the Cluster Manager.

If your application uses the Clinfo services, you must also include the **libcl_r.a** library.

Data Structure

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

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

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

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.

clmregister Routine

Before locking a lock resource, you must first register, or create, that lock resource. You register a lock resource by calling the **clmregister** 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.

Resource Handles

The **clmregister** 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.

Locking a Lock Resource

Use the **clmregionlock** routine to lock a lock resource region. Supply the following information with the **clmregionlock** routine:

- The resource handle returned from an earlier call to **clmregister** 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 CLM_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 **clmregionlock** 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.

Unlocking a Resource

Use the **clmregionlock** 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 **clmregionlock** 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 **CLM_NORMAL**.

Handling Returned Status Codes

The global variable **clm_errno** is declared as the enumerated type **clm_stats_t**, which is defined in the **/usr/include/cluster/clm.h** include file. Specify it in your application as follows:

```
clm_stats_t clm_errno;
```

The enumerated type **clm_stats_t** is made up of all the status codes returned by the lock manager API routines, both the CLM locking model and UNIX locking model routines. As with the standard AIX global variable **errno**, the value of **clm_errno** is set by the last lock operation.

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

clm_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 **clm_perror** routine obtains the value of the status code from the global variable **clm_errno**.
clm_errmsg	Returns a pointer to a printable version of the UNIX API status code. The status codes that make up the **clm_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 **clm_perror** routine).

Purging Locks

The Cluster Lock Manager **clm_purge** routine can be used to facilitate releasing UNIX locks. The **clm_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 Tuning the Cluster Lock Manager

This chapter describes tuning lock manager behavior to optimize lock throughput. The chapter also describes how to obtain statistics about lock resources.

Overview

To make optimal use of system resources and maximize performance, the Cluster Lock Manager can dynamically change the node on which a lock resource is mastered. By moving a lock resource master, the lock manager can avoid the overhead of continually accessing a remote lock master across the network. However, moving the master copy of a lock resource from one node to another incurs its own overhead. To avoid unnecessary migrations, the lock manager considers several factors before moving a lock resource master which you can tune to obtain optimal performance.

The parameters that you can use to tune lock migration throughout the cluster are the following:

- Specifying the frequency of migration evaluations, that is, how often the lock manager checks if migration is needed
- Specifying how much the lock manager should consider historical access patterns in its calculations

In addition to these two cluster-wide tuning parameters, the lock manager also allows applications to request that certain lock resources stay on a particular node by specifying the stickiness value of the lock resource.

Migration Evaluation Frequency

To determine when to move a lock resource master, the lock manager monitors lock resource access patterns. These migration evaluations are triggered when the total number of accesses to a lock resource reaches a threshold.

You can control how often the lock manager performs these migration evaluations by specifying this *evaluation threshold*. If you specify a high value for the migration evaluation threshold, the lock manager performs fewer lock migrations. While this can reduce the overhead incurred by the evaluations, it makes the lock manager less responsive to changing access patterns. For more information, see Specifying the Frequency of Migration Evaluations on page 6-2.

Historical Access Patterns

The lock manager moves a lock resource master when access patterns indicate that a remote node is using the lock resource more than the local node. However, this access pattern may be atypical. The lock manager may move a lock resource master only to have to move it back again at the next evaluation. To avoid moving lock resource masters to and from the same node, the lock manager includes previous access patterns in its migration calculations. In this way, the lock manager can balance the impact of atypical patterns and avoid spurious migrations.

You can specify how much these historical access patterns influence the migration calculation by setting the rate at which the lock manager discounts these values. This value, called the *decay rate*, specifies the percentage of the past access rates the lock manager includes in its migration calculations. If you specify a high decay rate, the lock manager puts more emphasis on past access patterns when making a migration evaluation. Emphasizing past access patterns lessens the impact of current access patterns and can make lock resource master migrations less likely. For more information, see Specifying the Decay Rate on page 6-5.

Stickiness Attribute

To avoid performing unnecessary lock resource master migrations, the lock manager gives the local node the advantage in migration calculations by adding 50% of the sum of the remote access rates to the local access rate. This addition tends to make the local access rate higher than any remote access rate, preventing migrations.

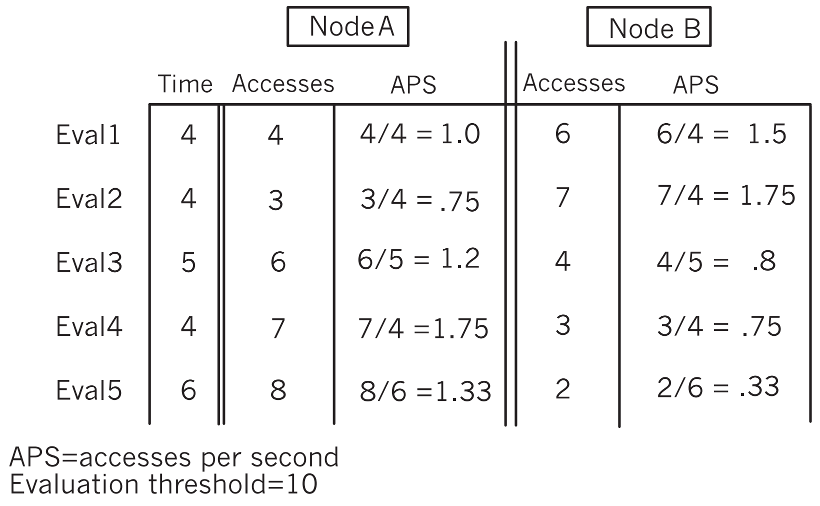
For individual lock resources, you can control how much of an advantage the lock manager gives the local node by specifying a value for the *stickiness* attribute of a lock resource. The stickiness attribute specifies what percentage of the sum of the remote access rates the lock manager adds to the local access rate when it performs migration evaluations. You can specify any value between 0 and 100. A stickiness value of 100 specifies that the lock manager add the sum of all remote access rates to the local access rate, guaranteeing that the local access rate will always at least match the highest remote access rate, preventing lock resource master migrations.

Note that the stickiness attribute is not a global parameter; it affects only a single lock resource. For more information, see Specifying the Stickiness Value of a Lock Resource on page 6-9.

Specifying the Frequency of Migration Evaluations

To control when the lock manager performs migration evaluations, specify the total number of accesses that trigger an evaluation in the evaluation threshold parameter. You can specify any positive integer as the value of this parameter.

For example, consider a cluster made up of two nodes (A and B) that share access to a common lock resource. The table summarizes the lock access patterns that result if you specify an evaluation threshold of 10.



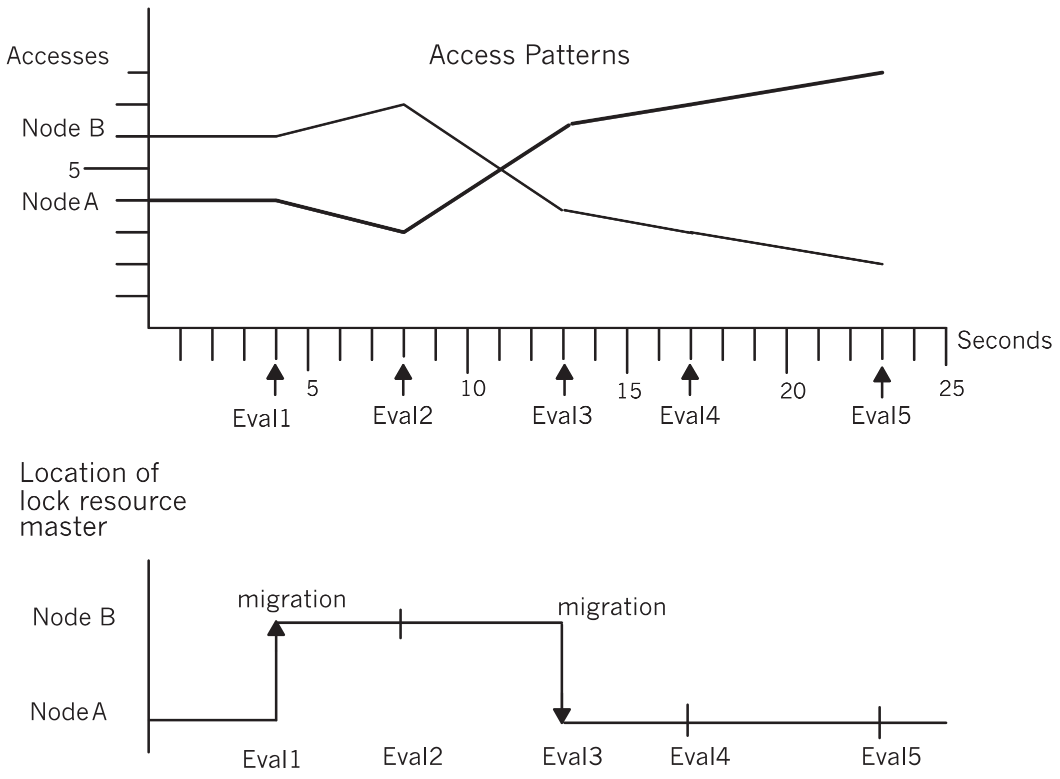
		Node A		Node B	
	Time	Accesses	APS	Accesses	APS
Eval1	4	4	4/4 = 1.0	6	6/4 = 1.5
Eval2	4	3	3/4 = .75	7	7/4 = 1.75
Eval3	5	6	6/5 = 1.2	4	4/5 = .8
Eval4	4	7	7/4 =1.75	3	3/4 = .75
Eval5	6	8	8/6 =1.33	2	2/6 = .33

APS=accesses per second
Evaluation threshold=10

Sample Access Patterns

The following figure graphically presents the access data in the above table. The figure also includes a graph that indicates when these access patterns would cause lock resource master migrations. In the example, the lock resource master starts out on Node A and migrates to Node B at the first evaluation point (Eval1) because the access rate from Node B is greater than the access rate from Node A. The lock resource master stays on Node B until the third evaluation point (Eval3) when accesses from Node A exceed those from Node B.

Note: This example is provided to illustrate the concept of lock resource master migration. The example does not include the other factors, such as historical access patterns, that the lock manager also considers when making a migration decision.



Lock Resource Master Migrations Caused by Sample Access Patterns

Using SMIT to Specify the Evaluation Threshold

You can specify the evaluation threshold using the HACMP for AIX SMIT menu. From the main menu, select **Cluster Configuration** > **Cluster Resources** > **Change/Show Cluster Lock Manager Resource Allocation**. SMIT displays a screen with the following two options:

Lock Tuning Statistic Recalculation Rate	Enter the value of the evaluation threshold.
Lock Tuning Statistic Decay Rate	Specify a decay rate value (see page 6-8). If you have no value to enter, leave the default value, 0.875.

You can also specify this value by using the **-r** flag with the **cllockd** command. For more information, see the man page for the **cllockd** command.

You can also specify this value using the **startsrc** command by using the **-a** flag to pass the **-r** flag as a subsystem argument string. The **startsrc** command passes the **-r** argument to the lock manager when it starts the subsystem. The first node to complete its node_up processing sets the value of the global tuning parameter. Once this node is up, you must use the **clm_setglobparams** routine to change the value of this global tuning parameter.

Note: Even if you are updating only a single field on the Change Resource Allocation screen, you must enter a value for each field on the screen. You can enter the default values shown on the screen above for the fields you are not updating.

Specifying the Evaluation Threshold from within an Application

Use the following routines to read or set the value of the evaluation threshold from within an application:

clm_getglobparams	Retrieves the current settings of the lock manager global parameters.
clm_setglobparams	Assigns a value to the lock manager global parameters.

Both routines accept a single argument: the address of a **clm_globparams_t** structure. When used with the **clm_getglobparams** routine, the lock manager writes the current values of the parameters in this structure. When used with the **clm_setglobparams** routine, applications use this structure to specify the desired values of the global parameters. This data structure has the following format:

```
typedef struct clm_globparams {
      unsigned  cg_valid;
      unsigned cg_recalc_time;
      float          cg_decay_rate;
} clm_globparams_t;
```

The content of the fields varies depending on which routine they are used with.

cg_valid	Indicates which fields in the **clm_globparams_t** structure contain valid data. This field is only used when specifying the value of the global parameters with the **clm_setglobparams** routine to specify the values of global parameters. When the **clm_getglobparams** routine returns successfully, both fields can be assumed to be valid.
cg_recalc_time	Used to specify the desired evaluation threshold value, when used with the **clm_setglobparams** routine. Contains the current value of the evaluation threshold, after the **clm_getglobparams** routine returns successfully.

Specifying the Decay Rate

To determine how much emphasis the lock manager puts on historical access patterns, specify the rate at which the lock manager discounts these values in the decay rate parameter. This value indicates the percentage of the past access patterns the lock manager includes in its lock resource master migration calculations. You can specify any value between 0.0 and 1.0 for this parameter.

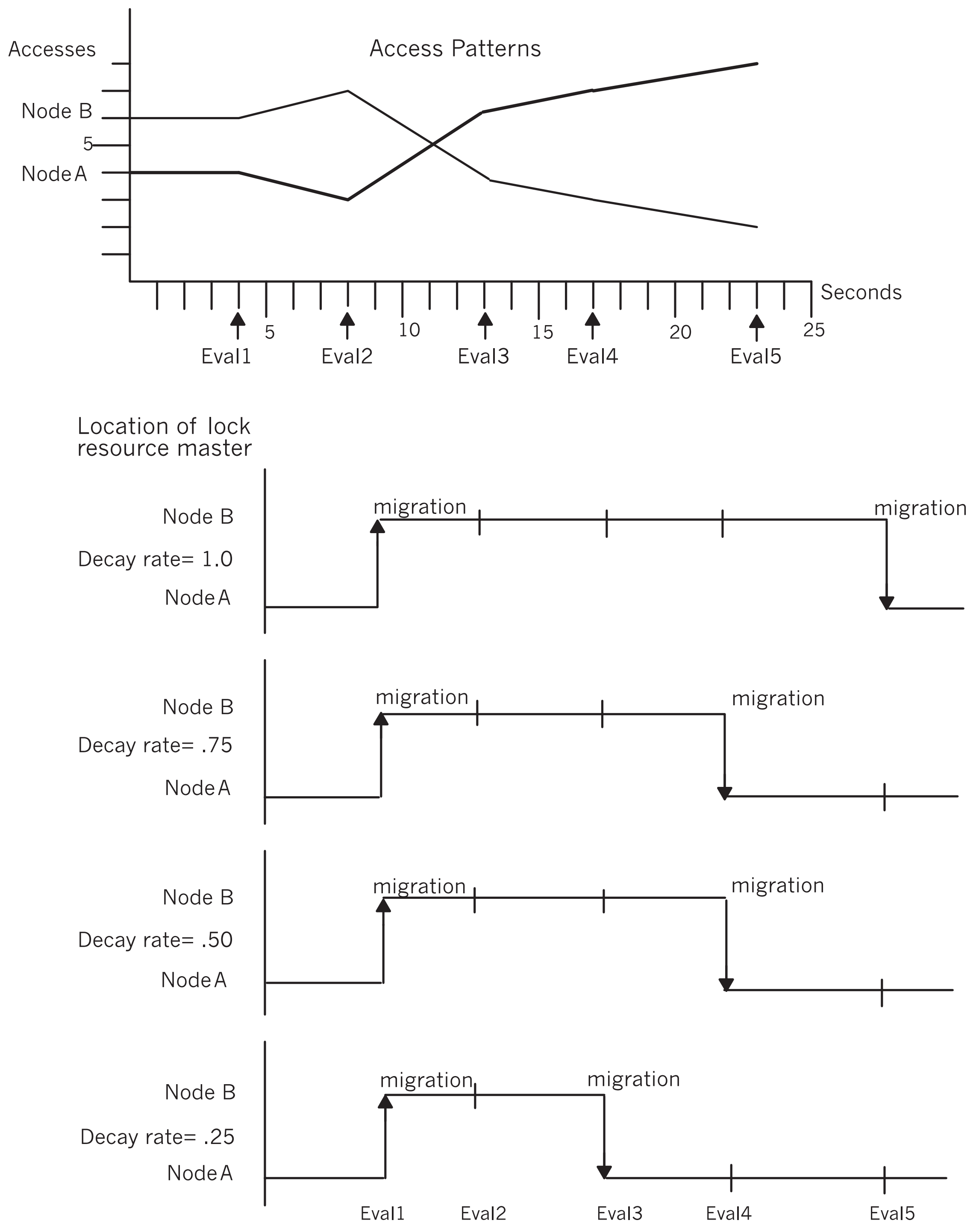
To illustrate decay rates, the following table presents the same access patterns as the table on page 6-3. This time, however, historical access patterns are factored in at various decay rates (.25, .50, .75, and 1.0). As in the earlier table, the evaluation threshold is set at 10 accesses.

		Node A						Node B					
				Decayed APSs						Decayed APSs			
	Time	accesses	APS	1.0	.75	.50	.25	accesses	APS	1.0	.75	.50	.25
Eval1	4	4	1.0	0	0	0	0	6	1.5	0	0	0	0
Eval2	4	3	.75	1.75	1.5	1.25	1.0	7	1.75	3.25	2.88	2.5	2.13
Eval3	5	6	1.2	2.95	2.33	1.83	1.45	4	.80	4.05	2.96	2.05	1.33
Eval4	4	7	1.75	4.7	3.49	2.66	2.11	3	.75	4.80	2.97	1.78	1.08
Eval5	6	8	1.33	6.03	3.95	2.66	1.86	2	.33	5.13	2.56	1.22	.60

Sample Access Patterns Decayed at Various Rates

For example, consider evaluation point 3 (Eval3). The lock resource master is on Node B. The access rate on Node A is 1.2; the access rate on Node B is .8. To these current access rates, the lock manager adds various portions of the past access rates, determined by the various decay rates. At evaluation point 3 on Node A, at a decay rate of .50, the lock manager adds the past access rate, reduced by 50%, to the current access rate, resulting in the decayed access rate of 1.83.

To show how these decay rates affect lock resource master migration, the following figure graphically presents these access patterns and the migrations they trigger at each decay rate. Note how high decay rates tend to make lock resource master migrations happen less frequently.



Lock Resource Master Migrations Caused by Decayed Sample Access Patterns

Using SMIT to Specify the Decay Rate

You can specify the decay rate using the HACMP for AIX SMIT menu. From the main menu, select **Cluster Configuration > Cluster Resources > Change/Show Cluster Lock Manager Resource Allocation**. SMIT displays a screen with the following two options:

Lock Tuning Statistic Recalculation Rate	Enter the rate or leave the default value of 9999999 in the field. (See page 6-4.)
Lock Tuning Statistic Decay Rate	Specify a decay rate value between 0.0 and 1.0. If you have no value to enter, leave the default value, 0.875.

Assign the value of the decay rate to the **Lock Tuning Statistic Decay Rate** field. Specify a value between 0.0 and 1.0.

You can also specify the lock resource migration decay rate using the **-D** flag to the **cllockd** command. For more information, see the manpage for the **cllockd** command.

You can also specify this value using the **startsrc** command by using the **-a** flag to pass the **-D** flag as a subsystem argument string. The **startsrc** command passes the **-D** argument to the lock manager when it starts the subsystem. The first node to complete its node_up processing sets the value of the global tuning parameter. Once this node is up, you must use the **clm_setglobparams** routine to change the value of this global tuning parameter.

Note: Even if you are updating only a single field on the Change Resource Allocation screen, you must enter a value for each field on the screen. You can enter the default values shown on the screen above for the fields you are not updating.

Specifying the Decay Rate from within an Application

Use the following routines to read or set the value of the decay rate from within an application:

clm_getglobparams	Retrieves the current settings of the lock manager global parameters.
clm_setglobparams	Assigns a value to the lock manager global parameters.

Both routines accept a single argument: the address of a **clm_globparams_t** structure. When used with the **clm_getglobparams** routine, the lock manager writes the current values of the parameters in this structure. When used with the **clm_setglobparams** routine, applications use this structure to specify the desired values of the global parameters. This data structure has the following format:

```
typedef struct clm_globparams {
      unsigned  cg_valid;
      unsigned cg_recalc_time;
      float          cg_decay_rate;
} clm_globparams_t;
```

The contents of the fields varies depending on which routine they are used with.

cg_valid	Indicates which fields in the **clm_globparams_t** structure contain valid data. This field is only used when specifying the value of the global parameters with the **clm_setglobparams** routine to specify the values of global parameters. When the **clm_getglobparams** routine returns successfully, both fields can be assumed to be valid.
cg_decay_rate	Used to specify the desired decay rate value, when used with the **clm_setglobparams** routine. Contains the current value of the decay rate, after the **clm_getglobparams** routine returns successfully.

Specifying the Stickiness Value of a Lock Resource

To control the migration behavior of an individual lock resource, assign a value to the stickiness attribute of the lock resource. You can assign any value between 0 and 100. This value determines the percentage of the sum of the remote access rates the lock manager adds to the local access rate to give the local node the advantage in migration calculations. A stickiness value of 100 guarantees that the lock resource remains on the local node. By default, the lock manager adds 50% of the sum of the remote access rates to the local access rate.

To illustrate how the stickiness attribute can affect lock resource master migration, consider a cluster configuration in which three nodes (A, B, and C) access the same lock resource. The lock resource is mastered on Node A and the stickiness attribute of the lock resource is set at 100. The following table describes the access patterns from each node at an evaluation point.

Node:	Node A	Node B	Node C
Accesses:	3	20	0

Given these access rates, the lock manager would move the lock resource to Node B which has the highest access rate. However, because the stickiness attribute of the lock resource is set to 100, the lock manager adds the sum of all the remote access rates to the local access rate before making the migration determination. In this case, the lock manager would add 20, the sum of the access rates from Node B and Node C, to the local access rate. This addition increases the local access rate to 23, making it greater than the highest access rate of any of the remote nodes. This ensures that the lock resource remains on the local node.

You specify the value of the stickiness attribute using the following routines:

clm_getresparams	Retrieves the current setting of the lock resource stickiness parameter.
clm_setresparams	Assigns a value to the lock resource stickiness parameter.

Both routines accept a single argument: the address of a **clm_resparams_t** structure. When used with the **clm_getresparams** routine, the lock manager writes the current values of the attribute in this structure. When used with the **clm_setresparams** routine, applications use this structure to specify the desired value of the attribute. The data structure has the following format:

```
typedef struct {
      unsigned cr_valid;
      unsigned cr_stickiness;
} clm_resparams_t;
```

The contents of the fields varies depending on which routine they are used with.

cr_valid	Indicates which of the other fields in the **clm_resparams_t** structure contain valid data. This field is only used when setting the value of the stickiness attribute with the **clm_setresparams** routine. When the **clm_getresparams** routine returns successfully, the stickiness field can be assumed to be valid.
cr_stickiness	Specifies the desired value of the stickiness parameter, when used with the **clm_setresparams** routine. Contains the current value of the stickiness attribute, after the **clm_getresparams** routine returns successfully.

Obtaining Lock Resource Statistics

Lock resource statistics provide information about the usage of a particular lock resource. How the resource statistics are used is completely up to the client application. You use the **clm_getstats** routine in your application to obtain these statistics.

The lock resource statistics indicate the number of times specific events have occurred. The **clm_statistics_t** structure, which represents the resource statistics information, has the following format:

```
typedef struct clm_statistics {
      unsigned long cs_requests;
      unsigned long cs_local;
      unsigned long cs_remote;
      unsigned long cs_same;
      unsigned long cs_migrations;
      unsigned long cs_compat;
      unsigned long cs_incompat;
      unsigned long cs_downgrade;
      float cs_total_aps;
      float cs_aps[CLM_MAXNODES];
    } clm_statistics_t;
```

The locking statistics track four types of information.

- Number and origin of lock requests
- Migrations
- Compatibility
- Accesses-per-second

All the locking statistics that increase incrementally (all the statistics except **cs_total_aps** and **cs_aps**) are defined using the unsigned long datatype which can accommodate values over four billion before overflowing. The **cs_total_aps** and **cs_aps** statistics, which can increase geometrically depending on the decay rate specified, are defined using the float datatype which can accommodate values over 3.4^{38} before overflowing.

Number and Origin of Lock Requests

The following statistics track the number and origin of lock requests on a resource.

cs_requests	Number of lock requests on the specified resource
cs_local	Number of local lock requests (same node as current lock resource master node)
cs_remote	Number of remote lock requests
cs_same	Number of successive lock requests by same node.

Migration of Lock Resources

The **cs_migrations** statistic indicates the number of times the lock has migrated.

Compatibility of Lock Resources

The following statistics track compatibility issues concerning a lock resource.

cs_compat	Number of compatible lock requests (new locks and up-conversions compatible with existing locks)
cs_incompat	Number of incompatible lock requests (incompatible new locks and up-conversions)
cs_downgrade	Number of down-conversions and unlocks.

Accesses-Per-Second (APS)

These statistics represent the accesses-per-second (aps) figures for a lock resource. The lock manager calculates these values when it periodically evaluates if the lock resource should be moved to another cluster node (lock resource master migration).

cs_aps	An array that contains the access rates to the lock resource from each cluster node. These values represent the raw access rate plus the decayed historical access rate. The ordering of the access rates in the array corresponds to the alphabetic ordering of the names of cluster nodes.
cs_total_aps	Sum of all the access rates in the **cs_aps** array.

Note that requests for statistics do not update the accesses per second figures.

Lock and Lock Resource Limits

Lock Manager Kernel Memory Usage

The lock manager kernel extension (**cllockd.x**) maintains its database of locks and lock resources in a private kernel segment. As applications request locks, the lock manager dynamically allocates memory for the lock database from its segment.

In AIX, segments are exactly 256 MB. However, only half of the lock manager segment (128MB) can be used to hold locks and lock resources. The remaining portion of the lock manager segment must be held in reserve to satisfy transient demands, such as those associated with remastering locks after a node failure. (Kernel memory is pageable so the amount of physical memory does not affect lock limits; it does affect performance, however.)

Maximum Acquired Locks Per Node

Given the lock manager's memory allocation algorithm, you can calculate approximate limits on the number of locks and lock resources an application can acquire.

Each lock resource requires approximately 450 bytes of storage and each lock requires 300 bytes. Each lock resource must have at least one lock against it. You can calculate the maximum number of locks and lock resources as follows:

```
Let x = Max no.locks = Max no.resources
450 bytes * x resources + 300 bytes * x locks = 128MB
750x = 128MB
x = 128MB/750
x = 178K locks and 178K lock resources (approximate)
```

Thus, the maximum number of locks you can acquire on a single node is approximately 178K, where the ratio of locks to lock resources is 1:1.

Realistically, however, there are multiple locks against each lock resource. Thus, the practical limit on the number of lock resources you can acquire is much less than 178K. For example, in multi-node clusters, the node that maintains the master copy of a lock resource acquires an additional lock against the lock resource for each lock taken on any other cluster node. The lock-to-lock resource ratio on the node maintaining the master copy can quickly reach 3:1 or 4:1. To calculate the practical maximum, you must consider the size of the cluster and be familiar with the locking characteristics of the applications you run.

Example of Kernel Memory Usage

A database program that pre-allocates 150,000 locks when it is started, shows how quickly the lock manager's 128MB limit can be reached when instances of the database program are started on multiple cluster nodes.

On Node A, in a two-node cluster, the database program is started and it requests 150,000 lock resources, each with a single lock. The lock manager grants these locks.

When another instance of the database program is started on Node B, it requests a lock on each of the 150,000 lock resources already created on Node A. On Node B. the lock manager grants the locks, allocating kernel memory for 150,000 lock resources, each with one lock against it.

On Node A, the lock manager must acquire an additional lock on each of the existing lock resources. These additional locks represent copies of Node B's locks. Node A maintains the master copies of these lock resources. However, when the lock manager attempts to acquire these new locks on the 150,000 lock resources, it reaches its 128MB limit for memory usage in its segment before it can acquire all the requested locks. The node maintaining the master copies of lock resources is the first node to start denying lock requests.

This example illustrates how you must be aware of the locking characteristics of the applications you intend to run on your cluster. This problem can be remedied by lowering the number of locks the application pre-allocates. (This is typically a tunable parameter in most database applications.) For example, lowering the number of locks pre-allocated by the database to 120,000 would allow it to start in a two-node cluster without exceeding the lock manager's kernel memory usage limit. However, as the number of cluster nodes increases, the number of locks pre-allocated by the database program must be reduced even further. With each new node, the ratio of locks to lock resources gets larger on the node maintaining the master copy of each lock resource.

The number of locks that can be supported in a cluster with N nodes, where each node requests a lock against a specified number of lock resources can be determined by the following formula:

```
128*1024*1024
-------------
(450 + N*300)
```

In round numbers, this gives:

Number of Nodes	Number of locks (thousands)
2	127
3	99
4	81
5	68
6	59
7	52
8	46

Migration of lock resources can, over time, even out lock manager memory allocation across cluster nodes. As lock resource access patterns cause lock resource masters to move, one node, such as Node A in the example, does not need to acquire copies of all the locks acquired cluster-wide.

When Locks are Denied

When the lock manager reaches its kernel memory usage limit, it returns an error (CLM_DENIED_NOLOCKS) to the calling application and leaves it up to the application to decide how to handle the failure (wait and retry or exit immediately).

When the lock manager denies a request for a lock on an existing lock resource, it issues the following error to the syslog (if kern.crit or lower priority messages are enabled in syslog.conf):

```
Jun 11 17:12:00 nodeA_svc unix: denying request to grow HR>lock tab  1
```

When the lock manager denies a request for a lock on a new lock resource, it issues the following error to the syslog:

```
Jun 11 17:12:00 nodeA_svc unix: kernel memory limit reached
```

After the above messages, several of the following errors appear:

```
Jun 11 17:12:00 nodeA_svc unix: get_le: NO MORE LOCKS<HR>
Jun 11 17:12:00 nodeA_svc unix: get_le: NO MORE LOCKS
```

Lock Value Block Changes

Lock clients now have greater flexibility in updating and invalidating the lock value block during lock conversion operations.

Previously, lock clients were only allowed to update or invalidate the lock value block during a down convert from EX mode or PW mode to a less restrictive mode. To get this previous behavior, lock clients must set the CLM_LVB_OLD environment variable. The value of the CLM_LVB_OLD environment variable is insignificant.

If LCM_LVB_OLD is not set, lock clients will observe the new behavior of also allowing LVB updates or invalidating when converting from EX or PW modes to the same mode.

Chapter 7 Lock Manager API Routines

This chapter provides reference information on the C language routines used to implement locking in an HACMP for AIX application.

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.

CLM Locking Model-Specific Routines

ASTpoll	Triggers the execution of pending AST routines.
clmlock	Makes an asynchronous (non-blocking) request to acquire or convert a lock on a lock resource.
clmlockx	Makes an asynchronous (non-blocking) request to acquire or convert a lock on a lock resource, and specifies a transaction ID for that lock.
clmlock_sync	Acquires or converts a lock on a lock resource and obtains a synchronous return.
clmlockx_sync	Acquires or converts a lock on a lock resource, specifies a transaction ID for that lock, and obtains a synchronous return.
clmunlock	Releases a lock on a lock resource or cancels a lock request that is in blocked or converting state.
clm_scnop	Manipulates the SCN, a specialized use of the lock value block associated with a lock resource.
clm_setnotify	Specifies which signal the lock manager should use to notify your application of a pending AST.

UNIX Locking Model-Specific Routines

clmregister	Registers a lock resource.
clmregionlock	Acquires a lock on a lock resource or releases a lock on a lock resource.

Routines Common to Both Locking Models

clm_errmsg	Returns a pointer to a printable version of the CLM API status code for single-threaded applications.
clm_getglobparams	Obtains the value of the global lock manager parameters.
clm_getresparams	Returns the value of a lock resource’s stickiness attribute.
clm_getstats	Obtains statistics on resource usage.
clm_grp_attach	Attaches a lock client to an existing lock group.
clm_grp_create	Creates a new lock group and associates the lock client process with the group.
clm_grp_detach	Removes a lock client process from an existing lock group.
clm_perror	Writes a message you specify to standard error.
clm_purge	Releases all locks owned by a particular client, identified by its process ID.
clm_setglobparams	Sets the value of the global lock manager parameters, including the evaluation threshold and the decay rate.
clm_setresparams	Sets the value of the lock resource’s stickiness attribute.

The following sections provide reference information about the routines.

ASTpoll Routine

Syntax

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

Description

Use the **ASTpoll** routine to trigger any pending ASTs resulting from the completion of previous **clmlock** or **clmlockx** routine requests or the delivery of blocking ASTs.

Parameters

pid

This argument indicates the process ID of the application having outstanding ASTs or blocking ASTs. This 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.

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.

Example

For an example of the ASTPoll routine, see the Sample Locking Application on page 3-6.

clmlock Routine

Syntax

```
clm_stats_t clmlock(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)();
```

Description

Use the **clmlock** 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 the Parameters section.

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 Parameters on page 7-2.

The lock manager returns status in two locations: the status value returned by the **clmlock** routine and the status field of the lock status block. The status value returned by the **clmlock** routine indicates whether the request was accepted by the lock manager. The CLM_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 **clmlock** routine returns an error status code. See Status Codes on page 7-3 for a list of these status values.

A success status from the **clmlock** 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

CLM_NORMAL status value indicates your request was granted. See Status Codes Returned in the Lock Status Block on page 7-8 for a list of other possible status values. (For information about the composition of the lock status block, see Data Structure on page 3-2.)

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.

Parameters

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 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 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 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 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 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.

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 Data Structure on page 3-2.

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 Requesting Local Locks on page 3-8. 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 lookup 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 Requesting Local Locks on page 3-8.

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 CLM_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 **clmlock** routine returns the status CLM_SYNC, instead of CLM_NORMAL, and there is no asynchronous return. If the lock manager cannot grant the request synchronously, the **clmlock** routine returns CLM_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 CLM_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 Manipulating the Lock Value Block on page 3-11.

name

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

namelen

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

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.

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);
```

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);
```

Status Codes

The status codes returned are listed alphabetically as follows:

CLM_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.)
CLM_BADPARAM	The lock mode specified is not a valid lock mode.
CLM_DENIED_NOASTS	No more ASTs are available.
CLM_IVBUFLEN	The **namelen** (name length) was less than 1 or greater than 31 characters.
CLM_IVLOCKID	The lock ID is invalid.
CLM_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.
CLM_NORMAL	The lock request completed successfully.
CLM_REJECTED	The lock manager does not recognize the client. This occurs if a lock manager is restarted during a lock session.
CLM_SYNC	The lock request included the LKM_SYNCSTS flag and the lock manager was able to grant it synchronously.
CLM_SYSERR	A data format error occurred, indicating a problem with the network facilities or an internal error with the lock manager.

Status Codes Returned in the Lock Status Block

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

CLM_ABORT	A waiting lock was canceled by a **clmunlock** call with the LKM_CANCEL flag set.
CLM_CANCEL	A blocked conversion was canceled by a **clmunlock** call with the LKM_CANCEL flag set. The lock retains its original granted mode.

CLM_CVTUNGRANT	The request attempted to convert a lock that was blocked in the WAIT state.
CLM_DEADLOCK	The lock manager canceled this request to prevent deadlock from occurring.
CLM_DENIED	The request attempted to convert a lock that was already blocked on a conversion request.
CLM_DENIED_NOLOCKS	Either no more locks or no more resources are available. For information about the lock manager allocates locks and lock resources, see Lock and Lock Resource Limits on page 6-11.
CLM_NORMAL	The lock request completed successfully.
CLM_NOTQUEUED	The request included the LKM_NOQUEUE flag and could not be satisfied immediately.
CLM_TIMEOUT	The timeout expired before this request was able to complete.
CLM_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.

Example

For an example, see the Sample Locking Application on page 3-6.

clmlockx Routine

Syntax

```
clm_stats_t clmlockx(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)();
clm_xid_t *xid;
```

Description

Use the **clmlockx** 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 **clmlockx** routine performs the same function as the **clmlock** routine. See the documentation for the **clmlock** routine on page 7-3 for a description of the base functionality.

Additionally, the **clmlockx** 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 **clmlockx** 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 **clmlockx** 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.

Parameters

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 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 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 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 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 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.

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 page 3-2.

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. The lock manager ignores the xid parameter.

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 Requesting Local Locks on page 3-8.

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 lookup 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 Requesting Local Locks on page 3-8.

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 CLM_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: This 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 **clmlockx** routine returns the status CLM_SYNC, instead of CLM_NORMAL, and there is no asynchronous return. If the lock manager cannot grant the request synchronously, the **clmlockx** routine returns CLM_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 CLM_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 page 3-2.
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.

name

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

namelen

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

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.

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);
```

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);
```

xid

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

Status Codes

The status codes returned are listed alphabetically as follows

CLM_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.)
CLM_BADPARAM	The lock mode specified is not a valid lock mode.
CLM_DENIED_NOASTS	No more ASTs are available.
CLM_IVBUFLEN	The **namelen** (name length) was less than 1 or greater than 31 characters.
CLM_IVLOCKID	The lock ID is invalid.
CLM_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.
CLM_NORMAL	The lock request completed successfully.
CLM_REJECTED	The lock manager does not recognize the client. This occurs if a lock manager is restarted during a lock session.
CLM_SYNC	The lock request included the LKM_SYNCSTS flag and the lock manager was able to grant it synchronously.
CLM_SYSERR	A data format error occurred, indicating a problem with the network facilities or an internal error with the lock manager.

Status Codes Returned in the Lock Status Block

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

CLM_ABORT	A waiting lock was canceled by a **clmunlock** call with the LKM_CANCEL flag set.

CLM_CANCEL	A blocked conversion was canceled by a **clmunlock** call with the LKM_CANCEL flag set. The lock retains its original granted mode.
CLM_CVTUNGRANT	The request attempted to convert a lock that was blocked in the WAIT state.
CLM_DEADLOCK	The lock manager killed this request to prevent deadlock from occurring.
CLM_DENIED	The request attempted to convert a lock that was already blocked on a conversion request.
CLM_DENIED_NOLOCKS	Either no more locks or no more resources are available. For information about the lock manager allocates locks and lock resources, see Lock and Lock Resource Limits on page 6-11.
CLM_NORMAL	The lock request completed successfully.
CLM_NOTQUEUED	The request included the LKM_NOQUEUE flag and could not be satisfied immediately.
CLM_TIMEOUT	The timeout expired before this request was able to complete.
CLM_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.

Example

```
clm_stats = status;

status = clmlockx(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 */
```

clmlock_sync Routine

Syntax

```
clm_stats_t clmlock_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 *);
```

Description

Use the **clmlock_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 **clmlock_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 the “Parameters” section.

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 Parameters on page 7-2.

The lock manager returns status in two locations: the status value returned by the **clmlock_sync** routine and the status field of the lock status block. The status value returned by the **clmlock_sync** routine indicates whether the request was accepted by the lock manager. The CLM_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 **clmlock_sync** routine returns an error status code. See Status Codes on page 7-3 for 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 CLM_NORMAL status value indicates your request was granted. Status Codes Returned in the Lock Status Block on page 7-8 for a list of other possible status values. (For information about the composition of the lock status block, see Data Structure on page 3-2.)

Parameters

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 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 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 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 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 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.

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 Data Structure on page 3-2.

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 Requesting Local Locks on page 3-8.

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 lookup 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 compromises 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 Requesting Local Locks on page 3-8.

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 CLM_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: This 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 CLM_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 Data Structure on page 3-2.

name

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

namelen

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

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);
```

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);
```

Status Codes

The status codes returned are listed alphabetically as follows:

CLM_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.)

CLM_BADPARAM
The lock mode specified is not a valid lock mode.

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

CLM_IVLOCKID
The lock ID is invalid.

CLM_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.

CLM_NORMAL
The lock request completed successfully.

Status Codes Returned in the Lock Status Block

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

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

CLM_DEADLOCK
The lock manager killed this request to prevent deadlock from occurring.

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

CLM_DENIED_NOLOCKS
Either no more locks or no more resources are available. For information about how the lock manager allocates locks and lock resources, see Lock and Lock Resource Limits on page 6-11.

CLM_NORMAL
The lock request completed successfully.

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

CLM_TIMEOUT
The timeout expired before this request was able to complete.

CLM_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.

Example

```
clm_stats = status;

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

clmlockx_sync Routine

Syntax

```
clm_stats_t clmlockx_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 *);
clm_xid_t *xid;
```

Description

Use the **clmlockx_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 **clmlockx_sync** routine performs the same function as the **clmlock_sync** routine. See the documentation for the **clmlock_sync** routine starting on page 7-16 for a description of the base functionality.

Additionally, the **clmlockx_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 **clmlockx_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 **clmlockx_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.

Parameters

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 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 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 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 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 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.

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 Data Structure on page 3-2.

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. The lock manager ignores the xid parameter.

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 Requesting Local Locks on page 3-8.

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 lookup 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 the application must acquire additional locks on a local lock resource, specify the LKM_FINDLOCAL flag when requesting the lock. See Requesting Local Locks on page 3-8.

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 CLM_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: This 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 CLM_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 Data Structure on page 3-2.

name

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

namelen

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

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);
```

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);
```

xid

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

Status Codes

The status codes returned are listed alphabetically as follows:

CLM_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.)

CLM_BADPARAM

The lock mode specified is not a valid lock mode.

CLM_IVBUFLEN

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

CLM_IVLOCKID

The lock ID is invalid.

CLM_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.

CLM_NORMAL

The lock request completed successfully.

Status Codes Returned in the Lock Status Block

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

CLM_CVTUNGRANT

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

CLM_DEADLOCK

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

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

CLM_DENIED_NOLOCKS
Either no more locks or no more resources are available. For information about how the lock manager allocates locks and lock resources, see Lock and Lock Resource Limits on page 6-11.

CLM_NORMAL
The lock request completed successfully.

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

CLM_TIMEOUT
The timeout expired before this request was able to complete.

CLM_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.

Example

```
clm_stats = status;

status = clmlockx_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 */
```

clmregister Routine

Syntax

```
union clm_rh clmregister(name)
char *name;
```

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 **clmregister** routine to register (create) a lock resource. A lock resource remains in existence until the last process to have it registered exits.

Parameters

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/clm.h** header file.

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 clm_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 **clm_errno** global variable contains the status code associated with the error.

Status Codes

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

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

CLM_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.

CLM_NORMAL
The register request completed successfully.

Example

```
union clm_rh reshandle;

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

clmregionlock Routine

Syntax

```
clm_stats_t clmregionlock(rh, offset, length, flags)
union clm_rh rh;
unsigned long offset;
unsigned long length;
unsigned long flags;
```

Description

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

Parameters

rh

A valid lock resource handle returned by an earlier call to the **clmregister** routine.

offset

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

length

The length of the region starting at offset.

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.

Status Codes

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

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

CLM_IVRESHANDLE
The resource handle is invalid.

CLM_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.

CLM_NORMAL
The lock request completed successfully.

Example

```
union clm_rh reshandle;
unsigned long offset;
unsigned long len;
clm_stats_t status;

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

/*  processing occurs here  */

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

clmunlock Routine

Syntax

```
clm_stats_t clmunlock(lockid, valueblock, flags)
int lockid;
char *valueblock;
int flags;
```

Description

Use the **clmunlock** 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 **clmunlock** 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.

Parameters

lockid

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

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.

If the request (1) is an unlock request (the LKM_CANCEL flag is not included), and (2) the current granted mode of the lock is either EX or PW, and (3) the LKM_VALBLK flag was included, the lock manager updates the contents of the lock value block associated with the lock name using the value contained in **valueblock**.

flags

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

LKM_CANCEL

When you set the LKM_CANCEL flag, the **clmunlock** 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 CLM_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 clmunlock or clmunlock_async, it is ignored.

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.

LKM_VALBLK

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.

Status Codes

The status codes returned are listed alphabetically as follows:

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

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

CLM_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.

CLM_IVLOCKID
The lock ID is invalid.

CLM_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.

CLM_NORMAL
The unlock request completed successfully.

Example

For an example, see Releasing a Lock on a Lock Resource on page 3-10.

clmunlock_async Routine

Syntax

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

Description

Use the **clmunlock_async** 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 **clmunlock_async** routine and the lstat argument passed to the unlock AST function. The status value returned by the **clmunlock_async** 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 **clmunlock_async** routine returns an error status code. See Status Codes on page 7-3 for a list of the status values.

A success status from the *clmunlock_async* 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 unlock AST function specified as an argument to the *clmunlock_async* routine.

The unlock AST function has the following declaration:

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

The unlockastargs and extrap values passed to the unlock AST routine are the same values passed to the call to the **clmunlock_async** 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 **clmunlock** routine.

An application triggers the unlock AST routine by calling the ASTpoll routine. SeeStatus Codes Returned in the Lock Status Block on page 7-8 for a list of other possible status values.

Parameters

lockid

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

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.

If the request (1) is an unlock request (the LKM_CANCEL flag is not included), and (2) the current granted mode of the lock is either EX or PW, and (3) the LKM_VALBLK flag is included, the lock manager updates the contents of the lock value block associated with the lock name using the value contained in **valueblock**.

flags

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

LKM_CANCEL

When you set the LKM_CANCEL flag, the **clmunlock** 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 CLM_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 **clmunlock** or **clmunlock_async**, it is ignored.

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.

LKM_VALBLK

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.

unlockast

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

unlockastargs

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

extrap

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

Status Codes

The status codes returned are listed alphabetically as follows:

CLM_BADARGS

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

CLM_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.

CLM_IVLOCKID

The lock ID is invalid.

CLM_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.

CLM_NORMAL

The unlock request completed successfully.

Status Codes Returned in the Unlock AST

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

CLM_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.

CLM_NORMAL
The unlock request completed successfully.

clm_errmsg Routine

Syntax

```
char *clm_errmsg(status)
clm_stats_t status;
```

Description

The **clm_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 **clm_stats_t** enumerated type are constants, not printable character strings.

Parameters

status

A CLM API status code.

Returns

A NULL-terminated character string. For example, if the status code returned is CLM_NORMAL, the string returned is “CLM_NORMAL.”

If the status parameter you specify as an argument is not a valid lock manager status code, the **clm_errmsg** routine returns the string “Invalid status.”

Example

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

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

char *msg;


clm_stats_t status;
.
.
```

```
.
status = clm_setnotify( SIGUSR1, NULL );
if ( status != CLM_NORMAL )
{
    msg = clm_errmsg(status);
    printf("clm_setnotify returns %s",msg);
}
```

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

```
clm_setnotify returns CLM_BADARGS
```

clm_getglobparams Routine

Syntax

```
clm_stats_t clm_getglobparams(params)
clm_globparams_t *params;
```

Description

The **clm_getglobparams** routine obtains the value of the global lock manager parameters.

Parameters

params

Address of the **clm_globparams_t** structure into which the lock manager writes the values of the global parameters. For information about interpreting the values returned, see Chapter 6, Tuning the Cluster Lock Manager.

Status Codes

The following is an alphabetical list of status values returned by the **clm_getglobparams** routine.

CLM_BADARGS

The pointer to the **clm_globparams_t** structure is invalid.

CLM_NORMAL

The operation completed successfully.

Example

The following code fragment illustrates how to obtain the current value of the decay rate parameter. When the **clm_getglobparams** routine returns successfully, both the **cg_recalc_rate** field and the **cg_decay_rate** field in the global parameters structure are valid.

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

clm_globparams_t  glob_params;
clm_stats_t       status;
float             decay_rate;
.
.
```

```
.
status = clm_getglobparams( &globparams );
if (status != CLM_NORMAL)
{
   clm_perror("Could not get global resource parameters")
}
else
{
   decay_rate = globparams.cg_decay_rate;
}
```

clm_getresparams Routine

Syntax

```
clm_stats_t clm_getresparams(res_name,namelen,res_type,params)
char *res_name;
short namelen;
short res_type;
clm_resparams_t *params;
```

Description

The **clm_getresparams** routine returns the value of a lock resource’s stickiness attribute.

Parameters

res_name

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

namelen

The number of characters in the name.

res_type

Specifies the type of lock resource. For CLM lock resources, specify the constant CLM_RES_VMS. For UNIX lock resources, specify the constant CLM_RES_UNIX.

params

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

Status Codes

The following is an alphabetical list of status values returned by the **clm_getresparams** routine.

CLM_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 **clm_resparams_t** structure is invalid.

CLM_BADRESOURCE
The lock resource specified is invalid.

CLM_NORMAL
The operation completed successfully.

Example

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

```
#include <cluster/clm.h>
clm_resparams_t  resparams;

clm_status_t status;

#define NAMELEN 7
.
.
.
status = clm_getresparams("my_lock",  /* name of lock resource */
                           NAMELEN,  /* length of name        */
                       CLM_RES_VMS,  /* resource type         */
                       &resparams);  /* address of resparms   */
                                     /*     structure         */
if(status != CLM_NORMAL)
{
      clm_perror("Can't read stickiness value");
      exit(1);
}
```

clm_getstats Routine

Syntax

```
clm_stats_t clm_getstats(resname, namelen, type, statistics)
char *resname;
short namelen;
short type;
clm_statistics_t *statistics;
```

Description

Use the **clm_getstats** routine to obtain statistics on resource usage, including the number and origin of lock requests on a resource, the number of times the lock has migrated, the compatibility of lock requests, and accesses-per-second per node on a lock resource. How the resource statistics are used is completely up to the client application.

Calling this function on a non-existent resource returns an error.

Parameters

resname

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

namelen

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

type

Specifies the type of lock resource. For CLM lock resources, specify the constant CLM_RES_VMS. For UNIX lock resources, specify the constant CLM_RES_UNIX.

statistics

Address of the **clm_statistics_t** structure into which the lock manager writes the current values of the lock statistics.

Status Codes

The status codes returned are below:

CLM_BADARGS

The request passed a NULL pointer to the statistics structure.

CLM_BADRESOURCE

Unable to find the specified resource. A copy must be on the local node.

CLM_IVBUFLEN

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

CLM_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.

CLM_NORMAL

The lock request completed successfully.

CLM_VERSION_CONFLICT

The request cannot be processed because a back-level version of Cluster Lock Manager is running in the cluster.

Example

```
int i;
clm_stats_t status;
clm_statistics_t statistics;

status = clm_getstats ("RESOURCE1", 9, CLM_RES_VMS, &statistics);

if (status != CLM_NORMAL)
{
    printf ("error: clm_getstats returned %s",clm_errmsg(status));
}
else
{
     printf ("Resource Statistics on RESOURCE1");
     printf ("cs_requests   = %d", statistics.cs_requests);
     printf ("cs_local      = %d", statistics.cs_local);
     printf ("cs_remote     = %d", statistics.cs_remote);
     printf ("cs_same       = %d", statistics.cs_same);
```

```
          printf ("cs_migrations = %d", statistics.cs_migrations);
          printf ("cs_compat     = %d", statistics.cs_compat);
          printf ("cs_incompat   = %d", statistics.cs_incompat);
          printf ("cs_downgrade  = %d", statistics.cs_downgrade);
          printf ("cs_total_aps  = %f", statistics.cs_total_aps);
          for ( i = 0 ; i < CLM_MAXNODES ; i++ )
          {
               printf ("cs_aps[%d]  = %f", i, statistics.cs_aps[i]);
          }
     }
```

clm_grp_attach Routine

Syntax

```
clm_stats_t clm_grp_attach(gid, flags)
int gid;
int flags;
```

Description

Use the **clm_grp_attach** routine to attach a lock client to an existing lock group. Use the group ID returned by the **clm_grp_create** routine to specify the group. A client may belong to only one group.

Parameters

gid

The group ID returned by the **clm_grp_create** routine.

flags

None.

Status Codes

The status codes are listed below:

CLM_DENIED
The process was already attached to a lock group.

CLM_IVGROUPID
The request specified an invalid lock group.

CLM_NORMAL
The request completed successfully.

Example

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

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

clm_grp_create Routine

Syntax

```
clm_stats_t clm_grp_create(gid, flags)
int *gid;
int flags;
```

Description

Use the **clm_grp_create** routine to create a new lock group and associate the client with this group. The **clm_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 **clmlock** or **clmlock_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.

Parameters

gid

Pointer to location to store the group ID.

flags

None.

Status Codes

CLM_DENIED
The process was already attached to a group.

CLM_NORMAL
The request completed successfully.

Example

```
int groupid;
int ret;

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

clm_grp_detach Routine

Syntax

```
clm_stats_t clm_grp_detach(flags)
int flags;
```

Description

Use the **clm_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 its terminates.

Parameters

flags

None.

Status Codes

The status codes are listed below:

CLM_DENIED
The process was not attached to a group.

CLM_NORMAL
The request completed successfully.

Example

```
int ret;

/* Detach the current process from lock group */

ret = clm_grp_detach(0);

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

clm_perror Routine

Syntax

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

Description

The **clm_perror** routine allows an application to write a message to standard error that indicates why a lock request failed. The **clm_perror** routine consults the **clm_errno** global variable to determine the status of the last lock request. The **clm_perror** routine appends the supplied message with a colon and a printable version of the status code.

Parameters

message

A NULL-terminated character string.

Example

The following code fragment uses the **clm_perror** return to print an error message if the **clm_setnotify** routine fails. The application includes the message string “clm_setnotify fails” for the **clm_perror** routine to print along with the status code return by the routine.

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

clm_stats_t status;
.
.
.
status = clm_setnotify( SIGUSR1, NULL );
if ( status != CLM_NORMAL )
{
    clm_perror("clm_setnotify fails");
}
```

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

```
clm_setnotify fails: CLM_BADARGS
```

clm_purge Routine

Syntax

```
clm_stats_t clm_purge(node_id, pid, flags)
int node_id;
int pid;
int flags;
```

Description

The **clm_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 **clm_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.

Note: The **clm_purge** function cannot be used to purge the locks of an active client other than the one calling the function.

Parameters

node_id

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

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.

flags

There are no flags for this routine.

Status Codes

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

CLM_BADARGS
The request specified an invalid node ID.

CLM_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.

CLM_NORMAL
The purge request completed successfully.

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. You use the routines provided by the HACMP Clinfo API to obtain the node ID. Clinfo provides routines you can use to obtain the cluster ID, node name and other information about the cluster environment. For more information about Clinfo, see *HACMP for AIX Programming Client Applications*.

Note: If your application uses Clinfo to obtain the node ID, you must link your application with the Clinfo library (**-lcl**).

```
#include <cluster/clm.h>
int nodeid;
.
.
.
status = clm_purge(nodeid, get_pid(), 0 );
if( status != CLM_NORMAL )
   clm_perror("clm_purge");
```

clm_scnop Routine

Syntax

```
clm_stats_t clm_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;
```

Description

The **clm_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 **clm_scnop** routine returns successfully, the SCN operation is complete.

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

Parameters

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 **clm_scnop** routine returns the status CLM_BLOCKED.

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.

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 $2^{\textbf{bit\_len} - 1}$. 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.

in_lvb

Pointer to the input value of the SCN.

out_lvb

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

Status Codes

The following is an alphabetized list of status values returned by the **clm_scnop** routine.

CLM_BADARGS

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

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

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

CLM_NORMAL
The SCN operation completed successfully.

CLM_NOLOCKMGR
The Lock Manager is not running.

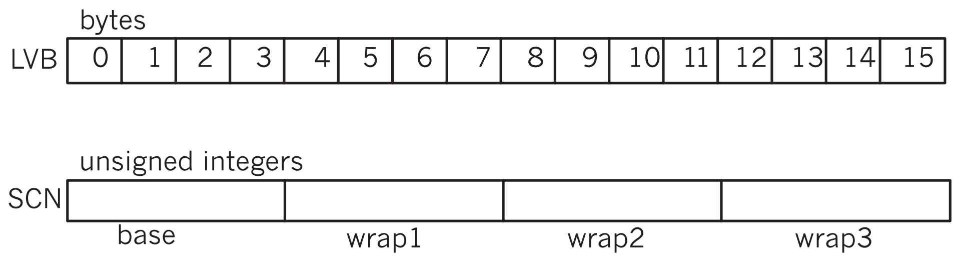
CLM_VALNOTVALID
The Lock Value Block (LVB) in which the SCN is stored is marked invalid.

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:

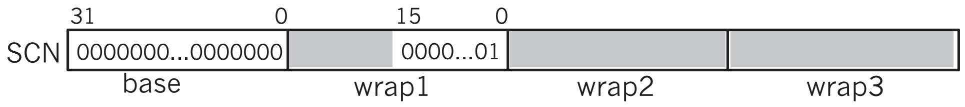


You define the range of the SCN counter by specifying, in the **bit_len** parameter passed to the **clm_scnop** routine, how many bits are used to represent its value. The value of the **bit_len** parameter controls which bits in the four fields are used.

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 $2^{\textbf{bit\_len}-1}$), 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 2^{32} 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.

Example

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

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

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

in_scn.base = 100000;

/* Set SCN value if less than in_scn */

status = clm_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 != CLM_NORMAL)
{
      clm_perror("Can't get SCN.");
}
```

clm_setglobparams Routine

Syntax

```
clm_stats_t clm_setglobparams(params)
clm_globparams_t *params;
```

Description

The **clm_setglobparams** routine sets the value of the global lock manager parameters, including the evaluation threshold and the decay rate.

Parameters

params

Address of the **clm_globparams_t** structure into which you write the values you want assigned to the lock manager global parameters.

Status Codes

The following is an alphabetical list of status values returned by the **clm_setglobparams** routine.

CLM_BADARGS
The pointer to the **clm_globparams_t** structure is invalid.

CLM_NORMAL
The operation completed successfully.

Example

The following code fragment specifies values for both the decay rate and evaluation threshold.

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

clm_globparams_t globparams;
clm_stats_t status;
.
.
.
globparams.cg_valid = CLMTUNE_GLOB_RECALC | CLMTUNE_GLOB_DECAY;
globparams.cg_decay_rate = .50;
globparams.cg_recalc_time = 100;

status = clm_setglobparams( &globparams );

if (status != CLM_NORMAL)
{
      clm_perror("Cannot set global parameters.")
}
```

clm_setnotify Routine

Syntax

```
clm_stats_t clm_setnotify(signo, oldsigp)
int signo;
int *oldsigp;
```

Description

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

Parameters

signo

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

oldsigp

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

Status Codes

CLM_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>**.

CLM_NORMAL
The request completed successfully.

Example

For an example, see the Sample Locking Application on page 3-6.

clm_setresparams Routine

Syntax

```
clm_stats_t clm_setresparams(res_name,namelen,res_type,params)
char           *res_name;
short            namelen;
short           res_type;
clm_resparams_t  *params;
```

Description

The **clm_setresparams** routine sets the value of a lock resource’s stickiness attribute.

Parameters

res_name

The name of the lock resource.

namelen

The number of characters in the resource name.

res_type

Specifies the type of lock resource. For CLM lock resources, specify the constant CLM_RES_VMS. For UNIX lock resources, specify the constant CLM_RES_UNIX.

params

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

Status Codes

The following is an alphabetical list of status values returned by the **clm_setresparams** routine.

CLM_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 **clm_resparams_t** structure is invalid.

CLM_BADRESOURCE

The lock resource specified is invalid.

CLM_NORMAL

The operation completed successfully.

Example

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

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

clm_stats_t status;

#define  NAMELEN 7

clm_resparams_t resparams;
.
.
.
resparams.cr_valid = CLM_RES_STICKINESS;
resparams.cr_stickiness = 50;


status = clm_setresparams( "my_lock",/* name of lock resource  */
                         NAMELEN,  /* length of name         */
                     CLM_RES_VMS,  /* resource type         */
                      &resparams); /* address of lock        */
                                   /* resource structure    */
if (status != CLM_NORMAL)
{
      clm_perror("Lock resource name invalid.");
}
```

Index

Vos remarques sur ce document / Technical publication remark form

Titre / Title :	Bull HACMP 4.4 Programming Locking Applications

Nº Reférence / Reference Nº :	86 A2 59KX 02	**Daté / Dated :**	August 2000

ERREURS DETECTEES / ERRORS IN PUBLICATION

AMELIORATIONS SUGGEREES / SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Vos remarques et suggestions seront examinées attentivement.
Si vous désirez une réponse écrite, veuillez indiquer ci-après votre adresse postale complète.

Your comments will be promptly investigated by qualified technical personnel and action will be taken as required.
If you require a written reply, please furnish your complete mailing address below.

NOM / NAME : ______________________________ Date : __________

SOCIETE / COMPANY : ______________________________

ADRESSE / ADDRESS : ______________________________

____

Remettez cet imprimé à un responsable BULL ou envoyez-le directement à :

Please give this technical publication remark form to your BULL representative or mail to:

BULL CEDOC
357 AVENUE PATTON
B.P.20845
49008 ANGERS CEDEX 01
FRANCE

Technical Publications Ordering Form
Bon de Commande de Documents Techniques

To order additional publications, please fill up a copy of this form and send it via mail to:
Pour commander des documents techniques, remplissez une copie de ce formulaire et envoyez-la à :

BULL CEDOC	**Managers /** Gestionnaires :		
ATTN / MME DUMOULIN	**Mrs.** / Mme :	**C. DUMOULIN**	+33 (0) 2 41 73 76 65
357 AVENUE PATTON	**Mr.** / M :	**L. CHERUBIN**	+33 (0) 2 41 73 63 96
B.P.20845			
49008 ANGERS CEDEX 01	**FAX :**		+33 (0) 2 41 73 60 19
FRANCE	**E–Mail** / Courrier Electronique :		srv.Cedoc@franp.bull.fr

Or visit our web site at: / Ou visitez notre site web à:
`http://www-frec.bull.com` (PUBLICATIONS, Technical Literature, Ordering Form)

CEDOC Reference # Nº Référence CEDOC	**Qty** Qté	**CEDOC Reference #** Nº Référence CEDOC	**Qty** Qté	**CEDOC Reference #** Nº Référence CEDOC	**Qty** Qté
__ __ ____ _ [__]		__ __ ____ _ [__]		__ __ ____ _ [__]	
__ __ ____ _ [__]		__ __ ____ _ [__]		__ __ ____ _ [__]	
__ __ ____ _ [__]		__ __ ____ _ [__]		__ __ ____ _ [__]	
__ __ ____ _ [__]		__ __ ____ _ [__]		__ __ ____ _ [__]	
__ __ ____ _ [__]		__ __ ____ _ [__]		__ __ ____ _ [__]	
__ __ ____ _ [__]		__ __ ____ _ [__]		__ __ ____ _ [__]	
__ __ ____ _ [__]		__ __ ____ _ [__]		__ __ ____ _ [__]	
[__] : **no revision number means latest revision** / pas de numéro de révision signifie révision la plus récente					

NOM / NAME : ____________________ Date : __________

SOCIETE / COMPANY : ____________________

ADRESSE / ADDRESS : ____________________

____

PHONE / TELEPHONE : ____________________ FAX : ____________________

E–MAIL : ____________________

For Bull Subsidiaries / Pour les Filiales Bull :
Identification: ____________________

For Bull Affiliated Customers / Pour les Clients Affiliés Bull :
Customer Code / Code Client : ____________________

For Bull Internal Customers / Pour les Clients Internes Bull :
Budgetary Section / Section Budgétaire : ____________________

For Others / Pour les Autres :
Please ask your Bull representative. / Merci de demander à votre contact Bull.

PLACE BAR CODE IN LOWER LEFT CORNER

BULL CEDOC
357 AVENUE PATTON
B.P.20845
49008 ANGERS CEDEX 01
FRANCE

ORDER REFERENCE
86 A2 59KX 02

Utiliser les marques de découpe pour obtenir les étiquettes.
Use the cut marks to get the labels.

AIX
HACMP 4.4
Programming
Locking
Applications
86 A2 59KX 02

AIX
HACMP 4.4
Programming
Locking
Applications
86 A2 59KX 02

AIX
HACMP 4.4
Programming
Locking
Applications
86 A2 59KX 02