Technical Standard

Extended API Set Part 3

The Open Group

Technical Standard

Extended API Set Part 3

ISBN: 1-931624-68-2
Document Number: C064

Published in the U.K. by The Open Group, October 2006.

Any comments relating to the material contained in this document may be submitted to:

The Open Group
Thames Tower
37-45 Station Road
Reading
Berkshire, RG1 1LX
United Kingdom

or by Electronic Mail to:

OGSpecs@opengroup.org

Contents

Preface

The Open Group

The Open Group is a vendor-neutral and technology-neutral consortium, whose vision of Boundaryless Information Flow will enable access to integrated information within and between enterprises based on open standards and global interoperability. The Open Group works with customers, suppliers, consortia, and other standards bodies. Its role is to capture, understand, and address current and emerging requirements, establish policies, and share best practices; to facilitate interoperability, develop consensus, and evolve and integrate specifications and Open Source technologies; to offer a comprehensive set of services to enhance the operational efficiency of consortia; and to operate the industry's premier certification service, including UNIX certification.

Further information on The Open Group is available at *www.opengroup.org*.

The Open Group has over 15 years' experience in developing and operating certification programs and has extensive experience developing and facilitating industry adoption of test suites used to validate conformance to an open standard or specification.

More information is available at *www.opengroup.org/certification*.

The Open Group publishes a wide range of technical documentation, the main part of which is focused on development of Technical and Product Standards and Guides, but which also includes white papers, technical studies, branding and testing documentation, and business titles. Full details and a catalog are available at *www.opengroup.org/bookstore*.

As with all *live* documents, Technical Standards and Specifications require revision to align with new developments and associated international standards. To distinguish between revised specifications which are fully backwards-compatible and those which are not:

- A new *Version* indicates there is no change to the definitive information contained in the previous publication of that title, but additions/extensions are included. As such, it *replaces* the previous publication.
- A new *Issue* indicates there is substantive change to the definitive information contained in the previous publication of that title, and there may also be additions/extensions. As such, both previous and new documents are maintained as current publications.

Readers should note that Corrigenda may apply to any publication. Corrigenda information is published at *www.opengroup.org/corrigenda*.

This Document

This document has been prepared by The Open Group Base Working Group. The Open Group Base Working Group is considering submitting a number of API sets to the Austin Group as input to the revision of the Base Specifications, Issue 6.

This is the third document in that set.

Trademarks

Boundaryless Information Flow™ and TOGAF™ are trademarks and Motif®, Making Standards Work®, OSF/1®, The Open Group®, UNIX®, and the ‘‘X’’ device are registered trademarks of The Open Group in the United States and other countries.

Acknowledgements

The contributions of the following to the development of this document are gratefully acknowledged:

- The Open Group Base Working Group

Chapter 1

Introduction

1.1 Scope

The purpose of this document is to define a set of new API extensions to further increase application capture and hence portability for systems built upon the Single UNIX Specification, Version 3.

The scope of this set of extensions has been to introduce the concept of robust mutexes which allow for recovery from some failure situations when programming with threads.

1.2 Relationship to Other Formal Standards

No decision has been made on whether these interfaces will be added to a future Technical Standard of The Open Group, how these interfaces would announce themselves in the name space, or whether related interfaces should be merged with existing reference pages. This Technical Standard is being forwarded to the Austin Group for consideration as input to the revision of the Base Specifications, Issue 6.

Chapter 2

Changes to the Base Definitions Volume

It is proposed that these additions comprise a new option, called the Robust Mutexes option.

2.1 Section 1.2, Conformance

Add _POSIX_ROBUST_MUTEXES in order in the list of options an implementation may support (on Page 19, after Line 704).

Add a new paragraph on Page 20, after Line 727:

If the symbolic constant _POSIX_ROBUST_MUTEXES is defined to have a value other than –1, then the symbolic constants _POSIX_THREADS and _POSIX_THREAD_PROCESS_SHARED shall also be defined to have a value other than –1.

2.2 Section 1.5.1, Codes

Add a new margin code as follows:

RM Robust Mutexes

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the RM margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the RM margin legend.

Notes:

1. This section is repeated in XBD, XSH, and XCU and therefore will appear in XBD (Section 1.5.1), XSH (Section 1.8.1), and XCU (Section 1.8.1).
2. The use of RM as a margin code is a placeholder and may change in the final publication.

2.3 Chapter 3, Definitions

Add the following new definition in alphabetic order:

n.n.n Robust Mutex

A mutex with the *robust* attribute set.

Note: The *robust* attribute is defined in detail by *pthread_mutexattr_getrobust*( ).

2.4 Chapter 13, Headers

The following header file reference pages will need the following additions.

<errno.h>

The following will be added to the list of values for *errno*:

RM [ENOTRECOVERABLE]
State not recoverable.

RM [EOWNERDEAD] Previous owner died.

<pthread.h>

The following will be added to the list of symbols defined by **<pthread.h>** (Lines 10236-).

PTHREAD_MUTEX_ROBUST
PTHREAD_MUTEX_STALLED

<unistd.h>

The following will be added on Page 405, after Line 14237:

RM _POSIX_ROBUST_MUTEXES
The implementation supports the Robust Mutexes option. If this symbol is defined in **<unistd.h>**, it shall be defined to be –1, 0, or 200ymmL. The value of this symbol reported by *sysconf*( ) shall be –1 or 200ymmL.

Note: 200ymmL is to be replaced by the year and month of approval of the standard.

The following will be added to the list of constants available for *sysconf*( ) on Page 414, after Line 14637:

_SC_ROBUST_MUTEXES

Chapter 3

Changes to the System Interfaces Volume

3.1 Section 2.9.3, Thread Mutexes

Add the following text:

A problem can occur if a process terminates while one of its threads holds a mutex lock. Depending on the mutex type, it might be possible for another thread to unlock the mutex and recover the state of the mutex. However, it is difficult to perform this recovery reliably.

Robust mutexes provide a means to enable the implementation to notify other threads in the event of a process terminating while one of its threads holds a mutex lock. The next thread that acquires the mutex is notified about the termination by the return value [EOWNERDEAD] from the locking function. The notified thread can then attempt to recover the state protected by the mutex, and if successful mark the state protected by the mutex as consistent by a call to *pthread_mutex_consistent*( ). If the notified thread is unable to recover the state, it can declare the state as not recoverable by a call to *pthread_mutex_unlock*( ) without a prior call to *pthread_mutex_consistent*( ).

Whether or not the state protected by a mutex can be recovered is dependent solely on the application using robust mutexes. The robust mutex support provided in the implementation provides notification only that a mutex owner has terminated while holding a lock, or that the state of the mutex is not recoverable.

In the Rationale (Informative) volume of IEEE Std 1003.1-2001, add the following new section on Page 159, after Line 6655:

n.n.n Robust Mutexes

Robust mutexes are intended to protect applications that use mutexes to protect data shared between different processes. If a process is terminated by a signal while a thread is holding a mutex, there is no chance for the process to clean up after it. Waiters for the locked mutex might wait indefinitely.

With robust mutexes the problem can be solved: whenever a fatal signal terminates a process, current or future waiters of the mutex are notified about this fact. The locking function provides notification of this condition through the error condition [EOWNERDEAD]. A thread then has the chance to clean up the state protected by the mutex and mark the state as consistent again by a call to *pthread_mutex_consistent*( ).

Pre-existing implementations have used the semantics of robust mutexes for a variety of situations, some of them not defined in the standard. Where a normally terminated process (i.e., when one thread calls *exit*( )) causes notification of other waiters of robust mutexes if the mutex is locked by any thread in the process. This behavior is defined in the standard and makes sense because no thread other than the thread calling *exit*( ) has the chance to clean up its data.

If a thread is terminated by cancellation or if it calls *pthread_exit*( ), the situation is different. In both these situations the thread has the chance to clean up after itself by registering appropriate cleanup handlers. There is no real reason to demand that other waiters for a robust mutex the terminating thread owns are notified. The committee felt that this is actively encouraging bad practice because programmers are tempted to rely on the robust mutex semantics instead of correctly cleaning up after themselves.

Therefore, the standard does not require notification of other waiters at the time a thread is terminated while the process continues to run. The mutex is still recognized as being locked by the process (with the thread gone it makes no sense to refer to the thread owning the mutex). Therefore, a terminating process will cause notifications about the dead owner to be sent to all waiters. This delay in the notification is not required, but programmers cannot rely on prompt notification after a thread is terminated.

For the same reason is it not required that an implementation supports robust mutexes that are not shared between processes. If a robust mutex is used only within one process, all the cleanup can be performed by the threads themselves by registering appropriate cleanup handlers. Fatal signals are of no importance in this case because after the signal is delivered there is no thread remaining to use the mutex.

Some implementations might choose to support intra-process robust mutexes and they might also send notification of a dead owner right after the previous owner died. But applications must not rely on this. Applications should only use robust mutexes for the purpose of handling fatal signals in situations where inter-process mutexes are in use.

3.2 Reference Pages

NAME

pthread_cond_timedwait, pthread_cond_wait — wait on a condition

DESCRIPTION

Add the following text to the end of Line 33278:

RM If *mutex* is a robust mutex where an owner terminated while holding the lock and the state is recoverable, the mutex shall be acquired even though the function returns an error code.

ERRORS

Add the following text after Line 33330:

RM [EOWNERDEAD] The mutex is a robust mutex and the process containing the previous owner thread terminated while holding the mutex lock. The mutex lock has been acquired and it is up to the new owner to make the state consistent.

RM [ENOTRECOVERABLE] The state protected by the mutex is not recoverable. The mutex is not locked.

Add the following text after Line 33332:

RM [EOWNERDEAD] The mutex is a robust mutex and the previous owning thread terminated while holding the mutex lock. The mutex lock has been acquired and it is up to the new owner to make the state consistent.

APPLICATION USAGE

Add the following text:

Applications that have assumed that non-zero return values are errors will need updating for use with robust mutexes, since a valid return for a thread acquiring a mutex which is protecting a currently inconsistent state is [EOWNERDEAD]. Applications that do not check the error returns, due to ruling out the possibility of such errors arising, should not use robust mutexes. If an application is supposed to work with normal and robust mutexes, it should check all return values for error conditions and if necessary take appropriate action.

NAME

pthread_mutex_consistent — mark state protected by robust mutex as consistent

SYNOPSIS

RM
```
#include <pthread.h>

int pthread_mutex_consistent(pthread_mutex_t *mutex);
```

DESCRIPTION

If *mutex* is a robust mutex in an inconsistent state, the *pthread_mutex_consistent*( ) function can be used to mark the state protected by the mutex referenced by *mutex* as consistent again.

If an owner of a robust mutex terminates while holding the mutex, the mutex becomes inconsistent and the next thread that acquires the mutex lock shall be notified of the state by the return value [EOWNERDEAD]. In this case, the mutex does not become normally usable again until the state is marked consistent.

If the thread which acquired the mutex lock with the return value [EOWNERDEAD] terminates before calling either *pthread_mutex_consistent*( ) or *pthread_mutex_unlock*( ), the next thread that acquires the mutex lock shall be notified about the state of the mutex by the return value [EOWNERDEAD].

RETURN VALUE

Upon successful completion, the *pthread_mutex_consistent*( ) function shall return zero. Otherwise, an error value shall be returned to indicate the error.

ERRORS

The *pthread_mutex_consistent*( ) function shall fail if:

[EINVAL] The mutex object referenced by *mutex* is not robust or does not protect an inconsistent state.

The *pthread_mutex_consistent*( ) function may fail if:

[EINVAL] The value *mutex* is invalid.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

The *pthread_mutex_consistent*( ) function is only responsible for notifying the implementation that the state protected by the mutex has been recovered and that normal operations with the mutex can be resumed. It is the responsibility of the application to recover the state so it can be reused. If the application is not able to perform the recovery, it can notify the implementation that the situation is unrecoverable by a call to *pthread_mutex_unlock*( ) without a prior call to *pthread_mutex_consistent*( ), in which case subsequent threads that attempt to lock the mutex will fail to acquire the lock and be returned [ENOTRECOVERABLE].

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_mutexattr_getrobust( ), *pthread_mutexattr_setrobust*( ), *pthread_mutex_unlock*( ), the Base Definitions volume of IEEE Std 1003.1-2001, **<pthread.h>**

CHANGE HISTORY

First released in Issue X.

NAME

pthread_mutex_destroy, pthread_mutex_init — destroy and initialize a mutex

SYNOPSIS

THR
```
#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
    const pthread_mutexattr_t *restrict attr);
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

ERRORS

Add the following text after Line 34513:

RM

[EINVAL]	The attributes object referenced by *attr* has the robust mutex attribute set without the process-shared attribute being set.

RATIONALE

Add the following text:

Implementations supporting the Robust Mutexes option are required to provide robust mutexes for mutexes with the process-shared attribute set to PTHREAD_PROCESS_SHARED. Implementations are allowed, but not required, to provide robust mutexes when the process-shared attribute is set to PTHREAD_PROCESS_PRIVATE.

NAME

pthread_mutex_lock, pthread_mutex_trylock, pthread_mutex_unlock — lock and unlock a mutex

DESCRIPTION

Add the following text after Line 34786:

RM If *mutex* is a robust mutex and the process containing the owning thread terminated while holding the mutex lock, a call to *pthread_mutex_lock*( ) shall return the error value [EOWNERDEAD]. If *mutex* is a robust mutex and the owning thread terminated while holding the mutex lock, a call to *pthread_mutex_lock*( ) may return the error value [EOWNERDEAD] even if the process in which the owning thread resides has not terminated. In these cases, the mutex is locked by the thread but the state it protects is marked as inconsistent. The application should ensure that the state is made consistent for reuse and when that is complete call *pthread_mutex_consistent*( ). If the application is unable to recover the state, it should unlock the mutex without a prior call to *pthread_mutex_consistent*( ), after which the mutex is marked permanently unusable.

ERRORS

Add the following text after Line 34796:

RM [EOWNERDEAD] The mutex is a robust mutex and the process containing the previous owning thread terminated while holding the mutex lock. The mutex lock has been acquired and it is up to the new owner to make the state consistent.

RM [ENOTRECOVERABLE] The state protected by the mutex is not recoverable. The mutex is not locked.

Add the following text before Line 34797:

The *pthread_mutex_unlock*( ) function shall fail if:

RM [EPERM] The current thread does not own the mutex and the mutex is a robust mutex.

Add the following text after Line 34803:

The *pthread_mutex_lock*( ) and *pthread_mutex_trylock*( ) functions may fail if:

RM [EOWNERDEAD] The mutex is a robust mutex and the previous owning thread terminated while holding the mutex lock. The mutex lock has been acquired and it is up to the new owner to make the state consistent.

APPLICATION USAGE

Add the following text:

Applications that have assumed that non-zero return values are errors will need updating for use with robust mutexes, since a valid return for a thread acquiring a mutex which is protecting a currently inconsistent state is [EOWNERDEAD]. Applications that do not check the error returns, due to ruling out the possibility of such errors arising, should not use robust mutexes. If an application is supposed to work with normal and robust mutexes it should check all return values for error conditions and if necessary take appropriate action.

NAME

pthread_mutex_timedlock — lock a mutex

DESCRIPTION

Add the following text after Line 34891:

RM If *mutex* is a robust mutex and the process containing the owning thread terminated while holding the mutex lock, a call to *pthread_mutex_timedlock*( ) shall return the error value [EOWNERDEAD]. If *mutex* is a robust mutex and the owning thread terminated while holding the mutex lock, a call to *pthread_mutex_timedlock*( ) may return the error value [EOWNERDEAD] even if the process in which the owning thread resides has not terminated. In these cases, the mutex is locked by the thread but the state it protects is marked as inconsistent. The application should ensure that the state is made consistent for reuse and when that is complete call *pthread_mutex_consistent*( ). If the application is unable to recover the state, it should unlock the mutex without a prior call to *pthread_mutex_consistent*( ), after which the mutex is marked permanently unusable.

ERRORS

Add the following after Line 34903:

RM [EOWNERDEAD] The mutex is a robust mutex and the process containing the previous owning thread terminated while holding the mutex lock. The mutex lock has been acquired and it is up to the new owner to make the state consistent.

RM [ENOTRECOVERABLE] The state protected by the mutex is not recoverable. The mutex is not locked.

Add the following after Line 34909:

RM [EOWNERDEAD] The mutex is a robust mutex and the previous owning thread terminated while holding the mutex lock. The mutex lock has been acquired and it is up to the new owner to make the state consistent.

APPLICATION USAGE

Add the following text:

Applications that have assumed that non-zero return values are errors will need updating for use with robust mutexes, since a valid return for a thread acquiring a mutex which is protecting a currently inconsistent state is [EOWNERDEAD]. Applications that do not check the error returns, due to ruling out the possibility of such errors arising, should not use robust mutexes. If an application is supposed to work with normal and robust mutexes, it should check all return values for error conditions and if necessary take appropriate action.

NAME

pthread_mutexattr_getrobust, pthread_mutexattr_setrobust — get and set the mutex robust attribute

SYNOPSIS

RM
```
#include <pthread.h>

int pthread_mutexattr_getrobust(const pthread_mutexattr_t *restrict
    attr, int *restrict robust);
int pthread_mutexattr_setrobust(pthread_mutexattr_t *attr,
    int robust);
```

DESCRIPTION

The *pthread_mutexattr_getrobust*( ) and *pthread_mutexattr_setrobust*( ) functions, respectively, shall get and set the mutex *robust* attribute. This attribute is set in the *robust* parameter. Valid values for *robust* include:

PTHREAD_MUTEX_STALLED
: No special actions are taken if the owner of the mutex is terminated while holding the mutex lock. This can lead to deadlocks if no other thread can unlock the mutex.
This is the default value.

PTHREAD_MUTEX_ROBUST
: If the process containing the owning thread of a robust mutex terminates while holding the mutex lock, the next thread that acquires the mutex shall be notified about the termination by the return value [EOWNERDEAD] from the locking function. If the owning thread of a robust mutex terminates while holding the mutex lock, the next thread that acquires the mutex may be notified about the termination by the return value [EOWNERDEAD]. The notified thread can then attempt to mark the state protected by the mutex as consistent again by a call to *pthread_mutex_consistent*( ). After a subsequent successful call to *pthread_mutex_unlock*( ), the mutex lock shall be released and can be used normally by other threads. If the mutex is unlocked without a call to *pthread_mutex_consistent*( ), it shall be in a permanently unusable state and all attempts to lock the mutex shall fail with the error [ENOTRECOVERABLE]. The only permissible operation on such a mutex is *pthread_mutex_destroy*( ).

RETURN VALUE

Upon successful completion, the *pthread_mutexattr_getrobust*( ) function shall return zero and store the value of the *robust* attribute of *attr* into the object referenced by the *robust* parameter. Otherwise, an error value shall be returned to indicate the error. If successful, the *pthread_mutexattr_setrobust*( ) function shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_mutexattr_setrobust*( ) function shall fail if:

[EINVAL] The value of *robust* is invalid.

The *pthread_mutexattr_getrobust*( ) and *pthread_mutexattr_setrobust*( ) functions may fail if:

[EINVAL] The value specified by *attr* is invalid.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

The actions required to make the state protected by the mutex consistent again are solely dependent on the application. If it is not possible to make the state of a mutex consistent, robust mutexes can be used to notify this situation by calling *pthread_mutex_unlock*( ) without a prior call to *pthread_mutex_consistent*( ).

If the state is declared inconsistent by calling *pthread_mutex_unlock*( ) without a prior call to *pthread_mutex_consistent*( ), a possible approach could be to destroy the mutex and then reinitialize it. However, it should be noted that this is possible only in certain situations where the state protected by the mutex has to be reinitialized and coordination achieved with other threads blocked on the mutex, because otherwise a call to a locking function with a reference to a mutex object invalidated by a call to *pthread_mutex_destroy*( ) results in undefined behavior.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_mutex_consistent( ), *pthread_mutex_destroy*( ), *pthread_mutex_unlock*( ), **<errno.h>**, the Base Definitions volume of IEEE Std 1003.1-2001, **<pthread.h>**

CHANGE HISTORY

First released in Issue X.

Index