Service Availability™ Forum Application Interface Specification

Volume 7: Lock Service SAI-AIS-LCK-B.01.01

SERVICE AVAILABILITY™ FORUM SPECIFICATION LICENSE AGREEMENT

The Service Availability™ Forum Application Interface Specification (the "Package") found at the URL **http://www.saforum.org** is generally made available by the Service Availability Forum (the "Copyright Holder") for use in developing products that are compatible with the standards provided in the Specification. The terms and conditions which govern the use of the Package are covered by the Artistic License 2.0 of the Perl Foundation, which is reproduced here.

The Artistic License 2.0

Copyright (c) 2000-2006, The Perl Foundation.

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

This license establishes the terms under which a given free software Package may be copied, modified, distributed, and/or redistributed.

The intent is that the Copyright Holder maintains some artistic control over the development of that Package while still keeping the Package available as open source and free software.

You are always permitted to make arrangements wholly outside of this license directly with the Copyright Holder of a given Package. If the terms of this license do not permit the full use that you propose to make of the Package, you should contact the Copyright Holder and seek a different licensing arrangement.

Definitions

"Copyright Holder" means the individual(s) or organization(s) named in the copyright notice for the entire Package.

"Contributor" means any party that has contributed code or other material to the Package, in accordance with the Copyright Holder's procedures.

"You" and "your" means any person who would like to copy, distribute, or modify the Package.

"Package" means the collection of files distributed by the Copyright Holder, and derivatives of that collection and/or of those files. A given Package may consist of either the Standard Version, or a Modified Version.

"Distribute" means providing a copy of the Package or making it accessible to anyone else, or in the case of a company or organization, to others outside of your company or organization.

"Distributor Fee" means any fee that you charge for Distributing this Package or providing support for this Package to another party. It does not mean licensing fees.

"Standard Version" refers to the Package if it has not been modified, or has been modified only in ways explicitly requested by the Copyright Holder.

"Modified Version" means the Package, if it has been changed, and such changes were not explicitly requested by the Copyright Holder.

"Original License" means this Artistic License as Distributed with the Standard Version of the Package, in its current version or as it may be modified by The Perl Foundation in the future.

"Source" form means the source code, documentation source, and configuration files for the Package.

"Compiled" form means the compiled bytecode, object code, binary, or any other form resulting from mechanical transformation or translation of the Source form.

Permission for Use and Modification Without Distribution

(1) You are permitted to use the Standard Version and create and use Modified Versions for any purpose without restriction, provided that you do not Distribute the Modified Version.

Permissions for Redistribution of the Standard Version

(2) You may Distribute verbatim copies of the Source form of the Standard Version of this Package in any medium without restriction, either gratis or for a Distributor Fee, provided that you duplicate all of the original copyright notices and associated disclaimers. At your discretion, such verbatim copies may or may not include a Compiled form of the Package.

(3) You may apply any bug fixes, portability changes, and other modifications made available from the Copyright Holder. The resulting Package will still be considered the Standard Version, and as such will be subject to the Original License.

Distribution of Modified Versions of the Package as Source

(4) You may Distribute your Modified Version as Source (either gratis or for a Distributor Fee, and with or without a Compiled form of the Modified Version) provided that you clearly document how it differs from the Standard Version, including, but not limited to, documenting any non-standard features, executables, or modules, and provided that you do at least ONE of the following:

(a) make the Modified Version available to the Copyright Holder of the Standard Version, under the Original License, so that the Copyright Holder may include your modifications in the Standard Version.

(b) ensure that installation of your Modified Version does not prevent the user installing or running the Standard Version. In addition, the Modified Version must bear a name that is different from the name of the Standard Version.

(c) allow anyone who receives a copy of the Modified Version to make the Source form of the Modified Version available to others under

(i) the Original License or

(ii) a license that permits the licensee to freely copy, modify and redistribute the Modified Version using the same licensing terms that apply to the copy that the licensee received, and requires that the Source form of the Modified Version, and of any works derived from it, be made freely available in that license fees are prohibited but Distributor Fees are allowed.

Distribution of Compiled Forms of the Standard Version or Modified Versions without the Source

(5) You may Distribute Compiled forms of the Standard Version without the Source, provided that you include complete instructions on how to get the Source of the Standard Version. Such instructions must be valid at the time of your distribution. If these instructions, at any time while you are carrying out such distribution, become invalid, you must provide new instructions on demand or cease further distribution.

If you provide valid instructions or cease distribution within thirty days after you become aware that the instructions are invalid, then you do not forfeit any of your rights under this license.

(6) You may Distribute a Modified Version in Compiled form without the Source, provided that you comply with Section 4 with respect to the Source of the Modified Version.

Aggregating or Linking the Package

(7) You may aggregate the Package (either the Standard Version or Modified Version) with other packages and Distribute the resulting aggregation provided that you do not charge a licensing fee for the Package. Distributor Fees are permitted, and licensing fees for other components in the aggregation are permitted. The terms of this license apply to the use and Distribution of the Standard or Modified Versions as included in the aggregation.

(8) You are permitted to link Modified and Standard Versions with other works, to embed the Package in a larger work of your own, or to build stand-alone binary or bytecode versions of applications that include the Package, and Distribute the result without restriction, provided the result does not expose a direct interface to the Package.

Items That are Not Considered Part of a Modified Version

(9) Works (including, but not limited to, modules and scripts) that merely extend or make use of the Package, do not, by themselves, cause the Package to be a Modified Version. In addition, such works are not considered parts of the Package itself, and are not subject to the terms of this license.

General Provisions

(10) Any use, modification, and distribution of the Standard or Modified Versions is governed by this Artistic License. By using, modifying or distributing the Package, you accept this license. Do not use, modify, or distribute the Package, if you do not accept this license.

(11) If your Modified Version has been derived from a Modified Version made by someone other than you, you are nevertheless required to ensure that your Modified Version complies with the requirements of this license.

(12) This license does not grant you the right to use any trademark, service mark, tradename, or logo of the Copyright Holder.

(13) This license includes the non-exclusive, worldwide, free-of-charge patent license to make, have made, use, offer to sell, sell, import and otherwise transfer the Package with respect to any patent claims licensable by the Copyright Holder that are necessarily infringed by the Package. If you institute patent litigation (including a cross-claim or counterclaim) against any party alleging that the Package constitutes direct or contributory patent infringement, then this Artistic License to you shall terminate on the date that such litigation is filed.

(14) Disclaimer of Warranty:

THE PACKAGE IS PROVIDED BY THE COPYRIGHT HOLDER AND CONTRIBUTORS "AS IS' AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES. THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT ARE DISCLAIMED TO THE EXTENT PERMITTED BY YOUR LOCAL LAW. UNLESS REQUIRED BY LAW, NO COPYRIGHT HOLDER OR CONTRIBUTOR WILL BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING IN ANY WAY OUT OF THE USE OF THE PACKAGE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Table of Contents

Volume 7, Lock Service

1 Document Introduction

1.1 Document Purpose

This document defines the Lock Service of the Application Interface Specification (AIS) of the Service Availability™ Forum. It is intended for use by implementors of the Application Interface Specification and by application developers who would use the Application Interface Specification to develop applications that must be highly available. The AIS is defined in the C programming language, and requires substantial knowledge of the C programming language.

Typically, the Service Availability™ Forum Application Interface Specification will be used in conjunction with the Service Availability™ Forum Hardware Interface Specification (HPI) and with the Service Availability™ Forum System Management Specification, which is still under development.

1.2 AIS Documents Organization

The Application Interface Specification is organized into the following volumes:

Volume 1, the Overview document, provides a brief guide to the remainder of the Application Interface Specification. It describes the objectives of the AIS specification as well as programming models and definitions that are common to all specifications. Additionally, it contains an overview of the Availability Management Framework and of the other services as well as the system description and conceptual models, including the physical and logical entities that make up the system. Volume 1 also contains a chapter that describes the main abbreviations, concepts and terms used in the AIS documents.
The name of the pdf file containing this document for the AIS version B.01.01 is:
aisOverview.B0101.pdf

Volume 2 describes the Availability Management Framework API.
The name of the pdf file containing this document for the AIS version B.01.01 is:
aisAmf.B0101.pdf

Volume 3 describes the Cluster Membership Service API.
The name of the pdf file containing this document for the AIS version B.01.01 is:
aisClm.B0101.pdf

Volume 4 describes the Checkpoint Service API.
The name of the pdf file containing this document for the AIS version B.01.01 is:
aisCkpt.B0101.pdf

Volume 5 describes the Event Service API.
The name of the pdf file containing this document for the AIS version B.01.01 is:
aisEvt.B0101.pdf

Volume 6 describes the Message Service API.
The name of the pdf file containing this document for the AIS version B.01.01 is:
aisMsg.B0101.pdf

Volume 7 (this volume) describes the Lock Service API.
The name of the pdf file containing this document for the AIS version B.01.01 is:
aisLck.B0101.pdf

1.3 How to Provide Feedback on the Specification

If you have a question or comment about this specification, you may submit feedback online by following the links provided for this purpose on the Service Availability™ Forum website (**http://www.saforum.org**).

You can also sign up to receive information updates on the Forum or the Specification.

1.4 How to Join the Service Availability™ Forum

The Promoter Members of the Forum require that all organizations wishing to participate in the Forum complete a membership application. Once completed, a representative of the Service Availability™ Forum will contact you to discuss your membership in the Forum. The Service Availability™ Forum Membership Application can be completed online by following the pertinent links provided on the Forum's website (**http://www.saforum.org**).

You can also submit information requests online. Information requests are generally responded to within three business days.

1.5 Additional Information

1.5.1 Member Companies

A list of the Service Availability™ Forum member companies can also be viewed online by using the links provided on the Forum's website (**http://www.saforum.org**).

1.5.2 Press Materials

The Service Availability™ Forum has available a variety of downloadable resource materials, including the Forum Press Kit, graphics, and press contact information.

Visit this area often for the latest press releases from the Service Availability™ Forum and its member companies by following the pertinent links provided on the Forum's website (**http://www.saforum.org**).

2 Overview

This specification defines the Lock Service within the Application Interface Specification (AIS).

2.1 Lock Service

The Lock Service is a distributed lock service, intended for use in a cluster, where processes in different nodes might compete with each other for access to a shared resource.

The Lock Service provides entities, called lock resources, that are used to synchronize access to shared resources between application processes.

The Lock Service provides a simple lock model supporting two locking modes for exclusive access and shared access. All implementations must offer synchronous and asynchronous calls, lock timeout, trylock, and lock wait notifications. Implementations may optionally offer the additional features of deadlock detection and lock orphaning. A Lock Service interface allows an application to query for support for one or more of the optional features. If an application depends on one of the optional features for proper operation, it should use this interface to check whether the feature is provided. Maximum portability is achieved by avoiding use of the optional features. However, because they offer powerful functionality, it may make sense to take advantage of them when they are available.

The locks provided by the Lock Service are not recursive. Thus, claiming one lock does not implicitly claim another lock; rather, each lock must be claimed individually.

3 SA Lock Service API

3.1 Lock Service Model

A l**ock resource** is a globally-named resource, access to which is controlled by entities called locks.

A **lock** is a protected access to a lock resource. A **lock request** is an attempt to obtain a lock. If a process has requested a lock, and the lock is granted to the process, the process is said to hold the lock, and it is referred to in this text as a **lock holder**. A **pending (or queued) lock request** is a lock request that has not yet been granted.

Locks can be requested in one of two **lock modes**: **exclusive mode** or **shared read mode.** Only one exclusive lock can be granted against a lock resource at any time, while any number of shared locks can be granted against a lock resource so long as there is no exclusive lock granted on that lock resource. At any time a lock is held against a lock resource, there can be one or more pending lock requests. A single process can request multiple locks against a single lock resource, in addition to the more usual case of multiple processes requesting locks against any single lock resource.
Locks cannot be converted from one mode to another, but must be explicitly dropped and reacquired in the new mode. More details on lock modes are given in Section 3.1.3 below.

The Lock Service informs an application about deadlocks, provided that the Lock Service implementation supports the optional deadlock detection feature.

The effectiveness of locks depends on cooperation of the processes that use the Lock Service. It is not the responsibility of the Lock Service to define the relationship between locks and the related lock resources. Application processes must understand what the Lock Service does and does not do, and obey the access controls granted by the locks.

3.1.1 Lock Resource Name, Lock Resource Handle, Lock Id

As stated earlier, a lock resource is a globally-named, cluster-wide resource. Each process interested in acquiring locks against this lock resource needs to create a reference to the global lock resource.

The **lock resource name** is used to allow all such requests to rendezvous on the same resource. The lock resource name is a string, on which no interpretation is placed.

Each process that creates a reference to a global lock resource will receive a **lock resource handle** *(SaLckResourceHandleT*). The scope of an *SaLckResourceHandleT* is local to the process that opened the resource. The process can create multiple references to the global lock resource and will receive a different *SaLckResourceHandleT* for each such reference.

The process specifies the appropriate *SaLckResourceHandleT* against which to acquire (lock) locks against the global lock resource, and the process will receive a different **lock Id** *(SaLckLockIdT)* for each such lock request made. The scope of an *SaLckLockIdT* is local to the process acquiring the lock. An *SaLckLockIdT* is used to specify which lock to release (unlock) against a global lock resource.

3.1.2 Deadlock

To prevent deadlock, processes that use the Lock Service might choose to define a numerical order 0,1,...,n for the locks that they use. If a lock with order number j is held and another lock with order number k is claimed, k must be greater than j. Lock Service implementations that offer deadlock detection will indicate a deadlock by setting the *lockStatus* parameter to the value SA_LCK_LOCK_DEADLOCK when returning from *saLckResourceLock(),* or when executing *saLckLockGrantCallback()* if the lock was requested via *saLckResourceLockAsync().*

It is the responsibility of a process that uses the Lock Service to ensure that multiple invocations of *saLckResourceLock()* and *saLckResourceLockAsync()* do not result in deadlock.

3.1.3 Lock Modes and Lock Waiter Callback

Locks support two lock modes:

- **Protected Read (PR)** - A shared read, i.e., any number of lockers may hold a read lock and no one may hold an exclusive lock.
- **Exclusive (EX)** - Only a single locker may hold the lock.

Lock requests can be blocked. For example, a request for an EX lock is blocked and the request is queued if the lock is already held in EX mode or if one or more PR locks exist. If such is the case, it can be specified that one or more processes holding the existing lock or locks are notified via a **lock waiter callback** that they are blocking a request. It is recommended that processes provide a lock waiter callback function and specify it in the *SaLckLockWaiterCallbackT* field of the *SaLckCallbacksT* structure. However, this is not required: If the field is NULL, a lock holder will not be informed that it is blocking lock requests. The lock holders are expected to drop their locks in response to this notification. Once all current lock holders have dropped their locks, the EX request is granted. While such an EX request is pending, new PR requests are queued, rather than granted immediately.

EX lock requests should generally be handled in the order that they are made, although no guarantees are made on this. The Lock Service retains knowledge of locks and lock requests across arbitrary failures of cluster components.

It is required that existing EX lock requests take priority over PR requests. As long as any EX request is pending or held, PR requests must be queued. Only after the existing EX lock has been released and all pending EX lock requests have been granted and released, shall any queued PR requests be granted.

It is implementation-dependent what happens if an EX request arrives during the granting of a set (one or more) of queued PR requests. The Lock Service may halt granting requests, keeping those not granted on the queue, or it may continue to drain the queue by granting locks. It is required to deliver a lock waiter callback to all PR holders to which the PR lock has been granted, and which specified a non-NULL lock waiter callback function.

This model is quite useful for locks that are held for limited amounts of time, where EX locks are most generally useful. This model also allows a high degree of control of resources where lock holders can maintain a PR lock, and be informed of requests to update the resource via a lock waiter callback that an EX request is pending. The holder can drop its PR lock and immediately request it again to be queued behind the EX request, so that it can be granted as soon as the resource is updated.

3.1.4 Lock Stripping, Process Failure and Orphan Locks

Locks can be stripped from a holder under the following conditions:

- The process owning the lock fails.
- The node that hosts the process owning the lock fails.
- The process owning the lock calls *saLckResourceClose()* against the locked resource or calls *saLckFinalize()* without first releasing all of its locks.

The Lock Service handles lock stripping as both an unlock against the lock and a close operation on the lock resource. If there are no other holders of the lock (for shared locks) and no queued requests for the lock, then the Lock Service is allowed to "forget about" the lock. Requests for the lock (opens, lock calls) regard it as a new lock.

If there are other holders of the lock (for shared locks), then they continue to hold the lock, and no change is indicated to them. If there are queued requests for the lock, all grantable requests are granted, as discussed earlier for priority of requests. If queued requests remain (e.g., EX and PR requests queued, the EX is granted and the PR is queued), then these requests remain on the queue.

On implementations, which support the orphan lock feature, stripping of locks can be disabled with an option, SA_LCK_LOCK_ORPHAN, to the lock call. In this case, when the process holding the lock exits, the lock remains on the lock resource grant queue and is called an **orphan lock** or simply an **orphan**. An API call is provided to purge existing orphan locks held on a lock resource. When a lock request is blocked by an orphan, the status code SA_LCK_LOCK_ORPHANED is returned to the caller. On implementations, which do not support the orphan lock feature, the SA_LCK_LOCK_ORPHAN option is ignored, and the lock will always be stripped in the situations described, and the SA_LCK_LOCK_ORPHANED status code will not be returned.

For best performance, the orphan feature should not be enabled. However, in some cases the feature is necessary, for example, when guarding resources that cannot be updated atomically.

If a failed process has queued requests, the queued requests are simply dropped.

3.1.5 Optional Lock Service Features

The optional features are deadlock detection and orphan lock. The *saLckOptionCheck()* routine can be invoked to determine whether these optional features are supported by the Lock Service in use.

3.2 Include File and Library Names

The following statement containing declarations of data types and function prototypes must be included in the source of an application using the Lock Service API:

```
#include <saLck.h>
```

To use the Lock Service API, an application must be bound with the following library:

libSaLck.so

3.3 Type Definitions

The Lock Service uses the types described in the following sections.

3.3.1 SaLckHandleT

typedef SaUint64T SaLckHandleT;

The type of the handle supplied by the Lock Service to a process during initialization of the Lock Service and used by a process when it invokes functions of the Lock Service API so that the Lock Service can recognize the process.

3.3.2 SaLckLockIdT

typedef SaUint64T SaLckLockIdT;

A type used to identify a lock that is either held, or requested to be held, against a lock resource. The scope of an *SaLckLockIdT* is process-wide, so each one that is given to the process via *saLckResourceLock()* or *saLckResourceLockAsync()* will be unique regardless of the lock resource for which it is used.

3.3.3 SaLckResourceHandleT

typedef SaUint64T SaLckResourceHandleT;

A type used to identify a lock resource.

3.3.4 SaLckCallbacksT

The callbacks structure, supplied by a process to the Lock Service, contains the callback functions that the Lock Service may invoke.

typedef struct {

SaLckResourceOpenCallbackT saLckResourceOpenCallback;

SaLckLockGrantCallbackT saLckLockGrantCallback;

SaLckLockWaiterCallbackT saLckLockWaiterCallback;

SaLckResourceUnlockCallbackT saLckResourceUnlockCallback;

} SaLckCallbacksT;

The fields of the *SaLckCallbacksT* structure have the following interpretation:

- *saLckResourceOpenCallback()* - The callback function that is invoked when a a (cluster-wide) lock resource is created for locking operations. This callback can be a NULL value, which means that the lock resource can be opened only synchronously.

- *saLckLockGrantCallback()* - The callback function that is invoked when the lock is granted asynchronously or deadlock is detected (provided that the implementation of the Lock Service supports the optional deadlock detection feature). This callback can be a NULL value, which means that the lock can be locked only synchronously.
- *saLckLockWaiterCallback()* - This callback function is invoked to inform a current holder of a lock that this lock is blocking another lock request. The callback is invoked with the lock mode being requested that is being blocked, and with a *waiterSignal*, a value specified in the lock request that is being blocked. This callback can be a NULL value, which means that the lock holder will not be notified if it is blocking another lock request.
- *saLckResourceUnlockCallback()* - The callback function that is invoked when the lock is asynchronously unlocked. This callback can be a NULL value, which means that the lock can be unlocked only synchronously.

3.3.5 SaLckResourceOpenFlagsT

#define SA_LCK_RESOURCE_CREATE 0x1

typedef SaUint32T SaLckResourceOpenFlagsT;

The *SaLckResourceOpenFlagsT* type defines flags that can be used to control lock resource open requests. In this version of the Lock Service, only one flag is defined.

If SA_LCK_RESOURCE_CREATE is specified, and if the named lock resource does not already exist, it is created. If the open request completes successfully, a resource handle for the named lock resource is returned.

If SA_LCK_RESOURCE_CREATE is not set, the processing is as follows:

- If the named lock resource already exists, a resource handle will be returned.
- If the named lock resource does not already exist, an error of SA_AIS_ERR_NOT_EXIST is returned.

3.3.6 SaLckLockFlagsT

#define SA_LCK_LOCK_NO_QUEUE 0x1

#define SA_LCK_LOCK_ORPHAN 0x2

typedef SaUint32T SaLckLockFlagsT;

The *SaLckLockFlagsT* type defines flags that can be used to control lock requests. The interpretation of these flag values is as follows:

- SA_LCK_LOCK_NO_QUEUE - Requests that the Lock Service not queue the lock request if the lock request cannot be granted immediately. The Lock Service returns the status SA_LCK_LOCK_NOT_QUEUED in the status block. Note: This allows the process to perform a “trylock” to query if the lock can be granted immediately, and if it cannot be, drop the request rather than queuing it to be granted later.
- SA_LCK_LOCK_ORPHAN - Requests that the Lock Service not purge this lock if the process or node hosting the process holding the lock fails, or if the process holding the lock calls *saLckResourceClose()* against the locked resource or calls *saLckFinalize()* without first releasing the lock.

3.3.7 SaLckLockStatusT

An enumeration type of the possible lock status return values from the Lock Service that indicates the status of the lock itself.

typedef enum {

SA_LCK_LOCK_GRANTED = 1,

SA_LCK_LOCK_DEADLOCK = 2,

SA_LCK_LOCK_NOT_QUEUED = 3,

SA_LCK_LOCK_ORPHANED = 4,

SA_LCK_LOCK_NO_MORE = 5,

SA_LCK_LOCK_DUPLICATE_EX = 6

} SaLckLockStatusT;

The values of the *SaLckLockStatusT* enumeration type have the following interpretation:

- SA_LCK_LOCK_GRANTED - Lock request was granted in the mode requested.
- SA_LCK_LOCK_DEADLOCK - Lock request would cause deadlock. This value will only be returned if the implementation supports the optional deadlock detection feature.

- SA_LCK_LOCK_NOT_QUEUED - Lock request is blocked and would have to be queued but the request was submitted with the SA_LCK_LOCK_NO_QUEUE flag.
- SA_LCK_LOCK_ORPHANED - Lock request could not be granted because the lock is an orphan, a now-failed lock holder specified the SA_LCK_LOCK_ORPHAN flag. This value will only be returned if the implementation supports the optional orphan locks feature.
- SA_LCK_LOCK_NO_MORE - The Lock Service cannot support any more locks.
- SA_LCK_LOCK_DUPLICATE_EX - The process requesting an EX lock already holds a granted or pending EX lock against the same *SaLckResourceHandleT,* as was specified in the lock request.

3.3.8 SaLckLockModeT

typedef enum {

SA_LCK_PR_LOCK_MODE = 1,

SA_LCK_EX_LOCK_MODE = 2

} SaLckLockModeT;

The *SaLckLockModeT* enumeration type defines the possible lock modes. These lock modes have the following interpretation:

- SA_LCK_PR_LOCK_MODE - Allows the requesting process to read from a resource while other processes simultaneously read from the same resource. No process can write to the resource while one or more PR locks is held on the resource. This is an example of a shared lock.
- SA_LCK_EX_LOCK_MODE - Allows the requesting process to read from, or write to, a resource while it prevents any other process from accessing that resource.

3.3.9 SaLckOptionsT

#define SA_LCK_OPT_ORPHAN_LOCKS 0x1

#define SA_LCK_OPT_DEADLOCK_DETECTION 0x2

typedef SaUint32T SaLckOptionsT;

The *SaLckOptionsT* type is used to hold a bitmap indicating the optional features supported by an implementation of the Lock Service. The application should call the *saLckOptionCheck()* function to fill this bitmap, and then use the flags described below to determine if one or more of the optional features are provided:

SA_LCK_OPT_ORPHAN_LOCKS indicates that the Lock Services supports orphan locks.
SA_LCK_OPT_DEADLOCK_DETECTION indicates that the Lock Service supports deadlock detection.

It is allowable for an arbitrary Lock Services implementation to support none of these, one of these, or all of these optional features.

3.3.10 SaLckWaiterSignalT

typedef SaUint64T SaLckWaiterSignalT;

The *SaLckWaiterSignalT* type is used in locking requests to pass information in the case that one or more processes are already holding a lock that blocks a locking request from being granted. The Lock Service defines no meaning for the values placed in this parameter by the caller, and the interpretation of the value is up to the recipient. It is assumed that all processes contending for a lock will be related and will have agreed upon the meaning of the values passed.

3.4 Library Life Cycle

3.4.1 saLckInitialize()

Prototype

SaAisErrorT saLckInitialize(

*SaLckHandleT *lckHandle,*

*const SaLckCallbacksT *lckCallbacks*,

*SaVersionT *version*

);

Parameters

lckHandle - [out] A pointer to the handle designating this particular initialization of the Lock Service that is to be returned by the Lock Service.

lckCallbacks - [in] If *lckCallbacks* is set to NULL, no callback is registered; otherwise, it is a pointer to a *SaLckCallbacksT* structure, containing the callback functions of the process that the Lock Service may invoke. Only non-NULL callback functions in this structure will be registered.

version - [in/out] As an input parameter, *version* is a pointer to the required Lock Service version. In this case, *minorVersion* is ignored and should be set to 0x00.
As an output parameter, the version actually supported by the Lock Service is delivered.

Description

This function initializes the Lock Service for the invoking process and registers the various callback functions. This function must be invoked prior to the invocation of any other Lock Service functionality. The handle *lckHandle* is returned as the reference to this association between the process and the Lock Service. The process uses this handle in subsequent communication with the Lock Service.

If the implementation supports the required *releaseCode,* and a major version >= the required *majorVersion*, SA_AIS_OK is returned. In this case, the *version* parameter is set by this function to:

- *releaseCode* = required release code
- *majorVersion* = highest value of the major version that this implementation can support for the required *releaseCode*
- *minorVersion* = highest value of the minor version that this implementation can support for the required value of *releaseCode* and the returned value of *majorVersion*

If the above mentioned condition cannot be met, SA_AIS_ERR_VERSION is returned, and the *version* parameter is set to:

if (implementation supports the required *releaseCode*)

releaseCode = required *releaseCode*

else *{*

if (implementation supports *releaseCode* higher than the required *releaseCode*)

releaseCode = the least value of the supported release codes that is higher than the required *releaseCode*

else

releaseCode = the highest value of the supported release codes that is less than the required *releaseCode*

}

majorVersion = highest value of the major versions that this implementation can support for the returned *releaseCode*

minorVersion = highest value of the minor versions that this implementation can support for the returned values of *releaseCode* and *majorVersion*

Return Values

SA_AIS_OK - The function completed successfully.

SA_AIS_ERR_LIBRARY - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it didn't.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The process may retry later.

SA_AIS_ERR_INVALID_PARAM - A parameter is not set correctly.

SA_AIS_ERR_NO_MEMORY - Either the Lock Service library or the provider of the service is out of memory and cannot provide the service.

SA_AIS_ERR_NO_RESOURCES - There are insufficient resources (other than memory).

SA_AIS_ERR_VERSION - The version parameter is not compatible with the version of the Lock Service implementation.

See Also

saLckSelectionObjectGet(), saLckDispatch(), saLckOptionCheck(), saLckFinalize()

3.4.2 saLckSelectionObjectGet()

Prototype

```
SaAisErrorT saLckSelectionObjectGet(
        SaLckHandleT lckHandle,
        SaSelectionObjectT *selectionObject
);
```

Parameters

lckHandle - [in] The handle, obtained through the *saLckInitialize()* function, designating this particular initialization of the Lock Service.

selectionObject - [out] A pointer to the operating system handle that the process can use to detect pending callbacks.

Description

The *saLckSelectionObjectGet()* function returns the operating system handle *selectionObject,* associated with the handle *lckHandle.* The invoking process can use this handle to detect pending callbacks, instead of repeatedly invoking *saLckDispatch()* for this purpose.

In a POSIX environment, the operating system handle is a file descriptor that is used with the *poll()* or *select()* system calls to detect incoming callbacks.

The *selectionObject* returned by *saLckSelectionObjectGet()* is valid until *saLckFinalize()* is invoked on the same handle *lckHandle*.

Return Values

SA_AIS_OK - The function completed successfully.

SA_AIS_ERR_LIBRARY - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it didn’t.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The process may retry later.

SA_AIS_ERR_BAD_HANDLE - The handle *lckHandle* is invalid, since it is corrupted, uninitialized, or has already been finalized.

SA_AIS_ERR_INVALID_PARAM - A parameter is not set correctly.

SA_AIS_ERR_NO_MEMORY - Either the Lock Service library or the provider of the service is out of memory and cannot provide the service.

SA_AIS_ERR_NO_RESOURCES - There are insufficient resources (other than memory).

See Also

saLckInitialize(), saLckDispatch(), saLckOptionCheck(), saLckFinalize()

3.4.3 saLckOptionCheck()

Prototype

```
SaAisErrorT saLckOptionCheck(
        SaLckHandleT lckHandle,
        SaLckOptionsT *lckOptions
);
```

Parameters

lckHandle - [in] The handle obtained from the *saLckInitialize()* function, designating this particular initialization of the Lock Service.

lckOptions - [out] Bitmap that contains flags indicating which, if any, of the optional Lock Service features are supported by the Lock Service.

Description

This function is used to determine if any of the optional Lock Service features are provided by the used implementation of the Lock Service. For each such supported optional feature, a flag will be ORed into the *lckOptions* bitmask. After returning from this function, the application should use the flags defined in Section 3.3.9 to test for the presence of any required optional features. If a desired optional feature is not supported by this implementation of the Lock Service, it is up to the application to

determine what action to take in response to this. Note that attempts to use an unsupported optional feature, such as specifying SA_LCK_LOCK_ORPHAN when orphan locks are not supported, will result in an error.

A more subtle problem can occur if an application expects to have deadlock detection support available, but does not check whether this option is supported. If deadlock detection is not supported, the application may encounter a deadlock situation but no error will be returned to the application.

Return Values

SA_AIS_OK - The function completed successfully.

SA_AIS_ERR_LIBRARY - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

SA_AIS_ERR_BAD_HANDLE - The handle *lckHandle* is invalid, since it is corrupted, uninitialized, or has already been finalized.

SA_AIS_ERR_INVALID_PARAM - A parameter is not set correctly.

See Also

saLckInitialize()

3.4.4 saLckDispatch()

Prototype

```
SaAisErrorT saLckDispatch(
        SaLckHandleT lckHandle,
        SaDispatchFlagsT dispatchFlags
);
```

Parameters

lckHandle - [in] The handle obtained from the *saLckInitialize()* function, designating this particular initialization of the Lock Service.

dispatchFlags - [in] Flags that specify the callback execution behavior of the *saLckDispatch()* function, which have the values SA_DISPATCH_ONE, SA_DISPATCH_ALL, or SA_DISPATCH_BLOCKING, as defined in volume 1 of the AIS specification.

Description

This function invokes, in the context of the calling thread, pending callbacks for the handle *lckHandle* in a way that is specified by the *dispatchFlags* parameter.

Return Values

SA_AIS_OK - The function completed successfully.

SA_AIS_ERR_LIBRARY - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it didn't.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The process may retry later.

SA_AIS_ERR_BAD_HANDLE - The handle *lckHandle* is invalid, since it is corrupted, uninitialized, or has already been finalized.

SA_AIS_ERR_INVALID_PARAM - The *dispatchFlags* parameter is invalid.

See Also

saLckInitialize(), saLckSelectionObjectGet(), saLckOptionCheck(), saLckFinalize()

3.4.5 saLckFinalize()

Prototype

```
SaAisErrorT saLckFinalize(
        SaLckHandleT lckHandle
);
```

Parameters

lckHandle - [in] The handle, obtained through the *saLckInitialize()* function, designating this particular initialization of the Lock Service.

Description

The *saLckFinalize()* function closes the association, represented by the *lckHandle* parameter, between the invoking process and the Lock Service. The process must

have invoked *saLckInitialize()* before it invokes this function. A process must invoke this function once for each handle it acquired by invoking *saLckInitialize()*.

If the *saLckFinalize()* function returns successfully, the *saLckFinalize()* function releases all resources acquired when *saLckInitialize()* was called. Moreover, if the process holds a lock when *saLckFinalize()* is called, it releases the lock. Similarly, if the process has a lock request queued when *saLckFinalize()* is called, it drops the pending request. Furthermore, it cancels all pending callbacks related to the particular handle. Note that because the callback invocation is asynchronous, it is still possible that some callback calls are processed after this call returns successfully.

After *saLckFinalize()* is invoked, the selection object is no longer valid.

Return Values

SA_AIS_OK - The function completed successfully.

SA_AIS_ERR_LIBRARY - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it didn't.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The process may retry later.

SA_AIS_ERR_BAD_HANDLE - The handle *lckHandle* is invalid, since it is corrupted, uninitialized, or has already been finalized.

See Also

saLckInitialize()

3.5 Lock Resource Operations

3.5.1 saLckResourceOpen() and saLckResourceOpenAsync()

Prototype

```
SaAisErrorT saLckResourceOpen(
        SaLckHandleT lckHandle,
        const SaNameT *lockResourceName,
        SaLckResourceOpenFlagsT resourceFlags,
        SaTimeT timeout,
        SaLckResourceHandleT *lockResourceHandle
);

SaAisErrorT saLckResourceOpenAsync(
        SaLckHandleT lckHandle,
        SaInvocationT invocation,
        const SaNameT *lockResourceName,
        SaLckResourceOpenFlagsT resourceFlags,
);
```

Parameters

lckHandle - [in] The handle, obtained through the *saLckInitialize()* function, designating this particular initialization of the Lock Service.

invocation - [in] This parameter allows the invoking component to match this invocation of *saLckResourceOpenAsync()* with the corresponding callback call.

lockResourceName - [in] A pointer to the name of the lock resource being requested that identifies a lock resource globally in a cluster.

resourceFlags - [in] Flags that are bitwise ORed together to select multiple options and to control the behavior of the call. For information on the flag settings, see Section 3.3.5 on page 18.

timeout - [in] The *saLckResourceOpen()* invocation is considered to have failed if it does not complete by the time specified.

lockResourceHandle - [out] A pointer to the lock resource handle, which is assigned by the Lock Service and returned to the caller. This handle must be used in subsequent requests to lock, unlock, purge, and close this lock resource.

Description

The *saLckResourceOpen()* and *saLckResourceOpenAsync()* functions open a (cluster-wide) lock resource associated with *lockResourceName* for locking operations.

The *saLckResourceOpen()* function is a blocking operation.

Completion of the *saLckResourceOpenAsync()* function is signaled by an invocation of the associated *saLckResourceOpenCallback()* callback function, which must have been supplied when the process invoked the *saLckInitialize()* call. The *saLckResourceOpenCallback()* will be executed only if the *saLckResourceOpenAsync()* function returns SA_AIS_OK.

The process supplies the value of *invocation* when it invokes the *saLckResourceOpenAsync()* function, and the Lock Service passes that value of *invocation* back to the application when it invokes the corresponding *saLckResourceOpenCallback()* function that returns the lock resource handle. The *invocation* parameter is a mechanism that enables the process to determine which call triggered which callback.

If the SA_LCK_RESOURCE_CREATE flag is specified in *resourceFlags,* and a lock resource, named by *lockResourceName,* does not already exist, a lock resource is created, and a lock resource handle, designated by *lockResourceHandle,* is returned. If the SA_LCK_RESOURCE_CREATE flag is not specified in *resourceFlags,* and a lock resource, identified by *lockResourceName,* already exists, a lock resource handle, designated by *lockResourceHandle,* is returned.

The handle *lockResourceHandle is returned* synchronously for the *saLckResourceOpen()* call, and asynchronously for the *saLckResourceOpenAsync()* call. This lock resource handle is used when closing the lock resource, in callback calls, and in calls to lock, unlock, and purge.

Note that until *saLckResourceLock()* or *saLckLockGrantCallback()* is called, no lock is actually held.

Return Values

SA_AIS_OK - The function completed successfully.

SA_AIS_ERR_LIBRARY - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout occurred, or the timeout, specified by the *timeout* parameter, occurred before the call could complete. It is unspecified whether the call succeeded or whether it didn't.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The process may retry later.

SA_AIS_ERR_BAD_HANDLE - The handle *lckHandle* is invalid, since it is corrupted, uninitialized, or has already been finalized.

SA_AIS_ERR_INIT - The previous initialization with *saLckInitialize()* was incomplete, since the *saLckResourceOpenCallback()* callback function is missing. This only applies to *saLckResourceOpenAsync().*

SA_AIS_ERR_INVALID_PARAM - A parameter is not set correctly.

SA_AIS_ERR_NO_MEMORY - Either the Lock Service library or the provider of the service is out of memory and cannot provide the service.

SA_AIS_ERR_NO_RESOURCES - There are insufficient resources (other than memory).

SA_AIS_ERR_NOT_EXIST - This is returned if SA_LCK_RESOURCE_CREATE is not set in *resourceFlags* and no lock resource exists for *lockResourceName.*

SA_AIS_ERR_BAD_FLAGS - The *resourceFlags* parameter is invalid.

See Also

SaLckResourceOpenCallbackT, saLckResourceClose(), saLckInitialize()

3.5.2 SaLckResourceOpenCallbackT

Prototype

```
typedef void (*SaLckResourceOpenCallbackT)(
        SaInvocationT invocation,
        SaLckResourceHandleT lockResourceHandle,
        SaAisErrorT error
);
```

Parameters

invocation - [in] This parameter was supplied by a process in the corresponding invocation of the *saLckResourceOpenAsync()* function and is used by the Lock Service in this callback. This invocation parameter allows the process to match the invocation of that function with this callback.

lockResourceHandle - [in] The lock resource handle, which is assigned by the Lock Service and passed to the caller. This handle must be used in subsequent requests to lock, unlock, purge, and close this lock resource.

error - [in] This parameter indicates whether the *saLckResourceOpenAsync()* function was successful. The values that can be returned are:

- SA_AIS_OK - The function completed successfully.
- SA_AIS_ERR_LIBRARY - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.
- SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it didn't.
- SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The process may try again.
- SA_AIS_ERR_NO_MEMORY - Either the Lock Service library or the provider of the service is out of memory and cannot provide the service.
- SA_AIS_ERR_NO_RESOURCES - There are insufficient resources (other than memory).
- SA_AIS_ERR_NOT_EXIST - This is returned if SA_LCK_RESOURCE_CREATE was not set in *resourceFlags* of the corresponding *saLckResourceOpenAsync()* call, and no lock resource existed for *lockResourceName* of the corresponding *saLckResourceOpenAsync()* call.

Description

The Lock Service invokes this callback function when the operation requested by the invocation of *saLckResourceOpenAsync()* completes. This callback is invoked in the context of a thread issuing an *saLckDispatch()* call on the handle *lckHandle,* which was specified in the *saLckResourceOpenAsync()* call.

If the SA_LCK_RESOURCE_CREATE flag was specified in *resourceFlags* of the corresponding *saLckResourceOpenAsync()* call, and a lock resource did not exist, a lock resource for *lockResourceName* of the *saLckResourceOpenAsync()* call is created, and a lock resource handle, designated by *lockResourceHandle,* is returned.
If the SA_LCK_RESOURCE_CREATE flag was not specified in *resourceFlags* of the corresponding *saLckResourceOpenAsync()* call, and a lock resource for the *lockResourceName* of the *saLckResourceOpenAsync()* call already existed, a lock resource handle, designated by *lockResourceHandle,* is returned.

This lock resource handle is used when closing the lock resource, in callback calls, and in calls to lock, unlock, and purge.

Note that until *saLckResourceLock()* or *saLckLockGrantCallback()* is called, no lock is actually held.

Return Values

None.

See Also

saLckResourceOpenAsync(), saLckDispatch(), saLckResourceLock(), SaLckLockGrantCallbackT

3.5.3 saLckResourceClose()

Prototype

```
SaAisErrorT saLckResourceClose(
        SaLckResourceHandleT lockResourceHandle
);
```

Parameters

lockResourceHandle - [in] The handle to the lock resource to be closed. The handle *lockResourceHandle* was previously obtained via a call to one of the *saLckResourceOpen()* or *saLckResourceOpenCallback()* functions.

Description

An invocation of this function causes the association between *lockResourceHandle* and the corresponding lock resource to be deleted. The requestor should normally have released all locks against the lock resource before calling this function; however, if the requestor holds a lock in PR or EX mode it shall be dropped. If the requestor has a pending PR or EX mode lock request against the lock resource, the pending request is dropped. If other processes hold locks in PR or EX mode, their locks are unaffected, and any pending requests from other processes remain and may be processed by the Lock Service.

The requestor should not call this function unless it is certain that the requestor no longer needs to maintain any locks against this lock resource.

This call cancels all pending callbacks that refer directly or indirectly to the handle *lockResourceHandle*. Note that because the callback invocation is asynchronous, it is still possible that some callback calls are processed after this call returns successfully.

Once all references to this lock resource have been closed, the Lock Service considers that the lock resource no longer exists. However, if the implementation supports orphan locks, and there are orphan locks against this lock resource, then the lock resource will continue to exist until all of the orphan locks have been purged.

Return Values

SA_AIS_OK - The function completed successfully.

SA_AIS_ERR_LIBRARY - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it didn't.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The process may retry later.

SA_AIS_ERR_BAD_HANDLE - The handle *lockResourceHandle* is invalid, due to one or both of the reasons below:

- It is corrupted, was not obtained via the *saLckResourceOpen()* or *saLckResourceOpenCallback()* functions, or the corresponding lock resource has already been closed.
- The handle *lckHandle* that was passed to the functions *saLckResourceOpen()* or *saLckResourceOpenAsync()* has already been finalized.

See Also

saLckResourceOpen(), saLckResourceOpenAsync(), SaLckResourceOpenCallbackT

3.5.4 saLckResourceLock()

Prototype

```
SaAisErrorT saLckResourceLock(
    SaLckResourceHandleT lockResourceHandle,
    SaLckLockIdT *lockId,
    SaLckLockModeT lockMode,
    SaLckLockFlagsT lockFlags,
    SaLckWaiterSignalT waiterSignal,
    SaTimeT timeout,
    SaLckLockStatusT *lockStatus
);
```

Parameters

lockResourceHandle - [in] The handle to the lock resource on which the lock is wanted. The handle *lockResourceHandle* was previously obtained via a call to the *saLckResourceOpen()* or *saLckResourceOpenCallback()* functions.

lockId - [out] A pointer to the identifier for the lock that is returned by the Lock Service, and the invoking process is to use in subsequent calls to unlock. The *lockId* is only valid if the return value is SA_AIS_OK and the *lockStatus* parameter is SA_LCK_LOCK_GRANTED.

lockMode - [in] The requested lock mode.

lockFlags - [in] Flags that are bitwise ORed together to select multiple options and to control the behavior of the call.

waiterSignal - [in] User-specified value that will be passed to the *saLckLockWaiterCallback()* function of all processes holding locks that block this lock request.

timeout - [in] The maximum amount of time within which the lock must be granted. If the Lock Service cannot grant the lock in the requested mode within this time, the request is dropped and returned to the caller with an SA_AIS_ERR_TIMEOUT value.

lockStatus - [out] A pointer to the actual status of the lock, as defined by the *SaLckLockStatusT* enumeration type in Section 3.3.7 on page 19, that is returned by the Lock Service. This value is only valid if the return value is SA_AIS_OK.

Description

This function may be used to acquire a lock on a lock resource synchronously. The lock resource must have already been created via the *saLckResourceOpen()* or *saLckResourceOpenAsync()* functions.

Implementations that support deadlock detection will detect deadlocks and indicate in the *lockStatus* field that granting the lock would cause a deadlock. No *saLckLockGrantCallback()* is used for this lock request.

If no error occurs, an invocation of *saLckResourceLock()* returns SA_AIS_OK, and the actual status of the lock is indicated in the *lockStatus* out parameter. It is possible that the lock may not have been granted even if SA_AIS_OK is returned.

The normal case is that the lock will have been granted in the requested mode, which is indicated by the *lockStatus* parameter having the value SA_LCK_LOCK_GRANTED. Other values of the *lockStatus* parameter indicate conditions in which the lock has not been granted in the requested mode. It is the responsibility of the invoking process to resolve the condition that caused this situation, and to resubmit the lock request.

After the lock is granted, it is possible for a lock waiter notification to be received for this lock via the *saLckLockWaiterCallback()* call if this callback was specified when the process initialized the Lock Service via *saLckInitialize()*.

The lock may be released either synchronously or asynchronously. In addition, a process can mix synchronous and asynchronous requests for different locks.

Return Values

SA_AIS_OK - The lock request completed successfully.

SA_AIS_ERR_LIBRARY - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout occurred, or the timeout, specified by the *timeout* parameter, occurred before the call could complete. It is unspecified whether the call succeeded or whether it didn't.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The process may retry later.

SA_AIS_ERR_BAD_HANDLE - The handle *lockResourceHandle* is invalid, due to one or both of the reasons below:

- It is corrupted, was not obtained via the *saLckResourceOpen()* or *saLckResourceOpenCallback()* functions, or the corresponding lock resource has already been closed.
- The handle *lckHandle* that was passed to the functions *saLckResourceOpen()* or *saLckResourceOpenAsync()* has already been finalized.

SA_AIS_ERR_INVALID_PARAM - A parameter is set incorrectly.

SA_AIS_ERR_NO_MEMORY - Either the Lock Service library or the provider of the service is out of memory and cannot provide the service.

SA_AIS_ERR_NO_RESOURCES - There are insufficient resources (other than memory).

SA_AIS_ERR_BAD_FLAGS - The *lockFlags* parameter is invalid.

SA_AIS_ERR_NOT_SUPPORTED - An optional feature is specified for use in the *lockFlags* parameter but the implementation does not support the optional feature.

See Also

SaLckLockWaiterCallbackT, saLckResourceLockAsync(), saLckResourceUnlock(), saLckResourceUnlockAsync(), SaLckResourceUnlockCallbackT, saLckResourceOpen(), *SaLckResourceOpenCallbackT*

3.5.5 saLckResourceLockAsync()

Prototype

```
SaAisErrorT SaLckResourceLockAsync(
        SaLckResourceHandleT lockResourceHandle,
        SaInvocationT invocation,
        SaLckLockIdT *lockId,
        SaLckLockModeT lockMode,
        SaLckLockFlagsT lockFlags,
        SaLckWaiterSignalT waiterSignal
);
```

Parameters

lockResourceHandle - [in] The handle to the lock resource on which the lock is wanted. The handle *lockResourceHandle* was previously obtained via a call to the *saLckResourceOpen()* or *saLckResourceOpenCallback()* functions.

invocation - [in] An argument that is passed to the *saLckLockGrantCallback()* callback function when it is invoked for this locking request to identify the asynchronous lock request being processed by the callback. This allows processes to have multiple asynchronous lock requests outstanding. Because the asynchronous requests may not be resolved by the Lock Service in the same order as they were issued by the process, this value of invocation is returned to the appropriate callback function. Typically, each such request should use a unique value of invocation to enable the processes to identify the lock request for which the callback function is being invoked, as the callback will need to use the invocation value it receives to index the resource and lock identification that is part of this lock request.

lockId - [out] A pointer to the identifier for the lock that is returned by the Lock Service, and the invoking process is to use in subsequent calls to unlock. The *lockId* is only valid if the return value is SA_AIS_OK. The lock request is considered pending, until the *saLckLockGrantCallback()* function is executed, granting the lock.

lockMode - [in] The requested lock mode.

lockFlags - [in] Flags that are ORed together to select multiple options and to control the behavior of the call.

waiterSignal - [in] User-specified value that will be passed to the *saLckLockWaiterCallback()* function of all processes holding locks that block this lock request.

Description

This function can be used to request asynchronously a lock on a lock resource. The lock resource must have already been created via a *saLckResourceOpen()* or a *saLckResourceOpenAsync()* call.

The value returned directly from the invocation of this function does not indicate the status of the lock, rather it indicates whether the Lock Service has accepted the request for processing or whether it hasn't.

The actual status of the lock request is indicated through execution of the *saLckLockGrantCallback()* callback function, which must have been supplied when the process invoked the *saLckInitialize()* call. If *saLckResourceLockAsync()* returns SA_AIS_OK, the *lockId* is valid, and the lock request is considered pending, until the *saLckLockGrantCallback()* is executed. Prior to that time, the lock request can be canceled using either the *saLckResourceUnlock()* or the *saLckResourceUnlockAsync()* function with this *lockId*. If the *saLckLockGrantCallback()* call indicates an error, then, at that time, the *lockId* is no longer valid.

It is allowable to use the SA_LCK_LOCK_NO_QUEUE flag with the *saLckResourceLockAsync()* function. In this case, if the lock is already held in a mode that would block this request, *saLckResourceLockAsync()* will return SA_AIS_OK, and the *saLckLockGrantCallback()* will return the value SA_LCK_LOCK_NO_QUEUE in the *lockStatus* field.

Return Values

SA_AIS_OK - The lock request completed successfully.

SA_AIS_ERR_LIBRARY - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it didn't.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The process may retry later.

SA_AIS_ERR_BAD_HANDLE - The handle *lockResourceHandle* is invalid, due to one or both of the reasons below:

- It is corrupted, was not obtained via the *saLckResourceOpen()* or *saLckResourceOpenCallback()* functions, or the corresponding lock resource has already been closed.
- The handle *lckHandle* that was passed to the functions *saLckResourceOpen()* or *saLckResourceOpenAsync()* has already been finalized.

SA_AIS_ERR_INIT - The previous initialization with *saLckInitialize()* was incomplete, since the *saLckLockGrantCallback()* callback function is missing.

SA_AIS_ERR_INVALID_PARAM - A parameter is set incorrectly.

SA_AIS_ERR_NO_MEMORY - Either the Lock Service library or the provider of the service is out of memory and cannot provide the service.

SA_AIS_ERR_NO_RESOURCES - There are insufficient resources (other than memory).

SA_AIS_ERR_BAD_FLAGS - The *lockFlags* parameter is invalid.

SA_AIS_ERR_NOT_SUPPORTED - An optional feature is specified for use in the *lockFlags* parameter but the implementation does not support the optional feature.

See Also

SaLckLockGrantCallbackT, SaLckLockWaiterCallbackT, saLckResourceLock(), saLckResourceUnlock(), saLckResourceUnlockAsync(), SaLckResourceUnlockCallbackT, saLckInitialize(), saLckResourceOpen(), *saLckResourceOpenAsync()*

3.5.6 SaLckLockGrantCallbackT

Prototype

*typedef void (*SaLckLockGrantCallbackT)(*

SaInvocationT invocation,

SaLckLockStatusT lockStatus,

SaAisErrorT error

);

Parameters

invocation - [in] The argument that was passed in during the invocation of the *saLckResourceLockAsync()* call that established this lock request. The callback func-

tion should use this value to determine the resource, lock mode, and lock identification for this lock request.

lockStatus - [in] The status of the lock, defined by the *SaLckLockStatusT* enumeration type in Section 3.3.7 on page 19. This value is only valid if the *error* parameter is SA_AIS_OK. It is the responsibility of the invoked process to check this parameter to determine whether the lock has been granted.

error - [in] The Lock Service supplies one of the following return values as an in parameter to this function. If the return value is not SA_AIS_OK, then the value of *lockStatus* is undefined.

- SA_AIS_OK - The lock request completed successfully. Check *lockStatus* for the status of the lock.
- SA_AIS_ERR_LIBRARY - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.
- SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it didn't.
- SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The process may retry later.
- SA_AIS_ERR_INVALID_PARAM - *A* parameter is invalid.
- SA_AIS_ERR_NO_MEMORY - Either the Lock Service library or the provider of the service is out of memory and cannot provide the service.
- SA_AIS_ERR_NO_RESOURCES - There are insufficient resources (other than memory).
- SA_AIS_ERR_NOT_EXIST - The *lockId* identifier that was specified in the *saLckResourceLockAsync()* call has become invalid prior to the invocation of this callback, due to the lock request having been canceled in the period between *saLckResourceLockAsync()* and the execution of this callback.

Description

This callback function is invoked by the Lock Service when a process requested a lock via the *saLckResourceLockAsync()* function. Once the lock request has been resolved, this callback is executed to inform the process. This callback is invoked in the context of a thread issuing an *saLckDispatch()* call on the handle *lckHandle,* which was specified in the *saLckResourceOpen()* or *saLckResourceOpenAsync()* call, leading to the handle *lockResourceHandle,* specified in the corresponding *saLckResourceLockAsync()* call. The normal case is that the lock will have been

granted in the requested mode. This is indicated via the SA_LCK_LOCK_GRANTED value being set in the *lockStatus* parameter.

Implementations that support deadlock detection will detect deadlocks and indicate in the *lockStatus* field that granting the lock would cause a deadlock.

Other values possible in the *lockStatus* parameter indicate possible conditions where the lock has not been granted in the requested mode. It is the responsibility of the process to resolve the condition that caused this situation, and to resubmit the lock request at a later time.

Return Values

None.

See Also

saLckResourceLockAsync(), saLckDispatch(), saLckResourceOpen(), saLckResourceOpenAsync()

3.5.7 SaLckLockWaiterCallbackT

Prototype

```
typedef void (*SaLckLockWaiterCallbackT)(
        SaLckWaiterSignalT waiterSignal,
        SaLckLockIdT lockId,
        SaLckLockModeT modeHeld,
        SaLckLockModeT modeRequested
);
```

Parameters

waiterSignal - [in] User-specified value from the *saLckResourceLock()* or *saLckResourceLockAsync()* call that requested this lock. It is up to the user to determine the meaning of the given value, for instance, priority, or other type of information.

lockId - [in] The lock identifier that was returned by the Lock Service to the process through the *saLckResourceLock()* or *saLckResourceLockAsync()* calls.

modeHeld - [in] The lock mode held by the process.

modeRequested - [in] The requested lock mode that is being blocked by the process's current lock.

Description

This callback function is invoked by the Lock Service when a process holds a lock that is blocking another lock request. For example, the process may hold the lock in PR mode and another process is requesting it in EX mode. The parameters of the callback function specify the mode in which the process holds the lock and the mode that is being requested and is blocked.

The *saLckLockWaiterCallback()* is invoked for every lock request that is blocked by this process. For example, if three other processes issue lock requests that are blocked by this process, the lock holder will have this callback executed three times. Each such lock holder (for example, a number of different processes holding PR locks against the lock resource) will each be so notified via this callback.

In addition, a single process that holds multiple PR locks against a single lock resource will have this callback executed against each different *lockId*, i.e., once for each PR lock held.

The *waiterSignal* is specified by a process requesting a lock for delivery to the holder of the lock blocking that request. It is assumed that the processes contending for a lock resource will understand and act upon it.

The Lock Service enforces no action to be taken by the process when it executes this callback. The process must assist in managing its locking interactions. One assumed action is that the process drops its lock to allow another process to acquire it, but that is the responsibility of the process to decide.

This callback is invoked in the context of a thread issuing an *saLckDispatch()* call on the handle *lckHandle,* which was specified in the *saLckResourceOpen()* or *saLckResourceOpenAsync()* call, leading to the handle *lockResourceHandle,* which was specified in the corresponding *saLckResourceLock()* or *saLckResourceLockAsync()* call to obtain the *lockId*.

See Also

saLckResourceLock(), saLckResourceLockAsync(), saLckLockGrantCallback(), saLckDispatch(), saLckResourceOpen(), saLckResourceOpenAsync()

3.5.8 saLckResourceUnlock()

Prototype

SaAisErrorT saLckResourceUnlock(

SaLckLockIdT lockId,

SaTimeT timeout

);

Parameters

lockId - [in] The identifier of the lock to be released or of the pending lock request to be canceled. This identifier was returned by the Lock Service to the process through a previous *saLckResourceLock()* or *saLckResourceLockAsync()* call. The *lockId* is no longer valid once the lock has been successfully released.

timeout - [in] The *saLckResourceUnlock()* invocation is considered to have failed if it does not complete by the time specified.

Description

An invocation of this function releases synchronously the lock identified by *lockId*. If the *lockId* identifies a pending lock request, then the pending lock request will be canceled.

The return value of this function indicates whether the function succeeded or failed.

Return Values

SA_AIS_OK - The unlock request was completed successfully.

SA_AIS_ERR_LIBRARY - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout occurred, or the timeout, specified by the *timeout* parameter, occurred before the call could complete. It is unspecified whether the call succeeded or whether it didn't.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The process may retry later.

SA_AIS_ERR_BAD_HANDLE - The handle *lockResourceHandle,* which was specified in the *saLckResourceOpen()* or *saLckResourceOpenAsync()* call, leading to the handle *lockResourceHandle,* which was specified in the corresponding

saLckResourceLock() or *saLckResourceLockAsync()* call to obtain the *lockId* has become invalid due to one or both of the reasons below:

- It is corrupted, or the corresponding lock resource has already been closed.
- The handle *lckHandle* that was passed to the functions *saLckResourceOpen()* or *saLckResourceOpenAsync()* has already been finalized.

SA_AIS_ERR_INVALID_PARAM - A parameter is set incorrectly.

SA_AIS_ERR_NOT_EXIST - The *lockId* identifier is invalid, because of one of the reasons below:

- The lock has already been unlocked.
- It is either corrupted, or it was not obtained via the *saLckResourceLock()* or *saLckResourceLockAsync()* functions.

See Also

saLckResourceLock(), saLckResourceLockAsync(), SaLckLockGrantCallbackT, saLckResourceUnlockASync()

3.5.9 saLckResourceUnlockAsync()

Prototype

```
SaAisErrorT saLckResourceUnlockAsync(
        SaInvocationT invocation,
        SaLckLockIdT lockId
);
```

Parameters

invocation - [in] An argument that is passed to the *saLckResourceUnlockCallback()* callback function when it is invoked for this unlock request to identify the asynchronous unlock request being processed by the callback. This allows processes to have multiple asynchronous unlock requests outstanding. Because the asynchronous requests may not be resolved by the Lock Service in the same order as they were issued by the process, this value of *invocation* is returned to the appropriate callback function. Typically, each such request should use a unique value of *invocation* to enable the processes to identify the unlock request for which the callback function is being invoked, as the callback function will need to use the *invocation* value it receives to index the resource and lock identification that is part of this unlock request.

lockId - [in] The identifier of the lock to be released or of the pending lock request to be canceled. This identifier was returned by the Lock Service to the process through a previous *saLckResourceLock()* or *saLckResourceLockAsync()* call.

Description

An invocation of this function releases asynchronously the lock identified by *lockId*. If the *lockId* identifies a pending lock request, then the pending lock request will be canceled.

The value returned directly from invocation of this function does not indicate the status of the lock; it rather indicates whether the request has been accepted for processing by the Lock Service.

The results of a successful unlock submission are returned via execution of the *saLckResourceUnlockCallback()* callback function, which must have been supplied when the process invoked the *saLckInitialize()* call.

Return Values

SA_AIS_OK - The unlock request was submitted successfully.

SA_AIS_ERR_LIBRARY - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it didn't.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The process may retry later.

SA_AIS_ERR_BAD_HANDLE - The handle *lockResourceHandle,* which was specified in the *saLckResourceOpen()* or *saLckResourceOpenAsync()* call, leading to the handle *lockResourceHandle,* which was specified in the corresponding *saLckResourceLock()* or *saLckResourceLockAsync()* call to obtain the *lockId* has become invalid due to one or both of the reasons below:

- It is corrupted, or the corresponding lock resource has already been closed.
- The handle *lckHandle* that was passed to the functions *saLckResourceOpen()* or *saLckResourceOpenAsync()* has already been finalized.

SA_AIS_ERR_INIT - The previous initialization with *saLckInitialize()* was incomplete, since the *saLckResourceUnlockCallback()* callback function is missing.

SA_AIS_ERR_INVALID_PARAM - A parameter is set incorrectly.

SA_AIS_ERR_NOT_EXIST - The *lockId* identifier is invalid, because of one of the reasons below:

- The lock has already been unlocked.
- It is either corrupted, or it was not obtained via the *saLckResourceLock()* or *saLckResourceLockAsync()* functions.

See Also

SaLckResourceUnlockCallbackT, saLckResourceLock(), saLckResourceLockAsync(), SaLckLockGrantCallbackT, saLckResourceUnlock(), saLckInitialize(), saLckResourceOpen(), saLckResourceOpenAsync()

3.5.10 SaLckResourceUnlockCallbackT

Prototype

```
typedef void (*SaLckResourceUnlockCallbackT)(
        SaInvocationT invocation,
        SaAisErrorT error
);
```

Parameters

invocation - [in] The argument that was passed in during the invocation of the corresponding *saLckResourceUnlockAsync()* call that established this unlock request. The callback should use this value to determine the resource and lock identification for this unlock request.

error - [in] The Lock Service supplies one of the following return values as an in parameter to this function.

- SA_AIS_OK - The unlock request was completed successfully.
- SA_AIS_ERR_LIBRARY - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.
- SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it didn't.
- SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The process may retry later.
- SA_AIS_ERR_INVALID_PARAM - A parameter is set incorrectly.

- SA_AIS_ERR_NOT_EXIST - The *lockId* identifier that was specified in *saLckResourceUnlockAsync()* has become invalid prior to the invocation of this callback, due to the lock having already been unlocked in the period between *saLckResourceUnlockAsync()* and the execution of this callback.

Description

The Lock Service invokes this callback function when the operation requested by the invocation of *saLckResourceUnlockAsync()* to release a lock or cancel a pending lock request completes. This callback is invoked in the context of a thread issuing an *saLckDispatch()* call on the handle *lckHandle,* which was specified in the *saLckResourceOpen()* or *saLckResourceOpenAsync()* call, leading to the handle *lockResourceHandle,* which was specified in the corresponding *saLckResourceLock()* or *saLckResourceLockAsync()* call to obtain the *lockId* of the corresponding *saLckResourceUnlockAsync()* call. If successful, the lock has been released; otherwise, an error is returned in the error parameter.

Return Values

None.

See Also

saLckResourceUnlockASync(), saLckResourceLock(), saLckResourceLockAsync(), SaLckLockGrantCallbackT, saLckDispatch(), saLckResourceOpen(), saLckResourceOpenAsync()

3.5.11 saLckLockPurge()

Prototype

SaAisErrorT saLckLockPurge(

SaLckResourceHandleT lockResourceHandle

);

Parameters

lockResourceHandle - [in] The handle to the lock resource on which one or more orphaned locks are held. The handle *lockResourceHandle* was previously obtained via a call to the *saLckResourceOpen()* or *saLckResourceOpenCallback()* functions.

Description

Orphaned locks are locks that were acquired with the SA_LCK_LOCK_ORPHAN flag set and that have not been unlocked properly by the owner process before it called *saLckFinalize()* on the handle used to open the associated lock resource, or closed the associated lock resource, or died.

If a lock request is not granted because an orphan lock still exists, *saLckLockPurge()* can be used to purge the existing lock or locks. When invoked, it purges all existing orphan locks held on the lock resource.

Purging a lock is equivalent to the invocation of the *saLckResourceUnlock()* (or *saLckResourceUnlockAsync())* and *saLckResourceClose()* functions by the process whose lock was stripped and orphaned. If other processes hold the lock in shared mode, they continue to do so. Queued lock requests that were blocked can be handled in the now-purged lock condition.

Return Values

SA_AIS_OK - The purge request completed successfully.

SA_AIS_ERR_LIBRARY - An unexpected problem occurred in the library (such as corruption). The library cannot be used anymore.

SA_AIS_ERR_TIMEOUT - An implementation-dependent timeout occurred before the call could complete. It is unspecified whether the call succeeded or whether it didn't.

SA_AIS_ERR_TRY_AGAIN - The service cannot be provided at this time. The process may retry later.

SA_AIS_ERR_BAD_HANDLE - The handle *lockResourceHandle* is invalid, due to one or both of the reasons below:

- It is corrupted, was not obtained via the *saLckResourceOpen()* or *saLckResourceOpenCallback()* functions, or the corresponding lock resource has already been closed.
- The handle *lckHandle* that was passed to the functions *saLckResourceOpen()* or *saLckResourceOpenAsync()* has already been finalized.

SA_AIS_ERR_NOT_SUPPORTED - This implementation does not support the optional orphan locks feature.

See Also

saLckResourceLock(), saLckResourceLockAsync(), SaLckLockGrantCallbackT, saLckResourceUnlock(), saLckResourceUnlockAsync(),

SaLckResourceUnlockCallbackT, saLckResourceOpen(), SaLckResourceOpenCallbackT, saLckFinalize()