

POSIX thread API concepts

Pthread APIs

- [Complete Pthread API list](#)
- [Thread management APIs](#)
- [Thread specific storage APIs](#)
- [Thread cancellation APIs](#)
- [Mutex synchronization API](#)
- [Condition variable synchronization APIs](#)
- [Read/write lock synchronization APIs](#)
- [Signals APIs](#)

Before you get started with Pthreads

Many details in [Multithreaded applications](#) will affect your interpretation of how the Pthread APIs work.

Multithreaded applications also contains important general information about threads. The information includes how process architecture and process behavior change when running a threaded program, what parts of the system are not available for use when running a threaded program, and tips on performance and debugging of threaded jobs.

Programming with Pthreads

- Pthread concepts and references
 - [What are Pthreads?](#)
 - [Primitive data types](#) -- Naming conventions for primitive data types in threaded programs.
 - [Feature test macros](#) -- Descriptions of supported and unsupported feature test macros.
 - [OS/400 Pthreads versus other threads implementations](#)
 - [Using header files for Pthread functions](#)
 - [Pthread glossary](#) -- Definitions of some common Pthread terms.
 - [Other sources of Pthread information](#)
- Pthread programming basic tasks -- Information to get you started with Pthreads programming.
 - [Writing and compiling threaded programs](#)
 - [Running threaded programs](#)
- [Troubleshooting Pthread errors](#) -- Descriptions of common errors users encounter when programming with Pthreads.

Complete Pthread API list

For information about the examples included with the APIs, see the [information on the API examples](#).

Complete Pthread API list by name

- [pthread_atfork\(\)--Register Fork Handlers](#)
- [pthread_atfork_np\(\)--Register Fork Handlers with Extended Options](#)
- [pthread_attr_destroy\(\)--Destroy Thread Attributes Object](#)
- [pthread_attr_getdetachstate\(\)--Get Thread Attributes Object Detachstate](#)
- [pthread_attr_getinheritsched\(\)--Get Thread Attribute Object Inherit Scheduling Attributes](#)
- [pthread_attr_getschedparam\(\)--Get Thread Attributes Object Scheduling Parameters](#)
- [pthread_attr_getschedpolicy\(\)--Get Scheduling Policy](#)
- [pthread_attr_getscope\(\)--Get Scheduling Scope](#)
- [pthread_attr_getstackaddr\(\)--Get Stack Address](#)
- [pthread_attr_getstacksize\(\)--Get Stack Size](#)
- [pthread_attr_init\(\)--Initialize Thread Attributes Object](#)
- [pthread_attr_setdetachstate\(\)--Set Thread Attributes Object Detachstate](#)
- [pthread_attr_setinheritsched\(\)--Set Thread Attribute Inherit Scheduling Attributes](#)
- [pthread_attr_setschedparam\(\)--Set Thread Attributes Object Scheduling Parameters](#)
- [pthread_attr_setschedpolicy\(\)--Set Scheduling Policy](#)
- [pthread_attr_setscope\(\)--Set Scheduling Scope](#)
- [pthread_attr_setstackaddr\(\)--Set Stack Address](#)
- [pthread_attr_setstacksize\(\)--Set Stack Size](#)
- [pthread_cancel\(\)--Cancel Thread](#)
- [pthread_cleanup_peek_np\(\)--Copy Cleanup Handler from Cancellation Cleanup Stack](#)
- [pthread_cleanup_pop\(\)--Pop Cleanup Handler off of Cancellation Cleanup Stack](#)
- [pthread_cleanup_push\(\)--Push Cleanup Handler onto Cancellation Cleanup Stack](#)
- [pthread_clear_exit_np\(\)--Clear Exit Status of Thread](#)
- [pthread_cond_broadcast\(\)--Broadcast Condition to All Waiting Threads](#)
- [pthread_cond_destroy\(\)--Destroy Condition Variable](#)
- [pthread_cond_init\(\)--Initialize Condition Variable](#)
- [pthread_cond_signal\(\)--Signal Condition to One Waiting Thread](#)
- [pthread_cond_timedwait\(\)--Timed Wait for Condition](#)
- [pthread_cond_wait\(\)--Wait for Condition](#)
- [pthread_condattr_destroy\(\)--Destroy Condition Variable Attributes Object](#)
- [pthread_condattr_init\(\)--Initialize Condition Variable Attributes Object](#)
- [pthread_condattr_getpshared\(\)--Get Process Shared Attribute from Condition Attributes Object](#)
- [pthread_condattr_setpshared\(\)--Set Process Shared Attribute in Condition Attributes Object](#)

- [pthread_create\(\)--Create Thread](#)
- [pthread_delay_np\(\)--Delay Thread for Requested Interval](#)
- [pthread_detach\(\)--Detach Thread](#)
- [pthread_equal\(\)--Compare Two Threads](#)
- [pthread_exit\(\)--Terminate Calling Thread](#)
- [pthread_extendedjoin_np\(\)--Wait for Thread with Extended Options](#)
- [pthread_get_expiration_np\(\)--Get Condition Expiration Time from Relative Time](#)
- [pthread_getcancelstate_np\(\)--Get Cancel State](#)
- [pthread_getconcurrency\(\)--Get Process Concurrency Level](#)
- [pthread_getptthreadoption_np\(\)--Get Pthread Run-Time Option Data](#)
- [pthread_getschedparam\(\)--Get Thread Scheduling Parameters](#)
- [pthread_getspecific\(\)--Get Thread Local Storage Value by Key](#)
- [pthread_getthreadid_np\(\)--Retrieve Unique ID for Calling Thread](#)
- [pthread_getunique_np\(\)--Retrieve Unique ID for Target Thread](#)
- [pthread_is_initialthread_np\(\)--Check if Running in the Initial Thread](#)
- [pthread_is_multithreaded_np\(\)--Check Current Number of Threads](#)
- [pthread_join\(\)--Wait for and Detach Thread](#)
- [pthread_join_np\(\)--Wait for Thread to End](#)
- [pthread_key_create\(\)--Create Thread Local Storage Key](#)
- [pthread_key_delete\(\)--Delete Thread Local Storage Key](#)
- [pthread_kill\(\)--Send Signal to Thread](#)
- [pthread_lock_global_np\(\)--Lock Global Mutex](#)
- [pthread_mutex_destroy\(\)--Destroy Mutex](#)
- [pthread_mutex_getprioceiling\(\)--Get Mutex Priority Ceiling](#)
- [pthread_mutex_init\(\)--Initialize Mutex](#)
- [pthread_mutex_lock\(\)--Lock Mutex](#)
- [pthread_mutex_setprioceiling\(\)--Set Mutex Priority Ceiling](#)
- [pthread_mutex_timedlock_np\(\)--Lock Mutex with Time-Out](#)
- [pthread_mutex_trylock\(\)--Lock Mutex with No Wait](#)
- [pthread_mutex_unlock\(\)--Unlock Mutex](#)
- [pthread_mutexattr_destroy\(\)--Destroy Mutex Attributes Object](#)
- [pthread_mutexattr_getkind_np\(\)--Get Mutex Kind Attribute](#)
- [pthread_mutexattr_getname_np\(\)--Get Name from Mutex Attributes Object](#)
- [pthread_mutexattr_getprioceiling\(\)--Get Mutex Priority Ceiling Attribute](#)
- [pthread_mutexattr_getprotocol\(\)--Get Mutex Protocol Attribute](#)
- [pthread_mutexattr_getpshared\(\)--Get Process Shared Attribute from Mutex Attributes Object](#)
- [pthread_mutexattr_gettype\(\)--Get Mutex Type Attribute](#)
- [pthread_mutexattr_init\(\)--Initialize Mutex Attributes Object](#)

- [pthread_mutexattr_setkind_np\(\)--Get Mutex Kind Attribute](#)
- [pthread_mutexattr_setname_np\(\)--Set Name in Mutex Attributes Object](#)
- [pthread_mutexattr_setprioceiling\(\)--Set Mutex Priority Ceiling Attribute](#)
- [pthread_mutexattr_setprotocol\(\)--Set Mutex Protocol Attribute](#)
- [pthread_mutexattr_setpshared\(\)--Set Process Shared Attribute in Mutex Attributes Object](#)
- [pthread_mutexattr_settype\(\)--Set Mutex Type Attribute](#)
- [pthread_once\(\)--Perform One-Time Initialization](#)
- [pthread_rwlock_destroy\(\)--Destroy Read/Write Lock](#)
- [pthread_rwlock_init\(\)--Initialize Read/Write Lock](#)
- [pthread_rwlock_rdlock\(\)--Get Shared Read Lock](#)
- [pthread_rwlock_timedrdlock_np\(\)--Get Shared Read Lock with Time-Out](#)
- [pthread_rwlock_timedwrlock_np\(\)--Get Exclusive Write Lock with Time-Out](#)
- [pthread_rwlock_tryrdlock\(\)--Get Shared Read Lock with No Wait](#)
- [pthread_rwlock_trywrlock\(\)--Get Exclusive Write Lock with No Wait](#)
- [pthread_rwlock_unlock\(\)--Unlock Exclusive Write or Shared Read Lock](#)
- [pthread_rwlock_wrlock\(\)--Get Exclusive Write Lock](#)
- [pthread_rwlockattr_destroy\(\)--Destroy Read/Write Lock Attribute](#)
- [pthread_rwlockattr_getpshared\(\)--Get Pshared Read/Write Lock Attribute](#)
- [pthread_rwlockattr_init\(\)--Initialize Read/Write Lock Attribute](#)
- [pthread_rwlockattr_setpshared\(\)--Set Pshared Read/Write Lock Attribute](#)
- [pthread_self\(\)--Get Pthread Handle](#)
- [pthread_set_mutexattr_default_np\(\)--Set Default Mutex Attributes Object Kind Attribute](#)
- [pthread_setcancelstate\(\)--Set Cancel State](#)
- [pthread_setcanceltype\(\)--Set Cancel Type](#)
- [pthread_setconcurrency\(\)--Set Process Concurrency Level](#)
- [pthread_setpthreadoption_np\(\)--Set Pthread Run-Time Option Data](#)
- [pthread_setschedparam\(\)--Set Target Thread Scheduling Parameters](#)
- [pthread_setspecific\(\)--Set Thread Local Storage by Key](#)
- [pthread_sigmask\(\)--Set or Get Signal Mask](#)
- [pthread_signal_to_cancel_np\(\)--Convert Signals to Cancel Requests](#)
- [pthread_test_exit_np\(\)--Test Thread Exit Status](#)
- [pthread_testcancel\(\)--Create Cancellation Point](#)
- [pthread_trace_init_np\(\)--Initialize or Reinitialize Pthread Tracing](#)
- [PTHREAD_TRACE_NP\(\)--Macro to optionally execute code based on trace level](#)
- [pthread_unlock_global_np\(\)--Unlock Global Mutex](#)
- [sched_yield\(\)--Yield Processor to Another Thread](#)

Thread management APIs

Thread management APIs allow a program to manipulate threads. The APIs actually create, destroy and otherwise manage the active or ended threads within the application. The APIs allow the manipulation of some of the thread attributes of an active thread.

A program can also setup or change the characteristics of a thread attributes object. The thread attributes object is used at thread creation time. The new thread is created with the attributes that are specified in the attributes object. After the thread has been created, the attributes object is no longer required.

The table below lists important thread attributes, their default values, and all supported values.

Attribute	Default value	Supported values
<i>detachstate</i>	PTHREAD_CREATE_JOINABLE	PTHREAD_CREATE_JOINABLE PTHREAD_CREATE_DETACHED
<i>schedparam</i>	SCHED_OTHER with priority equal to PRIORITY_DEFAULT (0)	SCHED_OTHER with priority <= PTHREAD_PRIO_MAX and priority >= PTHREAD_PRIO_MIN
<i>contentionscope</i>	PTHREAD_SCOPE_SYSTEM	PTHREAD_SCOPE_SYSTEM
<i>inheritsched</i>	PTHREAD_EXPLICIT_SCHED , priority equal PRIORITY_DEFAULT (0)	PTHREAD_EXPLICIT_SCHED or PTHREAD_INHERIT_SCHED
<i>schedpolicy</i>	SCHED_OTHER	SCHED_OTHER

For information about the examples included with the APIs, see the [information on the API examples](#).

The thread management APIs are:

- [pthread_attr_destroy\(\)](#)--Destroy Thread Attributes Object
- [pthread_attr_getdetachstate\(\)](#)--Get Thread Attributes Object Detachstate
- [pthread_attr_getinheritsched\(\)](#)--Get Thread Attribute Object Inherit Scheduling Attributes
- [pthread_attr_getschedparam\(\)](#)--Get Thread Attributes Object Scheduling Parameters
- [pthread_attr_init\(\)](#)--Initialize Thread Attributes Object
- [pthread_attr_setdetachstate\(\)](#)--Set Thread Attributes Object Detachstate
- [pthread_attr_setinheritsched\(\)](#)--Set Thread Attribute Inherit Scheduling Attributes
- [pthread_attr_setschedparam\(\)](#)--Set Thread Attributes Object Scheduling Parameters
- [pthread_clear_exit_np\(\)](#)--Clear Exit Status of Thread
- [pthread_create\(\)](#)--Create Thread
- [pthread_delay_np\(\)](#)--Delay Thread for Requested Interval
- [pthread_detach\(\)](#)--Detach Thread
- [pthread_equal\(\)](#)--Compare Two Threads
- [pthread_exit\(\)](#)--Terminate Calling Thread
- [pthread_extendedjoin_np\(\)](#)--Wait for Thread with Extended Options
- [pthread_getconcurrency\(\)](#)--Get Process Concurrency Level
- [pthread_getpthreadsoption_np\(\)](#)--Get Pthread Run-Time Option Data
- [pthread_getschedparam\(\)](#)--Get Thread Scheduling Parameters
- [pthread_getthreadid_np\(\)](#)--Retrieve Unique ID for Calling Thread
- [pthread_getunique_np\(\)](#)--Retrieve a Unique ID for Target Thread

- [pthread_is_initialthread_np\(\)](#)--Check if Running in the Initial Thread
- [pthread_is_multithreaded_np\(\)](#)--Check the Current Number of Threads
- [pthread_join\(\)](#)--Wait for and Detach Thread
- [pthread_join_np\(\)](#)--Wait for Thread to End
- [pthread_once\(\)](#)--Perform One-Time Initialization
- [pthread_self\(\)](#)--Get Pthread Handle
- [pthread_setconcurrency\(\)](#)--Set Process Concurrency Level
- [pthread_setpthreadsoption_np\(\)](#)--Set Pthread Run-Time Option Data
- [pthread_setschedparam\(\)](#)--Set Target Thread Scheduling Parameters
- [pthread_trace_init_np\(\)](#)--Initialize or Reinitialize Pthread Tracing
- [PTHREAD_TRACE_NP\(\)](#)--Execute Code Based on Trace Level (Macro)
- [sched_yield\(\)](#)--Yield Processor to Another Thread

pthread_attr_destroy()--Destroy Thread Attributes Object

Syntax

```
#include <pthread.h>
int pthread_attr_destroy(pthread_attr_t *attr);
Threadsafe: Yes
Signal Safe: Yes
```

The **pthread_attr_destroy()** function destroys a thread attributes object and allows the system to reclaim any resources associated with that thread attributes object. This does not have an effect on any threads created using this thread attributes object.

Parameters

attr

(Input) The address of the thread attributes object to be destroyed

Authorities and Locks

None.

Return Value

0

pthread_attr_destroy() was successful.

value

pthread_attr_destroy() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_attr_destroy()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

Invalid Argument Specified

Related Information

- The **<pthread.h>** header file. See [Header files for Pthread functions.](#)
- [pthread_attr_init\(\)--Initialize Thread Attributes Object](#)

Example

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include "check.h"

void *threadfunc(void *parm)
{
```

pthread_attr_destroy()--Destroy Thread Attributes Object

```
    printf("Thread created using an default attributes\n");
    return NULL;
}

int main(int argc, char **argv)
{
    pthread_t          thread;
    int                rc=0;
    pthread_attr_t      pta;

    printf("Enter Testcase - %s\n", argv[0]);

    printf("Create a thread attributes object\n");
    rc = pthread_attr_init(&pta);
    checkResults("pthread_attr_init()\n", rc);

    printf("Create a thread using the attributes object\n");
    rc = pthread_create(&thread, &pta, threadfunc, NULL);
    checkResults("pthread_create()\n", rc);

    printf("Create a thread using the default attributes\n");
    rc = pthread_create(&thread, NULL, threadfunc, NULL);
    checkResults("pthread_create()\n", rc);

    printf("Destroy thread attributes object\n");
    rc = pthread_attr_destroy(&pta);
    checkResults("pthread_attr_destroy()\n", rc);

    /* sleep() is not a very robust way to wait for the thread */
    sleep(5);

    printf("Main completed\n");
    return 0;
}
```

Output:

```
Enter Testcase - QP0WTEST/TAINI0
Create a thread attributes object
Create a thread using the attributes object
Create a thread using the default attributes
Destroy thread attributes object
Thread created using an default attributes
Thread created using an default attributes
Main completed
```


pthread_attr_getdetachstate()--Get Thread Attributes Object Detachstate

Syntax

```
#include <pthread.h>
int pthread_attr_getdetachstate(const pthread_attr_t *attr,
                               int *detachstate);
```

Threadsafe: Yes

Signal Safe: Yes

The **pthread_attr_getdetachstate()** function returns the detach state attribute from the thread attributes object specified. The detach state of a thread indicates whether the system is allowed to free thread resources when a thread terminates.

The detach state specifies one of **PTHREAD_CREATE_DETACHED** or **PTHREAD_CREATE_JOINABLE**. The default detach state (**DEFAULT_DETACHSTATE**) is **PTHREAD_CREATE_JOINABLE**.

Parameters

attr

(Input) The address of the thread attributes object

detachstate

(Output) The address of the variable to contain the returned detach state

Authorities and Locks

None.

Return Value

0

pthread_attr_getdetachstate() was successful.

value

pthread_attr_getdetachstate() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_attr_getdetachstate()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

Related Information

- The **<pthread.h>** header file. See [Header files for Pthread functions](#).
- [pthread_attr_setdetachstate\(\)--Set Thread Attributes Object Detachstate](#)
- [pthread_detach\(\)--Detach Thread](#)
- [pthread_join\(\)--Wait for and Detach Thread](#)

Example

```

#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include "check.h"

int main(int argc, char **argv)
{
    pthread_t          thread;
    int                rc=0;
    pthread_attr_t      pta;
    int                state;

    printf("Enter Testcase - %s\n", argv[0]);

    printf("Create a thread attributes object\n");
    rc = pthread_attr_init(&pta);
    checkResults("pthread_attr_init()\n", rc);

    printf("Get detach state\n");
    rc = pthread_attr_getdetachstate(&pta, &state);
    checkResults("pthread_attr_getdetachstate()\n", rc);

    printf("The thread attributes object indicates: ");
    switch (state) {
    case PTHREAD_CREATE_DETACHED:
        printf("DETACHED\n");
        break;
    case PTHREAD_CREATE_JOINABLE:
        printf("JOINABLE\n");
        break;
    }

    printf("Destroy thread attributes object\n");
    rc = pthread_attr_destroy(&pta);
    checkResults("pthread_attr_destroy()\n", rc);

    printf("Main completed\n");
    return 0;
}

```

Output:

```

Enter Testcase - QP0WTEST/TAGDS0
Create a thread attributes object
Get detach state
The thread attributes object indicates: JOINABLE
Destroy thread attributes object
Main completed

```

pthread_attr_getinheritsched()--Get Thread Attribute Object Inherit Scheduling Attributes

Syntax

```
#include <pthread.h>
#include <sched.h>
int pthread_attr_getinheritsched(pthread_attr_t *attr,
                                int *inheritsched);
```

Threadsafe: Yes

Signal Safe: Yes

The **pthread_attr_getinheritsched()** function returns the inheritsched attribute from the thread attributes object specified. The inheritsched attribute is one of **PTHREAD_EXPLICIT_SCHED** or **PTHREAD_INHERIT_SCHED**. The default inheritsched attribute is **PTHREAD_EXPLICIT_SCHED**, with a default priority of zero.

Use the *inheritsched* parameter to inherit or explicitly specify the scheduling attributes when creating new threads.

Parameters

attr

(Input) Address of thread creation attributes

inheritsched

(Output) Address of the variable to receive the inheritsched attribute

Authorities and Locks

None.

Return Value

0

pthread_attr_getinheritsched() was successful.

value

pthread_attr_getinheritsched() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_attr_getinheritsched()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

Related Information

- The **<pthread.h>** header file. See [Header files for Pthread functions](#).
- [pthread_attr_setinheritsched\(\)--Set Thread Attribute Inherit Scheduling Attributes](#)

Example

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include <except.h>
#include "check.h"

void showInheritSched(pthread_attr_t *attr) {
    int rc;
    int inheritsched;
    rc = pthread_attr_getinheritsched(attr, &inheritsched);
    checkResults("pthread_attr_getinheritsched()\n", rc);

    switch(inheritsched) {
    case PTHREAD_EXPLICIT_SCHED:
        printf("Inherit Sched - PTHREAD_EXPLICIT_SCHED\n");
        break;
    case PTHREAD_INHERIT_SCHED:
        printf("Inherit Sched - PTHREAD_INHERIT_SCHED\n");
        break;
    default:
        printf("Invalid inheritsched attribute!\n");
        exit(1);
    }
    return;
}

int main(int argc, char **argv)
{
    pthread_t          thread;
    int                rc=0;
    pthread_attr_t      attr;
    char               c;
    void               *status;

    printf("Enter Testcase - %s\n", argv[0]);

    rc = pthread_attr_init(&attr);
    checkResults("pthread_attr_init()\n", rc);

    showInheritSched(&attr);

    rc = pthread_attr_setinheritsched(&attr, PTHREAD_INHERIT_SCHED);
    checkResults("pthread_attr_setinheritsched()\n", rc);

    showInheritSched(&attr);

    rc = pthread_attr_destroy(&attr);
    checkResults("pthread_attr_destroy()\n", rc);

    printf("Main completed\n");
    return 0;
}
```

Output:

```
Enter Testcase - QP0WTEST/TPGIS0
Inherit Sched - PTHREAD_EXPLICIT_SCHED
Inherit Sched - PTHREAD_INHERIT_SCHED
```

pthread_attr_getinheritsched()--Get Thread Attribute Object Inherit Scheduling Attributes

Main completed

pthread_attr_getschedparam()--Get Thread Attributes Object Scheduling Parameters

Syntax

```
#include <pthread.h>
#include <sched.h>
int pthread_attr_getschedparam(const pthread_attr_t *attr,
                               struct sched_param *param);
```

Threadsafe: Yes

Signal Safe: Yes

The **pthread_attr_getschedparam()** function returns the scheduling parameters attribute from the thread attributes object. The default OS/400 scheduling policy is **SCHED_OTHER** and cannot be changed to another scheduling policy.

The *sched_policy* field of the *param* parameter is always returned as **SCHED_OTHER**. The *sched_priority* field of the *param* structure is set to the priority of the target thread at the time of the call.

*Do not use **pthread_setschedparam()** to set the priority of a thread if you also use another mechanism (outside of the pthread APIs) to set the priority of a thread. If you do, **pthread_getschedparam()** returns only that information that was set via the pthread interfaces. (**pthread_setschedparam()** or modification of the thread attribute using **pthread_attr_setschedparam()**).*

Parameters

attr

(Input) The address of the thread attributes object

param

(Output) The address of the variable to contain the returned scheduling parameters

Authorities and Locks

None.

Return Value

0

pthread_attr_getschedparam() was successful.

value

pthread_attr_getschedparam() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_attr_getschedparam()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

Related Information

- The <pthread.h> header file. See [Header files for Pthread functions](#).
- The <sched.h> header file. See [Header files for Pthread functions](#).
- [pthread_attr_setschedparam\(\)--Set Thread Attributes Object Scheduling Parameters](#).

Example

```
#define _MULTI_THREADED
#include <pthread.h>
#include <sched.h>
#include <stdio.h>
#include "check.h"

int main(int argc, char **argv)
{
    pthread_t          thread;
    int                rc=0;
    pthread_attr_t      pta;
    struct sched_param  param;

    printf("Enter Testcase - %s\n", argv[0]);

    printf("Create a thread attributes object\n");
    rc = pthread_attr_init(&pta);
    checkResults("pthread_attr_init()\n", rc);

    printf("Get scheduling parameters\n");
    rc = pthread_attr_getschedparam(&pta, &param);
    checkResults("pthread_attr_getschedparam()\n", rc);

    printf("The thread attributes object indicates: ");
    printf("priority %d\n", param.sched_priority);

    printf("Destroy thread attributes object\n");
    rc = pthread_attr_destroy(&pta);
    checkResults("pthread_attr_destroy()\n", rc);

    printf("Main completed\n");
    return 0;
}
```

Output:

```
Enter Testcase - QP0WTEST/TAGSP0
Create a thread attributes object
Get scheduling parameters
The thread attributes object indicates: priority 0
Destroy thread attributes object
Main completed
```

pthread_attr_init()--Initialize Thread Attributes Object

Syntax

```
#include <pthread.h>
int pthread_attr_init(pthread_attr_t *attr);
Threadsafe: Yes
Signal Safe: Yes
```

The **pthread_attr_init()** function initializes a thread attributes object to the default thread attributes. The thread attributes object can be used in a call to **pthread_create()** to specify attributes of the new thread.

Parameters

attr

(Input/Output) The address of the thread attributes object to be initialized

Authorities and Locks

None.

Return Value

0

pthread_attr_init() was successful.

value

pthread_attr_init() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_attr_init()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

Related Information

- The **<pthread.h>** header file. See [Header files for Pthread functions](#).
- [pthread_attr_destroy\(\)--Destroy Thread Attributes Object](#)
- [pthread_create\(\)--Create Thread](#)

Example

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include "check.h"

void *threadfunc(void *parm)
```


pthread_attr_init()--Initialize Thread Attributes Object

```
{
    printf("Thread created using an default attributes\n");
    return NULL;
}

int main(int argc, char **argv)
{
    pthread_t          thread;
    int                rc=0;
    pthread_attr_t      pta;

    printf("Enter Testcase - %s\n", argv[0]);

    printf("Create a thread attributes object\n");
    rc = pthread_attr_init(&pta);
    checkResults("pthread_attr_init()\n", rc);

    printf("Create a thread using the attributes object\n");
    rc = pthread_create(&thread, &pta, threadfunc, NULL);
    checkResults("pthread_create()\n", rc);

    printf("Create a thread using the default attributes\n");
    rc = pthread_create(&thread, NULL, threadfunc, NULL);
    checkResults("pthread_create()\n", rc);

    printf("Destroy thread attributes object\n");
    rc = pthread_attr_destroy(&pta);
    checkResults("pthread_attr_destroy()\n", rc);

    /* sleep() is not a very robust way to wait for the thread */
    sleep(5);

    printf("Main completed\n");
    return 0;
}
```

Output:

```
Enter Testcase - QPOWTEST/TAINIO
Create a thread attributes object
Create a thread using the attributes object
Create a thread using the default attributes
Destroy thread attributes object
Thread created using an default attributes
Thread created using an default attributes
Main completed
```

pthread_attr_setdetachstate()--Set Thread Attributes Object Detachstate

Syntax

```
#include <pthread.h>
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
Threadsafe: Yes
Signal Safe: Yes
```

The **pthread_attr_setdetachstate()** function sets the detach state of the thread attributes object. The detach state of a thread indicates whether the system is allowed to free thread resources (including but not limited to thread exit status) when the thread terminates. Some resources (like automatic storage) are always freed when a thread ends.

The detach state specifies one of **PTHREAD_CREATE_DETACHED** or **PTHREAD_CREATE_JOINABLE**. The default detach state (**DEFAULT_DETACHSTATE**) is **PTHREAD_CREATE_JOINABLE**.

Parameters

attr

(Input) The address of the thread attributes object.

detachstate

(Output) The detach state, one of **PTHREAD_CREATE_JOINABLE** or **PTHREAD_CREATE_DETACHED**.

Authorities and Locks

None.

Return Value

0

pthread_attr_setdetachstate() was successful.

value

pthread_attr_setdetachstate() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_attr_setdetachstate()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

Related Information

- The **<pthread.h>** header file. See [Header files for Pthread functions](#).
- [pthread_attr_getdetachstate\(\)--Get Thread Attributes Object Detachstate](#)
- [pthread_detach\(\)--Detach Thread](#)
- [pthread_join\(\)--Wait for and Detach Thread](#)

Example

```

#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include "check.h"

void showDetachState(pthread_attr_t *a)
{
    int    rc=0;
    int    state=0;

    printf("Get detach state\n");
    rc = pthread_attr_getdetachstate(a, &state);
    checkResults("pthread_attr_getdetachstate()\n", rc);

    printf("The thread attributes object indicates: ");
    switch (state) {
    case PTHREAD_CREATE_DETACHED:
        printf("DETACHED\n");
        break;
    case PTHREAD_CREATE_JOINABLE:
        printf("JOINABLE\n");
        break;
    }
    return;
}

int main(int argc, char **argv)
{
    pthread_t      thread;
    int            rc=0;
    pthread_attr_t pta;

    printf("Enter Testcase - %s\n", argv[0]);

    printf("Create a default thread attributes object\n");
    rc = pthread_attr_init(&pta);

    checkResults("pthread_attr_init()\n", rc);

    showDetachState(&pta);

    printf("Set the detach state\n");
    rc = pthread_attr_setdetachstate(&pta, PTHREAD_CREATE_DETACHED);
    checkResults("pthread_attr_setdetachstate()\n", rc);

    showDetachState(&pta);

    printf("Destroy thread attributes object\n");
    rc = pthread_attr_destroy(&pta);
    checkResults("pthread_attr_destroy()\n", rc);

    printf("Main completed\n");
    return 0;
}

```

Output:

pthread_attr_setdetachstate()--Set Thread Attributes Object Detachstate

Enter Testcase - QP0WTEST/TASDS0

Create a default thread attributes object

Get detach state

The thread attributes object indicates: JOINABLE

Set the detach state

Get detach state

The thread attributes object indicates: DETACHED

Destroy thread attributes object

Main completed

pthread_attr_setinheritsched()--Set Thread Attribute Inherit Scheduling Attributes

Syntax

```
#include <pthread.h>
#include <sched.h>
int pthread_attr_setinheritsched(pthread_attr_t *attr,
                                int *inheritsched);
```

Threadsafe: Yes

Signal Safe: Yes

The `pthread_attr_setinheritsched()` function sets the *inheritsched* attribute in the thread attributes object specified. The *inheritsched* attribute should be one of **PTHREAD_EXPLICIT_SCHED** or **PTHREAD_INHERIT_SCHED**. The default *inheritsched* attribute is **PTHREAD_EXPLICIT_SCHED**, with a default priority of zero.

Use the *inheritsched* attribute to inherit or explicitly specify the scheduling attributes when creating new threads.

Parameters

attr

(Input) Address of thread creation attributes

inheritsched

(Output) Address of the variable to receive the *inheritsched* attribute

Authorities and Locks

None.

Return Value

0

`pthread_attr_setinheritsched()` was successful

value

`pthread_attr_setinheritsched()` was not successful. *value* is set to indicate the error condition

Error Conditions

If `pthread_attr_setinheritsched()` was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

Related Information

- The `<pthread.h>` header file. See [Header files for Pthread functions](#).
- [pthread_attr_getinheritsched\(\)--Get Thread Attribute Object Inherit Scheduling Attributes](#)
- [pthread_attr_getschedparam\(\)--Get Thread Attributes Object Scheduling Parameters](#)

Example

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include <except.h>
#include "check.h"

void showInheritSched(pthread_attr_t *attr) {
    int rc;
    int inheritsched;
    rc = pthread_attr_getinheritsched(attr, &inheritsched);
    checkResults("pthread_attr_getinheritsched()\n", rc);

    switch(inheritsched) {
    case PTHREAD_EXPLICIT_SCHED:
        printf("Inherit Sched - PTHREAD_EXPLICIT_SCHED\n");
        break;
    case PTHREAD_INHERIT_SCHED:
        printf("Inherit Sched - PTHREAD_INHERIT_SCHED\n");
        break;
    default:
        printf("Invalid inheritsched attribute!\n");
        exit(1);
    }
    return;
}

int main(int argc, char **argv)
{
    pthread_t      thread;
    int            rc=0;
    pthread_attr_t attr;
    char          c;
    void           *status;

    printf("Enter Testcase - %s\n", argv[0]);

    rc = pthread_attr_init(&attr);
    checkResults("pthread_attr_init()\n", rc);

    showInheritSched(&attr);

    rc = pthread_attr_setinheritsched(&attr, PTHREAD_INHERIT_SCHED);
    checkResults("pthread_attr_setinheritsched()\n", rc);

    showInheritSched(&attr);

    rc = pthread_attr_destroy(&attr);
    checkResults("pthread_attr_destroy()\n", rc);

    printf("Main completed\n");
    return 0;
}
```

Output:

```
Enter Testcase - QP0WTEST/TPSIS0
Inherit Sched - PTHREAD_EXPLICIT_SCHED
Inherit Sched - PTHREAD_INHERIT_SCHED
```

pthread_attr_setinheritsched()--Set Thread Attribute Inherit Scheduling Attributes

Main completed

pthread_attr_setschedparam()--Set Thread Attributes Object Scheduling Parameters

Syntax

```
#include <pthread.h>
#include <sched.h>
int pthread_attr_setschedparam(pthread_attr_t *attr,
                               const struct sched_param *param);
```

Threadsafe: Yes

Signal Safe: Yes

The **pthread_attr_setschedparam()** function sets the scheduling parameters in the thread attributes object. The supported OS/400 scheduling policy is **SCHED_OTHER**. Attempting to set the *sched_policy* field of the *param* parameter other than **SCHED_OTHER** causes the **EINVAL** error. The *sched_priority* field of the *param* parameter must range from **PRIORITY_MIN** to **PRIORITY_MAX** or the **ENOTSUP** error occurs.

All reserved fields in the scheduling parameters structure must be binary zero or the **EINVAL** error occurs.

*Do not use **pthread_setschedparam()** to set the priority of a thread if you also use another mechanism (outside of the pthread APIs) to set the priority of a thread. If you do, **pthread_getschedparam()** returns only that information that was set via the pthread interfaces (**pthread_setschedparam()** or modification of the thread attribute using **pthread_attr_setschedparam()**).*

Parameters

attr

(Input/Output) The address of the thread attributes object

param

(Input) Address of the variable containing the scheduling parameters

Authorities and Locks

None.

Return Value

0

pthread_attr_setschedparam() was successful.

value

pthread_attr_setschedparam() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_attr_setschedparam()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

[ENOTSUP]

The value specified for the priority argument is not supported.

Related Information

- The `<pthread.h>` header file. See [Header files for Pthread functions](#).
- The `<sched.h>` header file. See [Header files for Pthread functions](#).
- [pthread_attr_getschedparam\(\)--Get Thread Attributes Object Scheduling Parameters](#)

Example

```
#define _MULTI_THREADED
#include <pthread.h>
#include <sched.h>
#include <stdio.h>
#include "check.h"

#define BUMP_PRIO 1
static int thePriority = 0;

void showSchedParam(pthread_attr_t *a)
{
    int rc=0;
    struct sched_param param;

    printf("Get scheduling parameters\n");
    rc = pthread_attr_getschedparam(a, &param);
    checkResults("pthread_attr_getschedparam()\n", rc);

    printf("The thread attributes object indicates priority: %d\n",
           param.sched_priority);
    thePriority = param.sched_priority;
    return;
}

int main(int argc, char **argv)
{
    pthread_t thread;
    int rc=0;
    pthread_attr_t pta;
    struct sched_param param;

    printf("Enter Testcase - %s\n", argv[0]);

    printf("Create a thread attributes object\n");
    rc = pthread_attr_init(&pta);
    checkResults("pthread_attr_init()\n", rc);

    showSchedParam(&pta);

    memset(&param, 0, sizeof(param));
    if (thePriority + BUMP_PRIO <= PRIORITY_MAX_NP) {
        param.sched_priority = thePriority + BUMP_PRIO;
    }

    printf("Setting scheduling parameters\n");
    rc = pthread_attr_setschedparam(&pta, &param);
    checkResults("pthread_attr_setschedparam()\n", rc);

    showSchedParam(&pta);
}
```

pthread_attr_setschedparam()--Set Thread Attributes Object Scheduling Parameters

```
    printf("Destroy thread attributes object\n");  
    rc = pthread_attr_destroy(&pta);  
    checkResults("pthread_attr_destroy()\n", rc);  
  
    printf("Main completed\n");  
    return 0;  
}
```

Output:

```
Enter Testcase - QP0WTEST/TASSP0  
Create a thread attributes object  
Get scheduling parameters  
The thread attributes object indicates priority: 0  
Setting scheduling parameters  
Get scheduling parameters  
The thread attributes object indicates priority: 0  
Destroy thread attributes object  
Main completed
```

pthread_clear_exit_np()--Clear Exit Status of Thread

Syntax

```
#include <pthread.h>
int pthread_clear_exit_np(void);
Threadsafe: Yes
Signal Safe: Yes
```

The **pthread_clear_exit_np()** function clears the exit status of the thread. If the thread is currently exiting due to a call to **pthread_exit()** or is the target of a **pthread_cancel()**, then **pthread_clear_exit_np()** can be used in conjunction with **setjmp()**, **longjmp()**, and **pthread_setcancelstate()** to prevent a thread from terminating, and 'handle' the exit condition.

The only supported way to prevent thread exit during the condition in which **pthread_exit()** was called, or action is being taken for the target of a **pthread_cancel()** is shown in the example. It consists of using **longjmp()** from a cancellation cleanup handler back into some thread routine that is still on the invocation stack. From that routine, the functions **pthread_clear_exit_np()**, and **pthread_setcancelstate()** are used to restore the state of the thread before the condition that was causing the thread exit.

This function is not portable.

Parameters

None.

Authorities and Locks

None.

Return Value

0

pthread_clear_exit_np() was successful.

value

pthread_clear_exit_np() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_clear_exit_np()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The thread is not currently exiting

Related Information

- The **<pthread.h>** header file. See [Header files for Pthread functions](#).
- [pthread_exit\(\)--Terminate Calling Thread](#)
- [pthread_cancel\(\)--Cancel Thread](#)

Example

```

#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include <except.h>
#include <setjmp.h>
#include "check.h"

int      threadStatus=1;

void cleanupHandler(void *p)
{
    jmp_buf      *j = (jmp_buf *)p;

    /* Warning, it is quite possible that using combinations of      */
    /* setjmp(), longjmp(), pthread_clear_exit_np(), and                */
    /* pthread_setcancelstate() to handle thread exits or              */
    /* cancellation could result in looping or non-cancelable          */
    /* threads if done incorrectly.                                     */
    printf("In cancellation cleanup handler. Handling the thread exit\n");
    longjmp(*j, 1);
    printf("The exit/cancellation was not stopped!\n");
    return;
}

void *threadfunc(void *parm)
{
    jmp_buf      j;
    int          rc, old;

    printf("Inside secondary thread\n");
    if (setjmp(j)) {
        /* Returned from longjmp after stopping the thread exit      */
        /* Since longjmp was called from within the cancellation        */
        /* cleanup handler, we must clear the exit state of the        */
        /* thread and reset the cancelability state to what it was     */
        /* before the cancellation cleanup handlers were invoked       */
        /* (Cancellation cleanup handlers are invoked with             */
        /* thread cancellation disabled)                                */
        printf("Stopped the thread exit, now clean up the states\n");

        printf("Clear exit state\n");
        rc = pthread_clear_exit_np();
        checkResults("pthread_clear_exit_np()\n", rc);

        printf("Restore cancel state\n");
        rc = pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, &old);
        checkResults("pthread_setcancelstate()\n", rc);
        /* This example was successful                                */
        threadStatus = 0;
    }
    else {
        printf("Pushing cleanup handler that will stop the exit\n");
        pthread_cleanup_push(cleanupHandler, &j);
        /* This exit will be stopped by cleanupHandler2 and the      */
        /* pthread_clear_exit_np() that is done above                 */
        pthread_exit(__VOID(threadStatus));
        printf("Did not expect to get here! Left status as 1.\n");
        pthread_cleanup_pop(0);
    }
}

```

pthread_clear_exit_np()--Clear Exit Status of Thread

```
    }
    pthread_exit(__VOID(threadStatus));
}

int main(int argc, char **argv)
{
    pthread_t      thread;
    int            rc=0;
    char           c;
    void           *status;

    printf("Enter Testcase - %s\n", argv[0]);

    printf("Create thread that will demonstrate handling an exit\n");
    rc = pthread_create(&thread, NULL, threadfunc, NULL);
    checkResults("pthread_create()\n", rc);

    rc = pthread_join(thread, &status);
    checkResults("pthread_join()\n", rc);
    if (__INT(status) != 0) {
        printf("Got an unexpected return status from the thread!\n");
        exit(1);
    }
    printf("Main completed\n");
    return 0;
}
```

Output:

```
Enter Testcase - QP0WTEST/TPCEXIT0
Create thread that will demonstrate handling an exit
Inside secondary thread
Pushing cleanup handler that will stop the exit
In cancellation cleanup handler. Handling the thread exit
Stopped the thread exit, now clean up the states
Clear exit state
Restore cancel state
Main completed
```

pthread_create()--Create Thread

Syntax

```
#include <pthread.h>
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                  void *(*start_routine)(void *), void *arg);
```

Threadsafe: Yes

Signal Safe: Yes

The **pthread_create()** function creates a thread with the specified attributes and runs the C function *start_routine* in the thread with the single pointer argument specified. The new thread may, but does not always, begin running before **pthread_create()** returns. If **pthread_create()** completes successfully, the Pthread handle is stored in the contents of the location referred to by *thread*.

If the *start_routine* returns normally, it is as if there was an implicit call to **pthread_exit()** using the return value of *start_routine* as the status. The function passed as *start_routine* should correspond to the following C function prototype:

```
void *threadStartRoutineName(void *);
```

If the thread attributes object represented by *attr* is modified later, the newly created thread is not affected. If *attr* is **NULL**, the default thread attributes are used.

With the following declarations and initialization,

```
pthread_t t;
void *foo(void *);
pthread_attr_t attr;
pthread_attr_init(&pta);
```

the following two thread creation mechanisms are functionally equivalent:

```
rc = pthread_create(&t, NULL, foo, NULL);
```

```
rc = pthread_create(&t, &attr, foo, NULL);
```

The cancellation state of the new thread is **PTHREAD_CANCEL_ENABLE**. The cancellation type of the new thread is **PTHREAD_CANCEL_DEFERRED**.

The signal information maintained in the new thread is as follows:

- The signal mask is inherited from the creating thread.
- The set of signals pending for the new thread is empty.

If you attempt to create a thread in a job that is not capable of starting threads, **pthread_create()** fails with the **EBUSY** error. If you attempt to create a thread from a location in which thread creation is not allowed, **pthread_create()** fails with the **EBUSY** error. See the **pthread_getptthreadoption_np()** function, option **PTHREAD_OPTION_THREAD_CAPABLE_NP**, for details about how to determine whether thread creation is currently allowed in your process.

In the OS/400 implementation, the initial thread is special. Termination of the initial thread via **pthread_exit()** or any other thread termination mechanism terminates the entire process.

The OS/400 implementation does not set a hard limit on the number of threads that can be created. The **PTHREAD_THREADS_MAX** macro is implemented as a function call, and returns different values depending on the administrative setting of the maximum number of threads for the process. The default is NO MAX and has the numeric value of 2147483647 (0x7FFFFFFF). Realistically, the number of threads is limited by the amount of storage available to the job.

Currently, thread creation is not allowed after process termination has been started. For example, after a call to **exit()**, destructors for C++ static objects, functions registered with **atexit()** or **CEE4RAGE()** are allowed to run. If these functions attempt to create a thread, **pthread_create()** fails with the **EBUSY** error. Similar failures occur if other mechanisms are used to call **pthread_create()** after process termination has started.

Usage Notes

1. If you attempt to create a thread in a job that is not capable of starting threads or for some other reason, thread creation is not allowed, and **pthread_create()** fails with the **EBUSY** error.
2. For the best performance during thread creation, you should always use **pthread_join()** or **pthread_detach()**. This allows resources to be reclaimed or reused when the thread terminates.
3. The OS/400 implementation of threads allows the user ID to be changed on a per-thread basis. If, at the time the application creates the first thread, the application has not associated a process user identity with the job, the system uses the identity of the current user to set the process user identity for the job. The process user identity is used by some operating system support when operations that require authorization checks are done against a multithreaded job from outside that job. »The application can set the process user identity using the [Set Job User Identify](#) (QWTSJUID) or [QwtSetJuid\(\)](#) Set Job User Identity APIs. See the [Security](#) APIs for more details.«

Parameters

thread

(Output) Pthread handle to the created thread

attr

(Input) The thread attributes object containing the attributes to be associated with the newly created thread. If **NULL**, the default thread attributes are used.

start_routine

(Input) The function to be run as the new threads start routine

arg

(Input) An address for the argument for the threads start routine

Authorities and Locks

None.

Return Value

0

pthread_create() was successful.

value

pthread_create() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_create()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

[EAGAIN]

The system did not have enough resources to create another thread or the maximum number of threads for this job has been reached

[EBUSY]

The system cannot allow thread creation in this process at this time.

Related Information

- The `<pthread.h>` header file. See [Header files for Pthread functions](#).
- [pthread_exit\(\)--Terminate Calling Thread](#)
- [pthread_cancel\(\)--Cancel Thread](#)
- [pthread_detach\(\)--Detach Thread](#)
- [pthread_join\(\)--Wait for and Detach Thread](#)

Example

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include "check.h"

typedef struct {
    int    value;
    char  string[128];
} thread_parm_t;

void *threadfunc(void *parm)
{
    thread_parm_t *p = (thread_parm_t *)parm;
    printf("%s, parm = %d\n", p->string, p->value);
    free(p);
    return NULL;
}

int main(int argc, char **argv)
{
    pthread_t      thread;
    int            rc=0;
    pthread_attr_t  pta;
    thread_parm_t  *parm=NULL;

    printf("Enter Testcase - %s\n", argv[0]);

    printf("Create a thread attributes object\n");
    rc = pthread_attr_init(&pta);
    checkResults("pthread_attr_init()\n", rc);

    /* Create 2 threads using default attributes in different ways */
    printf("Create thread using the NULL attributes\n");
    /* Set up multiple parameters to pass to the thread */
    parm = malloc(sizeof(thread_parm_t));
    parm->value = 5;
    strcpy(parm->string, "Inside secondary thread");
    rc = pthread_create(&thread, NULL, threadfunc, (void *)parm);
    checkResults("pthread_create(NULL)\n", rc);

    printf("Create thread using the default attributes\n");
    /* Set up multiple parameters to pass to the thread */
    parm = malloc(sizeof(thread_parm_t));
    parm->value = 77;
    strcpy(parm->string, "Inside secondary thread");
    rc = pthread_create(&thread, &pta, threadfunc, (void *)parm);
```


pthread_create()--Create Thread

```
    checkResults("pthread_create(&pta)\n", rc);

    printf("Destroy thread attributes object\n");
    rc = pthread_attr_destroy(&pta);
    checkResults("pthread_attr_destroy()\n", rc);

    /* sleep() is not a very robust way to wait for the thread */
    sleep(5);

    printf("Main completed\n");
    return 0;
}
```

Output:

```
Enter Testcase - QP0WTEST/TPCRT0
Create a thread attributes object
Create thread using the NULL attributes
Create thread using the default attributes
Destroy thread attributes object
Inside secondary thread, parm = 77
Inside secondary thread, parm = 5
Main completed
```

pthread_delay_np()--Delay Thread for Requested Interval

Syntax

```
#include <pthread.h>
#include <sched.h>
int pthread_delay_np(const struct timespec *deltatime);
Threadsafe: Yes
Signal Safe: Yes
```

The **pthread_delay_np()** function causes the calling thread to delay for the *deltatime* specified.

Although time is specified in seconds and nanoseconds, the system has approximately millisecond granularity. Due to scheduling and priorities, the amount of time you actually wait might be slightly more or less than the amount of time specified.

During the time that the thread is blocked in **pthread_delay_np()**, any asynchronous signals that are delivered to the thread have their actions taken. After the signal action (such as running a signal handler), the wait resumes if the specified interval has not yet elapsed.

The **pthread_delay_np()** function is a cancellation point.

This function is not portable.

Parameters

interval

(Input) Address of the *timespec* structure containing the interval to wait

Authorities and Locks

None.

Return Value

0

pthread_delay_np() was successful.

value

pthread_delay_np() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_delay_np()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

Related Information

- The `<pthread.h>` header file. See [Header files for Pthread functions](#).

Example

```
#define _MULTI_THREADED
#include <stdio.h>
#include <qp0z1170.h>
#include <time.h>
#include <pthread.h>
#include "check.h"

#define NTHREADS 5

void *threadfunc(void *parm)
{
    int rc;
    struct timespec ts = {0, 0};

    /* 5 and 1/2 seconds */
    ts.tv_sec = 5;
    ts.tv_nsec = 500000000;

    printf("Thread blocked\n");
    rc = pthread_delay_np(&ts);
    if (rc != 0) {
        printf("pthread_delay_np() - return code %d\n", rc);
        return (void*)&rc;
    }
    printf("Wait timed out!\n");

    return NULL;
}

int main(int argc, char **argv)
{
    int rc=0;
    int i;
    pthread_t threadid[NTHREADS];
    void *status;
    int fail=0;

    printf("Enter Testcase - %s\n", argv[0]);

    printf("Create %d threads\n", NTHREADS);
    for(i=0; i<NTHREADS; ++i) {
        rc = pthread_create(&threadid[i], NULL, threadfunc, NULL);
        checkResults("pthread_create()\n", rc);
    }

    printf("Wait for threads and cleanup\n");
    for (i=0; i<NTHREADS; ++i) {
        rc = pthread_join(threadid[i], &status);
        checkResults("pthread_join()\n", rc);
        if (status != NULL) {
            fail = 1;
        }
    }
}
```

pthread_delay_np()--Delay Thread for Requested Interval

```
    if (fail) {  
        printf("At least one thread failed!\n");  
        exit(1);  
    }  
    printf("Main completed\n");  
    return 0;  
}
```

Output:

```
Enter Testcase - QP0WTEST/TPDLY0  
Create 5 threads  
Thread blocked  
Thread blocked  
Thread blocked  
Thread blocked  
Wait for threads and cleanup  
Thread blocked  
Wait timed out!  
Wait timed out!  
Wait timed out!  
Wait timed out!  
Wait timed out!  
Main completed
```

pthread_detach()--Detach Thread

Syntax

```
#include <pthread.h>
int pthread_detach(pthread_t thread);
Threadsafe: Yes
Signal Safe: No
```

The **pthread_detach()** function indicates that system resources for the specified *thread* should be reclaimed when the thread ends. If the thread is already ended, resources are reclaimed immediately. This routine does not cause the thread to end. After **pthread_detach()** has been issued, it is not valid to try to **pthread_join()** with the target thread.

Eventually, you should call **pthread_join()** or **pthread_detach()** for every thread that is created joinable (with a detach state of **PTHREAD_CREATE_JOINABLE**) so that the system can reclaim all resources associated with the thread. Failure to join to or detach threads that can be joined causes memory and other resource leaks until the process ends.

If *thread* does not represent a valid undetached thread, **pthread_detach()** will return ESRCH.

Parameters

thread

(Input) Pthread handle to the target thread

Authorities and Locks

None.

Return Value

0

pthread_detach() was successful.

value

pthread_detach() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_detach()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

[ESRCH]

No item could be found that matches the specified value

Related Information

- The **<pthread.h>** header file. See [Header files for Pthread functions](#).
- [pthread_exit\(\)--Terminate Calling Thread](#)
- [pthread_create\(\)--Create Thread](#)
- [pthread_join\(\)--Wait for and Detach Thread](#)

Example

```

#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include "check.h"

void *threadfunc(void *parm)
{
    printf("Inside secondary thread\n");
    return NULL;
}

int main(int argc, char **argv)
{
    pthread_t      thread;
    int            rc=0;

    printf("Enter Testcase - %s\n", argv[0]);

    printf("Create thread using attributes that allow join or detach\n");
    rc = pthread_create(&thread, NULL, threadfunc, NULL);
    checkResults("pthread_create()\n", rc);

    sleep(5);

    printf("Detach the thread after it terminates\n");
    rc = pthread_detach(thread);
    checkResults("pthread_detach()\n", rc);

    printf("Detach the thread again (expect ESRCH)\n");
    rc = pthread_detach(thread);
    if (rc != ESRCH) {
        printf("Got an unexpected result! rc=%d\n",
            rc);
        exit(1);
    }
    printf("Second detach fails correctly\n");

    /* sleep() is not a very robust way to wait for the thread */
    sleep(5);
    printf("Main completed\n");
    return 0;
}

```

Output:

```

Enter Testcase - QP0WTEST/TPDET0
Create thread using attributes that allow join or detach
Inside secondary thread
Detach the thread after it terminates
Detach the thread again (expect ESRCH)
Second detach fails correctly
Main completed

```

pthread_equal()--Compare Two Threads

Syntax

```
#include <pthread.h>
int pthread_equal(pthread_t t1, pthread_t t2);
Threadsafe: Yes
Signal Safe: Yes
```

The **pthread_equal()** function compares two Pthread handles for equality.

Parameters

t1

(Input) Pthread handle for thread 1

t2

(Input) Pthread handle for thread 2

Authorities and Locks

None.

Return Value

0

The Pthread handles do not refer to the same thread.

1

The Pthread handles refer to the same thread.

Error Conditions

None.

Related Information

- The **<pthread.h>** header file. See [Header files for Pthread functions](#).
- [pthread_self\(\)--Get Pthread Handle](#)
- [pthread_create\(\)--Create Thread](#)

Example

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include "check.h"

pthread_t    theThread;

void *threadfunc(void *parm)
{
    printf("Inside secondary thread\n");
```

pthread_equal()--Compare Two Threads

```
    theThread = pthread_self();
    return NULL;
}

int main(int argc, char **argv)
{
    pthread_t      thread;
    int            rc=0;

    printf("Enter Testcase - %s\n", argv[0]);

    printf("Create thread using default attributes\n");
    rc = pthread_create(&thread, NULL, threadfunc, NULL);
    checkResults("pthread_create()\n", rc);

    /* sleep() is not a very robust way to wait for the thread */
    sleep(5);

    printf("Check if global vs local pthread_t are equal\n");
    if (!pthread_equal(thread, theThread)) {
        printf("Unexpected results on pthread_equal()!\n");
        exit(1);
    }
    printf("pthread_equal returns true\n");

    printf("Main completed\n");
    return 0;
}
```

Output:

```
Enter Testcase - QP0WTEST/TPEQU0
Create thread using default attributes
Inside secondary thread
Check if global vs local pthread_t are equal
pthread_equal returns true
Main completed
```


pthread_exit()--Terminate Calling Thread

Syntax

```
#include <pthread.h>
void pthread_exit(void *status);
Threadsafe: Yes
Signal Safe: No
```

The **pthread_exit()** function terminates the calling thread, making its exit *status* available to any waiting threads. Normally, a thread terminates by returning from the start routine that was specified in the **pthread_create()** call which started it. An implicit call to **pthread_exit()** occurs when any thread returns from its start routine. (With the exception of the initial thread, at which time an implicit call to **exit()** occurs). The **pthread_exit()** function provides an interface similar to **exit()** but on a per-thread basis.

Note that in the OS/400 implementation of threads, the initial thread is special. Termination of the initial thread via **pthread_exit()** or any thread termination mechanism terminates the entire process.

The following activities occur in this order when a thread terminates via a return from its start routine or **pthread_exit()** or thread cancellation:

1. Any cancellation cleanup handlers that have been pushed and not popped will be executed in reverse order with cancellation disabled.
2. Data destructors are called for any thread specific data entries that have a non NULL value for both the value and the destructor.
3. The thread terminates.
4. Thread termination may possibly cause the system to run OS/400 cancel handlers (registered with the `#pragma cancel_handler` directive), or C++ destructors for automatic objects.
5. If thread termination is occurring in the initial thread, it will cause the system to terminate all other threads, then run C++ static object destructors, activation group cleanup routines and `atexit()` functions.
6. Any mutexes that are held by a thread that terminates, become 'abandoned' and are no longer valid. Subsequent calls by other threads that attempt to acquire the abandoned mutex through **pthread_mutex_lock()** will deadlock. Subsequent calls by other threads that attempt to acquire the abandoned mutex through **pthread_mutex_trylock()** will return **EBUSY**.
7. No release of any application visible process resources occur. This includes but is not limited to mutexes, file descriptors, or any process level cleanup actions.

Do not call **pthread_exit()** from a cancellation cleanup handler or destructor function that was invoked as a result of either an implicit or explicit call to **pthread_exit()**. If **pthread_exit()** is called from a cancellation cleanup handler, the new invocation of **pthread_exit()** will continue cancellation cleanup processing using the next cancellation cleanup handler that was pushed. If **pthread_exit()** is called from a data destructor, the new invocation of **pthread_exit()** will skip all subsequent calls to any data destructors (regardless of the number of destructor iterations that have completed), and terminate the thread.

Cleanup handlers and data destructors are not called when the application calls `exit()` or `abort()` or otherwise terminates the process. Cleanup handlers and data destructors are not called when a thread terminates via any proprietary OS/400 mechanism other than the Pthread interfaces.

The meaning of the *status* parameter is determined by the application except for the following conditions:

1. When the thread has been canceled using **pthread_cancel()**, the exit status of **PTHREAD_CANCELED** will be made available.
2. When the thread has been terminated as a result of an unhandled OS/400 exception, operator intervention or other proprietary OS/400 mechanism, the exit status of **PTHREAD_EXCEPTION_NP** will be made available.

No address error checking is done on the *status* parameter. Do not call **pthread_exit()** with, or return the address of, a variable in a threads automatic storage. This storage will be unavailable after the thread terminates.

*If **pthread_exit()** is called by application code after step 3 in the above list, **pthread_exit()** will fail with the **CPF1F81***

*exception. This indicates that the thread is already considered terminated by the system, and **pthread_exit()** cannot continue. If your code does not handle this exception, it will appear as if the call to **pthread_exit()** was successful.*

Parameters

status

(Input) exit status of the thread

Authorities and Locks

None.

Return Value

pthread_exit() does not return.

Error Conditions

None.

Related Information

- The <**pthread.h**> header file. See [Header files for Pthread functions](#).
- [pthread_cancel\(\)--Cancel Thread](#)
- [pthread_create\(\)--Create Thread](#)
- [pthread_join\(\)--Wait for and Detach Thread](#)

Example

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include "check.h"

int  theStatus=5;

void *threadfunc(void *parm)
{
    printf("Inside secondary thread\n");
    pthread_exit(__VOID(theStatus));
    return __VOID(theStatus); /* Not needed, but this makes the compiler smile
*/
}

int main(int argc, char **argv)
{
    pthread_t      thread;
    int            rc=0;
    void           *status;

    printf("Enter Testcase - %s\n", argv[0]);

    printf("Create thread using attributes that allow join\n");
```

pthread_exit()--Terminate Calling Thread

```
rc = pthread_create(&thread, NULL, threadfunc, NULL);
checkResults("pthread_create()\n", rc);

printf("Wait for the thread to exit\n");
rc = pthread_join(thread, &status);
checkResults("pthread_join()\n", rc);
if (__INT(status) != theStatus) {
    printf("Secondary thread failed\n");
    exit(1);
}

printf("Got secondary thread status as expected\n");
printf("Main completed\n");
return 0;
}
```

Output:

```
Enter Testcase - QP0WTEST/TPEXIT0
Create thread using attributes that allow join
Wait for the thread to exit
Inside secondary thread
Got secondary thread status as expected
Main completed
```

pthread_extendedjoin_np()--Wait for Thread with Extended Options

Syntax

```
#include <pthread.h>
int pthread_extendedjoin_np(pthread_t thread, void **status,
                           pthread_joinoption_np_t *options);
```

Threadsafe: Yes

Signal Safe: No

The **pthread_extendedjoin_np()** function waits for a thread to terminate, optionally detaches the thread, then returns the thread's exit status.

If the *options* parameter is specified as **NULL** or the contents of the *pthread_joinoption_np_t* structure represented by *options* parameter is binary 0, then the behavior of **pthread_extendedjoin_np()** is equivalent to **pthread_join()**.

The *delatime* field of the *options* parameter can be used to specify the amount of elapsed time to wait before the wait times out. If the wait times out, the **ETIMEDOUT** error is returned and the thread is not detached. For an infinite wait, specify a seconds value of 0, and a nanoseconds value of 0.

The *leaveThreadAllocated* field of the *options* parameter can be used to specify that the **pthread_extendedjoin_np()** function should NOT implicitly detach the thread when the join completes successfully. If the *leaveThreadAllocated* option is used, the thread should later be detached using **pthread_join()**, **pthread_detach()**, or **pthread_extendedjoin_np()** without specifying the *leaveThreadAllocated* option.

The reserved fields of the *options* parameter are for use by possible future extensions to **pthread_extendedjoin_np()**. If any reserved fields of the *options* parameter are not zero, the **EINVAL** error is returned.

If the *status* parameter is **NULL**, the thread's exit status is not returned.

The meaning of the thread's exit status (value returned to the *status* memory location) is determined by the application except for the following conditions:

1. When the thread has been canceled using **pthread_cancel()**, the exit status of **PTHREAD_CANCELED** is made available.
2. When the thread has been terminated as a result of an unhandled OS/400 exception, operator intervention, or other proprietary OS/400 mechanism, the exit status of **PTHREAD_EXCEPTION_NP** is made available.

Eventually, you should call **pthread_join()**, **pthread_detach()** or **pthread_extendedjoin_np()** without specifying the *leaveThreadAllocated* option for every thread that is created joinable (with a detach state of **PTHREAD_CREATE_JOINABLE**) so that the system can reclaim all resources associated with the thread. Failure to join to or detach joinable threads causes memory and other resource leaks until the process ends.

Parameters

thread

(Input) Pthread handle to the target thread

status

(Input/Output) Address of the variable to receive the thread's exit status

options

(Input) Address of the join options structure specifying optional behavior of this API.

Authorities and Locks

None.

Return Value

0

pthread_extendedjoin_np() was successful.

value

pthread_extendedjoin_np() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_extendedjoin_np()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

[ESRCH]

The thread specified could not be found.

[ETIMEDOUT]

The time specified in the *deltatime* field of the *options* parameter elapsed without the target thread terminating.

Related Information

- The <pthread.h> header file. See [Header files for Pthread functions](#).
- [pthread_detach\(\)--Detach Thread](#)
- [pthread_exit\(\)--Terminate Calling Thread](#)
- [pthread_join\(\)--Wait for and Detach Thread](#)
- [pthread_join_np\(\)--Wait for Thread to End](#)

Example

```
#define _MULTI_THREADED
#include <pthread.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
#include <stdio.h>
#include "check.h"

static void *thread(void *parm)
{
    printf("Entered thread\n");
    sleep(10);
    printf("Ending thread\n");
    return __VOID(42);
}

int main (int argc, char *argv[])
```

pthread_extendedjoin_np())--Wait for Thread with Extended Options

```
{
    pthread_joinoption_np_t    joinoption;
    void                      *status;
    int                       rc;
    pthread_t                 t;

    printf("Entering testcase %s\n", argv[0]);

    printf("Create thread using attributes that allow join\n");
    rc = pthread_create(&t, NULL, thread, NULL);
    checkResults("pthread_create()\n", rc);

    memset(&joinoption, 0, sizeof(pthread_joinoption_np_t));
    joinoption.deltatime.tv_sec = 3;
    joinoption.leaveThreadAllocated = 1;

    printf("Join to the thread, timeout in 3 seconds, no implicit detach\n");
    rc = pthread_extendedjoin_np(t, &status, &joinoption);
    if (rc != ETIMEDOUT) {
        printf("Join did not timeout as expected! rc=%d\n", rc);
        exit(1);
    }

    /* Call pthread_extendedjoin_np the same as a normal      */
    /* pthread_join() call.                                   */
    /* i.e.          Implicit Detach is done, and Infinite wait */
    printf("Normal join to the thread\n");
    rc = pthread_extendedjoin_np(t, &status, NULL);
    checkResults("pthread_extendedjoin_np(no-options)\n", rc);

    if (__INT(status) != 42) {
        printf("Got the incorrect thread status!\n");
        exit(1);
    }
    printf("Main completed\n");
    return(0);
}
```

Output

```
Entering testcase QP0WTEST/TPJOINED0
Create thread using attributes that allow join
Join to the thread, timeout in 3 seconds, no implicit detach
Entered thread
Normal join to the thread
Ending thread
Main completed
```

pthread_getconcurrency()--Get Process Concurrency Level

Syntax

```
#include <pthread.h>
int pthread_getconcurrency( );
Threadsafe: Yes
Signal Safe: No
```

The **pthread_getconcurrency()** function retrieves the current concurrency level for the process. A value of 0 indicates that the threads implementation chooses the concurrency level that best suits the application. A concurrency level greater than zero indicates that the application wishes to inform the system of its desired concurrency level.

The concurrency level is not used by the OS/400 threads implementation. Each user thread is always bound to a kernel thread.

Parameters

None.

Authorities and Locks

None.

Return Value

value

pthread_getconcurrency() returns the current concurrency level.

Error Conditions

None.

Related Information

- The **<pthread.h>** header file. See [Header files for Pthread functions](#).
- [pthread_setconcurrency\(\)--Set Process Concurrency Level](#)

pthread_getpthreadoption_np()--Get Pthread Run-Time Option Data

Syntax

#include <pthread.h>
void pthread_getpthreadoption_np(pthread_option_np_t *optionData);
Threadsafe: Yes
Signal Safe: Yes

The **pthread_getpthreadoption_np()** function gets option data from the pthread run-time for the process.

Input and output data is specified and returned uniquely based on the specified *optionData*. See the table below for details about input and output. The option field in the *optionData* parameter is always required. Other fields may be input, output, or ignored, based on the specific option used.

For all options, every reserved field in the structure represented by *optionData* must be binary zero or the **EINVAL** error is returned. Unless otherwise noted for an option, the *target* field in the *option* parameter is always ignored.

The currently supported options, the data they represent, and the valid operations are as follows:

<i>option</i> field of the <i>option</i> parameter	Description
PTHREAD_OPTION_POOL_NP	When a thread terminates and it is detached or joined to, certain data structures from the pthreads run-time are maintained in a pool for possible reuse by future threads. This improves performance for creating threads. Typically, an application should not be concerned with this storage pool. Use this option to determine what the current maximum size of the allowed storage pool is. The <i>optionValue</i> field of the <i>optionData</i> parameter is set to the current maximum number of thread structures, which is maintained in the storage pool. By default, the maximum size of the storage reuse pool contains enough room for 512 thread structures.
PTHREAD_OPTION_POOL_CURRENT_NP	When a thread terminates and it is detached or joined to, certain data structures from the pthreads run-time are maintained in a pool for possible reuse by future threads. This improves performance for creating threads. Typically, an application should not be concerned with this storage pool. Use this option to determine how many thread structures are currently in the storage pool. The <i>optionValue</i> field of the <i>optionData</i> parameter is set to the current number of thread structures, which are contained in the storage pool. By default, the storage pool contains no thread structures. When a thread terminates and is detached or joined to and the current size of the pool is less than the maximum size, the thread structure is added to the pool.
PTHREAD_OPTION_THREAD_CAPABLE_NP	Not all OS/400 jobs can start threads at all times. Use this option to determine whether thread creation is currently allowed for your process. The <i>optionValue</i> field of the <i>optionData</i> parameter is set to indicate whether thread creation is currently allowed. The field is set to 0 to indicate that thread creation is not allowed, the field will be set to 1 to indicate thread creation is allowed. If thread creation is not allowed, pthread_create() fails with the EBUSY error. See pthread_create() for more details.

Parameters

option

(Input/Output) Address of the variable containing option information and to contain output option information.

Authorities and Locks

None.

Return Value

0

pthread_getpthreadoption_np() was successful.

value

pthread_getpthreadoption_np() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_getpthreadoption_np()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

Related Information

- The <pthread.h> header file. See [Header files for Pthread functions](#).
- [pthread_setpthreadoption_np\(\)--Set Pthread Run-Time Option Data](#)

Example

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include "check.h"

void *threadfunc(void *parm)
{
    printf("Inside the thread\n");
    return NULL;
}

void showCurrentSizeOfPool(void)
{
    int                rc;
    pthread_option_np_t  opt;

    memset(&opt, 0, sizeof(opt));
    opt.option = PTHREAD_OPTION_POOL_CURRENT_NP;
    rc = pthread_getpthreadoption_np(&opt);
    checkResults("pthread_getpthreadoption_np()\n", rc);
}
```

pthread_getpthreadoption_np())--Get Pthread Run-Time Option Data

```
    printf("Current number of thread structures in pool is %d\n",
           opt.optionValue);
    return;
}

int main(int argc, char **argv)
{
    pthread_t          thread;
    int                rc=0;
    pthread_option_np_t opt;

    printf("Enter Testcase - %s\n", argv[0]);

    printf("Create thread using the NULL attributes\n");
    rc = pthread_create(&thread, NULL, threadfunc, NULL);
    checkResults("pthread_create(NULL)\n", rc);

    memset(&opt, 0, sizeof(opt));
    opt.option = PTHREAD_OPTION_POOL_NP;
    rc = pthread_getpthreadoption_np(&opt);
    checkResults("pthread_getpthreadoption_np()\n", rc);

    printf("Current maximum pool size is %d thread structures\n",
           opt.optionValue);

    showCurrentSizeOfPool();

    printf("Joining to the thread may it to the storage pool\n");
    rc = pthread_join(thread, NULL);
    checkResults("pthread_join()\n", rc);

    showCurrentSizeOfPool();
    printf("Main completed\n");
    return 0;
}
```

Output:

```
Enter Testcase - QP0WTEST/TPGETOPT
Create thread using the NULL attributes
Current maximum pool size is 512 thread structures
Current number of thread structures in pool is 0
Joining to the thread may it to the storage pool
Inside the thread
Current number of thread structures in pool is 1
Main completed
```

pthread_getschedparam()--Get Thread Scheduling Parameters

Syntax

```
#include <pthread.h>
#include <sched.h>
int pthread_getschedparam(pthread_t thread, int *policy,
                          struct sched_param *param);
```

Threadsafe: Yes

Signal Safe: No

The **pthread_getschedparam()** function retrieves the scheduling parameters of the *thread*. The default OS/400 scheduling policy is **SCHED_OTHER** and cannot be changed to another scheduling policy.

The *sched_policy* field of the *param* parameter is always returned as **SCHED_OTHER**. The *sched_priority* field of the *param* structure is set to the priority of the target thread at the time of the call.

*Do not use **pthread_setschedparam()** to set the priority of a thread if you also use another mechanism (other than the pthread APIs) to set the priority of a thread. If you do, **pthread_getschedparam()** returns only that information that was set via the pthread interfaces such as **pthread_setschedparam()** or a modification of the thread attribute using **pthread_attr_setschedparam()**.*

Parameters

thread

(Input) Pthread handle representing the target thread

policy

(Output) Address of the variable to contain the scheduling policy

param

(Output) Address of the variable to contain the scheduling parameters

Authorities and Locks

None.

Return Value

0

pthread_getschedparam() was successful.

value

pthread_getschedparam was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_getschedparam()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

Related Information

- The <pthread.h> header file. See [Header files for Pthread functions](#).
- [pthread_setschedparam\(\)--Set Target Thread Scheduling Parameters](#)

Example

```
#define _MULTI_THREADED
#include <pthread.h>
#include <sched.h>
#include <stdio.h>
#include "check.h"

void *threadfunc(void *parm)
{
    printf("Inside secondary thread\n");
    sleep(5); /* Sleep is not a very robust way to serialize threads */
    return NULL;
}

int main(int argc, char **argv)
{
    pthread_t      thread;
    int            rc=0;
    struct sched_param param;
    int            policy;

    printf("Enter Testcase - %s\n", argv[0]);

    printf("Create thread using default attributes\n");
    rc = pthread_create(&thread, NULL, threadfunc, NULL);
    checkResults("pthread_create()\n", rc);

    printf("Get scheduling parameters\n");
    rc = pthread_getschedparam(thread, &policy, &param);
    checkResults("pthread_getschedparam()\n", rc);

    printf("The thread scheduling parameters indicate:\n"
           "policy = %d\n", policy);
    printf("priority = %d\n",
           param.sched_priority);

    printf("Main completed\n");
    return 0;
}
```

Output:

```
Enter Testcase - QP0WTEST/TPGSP0
Create thread using default attributes
Get scheduling parameters
The thread scheduling parameters indicate:
policy = 0
priority = 0
Main completed
```

pthread_getthreadid_np()--Retrieve Unique ID for Calling Thread

Syntax

```
#include <pthread.h>
pthread_id_np_t pthread_getthreadid_np(void);
Threadsafe: Yes
Signal Safe: Yes
```

The **pthread_getthreadid_np()** function retrieves the unique integral identifier that can be used to identify the calling thread in some context for application debugging or tracing support.

In some implementations, the thread ID is equivalent to the pthread_t type. In the OS/400 implementation, the pthread_t is an opaque Pthread handle. For the ability to identify a thread via a thread ID (unique number), the **pthread_getunique_np()** and **pthread_getthreadid_np()** interfaces are provided.

The OS/400 machine implementation of threads provides a 64-bit thread ID. The thread ID is returned as a structure containing the high and low order 4 bytes of the 64-bit ID. This allows applications created by compilers that do not yet support 64-bit integral values to effectively use the 64-bit thread ID.

If your code requires the unique integer identifier for the calling thread often, or in a loop, the **pthread_getthreadid_np()** function can significantly improve performance over the combination of **pthread_self()** and **pthread_getunique_np()** calls that provide equivalent behavior.

For example:

```
pthread_id_np_t    tid;
tid = pthread_getthreadid_np();
```

is significantly faster than these calls, but provides the same behavior.

```
pthread_id_np_t    tid;
pthread_t          self;
self = pthread_self();
```

```
pthread_getunique_np(&self, &tid);
```

As always, if you are calling any function too often, you can improve performance by storing the results in a variable or passing to other functions that require the results.

This function is not portable.

Parameters

None.

Authorities and Locks

None.

Return Value

The pthread_id_np_t structure identifying the thread

Error Conditions

None.

Related Information

- The <pthread.h> header file. See [Header files for Pthread functions](#).
- [pthread_self\(\)--Get Pthread Handle](#)
- [pthread_getunique_np\(\)--Retrieve Unique ID for Target Thread](#)

Example

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include "check.h"

#define          NUMTHREADS      3

void *threadfunc(void *parm)
{
    printf("Thread 0x%.8x %.8x started\n", pthread_getthreadid_np());
    return NULL;
}

int main(int argc, char **argv)
{
    pthread_t          thread[NUMTHREADS];
    int                rc=0;
    pthread_id_np_t    tid;
    int                i=0;

    printf("Enter Testcase - %s\n", argv[0]);

    printf("Main Thread 0x%.8x %.8x\n", pthread_getthreadid_np());

    printf("Create %d threads using joinable attributes\n",
          NUMTHREADS);
    for (i=0; i<NUMTHREADS; ++i) {
        rc = pthread_create(&thread[i], NULL, threadfunc, NULL);
        checkResults("pthread_create()\n", rc);
        pthread_getunique_np(&thread[i], &tid);
        printf("Created thread 0x%.8x %.8x\n", tid);
    }

    printf("Join to threads\n");
    for (i=0; i<NUMTHREADS; ++i) {
        rc = pthread_join(thread[i], NULL);
        checkResults("pthread_join()\n", rc);
    }

    printf("Main completed\n");
    return 0;
}
```

Output:

pthread_getthreadid_np()--Retrieve Unique ID for Calling Thread

```
Enter Testcase - QP0WTEST/TPGETT0
Main Thread 0x00000000 0000006c
Create 3 threads using joinable attributes
Created thread 0x00000000 0000006d
Thread 0x00000000 0000006d started
Created thread 0x00000000 0000006e
Created thread 0x00000000 0000006f
Join to threads
Thread 0x00000000 0000006f started
Thread 0x00000000 0000006e started
Main completed
```

pthread_getunique_np()--Retrieve Unique ID for Target Thread

Syntax

```
#include <pthread.h>
int pthread_getunique_np(pthread_t thread, pthread_id_np_t *id);
Threadsafe: Yes
Signal Safe: Yes
```

The **pthread_getunique_np()** function retrieves the unique integral identifier that can be used to identify the thread in some context for application debugging or tracing support.

In some implementations, the thread ID is equivalent to the pthread_t type. In the OS/400 implementation, the pthread_t is an opaque Pthread handle. For the ability to identify a thread via a thread id (unique number), the **pthread_getunique_np()** and **pthread_getthreadid_np()** interfaces are provided.

The OS/400 machine implementation of threads provides a 64-bit thread ID. The thread ID is returned as a structure containing the high and low order 4 bytes of the 64-bit ID. This allows applications created by compilers that do not yet support 64-bit integral values to effectively use the 64-bit thread ID.

If your code requires the unique integer identifier for the calling thread often, or in a loop, the **pthread_getthreadid_np()** function can significantly improve performance over the combination of **pthread_self()** and **pthread_getunique_np()** calls that provide equivalent behavior.

For example:

```
pthread_id_np_t    tid;
tid = pthread_getthreadid_np();
```

is significantly faster than these calls, but provides the same behavior.

```
pthread_id_np_t    tid;
pthread_t          self;
self = pthread_self();
pthread_getunique_np(&self, &tid);
```

As always, if you are calling any function too often, you can improve performance by storing the results in a variable or passing to other functions that require the results.

This function is not portable.

Parameters

thread

(Input) Thread to retrieve the unique integer ID for

id

(Output) Address of the thread ID structure to contain the 64-bit thread ID.

Authorities and Locks

None.

Return Value

0

pthread_getunique_np() was successful.

value

pthread_getunique_np() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_getunique_np()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

Related Information

- The **<pthread.h>** header file. See [Header files for Pthread functions](#)
- [pthread_self\(\)--Get Pthread Handle](#)
- [pthread_getthreadid_np\(\)--Retrieve Unique ID for Calling Thread](#)

Example

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include "check.h"

#define          NUMTHREADS      3

void *threadfunc(void *parm)
{
    pthread_id_np_t tid;
    pthread_t      me = pthread_self();

    pthread_getunique_np(&me, &tid);
    printf("Thread 0x%.8x %.8x started\n", tid);
    return NULL;
}

int main(int argc, char **argv)
{
    pthread_t      thread[NUMTHREADS];
    int            rc=0;
    pthread_id_np_t tid;
    int            i=0;
    pthread_t      me = pthread_self();

    printf("Enter Testcase - %s\n", argv[0]);

    pthread_getunique_np(&me, &tid);
    printf("Main Thread 0x%.8x %.8x\n", tid);

    printf("Create %d threads using joinable attributes\n",
          NUMTHREADS);
    for (i=0; i<NUMTHREADS; ++i) {
        rc = pthread_create(&thread[i], NULL, threadfunc, NULL);
        checkResults("pthread_create()\n", rc);
        pthread_getunique_np(&thread[i], &tid);
        printf("Created thread 0x%.8x %.8x\n", tid);
    }
}
```

pthread_getunique_np())--Retrieve Unique ID for Target Thread

```
printf("Join to threads\n");
for (i=0; i<NUMTHREADS; ++i) {
    rc = pthread_join(thread[i], NULL);
    checkResults("pthread_join()\n", rc);
}

printf("Main completed\n");
return 0;
}
```

Output:

```
Enter Testcase - QP0WTEST/TPGETU0
Main Thread 0x00000000 0000006c
Create 3 threads using joinable attributes
Created thread 0x00000000 0000006d
Thread 0x00000000 0000006d started
Created thread 0x00000000 0000006e
Created thread 0x00000000 0000006f
Join to threads
Thread 0x00000000 0000006f started
Thread 0x00000000 0000006e started
Main completed
```

pthread_is_initialthread_np()--Check if Running in the Initial Thread

Syntax

```
#include <pthread.h>
#include <sched.h>
int pthread_is_initialthread_np(void);
Threadsafe: Yes
Signal Safe: Yes
```

The **pthread_is_initialthread_np()** function returns true or false, indicating if the current thread is the initial thread of the process. A return value true (non 0) indicates that the calling thread is the initial thread. A return value of false (0) indicates that the calling thread is running in a secondary thread. *This function is not portable.*

Parameters

None.

Authorities and Locks

None.

Return Value

0

The calling thread is a secondary thread.

value

The calling thread is the initial thread.

Error Conditions

None.

Related Information

- The **<pthread.h>** header file. See [Header files for Pthread functions.](#)
- [pthread_is_multithreaded_np\(\)--Check the Current Number of Threads](#)

Example

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include "check.h"

#define NUMTHREADS 1

void *function(void *parm)
{
```

pthread_is_initialthread_np()--Check if Running in the Initial Thread

```
    printf("Inside the function\n");
    if (pthread_is_initialthread_np()) {
        printf("In the initial thread\n");
    }
    else {
        printf("In a secondary thread\n");
    }
    return NULL;
}

int main(int argc, char **argv)
{
    pthread_t          thread[NUMTHREADS];
    int                rc=0;
    int                i=0;

    printf("Enter Testcase - %s\n", argv[0]);

    printf("Create %d threads\n",  NUMTHREADS);

    for (i=0; i<NUMTHREADS; ++i) {
        rc = pthread_create(&thread[i], NULL, function, NULL);
        checkResults("pthread_create()\n", rc);
        printf("Main: Currently %d threads\n",
            pthread_is_initialthread_np() + 1);
    }

    printf("Join to threads\n");
    for (i=0; i<NUMTHREADS; ++i) {
        rc = pthread_join(thread[i], NULL);
        checkResults("pthread_join()\n", rc);
    }

    function(NULL);
    printf("Main completed\n");
    return 0;
}
```

Output:

```
Enter Testcase - QP0WTEST/TPISIN0
Create 1 threads
Join to threads
Inside the function
In a secondary thread
Inside the function
In the initial thread
Main completed
```

pthread_is_multithreaded_np()--Check Current Number of Threads

Syntax

```
#include <pthread.h>
#include <sched.h>
int pthread_is_multithreaded_np(void);
Threadsafe: Yes
Signal Safe: Yes
```

The **pthread_is_multithreaded_np()** function returns true or false, indicating whether the current process has more than one thread. A return value of zero indicates that the calling thread is the only thread in the process. A value not equal to zero, indicates that there were multiple other threads in the process at the time of the call to **pthread_is_multithreaded_np()**.

The total number of threads currently in the process can be determined by adding 1 to the return value of **pthread_is_multithreaded_np()**.

This function is not portable.

Parameters

None.

Authorities and Locks

None.

Return Value

0

No other threads exist in the process.

value

There are currently *value+1* total threads in the process.

Error Conditions

None.

Related Information

- The **<pthread.h>** header file. See [Header files for Pthread functions.](#)
- [pthread_is_initialthread_np\(\)--Check if Running in the Initial Thread](#)

Example

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include "check.h"
```

pthread_is_multithreaded_np()--Check Current Number of Threads

```
#define          NUMTHREADS      3

void *threadfunc(void *parm)
{
    int          myHiId;
    int          myId;
    pthread_t     me = pthread_self();

    printf("Inside the New Thread\n");
    sleep(2); /* Sleep is not a very robust way to serialize threads */
    return NULL;
}

int main(int argc, char **argv)
{
    pthread_t     thread[NUMTHREADS];
    int           rc=0;
    int           theHiId=0;
    int           theId=0;
    int           i=0;

    printf("Enter Testcase - %s\n", argv[0]);

    printf("Create %d threads\n",  NUMTHREADS);

    for (i=0; i<NUMTHREADS; ++i) {
        rc = pthread_create(&thread[i], NULL, threadfunc, NULL);
        checkResults("pthread_create()\n", rc);
        printf("Main: Currently %d threads\n",
            pthread_is_multithreaded_np() + 1);
    }

    printf("Join to threads\n");
    for (i=0; i<NUMTHREADS; ++i) {
        rc = pthread_join(thread[i], NULL);
        checkResults("pthread_join()\n", rc);
    }

    if (rc = pthread_is_multithreaded_np()) {
        printf("Error: %d Threads still exist!\n", rc+1);
        exit(1);
    }
    printf("Main completed\n");
    return 0;
}
```

Output:

```
Enter Testcase - QP0WTEST/TPISMTO
Create 3 threads
Main: Currently 2 threads
Main: Currently 3 threads
Main: Currently 4 threads
Join to threads
Inside the New Thread
Inside the New Thread
Inside the New Thread
Main completed
```

pthread_join()--Wait for and Detach Thread

Syntax

```
#include <pthread.h>
int pthread_join(pthread_t thread, void **status);
Threadsafe: Yes
Signal Safe: No
```

The **pthread_join()** function waits for a thread to terminate, detaches the thread, then returns the thread's exit status.

If the *status* parameter is **NULL**, the thread's exit status is not returned.

The meaning of the thread's exit status (value returned to the *status* memory location) is determined by the application, except for the following conditions:

1. When the thread has been canceled using **pthread_cancel()**, the exit status of **PTHREAD_CANCELED** is made available.
2. When the thread has been terminated as a result of an unhandled OS/400 exception, operator intervention or other proprietary OS/400 mechanism, the exit status of **PTHREAD_EXCEPTION_NP** is made available.

Eventually, you should call **pthread_join()**, **pthread_detach()** or **pthread_extendedjoin_np()** without specifying the *leaveThreadAllocated* option for every thread that is created joinable (with a detach state of **PTHREAD_CREATE_JOINABLE**) so that the system can reclaim all resources associated with the thread. Failure to join to or detach joinable threads causes memory and other resource leaks until the process ends.

Parameters

thread

(Input) Pthread handle to the target thread

status

(Output) Address of the variable to receive the thread's exit status

Authorities and Locks

None.

Return Value

0

pthread_join() was successful.

value

pthread_join() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_join()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

[ESRCH]

The thread specified could not be found.

Related Information

- The `<pthread.h>` header file. See [Header files for Pthread functions](#).
- [pthread_detach\(\)--Detach Thread](#)
- [pthread_exit\(\)--Terminate Calling Thread](#)
- [pthread_extendedjoin_np\(\)--Wait for Thread with Extended Options](#)
- [pthread_join_np\(\)--Wait for Thread to End](#)

Example

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include "check.h"

int  okStatus    = 34;

void *threadfunc(void *parm)
{
    printf("Inside secondary thread\n");
    return __VOID(okStatus);
}

int main(int argc, char **argv)
{
    pthread_t      thread;
    int            rc=0;
    void           *status;

    printf("Enter Testcase - %s\n", argv[0]);

    printf("Create thread using attributes that allow join\n");
    rc = pthread_create(&thread, NULL, threadfunc, NULL);
    checkResults("pthread_create()\n", rc);

    printf("Wait for the thread to exit\n");
    rc = pthread_join(thread, &status);
    checkResults("pthread_join()\n", rc);
    if (__INT(status) != okStatus) {
        printf("Secondary thread failed\n");
        exit(1);
    }

    printf("Got secondary thread status as expected\n");
    printf("Main completed\n");
    return 0;
}
```

Output:

```
Enter Testcase - QP0WTEST/TPJOIN0
Create thread using attributes that allow join
Wait for the thread to exit
Inside secondary thread
Got secondary thread status as expected
Main completed
```


pthread_join_np()--Wait for Thread to End

Syntax

```
#include <pthread.h>
int pthread_join_np(pthread_t thread, void **status);
Threadsafe: Yes
Signal Safe: No
```

The **pthread_join_np()** function waits for a thread to terminate, then returns the thread's exit status, while leaving the data structures of the thread available for a later call to **pthread_join()**, **pthread_join_np()**, **pthread_detach()**, or **pthread_extendedjoin_np()**.

If the *status* parameter is **NULL**, the thread's exit status is not returned.

The meaning of the thread's exit status (value returned to the *status* memory location) is determined by the application except for the following conditions:

1. When the thread has been canceled using **pthread_cancel()**, the exit status of **PTHREAD_CANCELED** is made available.
2. When the thread has been terminated as a result of an unhandled OS/400 exception, operator intervention, or other proprietary OS/400 mechanism, the exit status of **PTHREAD_EXCEPTION_NP** is made available.

Eventually, you should call **pthread_join()**, **pthread_detach()**, or **pthread_extendedjoin_np()** without specifying the *leaveThreadAllocated* option for every thread that is created joinable (with a detach state of **PTHREAD_CREATE_JOINABLE**) so that the system can reclaim all resources associated with the thread. Failure to join to or detach joinable threads causes memory and other resource leaks until the process ends.

This function is not portable.

Parameters

thread

(Input) Pthread handle to the target thread

status

(Output) Address of the variable to receive the thread's exit status

Authorities and Locks

None.

Return Value

0

pthread_join_np() was successful.

value

pthread_join_np() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_join_np()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

pthread_join_np()--Wait for Thread to End

[ESRCH]

The thread specified could not be found.

Related Information

- The <pthread.h> header file. See [Header files for Pthread functions](#).
- [pthread_detach\(\)--Detach Thread](#)
- [pthread_exit\(\)--Terminate Calling Thread](#)
- [pthread_extendedjoin_np\(\)--Wait for Thread with Extended Options](#)
- [pthread_join\(\)--Wait for and Detach Thread](#)

Example

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include "check.h"

int okStatus = 12;

void *threadfunc(void *parm)
{
    printf("Inside secondary thread\n");
    return __VOID(okStatus);
}

int main(int argc, char **argv)
{
    pthread_t      thread;
    int            rc=0;
    void           *status;

    printf("Enter Testcase - %s\n", argv[0]);

    printf("Create thread using attributes that allow join\n");
    rc = pthread_create(&thread, NULL, threadfunc, NULL);
    checkResults("pthread_create()\n", rc);

    printf("Wait for the thread to exit\n");
    rc = pthread_join_np(thread, &status);
    checkResults("pthread_join_np()\n", rc);
    if (__INT(status) != okStatus) {
        printf("Secondary thread failed\n");
        exit(1);
    }

    printf("With pthread_join_np(), we can join repeatedly\n");
    rc = pthread_join_np(thread, &status);
    checkResults("pthread_join_np()\n", rc);
    if (__INT(status) != okStatus) {
        printf("Secondary thread failed\n");
        exit(1);
    }

    printf("Got secondary thread status as expected\n");
    /* Eventually, we should use pthread_join() or pthread_detach() */
}
```

pthread_join_np()--Wait for Thread to End

```
rc = pthread_detach(thread);  
checkResults("pthread_detach()\n", rc);  
  
printf("Main completed\n");  
return 0;  
}
```

Output:

```
Enter Testcase - QP0WTEST/TPJOINNO  
Create thread using attributes that allow join  
Wait for the thread to exit  
Inside secondary thread  
With pthread_join_np(), we can join repeatedly  
Got secondary thread status as expected  
Main completed
```

pthread_once()--Perform One-Time Initialization

Syntax

```
#include <pthread.h>
int pthread_once(pthread_once_t *once_control, void (*init_routine)(void));
Threadsafe: Yes
Signal Safe: No
```

The **pthread_once()** function performs one time initialization based on a specific *once_control* variable. The *init_routine* is called only one time when multiple calls to **pthread_once()** use the same *once_control*.

The *once_control* variable is not set until the *init_routine* returns. If the *init_routine* is a cancellation point and the thread calling the *init_routine* via **pthread_once()** is canceled, the *once_control* variable will not be set and a subsequent call to **pthread_once()** using that *once_control* variable will result in another call to the *init_routine*.

You must initialize the *once_control* variable to PTHREAD_ONCE_INIT prior to calling **pthread_once()** with it.

The function passed as *init_routine* must correspond to the following C function prototype:

```
void initRoutine(void);
```

Parameters

once_control

(Input) The control variable associated with this initialization.

init_routine

(Input) A function pointer to a routine that takes no parameters and returns no value.

Authorities and Locks

None.

Return Value

0

pthread_once() was successful.

value

pthread_once() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_once()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

Related Information

- The <pthread.h> header file. See [Header files for Pthread functions](#).

Example

```

#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include "check.h"

#define          NUMTHREADS      3
int             number          = 0;
int             okStatus        = 777;
pthread_once_t  onceControl     = PTHREAD_ONCE_INIT;

void initRoutine(void)
{
    printf("In the initRoutine\n");
    number++;
}

void *threadfunc(void *parm)
{
    printf("Inside secondary thread\n");
    pthread_once(&onceControl, initRoutine);
    return __VOID(okStatus);
}

int main(int argc, char **argv)
{
    pthread_t      thread[NUMTHREADS];
    int            rc=0;
    int            i=NUMTHREADS;
    void           *status;

    printf("Enter Testcase - %s\n", argv[0]);

    for (i=0; i < NUMTHREADS; ++i) {
        printf("Create thread %d\n",
            i);
        rc = pthread_create(&thread[i], NULL, threadfunc, NULL);
        checkResults("pthread_create()\n", rc);
    }

    for (i=0; i < NUMTHREADS; ++i) {
        printf("Wait for thread %d\n", i);
        rc = pthread_join(thread[i], &status);
        checkResults("pthread_join()\n", rc);
        if (__INT(status) != okStatus) {
            printf("Secondary thread failed\n");
            exit(1);
        }
    }

    if (number != 1) {
        printf("An incorrect number of 1 one-time init routine was called!\n");
        exit(1);
    }
    printf("One-time init routine called exactly once\n");
    printf("Main completed\n");
    return 0;
}

```

Output:

```
Enter Testcase - QP0WTEST/TPONCE0
Create thread 0
Create thread 1
Create thread 2
Wait for thread 0
Inside secondary thread
In the initRoutine
Inside secondary thread
Wait for thread 1
Wait for thread 2
Inside secondary thread
One-time init routine called exactly once
Main completed
```

pthread_self()--Get Pthread Handle

Syntax

```
#include <pthread.h>
pthread_t pthread_self(void);
Threadsafe: Yes
Signal Safe: Yes
```

The **pthread_self()** function returns the Pthread handle of the calling thread. The pthread_self() function does **NOT** return the integral thread of the calling thread. You must use **pthread_getthreadid_np()** to return an integral identifier for the thread.

If your code requires the unique integer identifier for the calling thread often, or in a loop, the **pthread_getthreadid_np()** function can provide significant performance improvements over the combination of **pthread_self()**, and **pthread_getunique_np()** calls that provide equivalent behavior.

For example:

```
pthread_id_np_t    tid;
tid = pthread_getthreadid_np();
```

is significantly faster than these calls, but provides the same behavior.

```
pthread_id_np_t    tid;
pthread_t          self;
self = pthread_self();
pthread_getunique_np(&self, &tid);
```

As always, if you are calling any function too often, performance improvements can be gained by storing the results in a variable and or passing to other functions which require the results.

Parameters

None.

Authorities and Locks

None.

Return Value

pthread_t

pthread_self() returns the Pthread handle of the calling thread.

Error Conditions

None.

Related Information

- The **<pthread.h>** header file. See [Header files for Pthread functions](#).
- [pthread_equal\(\)--Compare Two Threads](#)
- [pthread_getthreadid_np\(\)--Retrieve Unique ID for Calling Thread](#)
- [pthread_getunique_np\(\)--Retrieve Unique ID for Target Thread](#)

Example

```
#include <pthread.h>
#include <pthread.h>
#include <stdio.h>
#include "check.h"

pthread_t    theThread;

void *threadfunc(void *parm)
{
    printf("Inside secondary thread\n");
    theThread = pthread_self();
    return NULL;
}

int main(int argc, char **argv)
{
    pthread_t    thread;
    int          rc=0;

    printf("Entering testcase\n");

    printf("Create thread using default attributes\n");
    rc = pthread_create(&thread, NULL, threadfunc, NULL);
    checkResults("pthread_create()\n", rc);

    /* sleep() is not a very robust way to wait for the thread */
    sleep(5);

    printf("Check if the thread got its thread handle\n");
    if (!pthread_equal(thread, theThread)) {
        printf("Unexpected results on pthread_equal()!\n");
        exit(1);
    }
    printf("pthread_self() returned the thread handle\n");
    printf("Main completed\n");
    return 0;
}
```

Output:

```
Entering testcase
Create thread using default attributes
Inside secondary thread
Check if the thread got its thread handle
pthread_self() returned the thread handle
Main completed
```


pthread_setconcurrency()--Set Process Concurrency Level

Syntax

```
#include <pthread.h>
int pthread_setconcurrency(int concurrency);
Threadsafe: Yes
Signal Safe: No
```

The **pthread_setconcurrency()** function sets the current concurrency level for the process.

A concurrency value of zero indicates that the threads implementation chooses the concurrency level that best suits the application. A concurrency level greater than zero indicates that the application wants to inform the system of its desired concurrency level.

The concurrency level is not used by the OS/400 threads implementation, but is stored for subsequent calls to **pthread_getconcurrency()**. Each user thread is always bound to a kernel thread.

Parameters

concurrency

(Input) The new concurrency level for the process

Authorities and Locks

None.

Return Value

0

pthread_setconcurrency() was successful.

value

pthread_setconcurrency() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_setconcurrency()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

Related Information

- The **<pthread.h>** header file. See [Header files for Pthread functions](#).
- [pthread_getconcurrency\(\)--Get Process Concurrency Level](#)

pthread_setpthreadoption_np()--Set Pthread Run-Time Option Data

Syntax

#include <pthread.h>
void pthread_setpthreadoption_np(pthread_option_np_t *optionData);
Threadsafe: Yes
Signal Safe: Yes

The **pthread_setpthreadoption_np()** function sets option data in the pthread run-time for the process.

Input data is specified uniquely based on the specified *optionData*. See the table below for details about input and output. The option field in the *optionData* parameter is always required; other fields may be input, output, or ignored, based on the specific option used.

For all options, every reserved field in the structure represented by *optionData* must be binary zero or the **EINVAL** error is returned. Unless otherwise noted for an option, the *target* field in the *option* parameter is always ignored, and the contents of the *optionData* structure is not changed by the **pthread_setpthreadoption_np()** function.

The currently supported options, the data they represent, and the valid operations are as follows:

<i>option</i> field of the <i>option</i> parameter	Description
PTHREAD_OPTION_POOL_NP	When a thread terminates and is detached or joined to, certain data structures from the pthreads run-time are maintained in a pool for possible reuse by future threads. This improves performance for creating threads. Typically, an application should not be concerned with this storage pool. Use this option to set the current maximum size of the allowed storage pool. The <i>optionValue</i> field of the <i>optionData</i> parameter is used to set the current maximum number of thread structures that will be allowed in the storage pool. By default, the <i>optionValue</i> field must be a valid integer greater than or equal to zero, or the EINVAL error is returned. The default maximum size of the storage reuse pool contains enough room for 512 thread structures.
PTHREAD_OPTION_POOL_CURRENT_NP	If the <i>option</i> field of the <i>optionData</i> parameter is set to this option, the EINVAL error is returned.
PTHREAD_OPTION_THREAD_CAPABLE_NP	If the <i>option</i> field of the <i>optionData</i> parameter is set to this option, the EINVAL error is returned.

Parameters

option

(Input/Output) Address of the variable containing option information and to contain output option information

Authorities and Locks

None.

Return Value

0

pthread_getpthreadoption_np() was successful.

value

pthread_getpthreadoption_np() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_getpthreadoption_np()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

Related Information

- The **<pthread.h>** header file. See [Header files for Pthread functions](#).
- [pthread_getpthreadoption_np\(\)--Get Pthread Run-Time Option Data](#)

Example

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include "check.h"

#define NUMTHREADS    5

void *threadfunc(void *parm)
{
    printf("Inside the thread\n");
    return NULL;
}

void showCurrentSizeOfPool(void) {
    int                rc;
    pthread_option_np_t  opt;

    memset(&opt, 0, sizeof(opt));
    opt.option = PTHREAD_OPTION_POOL_CURRENT_NP;
    rc = pthread_getpthreadoption_np(&opt);
    checkResults("pthread_getpthreadoption_np()\n", rc);

    printf("Current number of thread structures in pool is %d\n",
           opt.optionValue);
    return;
}

int main(int argc, char **argv)
{
    pthread_t          thread[NUMTHREADS];
    int                rc=0;
    int                i=0;
    pthread_option_np_t  opt;

    printf("Enter Testcase - %s\n", argv[0]);

    printf("Create threads and prime the storage pool\n");
    for (i=0; i<NUMTHREADS; ++i) {
        rc = pthread_create(&thread[i], NULL, threadfunc, NULL);
```

pthread_setpthreadoption_np()--Set Pthread Run-Time Option Data

```
    checkResults("pthread_create(NULL)\n", rc);
}

printf("Joining all threads at once so thread n does not reuse\n"
      "thread n-1's data structures\n");
for (i=0; i<NUMTHREADS; ++i) {
    rc = pthread_join(thread[i], NULL);
    checkResults("pthread_join()\n", rc);
}

showCurrentSizeOfPool();

/* Set the maximum size of the storage pool to 0. I.e. No reuse of  */
/* pthread structures                                              */
printf("Set the max size of the storage pool to 0\n");
memset(&opt, 0, sizeof(opt));
opt.option      = PTHREAD_OPTION_POOL_NP;
opt.optionValue = 0;
rc = pthread_setpthreadoption_np(&opt);
checkResults("pthread_setpthreadoption_np()\n", rc);

printf("Create some more threads. Each thread structure will come\n"
      "from the storage pool if it exists, but based on the max size of
0,\n"
      "the thread structure will not be allowed to be reused\n");
for (i=0; i<NUMTHREADS; ++i) {
    rc = pthread_create(&thread[i], NULL, threadfunc, NULL);
    checkResults("pthread_create(NULL)\n", rc);

    showCurrentSizeOfPool();

    rc = pthread_join(thread[i], NULL);
    checkResults("pthread_join()\n", rc);
}

printf("Main completed\n");
return 0;
}
```

Output:

```
Enter Testcase - QP0WTEST/TPSETOPT
Create threads and prime the storage pool
Joining all threads at once so thread n does not reuse
thread n-1's data structures
Inside the thread
Inside the thread
Inside the thread
Inside the thread
Inside the thread
Current number of thread structures in pool is 5
```

```
Set the max size of the storage pool to 0
Create some more threads. Each thread structure will come
from the storage pool if it exists, but based on the max size of 0,
the thread structure will not be allowed to be reused
Current number of thread structures in pool is 4
Inside the thread
```

pthread_setpthreadoption_np()--Set Pthread Run-Time Option Data

```
Current number of thread structures in pool is 3
Inside the thread
Current number of thread structures in pool is 2
Inside the thread
Current number of thread structures in pool is 1
Inside the thread
Current number of thread structures in pool is 0
Inside the thread
Main completed
```

pthread_setschedparam()--Set Target Thread Scheduling Parameters

Syntax

```
#include <pthread.h>
#include <sched.h>
int pthread_setschedparam(pthread_t thread, int policy,
                          const struct sched_param *param);
```

Threadsafe: Yes

Signal Safe: No

The **pthread_setschedparam()** function sets the scheduling parameters of the target thread. The supported OS/400 scheduling *policy* is **SCHED_OTHER**. An attempt to set the *policy* to a value other than this cause the **EINVAL** error. The *sched_priority* field of the *param* parameter must range from **PRIORITY_MIN** to **PRIORITY_MAX** or the **ENOTSUP** error occurs.

All reserved fields in the scheduling parameters structure must be binary 0 or the **EINVAL** error occurs.

*Do not use **pthread_setschedparam()** to set the priority of a thread if you also use another mechanism (outside of the pthread APIs) to set the priority of a thread. If you do, **pthread_getschedparam()** returns only that information that was set via the pthread interfaces (**pthread_setschedparam()** or modification of the thread attribute using **pthread_attr_setschedparam()**).*

Parameters

thread

(Input) Pthread handle of the target thread

policy

(Input) Scheduling policy (must be **SCHED_OTHER**)

param

(Input) Scheduling parameters

Authorities and Locks

None.

Return Value

0

pthread_setschedparam() was successful.

value

pthread_setschedparam() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_setschedparam()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

[ENOTSUP]

The value specified for the priority argument is not supported.

Related Information

- The <pthread.h> header file. See [Header files for Pthread functions](#).
- [pthread_getschedparam\(\)--Get Thread Scheduling Parameters](#)

Example

```
#define _MULTI_THREADED
#include <pthread.h>
#include <sched.h>
#include <stdio.h>
#include "check.h"

#define BUMP_PRIO 1
int thePriority = 0;

int showSchedParam(pthread_t thread)
{
    struct sched_param param;
    int policy;
    int rc;

    printf("Get scheduling parameters\n");
    rc = pthread_getschedparam(thread, &policy, &param);
    checkResults("pthread_getschedparam()\n", rc);

    printf("The thread scheduling parameters indicate:\n"
           "priority = %d\n", param.sched_priority);
    return param.sched_priority;
}

void *threadfunc(void *parm)
{
    int rc;

    printf("Inside secondary thread\n");
    thePriority = showSchedParam(pthread_self());
    sleep(5); /* Sleep is not a very robust way to serialize threads */
    return NULL;
}

int main(int argc, char **argv)
{
    pthread_t thread;
    int rc=0;
    struct sched_param param;
    int policy = SCHED_OTHER;
    int theChangedPriority=0;

    printf("Enter Testcase - %s\n", argv[0]);

    printf("Create thread using default attributes\n");
    rc = pthread_create(&thread, NULL, threadfunc, NULL);
    checkResults("pthread_create()\n", rc);

    sleep(2); /* Sleep is not a very robust way to serialize threads */
}
```

```

memset(&param, 0, sizeof(param));
/* Bump the priority of the thread a small amount */
if (thePriority - BUMP_PRIO >= PRIORITY_MIN_NP) {
    param.sched_priority = thePriority - BUMP_PRIO;
}

printf("Set scheduling parameters, prio=%d\n",
       param.sched_priority);
rc = pthread_setschedparam(thread, policy, &param);
checkResults("pthread_setschedparam()\n", rc);

/* Let the thread fill in its own last priority */
theChangedPriority = showSchedParam(thread);

if (thePriority == theChangedPriority ||
    param.sched_priority != theChangedPriority) {
    printf("The thread did not get priority set correctly, "
          "first=%d last=%d expected=%d\n",
          thePriority, theChangedPriority, param.sched_priority);
    exit(1);
}

sleep(5); /* Sleep is not a very robust way to serialize threads */
printf("Main completed\n");
return 0;
}

```

Output:

```

Enter Testcase - QP0WTEST/TPSSP0
Create thread using default attributes
Inside secondary thread
Get scheduling parameters
The thread scheduling parameters indicate:
priority = 0
Set scheduling parameters, prio=-1
Get scheduling parameters
The thread scheduling parameters indicate:
priority = -1
Main completed

```


pthread_trace_init_np()--Initialize or Re-initialize pthread tracing

Syntax

```
#include <pthread.h>
int pthread_trace_init_np(void);
Threadsafe: Yes
Signal Safe: No
```

The **pthread_trace_init_np()** API initializes or refreshes both the Pthreads library trace level and the application trace level. The Pthreads library trace level is maintained internally by the Pthreads library while the application trace level is stored in the *Qp0wTraceLevel* external variable and can be used via the **PTHREAD_TRACE_NP()** macro.

When a program or service program that uses the Pthread APIs causes the Pthread APIs to be loaded (activated), the Pthreads library automatically calls the **pthread_trace_init_np()** function in order to initialize tracing based on the value of the **QIBM_PTHREAD_TRACE_LEVEL** environment variable at that time.

The application can call **pthread_trace_init_np()** at an arbitrary time during execution to initialize or refresh the current Pthreads library tracing level and the application trace level. The trace level is set based on the value of the **QIBM_PTHREAD_TRACE_LEVEL** environment variable at the time of the call. The new tracing level is also returned.

The Pthreads library tracing level is used to control trace records written by the Pthreads library functions at runtime. The following table describes the preprocessor macros representing the various trace levels, the setting of the **QIBM_PTHREAD_TRACE_LEVEL** environment variable, and the conditions that are traced.

Trace Level	EnvVar	Description
PTHREAD_TRACE_NONE_NP	"QIBM_PTHREAD_TRACE_LEVEL=0" (or not set)	No tracing is performed by the Pthreads library. Application tracing may still be done.
PTHREAD_TRACE_ERROR_NP	"QIBM_PTHREAD_TRACE_LEVEL=1"	Error level traces error conditions and the causes of most error return codes.
PTHREAD_TRACE_INFO_NP	"QIBM_PTHREAD_TRACE_LEVEL=2"	Informational level traces error level tracepoints, plus entry to and exit from functions, parameters passed to and return codes from functions, major changes in control flow.

PTHREAD_TRACE_VERBOSE_NP	"QIBM_PTHREAD_TRACE_LEVEL=3"	Verbose level traces informational level tracepoints, plus detailed information about application parameters, threads and data structures including information about Pthreads library processing information.
---------------------------------	------------------------------	---

The application provides tracing support similar to the Pthreads library using the **PTHREAD_TRACE_NP()** macro.

The **PTHREAD_TRACE_NP()** macro uses the external variable *Qp0wTraceLevel*. *Qp0wTraceLevel* may be used directly by the application to set application trace level without effecting the current Pthread library trace level. Set the value of *Qp0wTraceLevel* to one of **PTHREAD_TRACE_NONE_NP**, **PTHREAD_TRACE_ERROR_NP**, **PTHREAD_TRACE_INFO_NP**, or **PTHREAD_TRACE_VERBOSE_NP**.

The **PTHREAD_TRACE_NP()** macro can be used in conjunction with the following APIs to put trace records into the user trace flight recorder. The following system APIs defined in the qp0ztrc.h header file:

- Qp0zUprintf - print formatted trace data
- Qp0zDump - dump formatted hex data
- Qp0zDumpStack - dump the call stack of the calling thread
- Qp0zDumpTargetStack - dump the call stack of the target thread

The trace records are written to the user trace flight recorder and can be accessed via the following CL commands:

- DMPUSRTRC - dump the contents of a specified job's trace
- CHGUSRTRC - change attributes (size, wrapping, clear) of a specified job's trace
- DLTUSRTRC - delete the persistent trace object associated with a job's trace

Parameters

None.

Authorities and Locks

None.

Return Value

value

The new trace level. One of **PTHREAD_TRACE_NONE_NP**, **PTHREAD_TRACE_ERROR_NP**, **PTHREAD_TRACE_INFO_NP**, or **PTHREAD_TRACE_VERBOSE_NP**.

Error Conditions

None.

Related Information

- The `<pthread.h>` header file. See [Header files for Pthread functions](#).
- [PTHREAD_TRACE_NP\(\)--Execute code based on trace level \(Macro\)](#)

Example

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <qp0ztrc.h>

#define checkResults(string, val) { \
    if (val) { \
        printf("Failed with %d at %s", val, string); \
        exit(1); \
    } \
}

typedef struct {
    int          threadSpecific1;
    int          threadSpecific2;
} threadSpecific_data_t;

#define          NUMTHREADS    2
pthread_key_t    threadSpecificKey;

void foo(void);
void bar(void);
void dataDestructor(void *);

void *theThread(void *parm) {
    int          rc;
    threadSpecific_data_t  *gData;
    PTHREAD_TRACE_NP({
        Qp0zUprintf("Thread Entered\n");
        Qp0zDump("Global Data", parm,
sizeof(threadSpecific_data_t));},
        PTHREAD_TRACE_INFO_NP);
    gData = (threadSpecific_data_t *)parm;
    rc = pthread_setspecific(threadSpecificKey, gData);
    checkResults("pthread_setspecific()\n", rc);
    foo();
    return NULL;
}

void foo() {
    threadSpecific_data_t *gData =
        (threadSpecific_data_t *)pthread_getspecific(threadSpecificKey);
    PTHREAD_TRACE_NP(Qp0zUprintf("foo(), threadSpecific data=%d %d\n",
        gData->threadSpecific1,
```

pthread_trace_init_np()--Initialize or Re-initialize pthread tracing

```
gData->threadSpecific2);,
        PTHREAD_TRACE_INFO_NP);
    bar();
    PTHREAD_TRACE_NP(Qp0zUprintf("foo(): This is an error tracepoint\n");,
        PTHREAD_TRACE_ERROR_NP);
}

void bar() {
    threadSpecific_data_t *gData =
        (threadSpecific_data_t *)pthread_getspecific(threadSpecificKey);
    PTHREAD_TRACE_NP(Qp0zUprintf("bar(), threadSpecific data=%d %d\n",
        gData->threadSpecific1,
gData->threadSpecific2);,
        PTHREAD_TRACE_INFO_NP);
    PTHREAD_TRACE_NP(Qp0zUprintf("bar(): This is an error tracepoint\n");
        Qp0zDumpStack("This thread's stack at time of error in
bar()");,
        PTHREAD_TRACE_ERROR_NP);
    return;
}

void dataDestructor(void *data) {
    PTHREAD_TRACE_NP(Qp0zUprintf("dataDestructor: Free data\n");,
        PTHREAD_TRACE_INFO_NP);
    pthread_setspecific(threadSpecificKey, NULL);    free(data);
    /* If doing verbose tracing we'll even write a message to the job log */
    PTHREAD_TRACE_NP(Qp0zLprintf("Free'd the thread specific data\n");,
        PTHREAD_TRACE_VERBOSE_NP);
}

/* Call this testcase with an optional parameter 'PTHREAD_TRACING' */
/* If the PTHREAD_TRACING parameter is specified, then the */
/* Pthread tracing environment variable will be set, and the */
/* pthread tracing will be re initialized from its previous value. */
/* NOTE: We set the trace level to informational, tracepoints cut */
/*      using PTHREAD_TRACE_NP at a VERBOSE level will NOT show up*/
int main(int argc, char **argv) {
    pthread_t          thread[NUMTHREADS];
    int                rc=0;
    int                i;
    threadSpecific_data_t *gData;
    char                buffer[50];

    PTHREAD_TRACE_NP(Qp0zUprintf("Enter Testcase - %s\n", argv[0]);,
        PTHREAD_TRACE_INFO_NP);
    if (argc == 2 && !strcmp("PTHREAD_TRACING", argv[1])) {
        /* Turn on internal pthread function tracing support */
        /* Or, use ADDENVVAR, CHGENVVAR CL commands to set this envvar*/
        sprintf(buffer, "QIBM_PTHREAD_TRACE_LEVEL=%d", PTHREAD_TRACE_INFO_NP);
        putenv(buffer);
        /* Refresh the Pthreads internal tracing with the environment */
        /* variables value. */
        pthread_trace_init_np();
    }
    else {
        /* Trace only our application, not the Pthread code */
        Qp0wTraceLevel = PTHREAD_TRACE_INFO_NP;
    }

    rc = pthread_key_create(&threadSpecificKey, dataDestructor);
    checkResults("pthread_key_create()\n", rc);
}
```

```

    for (i=0; i < NUMTHREADS; ++i) {
        PTHREAD_TRACE_NP(Qp0zUprintf("Create/start a thread\n");,
                        PTHREAD_TRACE_INFO_NP);
        /* Create per-thread threadSpecific data and pass it to the thread */
        gData = (threadSpecific_data_t *)malloc(sizeof
(threadSpecific_data_t));
        gData->threadSpecific1 = i;
        gData->threadSpecific2 = (i+1)*2;
        rc = pthread_create( &thread[i], NULL, theThread, gData);
        checkResults("pthread_create()\n", rc);
        PTHREAD_TRACE_NP(Qp0zUprintf("Wait for the thread to complete, "
                                "and release their resources\n");,
                        PTHREAD_TRACE_INFO_NP);
        rc = pthread_join(thread[i], NULL);
        checkResults("pthread_join()\n", rc);
    }

    pthread_key_delete(threadSpecificKey);
    PTHREAD_TRACE_NP(Qp0zUprintf("Main completed\n");,
                    PTHREAD_TRACE_INFO_NP);

    return 0;
}

```

Output

Use CL command **DMPUSRTRC** to output the following tracing information that the example creates. The **DMPUSRTRC** CL command causes the following information to be put into file QTEMP/QAP0ZDMP or to standard output depending on the options used for the CL command.

Note the following:

- The trace records are indented and labeled based on thread id plus a microsecond timestamp at the time the tracepoint was cut. In the following trace record, the value *00000018* indicates the thread ID of the thread that created the tracepoint. The value *972456* indicates that the tracepoint occurred 972456 microseconds after the last timestamp indicator.

```
00000018:972456 pthread_trace_init_np(): New traceLevel=2
```

- You can use the Pthread library tracepoints to debug incorrect calls to the Pthreads library from your application.
- The following trace output occurs when the optional parameter 'PTHREAD_TRACING' IS specified when calling this program. The 'PTHREAD_TRACING' parameter causes the **pthread_trace_init_np()** function to be used which initializes the Pthreads library tracing.
- There is **significantly** more information traced than the example shown in the documentation for the **PTHREAD_TRACE_NP()** macro
- The function names for threads and data destructors are traced.
- The values for many Pthread API parameters are traced, allowing application debug.
- Some internal Pthread API information is traced at an information-level tracing when the control flow information is critical.

User Trace Dump for job 097979/KULACK/PTHREADT. Size: 300K, Wrapped 0 times.

```
--- 11/09/1998 15:15:56 ---
```

```

00000018:972456 pthread_trace_init_np(): New traceLevel=2
00000018:972592 pthread_key_create(entry): dtor=a1000000 00000000 d161cc19
45001a00
00000018:993920                                destructor name is
'dataDestructor__FPv'
00000018:994048 pthread_key_create(exit): newKey=0, rc=0
00000018:994120 Create/start a thread

```

pthread_trace_init_np()--Initialize or Re-initialize pthread tracing

```
00000018:994224 pthread_create(entry): thread=80000000 00000000 f11d9cc7
23000400
00000018:994296 attr=00000000 00000000 00000000
00000000
00000018:994376 start_routine=a1000000 00000000
dl161cc19 45006980
00000018:995320 routine name is 'theThread__FPv'
00000018:995432 arg=80000000 00000000 e7c74b3e
04001cd0
00000018:995992 pthread_create(status): Create a new thread
00000018:996088 Joinable=1
00000018:996152 PrioInheritSched-EXPLICIT Prio=0
00000018:997488 pthread_create(exit): Success
00000018:997632 tcb=80000000 00000000 feb52907
07001000
00000018:997704 thread id=00000000 00000019
handle=00000007
00000018:997792 Wait for the thread to complete, and release their
resources
00000018:997896 pthread_join_processor(entry): Target 00000000 00000019,
Detach=1, time=00000000 sec, 00000000 nanosec.
00000018:997968 statusp = 00000000 00000000
00000000 00000000
00000019:998720 pthread_create_part2(status): run the new thread: 00000000
00000019
00000019:998864 Thread Entered
00000019:998984 E7C74B3E04:001CD0 L:0008 Global Data
00000019:999144 E7C74B3E04:001CD0 00000000 00000002
*.....*
00000019:999240 pthread_setspecific(entry): value=80000000 00000000
e7c74b3e 04001cd0, key=0
00000019:999320 pthread_getspecific(entry): key=0
00000019:999392 foo(), threadSpecific data=0 2
00000019:999464 pthread_getspecific(entry): key=0
00000019:999536 bar(), threadSpecific data=0 2
00000019:999600 bar(): This is an error tracepoint
00000019:999664 Stack Dump For Current Thread
00000019:999728 Stack: This thread's stack at time of error in bar()
--- 11/09/1998 15:15:57 ---
00000019:000304 Stack: Library / Program Module Stmt
Procedure
00000019:000472 Stack: QSYS / QLESPI QLECRTTH 774 :
LE_Create_Thread2__FP12crtth_parm_t
00000019:000560 Stack: QSYS / QP0WPTHr QP0WPTHr 1008 :
pthread_create_part2
00000019:000656 Stack: KULACK / PTHREAdT PTHREAdT 19 :
theThread__FPv
00000019:000728 Stack: KULACK / PTHREAdT PTHREAdT 29 :
foo__Fv
00000019:000808 Stack: KULACK / PTHREAdT PTHREAdT 46 :
bar__Fv
00000019:000888 Stack: QSYS / QP0ZCPA QP0ZUDBG 87 :
Qp0zDumpStack
00000019:007416 Stack: QSYS / QP0ZSCPA QP0ZSCPA 276 :
Qp0zSUDumpStack
00000019:007504 Stack: QSYS / QP0ZSCPA QP0ZSCPA 287 :
Qp0zSUDumpTargetStack
00000019:007544 Stack: Completed
00000019:007664 foo(): This is an error tracepoint
00000019:007752 pthread_create_part2(status): return from start routine,
status=00000000 00000000 00000000 00000000
```

pthread_trace_init_np()--Initialize or Re-initialize pthread tracing

```
00000019:007816 pthread_cleanup(entry): Thread termination started
00000019:007888 Qp0wTlsVector::invokeHandlers(entry):
00000019:007952 Qp0wTlsVector::invokeHandler(invoke): key=0
00000019:008040                                     dtor=a1000000
00000000 d161cc19 45001a00,
00000019:010792                                     destructor name is
'dataDestructor__FPv'
00000019:010920                                     arg=80000000
00000000 e7c74b3e 04001cd0
00000019:011008 dataDestructor: Free data
00000019:011096 pthread_setspecific(entry): value=00000000 00000000
00000000 00000000, key=0
00000019:011184 pthread_cleanup(exit): returning
00000018:011624 pthread_join_processor(status): target status=00000000
00000000 00000000 00000000, state=0x03, YES
00000018:011752 Create/start a thread
00000018:011880 pthread_create(entry): thread=80000000 00000000 f11d9cc7
23000430
00000018:011952                                     attr=00000000 00000000 00000000
00000000
00000018:012032                                     start_routine=a1000000 00000000
d161cc19 45006980
00000018:013464                                     routine name is 'theThread__FPv'
00000018:013576                                     arg=80000000 00000000 e7c74b3e
04001cd0
00000018:013704 Qp0wTcb::Qp0wTcb(status): Tcb was reused: tcb=80000000
00000000 feb52907 07001000
00000018:013784 pthread_create(status): Create a new thread
00000018:013848                                     Joinable-1
00000018:013912                                     PrioInheritSched-EXPLICIT Prio-0
00000018:014736 pthread_create(exit): Success
00000018:014912                                     tcb=80000000 00000000 feb52907
07001000
00000018:014984                                     thread id=00000000 0000001a
handle=00000007
00000018:015072 Wait for the thread to complete, and release their
resources
00000018:015168 pthread_join_processor(entry): Target 00000000 0000001a,
Detach=1, time=00000000 sec, 00000000 nanosec.
00000018:015240                                     statusp = 00000000 00000000
00000000 00000000
0000001A:015696 pthread_create_part2(status): run the new thread:
00000000 0000001a
0000001A:015840 Thread Entered
0000001A:015968 E7C74B3E04:001CD0 L:0008 Global Data
0000001A:016128 E7C74B3E04:001CD0 00000001 00000004
*.....*
0000001A:016232 pthread_setspecific(entry): value=80000000 00000000
e7c74b3e 04001cd0, key=0
0000001A:016304 pthread_getspecific(entry): key=0
0000001A:016384 foo(), threadSpecific data=1 4
0000001A:016456 pthread_getspecific(entry): key=0
0000001A:016528 bar(), threadSpecific data=1 4
0000001A:016584 bar(): This is an error tracepoint
0000001A:016648 Stack Dump For Current Thread
0000001A:016712 Stack: This thread's stack at time of error in bar()
0000001A:016904 Stack: Library      / Program      Module      Stmt
Procedure
0000001A:017048 Stack: QSYS          / QLESPI      QLECRTTH      774      :
LE_Create_Thread2__FP12crtth_parm_t
0000001A:017144 Stack: QSYS          / QP0WPTHRR   QP0WPTHRR     1008      :
```

pthread_trace_init_np()--Initialize or Re-initialize pthread tracing

pthread_create_part2

0000001A:017232 Stack: KULACK / PTHREADT PTHREADT 19 :

theThread__FPv

0000001A:018680 Stack: KULACK / PTHREADT PTHREADT 29 :

foo__Fv

0000001A:018760 Stack: KULACK / PTHREADT PTHREADT 46 :

bar__Fv

0000001A:018840 Stack: QSYS / QP0ZCPA QP0ZUDBG 87 :

Qp0zDumpStack

0000001A:018928 Stack: QSYS / QP0ZSCPA QP0ZSCPA 276 :

Qp0zSUDumpStack

0000001A:019000 Stack: QSYS / QP0ZSCPA QP0ZSCPA 287 :

Qp0zSUDumpTargetStack

0000001A:019040 Stack: Completed

0000001A:019136 foo(): This is an error tracepoint

0000001A:019224 pthread_create_part2(status): return from start routine,
status=00000000 00000000 00000000 00000000

0000001A:019288 pthread_cleanup(entry): Thread termination started

0000001A:019352 Qp0wTlsVector::invokeHandlers(entry):

0000001A:019424 Qp0wTlsVector::invokeHandler(invoke): key=0

0000001A:019504 dtor=a1000000

00000000 d161cc19 45001a00,

0000001A:021360 destructor name is

'dataDestructor__FPv'

0000001A:021496 arg=80000000

00000000 e7c74b3e 04001cd0

0000001A:021576 dataDestructor: Free data

0000001A:021664 pthread_setspecific(entry): value=00000000 00000000

00000000 00000000, key=0

0000001A:021752 pthread_cleanup(exit): returning

00000018:022112 pthread_join_processor(status): target status=00000000

00000000 00000000 00000000, state=0x03, YES

00000018:022272 pthread_key_delete(entry): key=0

00000018:022336 pthread_key_delete(exit): rc=0

00000018:022408 Main completed

PTHREAD_TRACE_NP()--Macro to optionally execute code based on trace level

Syntax

```
#include <pthread.h>
PTHREAD_TRACE_NP( optionalCode, desiredTraceLevel );
Threadsafe: Yes
Signal Safe: No
```

An application can use the **PTHREAD_TRACE_NP()** macro to execute optional code based on the current application trace level. The *optionalCode* to be executed can include multiple statements and can be surrounded by the C/C++ begin/end block operators (the curly brackets { }). The *optionalCode* can include pre-condition or post-condition logic, tracepoint information, or any other desired C/C++ statements.

If the current application trace level is set to a level equal to or higher than the *desiredTraceLevel*, then the code executes.

The current Pthread library trace level is set automatically when a program or service program that uses the Pthread APIs causes the Pthread APIs to be loaded (activated) or when the application explicitly calls the **pthread_trace_init_np()** function. In either case, the Pthreads library trace level is set based on the value of the **QIBM_PTHREAD_TRACE_LEVEL** environment variable at that time.

If the preprocessor value **PTHREAD_TRACE_NDEBUG** is defined, then the call to **PTHREAD_TRACE_NP()** is compiled out and does not generate any executable runtime code. Use **PTHREAD_TRACE_NDEBUG** for production level code that should not perform any tracing, or leave tracepoints in the code to assist user's of your application.

The **pthread_trace_init_np()** API initializes or refreshes both the Pthreads library trace level and the application trace level. The Pthreads library trace level is maintained internally by the Pthreads library, while the application trace level is stored in the *Qp0wTraceLevel* external variable, and can be used via the **PTHREAD_TRACE_NP()** macro.

The **PTHREAD_TRACE_NP()** macro uses the external variable *Qp0wTraceLevel*. *Qp0wTraceLevel* may be used directly by the application to set application trace level without effecting the current Pthread library trace level. Set the value of *Qp0wTraceLevel* to one of **PTHREAD_TRACE_NONE_NP**, **PTHREAD_TRACE_ERROR_NP**, **PTHREAD_TRACE_INFO_NP**, or **PTHREAD_TRACE_VERBOSE_NP**.

For consistent tracing behavior, the application should use the following table as a guide to choosing value of the *desiredTraceLevel* parameter.

Desired Trace Level	Description
PTHREAD_TRACE_NONE_NP	The <i>optionalCode</i> always runs, even when the current trace level is set to none. It is recommended that this level is only used at development time.
PTHREAD_TRACE_ERROR_NP	The <i>optionalCode</i> runs if the current trace level is set to an error level or higher. Use the error level to trace error conditions and the reasons for error return codes.
>PTHREAD_TRACE_INFO_NP	The <i>optionalCode</i> runs if the current trace level is set to an informational level or higher. Use the informational level to trace functions' entry and exit, functions' parameters and return codes and major changes in control flow.
PTHREAD_TRACE_VERBOSE_NP	The <i>optionalCode</i> runs if the current trace level is set to a verbose level or higher. Use the Verbose level traces informational level tracepoints, plus detailed information about application parameters, threads and data structures including information about Pthreads library processing information.

The **PTHREAD_TRACE_NP()** macro can be used in conjunction with the following APIs to put trace records into

the user trace flight recorder. The following system APIs defined in the qp0ztrc.h header file:

- Qp0zUprintf - print formatted trace data
- Qp0zDump - dump formatted hex data
- Qp0zDumpStack - dump the call stack of the calling thread
- Qp0zDumpTargetStack - dump the call stack of the target thread

The trace records are written to the user trace flight recorder and can be accessed via the following CL commands

- DMPUSRTRC - dump the contents of a specified job's trace
- CHGUSRTRC - change attributes (size, wrapping, clear) of a specified job's trace
- DLTUSRTRC - delete the persistent trace object associated with a job's trace

Parameters

None.

Authorities and Locks

None.

Return Value

None.

Error Conditions

None.

Related Information

- The `<pthread.h>` header file. See [Header files for Pthread functions](#).
- [pthread_trace_init_np\(\)--Initialize or Re-initialize pthread tracing](#)

Example

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <qp0ztrc.h>

#define checkResults(string, val) {
    if (val) {
        printf("Failed with %d at %s", val, string);
        exit(1);
    }
}

typedef struct {
    int          threadSpecific1;
    int          threadSpecific2;
```

```

} threadSpecific_data_t;

#define                NUMTHREADS    2
pthread_key_t          threadSpecificKey;

void foo(void);
void bar(void);
void dataDestructor(void *);

void *theThread(void *parm) {
    int                rc;
    threadSpecific_data_t *gData;
    PTHREAD_TRACE_NP({
        Qp0zUprintf("Thread Entered\n");
        Qp0zDump("Global Data", parm,
sizeof(threadSpecific_data_t));},
        PTHREAD_TRACE_INFO_NP);
    gData = (threadSpecific_data_t *)parm;
    rc = pthread_setspecific(threadSpecificKey, gData);
    checkResults("pthread_setspecific()\n", rc);
    foo();
    return NULL;
}

void foo() {
    threadSpecific_data_t *gData =
        (threadSpecific_data_t *)pthread_getspecific(threadSpecificKey);
    PTHREAD_TRACE_NP(Qp0zUprintf("foo(), threadSpecific data=%d %d\n",
        gData->threadSpecific1,
gData->threadSpecific2);,
        PTHREAD_TRACE_INFO_NP);
    bar();
    PTHREAD_TRACE_NP(Qp0zUprintf("foo(): This is an error tracepoint\n");,
        PTHREAD_TRACE_ERROR_NP);
}

void bar() {
    threadSpecific_data_t *gData =
        (threadSpecific_data_t *)pthread_getspecific(threadSpecificKey);
    PTHREAD_TRACE_NP(Qp0zUprintf("bar(), threadSpecific data=%d %d\n",
        gData->threadSpecific1,
gData->threadSpecific2);,
        PTHREAD_TRACE_INFO_NP);
    PTHREAD_TRACE_NP(Qp0zUprintf("bar(): This is an error tracepoint\n");
        Qp0zDumpStack("This thread's stack at time of error in
bar()");,
        PTHREAD_TRACE_ERROR_NP);
    return;
}

void dataDestructor(void *data) {
    PTHREAD_TRACE_NP(Qp0zUprintf("dataDestructor: Free data\n");,
        PTHREAD_TRACE_INFO_NP);
    pthread_setspecific(threadSpecificKey, NULL);    free(data);
    /* If doing verbose tracing we'll even write a message to the job log */
    PTHREAD_TRACE_NP(Qp0zLprintf("Free'd the thread specific data\n");,
        PTHREAD_TRACE_VERBOSE_NP);
}

/* Call this testcase with an optional parameter 'PTHREAD_TRACING' */
/* If the PTHREAD_TRACING parameter is specified, then the          */

```

PTHREAD_TRACE_NP)--Macro to optionally execute code based on trace level

```
/* Pthread tracing environment variable will be set, and the */
/* pthread tracing will be re initialized from its previous value. */
/* NOTE: We set the trace level to informational, tracepoints cut */
/*      using PTHREAD_TRACE_NP at a VERBOSE level will NOT show up*/
int main(int argc, char **argv) {
    pthread_t      thread[NUMTHREADS];
    int            rc=0;
    int            i;
    threadSpecific_data_t *gData;
    char           buffer[50];

    PTHREAD_TRACE_NP(Qp0zUprintf("Enter Testcase - %s\n", argv[0]);,
                     PTHREAD_TRACE_INFO_NP);
    if (argc == 2 && !strcmp("PTHREAD_TRACING", argv[1])) {
        /* Turn on internal pthread function tracing support */
        /* Or, use ADDENVVAR, CHGENVVAR CL commands to set this envvar */
        sprintf(buffer, "QIBM_PTHREAD_TRACE_LEVEL=%d", PTHREAD_TRACE_INFO_NP);
        putenv(buffer);
        /* Refresh the Pthreads internal tracing with the environment */
        /* variables value. */
        pthread_trace_init_np();
    }
    else {
        /* Trace only our application, not the Pthread code */
        Qp0wTraceLevel = PTHREAD_TRACE_INFO_NP;
    }

    rc = pthread_key_create(&threadSpecificKey, dataDestructor);
    checkResults("pthread_key_create()\n", rc);

    for (i=0; i < NUMTHREADS; ++i) {
        PTHREAD_TRACE_NP(Qp0zUprintf("Create/start a thread\n");,
                         PTHREAD_TRACE_INFO_NP);
        /* Create per-thread threadSpecific data and pass it to the thread */
        gData = (threadSpecific_data_t *)malloc(sizeof
(threadSpecific_data_t));
        gData->threadSpecific1 = i;
        gData->threadSpecific2 = (i+1)*2;
        rc = pthread_create( &thread[i], NULL, theThread, gData);
        checkResults("pthread_create()\n", rc);
        PTHREAD_TRACE_NP(Qp0zUprintf("Wait for the thread to complete, "
                                     "and release their resources\n");,
                         PTHREAD_TRACE_INFO_NP);
        rc = pthread_join(thread[i], NULL);
        checkResults("pthread_join()\n", rc);
    }

    pthread_key_delete(threadSpecificKey);
    PTHREAD_TRACE_NP(Qp0zUprintf("Main completed\n");,
                     PTHREAD_TRACE_INFO_NP);

    return 0;
}
```

Output

Use CL command **DMPUSRTRC** to output the following tracing information that the example creates. The **DMPUSRTRC** CL command causes the following information to be put into file QTEMP/QAP0ZDMP or to standard output depending on the options used for the CL command.

Note the following:

- The trace records are indented and labeled based on thread id plus a microsecond timestamp at the time the

tracepoint was cut. In the following trace record, the value *0000000D* indicates the thread ID of the thread that created the tracepoint. The value *133520* indicates that the tracepoint occurred 133520 microseconds after the last timestamp indicator.

0000000D:133520 Create/start a thread

- You can use the Pthread library tracepoints to debug incorrect calls to the Pthreads library from your application.
- The following trace output occurs when the optional parameter 'PTHREAD_TRACING' is NOT specified when calling this program. Since 'PTHREAD_TRACING' is not specified, the application directly sets the *Qp0wTraceLevel* external variable, causing only application level tracing to occur, and skipping any Pthreads library tracing.

User Trace Dump for job 096932/KULACK/PTHREADT. Size: 300K, Wrapped 0 times.

--- 11/06/1998 11:06:57 ---

0000000D:133520 Create/start a thread

0000000D:293104 Wait for the thread to complete, and release their resources

0000000E:294072 Thread Entered

0000000E:294272 DB51A4C80A:001CD0 L:0008 Global Data

0000000E:294416 DB51A4C80A:001CD0 00000000 00000002

.....

0000000E:294496 foo(), threadSpecific data=0 2

0000000E:294568 bar(), threadSpecific data=0 2

0000000E:294624 bar(): This is an error tracepoint

0000000E:294680 Stack Dump For Current Thread

0000000E:294736 Stack: This thread's stack at time of error in bar()

0000000E:333872 Stack: Library / Program Module Stmt

Procedure

0000000E:367488 Stack: QSYS / QLESPI QLECRTTH 774 :

LE_Create_Thread2__FP12crtth_parm_t

0000000E:371704 Stack: QSYS / QP0WPTHr QP0WPTHr 1008 :

pthread_create_part2

0000000E:371872 Stack: KULACK / PTHREAdT PTHREAdT 19 :

theThread__FPv

0000000E:371944 Stack: KULACK / PTHREAdT PTHREAdT 29 :

foo__Fv

0000000E:372016 Stack: KULACK / PTHREAdT PTHREAdT 46 :

bar__Fv

0000000E:372104 Stack: QSYS / QP0ZCPA QP0ZUDBG 87 :

Qp0zDumpStack

0000000E:379248 Stack: QSYS / QP0ZSCPA QP0ZSCPA 276 :

Qp0zSUDumpStack

0000000E:379400 Stack: QSYS / QP0ZSCPA QP0ZSCPA 287 :

Qp0zSUDumpTargetStack

0000000E:379440 Stack: Completed

0000000E:379560 foo(): This is an error tracepoint

0000000E:379656 dataDestructor: Free data

0000000D:413816 Create/start a thread

0000000D:414408 Wait for the thread to complete, and release their resources

0000000F:415672 Thread Entered

0000000F:415872 DB51A4C80A:001CD0 L:0008 Global Data

0000000F:416024 DB51A4C80A:001CD0 00000001 00000004

.....

0000000F:416104 foo(), threadSpecific data=1 4

0000000F:416176 bar(), threadSpecific data=1 4

0000000F:416232 bar(): This is an error tracepoint

0000000F:416288 Stack Dump For Current Thread

```

0000000F:416344 Stack:  This thread's stack at time of error in
bar()
0000000F:416552 Stack:  Library      / Program      Module      Stmt
Procedure
0000000F:416696 Stack:  QSYS        / QLESPI       QLECRTTH    774      :
LE_Create_Thread2__FP12crtth_parm_t
0000000F:416784 Stack:  QSYS        / QP0WPTHRR   QP0WPTHRR   1008     :
pthread_create_part2
0000000F:416872 Stack:  KULACK      / PTHREADT    PTHREADT    19       :
theThread__FPv
0000000F:416952 Stack:  KULACK      / PTHREADT    PTHREADT    29       :
foo__Fv
0000000F:531432 Stack:  KULACK      / PTHREADT    PTHREADT    46       :
bar__Fv
0000000F:531544 Stack:  QSYS        / QP0ZCPA     QP0ZUDBG    87       :
Qp0zDumpStack
0000000F:531632 Stack:  QSYS        / QP0ZSCPA    QP0ZSCPA    276      :
Qp0zSUDumpStack
0000000F:531704 Stack:  QSYS        / QP0ZSCPA    QP0ZSCPA    287      :
Qp0zSUDumpTargetStack
0000000F:531744 Stack:  Completed
0000000F:531856 foo(): This is an error tracepoint
0000000F:531952 dataDestructor: Free data
0000000D:532528 Main completed

```

sched_yield()--Yield Processor to Another Thread

Syntax

```
#include <sched.h>
int sched_yield(void);
Threadsafe: Yes
Signal Safe: Yes
```

The **sched_yield()** function yields the processor from the currently executing thread to another ready-to-run, active thread of equal or higher priority.

If no threads of equal or higher priority are active and ready to run, **sched_yield()** returns immediately, and the calling thread continues to run until its time has expired.

Parameters

None.

Authorities and Locks

None.

Return Value

0

sched_yield() was successful.

value

sched_yield() was not successful. *value* is set to indicate the error condition.

Error Conditions

The **sched_yield()** API does not currently return an error.

Related Information

- The **<pthread.h>** header file. See [Header files for Pthread functions](#).
- [pthread_getschedparam\(\)--Get Thread Scheduling Parameters](#)

Example

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include <errno.h>
#include "check.h"

#define LOOPCONSTANT 1000
#define THREADS 3

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int i, j, k, l;
```

```

void *threadfunc(void *parm)
{
    int    loop = 0;
    int    localProcessingCompleted = 0;
    int    numberOfLocalProcessingBursts = 0;
    int    processingCompletedThisBurst = 0;
    int    rc;

    printf("Entered secondary thread\n");
    for (loop=0; loop<LOOPCONSTANT; ++loop) {
        rc = pthread_mutex_lock(&mutex);
        checkResults("pthread_mutex_lock()\n", rc);
        /* Perform some not so important processing */
        i++, j++, k++, l++;

        rc = pthread_mutex_unlock(&mutex);
        checkResults("pthread_mutex_unlock()\n", rc);
        /* This work is not too important. Also, we just released a lock
           and would like to ensure that other threads get a chance in
           a more co-operative manner. This is an admittedly contrived
           example with no real purpose for doing the sched_yield().
        */
        sched_yield();
    }
    printf("Finished secondary thread\n");
    return NULL;
}

int main(int argc, char **argv)
{
    pthread_t      threadid[THREADS];
    int            rc=0;
    int            loop=0;

    printf("Enter Testcase - %s\n", argv[0]);

    rc = pthread_mutex_lock(&mutex);
    checkResults("pthread_mutex_lock()\n", rc);

    printf("Creating %d threads\n", THREADS);
    for (loop=0; loop<THREADS; ++loop) {
        rc = pthread_create(&threadid[loop], NULL, threadfunc, NULL);
        checkResults("pthread_create()\n", rc);
    }

    sleep(1);
    rc = pthread_mutex_unlock(&mutex);
    checkResults("pthread_mutex_unlock()\n", rc);

    printf("Wait for results\n");
    for (loop=0; loop<THREADS; ++loop) {
        rc = pthread_join(threadid[loop], NULL);
        checkResults("pthread_join()\n", rc);
    }

    pthread_mutex_destroy(&mutex);

    printf("Main completed\n");
    return 0;
}

```


Output:

```
Enter Testcase - QP0WTEST/TPSCHY0
Creating 3 threads
Entered secondary thread
Entered secondary thread

Entered secondary thread
Wait for results
Finished secondary thread
Finished secondary thread
Finished secondary thread
Main completed
```

Thread specific storage APIs

Thread specific storage is used by your threaded application when you need global storage that is 'private' to a thread. The storage is allocated and stored by the thread, and can be associated with a destructor function. When the thread ends using one of the pthread mechanisms, the destructor function runs and cleans up the thread local storage. The thread specific storage can replace global storage, because any function in a thread that requests the thread specific storage will get the same value. Functions in another thread that request the thread specific storage will get the thread specific storage owned by the thread that they are called in. For information about the examples included with the APIs, see the [information on the API examples](#).

The thread specific storage APIs are:

- [pthread_getspecific\(\)--Get Thread Local Storage Value by Key](#)
- [pthread_key_create\(\)--Create Thread Local Storage Key](#)
- [pthread_key_delete\(\)--Delete Thread Local Storage Key](#)
- [pthread_setspecific\(\)--Set Thread Local Storage by Key](#)

pthread_getspecific()--Get Thread Local Storage Value by Key

Syntax

```
#include <pthread.h>
void *pthread_getspecific(pthread_key_t key);
Threadsafe: Yes
Signal Safe: Yes
```

The **pthread_getspecific()** function retrieves the thread local storage value associated with the *key*. **pthread_getspecific()** may be called from a data destructor.

The thread local storage value is a variable of type `void *` that is local to a thread, but global to all of the functions called within that thread. It is accessed via the key.

Parameters

key

(Input) The thread local storage key returned from **pthread_key_create()**

Authorities and Locks

None.

Return Value

value

pthread_getspecific() was successful. *value* is set to indicate the current thread specific data pointer stored at the *key* location.

NULL

pthread_getspecific() returned the null thread specific data value stored at the *key* location or the *key* was out of range.

Error Conditions

None.

Related Information

- The `<pthread.h>` header file. See [Header files for Pthread functions](#).
- [pthread_key_create\(\)--Create Thread Local Storage Key](#)
- [pthread_key_delete\(\)--Delete Thread Local Storage Key](#)
- [pthread_setspecific\(\)--Set Thread Local Storage by Key](#)

Example

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include "check.h"
```

```

#define          NUMTHREADS      3
pthread_key_t    tlsKey = 0;

void globalDestructor(void *value)
{
    printf("In the globalDestructor\n");
    free(value);
    pthread_setspecific(tlsKey, NULL);
}

void showGlobal(void)
{
    void                *global;
    pthread_id_np_t      tid;

    global = pthread_getspecific(tlsKey);
    pthread_getunique_np((pthread_t *)global, &tid);
    printf("showGlobal: global data stored for thread 0x%.8x%.8x\n",
           tid);
}

void *threadfunc(void *parm)
{
    int                rc;
    int                *myThreadDataStructure;
    pthread_t          me = pthread_self();

    printf("Inside secondary thread\n");

    myThreadDataStructure = malloc(sizeof(pthread_t) + sizeof(int) * 10);
    memcpy(myThreadDataStructure, &me, sizeof(pthread_t));
    pthread_setspecific(tlsKey, myThreadDataStructure);
    showGlobal();
    pthread_exit(NULL);
}

int main(int argc, char **argv)
{
    pthread_t          thread[NUMTHREADS];
    int                rc=0;
    int                i=0;

    printf("Enter Testcase - %s\n", argv[0]);

    printf("Create a thread local storage key\n");
    rc = pthread_key_create(&tlsKey, globalDestructor);
    checkResults("pthread_key_create()\n", rc);
    /* The key can now be used from all threads */

    printf("Create %d threads using joinable attributes\n",
           NUMTHREADS);
    for (i=0; i<NUMTHREADS; ++i) {
        rc = pthread_create(&thread[i], NULL, threadfunc, NULL);
        checkResults("pthread_create()\n", rc);
    }

    printf("Join to threads\n");
    for (i=0; i<NUMTHREADS; ++i) {
        rc = pthread_join(thread[i], NULL);
    }
}

```

pthread_getspecific()--Get Thread Local Storage Value by Key

```
    checkResults("pthread_join()\n", rc);
}

printf("Delete a thread local storage key\n");
rc = pthread_key_delete(tlsKey);
checkResults("pthread_key_delete()\n", rc);
/* The key and any remaining values are now gone. */
printf("Main completed\n");
return 0;
}
```

Output:

```
Enter Testcase - QP0WTEST/TPGETS0
Create a thread local storage key
Create 3 threads using joinable attributes
Join to threads
Inside secondary thread
showGlobal: global data stored for thread 0x000000000000000b
In the globalDestructor
Inside secondary thread
showGlobal: global data stored for thread 0x000000000000000d
In the globalDestructor
Inside secondary thread
showGlobal: global data stored for thread 0x000000000000000c
In the globalDestructor
Delete a thread local storage key
Main completed
```

pthread_key_create()--Create Thread Local Storage Key

Syntax

```
#include <pthread.h>
int pthread_key_create(pthread_key_t *key, void (*destructor)(void *));
Threadsafe: Yes
Signal Safe: No
```

The **pthread_key_create()** function creates a thread local storage *key* for the process and associates the *destructor* function with that *key*. After a key is created, that key can be used to set and get per-thread data pointer. When **pthread_key_create()** completes, the value associated with the newly created key is NULL.

When a thread terminates, if **both** the value and the destructor associated with a thread local storage key are not **NULL**, the destructor function is called. The stored pointer associated with the key is set to NULL before the call to the destructor function. The parameter passed to the destructor function when it is called is the value of the pointer before it was set to NULL that is associated with that key in the thread that is terminating.

After calling the destructors, if there are still non **NULL** values in the thread associated with the keys, the process is repeated. After **PTHREAD_DESTRUCTOR_ITERATIONS** attempts to destroy the thread local storage, no further attempts are made for that thread local storage value/key combination.

Do not call **pthread_exit()** from a destructor function.

A destructor function is not called as a result of the application calling **pthread_key_delete()**.

Parameters

key

(Output) The address of the variable to contain the thread local storage key

destructor

(Input) The address of the function to act as a destructor for this thread local storage key

Authorities and Locks

None.

Return Value

0

pthread_key_create() was successful.

value

pthread_key_create() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_key_create()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

[EAGAIN]

pthread_key_create()--Create Thread Local Storage Key

The system did not have enough resources, or the maximum of **PTHREAD_KEYS_MAX** would have been exceeded.

[ENOMEM]

Not enough memory to create the key.

Related Information

- The `<pthread.h>` header file. See [Header files for Pthread functions](#).
- [pthread_getspecific\(\)--Get Thread Local Storage Value by Key](#)
- [pthread_key_delete\(\)--Delete Thread Local Storage Key](#)
- [pthread_setspecific\(\)--Set Thread Local Storage by Key](#)

Example

```
#define _MULTI_THREADED
#include <pthread.h>
#include <sched.h>
#include <stdio.h>
#include "check.h"

pthread_key_t    tlsKey = 0;

void globalDestructor(void *value)
{
    printf("In the data destructor\n");
    free(value);
    pthread_setspecific(tlsKey, NULL);
}

int main(int argc, char **argv)
{
    int                rc=0;
    int                i=0;

    printf("Enter Testcase - %s\n", argv[0]);

    printf("Create a thread local storage key\n");
    rc = pthread_key_create(&tlsKey, globalDestructor);
    checkResults("pthread_key_create()\n", rc);
    /* The key can now be used from all threads */

    printf("- The key can now be used from all threads\n");
    printf("- in the process to storage thread local\n");
    printf("- (but global to all functions in that thread)\n");
    printf("- storage\n");

    printf("Delete a thread local storage key\n");
    rc = pthread_key_delete(tlsKey);
    checkResults("pthread_key_delete()\n", rc);
    /* The key and any remaining values are now gone. */
    printf("Main completed\n");
    return 0;
}
```

Output:

```
Enter Testcase - QP0WTEST/TPKEYC0
Create a thread local storage key
- The key can now be used from all threads
- in the process to storage thread local
- (but global to all functions in that thread)
- storage
Delete a thread local storage key
Main completed
```


pthread_key_delete()--Delete Thread Local Storage Key

Syntax

```
#include <pthread.h>
int pthread_key_delete(pthread_key_t key);
Threadsafe: Yes
Signal Safe: No
```

The **pthread_key_delete()** function deletes a process-wide thread local storage *key*. The **pthread_key_delete()** function does not run any destructors for the values associated with *key* in any threads. After a key is deleted, it may be returned by a subsequent call to **pthread_key_create()**.

An attempt to delete a key that is out of range or not valid fails with EINVAL. An attempt to delete a valid key that has already been deleted or has not been returned from **pthread_key_create()** fails with ENOENT.

Parameters

key

(Input) The thread local storage key returned from **pthread_key_create()**

Authorities and Locks

None.

Return Value

0

pthread_key_delete() was successful.

value

pthread_key_delete() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_key_delete()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

A destructor function is not called as a result of the application calling **pthread_key_delete()**.

[EINVAL]

The value specified for the argument is not correct.

[ENOENT]

An entry for the key is not currently allocated.

Related Information

- The **<pthread.h>** header file. See [Header files for Pthread functions](#).
- [pthread_getspecific\(\)--Get Thread Local Storage Value by Key](#)
- [pthread_key_create\(\)--Create Thread Local Storage Key](#)
- [pthread_setspecific\(\)--Set Thread Local Storage by Key](#)

Example

```

#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include "check.h"

pthread_key_t    tlsKey = 0;

void globalDestructor(void *value)
{
    printf("In global data destructor\n");
    free(value);
    pthread_setspecific(tlsKey, NULL);
}

int main(int argc, char **argv)
{
    int                rc=0;
    int                i=0;

    printf("Enter Testcase - %s\n", argv[0]);

    printf("Create a thread local storage key\n");
    rc = pthread_key_create(&tlsKey, globalDestructor);
    checkResults("pthread_key_create()\n", rc);
    /* The key can now be used from all threads */

    printf("Delete a thread local storage key\n");
    rc = pthread_key_delete(tlsKey);
    checkResults("pthread_key_delete()\n", rc);

    printf("- The key should not be used from any thread\n");
    printf("- after destruction.\n");
    /* The key and any remaining values are now gone. */
    printf("Main completed\n");
    return 0;
}

```

Output:

```

Enter Testcase - QP0WTEST/TPKEYD0
Create a thread local storage key
Delete a thread local storage key
- The key should not be used from any thread
- after destruction.
Main completed

```

pthread_setspecific()--Set Thread Local Storage by Key

Syntax

```
#include <pthread.h>
int pthread_setspecific(pthread_key_t key, const void *value);
Threadsafe: Yes
Signal Safe: Yes
```

The **pthread_setspecific()** function sets the thread local storage *value* associated with a *key*. The **pthread_setspecific()** function may be called from within a data destructor.

The thread local storage value is a variable of type `void *` that is local to a thread, but global to all of the functions called within that thread. It is accessed via the key.

Parameters

key

(Input) The thread local storage key returned from **pthread_key_create()**.

value

(Input) The pointer to store at the *key* location for the calling thread.

Authorities and Locks

None.

Return Value

0

pthread_setspecific() was successful.

value

pthread_setspecific() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_setspecific()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the key is not correct.

Related Information

- The `<pthread.h>` header file. See [Header files for Pthread functions](#).
- [pthread_getspecific\(\)--Get Thread Local Storage Value by Key](#)
- [pthread_key_create\(\)--Create Thread Local Storage Key](#)
- [pthread_key_delete\(\)--Delete Thread Local Storage Key](#)

Example

```

#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include "check.h"

#define NUMTHREADS 3
pthread_key_t  tlsKey = 0;

void globalDestructor(void *value)
{
    printf("In global destructor\n");
    free(value);
    pthread_setspecific(tlsKey, NULL);
}

void showGlobal(void)
{
    void                *global;
    pthread_id_np_t     tid;

    global = pthread_getspecific(tlsKey);
    pthread_getunique_np((pthread_t *)global, &tid);
    printf("showGlobal: global data stored for thread 0x%.8x %.8x\n",
           tid);
}

void *threadfunc(void *parm)
{
    int                rc;
    int                *myThreadDataStructure;
    pthread_t          me = pthread_self();

    printf("Inside secondary thread\n");

    myThreadDataStructure = malloc(sizeof(pthread_t) + sizeof(int) * 10);
    memcpy(myThreadDataStructure, &me, sizeof(pthread_t));
    pthread_setspecific(tlsKey, myThreadDataStructure);
    showGlobal();
    return NULL;
}

int main(int argc, char **argv)
{
    pthread_t          thread[NUMTHREADS];
    int                rc=0;
    int                i=0;

    printf("Enter Testcase - %s\n", argv[0]);

    printf("Create a thread local storage key\n");
    rc = pthread_key_create(&tlsKey, globalDestructor);
    checkResults("pthread_key_create()\n", rc);
    /* The key can now be used from all threads */

    printf("Create %d threads using joinable attributes\n",
           NUMTHREADS);
    for (i=0; i<NUMTHREADS; ++i) {

```

pthread_setspecific()--Set Thread Local Storage by Key

```
    rc = pthread_create(&thread[i], NULL, threadfunc, NULL);
    checkResults("pthread_create()\n", rc);
}

printf("Join to threads\n");
for (i=0; i<NUMTHREADS; ++i) {
    rc = pthread_join(thread[i], NULL);
    checkResults("pthread_join()\n", rc);
}

printf("Delete a thread local storage key\n");
rc = pthread_key_delete(tlsKey);
checkResults("pthread_key_delete()\n", rc);
/* The key and any remaining values are now gone. */
printf("Main completed\n");
return 0;
}
```

Output:

```
Enter Testcase - QP0WTEST/TPSETS0
Create a thread local storage key
Create 3 threads using joinable attributes
Join to threads
Inside secondary thread
showGlobal: global data stored for thread 0x00000000 0000011a
In global destructor
Inside secondary thread
showGlobal: global data stored for thread 0x00000000 0000011b
In global destructor
Inside secondary thread
showGlobal: global data stored for thread 0x00000000 0000011c
In global destructor
Delete a thread local storage key
Main completed
```

Thread cancellation APIs

You can use thread cancellation APIs to cause a thread to end prematurely, or to aid in cleanup when a thread is ended (either prematurely or normally). The thread cancellation APIs work together to provide a mechanism for thread cleanup and protecting threaded resources from cancellation. The thread cancellation APIs only provide clean up and protection in relationship to other pthread APIs. You cannot protect from or clean up when a thread ends as a result of your process ending (normally or abnormally), or when the thread ends via some mechanism outside of the pthread API set. Some examples of mechanisms that can terminate a thread that are outside of the pthread API set are the ENDJOB *IMMED CL command, a thread ending from an unhandled exception, or the operator terminating a thread using the work with threads screen (Option 20 from the WRKJOB display).

The table below lists the thread cancelability states, the cancellation types, and the cancellation action. Cancelability consists of three separate states (disabled, deferred, asynchronous) that can be represented by two boolean values. The default cancelability state is deferred.

Cancelability	Cancelability State	Cancelability Type
disabled	PTHREAD_CANCEL_DISABLE	PTHREAD_CANCEL_DEFERRED
disabled	PTHREAD_CANCEL_DISABLE	PTHREAD_CANCEL_ASYNCHRONOUS
deferred	PTHREAD_CANCEL_ENABLE	PTHREAD_CANCEL_DEFERRED
asynchronous	PTHREAD_CANCEL_ENABLE	PTHREAD_CANCEL_ASYNCHRONOUS

For information about the examples included with the APIs, see the [information on the API examples](#).

The thread cancellation APIs are:

- [pthread_cancel\(\)--Cancel Thread](#)
- [pthread_cleanup_peek_np\(\)--Copy Cleanup Handler from Cancellation Cleanup Stack](#)
- [pthread_cleanup_pop\(\)--Pop Cleanup Handler off of Cancellation Cleanup Stack](#)
- [pthread_cleanup_push\(\)--Push Cleanup Handler onto Cancellation Cleanup Stack](#)
- [pthread_getcancelstate_np\(\)--Get Cancel State](#)
- [pthread_setcancelstate\(\)--Set Cancel State](#)
- [pthread_setcanceltype\(\)--Set Cancel Type](#)
- [pthread_testcancel\(\)--Create Cancellation Point](#)
- [pthread_test_exit_np\(\)--Test Thread Exit Status](#)

pthread_cancel()--Cancel Thread

Syntax

```
#include <pthread.h>
int pthread_cancel(pthread_t thread);
Threadsafe: Yes
Signal Safe: No
```

The **pthread_cancel()** function requests cancellation of the target thread. The target thread is canceled, based on its ability to be canceled.

When cancelability is disabled, all cancels are held pending in the target thread until the thread changes the cancelability. When cancelability is deferred, all cancels are held pending in the target thread until the thread changes the cancelability, calls a function that is a cancellation point, or calls **pthread_testcancel()**, thus creating a cancellation point. When cancelability is asynchronous, all cancels are acted upon immediately, interrupting the thread with its processing.

*You should not use asynchronous thread cancellation via the **PTHREAD_CANCEL_ASYNCHRONOUS** option of **pthread_setcanceltype()** in your application. See the common user errors section of this document for more information.*

The following functions are cancellation points:

- **pthread_cond_timedwait()**
- **pthread_cond_wait()**
- **pthread_join()**
- **pthread_join_np()**
- **pthread_extendedjoin_np()**
- **pthread_testcancel()**

After action is taken for the target thread to be canceled, the following events occur in that thread.

1. The thread invokes cancellation cleanup handlers with cancellation disabled until the last cancellation cleanup handler returns. The handlers are invoked in Last In, First Out (LIFO) order.
2. Data destructors are called for any thread-specific data entries that have a non NULL value for both the value and the destructor.
3. When the last cancellation cleanup handler returns, the thread is terminated and a status of **PTHREAD_CANCELED** is made available to any threads joining the target.
4. Any mutexes that are held by a thread that terminates, are abandoned and are no longer valid. Subsequent calls by other threads that attempt to acquire the abandoned mutex (**pthread_mutex_lock()** or **pthread_mutex_trylock()**) fail with an **EOWNERDEAD** error.
5. Application visible process resources are not released. This includes but is not limited to mutexes, file descriptors, or any process level cleanup actions.

A cancellation cleanup handler should not exit via **longjmp()** or **siglongjmp()**.

In the OS/400 implementation of threads, the initial thread is special. Termination of the initial thread via **pthread_exit()**, **pthread_cancel()** or any other thread termination mechanism terminates the entire process.

Parameters

thread

(Input) Pthread handle to the target thread

Authorities and Locks

None.

Return Value

0

pthread_cancel() was successful.

value

pthread_cancel() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_cancel()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

[ESRCH]

No thread could be found that matched the thread ID specified.

Related Information

- The `<pthread.h>` header file. See [Header files for Pthread functions](#).
- [pthread_cleanup_pop\(\)--Pop Cleanup Handler off of Cancellation Cleanup Stack](#)
- [pthread_cleanup_push\(\)--Push Cleanup Handler onto Cancellation Cleanup Stack](#)
- [pthread_exit\(\)--Terminate Calling Thread](#)
- [pthread_setcancelstate\(\)--Set Cancel State](#)
- [pthread_setcanceltype\(\)--Set Cancel Type](#)

Example

```
#include <pthread.h>
#include <stdio.h>
#include "check.h"

void *threadfunc(void *parm)
{
    printf("Entered secondary thread\n");
    while (1) {

        printf("Secondary thread is looping\n");
        pthread_testcancel();
        sleep(1);
    }
    return NULL;
}

int main(int argc, char **argv)
{
    pthread_t          thread;
```



```
int rc=0;

printf("Entering testcase\n");

/* Create a thread using default attributes */
printf("Create thread using the NULL attributes\n");
rc = pthread_create(&thread, NULL, threadfunc, NULL);
checkResults("pthread_create(NULL)\n", rc);

/* sleep() is not a very robust way to wait for the thread */
sleep(2);

printf("Cancel the thread\n");
rc = pthread_cancel(thread);
checkResults("pthread_cancel()\n", rc);

/* sleep() is not a very robust way to wait for the thread */
sleep(3);
printf("Main completed\n");
return 0;
}
```

Output:

```
Entering testcase
Create thread using the NULL attributes
Entered secondary thread
Secondary thread is looping
Secondary thread is looping
Cancel the thread
Main completed
```

pthread_cleanup_peek_np()--Copy Cleanup Handler from Cancellation Cleanup Stack

Syntax

```
#include <pthread.h>
void pthread_cleanup_peek_np(pthread_cleanup_entry_np_t *entry);
Threadsafe: Yes
Signal Safe: Yes
```

The **pthread_cleanup_peek_np()** function returns a copy of the cleanup handler entry that the next call to **pthread_cleanup_pop()** would pop. The handler remains on the cancellation cleanup stack after the call to **pthread_cleanup_peek_np()**.

During this thread cancellation cleanup, the thread invokes cancellation cleanup handlers with cancellation disabled until the last cancellation cleanup handler returns. The handlers are invoked in Last In, First Out (LIFO) order. Automatic storage for the invocation stack frame of the function that registered the handler is still present when the cancellation cleanup handler is executed.

The **pthread_cleanup_push()** and the matching **pthread_cleanup_pop()** call should be in the same lexical scope (i.e., same level of brackets {}).

The **pthread_cleanup_peek_np()** function has no scoping rules.

This function is not portable.

Parameters

None.

Authorities and Locks

None.

Return Value

0

pthread_cleanup_peek_np() was successful.

value

pthread_cleanup_peek_np() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_cleanup_peek_np()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

[ENOENT]

The cancellation cleanup stack is empty.

Related Information

- The `<pthread.h>` header file. See [Header files for Pthread functions](#).
- [pthread_cleanup_pop\(\)--Pop Cleanup Handler off of Cancellation Cleanup Stack](#)
- [pthread_cleanup_push\(\)--Push Cleanup Handler onto Cancellation Cleanup Stack](#)
- [pthread_exit\(\)--Terminate Calling Thread](#)

Example

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include "check.h"

void cleanupHandler1(void *arg) { printf("In Handler 1\n"); return; }
void cleanupHandler2(void *arg) { printf("In Handler 2\n"); return; }
void cleanupHandler3(void *arg) { printf("In Handler 3\n"); return; }
int
    args[3] = {0,0,0};

int main(int argc, char **argv)
{
    int                rc=0;
    pthread_cleanup_entry_np_t    entry;

    printf("Enter Testcase - %s\n", argv[0]);

    printf("Check for absence of cleanup handlers\n");
    rc = pthread_cleanup_peek_np(&entry);
    if (rc != ENOENT) {
        printf("pthread_cleanup_peek_np(), expected ENOENT\n");
        exit(1);
    }

    printf("Push some cancellation cleanup handlers\n");
    pthread_cleanup_push(cleanupHandler1, &args[0]);
    pthread_cleanup_push(cleanupHandler2, &args[1]);

    printf("Check for cleanupHandler2\n");
    rc = pthread_cleanup_peek_np(&entry);
    checkResults("pthread_cleanup_peek_np(2)\n", rc);
    if (entry.handler != cleanupHandler2 ||
        entry.arg != &args[1]) {
        printf("Did not get expected handler(2) information!\n");
        exit(1);
    }

    pthread_cleanup_push(cleanupHandler3, &args[2]);

    printf("Check for cleanupHandler3\n");
    rc = pthread_cleanup_peek_np(&entry);
    checkResults("pthread_cleanup_peek_np(3)\n", rc);
    if (entry.handler != cleanupHandler3 ||
        entry.arg != &args[2]) {
        printf("Did not get expected handler(3) information!\n");
        exit(1);
    }
}
```

pthread_cleanup_peek_np()--Copy Cleanup Handler from Cancellation Cleanup Stack

```
pthread_cleanup_pop(0);  
pthread_cleanup_pop(0);  
pthread_cleanup_pop(0);  
  
printf("Main completed\n");  
return 0;  
}
```

Output:

```
Enter Testcase - QP0WTEST/TPCLPP0  
Check for absence of cleanup handlers  
Push some cancellation cleanup handlers  
Check for cleanupHandler2  
Check for cleanupHandler3  
Main completed
```

pthread_cleanup_pop()--Pop Cleanup Handler off of Cancellation Cleanup Stack

Syntax

```
#include <pthread.h>
void pthread_cleanup_pop(int execute);
Threadsafe: Yes
Signal Safe: No
```

The **pthread_cleanup_pop()** function pops the last cleanup handler from the cancellation cleanup stack. If the *execute* parameter is non-zero, the handler is invoked with the argument specified via the **pthread_cleanup_push()** call that the handler was registered with.

The **pthread_cleanup_push()** and the matching **pthread_cleanup_pop()** call should be in the same lexical scope (i.e., same level of brackets {}).

When the thread calls **pthread_exit()** or is canceled via **pthread_cancel()**, the cancellation cleanup handlers are invoked with the argument specified via the **pthread_cleanup_push()** call that the handler was registered with.

During this thread cancellation cleanup, the thread invokes cancellation cleanup handlers with cancellation disabled until the last cancellation cleanup handler returns. The handlers are invoked in Last In, First Out (LIFO) order. Automatic storage for the invocation stack frame of the function that registered the handler is still present when the cancellation cleanup handler is executed.

When a cancellation cleanup handler is invoked because of a call to **pthread_cleanup_pop(1)**, the cancellation cleanup handler does not necessarily run with cancellation disabled. The cancellation state and cancellation type are not changed by a call to **pthread_cleanup_pop(1)**.

A cancellation cleanup handler should not exit via **longjmp()** or **siglongjmp()**. If a cleanup handler takes an exception, the exception condition is handled and ignored and processing continues. You can look in the job log of the job to see exception messages generated by cancellation cleanup handlers.

Parameters

execute

(Input) Boolean value indicating whether the cancellation cleanup handler should be executed

Authorities and Locks

None.

Return Value

None.

Related Information

- The **<pthread.h>** header file. See [Header files for Pthread functions](#).
- [pthread_cancel\(\)--Cancel Thread](#)
- [pthread_cleanup_push\(\)--Push Cleanup Handler onto Cancellation Cleanup Stack](#)
- [pthread_exit\(\)--Terminate Calling Thread](#)

Example

```

#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include "check.h"

void cleanupHandler(void *arg)
{
    printf("In the cleanup handler\n");
}

void *threadfunc(void *parm)
{
    printf("Entered secondary thread, you should see the cleanup handler\n");
    pthread_cleanup_push(cleanupHandler, NULL);
    sleep(1); /* Simulate more code here */
    pthread_cleanup_pop(1);
    return NULL;
}

int main(int argc, char **argv)
{
    pthread_t      thread;
    int            rc=0;

    printf("Enter Testcase - %s\n", argv[0]);

    /* Create a thread using default attributes */
    printf("Create thread using the NULL attributes\n");

    rc = pthread_create(&thread, NULL, threadfunc, NULL);
    checkResults("pthread_create(NULL)\n", rc);

    /* sleep() is not a very robust way to wait for the thread */
    sleep(5);
    printf("Main completed\n");
    return 0;
}

```

Output:

```

Enter Testcase - QP0WTEST/TPCLP00
Create thread using the NULL attributes
Entered secondary thread, you should see the cleanup handler
In the cleanup handler
Main completed

```

pthread_cleanup_push()--Push Cleanup Handler onto Cancellation Cleanup Stack

Syntax

```
#include <pthread.h>
void pthread_cleanup_push(void (*routine)(void *), void *arg);
Threadsafe: Yes
Signal Safe: No
```

The **pthread_cleanup_push()** function pushes a cancellation cleanup routine onto the calling threads cancellation cleanup stack. When the thread calls **pthread_exit()** or is canceled via **pthread_cancel()**, the cancellation cleanup handlers are invoked with the argument *arg*.

The cancellation cleanup handlers are also invoked when they are removed from the cancellation cleanup stack via a call to **pthread_cleanup_pop()** and a non-zero *execute* argument is specified.

The **pthread_cleanup_push()** and the matching **pthread_cleanup_pop()** call should be in the same lexical scope (i.e., same level of brackets {}).

When the thread calls **pthread_exit()** or is canceled via **pthread_cancel()**, the cancellation cleanup handlers are invoked with the argument specified via the **pthread_cleanup_push()** call that the handler was registered with.

During this thread cancellation cleanup processing, the thread invokes cancellation cleanup handlers with cancellation disabled until the last cancellation cleanup handler returns. The handlers are invoked in Last In, First Out (LIFO) order. Automatic storage for the invocation stack frame of the function that registered the handler are still present when the cancellation cleanup handler is executed.

When a cancellation cleanup handler is invoked because of a call to **pthread_cleanup_pop(1)**, the cancellation cleanup handler does not necessarily run with cancellation disabled. The cancellation state and cancellation type are not changed by a call to **pthread_cleanup_pop(1)**.

A cancellation cleanup handler should not exit via **longjmp()** or **siglongjmp()**. If a cleanup handler takes an exception, the exception condition is handled and ignored and processing continues. You can look in the job log of the job to see exception messages generated by cancellation cleanup handlers.

Parameters

routine

(Input) The cancellation cleanup routine

arg

(Input) Argument that is passed to the start routine if it is invoked

Authorities and Locks

None.

Return Value

None.

Related Information

- The `<pthread.h>` header file. See [Header files for Pthread functions](#).
- [pthread_cancel\(\)--Cancel Thread](#)
- [pthread_cleanup_pop\(\)--Pop Cleanup Handler off of Cancellation Cleanup Stack](#)
- [pthread_exit\(\)--Terminate Calling Thread](#)

Example

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include "check.h"

void cleanupHandler(void *arg)
{
    printf("In the cleanup handler\n");
}

void *threadfunc(void *parm)
{
    printf("Entered secondary thread\n");
    pthread_cleanup_push(cleanupHandler, NULL);
    while (1) {
        pthread_testcancel();
        sleep(1);
    }
    pthread_cleanup_pop(0);
    return NULL;
}

int main(int argc, char **argv)
{
    pthread_t      thread;
    int            rc=0;

    printf("Enter Testcase - %s\n", argv[0]);

    /* Create a thread using default attributes */
    printf("Create thread using the NULL attributes\n");
    rc = pthread_create(&thread, NULL, threadfunc, NULL);
    checkResults("pthread_create(NULL)\n", rc);

    /* sleep() is not a very robust way to wait for the thread */
    sleep(2);

    printf("Cancel the thread\n");
    rc = pthread_cancel(thread);
    checkResults("pthread_cancel()\n", rc);

    /* sleep() is not a very robust way to wait for the thread */
    sleep(3);
    printf("Main completed\n");
    return 0;
}
```

Output:

pthread_cleanup_push()--Push Cleanup Handler onto Cancellation Cleanup Stack

```
Enter Testcase - QP0WTEST/TPCLPU0
Create thread using the NULL attributes
Entered secondary thread
Cancel the thread
In the cleanup handler
Main completed
```

pthread_getcancelstate_np()--Get Cancel State

Syntax

```
#include <pthread.h>
int pthread_getcancelstate_np(int *cancelState);
Threadsafe: Yes
Signal Safe: Yes
```

The **pthread_getcancelstate_np()** function gets the current cancel *state* of the thread. Cancel state is either PTHREAD_CANCEL_ENABLE or PTHREAD_CANCEL_DISABLE. For more information on cancelability, see [Thread cancellation APIs](#).

When cancelability is disabled, all cancels are held pending in the target thread until the thread changes the cancelability. When cancelability is deferred, all cancels are held pending in the target thread until the thread changes the cancelability, calls a function that is a cancellation point, or calls **pthread_testcancel()**, thus creating a cancellation point. When cancelability is asynchronous, all cancels are acted upon immediately, interrupting the thread with its processing.

*Your application should not use asynchronous thread cancellation via the **PTHREAD_CANCEL_ASYNCHRONOUS** option of **pthread_setcanceltype()**. See the common user errors section of this document for more information.*

This function is not portable,

Parameters

cancelstate

(Output) Address of the variable to receive the cancel state.

Authorities and Locks

None.

Return Value

0

pthread_getcancelstate_np() was successful.

value

pthread_getcancelstate_np() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_getcancelstate_np()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

Related Information

- The <pthread.h> header file. See [Header files for Pthread functions](#).
- [pthread_cancel\(\)--Cancel Thread](#)

pthread_getcancelstate_np()--Get Cancel State

- [pthread_exit\(\)--Terminate Calling Thread](#)
- [pthread_setcancelstate\(\)--Set Cancel State](#)
- [pthread_setcanceltype\(\)--Set Cancel Type](#)
- [pthread_testcancel\(\)--Create Cancellation Point](#)

Example

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include <except.h>
#include <setjmp.h>
#include "check.h"

void showCancelState(void);
int threadStatus=42;

void showCancelState(void)
{
    int state, rc;

    rc = pthread_getcancelstate_np(&state);
    checkResults("pthread_getcancelstate_np()\n", rc);
    printf("current cancel state is %d\n", state);
}

void cleanupHandler2(void *p)
{
    printf("In cancellation cleanup handler\n");
    showCancelState();
    return;
}

void *threadfunc(void *parm)
{
    int rc, old;

    printf("Inside secondary thread\n");
    showCancelState();

    pthread_cleanup_push(cleanupHandler2, NULL);
    threadStatus = 0;
    printf("Calling pthread_exit() will allow cancellation "
           "cleanup handlers to run\n");
    pthread_exit(__VOID(threadStatus));
    pthread_cleanup_pop(0);
    return __VOID(-1);
}

int main(int argc, char **argv)
{
    pthread_t thread;
    int rc=0;
    char c;
    void *status;

    printf("Enter Testcase - %s\n", argv[0]);
```

pthread_getcancelstate_np()--Get Cancel State

```
    printf("Create thread that will demonstrate\n");
pthread_getcancelstate_np();
rc = pthread_create(&thread, NULL, threadfunc, NULL);
checkResults("pthread_create()\n", rc);

rc = pthread_join(thread, &status);
checkResults("pthread_join()\n", rc);

if (__INT(status) != threadStatus) {
    printf("Got an unexpected return status from the thread!\n");
    exit(1);
}
printf("Main completed\n");
return 0;
}
```

Output:

```
Enter Testcase - QP0WTEST/TPGETCANS0
Create thread that will demonstrate pthread_getcancelstate_np()
Inside secondary thread
current cancel state is 0
Calling pthread_exit() will allow cancellation cleanup handlers to run
In cancellation cleanup handler
current cancel state is 1
Main completed
```

pthread_setcancelstate()--Set Cancel State

Syntax

```
#include <pthread.h>
int pthread_setcancelstate(int state, int *oldstate);
Threadsafe: Yes
Signal Safe: Yes
```

The **pthread_setcancelstate()** function sets the cancel *state* to one of **PTHREAD_CANCEL_ENABLE** or **PTHREAD_CANCEL_DISABLE** and returns the old cancel state into the location specified by *oldstate* (if *oldstate* is non-**NULL**).

When cancelability is disabled, all cancels are held pending in the target thread until the thread changes the cancelability. When cancelability is deferred, all cancels are held pending in the target thread until the thread changes the cancelability, calls a function which is a cancellation point or calls **pthread_testcancel()**, thus creating a cancellation point. When cancelability is asynchronous, all cancels are acted upon immediately, interrupting the thread with its processing.

*It is recommended that your application not use asynchronous thread cancellation via the **PTHREAD_CANCEL_ASYNCHRONOUS** option of **pthread_setcanceltype()**. See the common user errors section of this document for more information.*

Parameters

state

(Input) New cancel state (one of **PTHREAD_CANCEL_ENABLE** or **PTHREAD_CANCEL_DISABLE**)

oldstate

(Output) Address of variable to contain old cancel state. (**NULL** is allowed)

Authorities and Locks

None.

Return Value

0

pthread_setcancelstate() was successful.

value

pthread_setcancelstate() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_setcancelstate()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

Related Information

- The `<pthread.h>` header file. See [Header files for Pthread functions](#).
- [pthread_cancel\(\)--Cancel Thread](#)
- [pthread_exit\(\)--Terminate Calling Thread](#)
- [pthread_setcanceltype\(\)--Set Cancel Type](#)
- [pthread_testcancel\(\)--Create Cancellation Point](#)

Example

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include "check.h"

void *threadfunc(void *parm)
{
    int    i = 0;
    printf("Entered secondary thread\n");
    pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
    while (1) {
        printf("Secondary thread is looping\n");
        pthread_testcancel();
        sleep(1);
        if (++i == 5) {
            /* Since default cancel type is deferred, changing the state */
            /* will allow the next cancellation point to cancel the thread */
            printf("Cancel state set to ENABLE\n");
            pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
        }
    } /* infinite */
    return NULL;
}

int main(int argc, char **argv)
{
    pthread_t      thread;
    int            rc=0;

    printf("Entering testcase\n");

    /* Create a thread using default attributes */
    printf("Create thread using the NULL attributes\n");
    rc = pthread_create(&thread, NULL, threadfunc, NULL);
    checkResults("pthread_create(NULL)\n", rc);

    /* sleep() is not a very robust way to wait for the thread */
    sleep(3);

    printf("Cancel the thread\n");
    rc = pthread_cancel(thread);
    checkResults("pthread_cancel()\n", rc);

    /* sleep() is not a very robust way to wait for the thread */
    sleep(3);
    printf("Main completed\n");
    return 0;
}
```

```
pthread_setcancelstate()--Set Cancel State
```

```
}
```

Output:

```
Entering testcase  
Create thread using the NULL attributes  
Entered secondary thread  
Secondary thread is looping  
Secondary thread is looping  
Secondary thread is looping  
Cancel the thread  
Secondary thread is looping  
Secondary thread is looping  
Cancel state set to ENABLE  
Main completed
```

pthread_setcanceltype()--Set Cancel Type

Syntax

```
#include <pthread.h>
int pthread_setcanceltype(int type, int *oldtype);
Threadsafe: Yes
Signal Safe: Yes
```

The **pthread_setcanceltype()** function sets the cancel *type* to one of **PTHREAD_CANCEL_DEFERRED** or **PTHREAD_CANCEL_ASYNCHRONOUS** and returns the old cancel type into the location specified by *oldtype* (if *oldtype* is non-NULL)

Cancelability consists of 3 separate states (disabled, deferred, asynchronous) that can be represented by 2 boolean values.

Cancelability	Cancelability State	Cancelability Type
disabled	PTHREAD_CANCEL_DISABLE	PTHREAD_CANCEL_DEFERRED
disabled	PTHREAD_CANCEL_DISABLE	PTHREAD_CANCEL_ASYNCHRONOUS
deferred	PTHREAD_CANCEL_ENABLE	PTHREAD_CANCEL_DEFERRED
asynchronous	PTHREAD_CANCEL_ENABLE	PTHREAD_CANCEL_ASYNCHRONOUS

The default cancelability state is deferred.

When cancelability is disabled, all cancels are held pending in the target thread until the thread changes the cancelability. When cancelability is deferred, all cancels are held pending in the target thread until the thread changes the cancelability, calls a function which is a cancellation point or calls **pthread_testcancel()**, thus creating a cancellation point. When cancelability is asynchronous, all cancels are acted upon immediately, interrupting the thread with its processing.

*It is recommended that your application not use asynchronous thread cancellation via the **PTHREAD_CANCEL_ASYNCHRONOUS** option of **pthread_setcanceltype()**. See the common user errors section of this document for more information.*

Parameters

type

(Input) New cancel type (one of **PTHREAD_CANCEL_DEFERRED** or **PTHREAD_CANCEL_ASYNCHRONOUS**)

oldtype

(Output) Address of variable to contain old cancel type. (**NULL** is allowed)

Authorities and Locks

None.

Return Value

0

pthread_setcanceltype() was successful.

value

pthread_setcanceltype() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_setcanceltype()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

Related Information

- The **<pthread.h>** header file. See [Header files for Pthread functions](#).
- [pthread_cancel\(\)--Cancel Thread](#)
- [pthread_exit\(\)--Terminate Calling Thread](#)
- [pthread_setcancelstate\(\)--Set Cancel State](#)
- [pthread_testcancel\(\)--Create Cancellation Point](#)

Example

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include "check.h"

pthread_mutex_t      mutex = PTHREAD_MUTEX_INITIALIZER;

void cleanupHandler(void *parm)
{
    int rc;
    printf("Inside cleanup handler, unlock mutex\n");
    rc = pthread_mutex_unlock((pthread_mutex_t *)parm);
    checkResults("pthread_mutex_unlock\n", rc);
}

void *threadfunc(void *parm)
{
    int          rc;
    int          oldtype;

    printf("Entered secondary thread, lock mutex\n");
    rc = pthread_mutex_lock(&mutex);
    checkResults("pthread_mutex_lock()\n", rc);

    pthread_cleanup_push(cleanupHandler, &mutex);
    /* We must assume there is a good reason for async. cancellability */
    /* and also, we must assume that if we get interrupted, it is      */
    /* appropriate to unlock the mutex. More than likely it is not      */
    /* because we will have left some data structures in a strange state */
    /* if we are async. interrupted while holding the mutex            */
    rc = pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, &oldtype);
    checkResults("pthread_setcanceltype()\n", rc);

    printf("Secondary thread is now looping\n");
    while (1) { sleep(1); }
    printf("Unexpectedly got out of loop!\n");
    pthread_cleanup_pop(0);
    return NULL;
}
```

pthread_setcanceltype(--Set Cancel Type

```
    }

int main(int argc, char **argv)
{
    pthread_t      thread;
    int            rc=0;
    void           *status;

    printf("Enter Testcase - %s\n", argv[0]);

    /* Create a thread using default attributes */
    printf("Create thread using the NULL attributes\n");
    rc = pthread_create(&thread, NULL, threadfunc, NULL);
    checkResults("pthread_create(NULL)\n", rc);

    /* sleep() is not a very robust way to wait for the thread */
    sleep(1);

    printf("Cancel the thread\n");
    rc = pthread_cancel(thread);
    checkResults("pthread_cancel()\n", rc);

    rc = pthread_join(thread, &status);
    if (status != PTHREAD_CANCELED) {
        printf("Unexpected thread status\n");
        exit(1);
    }
    printf("Main completed\n");
    return 0;
}
```

Output:

```
Enter Testcase - QP0WTEST/TPSETCANT0
Create thread using the NULL attributes
Entered secondary thread, lock mutex
Secondary thread is now looping
Cancel the thread
Inside cleanup handler, unlock mutex
Main completed
```

pthread_testcancel()--Create Cancellation Point

Syntax

```
#include <pthread.h>
void pthread_testcancel(void);
Threadsafe: Yes
Signal Safe: No
```

The **pthread_testcancel()** function creates a cancellation point in the calling thread. If cancelability is currently disabled, this function has no effect. For more information on cancelability, see [Thread cancellation APIs](#).

When cancelability is disabled, all cancels are held pending in the target thread until the thread changes the cancelability. When cancelability is deferred, all cancels are held pending in the target thread until the thread changes the cancelability, calls a function that is a cancellation point, or calls **pthread_testcancel()**, thus creating a cancellation point. When cancelability is asynchronous, all cancels are acted upon immediately, interrupting the thread with its processing.

*You should not use asynchronous thread cancellation via the **PTHREAD_CANCEL_ASYNCHRONOUS** option of **pthread_setcanceltype()**. See the common user errors section of this document for more information.*

Parameters

None.

Authorities and Locks

None.

Return Value

None.

Error Conditions

None.

Related Information

- The **<pthread.h>** header file. See [Header files for Pthread functions](#).
- [pthread_cancel\(\)--Cancel Thread](#)
- [pthread_setcancelstate\(\)--Set Cancel State](#)
- [pthread_setcanceltype\(\)--Set Cancel Type](#)

Example

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include "check.h"

void cleanupHandler(void *parm) {
    printf("Inside cancellation cleanup handler\n");
}
```

pthread_testcancel()--Create Cancellation Point

```
    }

void *threadfunc(void *parm)
{
    unsigned int    i=0;
    int             rc=0, oldState=0;
    printf("Entered secondary thread\n");
    pthread_cleanup_push(cleanupHandler, NULL);
    rc = pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &oldState);
    checkResults("pthread_setcancelstate()\n", rc);
    /* Allow cancel to be pending on this thread */
    sleep(2);
    while (1) {
        printf("Secondary thread is now looping\n");
        ++i;
        sleep(1);
        /* pthread_testcancel() has no effect until cancelability is enabled.*/
        /* At that time, a call to pthread_testcancel() should result in the */
        /* pending cancel being acted upon                                   */
        pthread_testcancel();
        if (i == 5) {
            printf("Cancel state set to ENABLE\n");
            rc = pthread_setcancelstate(PTHREAD_CANCEL_ENABLE,&oldState);
            checkResults("pthread_setcancelstate(2)\n", rc);
            /* Now, cancellation points will allow pending cancels
               to get through to this thread */
        }
    } /* infinite */
    pthread_cleanup_pop(0);
    return NULL;
}

int main(int argc, char **argv)
{
    pthread_t        thread;
    int              rc=0;
    void             *status=NULL;

    printf("Enter Testcase - %s\n", argv[0]);

    /* Create a thread using default attributes */
    printf("Create thread using the NULL attributes\n");
    rc = pthread_create(&thread, NULL, threadfunc, NULL);
    checkResults("pthread_create(NULL)\n", rc);

    sleep(1);
    printf("Cancel the thread\n");
    rc = pthread_cancel(thread);
    checkResults("pthread_cancel()\n", rc);

    rc = pthread_join(thread, &status);
    if (status != PTHREAD_CANCELED) {
        printf("Thread returned unexpected result!\n");
        exit(1);
    }
    printf("Main completed\n");
    return 0;
}
```

Output:

```
Enter Testcase - QP0WTEST/TPTESTC0
Create thread using the NULL attributes
Entered secondary thread
Cancel the thread
Secondary thread is now looping
Secondary thread is now looping
Secondary thread is now looping
Secondary thread is now looping
Secondary thread is now looping
Cancel state set to ENABLE
Secondary thread is now looping
Inside cancellation cleanup handler
Main completed
```

pthread_test_exit_np()--Test Thread Exit Status

Syntax

```
#include <pthread.h>
#include <sched.h>
int pthread_test_exit_np(void **status);
Threadsafe: Yes
Signal Safe: Yes
```

The pthread_test_exit_np() function returns the current state of the thread along with its exit status.

If the thread is currently processing an exit condition due to a call to **pthread_exit()** or cancellation due to being the target of a **pthread_cancel()**, **pthread_test_exit_np()** returns **PTHREAD_STATUS_EXIT_NP** and sets the exit status pointed to by the status parameter to be the current thread exit status.

If the thread is currently running and is not running cancellation cleanup handlers or data destructors while terminating, **pthread_test_exit_np()** returns **PTHREAD_STATUS_ACTIVE_NP**, and does not return the exit status.

This function is not portable.

Parameters

status

Pointer to the parameter to receive the exit status if **PTHREAD_STATUS_EXIT_NP** is returned

Authorities and Locks

None.

Return Value

PTHREAD_STATUS_ACTIVE_NP

The thread is currently not exiting.

PTHREAD_STATUS_EXIT_NP

The thread is currently exiting.

value

pthread_test_exit_np() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_test_exit_np()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The values specified for the argument are not correct.

Related Information

- The **<pthread.h>** header file. See [Header files for Pthread functions.](#)
- [pthread_cancel\(\)--Cancel Thread](#)

- [pthread_exit\(\)--Terminate Calling Thread](#)

Example

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include "check.h"

int      checkStatusFailed1=0;
int      missedHandler1=1;

int      thread1Status=42;

void cleanupHandler1(void *arg)
{
    int          rc;
    void          *status;
    printf("Thread 1 - cleanup handler\n");
    missedHandler1=0;
    rc = pthread_test_exit_np(&status);
    if (rc != PTHREAD_STATUS_EXIT_NP) {
        printf("Thread 1 - returned %d instead "
               "of PTHREAD_STATUS_EXIT_NP\n", rc);
        checkStatusFailed1 = 1;
        return;
    }
    if (__INT(status) != thread1Status) {
        printf("Thread 1 - status = %d\n"
               "Thread 1 - expected %d\n",
               __INT(status), thread1Status);
        checkStatusFailed1=1;
    }
    printf("Thread 1 - correctly got PTHREAD_STATUS_EXIT_NP "
           "and thread exit status of %d\n", thread1Status);
}

void *thread1func(void *parm)
{
    printf("Thread 1 - Entered\n");
    pthread_cleanup_push(cleanupHandler1, NULL);
    pthread_exit(__VOID(thread1Status));
    pthread_cleanup_pop(0);
    return __VOID(0);
}

int main(int argc, char **argv)
{
    pthread_t      thread;
    int            rc=0;
    void            *status;

    printf("Enter Testcase - %s\n", argv[0]);

    rc = pthread_test_exit_np(&status);
    if (rc != PTHREAD_STATUS_ACTIVE_NP) {
        printf("We should always be in an ACTIVE status here! rc=%d\n",
               rc);
        exit(1);
    }
}
```

```
printf("Create the pthread_exit thread\n");
rc = pthread_create(&thread, NULL, thread1func, NULL);
checkResults("pthread_create()\n", rc);

rc = pthread_join(thread, &status);
checkResults("pthread_join()\n", rc);
if (__INT(status) != thread1Status) {
    printf("Wrong status from thread 1\n");
}

if (checkStatusFailed1 || missedHandler1) {
    printf("The thread did not complete its test correctly! "
        " check=%d, missed=%d\n",
        checkStatusFailed1, missedHandler1);
    exit(1);
}
printf("Main completed\n");
return 0;
}
```

Output:

```
Enter Testcase - QP0WTEST/TPTEXIT0
Create the pthread_exit thread
Thread 1 - Entered
Thread 1 - cleanup handler
Thread 1 - correctly got PTHREAD_STATUS_EXIT_NP and thread exit status of 42
Main completed
```


Mutex synchronization APIs

Thread synchronization is required whenever two threads share a resource or need to be aware of what the other threads in a process are doing. Mutexes are the most simple and primitive object used for the co-operative mutual exclusion required to share and protect resources. One thread owns a mutex by locking it successfully, when another thread tries to lock the mutex, that thread will not be allowed to successfully lock the mutex until the owner unlocks it. The mutex support provides different types and behaviors for mutexes that can be tuned to your application requirements.

The table below lists important mutex attributes, their default values, and all supported values.

Attribute	Default value	Supported values
<i>pshared</i>	PTHREAD_PROCESS_PRIVATE	PTHREAD_PROCESS_PRIVATE or PTHREAD_PROCESS_SHARED
<i>kind</i> (non portable)	PTHREAD_MUTEX_NONRECURSIVE_NP	PTHREAD_MUTEX_NONRECURSIVE_NP or PTHREAD_MUTEX_RECURSIVE_NP
<i>name</i> (non portable)	PTHREAD_DEFAULT_MUTEX_NAME_NP "QPOWMTX UNNAMED"	Any name that is 15 characters or less. If not terminated by a null character, name is truncated to 15 characters.
<i>type</i>	PTHREAD_MUTEX_DEFAULT (PTHREAD_MUTEX_NORMAL)	PTHREAD_MUTEX_DEFAULT or PTHREAD_MUTEX_NORMAL or PTHREAD_MUTEX_RECURSIVE or PTHREAD_MUTEX_ERRORCHECK or PTHREAD_MUTEX_OWNERTERM_NP The PTHREAD_MUTEX_OWNERTERM_NP attribute value is non portable

For information about the examples included with the APIs, see the [information on the API examples](#).

To view the API list by description, see [Mutex synchronization APIs](#).

Mutex synchronization APIs by name

- [pthread_mutexattr_destroy\(\)--Destroy Mutex Attributes Object](#)
- [pthread_mutexattr_getkind_np\(\)--Get Mutex Kind Attribute](#)
- [pthread_mutexattr_getname_np\(\)--Get Name from Mutex Attributes Object](#)
- [pthread_mutexattr_getpshared\(\)--Get Process Shared Attribute from Mutex Attributes Object](#)
- [pthread_mutexattr_gettype\(\)--Get Mutex Type Attribute](#)
- [pthread_mutexattr_init\(\)--Initialize Mutex Attributes Object](#)
- [pthread_mutexattr_setkind_np\(\)--Get Mutex Kind Attribute](#)
- [pthread_mutexattr_setname_np\(\)--Set Name in Mutex Attributes Object](#)
- [pthread_mutexattr_setpshared\(\)--Set Process Shared Attribute in Mutex Attributes Object](#)
- [pthread_mutexattr_settype\(\)--Set Mutex Type Attribute](#)
- [pthread_set_mutexattr_default_np\(\)--Set Default Mutex Attributes Object Kind Attribute](#)
- [pthread_mutex_destroy\(\)--Destroy Mutex](#)
- [pthread_mutex_init\(\)--Initialize Mutex](#)
- [pthread_mutex_lock\(\)--Lock Mutex](#)
- [pthread_mutex_timedlock_np\(\)--Lock Mutex with Time-Out](#)
- [pthread_mutex_trylock\(\)--Lock Mutex with No Wait](#)

Mutex synchronization APIs

- [pthread_mutex_unlock\(\)--Unlock Mutex](#)
- [pthread_lock_global_np\(\)--Lock Global Mutex](#)
- [pthread_unlock_global_np\(\)--Unlock Global Mutex](#)

pthread_mutexattr_destroy()--Destroy Mutex Attributes Object

Syntax

```
#include <pthread.h>
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
Threadsafe: Yes
Signal Safe: Yes
```

The **pthread_mutexattr_destroy()** function destroys a mutex attributes object and allows the system to reclaim any resources associated with that mutex attributes object. This does not have an effect on any mutexes created using this mutex attributes object.

Parameters

attr

(Input) Address of the mutex attributes object to be destroyed

Authorities and Locks

None.

Return Value

0

pthread_mutexattr_destroy() was successful.

value

pthread_mutexattr_destroy() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_mutexattr_destroy()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

Related Information

- The **<pthread.h>** header file. See [Header files for Pthread functions](#).
- [pthread_mutexattr_init\(\)--Initialize Mutex Attributes Object](#)
- [pthread_mutex_init\(\)--Initialize Mutex](#)

Example

```
#include <pthread.h>
#include <stdio.h>
#include "check.h"

pthread_mutex_t      mutex;
```

```
int main(int argc, char **argv)
{
    int                rc=0;
    pthread_mutexattr_t mta;

    printf("Entering testcase\n");

    printf("Create a default mutex attribute\n");
    rc = pthread_mutexattr_init(&mta);
    checkResults("pthread_mutexattr_init\n", rc);

    printf("Create the mutex using a mutex attributes object\n");
    rc = pthread_mutex_init(&mutex, &mta);
    checkResults("pthread_mutex_init(mta)\n", rc);

    printf("- At this point, the mutex with its default attributes\n");
    printf("- Can be used from any threads that want to use it\n");

    printf("Destroy mutex attribute\n");
    rc = pthread_mutexattr_destroy(&mta);
    checkResults("pthread_mutexattr_destroy()\n", rc);

    printf("Destroy mutex\n");
    rc = pthread_mutex_destroy(&mutex);
    checkResults("pthread_mutex_destroy()\n", rc);

    printf("Main completed\n");
    return 0;
}
```

Output:

```
Entering testcase
Create a default mutex attribute
Create the mutex using a mutex attributes object
- At this point, the mutex with its default attributes
- Can be used from any threads that want to use it
Destroy mutex attribute
Destroy mutex
Main completed
```

pthread_mutexattr_getkind_np()--Get Mutex Kind Attribute

Syntax

```
#include <pthread.h>
#include <sched.h>
int pthread_mutexattr_getkind_np(const pthread_mutexattr_t *attr,
                                int *kind);
```

Threadsafe: Yes

Signal Safe: Yes

The **pthread_mutexattr_getkind_np()** function retrieves the *kind* attribute from the mutex attributes object specified by *attr*. The mutex kind attribute is used to create mutexes with different behaviors.

The *kind* returned is one of **PTHREAD_MUTEX_NONRECURSIVE_NP** or **PTHREAD_MUTEX_RECURSIVE_NP**.

A recursive mutex can be locked repeatedly by the owner. The mutex does not become unlocked until the owner has called **pthread_mutex_unlock()** for each successful lock request that it has outstanding on the mutexes.

This function is not portable.

Parameters

attr

(Input) Address of the mutex attributes object

kind

(Output) Address of the variable to receive the *kind* attribute

Authorities and Locks

None.

Return Value

0

pthread_mutexattr_getkind_np() was successful.

value

pthread_mutexattr_getkind_np() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_mutexattr_getkind_np()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

Related Information

- The `<pthread.h>` header file. See [Header files for Pthread functions](#).
- [pthread_mutexattr_init\(\)--Initialize Mutex Attributes Object](#)
- [pthread_mutexattr_setkind_np\(\)--Set Mutex Kind Attribute](#)
- [pthread_mutex_init\(\)--Initialize Mutex](#)

Example

```
#include <pthread.h>
#include <stdio.h>
#include "check.h"

void showKind(pthread_mutexattr_t *mta) {
    int          rc;
    int          kind;

    printf("Check kind attribute\n");
    rc = pthread_mutexattr_getkind_np(mta, &kind);
    checkResults("pthread_mutexattr_getpshared()\n", rc);

    printf("The pshared attributed is: ");
    switch (kind) {
    case PTHREAD_MUTEX_NONRECURSIVE_NP:
        printf("PTHREAD_MUTEX_NONRECURSIVE_NP\n");
        break;
    case PTHREAD_MUTEX_RECURSIVE_NP:
        printf("PTHREAD_MUTEX_RECURSIVE_NP\n");
        break;
    default :
        printf("! kind Error kind=%d !\n", kind);
        exit(1);
    }
    return;
}

int main(int argc, char **argv)
{
    int          rc=0;
    pthread_mutexattr_t  mta;
    int          pshared=0;

    printf("Enter Testcase - %s\n", argv[0]);

    printf("Create a default mutex attribute\n");
    rc = pthread_mutexattr_init(&mta);
    checkResults("pthread_mutexattr_init()\n", rc);
    showKind(&mta);

    printf("Change mutex kind attribute\n");
    rc = pthread_mutexattr_setkind_np(&mta, PTHREAD_MUTEX_RECURSIVE_NP);
    checkResults("pthread_mutexattr_setkind()\n", rc);
    showKind(&mta);

    printf("Destroy mutex attribute\n");
    rc = pthread_mutexattr_destroy(&mta);
    checkResults("pthread_mutexattr_destroy()\n", rc);
}
```

```
    printf("Main completed\n");  
    return 0;  
}
```

Output:

```
Enter Testcase - QP0WTEST/TPMTXAKNO  
Create a default mutex attribute  
Check kind attribute  
The pshared attributed is:  
PTHREAD_MUTEX_NONRECURSIVE_NP  
Change mutex kind attribute  
Check kind attribute  
The pshared attributed is:  
PTHREAD_MUTEX_RECURSIVE_NP  
Destroy mutex attribute  
Main completed
```

pthread_mutexattr_getname_np()--Get Name from Mutex Attributes Object

Syntax

```
#include <pthread.h>
int pthread_mutexattr_getname_np(const pthread_mutexattr_t *attr, char
*name);
```

Threadsafe: Yes

Signal Safe: Yes

The **pthread_mutexattr_getname_np()** function retrieves the name attribute associated with the mutex attribute specified by *attr*. The buffer specified by *name* must be at least 16 characters in length. If the length of the mutex name is less than or equal to 15 characters, it is null terminated in the output buffer.

By default, each pthread_mutex_t has the name "QP0WMTX UNNAMED" associated with it. The name attribute is used by various OS/400 system utilities to aid in debugging and service. One example is the WRKJOB command, which has a 'work with mutexes' menu choice to show which mutexes are currently locked and which mutexes are being waited for.

If you should give unique names to all mutexes created to aid in debugging deadlock or performance problems. Use the CL command **WRKJOB**, option 20, to help debug mutex deadlocks.

Parameters

attr

(Input) Address of the mutex attributes object

name

(Output) Address of a 16-byte character buffer to receive the name

Authorities and Locks

None.

Return Value

0

pthread_mutexattr_getname_np() was successful.

value

pthread_mutexattr_getname_np() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_mutexattr_getname_np()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

Related Information

- The `<pthread.h>` header file. See [Header files for Pthread functions](#).
- [pthread_mutexattr_init\(\)--Initialize Mutex Attributes Object](#)
- [pthread_mutexattr_setname_np\(\)--Set Name in Mutex Attributes Object](#)
- [pthread_mutex_init\(\)--Initialize Mutex](#)

Example

```
#include <pthread.h>
#include <stdio.h>
#include "check.h"

int main(int argc, char **argv)
{
    int                rc=0;
    pthread_mutexattr_t mta;
    char               mutexname[16];

    printf("Entering testcase\n");

    printf("Create a default mutex attribute\n");
    rc = pthread_mutexattr_init(&mta);
    checkResults("pthread_mutexattr_init\n", rc);

    memset(mutexname, 0, sizeof(mutexname));
    printf("Find out what the default name of the mutex is\n");
    rc = pthread_mutexattr_getname_np(&mta, mutexname);
    checkResults("pthread_mutexattr_getname_np()\n", rc);

    printf("The default mutex name will be: %.15s\n", mutexname);
    printf("- At this point, mutexes created with this attribute\n");
    printf("- will show up by name on many OS/400 debug and service\n");
    printf("screens\n");
    printf("- The default attribute contains a special automatically\n");
    printf("- incrementing name that changes for each mutex created in \n");
    printf("- the process\n");

    printf("Destroy mutex attribute\n");
    rc = pthread_mutexattr_destroy(&mta);
    checkResults("pthread_mutexattr_destroy()\n", rc);

    printf("Main completed\n");
    return 0;
}
```

Output:

```
Entering testcase
Create a default mutex attribute
Find out what the default name of the mutex is
The default mutex name is: QP0WMTX UNNAMED
- At this point, mutexes created with this attribute
- will show up by name on many OS/400 debug and service screens
- The default attribute contains a special automatically
- incrementing name that changes for each mutex created in
- the process
```

pthread_mutexattr_getname_np()--Get Name from Mutex Attributes Object

Destroy mutex attribute

Main completed

pthread_mutexattr_getpshared()--Get Process Shared Attribute from Mutex Attributes Object

Syntax

```
#include <pthread.h>
int pthread_mutexattr_getpshared(const pthread_mutexattr_t *attr, int
*pshared);
Threadsafe: Yes
Signal Safe: Yes
```

The **pthread_mutexattr_getpshared()** function retrieves the current setting of the process shared attribute from the mutex attributes object. The process shared attribute indicates whether the mutex that is created using the mutex attributes object can be shared between threads in separate processes (**PTHREAD_PROCESS_SHARED**) or shared between threads within the same process (**PTHREAD_PROCESS_PRIVATE**).

Even if the mutex in storage is accessible from two separate processes, it cannot be used from both processes unless the process shared attribute is **PTHREAD_PROCESS_SHARED**.

The default pshared attribute for mutex attributes objects is **PTHREAD_PROCESS_PRIVATE**.

Parameters

attr

(Input) Address of the variable that contains the mutex attributes object

pshared

(Output) Address of the variable to contain the pshared attribute result

Authorities and Locks

None.

Return Value

0

pthread_mutexattr_getpshared() was successful.

value

pthread_mutexattr_getpshared() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_mutexattr_getpshared()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

Related Information

- The **<pthread.h>** header file. See [Header files for Pthread functions](#).
- [pthread_mutexattr_init\(\)--Initialize Mutex Attributes Object](#)

- [pthread_mutexattr_setpshared\(\)--Set Process Shared Attribute in Mutex Attributes Object](#)
- [pthread_mutex_init\(\)--Initialize Mutex](#)

Example

```
#include <pthread.h>
#include <stdio.h>
#include "check.h"

void showPshared(int pshared) {
    printf("The pshared attribute is: ");
    switch (pshared) {
        case PTHREAD_PROCESS_PRIVATE:
            printf("PTHREAD_PROCESS_PRIVATE\n");
            break;
        case PTHREAD_PROCESS_SHARED:
            printf("PTHREAD_PROCESS_SHARED\n");
            break;
        default :
            printf("! pshared Error !\n");
            exit(1);
    }
    return;
}

int main(int argc, char **argv)
{
    int                rc=0;
    pthread_mutexattr_t mta;
    int                pshared=0;

    printf("Entering testcase\n");

    printf("Create a default mutex attribute\n");
    rc = pthread_mutexattr_init(&mta);
    checkResults("pthread_mutexattr_init()\n", rc);

    printf("Check pshared attribute\n");
    rc = pthread_mutexattr_getpshared(&mta, &pshared);
    checkResults("pthread_mutexattr_getpshared()\n", rc);
    showPshared(pshared);

    printf("Destroy mutex attribute\n");
    rc = pthread_mutexattr_destroy(&mta);
    checkResults("pthread_mutexattr_destroy()\n", rc);

    printf("Main completed\n");
    return 0;
}
```

Output:

```
Entering testcase
Create a default mutex attribute
Check pshared attribute
The pshared attribute is: PTHREAD_PROCESS_PRIVATE
Destroy mutex attribute
Main completed
```

pthread_mutexattr_gettype()--Get Mutex Type Attribute

Syntax

```
#include <pthread.h>
int pthread_mutexattr_gettype(const pthread_mutexattr_t *attr,
                              int *type);
```

Threadsafe: Yes

Signal Safe: Yes

The **pthread_mutexattr_gettype()** function retrieves the *type* attribute from the mutex attributes object specified by *attr*. The mutex type attribute is used to create mutexes with different behaviors.

The type returned is one of **PTHREAD_MUTEX_DEFAULT**, **PTHREAD_MUTEX_NORMAL**, **PTHREAD_MUTEX_RECURSIVE**, **PTHREAD_MUTEX_ERRORCHECK**, or **PTHREAD_MUTEX_OWNERTERM_NP**.

The default mutex type (or **PTHREAD_MUTEX_DEFAULT**) is **PTHREAD_MUTEX_NORMAL**.

Mutex Types

A normal mutex cannot be locked repeatedly by the owner. Attempts by a thread to relock an already held mutex, or to lock a mutex that was held by another thread when that thread terminated, cause a deadlock condition.

A recursive mutex can be locked repeatedly by the owner. The mutex does not become unlocked until the owner has called **pthread_mutex_unlock()** for each successful lock request that it has outstanding on the mutex.

An errorcheck mutex checks for deadlock conditions that occur when a thread relocks an already held mutex. If a thread attempts to relock a mutex that it already holds, the lock request fails with the **EDEADLK** error.

An ownerterm mutex is an OS/400 extension to the errorcheck mutex type. An ownerterm mutex checks for deadlock conditions that occur when a thread relocks an already held mutex. If a thread attempts to relock a mutex that it already holds, the lock request fails with the **EDEADLK** error. An ownerterm mutex also checks for deadlock conditions that occur when a thread attempts to lock a mutex that was held by another thread when that thread terminated (an orphaned mutex). If a thread attempts to lock an orphaned mutex, the lock request fails with the **EOWNERTERM** error.

Parameters

attr

(Input) Address of the mutex attributes object

type

(Output) Address of the variable to receive the type attribute

Authorities and Locks

None.

Return Value

0

pthread_mutexattr_gettype() was successful.

value

pthread_mutexattr_gettype()--Get Mutex Type Attribute

pthread_mutexattr_gettype() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_mutexattr_gettype()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

Related Information

- The `<pthread.h>` header file. See [Header files for Pthread functions](#).
- [pthread_mutexattr_init\(\)--Initialize Mutex Attributes Object](#)
- [pthread_mutexattr_settype\(\)--Set Mutex Type Attribute](#)
- [pthread_mutex_init\(\)--Initialize Mutex](#)

Example

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include "check.h"

int showType(pthread_mutexattr_t *mta) {
    int          rc;
    int          type;

    printf("Check type attribute\n");
    rc = pthread_mutexattr_gettype(mta, &type);
    checkResults("pthread_mutexattr_gettype()\n", rc);

    printf("The type attributed is: ");
    switch (type) {
    case PTHREAD_MUTEX_NORMAL:
        printf("PTHREAD_MUTEX_NORMAL (DEFAULT)\n");
        break;
    case PTHREAD_MUTEX_RECURSIVE:
        printf("PTHREAD_MUTEX_RECURSIVE\n");
        break;
    case PTHREAD_MUTEX_ERRORCHECK:
        printf("PTHREAD_MUTEX_ERRORCHECK\n");
        break;
    case PTHREAD_MUTEX_OWNERTERM_NP:
        printf("PTHREAD_MUTEX_OWNERTERM_NP\n");
        break;
    default :
        printf("! type Error type=%d !\n", type);
        exit(1);
    }
    return type;
}

int main(int argc, char **argv)
{
    int          rc=0;
    pthread_mutexattr_t  mta;
```

pthread_mutexattr_gettype())--Get Mutex Type Attribute

```
int                type=0;
pthread_mutex_t    mutex;
struct timespec    ts;

printf("Enter Testcase - %s\n", argv[0]);

printf("Create a default mutex attribute\n");
rc = pthread_mutexattr_init(&mta);
checkResults("pthread_mutexattr_init()\n", rc);

printf("Change mutex type attribute to recursive\n");
rc = pthread_mutexattr_settype(&mta, PTHREAD_MUTEX_RECURSIVE);
checkResults("pthread_mutexattr_settype()\n", rc);
showType(&mta);

rc = pthread_mutexattr_setname_np(&mta, "RECURSIVE ONE");
checkResults("pthread_mutexattr_setname_np()\n", rc);

printf("Create the named, recursive mutex\n");
rc = pthread_mutex_init(&mutex, &mta);
checkResults("pthread_mutex_init()\n", rc);

printf("Lock the named, recursive mutex\n");
rc = pthread_mutex_lock(&mutex);
checkResults("pthread_mutex_lock() 1\n", rc);

printf("ReLock the named, recursive mutex\n");
rc = pthread_mutex_lock(&mutex);
checkResults("pthread_mutex_lock() 2\n", rc);

printf("Trylock the named, recursive mutex\n");
rc = pthread_mutex_trylock(&mutex);
checkResults("pthread_mutex_trylock()\n", rc);

printf("Timedlock the named, recursive mutex\n");
ts.tv_sec = 5;
ts.tv_nsec = 0;
rc = pthread_mutex_timedlock_np(&mutex, &ts);
checkResults("pthread_mutex_timedlock_np()\n", rc);

printf("Sleeping for a short time holding the recursive mutex\n");
printf("Use DSPJOB, option 19 to see the held mutex\n");
sleep(30);

printf("Unlock the mutex 4 times\n");
rc = pthread_mutex_unlock(&mutex);
checkResults("pthread_mutex_unlock() 1\n", rc);

rc = pthread_mutex_unlock(&mutex);
checkResults("pthread_mutex_unlock() 2\n", rc);

rc = pthread_mutex_unlock(&mutex);
checkResults("pthread_mutex_unlock() 3\n", rc);

rc = pthread_mutex_unlock(&mutex);
checkResults("pthread_mutex_unlock() 4\n", rc);

printf("Cleanup\n");
rc = pthread_mutex_destroy(&mutex);
checkResults("pthread_mutex_destroy()\n", rc);
```

pthread_mutexattr_gettype())--Get Mutex Type Attribute

```
rc = pthread_mutexattr_destroy(&mta);  
checkResults("pthread_mutexattr_destroy()\n", rc);  
  
printf("Main completed\n");  
return 0;  
}
```

Output

```
Enter Testcase - QP0WTEST/TPMTXTYP0  
Create a default mutex attribute  
Change mutex type attribute to recursive  
Check type attribute  
The type attributed is: PTHREAD_MUTEX_RECURSIVE  
Create the named, recursive mutex  
Lock the named, recursive mutex  
ReLock the named, recursive mutex  
Trylock the named, recursive mutex  
Timedlock the named, recursive mutex  
Sleeping for a short time holding the recursive mutex  
Use DSPJOB, option 19 to see the held mutex  
Unlock the mutex 4 times  
Cleanup  
Main completed
```


pthread_mutexattr_init()--Initialize Mutex Attributes Object

Syntax

```
#include <pthread.h>
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
Threadsafe: Yes
Signal Safe: Yes
```

The **pthread_mutexattr_init()** function initializes the mutex attributes object referenced by *attr* to the default attributes. The mutex attributes object can be used in a call to **pthread_mutex_init()** to create a mutex.

Parameters

attr

(Input/Output) Address of the variable to contain the mutex attributes object

Authorities and Locks

None.

Return Value

0

pthread_mutexattr_init() was successful.

value

pthread_mutexattr_init() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_mutexattr_init()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

Related Information

- The **<pthread.h>** header file. See [Header files for Pthread functions](#).
- [pthread_mutexattr_destroy\(\)--Destroy Mutex Attributes Object](#)
- [pthread_mutex_destroy\(\)--Destroy Mutex](#)
- [pthread_mutex_init\(\)--Initialize Mutex](#)

Example

```
#include <pthread.h>
#include <stdio.h>
#include "check.h"

pthread_mutex_t    mutex;
```

```
int main(int argc, char **argv)
{
    int                rc=0;
    pthread_mutexattr_t mta;

    printf("Entering testcase\n");

    printf("Create a default mutex attribute\n");
    rc = pthread_mutexattr_init(&mta);
    checkResults("pthread_mutexattr_init\n", rc);

    printf("Create the mutex using a mutex attributes object\n");
    rc = pthread_mutex_init(&mutex, &mta);
    checkResults("pthread_mutex_init(mta)\n", rc);

    printf("- At this point, the mutex with its default attributes\n");
    printf("- Can be used from any threads that want to use it\n");

    printf("Destroy mutex attribute\n");
    rc = pthread_mutexattr_destroy(&mta);
    checkResults("pthread_mutexattr_destroy()\n", rc);

    printf("Destroy mutex\n");
    rc = pthread_mutex_destroy(&mutex);
    checkResults("pthread_mutex_destroy()\n", rc);

    printf("Main completed\n");
    return 0;
}
```

Output:

```
Entering testcase
Create a default mutex attribute
Create the mutex using a mutex attributes object
- At this point, the mutex with its default attributes
- Can be used from any threads that want to use it
Destroy mutex attribute
Destroy mutex
Main completed
```

pthread_mutexattr_setkind_np()--Set Mutex Kind Attribute

Syntax

```
#include <pthread.h>
#include <sched.h>
int pthread_mutexattr_setkind_np(pthread_mutexattr_t *attr,
                                int kind);
```

Threadsafe: Yes

Signal Safe: Yes

The **pthread_mutexattr_setkind_np()** function sets the *kind* attribute in the mutex attributes object specified by *attr*. The mutex kind attribute is used to create mutexes with different behaviors.

The *kind* set may be one of **PTHREAD_MUTEX_NONRECURSIVE_NP** or **PTHREAD_MUTEX_RECURSIVE_NP**.

A recursive mutex can be locked repeatedly by the owner. The mutex does not become unlocked until the owner has called **pthread_mutex_unlock()** for each successful lock request that it has outstanding on the mutexes. The maximum number of recursive locks by the owning thread is 32,767.

This function is not portable

Parameters

attr

(Input) Address of the mutex attributes object

kind

(Input) Variable containing the *kind* attribute.

Authorities and Locks

None.

Return Value

0

pthread_mutexattr_setkind_np() was successful.

value

pthread_mutexattr_setkind_np() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_mutexattr_setkind_np()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

Related Information

- The `<pthread.h>` header file. See [Header files for Pthread functions](#).
- [pthread_mutexattr_getkind_np\(\)--Get Mutex Kind Attribute](#)
- [pthread_mutex_init\(\)--Initialize Mutex](#)

Example

```
#include <pthread.h>
#include <stdio.h>
#include "check.h"

void showKind(pthread_mutexattr_t *mta) {
    int rc;
    int kind;

    printf("Check kind attribute\n");
    rc = pthread_mutexattr_getkind_np(mta, &kind);
    checkResults("pthread_mutexattr_getpshared()\n", rc);

    printf("The pshared attributed is: ");
    switch (kind) {
    case PTHREAD_MUTEX_NONRECURSIVE_NP:
        printf("PTHREAD_MUTEX_NONRECURSIVE_NP\n");
        break;
    case PTHREAD_MUTEX_RECURSIVE_NP:
        printf("PTHREAD_MUTEX_RECURSIVE_NP\n");
        break;
    default :
        printf("! kind Error kind=%d !\n", kind);
        exit(1);
    }
    return;
}

int main(int argc, char **argv)
{
    int rc=0;
    pthread_mutexattr_t mta;
    int pshared=0;

    printf("Enter Testcase - %s\n", argv[0]);

    printf("Create a default mutex attribute\n");
    rc = pthread_mutexattr_init(&mta);
    checkResults("pthread_mutexattr_init()\n", rc);
    showKind(&mta);

    printf("Change mutex kind attribute\n");
    rc = pthread_mutexattr_setkind_np(&mta, PTHREAD_MUTEX_RECURSIVE_NP);
    checkResults("pthread_mutexattr_setkind()\n", rc);
    showKind(&mta);

    printf("Destroy mutex attribute\n");
    rc = pthread_mutexattr_destroy(&mta);
    checkResults("pthread_mutexattr_destroy()\n", rc);

    printf("Main completed\n");
}
```

```
pthread_mutexattr_setkind_np()--Set Mutex Kind Attribute
```

```
    return 0;  
}
```

Output:

```
Enter Testcase - QP0WTEST/TPMTXAKNO  
Create a default mutex attribute  
Check kind attribute  
The pshared attributed is:  
PTHREAD_MUTEX_NONRECURSIVE_NP  
Change mutex kind attribute  
Check kind attribute  
The pshared attributed is:  
PTHREAD_MUTEX_RECURSIVE_NP  
Destroy mutex attribute  
Main completed
```

pthread_mutexattr_setname_np()--Set Name in Mutex Attributes Object

Syntax

```
#include <pthread.h>
int pthread_mutexattr_setname_np(pthread_mutexattr_t *attr, const char
*name);
Threadsafe: Yes
Signal Safe: Yes
```

The **pthread_mutexattr_setname_np()** function changes the name attribute associated with the mutex attribute specified by *attr*. The buffer specified by *name* must contain a null terminated string of 15 characters or less in length (not including the NULL). If the length of name is greater than 15 characters, the excess characters are ignored. If *name* is null, the mutex name attribute is reset to the default.

By default, each pthread_mutex_t has the name "QP0WMTX UNNAMED" associated with it. The name attribute is used by various OS/400 system utilities to aid in debug and service. One example is the WRKJOB command, which has a `work with mutexes' menu choice to show which mutexes are currently locked and which mutexes are being waited for.

If you should give unique names to all mutexes created to aid in debugging deadlock or performance problems. Use the CL command **WRKJOB**, option 20, to help debug mutex deadlocks.

This function is not portable.

Parameters

attr

(Input) Address of the mutex attributes object

name

(Input) Address of a null terminated character buffer containing the name

Authorities and Locks

None.

Return Value

0

pthread_mutexattr_setname_np() was successful.

value

pthread_mutexattr_setname_np() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_mutexattr_setname_np()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

Related Information

- The `<pthread.h>` header file. See [Header files for Pthread functions](#).
- [pthread_mutexattr_getname_np\(\)--Get Name from Mutex Attributes Object](#)
- [pthread_mutex_init\(\)--Initialize Mutex](#)

Example

```
#include <pthread.h>
#include <stdio.h>
#include "check.h"

int main(int argc, char **argv)
{
    int                rc=0;
    pthread_mutexattr_t mta;
    char               mutexname[16];

    printf("Entering testcase\n");

    printf("Create a default mutex attribute\n");
    rc = pthread_mutexattr_init(&mta);
    checkResults("pthread_mutexattr_init\n", rc);

    memset(mutexname, 0, sizeof(mutexname));
    printf("Find out what the default name of the mutex is\n");
    rc = pthread_mutexattr_getname_np(&mta, mutexname);
    checkResults("pthread_mutexattr_getname_np()\n", rc);

    printf("The default mutex name will be: %.15s\n", mutexname);
    printf("- At this point, mutexes created with this attribute\n");
    printf("- will show up by name on many OS/400 debug and service\n");
    printf("screens\n");
    printf("- The default attribute contains a special automatically\n");
    printf("- incrementing name that changes for each mutex created in \n");
    printf("- the process\n");

    printf("Destroy mutex attribute\n");
    rc = pthread_mutexattr_destroy(&mta);
    checkResults("pthread_mutexattr_destroy()\n", rc);

    printf("Main completed\n");
    return 0;
}
```

Output:

```
Entering testcase
Create a default mutex attribute
Find out what the default name of the mutex is
The default mutex name will be: QP0WMTX UNNAMED
The new mutex name will be: <My Mutex>
Destroy mutex attribute
Main completed
```

pthread_mutexattr_setpshared()--Set Process Shared Attribute in Mutex Attributes Object

Syntax

```
#include <pthread.h>
int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr,
                                int pshared);
```

Threadsafe: Yes

Signal Safe: Yes

The **pthread_mutexattr_setpshared()** function sets the current pshared attribute for the mutex attributes object. The process shared attribute indicates whether the mutex that is created using the mutex attributes object can be shared between threads in separate processes (**PTHREAD_PROCESS_SHARED**) or shared between threads within the same process (**PTHREAD_PROCESS_PRIVATE**).

Even if the mutex in storage is accessible from two separate processes, it cannot be used from both processes unless the process shared attribute is **PTHREAD_PROCESS_SHARED**.

The default pshared attribute for mutex attributes objects is **PTHREAD_PROCESS_PRIVATE**.

Parameters

attr

(Input) Address of the variable containing the mutex attributes object

pshared

(Input) One of **PTHREAD_PROCESS_SHARED** or **PTHREAD_PROCESS_PRIVATE**

Authorities and Locks

None.

Return Value

0

pthread_mutexattr_setpshared() was successful.

value

pthread_mutexattr_setpshared() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_mutexattr_setpshared()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

Related Information

- The **<pthread.h>** header file. See [Header files for Pthread functions](#).
- [pthread_mutexattr_getpshared\(\)--Get Process Shared Attribute from Mutex Attributes Object](#)

- [pthread_mutex_init\(\)--Initialize Mutex](#)

Example

```
#include <pthread.h>
#include <stdio.h>
#include "check.h"

void showPshared(pthread_mutexattr_t *mta) {
    int rc;
    int pshared;

    printf("Check pshared attribute\n");
    rc = pthread_mutexattr_getpshared(mta, &pshared);
    checkResults("pthread_mutexattr_getpshared()\n", rc);

    printf("The pshared attributed is: ");
    switch (pshared) {
    case PTHREAD_PROCESS_PRIVATE:
        printf("PTHREAD_PROCESS_PRIVATE\n");
        break;
    case PTHREAD_PROCESS_SHARED:
        printf("PTHREAD_PROCESS_SHARED\n");
        break;
    default :
        printf("! pshared Error !\n");
        exit(1);
    }
    return;
}

int main(int argc, char **argv)
{
    int rc=0;
    pthread_mutexattr_t mta;
    int pshared=0;

    printf("Entering testcase\n");

    printf("Create a default mutex attribute\n");
    rc = pthread_mutexattr_init(&mta);
    checkResults("pthread_mutexattr_init()\n", rc);
    showPshared(&mta);

    printf("Change pshared attribute\n");
    rc = pthread_mutexattr_setpshared(&mta, PTHREAD_PROCESS_SHARED);
    checkResults("pthread_mutexattr_setpshared()\n", rc);
    showPshared(&mta);

    printf("Destroy mutex attribute\n");
    rc = pthread_mutexattr_destroy(&mta);
    checkResults("pthread_mutexattr_destroy()\n", rc);

    printf("Main completed\n");
    return 0;
}
```

Output:

```
pthread_mutexattr_setpshared()--Set Process Shared Attribute in Mutex Attributes Object  
  
Entering testcase  
Create a default mutex attribute  
Check pshared attribute  
The pshared attribute is: PTHREAD_PROCESS_PRIVATE  
Change pshared attribute  
The pshared attribute is: PTHREAD_PROCESS_SHARED  
Destroy mutex attribute  
Main completed
```

pthread_mutexattr_settype()--Set Mutex Type Attribute

Syntax

```
#include <pthread.h>
int pthread_mutexattr_settype(pthread_mutexattr_t *attr,
                             int type);
```

Threadsafe: Yes

Signal Safe: Yes

The **pthread_mutexattr_settype()** function sets the *type* attribute in the mutex attributes object specified by *attr*. The mutex type attribute is used to create mutexes with different behaviors.

The type will be one of **PTHREAD_MUTEX_DEFAULT**, **PTHREAD_MUTEX_NORMAL**, **PTHREAD_MUTEX_RECURSIVE**, **PTHREAD_MUTEX_ERRORCHECK**, or **PTHREAD_MUTEX_OWNERTERM_NP** or the **EINVAL** error will be returned.

The default mutex type (or **PTHREAD_MUTEX_DEFAULT**) is **PTHREAD_MUTEX_NORMAL**.

Mutex Types

A normal mutex cannot be locked repeatedly by the owner. Attempts by a thread to relock an already held mutex, or to lock a mutex that was held by another thread when that thread terminated result in a deadlock condition.

A recursive mutex can be locked repeatedly by the owner. The mutex does not become unlocked until the owner has called **pthread_mutex_unlock()** for each successful lock request that it has outstanding on the mutex.

An errorcheck mutex checks for deadlock conditions that occur when a thread re-locks an already held mutex. If a thread attempts to relock a mutex that it already holds, the lock request fails with the **EDEADLK** error

An ownerterm mutex is an OS/400 extension to the errorcheck mutex type. An ownerterm mutex checks for deadlock conditions that occur when a thread re-locks an already held mutex. If a thread attempts to relock a mutex that it already holds, the lock request fails with the **EDEADLK** error. An ownerterm mutex also checks for deadlock conditions that occur when a thread attempts to lock a mutex that was held by another thread when that thread terminated (an orphaned mutex). If a thread attempts to lock an orphaned mutex, the lock request fails with the **EOWNERTERM** error.

Parameters

attr

(Input) Address of the mutex attributes object

type

(Input) Address of the type attribute to be set.

Authorities and Locks

None.

Return Value

0

pthread_mutexattr_settype() was successful.

value

pthread_mutexattr_settype() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_mutexattr_settype()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

Related Information

- The <pthread.h> header file. See [Header files for Pthread functions](#).
- [pthread_mutexattr_gettype\(\)--Get Mutex Type Attribute](#)
- [pthread_mutex_init\(\)--Initialize Mutex](#)

Example

See [pthread_mutexattr_gettype\(\)](#) for an example.

pthread_set_mutexattr_default_np()--Set Default Mutex Attributes Object Kind Attribute

Syntax

```
#include <pthread.h>
#include <sched.h>
int pthread_set_mutexattr_default_np(int kind);
Threadsafe: Yes
Signal Safe: Yes
```

The **pthread_set_mutexattr_default_np()** function sets the kind attribute in the default mutex attribute object. The default mutex attributes object is used when **pthread_mutex_init()** is called to specify a **NULL** pointer for the mutex attributes object parameter.

The *kind* set may be one of **PTHREAD_MUTEX_NONRECURSIVE_NP** or **PTHREAD_MUTEX_RECURSIVE_NP**.

The **pthread_set_mutexattr_default_np()** function does not affect any currently existing mutex attributes objects, nor does it affect the subsequent behavior of **pthread_mutexattr_init()** or the **PTHREAD_MUTEX_INITIALIZER** macro.

Calls to **pthread_set_mutexattr_default_np()** change how the run-time of the threads creates default mutexes for all code running in the current process. You can negatively affect other code in your process that uses pthread mutexes by using this function.

Use of this function is not recommended because it can affect the creation of mutexes that your application does not directly own.

This function is not portable.

Parameters

kind

(Input) Variable containing the kind attribute

Authorities and Locks

None.

Return Value

0

pthread_set_mutexattr_default() was successful.

value

pthread_set_mutexattr_default() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_set_mutexattr_default()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

Related Information

- The `<pthread.h>` header file. See [Header files for Pthread functions](#).
- [pthread_mutexattr_setkind_np\(\)--Set Mutex Kind Attribute](#)
- [pthread_mutex_init\(\)--Initialize Mutex](#)

pthread_mutex_destroy()--Destroy Mutex

Syntax

```
#include <pthread.h>
int pthread_mutex_destroy(pthread_mutex_t *mutex);
Threadsafe: Yes
Signal Safe: Yes
```

The **pthread_mutex_destroy()** function destroys the named mutex. The destroyed mutex can no longer be used.

If **pthread_mutex_destroy()** is called on a mutex that is locked by another thread, the request fails with an **EBUSY** error. If the calling thread has the mutex locked, any other threads waiting for the mutex via a call to **pthread_mutex_lock()** at the time of the call to **pthread_mutex_destroy()** fails with the **EDESTROYED** error.

Mutex initialization using the **PTHREAD_MUTEX_INITIALIZER** does not immediately initialize the mutex. Instead, on first use, **pthread_mutex_lock()** or **pthread_mutex_trylock()** branches into a slow path and causes the initialization of the mutex. Because a mutex is not just a simple memory object and requires that some resources be allocated by the system, an attempt to call **pthread_mutex_destroy()** or **pthread_mutex_unlock()** on a mutex that has statically initialized using **PTHREAD_MUTEX_INITIALIZER** and was not yet locked causes an **EINVAL** error.

Every mutex must eventually be destroyed with **pthread_mutex_destroy()**. The machine eventually detects the error if a mutex is not destroyed, but the storage is deallocated or corrupted. The machine then creates LIC log synchronization entries that indicate the failure to help debug the problem. Large numbers of these entries can affect system performance and hinder debug capabilities for other system problems. Always use **pthread_mutex_destroy()** before freeing mutex storage to prevent these debug LIC log entries.

Once a mutex is created, it cannot be validly copied or moved to a new location.

Parameters

mutex

(Input) Address of the mutex to be destroyed

Authorities and Locks

None.

Return Value

0

pthread_mutex_destroy() was successful.

value

pthread_mutex_destroy() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_mutex_destroy()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EBUSY]

The mutex is currently owned by another thread.

[EINVAL]

The value specified for the argument is not correct.

Related Information

- The `<pthread.h>` header file. See [Header files for Pthread functions](#).
- [pthread_mutex_init\(\)--Initialize Mutex](#)
- [pthread_mutex_lock\(\)--Lock Mutex](#)
- [pthread_mutex_trylock\(\)--Lock Mutex with No Wait](#)
- [pthread_mutex_unlock\(\)--Unlock Mutex](#)

Example

```
#include <pthread.h>
#include <stdio.h>
#include "check.h"

pthread_mutex_t    mutex;

int main(int argc, char **argv)
{
    int                rc=0;
    pthread_mutexattr_t mta;

    printf("Entering testcase\n");

    printf("Create the mutex using the NULL attributes (default)\n");
    rc = pthread_mutex_init(&mutex, NULL);
    checkResults("pthread_mutex_init(NULL)\n", rc);

    printf("Destroy all mutexes\n");

    pthread_mutex_destroy(&mutex);
    checkResults("pthread_mutex_destroy()\n", rc);

    printf("Main completed\n");
    return 0;
}
```

Output:

```
Entering testcase
Create the mutex using the NULL attributes (default)
Destroy all mutexes
Main completed
```


pthread_mutex_init()--Initialize Mutex

Syntax

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *mutex,
                      const pthread_mutexattr_t *attr);

pthread_mutex_t  mutex = PTHREAD_MUTEX_INITIALIZER;
Threadsafe: Yes
Signal Safe: Yes
```

The **pthread_mutex_init()** function initializes a mutex with the specified attributes for use. The new mutex may be used immediately for serializing critical resources. If *attr* is specified as **NULL**, all attributes are set to the default mutex attributes for the newly created mutex.

With these declarations and initialization:

```
pthread_mutex_t      mutex2;
pthread_mutex_t      mutex3;
pthread_mutexattr_t  mta;
pthread_mutexattr_init(&mta);
```

The following three mutex initialization mechanisms have equivalent function.

```
pthread_mutex_t      mutex1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_init(&mutex2, NULL);
pthread_mutex_init(&mutex3, &mta);
```

All three mutexes are created with the default mutex attributes.

Every mutex must eventually be destroyed with **pthread_mutex_destroy()**. The machine eventually detects the error if a mutex is not destroyed. Large numbers of these entries can affect system performance. Always use **pthread_mutex_destroy()** before freeing or reusing mutex storage.

Once a mutex is created, it cannot be validly copied or moved to a new location. If the mutex is copied or moved to a new location, the new object is not valid and cannot be used. Attempts to use the new object result in the EINVAL error.

*Mutex initialization using the **PTHREAD_MUTEX_INITIALIZER** does not immediately initialize the mutex. Instead, on first use, the **pthread_mutex_lock()** or **pthread_mutex_trylock()** functions branch into a slow path and cause the initialization of the mutex. Because a mutex is not just a simple memory object and requires that some resources be allocated by the system, an attempt to call **pthread_mutex_destroy()** or **pthread_mutex_unlock()** on a mutex that was statically initialized using **PTHREAD_MUTEX_INITIALIZER** and was not yet locked causes an **EINVAL** error.*

Parameters

mutex

(Input) The address of the variable to contain a mutex object.

attr

(Input) The address of the variable containing the mutex attributes object.

Authorities and Locks

None.

Return Value

0

pthread_mutex_init() was successful.

value

pthread_mutex_init() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_mutex_init()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

[ENOMEM]

The system cannot allocate the resources required to create the mutex.

Related Information

- The **<pthread.h>** header file. See [Header files for Pthread functions](#).
- [pthread_mutex_destroy\(\)--Destroy Mutex](#)
- [pthread_mutex_lock\(\)--Lock Mutex](#)
- [pthread_mutex_trylock\(\)--Lock Mutex with No Wait](#)
- [pthread_mutex_unlock\(\)--Unlock Mutex](#)

Example

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include "check.h"

pthread_mutex_t    mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t    mutex2;
pthread_mutex_t    mutex3;

int main(int argc, char **argv)
{
    int                rc=0;
    pthread_mutexattr_t mta;

    printf("Enter Testcase - %s\n", argv[0]);

    printf("Create a default mutex attribute\n");
    rc = pthread_mutexattr_init(&mta);
    checkResults("pthread_mutexattr_init\n", rc);

    printf("Create the mutexes using the default mutex attributes\n");

    printf("First mutex created via static PTHREAD_MUTEX_INITIALIZER\n");

    printf("Create the mutex using the NULL attributes (default)\n");
    rc = pthread_mutex_init(&mutex3, NULL);
```

pthread_mutex_init()--Initialize Mutex

```
    checkResults("pthread_mutex_init(NULL)\n", rc);

    printf("Create the mutex using a mutex attributes object\n");
    rc = pthread_mutex_init(&mutex2, &mta);
    checkResults("pthread_mutex_init(mta)\n", rc);

    printf("- At this point, all mutexes can be used with their\n");
    printf("- default attributes from any threads that want to\n");
    printf("- use them\n");

    printf("Destroy all mutexes\n");
    pthread_mutex_destroy(&mutex);
    pthread_mutex_destroy(&mutex2);
    pthread_mutex_destroy(&mutex3);

    printf("Main completed\n");
    return 0;
}
```

Output:

```
Enter Testcase - QP0WTEST/TPMTXINIO
Create a default mutex attribute
Create the mutexes using the default mutex attributes
First mutex created via static PTHREAD_MUTEX_INITIALIZER
Create the mutex using the NULL attributes (default)
Create the mutex using a mutex attributes object
- At this point, all mutexes can be used with their
- default attributes from any threads that want to
- use them
Destroy all mutexes
Main completed
```

pthread_mutex_lock()--Lock Mutex

Syntax

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mutex);
Threadsafe: Yes
Signal Safe: Yes
```

The **pthread_mutex_lock()** function acquires ownership of the mutex specified. If the mutex is currently locked by another thread, the call to **pthread_mutex_lock()** blocks until that thread relinquishes ownership via a call to **pthread_mutex_unlock()**.

If a signal is delivered to a thread while that thread is waiting for a mutex, when the signal handler returns, the wait resumes. **pthread_mutex_lock()** does not return **EINTR** like some other blocking function calls.

Use the CL command WRKJOB, option 20, to help you debug mutex deadlocks.

Destroying a held mutex is a common way to serialize destruction of objects that are protected by that mutex. This action is allowed. The call to **pthread_mutex_lock()** may fail with the **EDESTROYED** error if the mutex is destroyed by the thread that was currently holding it.

Mutex initialization using the **PTHREAD_MUTEX_INITIALIZER** does not immediately initialize the mutex. Instead, on first use, **pthread_mutex_timedlock_np()** or **pthread_mutex_lock()** or **pthread_mutex_trylock()** branches into a slow path and causes the initialization of the mutex. Because a mutex is not just a simple memory object and requires that some resources be allocated by the system, an attempt to call **pthread_mutex_destroy()** or **pthread_mutex_unlock()** on a mutex that was statically initialized using **PTHREAD_MUTEX_INITIALIZER** and was not yet locked causes an **EINVAL** error.

A pthread mutex is a structure of type **pthread_mutex_t** that implement a behavior based on the Pthread mutexes. An MI mutex is a structure built into the machine that implement a similar sort of serialization construct.

The maximum number of recursive locks by the owning thread is 32,767. When this number is exceeded, attempts to lock the mutex return the **ERECURSE** error.

Mutex Types

A normal mutex cannot be locked repeatedly by the owner. Attempts by a thread to relock an already held mutex, or to lock a mutex that was held by another thread when that thread terminated, result in a deadlock condition.

A recursive mutex can be locked repeatedly by the owner. The mutex does not become unlocked until the owner has called **pthread_mutex_unlock()** for each successful lock request that it has outstanding on the mutex.

An errorcheck mutex checks for deadlock conditions that occur when a thread relocks an already held mutex. If a thread attempts to relock a mutex that it already holds, the lock request fails with the **EDEADLK** error.

An ownerterm mutex is an OS/400 extension to the errorcheck mutex type. An ownerterm mutex checks for deadlock conditions that occur when a thread relocks an already held mutex. If a thread attempts to relock a mutex that it already holds, the lock request fails with the **EDEADLK** error. An ownerterm mutex also checks for deadlock conditions that occur when a thread attempts to lock a mutex that was held by another thread when that thread terminated (an orphaned mutex). If a thread attempts to lock an orphaned mutex, the lock request fails with the **EOWNERTERM** error.

When a thread terminates the holding of a mutex lock on a normal or errorcheck mutex, other threads that wait for that mutex will block forever. The pthreads run-time simulates the deadlock that has occurred in your application. When you are attempting to debug these deadlock scenarios, the CL command WRKJOB, option 20 shows the thread as in a condition wait. Displaying the call stack shows that the function **deadlockedOnOrphanedMutex** is in the call stack.

When a thread attempts to acquire a normal mutex that it already holds, the thread will block forever. The pthreads

run-time simulates the deadlock that has occurred in your application. When you are attempting to debug these deadlock scenarios, the CL command WRKJOB, option 20, shows the thread as in a condition wait. Displaying the call stack will show that the function **deadlockedOnAlreadyLockedMutex** is in the call stack.

To change these behaviors, use an errorcheck or ownerterm mutex type.

Parameters

mutex

(Input) The address of the mutex to lock

Authorities and Locks

None.

Return Value

0

pthread_mutex_lock() was successful.

value

pthread_mutex_lock() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_mutex_lock()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

[EDESTROYED]

While waiting for the mutex lock to be satisfied, the mutex was destroyed.

[EOWNERTERM]

A thread terminated the holding of the mutex, and the mutex is an ownerterm mutex type.

[EDEADLK]

A thread attempted to relock an already held mutex, and the mutex is an errorcheck mutex type.

[ERECURSE]

The recursive mutex cannot be recursively locked again.

Related Information

- The <**pthread.h**> header file. See [Header files for Pthread functions](#).
- [pthread_mutex_destroy\(\)--Destroy Mutex](#)
- [pthread_mutex_init\(\)--Initialize Mutex](#)
- [pthread_mutex_trylock\(\)--Lock Mutex with No Wait](#)
- [pthread_mutex_unlock\(\)--Unlock Mutex](#)

Example

```

#include <pthread.h>
#include <stdio.h>
#include "check.h"
/*
   This example shows the corruption that can result if no
   serialization is done and also shows the use of
   pthread_mutex_lock(). Call it with no parameters
   to use pthread_mutex_lock() to protect the critical section,
   or 1 or more parameters to show data corruption that occurs
   without locking.
*/
#define LOOPCONSTANT 100000
#define THREADS 10

pthread_mutex_t  mutex = PTHREAD_MUTEX_INITIALIZER;
int              i,j,k,l;
int              uselock=1;

void *threadfunc(void *parm)
{
    int  loop = 0;
    int  rc;

    for (loop=0; loop<LOOPCONSTANT; ++loop) {
        if (uselock) {
            rc = pthread_mutex_lock(&mutex);
            checkResults("pthread_mutex_lock()\n", rc);
        }
        ++i; ++j; ++k; ++l;
        if (uselock) {
            rc = pthread_mutex_unlock(&mutex);
            checkResults("pthread_mutex_unlock()\n", rc);
        }
    }
    return NULL;
}

int main(int argc, char **argv)
{
    pthread_t      threadid[THREADS];
    int            rc=0;

    int            loop=0;
    pthread_attr_t pta;

    printf("Entering testcase\n");
    printf("Give any number of parameters to show data corruption\n");
    if (argc != 1) {
        printf("A parameter was specified, no serialization is being done!\n");
        uselock = 0;
    }

    pthread_attr_init(&pta);
    pthread_attr_setdetachstate(&pta, PTHREAD_CREATE_JOINABLE);

    printf("Creating %d threads\n", THREADS);
    for (loop=0; loop<THREADS; ++loop) {
        rc = pthread_create(&threadid[loop], &pta, threadfunc, NULL);

```

pthread_mutex_lock()--Lock Mutexe

```
    checkResults("pthread_create()\n", rc);
}

printf("Wait for results\n");
for (loop=0; loop<THREADS; ++loop) {
    rc = pthread_join(threadid[loop], NULL);
    checkResults("pthread_join()\n", rc);
}

printf("Cleanup and show results\n");
pthread_attr_destroy(&pta);
pthread_mutex_destroy(&mutex);

printf("\nUsing %d threads and LOOPCONSTANT = %d\n",
        THREADS, LOOPCONSTANT);
printf("Values are: (should be %d)\n", THREADS * LOOPCONSTANT);
printf("  ==>%d, %d, %d, %d\n", i, j, k, l);

printf("Main completed\n");
return 0;
}
```

Output:

```
Entering testcase
Give any number of parameters to show data corruption
Creating 10 threads
Wait for results
Cleanup and show results
```

```
Using 10 threads and LOOPCONSTANT = 100000
Values are: (should be 1000000)
  ==>1000000, 1000000, 1000000, 1000000
Main completed
```

Output:

(data corruption without locking example)

```
Entering testcase
Give any number of parameters to show data corruption
A parameter was specified, no serialization is being done!
Creating 10 threads
Wait for results
Cleanup and show results
```

```
Using 10 threads and LOOPCONSTANT = 100000
Values are: (should be 1000000)
  ==>883380, 834630, 725131, 931883
Main completed
```

pthread_mutex_timedlock_np()--Lock Mutex with Time-Out

Syntax

```
#include <pthread.h>
int pthread_mutex_timedlock_np(pthread_mutex_t *mutex,
                               const struct timespec *deltatime);
```

Threadsafe: Yes

Signal Safe: Yes

The **pthread_mutex_timedlock_np()** function acquires ownership of the *mutex* specified. If the mutex is currently locked by another thread, the call to **pthread_mutex_timedlock_np()** will block until the specified *deltatime* has elapsed or the holding thread relinquishes ownership via a call to **pthread_mutex_unlock()**.

Performing a **pthread_mutex_timedlock_np()** wait for a mutex has different semantics related to signal handling than the **pthread_mutex_lock()** function. If a signal is delivered to a thread while that thread is performing a timed wait for a mutex, the signal is held pending until either the mutex is acquired or the time-out occurs. At that time the signal handler will run, when the signal handler returns, **pthread_mutex_timedlock()** will return the results of the timed mutex wait.

Use the CL command WRKJOB, option 20 for a screen that will aid in debugging mutex deadlocks.

Destroying a held mutex is a common way to serialize destruction of objects that are protected by that mutex, and is allowed. The call to **pthread_mutex_timedlock_np()** may fail with the **EDESTROYED** error if the mutex is destroyed by the thread that was currently holding it.

Note that mutex initialization using the **PTHREAD_MUTEX_INITIALIZER** does not immediately initialize the mutex. Instead, on first use, **pthread_mutex_timedlock_np()**, **pthread_mutex_lock()** or **pthread_mutex_trylock()** branches into a slow path and causes the initialization of the mutex. Because a mutex is not just a simple memory object, and requires that some resources be allocated by the system, an attempt to call **pthread_mutex_destroy()** or **pthread_mutex_unlock()** on a mutex that has was statically initialized using **PTHREAD_MUTEX_INITIALIZER** and was not yet locked will result in an **EINVAL** error.

A pthread mutex is a structure of type `pthread_mutex_t` that implement a behavior based on the Pthread mutexes. An MI mutex is a structure built into the machine that implement a similar sort of serialization construct.

The maximum number of recursive locks by the owning thread is 32,767. After which, attempts to lock the mutex will return the **ERECURSE** error.

This function is not portable

Mutex Types

A normal mutex cannot be locked repeatedly by the owner. Attempts by a thread to relock an already held mutex, or to lock a mutex that was held by another thread when that thread terminated result in a deadlock condition.

A recursive mutex can be locked repeatedly by the owner. The mutex does not become unlocked until the owner has called **pthread_mutex_unlock()** for each successful lock request that it has outstanding on the mutex.

An errorcheck mutex checks for deadlock conditions that occur when a thread re-locks an already held mutex. If a thread attempts to relock a mutex that it already holds, the lock request fails with the **EDEADLK** error

An ownerterm mutex is an OS/400 extension to the errorcheck mutex type. An ownerterm mutex checks for deadlock conditions that occur when a thread re-locks an already held mutex. If a thread attempts to relock a mutex that it already holds, the lock request fails with the **EDEADLK** error. An ownerterm mutex also checks for deadlock conditions that occur when a thread attempts to lock a mutex that was held by another thread when that thread terminated (an orphaned mutex). If a thread attempts to lock an orphaned mutex, the lock request fails with the **EOWNERTERM** error.

When a thread terminates holding a mutex lock on a normal or errorcheck mutex, other threads that wait for that mutex will block forever. The pthreads run-time simulates the deadlock that has occurred in your application. When attempting to debug these deadlock scenarios, the CL command WRKJOB, option 20 will show the thread as in a condition wait. Displaying the call stack will show that the function **deadlockedOnOrphanedMutex** is in the call stack.

When a thread attempts to acquire a normal mutex that it already holds, the thread will block forever. The pthreads run-time simulates the deadlock that has occurred in your application. When attempting to debug these deadlock scenarios, the CL command WRKJOB, option 20 will show the thread as in a condition wait. Displaying the call stack will show that the function **deadlockedOnAlreadyLockedMutex** is in the call stack.

In order to change these behaviors, use an errorcheck or ownerterm mutex type.

Parameters

mutex

(Input) The address of the mutex to lock

Authorities and Locks

None.

Return Value

0

pthread_mutex_timedlock_np() was successful.

value

pthread_mutex_timedlock_np() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_mutex_timedlock_np()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

[EDESTROYED]

While waiting for the mutex lock to be satisfied, the mutex was destroyed.

[EBUSY]

The attempt to lock the mutex timed out because the mutex was already locked.

[EOWNERTERM]

A thread terminated holding the mutex, and the mutex is an ownerterm mutex type.

[EDEADLK]

A thread attempted to relock an already held mutex, and the mutex is an errorcheck mutex type.

[ERECURSE]

The recursive mutex cannot be recursively locked again.

Related Information

- The `<pthread.h>` header file. See [Header files for Pthread functions](#).
- [pthread_mutex_destroy\(\)--Destroy Mutex](#)
- [pthread_mutex_init\(\)--Initialize Mutex](#)
- [pthread_mutex_lock\(\)--Lock Mutex](#)
- [pthread_mutex_trylock\(\)--Lock Mutex with No Wait](#)
- [pthread_mutex_unlock\(\)--Unlock Mutex](#)

Example

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include "check.h"

pthread_mutex_t      mutex = PTHREAD_MUTEX_INITIALIZER;

void *threadFunc(void *parm)
{
    int          rc;
    int          i;
    struct timespec deltatime;

    deltatime.tv_sec = 5;
    deltatime.tv_nsec = 0;

    printf("Timed lock the mutex from a secondary thread\n");
    rc = pthread_mutex_timedlock_np(&mutex, &deltatime);
    if (rc != EBUSY) {
        printf("Got an incorrect return code from
pthread_mutex_timedlock_np\n");
    }
    printf("Thread mutex timeout\n");
    return 0;
}

int main(int argc, char **argv)
{
    int          rc=0;
    pthread_t      thread;

    printf("Enter Testcase - %s\n", argv[0]);

    printf("Acquire the mutex in the initial thread\n");
    rc = pthread_mutex_lock(&mutex);
    checkResults("pthread_mutex_lock()\n", rc),

    printf("Create a thread\n");
    rc = pthread_create(&thread, NULL, threadFunc, NULL);
    checkResults("pthread_create()\n", rc);

    printf("Join to the thread\n");
    rc = pthread_join(thread, NULL);
    checkResults("pthread_join()\n", rc);
}
```

pthread_mutex_timedlock_np())--Lock Mutex with Time-Out

```
printf("Destroy mutex\n");  
pthread_mutex_destroy(&mutex);  
  
printf("Main completed\n");  
return 0;  
}
```

Output:

```
Enter Testcase - QP0WTEST/TPMTXTIM0  
Acquire the mutex in the initial thread  
Create a thread  
Join to the thread  
Timed lock the mutex from a secondary thread  
Thread mutex timeout  
Destroy mutex  
Main completed
```

pthread_mutex_trylock()--Lock Mutex with No Wait

Syntax

```
#include <pthread.h>
int pthread_mutex_trylock(pthread_mutex_t *mutex);
Threadsafe: Yes
Signal Safe: Yes
```

The **pthread_mutex_trylock()** function attempts to acquire ownership of the mutex specified without blocking the calling thread. If the mutex is currently locked by another thread, the call to **pthread_mutex_trylock()** returns an error of **EBUSY**.

A failure of **EDEADLK** indicates that the mutex is already held by the calling thread.

Mutex initialization using the **PTHREAD_MUTEX_INITIALIZER** does not immediately initialize the mutex. Instead, on first use, **pthread_mutex_timedlock_np()** or **pthread_mutex_lock()** or **pthread_mutex_trylock()** branches into a slow path and causes the initialization of the mutex. Because a mutex is not just a simple memory object and requires that some resources be allocated by the system, an attempt to call **pthread_mutex_destroy()** or **pthread_mutex_unlock()** on a mutex that was statically initialized using **PTHREAD_MUTEX_INITIALIZER** and was not yet locked causes an **EINVAL** error.

The maximum number of recursive locks by the owning thread is 32,767. When this number is exceeded, attempts to lock the mutex return the **ERECURSE** error.

Mutex Types

A normal mutex cannot be locked repeatedly by the owner. Attempts by a thread to relock an already held mutex, or to lock a mutex that was held by another thread when that thread terminated, cause a deadlock condition.

A recursive mutex can be locked repeatedly by the owner. The mutex does not become unlocked until the owner has called **pthread_mutex_unlock()** for each successful lock request that it has outstanding on the mutex.

An errorcheck mutex checks for deadlock conditions that occur when a thread relocks an already held mutex. If a thread attempts to relock a mutex that it already holds, the lock request fails with the **EDEADLK** error.

An ownerterm mutex is an OS/400 extension to the errorcheck mutex type. An ownerterm mutex checks for deadlock conditions that occur when a thread relocks an already held mutex. If a thread attempts to relock a mutex that it already holds, the lock request fails with the **EDEADLK** error. An ownerterm mutex also checks for deadlock conditions that occur when a thread attempts to lock a mutex that was held by another thread when that thread terminated (an orphaned mutex). If a thread attempts to lock an orphaned mutex, the lock request fails with the **EOWNERTERM** error.

When a thread terminates the holding of a mutex lock on a normal or errorcheck mutex, other threads that wait for that mutex will block forever. The pthreads run-time simulates the deadlock that has occurred in your application. When you are attempting to debug these deadlock scenarios, the CL command **WRKJOB**, option 20, shows the thread as in a condition wait. Displaying the call stack shows that the function **deadlockedOnOrphanedMutex** is in the call stack.

When a thread attempts to acquire a normal mutex that it already holds, the thread will block forever. The pthreads run-time simulates the deadlock that has occurred in your application. When you are attempting to debug these deadlock scenarios, the CL command **WRKJOB**, option 20, shows the thread as in a condition wait. Displaying the call stack shows that the function **deadlockedOnAlreadyLockedMutex** is in the call stack.

To change these behaviors, use an errorcheck or ownerterm mutex type.

Parameters

mutex

(Input) Address of the mutex to lock

Authorities and Locks

None.

Return Value

0

pthread_mutex_trylock() was successful.

value

pthread_mutex_trylock() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_mutex_trylock()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

[EBUSY]

The mutex is currently locked by another thread.

A thread terminated the holding of the mutex, and the mutex is an ownerterm mutex type.

A thread attempted to relock an already held mutex, and the mutex is an errorcheck mutex type.

[ERECURSE]

The recursive mutex cannot be recursively locked again.

Related Information

- The `<pthread.h>` header file. See [Header files for Pthread functions.](#)
- [pthread_mutex_destroy\(\)--Destroy Mutex](#)
- [pthread_mutex_init\(\)--Initialize Mutex](#)
- [pthread_mutex_lock\(\)--Lock Mutex](#)
- [pthread_mutex_timedlock_np\(\)--Lock Mutex with Time-Out](#)
- [pthread_mutex_unlock\(\)--Unlock Mutex](#)

Example

```
#include <pthread.h>
#include <stdio.h>
#include <errno.h>
#include "check.h"
```

```
/*
```

```
   This example simulates a number of threads working on a parallel
   problem. The threads use pthread_mutex_trylock() so that
```

they do not spend time blocking on a mutex and instead spend more of the time making progress towards the final solution. When trylock fails, the processing is done locally, eventually to be merged with the final parallel solution.

This example should complete faster than the example for pthread_mutex_lock() in which threads solve the same parallel problem but spend more time waiting in resource contention.

```

*/
#define          LOOPCONSTANT      100000
#define          THREADS          10

pthread_mutex_t  mutex = PTHREAD_MUTEX_INITIALIZER;
int             i,j,k,l;

void *threadfunc(void *parm)
{
    int    loop = 0;
    int    localProcessingCompleted = 0;
    int    numberOfLocalProcessingBursts = 0;
    int    processingCompletedThisBurst = 0;
    int    rc;

    for (loop=0; loop<LOOPCONSTANT; ++loop) {
        rc = pthread_mutex_trylock(&mutex);
        if (rc == EBUSY) {
            /* Process continue processing the part of the problem */
            /* that we can without the lock. We do not want to waste */
            /* time blocking. Instead, we'll count locally.          */
            ++localProcessingCompleted;
            ++numberOfLocalProcessingBursts;
            continue;
        }
        /* We acquired the lock, so this part of the can be global*/
        checkResults("pthread_mutex_trylock()\n", rc);
        /* Processing completed consist of last local processing */
        /* plus the 1 unit of processing this time through        */
        processingCompletedThisBurst = 1 + localProcessingCompleted;
        localProcessingCompleted = 0;
        i+=processingCompletedThisBurst; j+=processingCompletedThisBurst;
        k+=processingCompletedThisBurst; l+=processingCompletedThisBurst;

        rc = pthread_mutex_unlock(&mutex);
        checkResults("pthread_mutex_unlock()\n", rc);
    }
    /* If any local processing remains, merge it with the global*/
    /* problem so our part of the solution is accounted for      */
    if (localProcessingCompleted) {
        rc = pthread_mutex_lock(&mutex);
        checkResults("final pthread_mutex_lock()\n", rc);

        i+=localProcessingCompleted; j+=localProcessingCompleted;
        k+=localProcessingCompleted; l+=localProcessingCompleted;

        rc = pthread_mutex_unlock(&mutex);
        checkResults("final pthread_mutex_unlock()\n", rc);
    }
    printf("Thread processed about %d%% of the problem locally\n",
           (numberOfLocalProcessingBursts * 100) / LOOPCONSTANT);
    return NULL;
}

```

```

int main(int argc, char **argv)
{
    pthread_t          threadid[THREADS];
    int                rc=0;
    int                loop=0;
    pthread_attr_t      pta;

    printf("Entering testcase\n");

    pthread_attr_init(&pta);
    pthread_attr_setdetachstate(&pta, PTHREAD_CREATE_JOINABLE);

    printf("Creating %d threads\n", THREADS);
    for (loop=0; loop<THREADS; ++loop) {
        rc = pthread_create(&threadid[loop], &pta, threadfunc, NULL);
        checkResults("pthread_create()\n", rc);
    }

    printf("Wait for results\n");
    for (loop=0; loop<THREADS; ++loop) {
        rc = pthread_join(threadid[loop], NULL);
        checkResults("pthread_join()\n", rc);
    }

    printf("Cleanup and show results\n");
    pthread_attr_destroy(&pta);
    pthread_mutex_destroy(&mutex);

    printf("\nUsing %d threads and LOOPCONSTANT = %d\n",
          THREADS, LOOPCONSTANT);
    printf("Values are: (should be %d)\n", THREADS * LOOPCONSTANT);
    printf("  ==>%d, %d, %d, %d\n", i, j, k, l);

    printf("Main completed\n");
    return 0;
}

```

Output:

```

Entering testcase
Creating 10 threads
Wait for results
Thread processed about 100% of the problem locally
Thread processed about 90% of the problem locally
Thread processed about 88% of the problem locally
Thread processed about 94% of the problem locally
Thread processed about 93% of the problem locally
Thread processed about 96% of the problem locally
Thread processed about 90% of the problem locally
Thread processed about 91% of the problem locally
Thread processed about 81% of the problem locally
Thread processed about 76% of the problem locally
Cleanup and show results

Using 10 threads and LOOPCONSTANT = 100000
Values are: (should be 1000000)
  ==>1000000, 1000000, 1000000, 1000000
Main completed

```

pthread_mutex_unlock()--Unlock Mutex

Syntax

```
#include <pthread.h>
int pthread_mutex_unlock(pthread_mutex_t *mutex);
Threadsafe: Yes
Signal Safe: Yes
```

The **pthread_mutex_unlock()** function unlocks the mutex specified. If the calling thread does not currently hold the mutex (via a previous call to **pthread_mutex_lock()** or **pthread_mutex_trylock()**) the unlock request fails with the **EPERM** error.

Mutex initialization using the **PTHREAD_MUTEX_INITIALIZER** does not immediately initialize the mutex. Instead, on first use, **pthread_mutex_lock()** or **pthread_mutex_trylock()** branches into a slow path and causes the initialization of the mutex. Because a mutex is not just a simple memory object and requires that some resources be allocated by the system, an attempt to call **pthread_mutex_destroy()** or **pthread_mutex_unlock()** on a mutex that was statically initialized using **PTHREAD_MUTEX_INITIALIZER** and was not yet locked causes an **EINVAL** error.

Mutex Types

A normal mutex cannot be locked repeatedly by the owner. Attempts by a thread to relock an already held mutex, or to lock a mutex that was held by another thread when that thread terminated, cause a deadlock condition.

A recursive mutex can be locked repeatedly by the owner. The mutex does not become unlocked until the owner has called **pthread_mutex_unlock()** for each successful lock request that it has outstanding on the mutex.

An errorcheck mutex checks for deadlock conditions that occur when a thread relocks an already held mutex. If a thread attempts to relock a mutex that it already holds, the lock request fails with the **EDEADLK** error.

An ownerterm mutex is an OS/400 extension to the errorcheck mutex type. An ownerterm mutex checks for deadlock conditions that occur when a thread relocks an already held mutex. If a thread attempts to relock a mutex that it already holds, the lock request fails with the **EDEADLK** error. An ownerterm mutex also checks for deadlock conditions that occur when a thread attempts to lock a mutex that was held by another thread when that thread terminated (an orphaned mutex). If a thread attempts to lock an orphaned mutex, the lock request fails with the **EOWNERTERM** error.

When a thread terminates the holding of a mutex lock on a normal or errorcheck mutex, other threads that wait for that mutex will block forever. The pthreads run-time simulates the deadlock that has occurred in your application. When you are attempting to debug these deadlock scenarios, the CL command WRKJOB, option 20, shows the thread as in a condition wait. Displaying the call stack shows that the function **deadlockedOnOrphanedMutex** is in the call stack.

When a thread attempts to acquire a normal mutex that it already holds, the thread will block forever. The pthreads run-time simulates the deadlock that has occurred in your application. When you are attempting to debug these deadlock scenarios, the CL command WRKJOB, option 20, shows the thread as in a condition wait. Displaying the call stack shows that the function **deadlockedOnAlreadyLockedMutex** is in the call stack.

To change these behaviors, use an errorcheck or ownerterm mutex type.

Parameters

mutex

(Input) Address of the mutex to unlock

Authorities and Locks

For successful completion, the mutex lock must be held before you call **pthread_mutex_unlock()**.

Return Value

0

pthread_mutex_unlock() was successful.

value

pthread_mutex_unlock() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_mutex_unlock()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

[EPERM]

The mutex is not currently held by the caller

Related Information

- The `<pthread.h>` header file. See [Header files for Pthread functions](#).
- [pthread_mutex_destroy\(\)--Destroy Mutex](#)
- [pthread_mutex_init\(\)--Initialize Mutex](#)
- [pthread_mutex_lock\(\)--Lock Mutex](#)
- [pthread_mutex_trylock\(\)--Lock Mutex with No Wait](#)

Example

```
#include <pthread.h>
#include <stdio.h>
#include "check.h"

pthread_mutex_t    mutex = PTHREAD_MUTEX_INITIALIZER;

int main(int argc, char **argv)
{
    int                rc=0;

    printf("Entering testcase\n");

    printf("Lock the mutex\n");
    rc = pthread_mutex_lock(&mutex);
    checkResults("pthread_mutex_lock()\n", rc);

    /* All other threads will be blocked from the resource here */

    printf("Unlock the mutex\n");
    rc = pthread_mutex_unlock(&mutex);
    checkResults("pthread_mutex_unlock()\n", rc);
}
```

pthread_mutex_unlock()--Unlock Mutex

```
printf("Destroy the mutex\n");  
rc = pthread_mutex_destroy(&mutex);  
checkResults("pthread_mutex_destroy()\n", rc);  
  
printf("Main completed\n");  
return 0;  
}
```

Output:

```
Entering testcase  
Lock the mutex  
Unlock the mutex  
Destroy the mutex  
Main completed
```

pthread_lock_global_np()--Lock Global Mutex

Syntax

```
#include <pthread.h>
#include <sched.h>
int pthread_lock_global_np(void);
Threadsafe: Yes
Signal Safe: Yes
```

The **pthread_lock_global_np()** function locks a global mutex provided by the pthreads run-time. The global mutex is a recursive mutex with a name of "QP0W_GLOBAL_MTX". The global mutex is not currently used by the pthreads run-time to serialize access to any system resources, and is provided for application use only.

The maximum number of recursive locks by the owning thread is 32,767. After which, attempts to lock the mutex will return the **ERECURSE** error.

This function is not portable

Parameters

None.

Authorities and Locks

None.

Return Value

0

pthread_lock_global_np() was successful.

value

pthread_lock_global_np() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_lock_global_np()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

[ERECURSE]

The recursive mutex cannot be recursively locked again.

Related Information

- The **<pthread.h>** header file. See [Header files for Pthread functions](#).
- [pthread_unlock_global_np\(\)--Unlock Global Mutex](#)

Example

```
#include <pthread.h>
#include <stdio.h>
#include "check.h"
/*
   This example shows the corruption that can result if no
   serialization is done and also shows the use of
   pthread_lock_global_np(). Call this test with no parameters
   to use pthread_lock_gloabl_np() to protect the critical data,
   between more than one (possibly unrelated) functions.
   Use 1 or more parameters to skip locking and
   show data corruption that occurs without locking.
*/
#define LOOPCONSTANT 50000
#define THREADS 10

int i,j,k,l;
int uselock=1;

void secondFunction(void)
{
    int rc;
    if (uselock) {
        rc = pthread_lock_global_np();
        checkResults("pthread_lock_global_np()\n", rc);
    }
    --i; --j; --k; --l;
    if (uselock) {
        rc = pthread_unlock_global_np();
        checkResults("pthread_unlock_global_np()\n", rc);
    }
}

void *threadfunc(void *parm)
{
    int loop = 0;
    int rc;

    for (loop=0; loop<LOOPCONSTANT; ++loop) {
        if (uselock) {
            rc = pthread_lock_global_np();
            checkResults("pthread_lock_global_np()\n", rc);
        }
        ++i; ++j; ++k; ++l;
        secondFunction();
        ++i; ++j; ++k; ++l;
        if (uselock) {
            rc = pthread_unlock_global_np();
            checkResults("pthread_unlock_global_np()\n", rc);
        }
    }
    return NULL;
}

int main(int argc, char **argv)
{
    pthread_t threadid[THREADS];
    int rc=0;
    int loop=0;
```

```

printf("Enter Testcase - %s\n", argv[0]);
printf("Give any number of parameters to show data corruption\n");
if (argc != 1) {
    printf("A parameter was specified, no serialization is being done!\n");
    uselock = 0;
}

if (uselock) {
    rc = pthread_lock_global_np();
    checkResults("pthread_lock_global_np() (main)\n", rc);
}

printf("Creating %d threads\n", THREADS);
for (loop=0; loop<THREADS; ++loop) {
    rc = pthread_create(&threadid[loop], NULL, threadfunc, NULL);
    checkResults("pthread_create()\n", rc);
}

sleep(5);
if (uselock) {
    rc = pthread_unlock_global_np();
    checkResults("pthread_unlock_global_np() (main)\n", rc);
}

printf("Wait for results\n");
for (loop=0; loop<THREADS; ++loop) {
    rc = pthread_join(threadid[loop], NULL);
    checkResults("pthread_join()\n", rc);
}

printf("\nUsing %d threads and LOOPCONSTANT = %d\n",
        THREADS, LOOPCONSTANT);
printf("Values are: (should be %d)\n", THREADS * LOOPCONSTANT);
printf("  ==>%d, %d, %d, %d\n", i, j, k, l);

printf("Main completed\n");
return 0;
}

```

Output:

```

Enter Testcase - QP0WTEST/TPMTXGLB0
Give any number of parameters to show data corruption
Creating 10 threads
Wait for results
Using 10 threads and LOOPCONSTANT = 50000
Values are: (should be 500000)
  ==>500000, 500000, 500000, 500000
Main completed

```

pthread_unlock_global_np()--Unlock Global Mutex

Syntax

```
#include <pthread.h>
#include <sched.h>
int pthread_lock_global_np(void);
Threadsafe: Yes
Signal Safe: Yes
```

The **pthread_unlock_global_np()** function unlocks a global mutex provided by the pthreads run-time. The global mutex is a recursive mutex with a name of "QP0W_GLOBAL_MTX". The global mutex is not currently used by the pthreads run-time to serialize access to any system resources, and is provided for application use only.

This function is not portable

Parameters

None.

Authorities and Locks

For successful completion, the global mutex lock must be held prior to calling **pthread_unlock_global_np()**.

Return Value

0

pthread_unlock_global_np() was successful.

value

pthread_unlock_global_np() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_unlock_global_np()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

[EPERM]

The mutex is not currently held by the caller.

Related Information

- The **<pthread.h>** header file. See [Header files for Pthread functions](#).
- [pthread_lock_global_np\(\)--Lock Global Mutex](#)

Example

See the [pthread_lock_global_np\(\)](#) example.

Condition variable synchronization APIs

Condition variables are synchronization objects that allow threads to wait for certain events (conditions) to occur. Condition variables are slightly more complex than mutexes, and the correct use of condition variables requires the thread to co-operatively use a specific protocol in order to ensure safe and consistent serialization. The protocol for using condition variables includes a mutex, a boolean predicate (true/false expression) and the condition variable itself. The threads that are cooperating using condition variables can wait for a condition to occur, or can wake up other threads that are waiting for a condition.

The table below lists important conditional variables attributes, their default values, and all supported values.

Attribute	Default value	supported values
<i>pshared</i>	PTHREAD_PROCESS_PRIVATE	PTHREAD_PROCESS_PRIVATE or PTHREAD_PROCESS_SHARED

For information about the examples included with the APIs, see the [information on the API examples](#).

To view the API list by description, see [Condition variable synchronization APIs](#).

Condition variable synchronization APIs list by name

- [pthread_condattr_destroy\(\)--Destroy Condition Variable Attributes Object](#)
- [pthread_condattr_getpshared\(\)--Get Process Shared Attribute from Condition Attributes Object](#)
- [pthread_condattr_init\(\)--Initialize Condition Variable Attributes Object](#)
- [pthread_condattr_setpshared\(\)--Set Process Shared Attribute in Condition Attributes Object](#)
- [pthread_cond_broadcast\(\)--Broadcast Condition to All Waiting Threads](#)
- [pthread_cond_destroy\(\)--Destroy Condition Variable](#)
- [pthread_cond_init\(\)--Initialize Condition Variable](#)
- [pthread_cond_signal\(\)--Signal Condition to One Waiting Thread](#)
- [pthread_cond_timedwait\(\)--Timed Wait for Condition](#)
- [pthread_cond_wait\(\)--Wait for Condition](#)
- [pthread_get_expiration_np\(\)--Get Condition Expiration Time from Relative Time](#)

pthread_condattr_destroy()--Destroy Condition Variable Attributes Object

Syntax

```
#include <pthread.h>
int pthread_condattr_destroy(pthread_condattr_t *attr);
Threadsafe: Yes
Signal Safe: Yes
```

The **pthread_condattr_destroy()** function destroys the condition variable attributes object specified by *attr*, and indicates that any storage that the system has associated with the object be de-allocated. Destroying a condition variable object in no way affects any of the condition variables that were created with that object.

Parameters

attr

(Input) The address of the condition variable attributes object to be destroyed

Authorities and Locks

None.

Return Value

0

pthread_condattr_destroy() was successful.

value

pthread_condattr_destroy() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_condattr_destroy()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

Related Information

- The **<pthread.h>** header file. See [Header files for Pthread functions](#).
- [pthread_condattr_init\(\)--Initialize Condition Variable Attributes Object](#)
- [pthread_cond_init\(\)--Initialize Condition Variable](#)

Example

```
#include <pthread.h>
#include <stdio.h>
#include "check.h"

pthread_cond_t      cond;
```



```

int main(int argc, char **argv)
{
    int                rc=0;
    pthread_condattr_t attr;

    printf("Entering testcase\n");

    printf("Create a default condition attribute\n");
    rc = pthread_condattr_init(&attr);
    checkResults("pthread_condattr_init\n", rc);

    printf("Create the condition using the condition attributes object\n");
    rc = pthread_cond_init(&cond, &attr);
    checkResults("pthread_cond_init()\n", rc);

    printf("- At this point, the condition with its default attributes\n");
    printf("- Can be used from any threads that want to use it\n");

    printf("Destroy cond attribute\n");
    rc = pthread_condattr_destroy(&attr);
    checkResults("pthread_condattr_destroy()\n", rc);

    printf("Destroy condition\n");
    rc = pthread_cond_destroy(&cond);
    checkResults("pthread_cond_destroy()\n", rc);

    printf("Main completed\n");
    return 0;
}

```

Output:

```

Entering testcase
Create a default condition attribute
Create the condition using the condition attributes object
- At this point, the condition with its default attributes
- Can be used from any threads that want to use it
Destroy cond attribute
Destroy condition
Main completed

```

pthread_condattr_getpshared()--Get Process Shared Attribute from Condition Attributes Object

Syntax

```
#include <pthread.h>
int pthread_condattr_getpshared(const pthread_condattr_t *attr, int
*pshared);
Threadsafe: Yes
Signal Safe: Yes
```

The **pthread_condattr_getpshared()** function retrieves the current setting of the process shared attribute from the condition attributes object. The process shared attribute indicates whether the condition that is created using the condition attributes object can be shared between threads in separate processes (**PTHREAD_PROCESS_SHARED**) or shared only between threads within the same process (**PTHREAD_PROCESS_PRIVATE**).

Even if the condition in storage is accessible from two separate processes, it cannot be used from both processes unless the process shared attribute is **PTHREAD_PROCESS_SHARED**.

The default pshared attribute for condition attributes objects is **PTHREAD_PROCESS_PRIVATE**.

Parameters

attr

(Input) Address of the variable that contains the condition attributes object

pshared

(Output) Address of the variable to contain the pshared attribute result

Authorities and Locks

None.

Return Value

0

pthread_condattr_getpshared() was successful.

value

pthread_condattr_getpshared() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_condattr_getpshared()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

Related Information

- The **<pthread.h>** header file. See [Header files for Pthread functions](#).
- [pthread_condattr_init\(\)--Initialize Condition Variable Attributes Object](#)

`pthread_condattr_getpshared()`--Get Process Shared Attribute from Condition Attributes Object

- [`pthread_condattr_setpshared\(\)`--Set Process Shared Attribute in Condition Attributes Object](#)
- [`pthread_cond_init\(\)`--Initialize Condition Variable](#)

Example

See the example for [`pthread_condattr_setpshared\(\)`](#).

pthread_condattr_init()--Initialize Condition Variable Attributes Object

Syntax

```
#include <pthread.h>
int pthread_condattr_init(pthread_condattr_t *attr);
Threadsafe: Yes
Signal Safe: Yes
```

The **pthread_condattr_init()** function initializes the condition variable attributes object specified by *attr* to the default attributes. The condition variable attributes object is used to create condition variables with the **pthread_cond_init()** function.

Parameters

attr

(Output) The address of the variable to contain the condition variable attributes object

Authorities and Locks

None.

Return Value

0

pthread_condattr_init() was successful.

value

pthread_condattr_init() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_condattr_init()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

Related Information

- The **<pthread.h>** header file. See [Header files for Pthread functions](#).
- [pthread_condattr_destroy\(\)--Destroy Condition Variable Attributes Object](#)
- [pthread_cond_init\(\)--Initialize Condition Variable](#)

Example

```
#include <pthread.h>
#include <stdio.h>
#include "check.h"

pthread_cond_t      cond;
```

```

int main(int argc, char **argv)
{
    int                rc=0;
    pthread_condattr_t attr;

    printf("Entering testcase\n");

    printf("Create a default condition attribute\n");
    rc = pthread_condattr_init(&attr);
    checkResults("pthread_condattr_init\n", rc);

    printf("Create the condition using the condition attributes object\n");
    rc = pthread_cond_init(&cond, &attr);
    checkResults("pthread_cond_init()\n", rc);

    printf("- At this point, the condition with its default attributes\n");
    printf("- Can be used from any threads that want to use it\n");

    printf("Destroy cond attribute\n");
    rc = pthread_condattr_destroy(&attr);
    checkResults("pthread_condattr_destroy()\n", rc);

    printf("Destroy condition\n");
    rc = pthread_cond_destroy(&cond);
    checkResults("pthread_cond_destroy()\n", rc);

    printf("Main completed\n");
    return 0;
}

```

Output:

```

Entering testcase
Create a default condition attribute
Create the condition using the condition attributes object
- At this point, the condition with its default attributes
- Can be used from any threads that want to use it
Destroy cond attribute
Destroy condition
Main completed

```

pthread_condattr_setpshared()--Set Process Shared Attribute in Condition Attributes Object

Syntax

```
#include <pthread.h>
int pthread_condattr_setpshared(pthread_condattr_t *attr,
                               int pshared);
```

Threadsafe: Yes

Signal Safe: Yes

The **pthread_condattr_setpshared()** function sets the current pshared attribute for the condition attributes object. The process shared attribute indicates whether the condition that is created using the condition attributes object can be shared between threads in separate processes (**PTHREAD_PROCESS_SHARED**) or shared between threads within the same process (**PTHREAD_PROCESS_PRIVATE**).

Even if the condition is in storage that is accessible from two separate processes, it cannot be used from both processes unless the process shared attribute is **PTHREAD_PROCESS_SHARED**.

The default pshared attribute for condition attributes objects is **PTHREAD_PROCESS_PRIVATE**.

Parameters

attr

(Input) Address of the variable containing the condition attributes object

pshared

(Output) One of **PTHREAD_PROCESS_SHARED** or **PTHREAD_PROCESS_PRIVATE**

Authorities and Locks

None.

Return Value

0

pthread_condattr_setpshared() was successful.

value

pthread_condattr_setpshared() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_condattr_setpshared()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

Related Information

- The **<pthread.h>** header file. See [Header files for Pthread functions](#).
- [pthread_condattr_getpshared\(\)--Get Process Shared Attribute from Condition Attributes Object](#)

- [pthread_condattr_init\(\)--Initialize Condition Variable Attributes Object](#)
- [pthread_cond_init\(\)--Initialize Condition Variable](#)

Example

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include <spawn.h>
#include <sys/wait.h>
#include <unistd.h>
#include <sys/shm.h>
#include "check.h"

typedef struct {
    int                eventOccured;
    int                numberWaiting;
    int                wakeup;
    int                reserved[1];
    pthread_cond_t     cond;
    pthread_mutex_t     mutex;          /* Protects this shared data and
condition
*/
} shared_data_t;

extern char            **environ;
shared_data_t          *sharedMem=NULL;
pid_t                  childPid=0;
int                     childStatus=-99;
int                     shmid=0;

/* Change this path to be the path to where you create this example program
*/
#define MYPATH          "/QSYS.LIB/QP0WTEST.LIB/TPCOSP0.PGM"

#define NTHREADSTHISJOB    2
#define NTHREADSTOTAL     4

void parentSetup(void);
void childSetup(void);
void parentCleanup(void);
void childCleanup(void);

void *parentThreadFunc(void *parm)
{
    int                rc;

    rc = pthread_mutex_lock(&sharedMem->mutex);
    checkResults("pthread_mutex_lock()\n", rc);
    /* Under protection of the lock, increment the count */
    ++sharedMem->numberWaiting;

    while (!sharedMem->eventOccured) {
        printf("PARENT - Thread blocked\n");
        rc = pthread_cond_wait(&sharedMem->cond, &sharedMem->mutex);
        checkResults("pthread_cond_wait()\n", rc);
    }
    printf("PARENT - Thread awake!\n");
```

```

    /* Under protection of the lock, decrement the count */
    --sharedMem->numberWaiting;

    /* After incrementing the wakeup flage and unlocking the mutex */
    /* we no longer use the shared memory, the parent could destroy*/
    /* it. We indicate we are finished with it using the wakeup flag*/
    ++sharedMem->wakeup;
    rc = pthread_mutex_unlock(&sharedMem->mutex);
    checkResults("pthread_mutex_lock()\n", rc);
    return NULL;
}

void *childThreadFunc(void *parm)
{
    int          rc;

    rc = pthread_mutex_lock(&sharedMem->mutex);
    checkResults("pthread_mutex_lock()\n", rc);
    /* Under protection of the lock, increment the count */
    ++sharedMem->numberWaiting;

    while (!sharedMem->eventOccured) {
        printf("CHILD - Thread blocked\n");
        rc = pthread_cond_wait(&sharedMem->cond, &sharedMem->mutex);
        checkResults("pthread_cond_wait()\n", rc);
    }
    printf("CHILD - Thread awake!\n");

    /* Under protection of the lock, decrement the count */
    --sharedMem->numberWaiting;

    /* After incrementing the wakeup flage and unlocking the mutex */
    /* we no longer use the shared memory, the parent could destroy*/
    /* it. We indicate we are finished with it using the wakeup flag*/
    ++sharedMem->wakeup;
    rc = pthread_mutex_unlock(&sharedMem->mutex);
    checkResults("pthread_mutex_lock()\n", rc);
    return NULL;
}

int main(int argc, char **argv)
{
    int          rc=0;
    int          i;
    pthread_t     threadid[NTHREADSTHISJOB];
    int          parentJob=0;

    /* If we run this from the QSHELL interpreter on the AS/400, we want
    */
    /* it to be line buffered even if we run it in batch so the output
    between*/
    /* parent and child is intermixed.
    */
    setvbuf(stdout, NULL, _IOLBF, 4096);
    /* Determine if we are running in the parent or child */
    if (argc != 1 && argc != 2) {
        printf("Incorrect usage\n");
        printf("Pass no parameters to run as the parent testcase\n");
        printf("Pass one parameter `ASCHILD' to run as the child testcase\n");
        exit(1);
    }
}

```



```

    if (argc == 1) {
        parentJob = 1;
    }
    else {
        if (strcmp(argv[1], "ASCHILD")) {
            printf("Incorrect usage\n");
            printf("Pass no parameters to run as the parent testcase\n");
            printf("Pass one parameter `ASCHILD' to run as the child
testcase\n");
            exit(1);
        }
        parentJob = 0;
    }

    /* PARENT
*****
    if (parentJob) {
        printf("PARENT - Enter Testcase - %s\n", argv[0]);
        parentSetup();

        printf("PARENT - Create %d threads\n", NTHREADSTHISJOB);
        for (i=0; i<NTHREADSTHISJOB; ++i) {
            rc = pthread_create(&threadid[i], NULL, parentThreadFunc, NULL);
            checkResults("pthread_create()\n", rc);
        }

        rc = pthread_mutex_lock(&sharedMem->mutex);
        checkResults("pthread_mutex_lock()\n", rc);
        while (sharedMem->numberWaiting != NTHREADSTOTAL) {
            printf("PARENT - Waiting for %d threads to wait, "
                "currently %d waiting\n",
                NTHREADSTOTAL, sharedMem->numberWaiting);

            rc = pthread_mutex_unlock(&sharedMem->mutex);
            checkResults("pthread_mutex_unlock()\n", rc);
            sleep(1);
            rc = pthread_mutex_lock(&sharedMem->mutex);
            checkResults("pthread_mutex_lock()\n", rc);
        }

        printf("PARENT - wake up all of the waiting threads...\n");
        sharedMem->eventOccured = 1;
        rc = pthread_cond_broadcast(&sharedMem->cond);
        checkResults("pthread_cond_signal()\n", rc);

        printf("PARENT - Wait for waking threads and cleanup\n");
        while (sharedMem->wokeup != NTHREADSTOTAL) {
            printf("PARENT - Waiting for %d threads to wake, "
                "currently %d wokeup\n",
                NTHREADSTOTAL, sharedMem->wokeup);

            rc = pthread_mutex_unlock(&sharedMem->mutex);
            checkResults("pthread_mutex_unlock()\n", rc);
            sleep(1);
            rc = pthread_mutex_lock(&sharedMem->mutex);
            checkResults("pthread_mutex_lock()\n", rc);
        }

        parentCleanup();
        printf("PARENT - Main completed\n");
    }

```

```

    exit(0);
}

/* CHILD
*****/
{
    void *status=NULL;

    printf("CHILD - Enter Testcase - %s\n", argv[0]);
    childSetup();

    printf("CHILD - Create %d threads\n", NTHREADSTHISJOB);
    for (i=0; i<NTHREADSTHISJOB; ++i) {
        rc = pthread_create(&threadid[i], NULL, childThreadFunc, NULL);
        checkResults("pthread_create()\n", rc);
    }
    /* The parent will wake up all of these threads using the */
    /* pshared condition variable. We will just join to them... */
    printf("CHILD - Joining to all threads\n");

    for (i=0; i<NTHREADSTHISJOB; ++i) {
        rc = pthread_join(threadid[i], &status);
        checkResults("pthread_join()\n", rc);
        if (status != NULL) {
            printf("CHILD - Got a bad status from a thread, "
                "%.8x %.8x %.8x %.8x\n", status);
            exit(1);
        }
    }
    /* After all the threads are awake, the parent will destroy */
    /* the condition and mutex. Do not use it anymore          */
    childCleanup();
    printf("CHILD - Main completed\n");
}
return 0;
}

/*****/
/* This function initializes the shared memory for the job, */
/* sets up the environment variable indicating where the shared */
/* memory is, and spawns the child job.                      */
/*                                                            */
/* It creates and initializes the shared memory segment, and */
/* It initializes the following global variables in this     */
/* job.                                                       */
/*    sharedMem                                              */
/*    childPid                                              */
/*    shmid                                                  */
/*                                                            */
/* If any of this setup/initialization fails, it will exit(1) */
/* and terminate the test.                                    */
/*                                                            */
/*****/
void parentSetup(void)
{
    int rc;

    /*****/
    /* Create shared memory for shared_data_t above          */
    /* attach the shared memory                               */

```

```

/* set the static/global sharedMem pointer to it */
/*****/
printf("PARENT - Create the shared memory segment\n");
rc = shmget(IPC_PRIVATE, sizeof(shared_data_t), 0666);
if (rc == -1) {
    printf("PARENT - Failed to create a shared memory segment\n");
    exit(1);
}
shmid = rc;

printf("PARENT - Attach the shared memory\n");
sharedMem = shmat(shmid, NULL, 0);
if (sharedMem == NULL) {
    shmctl(shmid, IPC_RMID, NULL);
    printf("PARENT - Failed to attach shared memory\n");
    exit(1);
}
/*****/
/* Initialize the mutex/condition and other shared memory data */
/*****/
{
    pthread_mutexattr_t      mattr;
    pthread_condattr_t       cattr;

    printf("PARENT - Init shared memory mutex/cond\n");
    memset(sharedMem, 0, sizeof(shared_data_t));

    /* Process Shared Mutex */
    rc = pthread_mutexattr_init(&mattr);
    checkResults("pthread_mutexattr_init()\n", rc);

    rc = pthread_mutexattr_setpshared(&mattr, PTHREAD_PROCESS_SHARED);
    checkResults("pthread_mutexattr_setpshared()\n", rc);

    rc = pthread_mutex_init(&sharedMem->mutex, &mattr);
    checkResults("pthread_mutex_init()\n", rc);

    /* Process Shared Condition */
    rc = pthread_condattr_init(&cattr);
    checkResults("pthread_condattr_init()\n", rc);

    rc = pthread_condattr_setpshared(&cattr, PTHREAD_PROCESS_SHARED);
    checkResults("pthread_condattr_setpshared()\n", rc);

    rc = pthread_cond_init(&sharedMem->cond, &cattr);
    checkResults("pthread_cond_init()\n", rc);
}
/*****/
/* Set and environment variable so that the child can inherit */
/* it and know the shared memory ID */
/*****/
{
    char      shmIdEnvVar[128];
    sprintf(shmIdEnvVar, "TPCOSP0_SHMID=%d\n", shmid);
    rc = putenv(shmIdEnvVar);
    if (rc) {
        printf("PARENT - Failed to store env var %s, errno=%d\n",
            shmIdEnvVar, errno);
        exit(1);
    }
    printf("PARENT - Stored shared memory id of %d\n", shmid);
}

```

```

    }

    /*******
    /* Spawn the child job
    /*******
    {
        inheritance_t  in;
        char          *av[3] = {NULL, NULL, NULL};

        /* Allow thread creation in the spawned child
        memset(&in, 0, sizeof(in));
        in.flags = SPAWN_SETTHREAD_NP;

        /* Set up the arguments to pass to spawn based on the
        /* arguments passed in
        av[0] = MYPATH;
        av[1] = "ASCHILD";
        av[2] = NULL;

        /* Spawn the child that was specified, inheriting all
        /* of the environment variables.
        childPid = spawn(MYPATH, 0, NULL, &in, av, environ);
        if (childPid == -1) {
            /* spawn failure */
            printf("PARENT - spawn() failed, errno=%d\n", errno);
            exit(1);
        }

        printf("PARENT - spawn() success, [PID=%d]\n", childPid);
    }

    return;
}

/*******
/* This function attaches the shared memory for the child job,
/* It uses the environment variable indicating where the shared
/* memory is.
/*
/*
/* If any of this setup/initialization fails, it will exit(1)
/* and terminate the test.
/*
/*
/* It initializes the following global variables:
/*     sharedMem
/*     shmid
/*******
void childSetup(void)
{
    int rc;

    printf("CHILD - Child setup\n");
    /*******
    /* Set and environment variable so that the child can inherit
    /* it and know the shared memory ID
    /*******
    {
        char          *shmIdEnvVar;
        shmIdEnvVar = getenv("TPCOSP0_SHMID");
        if (shmIdEnvVar == NULL) {
            printf("CHILD - Failed to get env var \"TPCOSP0_SHMID\",
            errno=%d\n",

```

```

        errno);
    exit(1);
}
shmId = atoi(shmIdEnvVar);
printf("CHILD - Got shared memory id of %d\n", shmId);
}
/*****
/* Create shared memory for shared_data_t above */
/* attach the shared memory */
/* set the static/global sharedMem pointer to it */
*****/
printf("CHILD - Attach the shared memory\n");
sharedMem = shmat(shmId, NULL, 0);
if (sharedMem == NULL) {
    shmctl(shmId, IPC_RMID, NULL);
    printf("CHILD - Failed to attach shared memory\n");
    exit(1);
}
return;
}

/*****
/* wait for child to complete and get return code. */
/* Destroy mutex and condition in shared memory */
/* detach and remove shared memory */
/* set the child's return code in global storage */
/*
/* If this function fails, it will call exit(1)
/*
/* This function sets the following global variables:
/*     sharedMem
/*     childStatus
/*     shmId
*****/
void parentCleanup(void)
{
    int         status=0;
    int         rc;
    int         waitedPid=0;

    /* Even though there is no thread left in the child using the
    /* contents of the shared memory, before we destroy the mutex
    /* and condition in that shared memory, we will wait for the
    /* child job to complete, we know for 100% certainty that no
    /* threads in the child are using it then.
    printf("PARENT - Parent cleanup\n");
    /* Wait for the child to complete */
    waitedPid = waitpid(childPid,&status,0);
    if (rc == -1) {
        printf("PARENT - waitpid failed, errno=%d\n", errno);
        exit(1);
    }
    childStatus = status;

    /* Cleanup resources */
    rc = pthread_mutex_destroy(&sharedMem->mutex);
    checkResults("pthread_mutex_destroy()\n", rc);

    rc = pthread_cond_destroy(&sharedMem->cond);
    checkResults("pthread_cond_destroy()\n", rc);

```

```

    /* Detach/Remove shared memory */
    rc = shmdt(sharedMem);
    if (rc) {
        printf("PARENT - Failed to detach shared memory, errno=%d\n", errno);
        exit(1);
    }
    rc = shmctl(shmid, IPC_RMID, NULL);
    if (rc) {
        printf("PARENT - Failed to remove shared memory id=%d, errno=%d\n",
            shmid, errno);
        exit(1);
    }
    shmid = 0;
    return;
}

/*****
/* Detach the shared memory */
/* At this point, there is no serialization, so the contents */
/* of the shared memory should not be used. */
/* */
/* If this function fails, it will call exit(1) */
/* */
/* This function sets the following global variables: */
/*     sharedMem */
*****/
void childCleanup(void)
{
    int rc;

    printf("CHILD - Child cleanup\n");
    rc = shmdt(sharedMem);

    sharedMem = NULL;
    if (rc) {
        printf("CHILD - Failed to detach shared memory, errno=%d\n", errno);
        exit(1);
    }
    return;
}

```

Output:

This example was run under the OS/400 QShell Interpreter. In the QShell Interpreter, a program gets descriptors 0, 1, and 2 as the standard files; the parent and child I/O is directed to the console. When run in the QShell Interpreter, the output shows the intermixed output from both parent and child processes and gives a feeling for the time sequence of operations occurring in each job.

The QShell interpreter allows you to run multithreaded programs as if they were interactive. See the QShell documentation for a description of the QIBM_MULTI_THREADED shell variable, which allows you to start multithreaded programs.

The QShell Interpreter is option 30 of Base OS/400.

.

PARENT - Enter Testcase - QP0WTEST/TPCOSP0

```
PARENT - Create the shared memory segment
PARENT - Attach the shared memory
PARENT - Init shared memory mutex/cond
PARENT - Stored shared memory id of 862
PARENT - spawn() success, [PID=2651]
PARENT - Create 2 threads
PARENT - Thread blocked
PARENT - Waiting for 4 threads to wait, currently 1 waiting
PARENT - Thread blocked
CHILD - Enter Testcase - QP0WTEST/TPCOSP0
CHILD - Child setup
CHILD - Got shared memory id of 862
CHILD - Attach the shared memory
CHILD - Create 2 threads
CHILD - Thread blocked
CHILD - Joining to all threads
CHILD - Thread blocked
PARENT - wake up all of the waiting threads...
PARENT - Wait for waking threads and cleanup
PARENT - Waiting for 4 threads to wake, currently 0 wokeup
PARENT - Thread awake!
CHILD - Thread awake!
PARENT - Thread awake!
CHILD - Thread awake!
CHILD - Child cleanup
CHILD - Main completed
PARENT - Parent cleanup
PARENT - Main completed
```

pthread_cond_broadcast()--Broadcast Condition to All Waiting Threads

Syntax

```
#include <pthread.h>
int pthread_cond_broadcast(pthread_cond_t *cond);
Threadsafe: Yes
Signal Safe: No
```

The **pthread_cond_broadcast()** function wakes up all threads that are currently waiting on the condition variable specified by *cond*. If no threads are currently blocked on the condition variable, this call has no effect.

When the threads that were the target of the broadcast wake up, they contend for the mutex that they have associated with the condition variable on the call to **pthread_cond_timedwait()** or **pthread_cond_wait()**.

The signal and broadcast functions can be called by a thread whether or not it currently owns the mutex associated with the condition variable. If predictable scheduling behavior is required from the applications viewpoint however, the mutex should be locked by the thread calling **pthread_cond_signal()** or **pthread_cond_broadcast()**.

For dependable use of condition variables, and to ensure that you do not lose wake up operations on condition variables, your application should always use a boolean predicate and a mutex with the condition variable.

Parameters

cond

(Input) Pointer to the condition variable that is to be broadcast to

Authorities and Locks

None.

Return Value

0

pthread_cond_broadcast() was successful.

value

pthread_cond_broadcast() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_cond_broadcast()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

Related Information

- The **<pthread.h>** header file. See [Header files for Pthread functions](#).
- [pthread_cond_init\(\)--Initialize Condition Variable](#)
- [pthread_cond_signal\(\)--Signal Condition to One Waiting Thread](#)

pthread_cond_broadcast()--Broadcast Condition to All Waiting Threads

- [pthread_cond_timedwait\(\)--Timed Wait for Condition](#)
- [pthread_cond_wait\(\)--Wait for Condition](#)

Example

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include "check.h"

/* For safe condition variable usage, must use a boolean predicate and */
/* a mutex with the condition. */
int          conditionMet = 0;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

#define NTHREADS 5

void *threadfunc(void *parm)
{
    int          rc;

    rc = pthread_mutex_lock(&mutex);
    checkResults("pthread_mutex_lock()\n", rc);

    while (!conditionMet) {
        printf("Thread blocked\n");
        rc = pthread_cond_wait(&cond, &mutex);
        checkResults("pthread_cond_wait()\n", rc);
    }

    rc = pthread_mutex_unlock(&mutex);
    checkResults("pthread_mutex_lock()\n", rc);
    return NULL;
}

int main(int argc, char **argv)
{
    int          rc=0;
    int          i;
    pthread_t     threadid[NTHREADS];

    printf("Enter Testcase - %s\n", argv[0]);

    printf("Create %d threads\n", NTHREADS);
    for(i=0; i<NTHREADS; ++i) {
        rc = pthread_create(&threadid[i], NULL, threadfunc, NULL);
        checkResults("pthread_create()\n", rc);
    }

    sleep(5); /* Sleep is not a very robust way to serialize threads */
    rc = pthread_mutex_lock(&mutex);
    checkResults("pthread_mutex_lock()\n", rc);

    /* The condition has occurred. Set the flag and wake up any waiting threads */
    conditionMet = 1;
    printf("Wake up all waiting threads...\n");
    rc = pthread_cond_broadcast(&cond);
```

pthread_cond_broadcast()--Broadcast Condition to All Waiting Threads

```
    checkResults("pthread_cond_broadcast()\n", rc);

    rc = pthread_mutex_unlock(&mutex);
    checkResults("pthread_mutex_unlock()\n", rc);

    printf("Wait for threads and cleanup\n");
    for (i=0; i<NTHREADS; ++i) {
        rc = pthread_join(threadid[i], NULL);
        checkResults("pthread_join()\n", rc);
    }
    pthread_cond_destroy(&cond);
    pthread_mutex_destroy(&mutex);

    printf("Main completed\n");
    return 0;
}
```

Output:

```
Entering testcase
Create 5 threads
Thread blocked
Thread blocked
Thread blocked
Thread blocked
Thread blocked
Wake up all waiting threads...
Wait for threads and cleanup
Main completed
```

pthread_cond_destroy()--Destroy Condition Variable

Syntax

```
#include <pthread.h>
int pthread_cond_destroy(pthread_cond_t *cond);
Threadsafe: Yes
Signal Safe: Yes
```

The **pthread_cond_destroy()** function destroys the condition variable specified by *cond*. If threads are currently blocked on the condition variable, the **pthread_cond_destroy()** fails with the **EBUSY** error.

Parameters

cond

(Input) Address of the condition variable to destroy

Authorities and Locks

None.

Return Value

0

pthread_cond_destroy() was successful.

value

pthread_cond_destroy() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_cond_destroy()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

[EBUSY]

The condition variable was in use.

Related Information

- The **<pthread.h>** header file. See [Header files for Pthread functions](#).
- [pthread_cond_broadcast\(\)--Broadcast Condition to All Waiting Threads](#)
- [pthread_cond_init\(\)--Initialize Condition Variable](#)
- [pthread_cond_signal\(\)--Signal Condition to One Waiting Thread](#)
- [pthread_cond_timedwait\(\)--Timed Wait for Condition](#)
- [pthread_cond_wait\(\)--Wait for Condition](#)

Example

```

#include <pthread.h>
#include <stdio.h>
#include "check.h"

pthread_cond_t      cond;

int main(int argc, char **argv)
{
    int                rc=0;
    pthread_mutexattr_t attr;

    printf("Entering testcase\n");

    printf("Create the condition using the condition attributes object\n");
    rc = pthread_cond_init(&cond, NULL);
    checkResults("pthread_cond_init()\n", rc);

    printf("- At this point, the condition with its default attributes\n");
    printf("- Can be used from any threads that want to use it\n");

    printf("Destroy condition\n");
    rc = pthread_cond_destroy(&cond);
    checkResults("pthread_cond_destroy()\n", rc);

    printf("Main completed\n");
    return 0;
}

```

Output:

```

Entering testcase
Create the condition using the condition attributes object
- At this point, the condition with its default attributes
- Can be used from any threads that want to use it
Destroy condition
Main completed

```

pthread_cond_init()--Initialize Condition Variable

Syntax

```
#include <pthread.h>
int pthread_cond_init(pthread_cond_t *cond,
                      const pthread_condattr_t *attr);

pthread_cond_t  cond = PTHREAD_COND_INITIALIZER;
Threadsafe: Yes
Signal Safe: Yes
```

The **pthread_cond_init()** function initializes a condition variable object with the specified attributes for use. The new condition may be used immediately for serializing threads. If *attr* is specified as **NULL**, all attributes are set to the default condition attributes for the newly created condition.

With these declarations and initialization:

```
pthread_cond_t      cond2;
pthread_cond_t      cond3;
pthread_condattr_t  attr;
pthread_condattr_init(&attr);
```

The following four condition variable initialization mechanisms have equivalent function:

```
pthread_cond_t      cond1 = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_init(&cond2, NULL);
pthread_cond_init(&cond3, &attr);
```

All four condition variables are created with the default condition attributes.

Every condition variable must eventually be destroyed with **pthread_cond_destroy()**.

Once a condition variable is created, it cannot be validly copied or moved to a new location. If the condition variable is copied or moved to a new location, the new object is not valid and cannot be used. Attempts to use the new object cause the **EINVAL** error.

Static initialization using the **PTHREAD_COND_INITIALIZER** does not immediately initialize the mutex. Instead, on first use, the functions **pthread_cond_wait()**, **pthread_cond_timedwait()**, **pthread_cond_signal()**, and **pthread_cond_broadcast()** branch into a slow path and cause the initialization of the condition. Due to this delayed initialization, the results of calling **pthread_cond_destroy()** on a condition variable that was initialized using static initialization and not used yet cause **pthread_cond_destroy()** to fail with the **EINVAL** error.

Parameters

cond

(Output) The address of the condition variable to initialize

attr

(Input) The address of the condition attributes object to use for initialization

Authorities and Locks

None.

Return Value

0

pthread_cond_init() was successful.

value

pthread_cond_init() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_cond_init()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

Related Information

- The `<pthread.h>` header file. See [Header files for Pthread functions](#).
- [pthread_cond_broadcast\(\)--Broadcast Condition to All Waiting Threads](#)
- [pthread_cond_destroy\(\)--Destroy Condition Variable](#)
- [pthread_cond_signal\(\)--Signal Condition to One Waiting Thread](#)
- [pthread_cond_timedwait\(\)--Timed Wait for Condition](#)
- [pthread_cond_wait\(\)--Wait for Condition](#)

Example

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include "check.h"

pthread_cond_t      cond1 = PTHREAD_COND_INITIALIZER;
pthread_cond_t      cond2;
pthread_cond_t      cond3;

int main(int argc, char **argv)
{
    int                rc=0;
    pthread_condattr_t attr;

    printf("Enter Testcase - %s\n", argv[0]);

    printf("Create the default cond attributes object\n");
    rc = pthread_condattr_init(&attr);
    checkResults("pthread_condattr_init()\n", rc);

    printf("Create the all of the default conditions in different ways\n");
    rc = pthread_cond_init(&cond2, NULL);
    checkResults("pthread_cond_init()\n", rc);

    rc = pthread_cond_init(&cond3, &attr);
    checkResults("pthread_cond_init()\n", rc);

    printf("- At this point, the conditions with default attributes\n");
```

pthread_cond_init()--Initialize Condition Variable

```
    printf("- Can be used from any threads that want to use them\n");

    printf("Cleanup\n");
    pthread_condattr_destroy(&attr);
    pthread_cond_destroy(&cond1);
    pthread_cond_destroy(&cond2);
    pthread_cond_destroy(&cond3);

    printf("Main completed\n");
    return 0;
}
```

Output:

```
Enter Testcase - QP0WTEST/TPCOI0
Create the default cond attributes object
Create the all of the default conditions in different ways
- At this point, the conditions with default attributes
- Can be used from any threads that want to use them
Cleanup
Main completed
```

pthread_cond_signal()--Signal Condition to One Waiting Thread

Syntax

```
#include <pthread.h>
int pthread_cond_signal(pthread_cond_t *cond);
Threadsafe: Yes
Signal Safe: No
```

The **pthread_cond_signal()** function wakes up at least one thread that is currently waiting on the condition variable specified by *cond*. If no threads are currently blocked on the condition variable, this call has no effect.

When the thread that was the target of the signal wakes up, it contends for the mutex that it has associated with the condition variable on the call to **pthread_cond_timedwait()** or **pthread_cond_wait()**.

The signal and broadcast functions can be called by a thread whether or not it currently owns the mutex associated with the condition variable. If predictable scheduling behavior is required from the applications viewpoint, however, the mutex should be locked by the thread that calls **pthread_cond_signal()** or **pthread_cond_broadcast()**.

For dependable use of condition variables, and to ensure that you do not lose wake-up operations on condition variables, your application should always use a Boolean predicate and a mutex with the condition variable.

Parameters

cond

(Input) Address of the condition variable to be signaled

Authorities and Locks

None.

Return Value

0

pthread_cond_signal() was successful.

value

pthread_cond_signal() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_cond_signal()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The condition specified is not valid.

Related Information

- The **<pthread.h>** header file. See [Header files for Pthread functions](#).
- [pthread_cond_broadcast\(\)--Broadcast Condition to All Waiting Threads](#)
- [pthread_cond_init\(\)--Initialize Condition Variable](#)

- [pthread_cond_timedwait\(\)--Timed Wait for Condition](#)
- [pthread_cond_wait\(\)--Wait for Condition](#)

Example

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include "check.h"

/* For safe condition variable usage, must use a boolean predicate and */
/* a mutex with the condition. */
int          workToDo = 0;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

#define NTHREADS      2

void *threadfunc(void *parm)
{
    int          rc;

    while (1) {
        /* Usually worker threads will loop on these operations */
        rc = pthread_mutex_lock(&mutex);
        checkResults("pthread_mutex_lock()\n", rc);

        while (!workToDo) {
            printf("Thread blocked\n");
            rc = pthread_cond_wait(&cond, &mutex);
            checkResults("pthread_cond_wait()\n", rc);
        }
        printf("Thread awake, finish work!\n");

        /* Under protection of the lock, complete or remove the work */
        /* from whatever worker queue we have. Here it is simply a flag */
        workToDo = 0;

        rc = pthread_mutex_unlock(&mutex);
        checkResults("pthread_mutex_unlock()\n", rc);
    }
    return NULL;
}

int main(int argc, char **argv)
{
    int          rc=0;
    int          i;
    pthread_t     threadid[NTHREADS];

    printf("Enter Testcase - %s\n", argv[0]);

    printf("Create %d threads\n", NTHREADS);
    for(i=0; i<NTHREADS; ++i) {
        rc = pthread_create(&threadid[i], NULL, threadfunc, NULL);
        checkResults("pthread_create()\n", rc);
    }

    sleep(5); /* Sleep is not a very robust way to serialize threads */
}
```

```

for(i=0; i<5; ++i) {
    printf("Wake up a worker, work to do...\n");

    rc = pthread_mutex_lock(&mutex);
    checkResults("pthread_mutex_lock()\n", rc);

    /* In the real world, all the threads might be busy, and      */
    /* we would add work to a queue instead of simply using a flag */
    /* In that case the boolean predicate might be some boolean    */
    /* statement like: if (the-queue-contains-work)                */
    if (workToDo) {
        printf("Work already present, likely threads are busy\n");
    }
    workToDo = 1;
    rc = pthread_cond_signal(&cond);
    checkResults("pthread_cond_broadcast()\n", rc);

    rc = pthread_mutex_unlock(&mutex);
    checkResults("pthread_mutex_unlock()\n", rc);
    sleep(5); /* Sleep is not a very robust way to serialize threads */
}

printf("Main completed\n");
exit(0);
return 0;
}

```

Output:

```

Enter Testcase - QP0WTEST/TPCOS0
Create 2 threads
Thread blocked
Thread blocked
Wake up a worker, work to do...
Thread awake, finish work!
Thread blocked
Wake up a worker, work to do...
Thread awake, finish work!
Thread blocked
Wake up a worker, work to do...
Thread awake, finish work!
Thread blocked
Wake up a worker, work to do...
Thread awake, finish work!
Thread blocked
Wake up a worker, work to do...
Thread awake, finish work!
Thread blocked
Main completed

```

pthread_cond_timedwait()--Timed Wait for Condition

Syntax

```
#include <pthread.h>
int pthread_cond_timedwait(pthread_cond_t *cond,
                           pthread_mutex_t *mutex,
                           const struct timespec *abstime);
```

Threadsafe: Yes

Signal Safe: No

The **pthread_cond_timedwait()** function blocks the calling thread, waiting for the condition specified by *cond* to be signaled or broadcast to.

When **pthread_cond_timedwait()** is called, the calling thread must have *mutex* locked. The **pthread_cond_timedwait()** function atomically unlocks the mutex and performs the wait for the condition. In this case, atomically means with respect to the mutex and the condition variable and other access by threads to those objects through the pthread condition variable interfaces.

If the wait is satisfied or times out, or if the thread is canceled, before the thread is allowed to continue, the mutex is automatically acquired by the calling thread. If *mutex* is not currently locked, an **ENOTLOCKED** error results. You should always associate only one mutex with a condition at a time. Using two different mutexes with the same condition at the same time could lead to unpredictable serialization in your application.

The time to wait is specified by the *abstime* parameter as an absolute system time at which the wait expires. If the current system clock time passes the absolute time specified before the condition is signaled, an **ETIMEDOUT** error results. After the wait begins, the wait time is not affected by changes to the software clock.

Although time is specified in seconds and nanoseconds, the system has approximately millisecond granularity. Due to scheduling and priorities, the amount of time you actually wait might be slightly more or less than the amount of time specified.

The current absolute system time can be retrieved as a timeval structure using the software clock interface **gettimeofday()**. The timeval structure can easily have a delta value added to it and be converted to a timespec structure. The MI time interfaces can be used to retrieve the current system time. The MI time also needs to be converted to a timespec structure before use by **pthread_cond_timedwait()** using the **Qp0zConvertTime()** interface.

This function is a cancellation point.

For dependable use of condition variables, and to ensure that you do not lose wake-up operations on condition variables, your application should always use a Boolean predicate and a mutex with the condition variable.

Parameters

cond

(Input) Address of the condition variable to wait for

mutex

(Input) Address of the locked mutex associated with the condition variable

abstime

(Input) Address of the absolute system time at which the wait expires

Authorities and Locks

For successful completion, the mutex lock associated with the condition variable must be locked before you call `pthread_cond_timedwait()`.

Return Value

0

`pthread_cond_timedwait()` was successful.

value

`pthread_cond_timedwait()` was not successful. *value* is set to indicate the error condition.

Error Conditions

If `pthread_cond_timedwait()` was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

[ENOTLOCKED]

The mutex specified is not locked by the caller.

[ETIMEDOUT]

The wait timed out without being satisfied.

Related Information

- The `<pthread.h>` header file. See [Header files for Pthread functions](#).
- [pthread_cond_broadcast\(\)--Broadcast Condition to All Waiting Threads](#)
- [pthread_cond_init\(\)--Initialize Condition Variable](#)
- [pthread_cond_signal\(\)--Signal Condition to One Waiting Threads](#)
- [pthread_cond_wait\(\)--Wait for Condition](#)

Example

```
#define _MULTI_THREADED
#include <stdio.h>
#include <qp0z1170.h>
#include <time.h>
#include <pthread.h>
#include "check.h"

/* For safe condition variable usage, must use a boolean predicate and */
/* a mutex with the condition. */
int          workToDo = 0;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

#define NTHREADS          3
#define WAIT_TIME_SECONDS 15

void *threadfunc(void *parm)
{
```

pthread_cond_timedwait)--Timed Wait for Condition

```
int rc;
struct timespec ts;
struct timeval tp;

rc = pthread_mutex_lock(&mutex);
checkResults("pthread_mutex_lock()\n", rc);

/* Usually worker threads will loop on these operations */
while (1) {
    rc = gettimeofday(&tp, NULL);
    checkResults("gettimeofday()\n", rc);

    /* Convert from timeval to timespec */
    ts.tv_sec = tp.tv_sec;
    ts.tv_nsec = tp.tv_usec * 1000;
    ts.tv_sec += WAIT_TIME_SECONDS;

    while (!workToDo) {
        printf("Thread blocked\n");
        rc = pthread_cond_timedwait(&cond, &mutex, &ts);
        /* If the wait timed out, in this example, the work is complete, and
        */
        /* the thread will end.
        */
        /* In reality, a timeout must be accompanied by some sort of checking
        */
        /* to see if the work is REALLY all complete. In the simple example
        */
        /* we will just go belly up when we time
        */
        out.
        if (rc == ETIMEDOUT) {
            printf("Wait timed out!\n");
            rc = pthread_mutex_unlock(&mutex);
            checkResults("pthread_mutex_lock()\n", rc);
            pthread_exit(NULL);
        }
        checkResults("pthread_cond_timedwait()\n", rc);
    }

    printf("Thread consumes work here\n");
    workToDo = 0;
}

rc = pthread_mutex_unlock(&mutex);
checkResults("pthread_mutex_lock()\n", rc);
return NULL;
}

int main(int argc, char **argv)
{
    int rc=0;
    int i;
    pthread_t threadid[NTHREADS];

    printf("Enter Testcase - %s\n", argv[0]);

    printf("Create %d threads\n", NTHREADS);
    for(i=0; i<NTHREADS; ++i) {
        rc = pthread_create(&threadid[i], NULL, threadfunc, NULL);
        checkResults("pthread_create()\n", rc);
    }
}
```

```

rc = pthread_mutex_lock(&mutex);
checkResults("pthread_mutex_lock()\n", rc);

printf("One work item to give to a thread\n");
workToDo = 1;
rc = pthread_cond_signal(&cond);
checkResults("pthread_cond_signal()\n", rc);

rc = pthread_mutex_unlock(&mutex);
checkResults("pthread_mutex_unlock()\n", rc);

printf("Wait for threads and cleanup\n");
for (i=0; i<NTHREADS; ++i) {
    rc = pthread_join(threadid[i], NULL);
    checkResults("pthread_join()\n", rc);
}

pthread_cond_destroy(&cond);
pthread_mutex_destroy(&mutex);
printf("Main completed\n");
return 0;
}

```

Output:

```

Enter Testcase - QP0WTEST/TPCOT0
Create 3 threads
Thread blocked
One work item to give to a thread
Wait for threads and cleanup
Thread consumes work here
Thread blocked
Thread blocked
Thread blocked
Wait timed out!
Wait timed out!
Wait timed out!
Main completed

```

pthread_cond_wait()--Wait for Condition

Syntax

```
#include <pthread.h>
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
```

Threadsafe: Yes

Signal Safe: No

The **pthread_cond_wait()** function blocks the calling thread, waiting for the condition specified by *cond* to be signaled or broadcast to.

When **pthread_cond_wait()** is called, the calling thread must have *mutex* locked. The **pthread_cond_wait()** function atomically unlocks *mutex* and performs the wait for the condition. In this case, atomically means with respect to the *mutex* and the condition variable and another threads access to those objects through the pthread condition variable interfaces.

If the wait is satisfied, or if the thread is canceled, before the thread is allowed to continue, the *mutex* is automatically acquired by the calling thread. If *mutex* is not currently locked, an **ENOTLOCKED** error results. You should always associate only one *mutex* with a condition at a time. Using two different *mutexes* with the same condition at the same time could lead to unpredictable serialization issues in your application.

This function is a cancellation point.

For dependable use of condition variables, and to ensure that you do not lose wake up operations on condition variables, your application should always use a boolean predicate and a mutex with the condition variable.

Parameters

cond

(Input) Address of the condition variable to wait on

mutex

(Input) Address of the mutex associated with the condition variable

Authorities and Locks

For successful completion, the mutex lock associated with the condition variable is must be locked prior to calling **pthread_cond_wait()**.

Return Value

0

pthread_cond_wait() was successful.

value

pthread_cond_wait() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_cond_wait()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

[ENOTLOCKED]

The mutex associated with the condition variable is not locked.

Related Information

- The `<pthread.h>` header file. See [Header files for Pthread functions](#).
- [pthread_cond_broadcast\(\)--Broadcast Condition to All Waiting Threads](#)
- [pthread_cond_init\(\)--Initialize Condition Variable](#)
- [pthread_cond_signal\(\)--Signal Condition to One Waiting Thread](#)
- [pthread_cond_timedwait\(\)--Timed Wait for Condition](#)

Example

```
#define _MULTI_THREADED
#include <pthread.h>

#include <stdio.h>
#include "check.h"

/* For safe condition variable usage, must use a boolean predicate and */
/* a mutex with the condition. */
int          conditionMet = 0;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

#define NTHREADS 5

void *threadfunc(void *parm)
{
    int          rc;

    rc = pthread_mutex_lock(&mutex);
    checkResults("pthread_mutex_lock()\n", rc);

    while (!conditionMet) {
        printf("Thread blocked\n");
        rc = pthread_cond_wait(&cond, &mutex);
        checkResults("pthread_cond_wait()\n", rc);
    }

    rc = pthread_mutex_unlock(&mutex);
    checkResults("pthread_mutex_lock()\n", rc);
    return NULL;
}

int main(int argc, char **argv)
{
    int          rc=0;
    int          i;
    pthread_t     threadid[NTHREADS];

    printf("Enter Testcase - %s\n", argv[0]);

    printf("Create %d threads\n", NTHREADS);
    for(i=0; i<NTHREADS; ++i) {
        rc = pthread_create(&threadid[i], NULL, threadfunc, NULL);
    }
}
```


pthread_cond_wait()--Wait for Condition

```
    checkResults("pthread_create()\n", rc);
}

sleep(5); /* Sleep is not a very robust way to serialize threads */
rc = pthread_mutex_lock(&mutex);
checkResults("pthread_mutex_lock()\n", rc);

/* The condition has occurred. Set the flag and wake up any waiting threads
*/
conditionMet = 1;
printf("Wake up all waiting threads...\n");
rc = pthread_cond_broadcast(&cond);
checkResults("pthread_cond_broadcast()\n", rc);

rc = pthread_mutex_unlock(&mutex);
checkResults("pthread_mutex_unlock()\n", rc);

printf("Wait for threads and cleanup\n");
for (i=0; i<NTHREADS; ++i) {
    rc = pthread_join(threadid[i], NULL);
    checkResults("pthread_join()\n", rc);
}
pthread_cond_destroy(&cond);
pthread_mutex_destroy(&mutex);

printf("Main completed\n");
return 0;
}
```

Output:

```
Entering testcase
Create 5 threads
Thread blocked
Thread blocked
Thread blocked
Thread blocked
Thread blocked
Thread blocked
Wake up all waiting threads...
Wait for threads and cleanup
Main completed
```

pthread_get_expiration_np()--Get Condition Expiration Time from Relative Time

Syntax

```
#include <pthread.h>
#include <sched.h>
int pthread_get_expiration_np(const struct timespec *delta,
                             struct timespec *abstime);
```

Threadsafe: Yes

Signal Safe: Yes

The **pthread_get_expiration_np()** function computes an absolute time by adding the specified relative time (*delta*) to the current system time. The resulting absolute time output in the *abstime* parameter can be used as the expiration time in a call to **pthread_cond_timedwait()**.

The current system time is retrieved from the systems software clock.

This function is not portable.

Parameters

delta

(Input) Elapsed time to add to the current system time

abstime

(Output) Address of the returned value representing the expiration time

Authorities and Locks

None.

Return Value

0

pthread_get_expiration_np() was successful.

value

pthread_get_expiration_np() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_get_expiration_np()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

Related Information

- The **<pthread.h>** header file. See [Header files for Pthread functions](#).
- [pthread_cond_timedwait\(\)--Timed Wait for Condition](#)

Example

```

#define _MULTI_THREADED
#include <stdio.h>
#include <qp0z1170.h>
#include <time.h>
#include <pthread.h>
#include "check.h"

/* For safe condition variable usage, must use a boolean predicate and */
/* a mutex with the condition. */
int          workToDo = 0;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int          failStatus=99;

#define NTHREADS          2
#define WAIT_TIME_SECONDS 3

void *threadfunc(void *parm)
{
    int          rc;
    struct timespec delta;
    struct timespec abstime;
    int          retries = 2;
    pthread_id_np_t tid;

    tid = pthread_getthreadid_np();

    rc = pthread_mutex_lock(&mutex);
    checkResults("pthread_mutex_lock()\n", rc);

    while (retries--> 0) {
        delta.tv_sec = WAIT_TIME_SECONDS;
        delta.tv_nsec = 0;
        rc = pthread_get_expiration_np(&delta, &abstime);
        checkResults("pthread_get_expiration_np()\n", rc);

        while (!workToDo) {
            printf("Thread 0x%.8x %.8x blocked\n", tid);
            rc = pthread_cond_timedwait(&cond, &mutex, &abstime);
            if (rc != ETIMEDOUT) {
                printf("pthread_cond_timedwait() - expect timeout %d\n", rc);
                rc = pthread_mutex_unlock(&mutex);
                checkResults("pthread_mutex_lock()\n", rc);
                return __VOID(failStatus);
            }
            /* Since there is no code in this example to wake up any */
            /* thread on the condition variable, we know we are done */
            /* because we have timed out. */
            break;
        }
        printf("Wait timed out! tid=0x%.8x %.8x\n", tid);
    }

    rc = pthread_mutex_unlock(&mutex);
    checkResults("pthread_mutex_lock()\n", rc);
    return __VOID(0);
}

```

pthread_get_expiration_np())--Get Condition Expiration Time from Relative Time

```
int main(int argc, char **argv)
{
    int                rc=0;
    int                i;
    pthread_t          threadid[NTHREADS];
    void                *status;
    int                results=0;

    printf("Enter Testcase - %s\n", argv[0]);

    printf("Create %d threads\n", NTHREADS);
    for(i=0; i<NTHREADS; ++i) {
        rc = pthread_create(&threadid[i], NULL, threadfunc, NULL);
        checkResults("pthread_create()\n", rc);
    }

    printf("Wait for threads and cleanup\n");
    for (i=0; i<NTHREADS; ++i) {
        rc = pthread_join(threadid[i], &status);
        checkResults("pthread_join()\n", rc);
        if (__INT(status) == failStatus) {
            printf("A thread failed!\n");
            results++;
        }
    }

    pthread_cond_destroy(&cond);
    pthread_mutex_destroy(&mutex);
    printf("Main completed\n");
    return results;
}
```

Output:

```
Enter Testcase - QP0WTEST/TPGETEX0
Create 2 threads
Wait for threads and cleanup
Thread 0x00000000 000002ab blocked
Thread 0x00000000 000002ac blocked
Wait timed out! tid=0x00000000 000002ab
Thread 0x00000000 000002ab blocked
Wait timed out! tid=0x00000000 000002ac
Thread 0x00000000 000002ac blocked
Wait timed out! tid=0x00000000 000002ab
Wait timed out! tid=0x00000000 000002ac
Main completed
```

Read/write lock synchronization APIs

Read/write locks help you build more complex applications without using mutexes and condition variables to provide your own read/write locking primitive object. Read/Write locks provide a synchronization mechanism that allow threads in an application to more accurately reflect the type of access to a shared resource that they require.

Many threads can acquire the same read/write lock if they acquire a shared read lock on the read/write lock object. Only one thread can acquire an exclusive write lock on a read/write lock object. When an exclusive write lock is held, no other threads are allowed to hold any lock.

The table below lists important read/write lock attributes, their default values, and all supported values.

Attribute	Default value	Supported values
<i>pshared</i>	PTHREAD_PROCESS_PRIVATE	PTHREAD_PROCESS_PRIVATE or PTHREAD_PROCESS_SHARED

For information about the examples included with the APIs, see the [information on the API examples](#).

To view the API list by description, see [Read/write lock synchronization APIs](#).

Read/write lock synchronization APIs by name

- [pthread_rwlockattr_destroy\(\)](#)--Destroy Read/Write Lock Attribute
- [pthread_rwlockattr_getpshared\(\)](#)--Get Pshared Read/Write Lock Attribute
- [pthread_rwlockattr_init\(\)](#)--Initialize Read/Write Lock Attribute
- [pthread_rwlockattr_setpshared\(\)](#)--Set Pshared Read/Write Lock Attribute
- [pthread_rwlock_destroy\(\)](#)--Destroy Read/Write Lock
- [pthread_rwlock_init\(\)](#)--Initialize Read/Write Lock
- [pthread_rwlock_rdlock\(\)](#)--Get Shared Read Lock
- [pthread_rwlock_timedrdlock_np\(\)](#)--Get Shared Read Lock with Time-Out
- [pthread_rwlock_timedwrlock_np\(\)](#)--Get Exclusive Write Lock with Time-Out
- [pthread_rwlock_tryrdlock\(\)](#)--Get Shared Read Lock with No Wait
- [pthread_rwlock_trywrlock\(\)](#)--Get Exclusive Write Lock with No Wait
- [pthread_rwlock_unlock\(\)](#)--Unlock Exclusive Write or Shared Read Lock
- [pthread_rwlock_wrlock\(\)](#)--Get Exclusive Write Lock

pthread_rwlockattr_destroy()--Destroy Read/Write Lock Attribute

Syntax

```
#include <pthread.h>
int pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);
Threadsafe: Yes
Signal Safe: Yes
```

The **pthread_rwlockattr_destroy()** function destroys a read/write lock attributes object and allows the systems to reclaim any resources associated with that read/write lock attributes object. This does not have an effect on any read/write lock already created using this read/write lock attributes object.

Parameters

attr

(Input) Address of the read/write lock attributes object to be destroyed

Authorities and Locks

None.

Return Value

0

pthread_rwlockattr_destroy() was successful.

value

pthread_rwlockattr_destroy() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_rwlockattr_destroy()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

Related Information

- The **<pthread.h>** header file. See [Header files for Pthread functions](#).
- [pthread_rwlockattr_init\(\)--Initialize Read/Write Lock Attribute](#)
- [pthread_rwlock_init\(\)--Initialize Read/Write Lock](#)

Example

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include "check.h"
```

pthread_rwlockattr_destroy()--Destroy Read/Write Lock Attribute

```
pthread_rwlock_t      rwlock1;
pthread_rwlock_t      rwlock2 = PTHREAD_RWLOCK_INITIALIZER;

int main(int argc, char **argv)
{
    int                rc=0;
    pthread_rwlockattr_t  attr;

    printf("Enter Testcase - %s\n", argv[0]);

    printf("Create a default rwlock attribute\n");
    rc = pthread_rwlockattr_init(&attr);
    checkResults("pthread_rwlockattr_init()\n", rc);

    printf("Use the rwlock attributes to created rwlocks here\n");
    rc = pthread_rwlock_init(&rwlock1, &attr);
    checkResults("pthread_rwlock_init()\n", rc);

    printf("The rwlock1 is now ready for use.\n");
    printf("The rwlock2 that was statically initialized was ready when\n"
           "the main routine was entered\n");

    printf("Destroy rwlock attribute\n");
    rc = pthread_rwlockattr_destroy(&attr);
    checkResults("pthread_rwlockattr_destroy()\n", rc);

    printf("Use the rwlocks\n");
    rc = pthread_rwlock_rdlock(&rwlock1);
    checkResults("pthread_rwlock_rdlock()\n", rc);

    rc = pthread_rwlock_wrlock(&rwlock2);
    checkResults("pthread_rwlock_wrlock()\n", rc);

    rc = pthread_rwlock_unlock(&rwlock1);
    checkResults("pthread_rwlock_unlock(1)\n", rc);

    rc = pthread_rwlock_unlock(&rwlock2);
    checkResults("pthread_rwlock_unlock(2)\n", rc);

    printf("Destroy the rwlocks\n");
    rc = pthread_rwlock_destroy(&rwlock1);
    checkResults("pthread_rwlock_destroy(1)\n", rc);

    rc = pthread_rwlock_destroy(&rwlock2);
    checkResults("pthread_rwlock_destroy(2)\n", rc);

    printf("Main completed\n");
    return 0;
}
```

Output:

```
Enter Testcase - QP0WTEST/TPRWLAI0
Create a default rwlock attribute
Use the rwlock attributes to created rwlocks here
The rwlock is now ready for use.
The rwlock that was statically initialized was ready when
the main routine was entered
Destroy rwlock attribute
Use the rwlocks
```

pthread_rwlockattr_destroy()--Destroy Read/Write Lock Attribute

Destroy the rwlocks

Main completed

pthread_rwlockattr_getpshared()--Get Pshared Read/Write Lock Attribute

Syntax

```
#include <pthread.h>
int pthread_rwlockattr_getpshared(pthread_rwlockattr_t *attr, int *pshared);
Threadsafe: Yes
Signal Safe: Yes
```

The **pthread_rwlockattr_getpshared()** function retrieves the current setting of the process shared attribute from the read/write lock attributes object. The process shared attribute indicates whether the read/write lock that is created using the read/write lock attributes object can be shared between threads in separate processes (**PTHREAD_PROCESS_SHARED**) or shared only between threads within the same process (**PTHREAD_PROCESS_PRIVATE**).

Even if the read/write lock in storage is accessible from two separate processes, it cannot be used from both processes unless the process shared attribute is **PTHREAD_PROCESS_SHARED**.

The default pshared attribute for read/write lock attributes objects is **PTHREAD_PROCESS_PRIVATE**.

Parameters

attr

(Input) Address of the variable that contains the read/write lock attributes object

attr

(Output) Address of the variable to contain the pshared attribute result

Authorities and Locks

None.

Return Value

0

pthread_rwlockattr_getpshared() was successful.

value

pthread_rwlockattr_getpshared() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_rwlockattr_getpshared()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

Related Information

- The **<pthread.h>** header file. See [Header files for Pthread functions](#).
- [pthread_rwlockattr_init\(\)--Initialize Read/Write Lock Attribute](#)

- [pthread_rwlockattr_setpshared\(\)--Set Pshared Read/Write Lock Attribute](#)
- [pthread_rwlock_init\(\)--Initialize Read/Write Lock](#)

Example

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include <spawn.h>
#include <sys/wait.h>
#include <unistd.h>
#include <sys/shm.h>
#include "check.h"

typedef struct {
    int                protectedResource;
    pthread_rwlock_t   rwlock;
} shared_data_t;

extern char            **environ;
shared_data_t          *sharedMem=NULL;
pid_t                  childPid=0;
int                     childStatus=-99;
int                     shmId=0;

/* Change this path to be the path to where you create this example program
*/
#define MYPATH           "/QSYS.LIB/QP0WTEST.LIB/TPRWLSH0.PGM"

#define NTHREADSTHISJOB    2
#define NTHREADSTOTAL     4

void parentSetup(void);
void childSetup(void);
void parentCleanup(void);
void childCleanup(void);

void *childReaderThreadFunc(void *parm)
{
    int                rc;
    int                retries = 5;

    while (retries--) {
        rc = pthread_rwlock_rdlock(&sharedMem->rwlock);
        checkResults("pthread_rwlock_rdlock()\n", rc);
        /* Under protection of the shared read lock, read the resource */
        printf("CHILD READER - current protectedResource = %d\n",
            sharedMem->protectedResource);
        sleep(1);

        printf("CHILD READER - unlock\n");
        rc = pthread_rwlock_unlock(&sharedMem->rwlock);
        checkResults("pthread_rwlock_unlock()\n", rc);
    }
    return NULL;
}

void *parentWriterThreadFunc(void *parm)
{

```

```

    int                rc;

    rc = pthread_rwlock_wrlock(&sharedMem->rwlock);
    checkResults("pthread_rwlock_rdlock()\n", rc);
    /* Under protection of the exclusive write lock, write the resource */
    ++sharedMem->protectedResource;
    printf("PARENT WRITER - current protectedResource = %d\n",
          sharedMem->protectedResource);
    sleep(5);

    printf("PARENT WRITER - unlock\n");
    rc = pthread_rwlock_unlock(&sharedMem->rwlock);

    checkResults("pthread_rwlock_unlock()\n", rc);
    return NULL;
}

int main(int argc, char **argv)
{
    int                rc=0;
    int                i;
    pthread_t          threadid[NTHREADSTHISJOB];
    int                parentJob=0;
    void                *status=NULL;

    /* If we run this from the QSHELL interpreter on the AS/400, we want
    */
    /* it to be line buffered even if we run it in batch so the output
    between*/
    /* parent and child is intermixed.
    */
    setvbuf(stdout, NULL, _IOLBF, 4096);
    /* Determine if we are running in the parent or child */
    if (argc != 1 && argc != 2) {
        printf("Incorrect usage\n");
        printf("Pass no parameters to run as the parent testcase\n");
        printf("Pass one parameter `ASCHILD' to run as the child testcase\n");
        exit(1);
    }

    if (argc == 1) {
        parentJob = 1;
    }
    else {
        if (strcmp(argv[1], "ASCHILD")) {
            printf("Incorrect usage\n");
            printf("Pass no parameters to run as the parent testcase\n");
            printf("Pass one parameter `ASCHILD' to run as the child
testcase\n");
            exit(1);
        }
        parentJob = 0;
    }

    /* PARENT
    *****/
    if (parentJob) {

```

```

    printf("PARENT - Enter Testcase - %s\n", argv[0]);
    parentSetup();

    printf("PARENT - Create %d threads\n", NTHREADSTHISJOB);
    for (i=0; i<NTHREADSTHISJOB; ++i) {
        rc = pthread_create(&threadid[i], NULL, parentWriterThreadFunc,
NULL);
        checkResults("pthread_create()\n", rc);
    }

    for (i=0; i<NTHREADSTHISJOB; ++i) {
        rc = pthread_join(threadid[i], NULL);
        checkResults("pthread_create()\n", rc);
        if (status != NULL) {
            printf("PARENT - Got a bad status from a thread, "
                "%.8x %.8x %.8x %.8x\n", status);
            exit(1);
        }
    }

    parentCleanup();
    printf("PARENT - Main completed\n");
    exit(0);
}

/* CHILD
*****/
{
    printf("CHILD - Enter Testcase - %s\n", argv[0]);
    childSetup();

    printf("CHILD - Create %d threads\n", NTHREADSTHISJOB);
    for (i=0; i<NTHREADSTHISJOB; ++i) {
        rc = pthread_create(&threadid[i], NULL, childReaderThreadFunc,
NULL);
        checkResults("pthread_create()\n", rc);
    }
    /* The parent will wake up all of these threads using the */
    /* pshared condition variable. We will just join to them... */
    printf("CHILD - Joining to all threads\n");

    for (i=0; i<NTHREADSTHISJOB; ++i) {
        rc = pthread_join(threadid[i], &status);
        checkResults("pthread_join()\n", rc);
        if (status != NULL) {
            printf("CHILD - Got a bad status from a thread, "
                "%.8x %.8x %.8x %.8x\n", status);
            exit(1);
        }
    }
    /* After all the threads are awake, the parent will destroy */
    /* the read/write lock. Do not use it anymore */
    childCleanup();
    printf("CHILD - Main completed\n");
}
return 0;
}

```

```

/*****/
/* This function initializes the shared memory for the job, */

```

```

/* sets up the environment variable indicating where the shared*/
/* memory is, and spawns the child job.                                */
/*                                                                    */
/* It creates and initializes the shared memory segment, and          */
/* It initializes the following global variables in this              */
/* job.                                                                */
/*    sharedMem                                                        */
/*    childPid                                                         */
/*    shmid                                                            */
/*                                                                    */
/* If any of this setup/initialization fails, it will exit(1)        */
/* and terminate the test.                                           */
/*                                                                    */
/*****
void parentSetup(void)
{
    int rc;

    /*****/
    /* Create shared memory for shared_data_t above                  */
    /* attach the shared memory                                      */
    /* set the static/global sharedMem pointer to it                */
    /*****/
    printf("PARENT - Create the shared memory segment\n");
    rc = shmget(IPC_PRIVATE, sizeof(shared_data_t), 0666);
    if (rc == -1) {
        printf("PARENT - Failed to create a shared memory segment\n");
        exit(1);
    }
    shmid = rc;

    printf("PARENT - Attach the shared memory\n");
    sharedMem = shmat(shmid, NULL, 0);
    if (sharedMem == NULL) {
        shmctl(shmid, IPC_RMID, NULL);
        printf("PARENT - Failed to attach shared memory\n");
        exit(1);
    }
    /*****/
    /* Initialize the read/write lock and other shared memory data */
    /*****/
    {
        pthread_rwlockattr_t          rwlattr;
        printf("PARENT - Init shared memory and read/write lock\n");
        memset(sharedMem, 0, sizeof(shared_data_t));

        /* Process Shared Read/Write lock */
        rc = pthread_rwlockattr_init(&rwlattr);
        checkResults("pthread_rwlockattr_init()\n", rc);

        rc = pthread_rwlockattr_setpshared(&rwlattr, PTHREAD_PROCESS_SHARED);
        checkResults("pthread_rwlockattr_setpshared()\n", rc);

        rc = pthread_rwlock_init(&sharedMem->rwlock, &rwlattr);
        checkResults("pthread_rwlock_init()\n", rc);
    }

    /*****/
    /* Set and environment variable so that the child can inherit */
    /* it and know the shared memory ID                            */
    /*****/

```

```

{
    char            shmIdEnvVar[128];
    sprintf(shmIdEnvVar, "TPRWLSH0_SHMID=%d\n", shmId);
    rc = putenv(shmIdEnvVar);
    if (rc) {
        printf("PARENT - Failed to store env var %s, errno=%d\n",
            shmIdEnvVar, errno);
        exit(1);
    }
    printf("PARENT - Stored shared memory id of %d\n", shmId);
}

/*****
/* Spawn the child job */
*****/
{
    inheritance_t   in;
    char            *av[3] = {NULL, NULL, NULL};

    /* Allow thread creation in the spawned child */
    memset(&in, 0, sizeof(in));
    in.flags = SPAWN_SETTHREAD_NP;

    /* Set up the arguments to pass to spawn based on the */
    /* arguments passed in */
    av[0] = MYPATH;
    av[1] = "ASCHILD";
    av[2] = NULL;

    /* Spawn the child that was specified, inheriting all */
    /* of the environment variables. */
    childPid = spawn(MYPATH, 0, NULL, &in, av, environ);
    if (childPid == -1) {
        /* spawn failure */
        printf("PARENT - spawn() failed, errno=%d\n", errno);
        exit(1);
    }

    printf("PARENT - spawn() success, [PID=%d]\n", childPid);
}

return;
}

/*****
/* This function attaches the shared memory for the child job, */
/* It uses the environment variable indicating where the shared */
/* memory is. */
/*
/* If any of this setup/initialization fails, it will exit(1) */
/* and terminate the test. */
/*
/* It initializes the following global variables: */
/*     sharedMem */
/*     shmId */
*****/
void childSetup(void)
{
    int rc;

    printf("CHILD - Child setup\n");

```

```

/*****
/* Set and environment variable so that the child can inherit */
/* it and know the shared memory ID */
/*****
{
    char          *shmIdEnvVar;
    shmIdEnvVar = getenv("TPRWLSH0_SHMID");
    if (shmIdEnvVar == NULL) {
        printf("CHILD - Failed to get env var \"TPRWLSH0_SHMID\",
errno=%d\n",
                errno);
        exit(1);
    }
    shmId = atoi(shmIdEnvVar);
    printf("CHILD - Got shared memory id of %d\n", shmId);
}
/*****
/* Create shared memory for shared_data_t above */
/* attach the shared memory */
/* set the static/global sharedMem pointer to it */
/*****
printf("CHILD - Attach the shared memory\n");
sharedMem = shmat(shmId, NULL, 0);
if (sharedMem == NULL) {
    shmctl(shmId, IPC_RMID, NULL);
    printf("CHILD - Failed to attach shared memory\n");
    exit(1);
}
return;
}

/*****
/* wait for child to complete and get return code. */
/* Destroy read/write lock in shared memory */
/* detach and remove shared memory */
/* set the child's return code in global storage */
/* */
/* If this function fails, it will call exit(1) */
/* */
/* This function sets the following global variables: */
/*     sharedMem */
/*     childStatus */
/*     shmId */
/*****
void parentCleanup(void)
{
    int          status=0;
    int          rc;
    int          waitedPid=0;

    /* Even though there is no thread left in the child using the */
    /* contents of the shared memory, before we destroy the */
    /* read/write lock in that shared memory, we will wait for the */
    /* child job to complete, we know for 100% certainty that no */
    /* threads in the child are using it then, because the child */
    /* is terminated. */
    printf("PARENT - Parent cleanup\n");
    /* Wait for the child to complete */
    waitedPid = waitpid(childPid,&status,0);
    if (rc == -1) {
        printf("PARENT - waitpid failed, errno=%d\n", errno);

```

```

        exit(1);
    }
    childStatus = status;

    /* Cleanup resources */
    rc = pthread_rwlock_destroy(&sharedMem->rwlock);
    checkResults("pthread_rwlock_destroy()\n", rc);

    /* Detach/Remove shared memory */
    rc = shmdt(sharedMem);
    if (rc) {
        printf("PARENT - Failed to detach shared memory, errno=%d\n", errno);
        exit(1);
    }
    rc = shmctl(shmid, IPC_RMID, NULL);
    if (rc) {
        printf("PARENT - Failed to remove shared memory id=%d, errno=%d\n",
            shmid, errno);
        exit(1);
    }
    shmid = 0;
    return;
}

/*****
/* Detach the shared memory
/* At this point, there is no serialization, so the contents
/* of the shared memory should not be used.
/*
/* If this function fails, it will call exit(1)
/*
/* This function sets the following global variables:
/*     sharedMem
*****/
void childCleanup(void)
{
    int rc;

    printf("CHILD - Child cleanup\n");
    rc = shmdt(sharedMem);
    sharedMem = NULL;
    if (rc) {
        printf("CHILD - Failed to detach shared memory, errno=%d\n", errno);
        exit(1);
    }
    return;
}

```

Output:

This example was run under the OS/400 QShell Interpreter. In the QShell Interpreter, a program gets descriptors 0, 1, 2 as the standard files, the parent and child I/O is directed to the console. When run in the QShell Interpreter, the output shows the intermixed output from both parent and child processes, and gives a feeling for the time sequence of operations occurring in each job.

The QShell interpreter allows you to run multithreaded programs as if they were interactive. See the QShell documentation for a description of the QIBM_MULTI_THREADED shell variable which allows you to start multithreaded programs.

The QShell Interpreter is option 30 of Base OS/400.

[illegible]

pthread_rwlockattr_init()--Initialize Read/Write Lock Attribute

Syntax

```
#include <pthread.h>
int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);
Threadsafe: Yes
Signal Safe: Yes
```

The **pthread_rwlockattr_init()** function initializes the read/write lock attributes object referred to by *attr* to the default attributes. The read/write lock attributes object can be used in a call to **pthread_rwlock_init()** to create a read/write lock.

Parameters

attr

(Output) Address of the variable to contain the read/write lock attributes object

Authorities and Locks

None.

Return Value

0

pthread_rwlockattr_init() was successful.

value

pthread_rwlockattr_init() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_rwlockattr_init()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

Related Information

- The **<pthread.h>** header file. See [Header files for Pthread functions](#).
- [pthread_rwlockattr_destroy\(\)--Destroy Read/Write Lock Attribute](#)
- [pthread_rwlock_init\(\)--Initialize Read/Write Lock](#)

Example

See the [pthread_rwlockattr_destroy\(\) example](#).

pthread_rwlockattr_setpshared()--Set Pshared Read/Write Lock Attribute

Syntax

```
#include <pthread.h>
int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr, int pshared);
Threadsafe: Yes
Signal Safe: Yes
```

The **pthread_rwlockattr_setpshared()** function sets the current pshared attribute for the read/write attributes object. The process shared attribute indicates whether the read/write lock that is created using the read/write lock attributes object can be shared between threads in separate processes (**PTHREAD_PROCESS_SHARED**) or shared only between threads in the same process (**PTHREAD_PROCESS_PRIVATE**).

Even if the read/write lock in storage is accessible from two separate processes, it cannot be used from both processes unless the process shared attribute is **PTHREAD_PROCESS_SHARED**.

Parameters

attr

(Input) Address of the variable containing the read/write lock attributes object

pshared

(Input) One of **PTHREAD_PROCESS_SHARED** or **PTHREAD_PROCESS_PRIVATE**

Authorities and Locks

None.

Return Value

0

pthread_rwlockattr_setpshared() was successful.

value

pthread_rwlockattr_setpshared() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_rwlockattr_setpshared()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

Related Information

- The **<pthread.h>** header file. See [Header files for Pthread functions](#).
- [pthread_rwlockattr_init\(\)--Initialize Read/Write Lock Attribute](#)
- [pthread_rwlockattr_getpshared\(\)--Get Pshared Read/Write Lock Attribute](#)
- [pthread_rwlock_init\(\)--Initialize Read/Write Lock](#)

Example

See the [pthread_rwlockattr_getpshared\(\) example](#).

pthread_rwlock_destroy()--Destroy Read/Write Lock

Syntax

```
#include <pthread.h>
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
Threadsafe: Yes
Signal Safe: Yes
```

The **pthread_rwlock_destroy()** function destroys the named read/write lock. The destroyed read/write lock can no longer be used.

If **pthread_rwlock_destroy()** is called on a read/write lock on a mutex that is locked by another thread for either reading or writing, the request fails with an **EBUSY** error.

If **pthread_rwlock_destroy()** is used by a thread when it owns the read/write lock, and other threads are waiting for the read/write lock to become available (with calls to **pthread_rwlock_rdlock()**, **pthread_rwlock_wrlock()**, **pthread_rwlock_timedrdlock_np()** or **pthread_rwlock_timedwrlock_np()** APIs), the read/write lock is destroyed safely, and the waiting threads wake up with the **EDESTROYED** error. Threads calling **pthread_rwlock_tryrdlock()** or **pthread_rwlock_trywrlock()** return with either the **EBUSY** or **EINVAL** error, depending on when they called those functions.

Once a read/write lock is created, it cannot be validly copied or moved to a new location.

Parameters

rwlock

(Input) Address of the read/write lock to be destroyed

Authorities and Locks

None.

Return Value

0

pthread_rwlock_destroy() was successful.

value

pthread_rwlock_destroy() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_rwlock_destroy()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

Related Information

- The **<pthread.h>** header file. See [Header files for Pthread functions](#).
- [pthread_rwlock_init\(\)--Initialize Read/Write Lock](#)

Example

See the [pthread_rwlock_init\(\) example](#).

pthread_rwlock_init()--Initialize Read/Write Lock

Syntax

```
#include <pthread.h>
int pthread_rwlock_init(pthread_rwlock_t *rwlock,
                        const pthread_rwlockattr_t *attr);
```

Threadsafe: Yes

Signal Safe: Yes

The **pthread_rwlock_init()** function initializes a new read/write lock with the specified attributes for use. The new read/write lock may be used immediately for serializing critical resources. If *attr* is specified as **NULL**, all attributes are set to the default read/write lock attributes for the newly created read/write lock.

With these declarations and initializations:

```
pthread_rwlock_t      rwlock2;
pthread_rwlock_t      rwlock3;
pthread_rwlockattr_t  attr;
pthread_rwlockattr_init(&attr);
```

The following three read/write lock initialization mechanisms have equivalent function.

```
pthread_rwlock_t      rwlock1 = PTHREAD_RWLOCK_INITIALIZER;
pthread_rwlock_init(&rwlock2, NULL);
pthread_rwlock_init(&rwlock, &attr);
```

All three read/write locks are created with the default read/write lock attributes.

Every read/write lock must eventually be destroyed with **pthread_rwlock_destroy()**. Always use **pthread_rwlock_destroy()** before freeing or reusing read/write lock storage.

Parameters

rwlock

(Output) The address of the variable to contain a read/write lock

attr

(Input) The address of the variable containing the read/write lock attributes object

Authorities and Locks

None.

Return Value

0

pthread_rwlock_init() was successful.

value

pthread_rwlock_init() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_rwlock_init()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

Related Information

- The `<pthread.h>` header file. See [Header files for Pthread functions](#).
- [pthread_rwlockattr_init\(\)--Initialize Read/Write Lock Attribute](#)
- [pthread_rwlock_destroy\(\)--Destroy Read/Write Lock](#)

Example

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include "check.h"

pthread_rwlock_t      rwlock;

void *rdlockThread(void *arg)
{
    int rc;

    printf("Entered thread, getting read lock\n");
    rc = pthread_rwlock_rdlock(&rwlock);
    checkResults("pthread_rwlock_rdlock()\n", rc);
    printf("got the rwlock read lock\n");

    sleep(5);

    printf("unlock the read lock\n");
    rc = pthread_rwlock_unlock(&rwlock);
    checkResults("pthread_rwlock_unlock()\n", rc);
    printf("Secondary thread unlocked\n");
    return NULL;
}

void *wrlockThread(void *arg)
{
    int rc;

    printf("Entered thread, getting write lock\n");
    rc = pthread_rwlock_wrlock(&rwlock);
    checkResults("pthread_rwlock_wrlock()\n", rc);

    printf("Got the rwlock write lock, now unlock\n");
    rc = pthread_rwlock_unlock(&rwlock);
    checkResults("pthread_rwlock_unlock()\n", rc);
    printf("Secondary thread unlocked\n");
    return NULL;
}

int main(int argc, char **argv)
{
    int          rc=0;
    pthread_t     thread, thread1;

    printf("Enter Testcase - %s\n", argv[0]);

    printf("Main, initialize the read write lock\n");
    rc = pthread_rwlock_init(&rwlock, NULL);
```



```

    checkResults("pthread_rwlock_init()\n", rc);

    printf("Main, grab a read lock\n");
    rc = pthread_rwlock_rdlock(&rwlock);
    checkResults("pthread_rwlock_rdlock()\n",rc);

    printf("Main, grab the same read lock again\n");
    rc = pthread_rwlock_rdlock(&rwlock);
    checkResults("pthread_rwlock_rdlock() second\n", rc);

    printf("Main, create the read lock thread\n");
    rc = pthread_create(&thread, NULL, rdlockThread, NULL);
    checkResults("pthread_create\n", rc);

    printf("Main - unlock the first read lock\n");
    rc = pthread_rwlock_unlock(&rwlock);
    checkResults("pthread_rwlock_unlock()\n", rc);

    printf("Main, create the write lock thread\n");
    rc = pthread_create(&thread1, NULL, wrlockThread, NULL);
    checkResults("pthread_create\n", rc);

    sleep(5);
    printf("Main - unlock the second read lock\n");
    rc = pthread_rwlock_unlock(&rwlock);
    checkResults("pthread_rwlock_unlock()\n", rc);

    printf("Main, wait for the threads\n");
    rc = pthread_join(thread, NULL);
    checkResults("pthread_join\n", rc);

    rc = pthread_join(thread1, NULL);
    checkResults("pthread_join\n", rc);

    rc = pthread_rwlock_destroy(&rwlock);
    checkResults("pthread_rwlock_destroy()\n", rc);

    printf("Main completed\n");
    return 0;
}

```

Output:

```

Enter Testcase - QP0WTEST/TPRWLINIO
Main, initialize the read write lock
Main, grab a read lock
Main, grab the same read lock again
Main, create the read lock thread
Main - unlock the first read lock
Main, create the write lock thread
Entered thread, getting read lock
got the rwlock read lock
Entered thread, getting write lock
Main - unlock the second read lock
Main, wait for the threads
unlock the read lock
Secondary thread unlocked
Got the rwlock write lock, now unlock
Secondary thread unlocked
Main completed

```

pthread_rwlock_rdlock()--Get Shared Read Lock

Syntax

```
#include <pthread.h>
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
Threadsafe: Yes
Signal Safe: Yes
```

The **pthread_rwlock_rdlock()** function attempts to acquire a shared read lock on the read/write lock specified by *rwlock*.

Any number of threads can hold shared read locks on the same read/write lock object. If any thread holds an exclusive write lock on a read/write lock object, no other threads are allowed to hold a shared read or exclusive write lock.

If no threads are holding an exclusive write lock on the read/write lock, the calling thread successfully acquires the shared read lock.

If the calling thread already holds a shared read lock on the read/write lock, another read lock can be successfully acquired by the calling thread. If more than one shared read lock is successfully acquired by a thread on a read/write lock object, that thread is required to successfully call **pthread_rwlock_unlock()** a matching number of times.

With a large number of readers and relatively few writers, there is the possibility of writer starvation. If threads are waiting for an exclusive write lock on the read/write lock and there are threads that currently hold a shared read lock, the shared read lock request is granted.

If the read/write lock is destroyed while **pthread_rwlock_rdlock()** is waiting for the shared read lock, the **EDESTROYED** error is returned.

If a signal is delivered to the thread while it is waiting for the lock, the signal handler (if any) runs, and the thread resumes waiting.

Read/Write Lock Deadlocks

If a thread ends while holding a write lock, the attempt by another thread to acquire a shared read or exclusive write lock will not be successful. In this case, the attempt to acquire the lock will deadlock. If a thread ends while holding a read lock, the system automatically releases the read lock.

For the **pthread_rwlock_rdlock()** function, the pthreads run-time simulates the deadlock that has occurred in your application. When you are attempting to debug these deadlock scenarios, the CL command WRKJOB, option 20, shows the thread as in a condition wait. Displaying the call stack shows that the function **deadlockOnOrphanedRWLock** is in the call stack.

Upgrade / Downgrade a Lock

If the calling thread currently holds an exclusive write lock on the read/write lock object, the shared read lock request is granted. After the shared read lock request is granted, the calling thread holds **both** the shared read **and** the exclusive write lock for the specified read/write lock object. If the thread calls **pthread_rwlock_unlock()** while holding one or more shared read locks **and** one or more exclusive write locks, the exclusive write locks are unlocked first. If more than one outstanding exclusive write lock was held by the thread, a matching number of successful calls to **pthread_rwlock_unlock()** must be done before all write locks are unlocked. At that time, subsequent calls to **pthread_rwlock_unlock()** unlock the shared read locks.

This behavior can be used to allow your application to upgrade or downgrade one lock type to another. See [Read/write locks can be upgraded/downgraded](#).

Parameters

rwlock

(Input) The address of the read/write lock

Authorities and Locks

None.

Return Value

0

pthread_rwlock_rdlock() was successful.

value

pthread_rwlock_rdlock() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_rwlock_rdlock()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

[EDESTROYED]

The lock was destroyed while waiting.

Related Information

- The **<pthread.h>** header file. See [Header files for Pthread functions](#).
- [pthread_rwlock_init\(\)--Initialize Read/Write Lock](#)
- [pthread_rwlock_timedrdlock_np\(\)--Get Shared Read Lock with Time-out](#)
- [pthread_rwlock_timedwrlock_np\(\)--Get Exclusive Write Lock with Time-out](#)
- [pthread_rwlock_tryrdlock\(\)--Get Shared Read Lock with No Wait](#)
- [pthread_rwlock_trywrlock\(\)--Get Exclusive Write Lock with No Wait](#)
- [pthread_rwlock_unlock\(\)--Unlock Exclusive Write or Shared Read Lock](#)
- [pthread_rwlock_wrlock\(\)--Get Exclusive Write Lock](#)

Example

See the [pthread_rwlock_init\(\) example](#).

pthread_rwlock_timedrdlock_np()--Get Shared Read Lock with Time-Out

Syntax

```
#include <pthread.h>
int pthread_rwlock_timedrdlock_np(pthread_rwlock_t *rwlock,
                                   const struct timespec *deltatime);
```

Threadsafe: Yes

Signal Safe: Yes

The **pthread_rwlock_timedrdlock_np()** function attempts to acquire a shared read lock on the read/write lock specified by *rwlock*. If the shared read lock cannot be acquired in the *deltatime* specific, **pthread_rwlock_timedrdlock_np()** returns the **EBUSY** error.

Any number of threads can hold shared read locks on the same read/write lock object. If any thread holds an exclusive write lock on a read/write lock object, no other threads are allowed to hold a shared read or exclusive write lock.

If no threads are holding an exclusive write lock on the read/write lock, the calling thread successfully acquires the shared read lock.

If the calling thread already holds a shared read lock on the read/write lock, another read lock can be successfully acquired by the calling thread. If more than one shared read lock is successfully acquired by a thread on a read/write lock object, that thread is required to successfully call **pthread_rwlock_unlock()** a matching number of times.

With a large number of readers and relatively few writers, there is the possibility of writer starvation. If threads are waiting for an exclusive write lock on the read/write lock and there are threads that currently hold a shared read lock, the shared read lock request is granted.

If the read/write lock is destroyed while **pthread_rwlock_timedrdlock_np()** is waiting for the shared read lock, the **EDESTROYED** error is returned.

If a signal is delivered to the thread while it is waiting for the lock, the signal handler (if any) runs, and the thread resumes waiting. For a timed wait, when the thread resumes waiting after the signal handler runs, the wait time is reset. For example, suppose a thread specifies that it should wait for a lock for 5 seconds, and a signal handler runs in that thread after 2.5 seconds. After returning from the signal handler, the thread will resume its wait for another 5 seconds. The resulting wait is longer than the specified 5 seconds.

Read/Write Lock Deadlocks

If a thread ends while holding a write lock, the attempt by another thread to acquire a shared read or exclusive write lock will not succeed. In this case, the attempt to acquire the lock will return the **EBUSY** error after the specified time elapses for the lock operation. If a thread ends while holding a read lock, the system automatically releases the read lock.

For the **pthread_rwlock_timedrdlock_np()** function, the pthreads run-time simulates the deadlock that has occurred in your application. When you are attempting to debug these deadlock scenarios, the CL command WRKJOB, option 20, shows the thread as in a condition wait. Displaying the call stack shows that the function **timedDeadlockOnOrphanedRWLock** is in the call stack.

Upgrade / Downgrade a Lock

If the calling thread currently holds an exclusive write lock on the read/write lock object, the shared read lock request is granted. After the shared read lock request is granted, the calling thread holds **both** the shared read **and** the exclusive write lock for the specified read/write lock object. If the thread calls **pthread_rwlock_unlock()** while holding one or more shared read locks **and** one or more exclusive write locks, the exclusive write locks are unlocked first. If more than one outstanding exclusive write lock was held by the thread, a matching number of successful calls to **pthread_rwlock_unlock()** must be done before all write locks are unlocked. At that time, subsequent calls to

pthread_rwlock_unlock() will unlock the shared read locks.

You can use this behavior to allow your application to upgrade or downgrade one lock type to another. See [Read/write locks can be upgraded/downgraded](#).

Parameters

rwlock

(Input) The address of the read/write lock

deltatime

(Input) The number of seconds and nanoseconds to wait for the lock before returning an error

Authorities and Locks

None.

Return Value

0

pthread_rwlock_timedrdlock_np() was successful.

value

pthread_rwlock_timedrdlock_np() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_rwlock_timedrdlock_np()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

[EBUSY]

The lock could not be acquired in the time specified.

[EDESTROYED]

The lock was destroyed while waiting.

Related Information

- The <pthread.h> header file. See [Header files for Pthread functions](#).
- [pthread_rwlock_init\(\)--Initialize Read/Write Lock](#)
- [pthread_rwlock_rdlock\(\)--Get Shared Read Lock](#)
- [pthread_rwlock_timedwrlock_np\(\)--Get Exclusive Write Lock with Time-Out](#)
- [pthread_rwlock_tryrdlock\(\)--Get Shared Read Lock with No Wait](#)
- [pthread_rwlock_trywrlock\(\)--Get Exclusive Write Lock with No Wait](#)
- [pthread_rwlock_unlock\(\)--Unlock Exclusive Write or Shared Read Lock](#)
- [pthread_rwlock_wrlock\(\)--Get Exclusive Write Lock](#)

Example

```

#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include "check.h"

pthread_rwlock_t      rwlock = PTHREAD_RWLOCK_INITIALIZER;

void *rdlockThread(void *arg)
{
    int          rc;
    int          count=0;
    struct timespec ts;

    /* 1.5 seconds */
    ts.tv_sec = 1;
    ts.tv_nsec = 500000000;

    printf("Entered thread, getting read lock with timeout\n");
Retry:
    rc = pthread_rwlock_timedrdlock_np(&rwlock, &ts);
    if (rc == EBUSY) {
        if (count >= 10) {
            printf("Retried too many times, failure!\n");
            exit(EXIT_FAILURE);
        }
        ++count;
        printf("RETRY...\n");
        goto Retry;
    }
    checkResults("pthread_rwlock_rdlock() 1\n", rc);

    sleep(2);

    printf("unlock the read lock\n");
    rc = pthread_rwlock_unlock(&rwlock);
    checkResults("pthread_rwlock_unlock()\n", rc);

    printf("Secondary thread complete\n");
    return NULL;
}

int main(int argc, char **argv)
{
    int          rc=0;
    pthread_t     thread;

    printf("Enter Testcase - %s\n", argv[0]);

    printf("Main, get the write lock\n");
    rc = pthread_rwlock_wrlock(&rwlock);
    checkResults("pthread_rwlock_wrlock()\n", rc);

    printf("Main, create the timed rd lock thread\n");
    rc = pthread_create(&thread, NULL, rdlockThread, NULL);
    checkResults("pthread_create\n", rc);

    printf("Main, wait a bit holding the write lock\n");
    sleep(5);

```

```
printf("Main, Now unlock the write lock\n");
rc = pthread_rwlock_unlock(&rwlock);
checkResults("pthread_rwlock_unlock()\n", rc);

printf("Main, wait for the thread to end\n");
rc = pthread_join(thread, NULL);
checkResults("pthread_join\n", rc);

rc = pthread_rwlock_destroy(&rwlock);
checkResults("pthread_rwlock_destroy()\n", rc);
printf("Main completed\n");
return 0;
}
```

Output:

```
Enter Testcase - QP0WTEST/TPRWLRD0
Main, get the write lock
Main, create the timed rd lock thread
Main, wait a bit
Entered thread, getting read lock with timeout
RETRY...
RETRY...
RETRY...
Main, Now unlock the write lock
Main, wait for the thread to end
unlock the read lock
Secondary thread complete
Main completed
```

pthread_rwlock_timedwrlock_np()--Get Exclusive Write Lock with Time-Out

Syntax

```
#include <pthread.h>
int pthread_rwlock_timedwrlock_np(pthread_rwlock_t *rwlock,
                                   const struct timespec *deltatime);
```

Threadsafe: Yes

Signal Safe: Yes

The **pthread_rwlock_timedwrlock_np()** function attempts to acquire an exclusive write lock on the read/write lock specified by *rwlock*. If the exclusive write lock cannot be acquired in the *deltatime* specific, **pthread_rwlock_timedwrlock_np()** returns the **EBUSY** error.

Only one thread can hold an exclusive write lock on a read/write lock object. If any thread holds an exclusive write lock on a read/write lock object, no other threads are allowed to hold a shared read or exclusive write lock.

If no threads are holding an exclusive write lock or shared read lock on the read/write lock, the calling thread successfully acquires the exclusive write lock.

If the calling thread already holds an exclusive write lock on the read/write lock, another write lock can be successfully acquired by the calling thread. If more than one exclusive write lock is successfully acquired by a thread on a read/write lock object, that thread is required to successfully call **pthread_rwlock_unlock()** a matching number of times.

With a large number of readers and relatively few writers, there is the possibility of writer starvation. If threads are waiting for an exclusive write lock on the read/write lock and there are threads that currently hold a shared read lock, the subsequent attempts to acquire a shared read lock request are granted, while attempts to acquire the exclusive write lock wait.

If the read/write lock is destroyed while **pthread_rwlock_timedwrlock_np()** is waiting for the shared read lock, the **EDESTROYED** error is returned.

If a signal is delivered to the thread while it is waiting for the lock, the signal handler (if any) runs, and the thread resumes waiting. For a timed wait, when the thread resumes waiting after the signal handler runs, the wait time is reset. For example, suppose a thread specifies that it should wait for a lock for 5 seconds, and a signal handler runs in that thread after 2.5 seconds. After returning from the signal handler, the thread resumes its wait for another 5 seconds. The resulting wait is longer than the specified 5 seconds.

Read/Write Lock Deadlocks

If a thread ends while holding a write lock, the attempt by another thread to acquire a shared read or exclusive write lock will not succeed. In this case, the attempt to acquire the lock returns the **EBUSY** error after the specified time elapses for the lock operation. If a thread ends while holding a read lock, the system automatically releases the read lock.

For the **pthread_rwlock_timedwrlock_np()** function, the pthreads run-time simulates the deadlock that has occurred in your application. When you are attempting to debug these deadlock scenarios, the CL command WRKJOB, option 20, shows the thread as in a condition wait. Displaying the call stack shows that the function **timedDeadlockOnOrphanedRWLock** is in the call stack.

Upgrade / Downgrade a Lock

If the calling thread currently holds a shared read lock on the read/write lock object and no other threads are holding a shared read lock, the exclusive write request is granted. After the exclusive write lock request is granted, the calling thread holds **both** the shared read **and** the exclusive write lock for the specified read/write lock object. If the thread calls **pthread_rwlock_unlock()** while holding one or more shared read locks **and** one or more exclusive write locks,

the exclusive write locks are unlocked first. If more than one outstanding exclusive write lock was held by the thread, a matching number of successful calls to **pthread_rwlock_unlock()** must be done before all write locks are unlocked. At that time, subsequent calls to **pthread_rwlock_unlock()** unlock the shared read locks.

You can use this behavior to allow your application to upgrade or downgrade one lock type to another. See [Read/write locks can be upgraded and downgraded.](#)

Parameters

rwlock

(Input) The address of the read/write lock

deltatime

(Input) The number of seconds and nanoseconds to wait for the lock before returning an error

Authorities and Locks

None.

Return Value

0

pthread_rwlock_timedwrlock_np() was successful.

value

pthread_rwlock_timedwrlock_np() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_rwlock_timedwrlock_np()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

[EBUSY]

The lock could not be acquired in the time specified.

[EDESTROYED]

The lock was destroyed while waiting.

Related Information

- The <pthread.h> header file. See [Header files for Pthread functions.](#)
- [pthread_rwlock_init\(\)--Initialize Read/Write Lock](#)
- [pthread_rwlock_rdlock\(\)--Get Shared Read Lock](#)
- [pthread_rwlock_timedrdlock_np\(\)--Get Shared Read Lock with Time-Out](#)
- [pthread_rwlock_tryrdlock\(\)--Get Shared Read Lock with No Wait](#)
- [pthread_rwlock_trywrlock\(\)--Get Exclusive Write Lock with No Wait](#)
- [pthread_rwlock_unlock\(\)--Unlock Exclusive Write or Shared Read Lock](#)
- [pthread_rwlock_wrlock\(\)--Get Exclusive Write Lock](#)

Example

```

#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include "check.h"

pthread_rwlock_t      rwlock = PTHREAD_RWLOCK_INITIALIZER;

void *wrlockThread(void *arg)
{
    int          rc;
    int          count=0;
    struct timespec ts;

    /* 1.5 seconds */
    ts.tv_sec = 1;
    ts.tv_nsec = 500000000;

    printf("%.8x %.8x: Entered thread, getting write lock with timeout\n",
           pthread_getthreadid_np());
    Retry:
    rc = pthread_rwlock_timedwrlock_np(&rwlock, &ts);
    if (rc == EBUSY) {
        if (count >= 10) {
            printf("%.8x %.8x: Retried too many times, failure!\n",
                   pthread_getthreadid_np());
            exit(EXIT_FAILURE);
        }
        ++count;
        printf("%.8x %.8x: RETRY...\n", pthread_getthreadid_np());
        goto Retry;
    }
    checkResults("pthread_rwlock_wrlock() 1\n", rc);
    printf("%.8x %.8x: Got the write lock\n", pthread_getthreadid_np());

    sleep(2);

    printf("%.8x %.8x: Unlock the write lock\n",
           pthread_getthreadid_np());
    rc = pthread_rwlock_unlock(&rwlock);
    checkResults("pthread_rwlock_unlock()\n", rc);

    printf("%.8x %.8x: Secondary thread complete\n",
           pthread_getthreadid_np());
    return NULL;
}

int main(int argc, char **argv)
{
    int          rc=0;
    pthread_t     thread, thread2;

    printf("Enter Testcase - %s\n", argv[0]);

    printf("Main, get the write lock\n");
    rc = pthread_rwlock_wrlock(&rwlock);
    checkResults("pthread_rwlock_wrlock()\n", rc);

    printf("Main, create the timed write lock threads\n");

```

pthread_rwlock_timedwrlock_np()--Get Exclusive Write Lock with Time-Out

```
rc = pthread_create(&thread, NULL, wrlockThread, NULL);
checkResults("pthread_create\n", rc);

rc = pthread_create(&thread2, NULL, wrlockThread, NULL);
checkResults("pthread_create\n", rc);

printf("Main, wait a bit holding this write lock\n");
sleep(3);

printf("Main, Now unlock the write lock\n");
rc = pthread_rwlock_unlock(&rwlock);
checkResults("pthread_rwlock_unlock()\n", rc);

printf("Main, wait for the threads to end\n");
rc = pthread_join(thread, NULL);
checkResults("pthread_join\n", rc);

rc = pthread_join(thread2, NULL);
checkResults("pthread_join\n", rc);

rc = pthread_rwlock_destroy(&rwlock);
checkResults("pthread_rwlock_destroy()\n", rc);
printf("Main completed\n");
return 0;
}
```

Output:

```
Enter Testcase - QP0WTEST/TPRWLWR0
Main, get the write lock
Main, create the timed write lock threads
Main, wait a bit holding this write lock
00000000 00000017: Entered thread, getting write lock with timeout
00000000 00000018: Entered thread, getting write lock with timeout
00000000 00000017: RETRY...
00000000 00000018: RETRY...
Main, Now unlock the write lock
Main, wait for the threads to end
00000000 00000017: Got the write lock
00000000 00000018: RETRY...
00000000 00000018: RETRY...
00000000 00000017: Unlock the write lock
00000000 00000017: Secondary thread complete
00000000 00000018: Got the write lock
00000000 00000018: Unlock the write lock
00000000 00000018: Secondary thread complete
Main completed
```

pthread_rwlock_tryrdlock()--Get Shared Read Lock with No Wait

Syntax

```
#include <pthread.h>
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
Threadsafe: Yes
Signal Safe: Yes
```

The **pthread_rwlock_tryrdlock()** function attempts to acquire a shared read lock on the read/write lock specified by *rwlock*. If the shared read lock cannot be acquired immediately, **pthread_rwlock_tryrdlock()** returns the **EBUSY** error.

Any number of threads can hold shared read locks on the same read/write lock object. If any thread holds an exclusive write lock on a read/write lock object, no other threads will be allowed to hold a shared read or exclusive write lock.

If there are no threads holding an exclusive write lock on the read/write lock, the calling thread will successfully acquire the shared read lock.

If the calling thread already holds a shared read lock on the read/write lock, another read lock can be successfully acquired by the calling thread. If more than one shared read lock is successfully acquired by a thread on a read/write lock object, that thread is required to successfully call **pthread_rwlock_unlock()** a matching number of times.

With a large number of readers, and relatively few writers, there is the possibility of writer starvation. If there are threads waiting for an exclusive write lock on the read/write lock and there are threads that currently hold a shared read lock, the shared read lock request will be granted.

Read/Write Lock Deadlocks

If a thread ends while holding a write lock, the attempt by another thread to acquire a shared read or exclusive write lock will not be successful. In this case, the attempt to acquire the lock will return the **EBUSY** error. If a thread ends while holding a read lock, the system automatically releases the read lock.

Upgrade / Downgrade a Lock

If the calling thread currently holds an exclusive write lock on the read/write lock object, the shared read lock request will be granted. After the shared read lock request is granted, the calling thread holds **both** the shared read, **and** the exclusive write lock for the specified read/write lock object. If the thread calls **pthread_rwlock_unlock()** while holding one or more shared read locks **and** one or more exclusive write locks, the exclusive write locks are unlocked first. If more than one outstanding exclusive write lock was held by the thread, a matching number of successful calls to **pthread_rwlock_unlock()** must be done before all write locks are unlocked. At that time, subsequent calls to **pthread_rwlock_unlock()** will unlock the shared read locks.

This behavior can be used to allow your application to upgrade or downgrade one lock type to another. See [Read/write locks can be upgraded/downgraded.](#)

Parameters

rwlock

(Input) The address of the read/write lock

Authorities and Locks

None.

Return Value

0

pthread_rwlock_tryrdlock() was successful.

value

pthread_rwlock_tryrdlock() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_rwlock_tryrdlock()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

[EBUSY]

The lock could not be immediately acquired.

Related Information

- The `<pthread.h>` header file. See [Header files for Pthread functions](#).
- [pthread_rwlock_init\(\)--Initialize Read/Write Lock](#)
- [pthread_rwlock_rdlock\(\)--Get Shared Read Lock](#)
- [pthread_rwlock_timedrdlock_np\(\)--Get Shared Read Lock with Time-Out](#)
- [pthread_rwlock_timedwrlock_np\(\)--Get Exclusive Write Lock with Time-Out](#)
- [pthread_rwlock_trywrlock\(\)--Get Exclusive Write Lock with No Wait](#)
- [pthread_rwlock_unlock\(\)--Unlock Exclusive Write or Shared Read Lock](#)
- [pthread_rwlock_wrlock\(\)--Get Exclusive Write Lock](#)

Example

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include "check.h"

pthread_rwlock_t      rwlock = PTHREAD_RWLOCK_INITIALIZER;

void *rdlockThread(void *arg)
{
    int          rc;
    int          count=0;

    printf("Entered thread, getting read lock with mp wait\n");
    Retry:
    rc = pthread_rwlock_tryrdlock(&rwlock);
    if (rc == EBUSY) {
        if (count >= 10) {
```

pthread_rwlock_tryrdlock())--Get Shared Read Lock with No Wait

```
        printf("Retried too many times, failure!\n");

        exit(EXIT_FAILURE);
    }
    ++count;
    printf("Could not get lock, do other work, then RETRY...\n");
    sleep(1);
    goto Retry;
}
checkResults("pthread_rwlock_tryrdlock() 1\n", rc);

sleep(2);

printf("unlock the read lock\n");
rc = pthread_rwlock_unlock(&rwlock);
checkResults("pthread_rwlock_unlock()\n", rc);

printf("Secondary thread complete\n");
return NULL;
}

int main(int argc, char **argv)
{
    int                rc=0;
    pthread_t          thread;

    printf("Enter Testcase - %s\n", argv[0]);

    printf("Main, get the write lock\n");
    rc = pthread_rwlock_wrlock(&rwlock);
    checkResults("pthread_rwlock_wrlock()\n", rc);

    printf("Main, create the try read lock thread\n");
    rc = pthread_create(&thread, NULL, rdlockThread, NULL);
    checkResults("pthread_create\n", rc);

    printf("Main, wait a bit holding the write lock\n");
    sleep(5);

    printf("Main, Now unlock the write lock\n");
    rc = pthread_rwlock_unlock(&rwlock);
    checkResults("pthread_rwlock_unlock()\n", rc);

    printf("Main, wait for the thread to end\n");
    rc = pthread_join(thread, NULL);
    checkResults("pthread_join\n", rc);

    rc = pthread_rwlock_destroy(&rwlock);
    checkResults("pthread_rwlock_destroy()\n", rc);
    printf("Main completed\n");
    return 0;
}
```

Output

```
Enter Testcase - QP0WTEST/TPRWLRD1
Main, get the write lock
Main, create the try read lock thread
Main, wait a bit holding the write lock
```

pthread_rwlock_tryrdlock()--Get Shared Read Lock with No Wait

```
Entered thread, getting read lock with mp wait
Could not get lock, do other work, then RETRY...
Could not get lock, do other work, then RETRY...
Could not get lock, do other work, then RETRY...
Could not get lock, do other work, then RETRY...
Could not get lock, do other work, then RETRY...
Main, Now unlock the write lock
Main, wait for the thread to end
unlock the read lock
Secondary thread complete
Main completed
```

pthread_rwlock_trywrlock()--Get Exclusive Write Lock with No Wait

Syntax

```
#include <pthread.h>
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
Threadsafe: Yes
Signal Safe: Yes
```

The **pthread_rwlock_trywrlock()** function attempts to acquire an exclusive write lock on the read/write lock specified by *rwlock*. If the exclusive write lock cannot be immediately acquired, **pthread_rwlock_timedwrlock_np()** returns the **EBUSY** error.

Only one thread can hold an exclusive write lock on a read/write lock object. If any thread holds an exclusive write lock on a read/write lock object, no other threads will be allowed to hold a shared read or exclusive write lock.

If there are no threads holding an exclusive write lock or shared read lock on the read/write lock, the calling thread will successfully acquire the exclusive write lock.

If the calling thread already holds an exclusive write lock on the read/write lock, another write lock can be successfully acquired by the calling thread. If more than one exclusive write lock is successfully acquired by a thread on a read/write lock object, that thread is required to successfully call **pthread_rwlock_unlock()** a matching number of times.

With a large number of readers, and relatively few writers, there is the possibility of writer starvation. If there are threads waiting for an exclusive write lock on the read/write lock and there are threads that currently hold a shared read lock, the subsequent attempts to acquire a shared read lock request will be granted, while attempts to acquire the exclusive write lock will return the **EBUSY** error.

Read/Write Lock Deadlocks

If a thread ends while holding a write lock, the attempt by another thread to acquire a shared read or exclusive write lock will not be successful. In this case, the attempt to acquire the lock will return the **EBUSY** error. If a thread ends while holding a read lock, the system automatically releases the read lock.

Upgrade / Downgrade a Lock

If the calling thread currently holds a shared read lock on the read/write lock object and there are no other threads holding a shared read lock, the exclusive write request will be granted. After the exclusive write lock request is granted, the calling thread holds **both** the shared read, **and** the exclusive write lock for the specified read/write lock object. If the thread calls **pthread_rwlock_unlock()** while holding one or more shared read locks **and** one or more exclusive write locks, the exclusive write locks are unlocked first. If more than one outstanding exclusive write lock was held by the thread, a matching number of successful calls to **pthread_rwlock_unlock()** must be done before all write locks are unlocked. At that time, subsequent calls to **pthread_rwlock_unlock()** will unlock the shared read locks.

This behavior can be used to allow your application to upgrade or downgrade one lock type to another. See [Read/write locks can be upgraded/downgraded.](#)

Parameters

rwlock

(Input) The address of the read/write lock

Authorities and Locks

None.

Return Value

0

pthread_rwlock_trywrlock() was successful.

value

pthread_rwlock_trywrlock() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_rwlock_trywrlock()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

[EBUSY]

The lock could not be acquired in the timed specified.

Related Information

- The `<pthread.h>` header file. See [Header files for Pthread functions](#).
- [pthread_rwlock_init\(\)--Initialize Read/Write Lock](#)
- [pthread_rwlock_rdlock\(\)--Get Shared Read Lock](#)
- [pthread_rwlock_timedrdlock_np\(\)--Get Shared Read Lock with Time-Out](#)
- [pthread_rwlock_timedwrlock_np\(\)--Get Exclusive Write Lock with Time-Out](#)
- [pthread_rwlock_tryrdlock\(\)--Get Shared Read Lock with No Wait](#)
- [pthread_rwlock_unlock\(\)--Unlock Exclusive Write or Shared Read Lock](#)
- [pthread_rwlock_wrlock\(\)--Get Exclusive Write Lock](#)

Example

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include "check.h"

pthread_rwlock_t      rwlock = PTHREAD_RWLOCK_INITIALIZER;

void *wrlockThread(void *arg)
{
    int          rc;
    int          count=0;

    printf("%.8x %.8x: Entered thread, getting write lock with timeout\n",
           pthread_getthreadid_np());
    Retry:
    rc = pthread_rwlock_trywrlock(&rwlock);
    if (rc == EBUSY) {
```

pthread_rwlock_trywrlock())--Get Exclusive Write Lock with No Wait

```
    if (count >= 10) {
        printf("%.8x %.8x: Retried too many times, failure!\n",
            pthread_getthreadid_np());
        exit(EXIT_FAILURE);
    }
    ++count;
    printf("%.8x %.8x: Go off an do other work, then RETRY...\n",
        pthread_getthreadid_np());
    sleep(1);
    goto Retry;
}
checkResults("pthread_rwlock_trywrlock() 1\n", rc);
printf("%.8x %.8x: Got the write lock\n", pthread_getthreadid_np());

sleep(2);

printf("%.8x %.8x: Unlock the write lock\n",
    pthread_getthreadid_np());
rc = pthread_rwlock_unlock(&rwlock);
checkResults("pthread_rwlock_unlock()\n", rc);

printf("%.8x %.8x: Secondary thread complete\n",
    pthread_getthreadid_np());
return NULL;
}

int main(int argc, char **argv)
{
    int                rc=0;
    pthread_t          thread, thread2;

    printf("Enter Testcase - %s\n", argv[0]);

    printf("Main, get the write lock\n");
    rc = pthread_rwlock_wrlock(&rwlock);
    checkResults("pthread_rwlock_wrlock()\n", rc);

    printf("Main, create the timed write lock threads\n");
    rc = pthread_create(&thread, NULL, wrlockThread, NULL);
    checkResults("pthread_create\n", rc);

    rc = pthread_create(&thread2, NULL, wrlockThread, NULL);
    checkResults("pthread_create\n", rc);

    printf("Main, wait a bit holding this write lock\n");
    sleep(1);

    printf("Main, Now unlock the write lock\n");
    rc = pthread_rwlock_unlock(&rwlock);
    checkResults("pthread_rwlock_unlock()\n", rc);

    printf("Main, wait for the threads to end\n");
    rc = pthread_join(thread, NULL);
    checkResults("pthread_join\n", rc);

    rc = pthread_join(thread2, NULL);
    checkResults("pthread_join\n", rc);

    rc = pthread_rwlock_destroy(&rwlock);
    checkResults("pthread_rwlock_destroy()\n", rc);
    printf("Main completed\n");
}
```

pthread_rwlock_trywrlock())--Get Exclusive Write Lock with No Wait

```
    return 0;  
}
```

Output:

```
Enter Testcase - QP0WTEST/TPRWLWR1  
Main, get the write lock  
Main, create the timed write lock threads  
00000000 0000000d: Entered thread, getting write lock with timeout  
00000000 0000000d: Go off an do other work, then RETRY...  
Main, wait a bit holding this write lock  
00000000 0000000e: Entered thread, getting write lock with timeout  
00000000 0000000e: Go off an do other work, then RETRY...  
00000000 0000000d: Go off an do other work, then RETRY...  
Main, Now unlock the write lock  
Main, wait for the threads to end  
00000000 0000000e: Got the write lock  
00000000 0000000d: Go off an do other work, then RETRY...  
00000000 0000000e: Unlock the write lock  
00000000 0000000e: Secondary thread complete  
00000000 0000000d: Got the write lock  
00000000 0000000d: Unlock the write lock  
00000000 0000000d: Secondary thread complete  
Main completed
```

pthread_rwlock_unlock()--Unlock Exclusive Write or Shared Read Lock

Syntax

```
#include <pthread.h>
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
Threadsafe: Yes
Signal Safe: Yes
```

The **pthread_rwlock_unlock()** function unlocks a shared read or exclusive write lock held by the calling thread.

A thread should call **pthread_rwlock_unlock()** once for each time that the thread successfully called **pthread_rwlock_rdlock()**, **pthread_rwlock_tryrdlock()**, **pthread_rwlock_trywrlock()**, **pthread_rwlock_timedrdlock_np()**, or **pthread_rwlock_timedwrlock_np()** to acquire a shared read or exclusive write lock. For example, if a thread holds 4 shared read locks on a read/write lock object, the thread must call **pthread_rwlock_unlock()** 4 times before the read/write lock becomes completely unlocked.

If a thread holds **both** shared read **and** exclusive write locks for the specified read/write lock object, the exclusive write locks are unlocked first. If more than one outstanding exclusive write lock was held by the thread, a matching number of successful calls to **pthread_rwlock_unlock()** must be done before all write locks are unlocked. When all write locks are unlocked, subsequent calls to **pthread_rwlock_unlock()** unlock the shared read locks.

Parameters

rwlock

(Input) The address of the read/write lock

Authorities and Locks

For successful completion, either a shared read or exclusive write lock must be held on the read/write lock before you call **pthread_rwlock_unlock()**.

Return Value

0

pthread_rwlock_unlock() was successful.

value

pthread_rwlock_unlock() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_rwlock_unlock()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

[EPERM]

A shared read or exclusive write lock was not held by the calling thread and could not be unlocked.

Related Information

- The <pthread.h> header file. See [Header files for Pthread functions](#).
- [pthread_rwlock_init\(\)](#)--Initialize Read/Write Lock
- [pthread_rwlock_rdlock\(\)](#)--Get Shared Read Lock
- [pthread_rwlock_timedrdlock_np\(\)](#)--Get Shared Read Lock with Time-Out
- [pthread_rwlock_timedwrlock_np\(\)](#)--Get Exclusive Write Lock with Time-Out
- [pthread_rwlock_tryrdlock\(\)](#)--Get Shared Read Lock with No Wait
- [pthread_rwlock_trywrlock\(\)](#)--Get Exclusive Write Lock with No Wait
- [pthread_rwlock_wrlock\(\)](#)--Get Exclusive Write Lock

Example

See any of the following examples:

- [pthread_rwlock_tryrdlock\(\)](#)
- [pthread_rwlock_trywrlock\(\)](#)
- [pthread_rwlock_timedrdlock_np\(\)](#)
- [pthread_rwlock_timedwrlock_np\(\)](#)

pthread_rwlock_wrlock()--Get Exclusive Write Lock

Syntax

```
#include <pthread.h>
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
Threadsafe: Yes
Signal Safe: Yes
```

The **pthread_rwlock_wrlock()** function attempts to acquire an exclusive write lock on the read/write lock specified by *rwlock*.

Only one thread can hold an exclusive write lock on a read/write lock object. If any thread holds an exclusive write lock on a read/write lock object, no other threads are allowed to hold a shared read or exclusive write lock.

If no threads are holding an exclusive write lock or shared read lock on the read/write lock, the calling thread successfully acquires the exclusive write lock.

If the calling thread already holds an exclusive write lock on the read/write lock, another write lock can be successfully acquired by the calling thread. If more than one exclusive write lock is successfully acquired by a thread on a read/write lock object, that thread is required to successfully call **pthread_rwlock_unlock()** a matching number of times.

With a large number of readers and relatively few writers, there is the possibility of writer starvation. If threads are waiting for an exclusive write lock on the read/write lock and there are threads that currently hold a shared read lock, the subsequent attempts to acquire a shared read lock request are granted, while attempts to acquire the exclusive write lock wait.

If the read/write lock is destroyed while **pthread_rwlock_wrlock()** is waiting for the shared read lock, the **EDESTROYED** error is returned.

If a signal is delivered to the thread while it is waiting for the lock, the signal handler (if any) runs, and the thread resumes waiting.

Read/Write Lock Deadlocks

If a thread ends while holding of a write lock, the attempt by another thread to acquire a shared read or exclusive write lock will not succeed. In this case, the attempt to acquire the lock does not return and will deadlock. If a thread ends while holding a read lock, the system automatically releases the read lock.

For the **pthread_rwlock_wrlock()** function, the pthreads run-time simulates the deadlock that has occurred in your application. When you are attempting to debug these deadlock scenarios, the CL command WRKJOB, option 20, shows the thread as in a condition wait. Displaying the call stack shows that the function **timedDeadlockOnOrphanedRWLock** is in the call stack.

Upgrade / Downgrade a Lock

If the calling thread currently holds a shared read lock on the read/write lock object and no other threads are holding a shared read lock, the exclusive write request is granted. After the exclusive write lock request is granted, the calling thread holds **both** the shared read, **and** the exclusive write lock for the specified read/write lock object. If the thread calls **pthread_rwlock_unlock()** while holding one or more shared read locks **and** one or more exclusive write locks, the exclusive write locks are unlocked first. If more than one outstanding exclusive write lock was held by the thread, a matching number of successful calls to **pthread_rwlock_unlock()** must be done before all write locks are unlocked. At that time, subsequent calls to **pthread_rwlock_unlock()** unlock the shared read locks.

You can use this behavior to allow your application to upgrade or downgrade one lock type to another. See [Read/write locks can be upgraded/downgraded](#).

Parameters

rwlock

(Input) The address of the read/write lock

Authorities and Locks

None.

Return Value

0

pthread_rwlock_wrlock() was successful.

value

pthread_rwlock_wrlock() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_rwlock_wrlock()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

[EDESTROYED]

The lock was destroyed while waiting.

Related Information

- The `<pthread.h>` header file. See [Header files for Pthread functions](#)
- [pthread_rwlock_init\(\)--Initialize a Read/Write Lock](#)
- [pthread_rwlock_rdlock\(\)--Get a Shared Read Lock](#)
- [pthread_rwlock_timedrdlock_np\(\)--Get a Shared Read Lock with Time-Out](#)
- [pthread_rwlock_timedwrlock_np\(\)--Get an Exclusive Write Lock with Time-Out](#)
- [pthread_rwlock_tryrdlock\(\)--Get a Shared Read Lock with No Wait](#)
- [pthread_rwlock_trywrlock\(\)--Get an Exclusive Write Lock with No Wait](#)
- [pthread_rwlock_unlock\(\)--Unlock an Exclusive Write or Shared Read Lock](#)

Example

See the [pthread_rwlock_init\(\) example](#).

Signals APIs

Signal APIs can be used to manipulate signals in a threaded process. Signals can be sent to individual threads, the signal mask of a thread can be changed. When a signal is sent to a thread, the actions associated with the signal (such as stopping, continuing or terminating) **never** affect only the thread, all signal actions are defined to affect the process. When a signal handler is invoked, it is invoked in the thread that the signal was delivered to.

Using signals correctly in a multithreaded process can be difficult. The recommended way to handle signals in a multithreaded process is to mask off all signals in all threads, then use the signals `sigwait()` API in a single thread to wait for any signal to be delivered to the process. For information about the examples included with the APIs, see the [information on the API examples](#).

To view the API list by description, see [Signals APIs](#).

Signals APIs by name

- [pthread_kill\(\)--Send Signal to Thread](#)
- [pthread_sigmask\(\)--Set or Get Signal Mask](#)
- [pthread_signal_to_cancel_np\(\)--Convert Signals to Cancel Requests](#)

pthread_kill()--Send Signal to Thread

Syntax

```
#include <pthread.h>
#include <sys/signal.h>
int pthread_kill(pthread_t thread, int sig);
Threadsafe: Yes
Signal Safe: No
```

The **pthread_kill()** function requests that the signal *sig* be delivered to the specified *thread*. The signal to be sent is specified by *sig* and is either zero or one of the signals from the list of defined signals in the **<sys/signal.h>** header file. If *sig* is zero, error checking is performed, but no signal is sent to the target *thread*.

A thread can use **pthread_kill()** to send a signal to itself. If the signal is not blocked or ignored, at least one pending unblocked signal is delivered to the sender before **pthread_kill()** returns. If there are no other pending unblocked signals, the delivered signal is *sig*.

The **pthread_kill()** API in no way changes the effect or scope of a signal. Even though a signal can be sent to a specific thread using the **pthread_kill()** API, the behavior that occurs when the signal is delivered is unchanged.

For example, sending a **SIGKILL** signal to a thread using **pthread_kill()** ends the entire process, not simply the target thread. **SIGKILL** is defined to end the entire process, regardless of the thread it is delivered to, or how it is sent.

Parameters

thread

(Input) Pthread handle of the target thread

sig

(Input) The signal number to be delivered or zero to validate the pthread_t

Authorities and Locks

None.

Return Value

0

pthread_kill() was successful.

value

pthread_kill() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_kill()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[ESRCH]

No thread could be found that matched the thread ID specified.

[EINVAL]

The value specified for the argument is not correct.

[ENOTSIGINIT]

The process is not enabled for signals.

Related Information

- The `<pthread.h>` header file. See [Header files for Pthread functions](#).
- [pthread_sigmask\(\)--Set or Get Signal Mask](#)
- [pthread_signal_to_cancel_np\(\)--Convert Signals to Cancel Requests](#)

Example

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include <sys/signal.h>
#include "check.h"

#define NUMTHREADS 3
void sighand(int signo);

void *threadfunc(void *parm)
{
    pthread_t          self = pthread_self();
    pthread_id_np_t    tid;
    int                rc;

    pthread_getunique_np(&self, &tid);
    printf("Thread 0x%.8x %.8x entered\n", tid);
    errno = 0;
    rc = sleep(30);
    if (rc != 0 && errno == EINTR) {
        printf("Thread 0x%.8x %.8x got a signal delivered to it\n",
            tid);
        return NULL;
    }
    printf("Thread 0x%.8x %.8x did not get expected results! rc=%d,
        errno=%d\n",
        tid, rc, errno);
    return NULL;
}

int main(int argc, char **argv)
{
    int                rc;
    int                i;
    struct sigaction   actions;
    pthread_t          threads[NUMTHREADS];

    printf("Enter Testcase - %s\n", argv[0]);

    printf("Set up the alarm handler for the process\n");
    memset(&actions, 0, sizeof(actions));
    sigemptyset(&actions.sa_mask);
    actions.sa_flags = 0;
    actions.sa_handler = sighand;

    rc = sigaction(SIGALRM, &actions, NULL);
    checkResults("sigaction\n", rc);
```

pthread_kill()--Send Signal to Thread

```
    for(i=0; i<NUMTHREADS; ++i) {
        rc = pthread_create(&threads[i], NULL, threadfunc, NULL);
        checkResults("pthread_create()\n", rc);
    }

    sleep(3);
    for(i=0; i<NUMTHREADS; ++i) {
        rc = pthread_kill(threads[i], SIGALRM);
        checkResults("pthread_kill()\n", rc);
    }

    for(i=0; i<NUMTHREADS; ++i) {
        rc = pthread_join(threads[i], NULL);
        checkResults("pthread_join()\n", rc);
    }
    printf("Main completed\n");
    return 0;
}

void sighand(int signo)
{
    pthread_t          self = pthread_self();
    pthread_id_np_t    tid;

    pthread_getunique_np(&self, &tid);
    printf("Thread 0x%.8x %.8x in signal handler\n",
          tid);
    return;
}
```

Output:

```
Enter Testcase - QP0WTEST/TPKILL0
Set up the alarm handler for the process
Thread 0x00000000 0000000c entered
Thread 0x00000000 0000000d entered
Thread 0x00000000 0000000e entered
Thread 0x00000000 0000000c in signal handler
Thread 0x00000000 0000000c got a signal delivered to it
Thread 0x00000000 0000000d in signal handler
Thread 0x00000000 0000000d got a signal delivered to it
Thread 0x00000000 0000000e in signal handler
Thread 0x00000000 0000000e got a signal delivered to it
Main completed
```

pthread_sigmask()--Set or Get Signal Mask

Syntax

```
#include <pthread.h>
#include <signal.h>
int pthread_sigmask(int how, const sigset_t *set,
                    sigset_t *oset);
```

Threadsafe: Yes

Signal Safe: Yes

The **pthread_sigmask()** function examines or modifies the signal blocking mask for the current thread.

The signals **SIGKILL** or **SIGSTOP** cannot be blocked. Any attempt to use **pthread_sigmask()** to block these signals is simply ignored, and no error is returned.

SIGFPE, **SIGILL**, and **SIGSEGV** signals that are not artificially generated by **kill()**, **pthread_kill()** or **raise()** (that is, were generated by the system as a result of a hardware or software exception) are not blocked.

If there are any pending unblocked signals after **pthread_sigmask()** has changed the signal mask, at least one of those signals is delivered to the process before **pthread_sigmask()** returns.

If **pthread_sigmask()** fails, the signal mask of the thread is not changed.

The possible values for *how*, which are defined in the **<sys/signal.h>** header file, are as follows:

SIG_BLOCK

Indicates that the set of signals given by *set* should be blocked, in addition to the set currently being blocked

SIG_UNBLOCK

Indicates that the set of signals given by *set* should not be blocked. These signals are removed from the current set of signals being blocked

SIG_SETMASK

Indicates that the set of signals given by *set* should replace the old set of signals being blocked

The *set* parameter points to a signal set that contains the new signals that should be blocked or unblocked (depending on the value of *how*), or it points to the new signal mask if the value of *how* is **SIG_SETMASK**. If *set* is a **NULL** pointer, the set of blocked signals is not changed. If *set* is **NULL**, the value of *how* is ignored.

The signal set manipulation functions (**sigemptyset()**, **sigfillset()**, **sigaddset()**, and **sigdelset()**) must be used to establish the new signal set pointed to by *set*.

The **pthread_sigmask()** function determines the current signal set and returns this information in **oset*. If *set* is **NULL**, *oset* returns the current set of signals being blocked. When *set* is not **NULL**, the set of signals pointed to by *oset* is the previous set.

Parameters

how

(Input) The way in which the signal set is changed

set

(Input) A pointer to a set of signals to be used to change the currently blocked set. This value can be **NULL**

oset

(Output) A pointer to the space where the previous signal mask is stored. This value can be **NULL**

Authorities and Locks

None.

Return Value

0

pthread_sigmask() was successful.

value

pthread_sigmask() was not successful. *value* is set to indicate the error condition.

Error Conditions

If **pthread_sigmask()** was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

[ENOTSIGINIT]

The process is not enabled for signals.

Related Information

- The `<pthread.h>` header file. See [Header files for Pthread functions](#).
- [pthread_kill\(\)--Send Signal to Thread](#)
- [pthread_signal_to_cancel_np\(\)--Convert Signals to Cancel Requests](#)

Example

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include <sys/signal.h>
#include "check.h"

#define NUMTHREADS 3
void sighand(int signo);

void *threadfunc(void *parm)
{
    pthread_t          self = pthread_self();
    pthread_id_np_t    tid;
    int                rc;

    pthread_getunique_np(&self, &tid);
    printf("Thread 0x%.8x %.8x entered\n", tid);
    errno = 0;
    rc = sleep(30);
    if (rc != 0 && errno == EINTR) {
        printf("Thread 0x%.8x %.8x got a signal delivered to it\n",
            tid);
        return NULL;
    }
    printf("Thread 0x%.8x %.8x did not get expected results! rc=%d,
```

```

    errno=%d\n",
        tid, rc, errno);
    return NULL;
}

void *threadmasked(void *parm)
{
    pthread_t          self = pthread_self();
    pthread_id_np_t    tid;
    sigset_t           mask;
    int                rc;

    pthread_getunique_np(&self, &tid);
    printf("Masked thread 0x%.8x %.8x entered\n", tid);

    sigfillset(&mask); /* Mask all allowed signals */
    rc = pthread_sigmask(SIG_BLOCK, &mask, NULL);
    checkResults("pthread_sigmask()\n", rc);

    errno = 0;
    rc = sleep(15);
    if (rc != 0) {
        printf("Masked thread 0x%.8x %.8x did not get expected results! "
            "rc=%d, errno=%d\n",
            tid, rc, errno);
        return NULL;
    }
    printf("Masked thread 0x%.8x %.8x completed masked work\n",
        tid);
    return NULL;
}

int main(int argc, char **argv)
{
    int                rc;
    int                i;
    struct sigaction   actions;
    pthread_t          threads[NUMTHREADS];
    pthread_t          maskedthreads[NUMTHREADS];

    printf("Enter Testcase - %s\n", argv[0]);

    printf("Set up the alarm handler for the process\n");
    memset(&actions, 0, sizeof(actions));
    sigemptyset(&actions.sa_mask);
    actions.sa_flags = 0;
    actions.sa_handler = sighand;

    rc = sigaction(SIGALRM, &actions, NULL);
    checkResults("sigaction\n", rc);

    printf("Create masked and unmasked threads\n");
    for(i=0; i<NUMTHREADS; ++i) {
        rc = pthread_create(&threads[i], NULL, threadfunc, NULL);
        checkResults("pthread_create()\n", rc);

        rc = pthread_create(&maskedthreads[i], NULL, threadmasked, NULL);
        checkResults("pthread_create()\n", rc);
    }

    sleep(3);

```

pthread_sigmask(--Set or Get Signal Mask

```
    printf("Send a signal to masked and unmasked threads\n");
    for(i=0; i<NUMTHREADS; ++i) {
        rc = pthread_kill(threads[i], SIGALRM);
        checkResults("pthread_kill()\n", rc);

        rc = pthread_kill(maskedthreads[i], SIGALRM);
        checkResults("pthread_kill()\n", rc);
    }

    printf("Wait for masked and unmasked threads to complete\n");
    for(i=0; i<NUMTHREADS; ++i) {
        rc = pthread_join(threads[i], NULL);
        checkResults("pthread_join()\n", rc);

        rc = pthread_join(maskedthreads[i], NULL);
        checkResults("pthread_join()\n", rc);
    }
    printf("Main completed\n");
    return 0;
}

void sighand(int signo)
{
    pthread_t          self = pthread_self();
    pthread_id_np_t    tid;

    pthread_getunique_np(&self, &tid);
    printf("Thread 0x%.8x %.8x in signal handler\n",
        tid);
    return;
}
```

Output:

```
Thread 0x00000000 0000000d entered
Masked thread 0x00000000 0000000a entered
Thread 0x00000000 00000009 entered
Thread 0x00000000 0000000b entered
Masked thread 0x00000000 0000000e entered
Masked thread 0x00000000 0000000c entered
Send a signal to masked and unmasked threads
Wait for masked and unmasked threads to complete
Thread 0x00000000 00000009 in signal handler
Thread 0x00000000 00000009 got a signal delivered to it
Thread 0x00000000 0000000b in signal handler
Thread 0x00000000 0000000b got a signal delivered to it
Thread 0x00000000 0000000d in signal handler
Thread 0x00000000 0000000d got a signal delivered to it
Masked thread 0x00000000 0000000a completed masked work
Masked thread 0x00000000 0000000e completed masked work
Masked thread 0x00000000 0000000c completed masked work
Main completed
```

pthread_signal_to_cancel_np()--Convert Signals to Cancel Requests

Syntax

```
#include <pthread.h>
#include <sched.h>
int pthread_signal_to_cancel_np(sigset_t *set, pthread_t *thread);
Threadsafe: Yes
Signal Safe: No
```

The **pthread_signal_to_cancel_np()** function causes a **pthread_cancel()** to be delivered to the target thread when the first signal specified in *set* arrives.

All threads in the process should have the signals specified by *set* blocked from the time of the call to **pthread_signal_to_cancel_np()** until the time when the **pthread_cancel()** is delivered to the target thread.

If **pthread_signal_to_cancel_np()** has been called, but a signal has not yet been converted to a **pthread_cancel()**, a subsequent call to **pthread_signal_to_cancel_np()** overrides the first call.

The **pthread_signal_to_cancel_np()** function creates a service thread (called the SignalToCancel thread) to perform the signal to cancel conversion. This conversion occurs asynchronously to the thread that called **pthread_signal_to_cancel_np()**.

The SignalToCancel thread blocks all signals and performs a **sigwait()** on the *set* of signals specified by *set*. When **sigwait()** returns, indicating that one of the signals in *set* was synchronously received, the SignalToCancel thread calls **pthread_cancel()** using the *thread* specified as the target.

Since the SignalToCancel thread processing occurs asynchronously, the caller of **pthread_signal_to_cancel_np()** is not notified of errors that may occur during the processing of the SignalToCancel thread. If the target thread has terminated or the signals specified by *set* are not valid, the caller of **pthread_signal_to_cancel_np()** is not notified..

This function is not portable.

Parameters

set

(Input) The set of signals that will be converted to **pthread_cancel()** requests

thread

(Input) The thread that will be canceled when a signal in *set* arrives

Authorities and Locks

None.

Return Value

0

pthread_signal_to_cancel_np() was successful.

value

pthread_signal_to_cancel_np() was not successful. *value* is set to indicate the error condition.

Error Conditions

If `pthread_signal_to_cancel_np()` was not successful, the error condition returned usually indicates one of the following errors. Under some conditions, the value returned could indicate an error other than those listed here.

[EINVAL]

The value specified for the argument is not correct.

Related Information

- The `<pthread.h>` header file. See [Header files for Pthread functions](#).
- [pthread_cancel\(\)--Cancel Thread](#)
- [pthread_kill\(\)--Send Signal to Thread](#)
- [pthread_sigmask\(\)--Set or Get Signal Mask](#)

Example

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include <sys/signal.h>
#include "check.h"

void sighand(int signo);

void cancellationCleanup(void *parm) { printf("Thread was canceled\n"); }

void *threadfunc(void *parm)
{
    pthread_t          self = pthread_self();
    pthread_id_np_t     tid;
    int                rc;
    int                i=5;

    pthread_getunique_np(&self, &tid);
    printf("Thread 0x%.8x %.8x entered\n", tid);
    while (i-->0) {
        printf("Thread 0x%.8x %.8x looping\n",
            tid, rc, errno);
        sleep(2);
        pthread_testcancel();
    }
    printf("Thread 0x%.8x %.8x did not expect to get here\n",
        tid);
    return NULL;
}

int main(int argc, char **argv)
{
    int                rc;
    int                i;
    pthread_t          thread;
    struct sigaction    actions;
    sigset_t           mask;
    void                *status;
    pthread_t          self;
```

```

pthread_id_np_t      tid;

printf("Enter Testcase - %s\n", argv[0]);

printf("Set up the alarm handler for the process\n");
memset(&actions, 0, sizeof(actions));
sigemptyset(&actions.sa_mask);
actions.sa_flags = 0;
actions.sa_handler = sighand;

rc = sigaction(SIGALRM,&actions,NULL);
checkResults("sigaction\n", rc);

printf("Block all signals in the parent so they can be inherited\n");
sigfillset(&mask); /* Mask all allowed signals */
rc = pthread_sigmask(SIG_BLOCK, &mask, NULL);
checkResults("pthread_sigmask()\n", rc);

printf("Create thread that inherits blocking mask\n");
/* Thread will inherit blocking mask */
rc = pthread_create(&thread, NULL, threadfunc, NULL);
checkResults("pthread_create()\n", rc);

/* Convert signals to cancels */
rc = pthread_signal_to_cancel_np(&mask, &thread);
checkResults("pthread_signal_to_cancel()\n", rc);

sleep(3);
self = pthread_self();
pthread_getunique_np(&self, &tid);

printf("Thread 0x%.8x %.8x sending a signal to the process\n", tid);
kill(getpid(), SIGALRM);
checkResults("kill()\n", rc);

printf("Wait for masked and unmasked threads to complete\n");
rc = pthread_join(thread, &status);
checkResults("pthread_join()\n", rc);

if (status != PTHREAD_CANCELED) {
    printf("Got an incorrect thread status\n");
    return 1;
}
printf("The target thread was canceled\n");
printf("Main completed\n");
return 0;
}

void sighand(int signo)
{
    pthread_t      self = pthread_self();
    pthread_id_np_t      tid;

    pthread_getunique_np(&self, &tid);
    printf("Thread 0x%.8x %.8x in signal handler\n",
           tid);
    return;
}

```

Output:

```
Enter Testcase - QP0WTEST/TPSIG2C0
Set up the alarm handler for the process
Block all signals in the parent so they can be inherited
Create thread that inherits blocking mask
Thread 0x00000000 00000007 entered
Thread 0x00000000 00000007 looping
Thread 0x00000000 00000007 looping
Thread 0x00000000 00000006 sending a signal to the process
Wait for masked and unmasked threads to complete
The target thread was canceled
Main completed
```

What are Pthreads?

Portable Operating System Interface for Computer Environments (POSIX) is an interface standard governed by the IEEE and based on UNIX. POSIX is an evolving family of standards that describe a wide spectrum of operating system components ranging from C language and shell interfaces to system administration.

The Pthread interfaces described in this section are based on a subset of the application programming interfaces (APIs) defined in the POSIX standard (ANSI/IEEE Standard 1003.1, 1996 Edition OR ISO/IEC 9945-1: 1996) and the Single UNIX Specification, Version 2, 1997. The implementation of these APIs is not compliant with these standards. However, the implementation does attempt to duplicate the portable nature of the interfaces defined by the standards. Differences between Pthreads in OS/400 and other thread types are described in [OS/400 Pthreads versus other threads implementations](#).

Primitive data types for Pthreads

The Pthread types and functions have the following naming conventions. If the type of object is not a thread, **object** represents the type of object, **action** is an operation to be performed on the object, **np** or **NP** indicates that the name or symbol is a non-portable extension to the API set, and **PURPOSE** indicates the use or purpose of the symbol.

types

`pthread[_object][_np]_t`

functions

`pthread[_object]_action[_np]`

Constants and Macros

`PTHREAD_PURPOSE[_NP]`

Type	Description
<code>pthread_attr_t</code>	Thread creation attribute
<code>pthread_cleanup_entry_np_t</code>	Cancellation cleanup handler entry
<code>pthread_condattr_t</code>	Condition variable creation attribute
<code>pthread_cond_t</code>	Condition Variable synchronization primitive
<code>pthread_joinoption_np_t</code>	Options structure for extensions to <code>pthread_join()</code>
<code>pthread_key_t</code>	Thread local storage key
<code>pthread_mutexattr_t</code>	Mutex creation attribute
<code>pthread_mutex_t</code>	Mutex (Mutual exclusion) synchronization primitive
<code>pthread_once_t</code>	Once time initialization control variable
<code>pthread_option_np_t</code>	Pthread run-time options structure
<code>pthread_rwlockattr_t</code>	Read/Write lock attribute
<code>pthread_rwlock_t</code>	Read/Write synchronization primitive
<code>pthread_t</code>	Pthread handle
<code>pthread_id_np_t</code>	Thread ID. For use as an integral type.
<code>struct sched_param</code>	Scheduling parameters (priority and policy)

After creating the primitive objects of type **pthread_cond_t** and **pthread_mutex_t** using the appropriate initialization functions, those objects must not be copied or moved to a new location. If the condition variable or mutex is copied or moved to a new location, the new primitive object is not valid or usable. Attempts to use the new object result in the **EINVAL** error.

Feature test macros for Pthreads

Constant	Description
_POSIX_THREADS	Base threads
_POSIX_THREAD_ATTR_STACKADDR	Stack address attribute. Not present in the OS/400 implementation.
_POSIX_THREAD_ATTR_STACKSIZE	Stack size attribute. Not present in the OS/400 implementation.
_POSIX_THREAD_PRIORITY_SCHEDULING	Thread priority scheduling. Not present in the OS/400 implementation.
_POSIX_THREAD_PRIO_INHERIT	Mutex priority inheritance. Not present in the OS/400 implementation.
_POSIX_THREAD_PRIO_PROTECT	Mutex priority ceiling. Not present in the OS/400 implementation.
_POSIX_THREAD_PROCESS_SHARED	Synchronization primitives may be shared between processes.

The OS/400 implementation of pthreads defines the **_POSIX_THREADS** and **_POSIX_THREAD_PROCESS_SHARED** feature test macros. See [Unsupported preprocessor and feature test macros](#) for a complete list of unsupported feature test macros.

OS/400 Pthreads versus the POSIX standard, the Single UNIX Specification, and other threads implementations

Although the Pthread interfaces described in this document are based on a subset of the APIs defined in the POSIX standard and the Single UNIX Specification, the implementation of these APIs is not compliant with these standards. This means that applications written in other versions of threads are not necessarily portable to OS/400. Below is a list of the differences between the Pthread APIs and other threads implementations.

- [All thread definitions in pthread.h](#)
- [Unsupported preprocessor and feature test macros](#)
- [Unsupported APIs](#)
- [Unsupported constants](#)
- [Unsupported cancellation points](#)
- [Unsupported sysconf\(\) configuration variables](#)
- [Thread priority and scheduling](#)
- [Thread ID vs. Pthread Handle \(pthread_t\)](#)
- [Thread ID value and size](#)
- [Mutexes return EDEADLK when re-locked by owner](#)
- [Return values from thread start routines are not integers](#)
- [Threads do not necessarily start before pthread_create\(\) returns](#)
- [Initial thread is special, cannot pthread_exit\(\)](#)
- [Pthread APIs cause asynchronous signals initialization](#)
- [Not all jobs can create threads; pthread_create\(\) fails with EBUSY](#)
- [Read/write locks are recursive](#)
- [Shared read/write locks are released at thread termination](#)
- [Read/write locks can be upgraded / downgraded](#)
- [Read/write locks do not favor writers](#)
- [Spawn API provides more POSIX-like process model](#)
- [C++ destructors and Pthread termination](#)
- [Unhandled exceptions terminate the thread \(not the process\)](#)
- [Exceptions vs. Asynchronous signals vs. ANSI C signals](#)
- [Mutexes can be named to aid in application debug](#)

All thread definitions in pthread.h

For Pthreads on AS/400, all feature test macros, preprocessor values, data structures, types, and function prototypes are located in the pthread.h header file instead of the system header files that are specified by POSIX or the Single Unix Specification.

Unsupported preprocessor and feature test macros

The following Pthread feature test macros are not defined on AS/400:

- **_POSIX_THREAD_ATTR_STACKADDR**
- **_POSIX_THREAD_ATTR_STACKSIZE**
- **_POSIX_THREAD_PRIO_INHERIT**
- **_POSIX_THREAD_PRIO_PROTECT**
- **_POSIX_THREAD_SAFE_FUNCTIONS**
- **_POSIX_THREAD_PRIORITY_SCHEDULING**

Unsupported APIs

The following functions are not supported by the iSeries implementation of pthreads. These functions are all defined and provided by the system. You can create and compile with these functions in your application. If the unsupported functions are called, when the application runs the functions immediately fail with the ENOSYS error, and your application can take the appropriate action, such as ignoring the error and continuing.

- [pthread_atfork\(\)--Register Fork Handlers](#)
- [pthread_atfork_np\(\)--Register Fork Handlers with Extended Options](#)
- [pthread_attr_getschedpolicy\(\)--Get Scheduling Policy](#)
- [pthread_attr_getscope\(\)--Get Scheduling Scope](#)
- [pthread_attr_getstackaddr\(\)--Get Stack Address](#)
- [pthread_attr_getstacksize\(\)--Get Stack Size](#)
- [pthread_attr_setschedpolicy\(\)--Set Scheduling Policy](#)
- [pthread_attr_setscope\(\)--Set Scheduling Scope](#)
- [pthread_attr_setstackaddr\(\)--Set Stack Address](#)
- [pthread_attr_setstacksize\(\)--Set Stack Size](#)
- [pthread_mutexattr_getprioceiling\(\)--Get Mutex Priority Ceiling Attribute](#)
- [pthread_mutexattr_getprotocol\(\)--Get Mutex Protocol Attribute](#)
- [pthread_mutexattr_setprioceiling\(\)--Set Mutex Priority Ceiling Attribute](#)
- [pthread_mutexattr_setprotocol\(\)--Set Mutex Protocol Attribute](#)
- [pthread_mutex_getprioceiling\(\)--Get Mutex Priority Ceiling](#)
- [pthread_mutex_setprioceiling\(\)--Set Mutex Priority Ceiling](#)

pthread_atfork()--Register Fork Handlers

Syntax

```
#include <pthread.h>
#include <sched.h>
int pthread_atfork(void (*prepare)(void),
                  void (*parent)(void),
                  void (*child)(void));
```

The pthread_atfork() function is not supported by this implementation. The function returns ENOSYS.

pthread_atfork_np()--Register Fork Handlers with Extended Options

Syntax

```
#include <pthread.h>
#include <sched.h>
int pthread_atfork(int *userstate,
                  void (*prepare)(void),
                  void (*parent)(void),
                  void (*child)(void));
```

The pthread_atfork_np() function is not supported by this implementation. The function returns ENOSYS.

pthread_attr_getschedpolicy()--Get Scheduling Policy

Syntax

```
#include <pthread.h>
#include <sched.h>
int pthread_attr_getschedpolicy(pthread_attr_t *attr,
                                int *policy);
```

The pthread_attr_getschedpolicy() function is not supported by this implementation. The function returns ENOSYS.

pthread_attr_getscope()--Get Scheduling Scope

Syntax

```
#include <pthread.h>
#include <sched.h>
int pthread_attr_getscope(pthread_attr_t *attr,
                           int *contentionscope);
```

The pthread_attr_getscope() function is not supported by this implementation. The function returns ENOSYS.

pthread_attr_getstackaddr()--Get Stack Address

Syntax

```
#include <pthread.h>
#include <sched.h>
int pthread_attr_getstackaddr(const pthread_attr_t *attr,
                              void **stackaddr);
```

The pthread_attr_getstackaddr() function is not supported by this implementation. The function returns ENOSYS.

pthread_attr_getstacksize()--Get Stack Size

Syntax

```
#include <pthread.h>
#include <sched.h>
int pthread_attr_getstacksize(const pthread_attr_t *attr,
                             size_t *stacksize);
```

The pthread_attr_getstacksize() function is not supported by this implementation. The function returns ENOSYS.

pthread_attr_setschedpolicy()--Set Scheduling Policy

Syntax

```
#include <pthread.h>
#include <sched.h>
int pthread_attr_setschedpolicy(pthread_attr_t *attr,
                                int policy);
```

The pthread_attr_setschedpolicy() function is not supported by this implementation. The function returns ENOSYS.

pthread_attr_setscope()--Set Scheduling Scope

Syntax

```
#include <pthread.h>
#include <sched.h>
int pthread_attr_setscope(pthread_attr_t *attr,
                          int contentionscope);
```

The pthread_attr_setscope() function is not supported by this implementation. The function returns ENOSYS.

pthread_attr_setstackaddr()--Set Stack Address

Syntax

```
#include <pthread.h>
#include <sched.h>
int pthread_attr_setstackaddr(pthread_attr_t *attr,
                              void *stackaddr);
```

The pthread_attr_setstackaddr() function is not supported by this implementation. The function returns ENOSYS.

pthread_attr_setstacksize()--Set Stack Size

Syntax

```
#include <pthread.h>
#include <sched.h>
int pthread_attr_setstacksize(pthread_attr_t *attr,
                             size_t stacksize);
```

The pthread_attr_setstacksize() function is not supported by this implementation. The function returns ENOSYS.

pthread_mutexattr_getprioceiling()--Get Mutex Priority Ceiling Attribute

Syntax

```
#include <pthread.h>
#include <sched.h>
int pthread_mutexattr_getprioceiling(const pthread_mutexattr_t *attr,
                                     int *prioceiling);
```

The pthread_mutexattr_getprioceiling() function is not supported by this implementation. The function returns ENOSYS.

pthread_mutexattr_getprotocol()--Get Mutex Protocol Attribute

Syntax

```
#include <pthread.h>
#include <sched.h>
int pthread_mutexattr_getprotocol(const pthread_mutexattr_t *attr,
                                  int *protocol);
```

The pthread_mutexattr_getprotocol() function is not supported by this implementation. The function returns ENOSYS.

pthread_mutexattr_setprioceiling()--Set Mutex Priority Ceiling Attribute

Syntax

```
#include <pthread.h>
#include <sched.h>
int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr,
                                     int prioceiling);
```

The pthread_mutexattr_setprioceiling() function is not supported by this implementation. The function returns

ENOSYS.

pthread_mutexattr_setprotocol()--Set Mutex Protocol Attribute

Syntax

```
#include <pthread.h>
#include <sched.h>
int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr,
                                  int protocol);
```

The pthread_mutexattr_setprotocol() function is not supported by this implementation. The function returns ENOSYS.

pthread_mutex_getprioceiling()--Get Mutex Priority Ceiling

Syntax

```
#include <pthread.h>
#include <sched.h>
int pthread_mutex_getprioceiling(const pthread_mutex_t *mutex,
                                  int *prioceiling);
```

The pthread_mutex_getprioceiling() function is not supported by this implementation. The function returns ENOSYS.

pthread_mutex_setprioceiling()--Set Mutex Priority Ceiling

Syntax

```
#include <pthread.h>
#include <sched.h>
int pthread_mutex_setprioceiling(pthread_mutex_t *mutex,
                                  int prioceiling, int *oldceiling);
```

The pthread_mutex_setprioceiling() function is not supported by this implementation. The function returns ENOSYS.

Unsupported constants

The following constants related to threads are not defined on AS/400.

- **PTHREAD_STACK_MIN**
- **PTHREAD_PRIO_INHERIT**
- **PTHREAD_PRIO_NONE**
- **PTHREAD_PRIO_PROTECT**

Unsupported cancellation points

OS/400 does not support the full set of cancellation points. Although the APIs may be provided, they are not necessarily cancellation points. The only cancellation points currently supported are those APIs that are part of the Pthread run-time. Those APIs are the following:

- `pthread_cond_timedwait()`
- `pthread_cond_wait()`
- `pthread_delay_np()`
- `pthread_join()`
- `pthread_join_np()`
- `pthread_testcancel()`

An appropriate alternative to create cancellation points for these APIs might be like the following example. You can use this example to create a cancellation point out of any function that is asynchronous signal safe. See the Signal Concepts section of the Unix Type APIs section in the System API reference book for a list of functions that are asynchronous signal safe. If a function is not asynchronous signal safe, you should not use this form of asynchronous cancellation because it corrupt data.

Example

```
... preceding code ...
int oldtype=0;
/* If cancellation is currently disabled, this will have no effect */
/* if cancellation is currently enabled, we'll set it to asynchronous */
/* for the duration of this call to try to simulate a cancellation point */
pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, &oldtype);
/* Call kernel API that you want to be a cancel point. You should */
/* only call functions which are asynchronous signal safe in this block. */
/* Validating the asynchronous signal safety of the function will */
/* ensure that the asynchronous cancellation does not negatively */
/* affect the API or corrupt the data that the API uses */
APICallHere();
/* Restore the cancellation type that was previously in effect */
pthread_setcanceltype(oldtype, &oldtype);
... following code ...
```


Unsupported sysconf() configuration variables

The following sysconf() configuration variables related to threads are not supported on AS/400.

- **_SC_THREAD_DESTRUCTOR_ITERATIONS**
- **_SC_THREAD_PRIORITY_SCHEDULING**
- **_SC_THREADS**
- **_SC_THREAD_ATTR_STACKADDR**
- **_SC_THREAD_ATTR_STACKSIZE**
- **_SC_THREAD_KEYS_MAX**
- **_SC_THREAD_PRIO_INHERIT**
- **_SC_THREAD_PRIO_PROTECT**
- **_SC_THREAD_PROCESS_SHARED**
- **_SC_THREAD_SAFE_FUNCTIONS**
- **_SC_THREAD_STACK_MIN**
- **_SC_THREAD_THREADS_MAX**

Thread priority and scheduling

The default thread creation attributes of the AS/400 implementation of Pthreads uses an explicitly specified priority of **DEFAULT_PRIO_NP**. Some implementations inherit the scheduling priority and policy of the creating thread by default. For better performance, the AS/400 implementation chooses to start each thread with an explicit priority so that, when a thread is created, the priority of the creating thread does not need to be retrieved at run-time.

An AS/400 thread competes for scheduling resources against other threads in the system, not solely against other threads in the process. The scheduler is a delay cost scheduler based on several delay cost curves (priority ranges). The POSIX standard and the Single UNIX Specification refers to this as scheduling scope and scheduling policy, which cannot be changed from the default of **SCHED_OTHER** in this implementation.

The following Pthread APIs support a scheduling policy of only **SCHED_OTHER**.

- **pthread_setschedparam** (**SCHED_OTHER** only supported)
- **pthread_getschedparam**
- **pthread_attr_setschedparam**
- **pthread_attr_getschedparam**

The priority of a thread is specified as a number that represents the value that is added to the priority of the process. Changing the priority of the process affects the priority of all of the threads within that process. The default priority for a thread is **DEFAULT_PRIO_NP**, which is no change from the process priority.

On AS/400, numerically lower priority values indicate higher priority with regard to scheduling. The **pthread.h** and **sched.h** header files define the priority constants in a way that is consistent with the threads standard, but opposite of priority specifications on AS/400. When you specify a priority of -99 in a call to **pthread_setschedparam()**, the priority of the target thread is lowered to the lowest possible value.

For example, process P1 is at AS/400 priority 20 and contains a thread T1 that specifies a Pthread priority adjustment of -18. Process P2 is at AS/400 priority 25 and contains thread T2 that specifies a priority of -5. The result is that the system schedules the threads using the AS/400 priority for T1 as 38 and for T2 as 30. The thread scheduling is specified at a system level, and although process P2 runs at a lower priority ranking than process P1, thread T2 within process P2 runs at a higher priority ranking than thread T1 in process P1, and thus gets more processing resources.

Thread ID vs. Pthread Handle (pthread_t)

In many threads implementations, the pthread_t abstract type is implemented as an integer (4 byte) thread ID. In the AS/400 implementation of Pthreads, the thread ID is a 64-bit integral value and the pthread_t is an abstraction (structure) that contains that value and others. This abstraction helps to allow the implementation to scale to thousands of threads in a process.

Do not allow your program to rely on the internal structure or size of the pthread_t in a non-portable fashion, such as comparisons of thread IDs. For portable comparison, use the pthread_equal() API. This documentation occasionally refers to the pthread_t as a Pthread handle to try to prevent the misconception that it represents a single integer value.

Thread ID value and size

In some threads implementations, the thread ID is a 4-byte integer that starts at 1 and increases by 1 every time a thread is created. This integer can be used in a non-portable fashion by an application.

To assist in the portability problem with the application and to allow retrieval of the thread ID, the AS/400 implementation has provided the **pthread_getunique_np()** function to retrieve the thread ID from the Pthread handle. This thread ID is a 64-bit integer value. Because some compilers do not yet support a full 64-bit integer data type, the value is returned in a structure containing two 4-byte integers.

Mutexes return EDEADLK when re-locked by owner

Some threads implementations return the EDEADLK error when a mutex attempts to relock a mutex that it already owns. The POSIX standard specifies that the results are undefined when a mutex is re-locked by the owner. The Single UNIX Specification addresses these issues by providing a new mutex attribute called *type*.

The AS/400 threads support takes the same implementation route that the Single Unix Specification suggests, and it also causes the thread to deadlock when it attempts to re-lock a normal (non-recursive) mutex. Because many users of Pthreads do not check return codes from functions, the deadlock protects applications from corrupted data that might result if they attempt to relock an already held mutex, then unlock the mutex as if the lock was successful.

See [pthread_mutexattr_gettype\(\)--Get Mutex Type Attribute](#) and [pthread_mutexattr_settype\(\)--Set Mutex Type Attribute](#) if you need error-checking mutexes for your application.

Return values from thread start routines are not integers

Return values from a thread are defined to be of type **void ***. On some platforms, a void * and an integer can be easily interchanged with no loss of information. Until Version 4 Release 2, this was not true on the AS/400. The AS/400 enforces more strict pointer rules to both prevent and detect application bugs or a malicious program's behavior. Thus, when converting integers to pointers by a mechanism not directly supported by your compiler, the valid pointer information is lost, and the pointer is always set to **NULL** (regardless of its binary value).

New support put into the system in Version 4 Release 2 allows you to store an integer into a pointer, and still have the pointer be non-**NULL**. You cannot store to, read from, or defer a pointer created by this mechanism, but the pointer appears non-**NULL**.

The macros **__INT()** and **__VOID()** are provided to aid in compatibility and allow you to easily store and retrieve integer information in pointer variables even if your compiler does not support the direct typecast. These macros allow explicit conversion from a pointer to an integer and from an integer to a pointer.

*The macros **__INT()** and **__VOID()** result in function calls.*

Example

The following example shows the correct way to store and retrieve integer information in pointer variables.

```
#define __MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include "check.h"

int main(int argc, char **argv)
{
    void *status1 = __VOID(5);
    void *status2 = __VOID(999);

    if (status1 == NULL) {
        printf("Status1 pointer is NULL\n");
    }
    else {
        printf("Status1 pointer is non-NULL\n");
    }

    if (status1 == status2) {
        printf("Both status variables as pointers are equal\n");
    }
    else {
        if (status1 < status2) {
            printf("Status1 is greater than status2\n");
        }
        else {
            if (status1 < status2) {
                printf("Status1 is less than status2\n");
            }
            else {
                printf("The pointers are unordered!\n");
            }
        }
    }

    printf("Pointer values stored in status variables are:\n"
           "  status1 = %.8x %.8x %.8x %.8x\n");
}
```

Return values from thread start routines are not integers

```
        " status2 = %.8x %.8x %.8x %.8x\n",
        status1, status2);
printf("Integer values stored in status variables are:\n"
        " status1 = %d\n"
        " status2 = %d\n",
        __INT(status1), __INT(status2));
return;
}
```

Output:

```
Status1 pointer is non-NULL
Status1 is less then status2
Pointer values stored in status variables are:
 status1 = 80000000 00000000 00008302 00000005
 status2 = 80000000 00000000 00008302 000003e7
Integer values stored in status variables are:
 status1 = 5
 status2 = 999
```

Threads do not necessarily start before pthread_create() returns

A thread may or may not start running before the return from **pthread_create()**. Depending on the amount of time left in the creating threads, time slice, and the other activity on the system, the creating thread may return before the new thread runs.

The thread implementations of some systems guarantee a certain ordered behavior for thread creation versus the execution of the first statement in the new thread. On AS/400, it is unknown which happens first, the execution of the first instruction in the new thread or the return from **pthread_create()**.

The following example shows an incorrectly written application.

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include "check.h"

pthread_t      thread

void *threadfunc(void *parm)
{
    pthread_id_np_t tid;
    #error "This is an ERROR."
    #error "The 'thread' variable is shared between threads"
    #error "and must be protected by a mutex."
    pthread_getunique_np(&thread, &tid);
    printf("Thread 0x%.8x %.8x started\n", tid);
    return NULL;
}

int main(int argc, char **argv)
{
    int                rc=0;

    printf("Enter Testcase - %s\n", argv[0]);

    #error "This is an ERROR."
    #error "The order of thread thread startup, and return from"
    #error "the pthread_create() API is NOT deterministic."
    rc = pthread_create(&thread, NULL, threadfunc, NULL);
    checkResults("pthread_create(NULL)\n", rc);

    /* sleep() isn't a very robust way to wait for the thread */
    sleep(5);

    printf("Main completed\n");
    return 0;
}
```


Initial thread is special, cannot pthread_exit()

The initial thread in an OS/400 process is special because of these characteristics:

- If the initial thread calls **pthread_exit()**, the process terminates.
- If the initial thread is the target of a **pthread_cancel()** request that is acted upon, the process terminates.
- If the initial thread terminates through any other action, the process terminates.
- Many OS/400 APIs and commands target jobs. Some of those APIs target resources that are allocated to threads for retrieval or modification. If this is the case, the resources that displayed, modified, or retrieved may be the resources owned by the initial thread.
For example, the CL command **WRKACTJOB** allows you to display information such as the call stack for a job. Since a job does not have a call stack and the call stack is thread scoped, the call stack of the initial thread is displayed when you choose to display the call stack of a job.
Other APIs or CL commands that operate against jobs have undergone similar changes. See the specific documentation for the API or CL command of concern.

Pthread APIs cause asynchronous signals initialization

When a job is running in OS/400, by default it is not enabled for POSIX signals. The system never delivers a Posix signal to a job that is not enabled for signals.

The job is initialized for signals with the default POSIX signals environment when any thread in the job calls any API defined to implicitly enable signals. The main categories of APIs that enable signals are the signals APIs themselves and some process-related APIs related to signals. For example, some of the APIs that enable signals are **Qp0sEnableSignals()**, **kill()**, **sigaction()**, **sigprocmask()**, **getpid()**, and **spawn()**. After the initialization for signals occurs within a job, the system can deliver signals to that job if they are generated by another job or by the system.

When a program in a job uses Pthreads, that job is automatically enabled for signals when the Pthreads service program is loaded (either dynamically or statically). Loading the service program that contains the Pthread APIs causes the job to be initialized for signals, regardless of whether the application actually calls the pthread APIs. All pthread programs can implicitly receive signals if another job or the system generates a signal for the threaded job.

If the application calls **Qp0sDisableSignals()** to disable signals for the job, the Pthreads APIs do not function correctly. Do not use **Qp0sDisableSignals()** in a threaded job.

For more information about signals and the APIs mentioned in this section, see the *Signal APIs* and *Process-Related APIs* chapter of the *Unix Type APIs* section of the System API Reference.

Not all jobs can create threads; pthread_create() fails with EBUSY

Because many parts of the operating system are not yet thread safe, not every AS/400 job is allowed to start threads. The **pthread_create()** API fails with the **EBUSY** error when the process is not allowed to create threads. See [Running threaded programs](#) for information about how to start a job that can create threads.

For details about how to determine whether thread creation is currently allowed for your process, you can see the **pthread_getpthreadoption_np()** API, option **PTHREAD_OPTION_THREAD_CAPABLE_NP**.

See [Multithreaded Applications](#) for an introduction to threads and general API information about AS/400 threads.

Read/write locks are recursive

The OS/400 implementation of read/write locks provides a recursive behavior not only for shared read locks (as the thread standard specifies), but for exclusive write locks as well. The following statements apply to read/write locks on OS/400:

- A thread can acquire any number of shared read locks on a read/write lock. Each successful shared read lock that is acquired must be released by a call to **pthread_rwlock_unlock()**.
- A thread can acquire any number of exclusive write locks on a read/write lock. Each successful exclusive write lock that is acquired must be released by a call to **pthread_rwlock_unlock()**.

Shared read/write locks are released at thread termination

If a thread is the owner of one or more shared read locks acquired by **pthread_rwlock_unlock()**, **pthread_rwlock_tryrdlock()**, or **pthread_rwlock_timedrdlock_np()**, when that thread terminates, the shared read locks are automatically released by the system. If a thread holds a shared read lock, it does not modify the resources associated with that lock. It is then safe for the runtime support to unlock the read lock without indicating an error condition or causing the process to wait. For performance reasons, your application should unlock all held locks before the thread ends.

If a thread is the owner of one or more exclusive write locks acquired by **pthread_rwlock_wrlock()**, **pthread_rwlock_trywrlock()**, or **pthread_rwlock_timedwrlock_np()**, when that thread terminates, the exclusive write locks are not automatically released by the system. This is an error in the application and indicates that the data associated with the lock is in an inconsistent state. If another thread attempts to get a shared read or exclusive write lock on the lock, that thread blocks forever.

Read/write locks can be upgraded / downgraded

The OS/400 implementation of read/write locks allows a thread to effectively change a read lock to a write lock, or change a write lock to a read lock, without an intervening unlocked and unprotected section of code. The following items describe read/write lock behavior that allows these changes. This behavior is outside of the definition of the Single UNIX Specification. An application written to be portable to the Single UNIX Specification should not attempt to acquire a shared read lock and a shared write lock on the same read/write lock at the same time.

- If a thread currently holds a shared read lock, an attempt by the same thread to acquire an exclusive write lock succeeds if no other threads hold a shared read lock. The thread then holds both an exclusive write lock and a shared read lock.
- If a thread currently holds an exclusive write lock, an attempt by the thread to acquire a shared read lock succeeds. The thread then holds both an exclusive write lock and a shared read lock.
- If a thread holds one or more shared read locks and one or more exclusive write locks on the same read/write lock object at the same time, a call to **pthread_rwlock_unlock()** always unlocks the exclusive write lock FIRST.
- When multiple exclusive write locks and multiple exclusive read locks are held by the same thread on the same read/write lock object, the behavior of **pthread_rwlock_unlock()** is as follows:
 - A call to the **pthread_rwlock_unlock()** function always unlocks the most recent exclusive write lock first.
 - Subsequent calls to **pthread_rwlock_unlock()** first reduce the count of any outstanding exclusive write locks held by the thread until all exclusive write locks are unlocked.
 - After all outstanding exclusive write locks are unlocked and the thread holds only shared read locks on the read/write lock object, a call to **pthread_rwlock_unlock()** function then unlocks the most recent shared read lock.
 - Subsequent calls to **pthread_rwlock_unlock()** reduce the count of any outstanding shared read locks held by the thread until all shared read locks are unlocked.

For a thread to change a shared read lock to an exclusive write lock, the thread should perform the following actions:

```
{
pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER;
pthread_rwlock_rdlock(&rwlock);
...
/* Thread holding a read lock decides it needs to upgrade to a write lock */
/* Now Upgrade to write lock */
pthread_rwlock_wrlock(&rwlock);
```

```

...
/* write lock (and read lock) are held here.*/
/* We have effectively upgraded to a write lock */
...
/* `Downgrade' back to a only the read lock */
pthread_rwlock_unlock(&rwlock);
...
/* unlock the read lock */
pthread_rwlock_unlock(&rwlock);
}

```

For a thread to change an exclusive write lock to a shared read lock, the thread should perform the following actions:

```

{
pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER;
pthread_rwlock_wrlock(&rwlock);
...
/* Thread holding the write lock decides it needs to downgrade to a read
lock */
/* Get the read lock, so we are holding BOTH read and write locks */
pthread_rwlock_rdlock(&rwlock);
...
/* An unlock always unlocks the write lock first */
pthread_wrlock_unlock(&rwlock);
...
/* At this point, we are only holding the read lock. */
/* We have effectively downgraded the write lock to a read lock */
...
/* Use unlock to unlock the last read lock. */
pthread_wrlock_unlock(&rwlock);
}

```

Read/write locks do not favor writers

The OS/400 implementation of read/write locks does not favor writers. If your application has a large number of readers contending for the same lock, the writers may not be allowed to write.

The OS/400 implementation of **pthread_rwlock_tryrdlock()**, for example, does not completely honor the Single UNIX Specification in its treatment of reader/writer contention.

The standard states the following: "The function `pthread_rwlock_tryrdlock()` applies a read lock as in the `pthread_rwlock_rdlock()` function with the exception that the function fails if any thread holds a write lock on `rwlock` or there are writers blocked on `rwlock`. "

In the OS/400 implementation, if **pthread_rwlock_tryrdlock()** is used on a read/write lock that has multiple readers holding the lock and multiple waiting writers blocked on the lock, the **pthread_rwlock_tryrdlock()** are allowed to complete successfully.

Spawn API provides more POSIX-like process model

AS/400 uses a call/return mechanism when your application calls programs. A new process is not started when you call a program, instead the program runs and returns to its caller. You can use activation groups to separate or partition the program resources from the caller.

For the more POSIX-like behavior of running each program in a separate process (and thus taking advantage of thread safety, encapsulation, and protection that the new process may give you), use the **spawn()** API to start the program.

You can also use the capability provided in **spawn()** to allow the child process to start multiple threads. See the **spawn()** API documentation for a description of the **SPAWN_SETTHREAD_NP** flag in the inheritance structure.

A CL command for using **SPAWN** is also available from the **QUSRTOOL** library. See [SPAWN CL command](#), [QUSRTOOL example](#) for more information about the **SPAWN** CL command.

C++ destructors and Pthread termination

Unlike some other implementations of threads, C++ destructors for automatic objects are allowed to run in a well defined and consistent manner when a thread is terminated.

The following list includes some of the causes of thread termination:

- A thread calls **pthread_exit()** or returns from the thread start routine.
- A thread is the target of **pthread_cancel()**.
- A thread is ended due to an unhandled exception.
- A thread is in a process that contains another thread that calls **exit()** or **abort()**.
- A thread is in a process that is terminated by the system administrator.
- A thread is being terminated by the system administrator.

When a thread terminates, the following occurs:

1. If the thread was ended using **pthread_exit()**, **pthread_cancel()** or return from the thread start routine, then cancellation cleanup handlers and data destructors are run.
2. The thread is terminated. At the time that the thread is terminated, both C++ destructors for automatic objects and AS/400 cancel handlers run.

If a Pthread is terminated using a non-Pthread method (an AS/400 exception, a different thread termination primitive provided by the system, **exit()** or **abort()**, or other job termination method), Pthread cancellation cleanup handlers and data destructors do not run.

Example

This example shows the relationship between C++ destructors and Pthread cleanup mechanisms.

```
#define _MULTI_THREADED
#include <stdio.h>
#include <qp0z1170.h>
#include <time.h>
#include <pthread.h>
#include "check.h"

#define bufferSize 100
#define threadRc 55

pthread_key_t  tlskey;

void dataDestructor(void *parm);
void cancelHandler(void *parm);
void *threadfunc(void *parm);
void level2(void);
void level3(void);

class A {
public:
    A(char *label);
    ~A();
private:
    pthread_id_np_t    tid;
    char               buffer[bufferSize];
};
```

```

void dataDestructor(void *parm) {
    printf("In data destructor\n");
    pthread_setspecific(tlskey, NULL);
}

void cancelHandler(void *parm) {
    printf("In cancellation cleanup handler\n");
}

void *threadfunc(void *parm) {
    A          object("start routine object");
    level2();
    return NULL;
}

void level2(void) {
    A          object("Second level object");
    level3();
}

void level3(void) {
    int          rc;
    struct timespec  ts = {5, 0};
    A          object("Third level object");

    pthread_setspecific(tlskey, &tlskey);
    pthread_cleanup_push(cancelHandler, NULL);
    printf("Thread blocked\n");
    rc = pthread_delay_np(&ts);
    if (rc != 0) {
        printf("pthread_delay_np() - return code %d\n", rc);
        return;
    }
    printf("Calling pthread_exit()\n");
    pthread_exit(__VOID(threadRc));
    pthread_cleanup_pop(0);
}

int main(int argc, char **argv)
{
    int          rc=0;
    int          i;
    pthread_t    threadid;
    void          *status;
    int          fail=0;

    printf("Enter Testcase - %s\n", argv[0]);
    rc = pthread_key_create(&tlskey, dataDestructor);
    checkResults("pthread_key_create()\n", rc);

    printf("----- Start pthread_cancel() example ----- \n");
    printf("Create a thread\n");
    rc = pthread_create(&threadid, NULL, threadfunc, NULL);
    checkResults("pthread_create()\n", rc);

    sleep(2);
    rc = pthread_cancel(threadid);
    checkResults("pthread_cancel()\n", rc);

    rc = pthread_join(threadid, &status);

```

```

    checkResults("pthread_join()\n", rc);
    if (status != PTHREAD_CANCELED) {
        printf("Canceled thread did not return the expected results\n");
        fail = 1;
    }

    printf("----- Start pthread_exit() example ----- \n");
    printf("Create a thread\n");
    rc = pthread_create(&threadid, NULL, threadfunc, NULL);
    checkResults("pthread_create()\n", rc);

    rc = pthread_join(threadid, &status);
    checkResults("pthread_join()\n", rc);
    if (__INT(status) != threadRc) {
        printf("pthread_exit() thread did not return the expected results\n");
        fail = 1;
    }

    pthread_key_delete(tlskey);
    if (fail) {
        printf("At least one thread failed!\n");
        exit(1);
    }
    printf("Main completed\n");
    return 0;
}

A::A(char *label) {
    strncpy(buffer, label, bufferSize);
    pthread_t      me;
    me = pthread_self();
    pthread_getunique_np(&me, &tid);
    printf("`%s' instantiated in thread 0x%.8x %.8x\n",
        buffer, tid);
}

A::~~A() {
    printf("`%s' destroyed in thread 0x%.8x %.8x\n",
        buffer, tid);
}

```

Output:

```

Enter Testcase - QP0WTEST/TPCPP0
----- Start pthread_cancel() example -----
Create a thread
`start routine object' instantiated in thread 0x00000000 00000161
`Second level object' instantiated in thread 0x00000000 00000161
`Third level object' instantiated in thread 0x00000000 00000161
Thread blocked
In cancellation cleanup handler
In data destructor
`Third level object' destroyed in thread 0x00000000 00000161
`Second level object' destroyed in thread 0x00000000 00000161
`start routine object' destroyed in thread 0x00000000 00000161
----- Start pthread_exit() example -----
Create a thread
`start routine object' instantiated in thread 0x00000000 00000162
`Second level object' instantiated in thread 0x00000000 00000162
`Third level object' instantiated in thread 0x00000000 00000162
Thread blocked

```

C++ destructors and Pthread termination

```
Calling pthread_exit()  
In cancellation cleanup handler  
In data destructor  
`Third level object' destroyed in thread 0x00000000 00000162  
`Second level object' destroyed in thread 0x00000000 00000162  
`start routine object' destroyed in thread 0x00000000 00000162  
Main completed
```

Unhandled exceptions terminate the thread (not the process)

On a Unix system, when an invalid or illegal software condition is encountered (such as dividing by zero or using an invalid pointer), a signal is generated. If the signal is not handled, the process is terminated.

OS/400 does not generate a signal for these events, but instead, generates an AS/400 exception message. The exception message moves up the call stack, allowing each stack frame (function on the stack or invocation entry) a chance to handle the exception. Each function invocation may choose to handle or not to handle the exception. If the exception is not handled, the message continues to the next stack frame.

When the exception message reaches certain boundaries on the call stack (like a main() entry point, usually called control boundaries) certain events take place. These events include changing the exception to a different type, terminating the process, terminating the activation group, or terminating the thread. If an unhandled exception condition happens in a secondary thread and moves all the way to the first invocation in the thread without being handled, the resulting action will be to terminate the thread. During this percolation, if the exception hits a control boundary and is not handled, it may terminate the process.

A signal is never automatically generated for an exception message. When an unhandled exception terminates the thread, Pthread cancellation cleanup handlers and Pthread data destructors do not run and the thread is terminated immediately with a return status of **PTHREAD_EXCEPTION_NP**. **PTHREAD_EXCEPTION_NP** is a macro similar to the **PTHREAD_CANCELED** macro, and is not NULL or a valid pointer.

On a Unix system, this same activity may terminate the process due to the signal that is generated.

In order to have your application terminate the process, when the exception occurs, you must handle it and explicitly terminate the process. The following example handles all hardware exceptions using the ANSI C signal model and uses the Pthread signal SIGABRT to terminate the process.

You can also turn the exception message into a Posix signal and it may be handled. See [Exceptions vs. Asynchronous signals vs. ANSI C signals](#) for more information.

Example

```
#define _MULTI_THREADED
#include <stdio.h>
#include <qp0z1170.h>
#include <time.h>
#include <signal.h>
#include <pthread.h>
#include "check.h"

void abortTheProcessWhenAnExceptionOccurs(int sigNumber);
void *threadfuncl(void *parm);

void *threadfuncl(void *parm)
{
    char *p=NULL;
    printf("Thread1: Unhandled exception (pointer fault) about to happen\n");
    *p = '!';
    printf("Thread1: After exception\n");
    return NULL;
}
```

Unhandled exceptions terminate the thread (not the process)

```
void abortTheProcessWhenAnExceptionOccurs(int sigNumber) {
    /* In a multithreaded environment this is a little difficult. We have to
    */
    /* re-enable the ANSI C handler immediately, because that is the way it
    */
    /* is defined. (A better alternative may be direct monitor exception
    */
    /* handlers that are always valid in the function which they are
    */
    /* registered, and with direct monitors, we can catch the hardware
    */
    /* exception before it is converted to an ANSI C
    signal
    */
    signal(SIGALL, abortTheProcessWhenAnExceptionOccurs);
    /* Since ANSI C signals and hardware exceptions are only handled in
    */
    /* the same thread that caused them, we send the Posix signal to
    */
    /* the calling thread (The signal is delivered before returning from
    */
    /* pthread_kill().
    */
    printf("Mapping ANSI signal %d to posix signal SIGABRT. "
           "Aborting the process\n", sigNumber);

    /* If we want to do some debug processing, we can put it here.
    */
    pthread_kill(pthread_self(), SIGABRT);
    return;
}

int main(int argc, char **argv)
{
    int                rc=0;
    pthread_t          threadid;
    void                *status;

    printf("----- Setup Signal Mapping/Handling -----\\n");
    printf("- Register ANSI C signal handler to map ALL\\n"
           "  ANSI C signals & hardware exceptions to Posix signals\\n");
    /* If we want to do debug, or determine what when wrong a little more
    easily,
    */
    /* we could use the abortTheProcessWhenAnExceptionOccurs function to
    delay
    the thread, or
    */
    /* dump failure data of some
    sort.
    */
    signal(SIGALL, abortTheProcessWhenAnExceptionOccurs);

    printf("----- Start memory fault thread -----\\n");
    printf("Create a thread\\n");
    rc = pthread_create(&threadid, NULL, threadfunc1, NULL);
    checkResults("pthread_create()\\n", rc);

    rc = pthread_join(threadid, &status);
    checkResults("pthread_join()\\n", rc);

    printf("Main completed\\n");
    return 0;
}
```

Unhandled exceptions terminate the thread (not the process)

```
}
```

Output:

```
----- Setup Signal Mapping/Handling -----  
- Register ANSI C signal handler to map ALL  
  ANSI C signals & hardware exceptions to Posix signals  
----- Start memory fault thread -----  
Create a thread  
Thread1: Unhandled exception (pointer fault) about to happen  
Mapping ANSI signal 5 to posix signal SIGABRT. Aborting the process
```

Exceptions vs. Asynchronous signals vs. ANSI C signals

AS/400 distinguishes between hardware exceptions, POSIX signals (sometimes called asynchronous signals on AS/400), and ANSI C signals. POSIX signals use the APIs `kill()`, `sigaction()`, `pthread_kill()`, `alarm()`, `pause()`, and others for signal interaction. ANSI C signals use the APIs `raise()`, `signal()`, and `abort()` for signal interaction.

Many other systems, by default, generate a POSIX signal whenever a software or hardware exception occurs (such as using a pointer that is not valid, or an error caused by dividing by zero), and on those systems, a POSIX signal may be equivalent and indistinguishable from an ANSI C signal. If the signal is not handled, this results in the termination of the process.

OS/400 does not generate a signal for these hardware or software problems, but instead, generates an AS/400 exception message. The exception message moves up the call stack, allowing each stack frame (function on the stack or invocation entry) a chance to handle the exception. Each function invocation may choose whether or not to handle the exception. If the exception is not handled, the message continues to the next stack frame.

When the exception message reaches certain boundaries on the call stack (such as a `main()` entry point, usually called control boundaries), certain events take place. These events include changing the exception to a different type, terminating the process, terminating the activation group, or terminating the thread. If an exception that is not handled occurs in a secondary thread and moves all the way to the first invocation in the thread without being handled, the resulting action is to terminate the thread. During this movement, if the exception hits a control boundary and is not handled, it may terminate the process.

The integrated language environment (ILE) C was present on the system before the POSIX signals implementation. Therefore, the ILE C uses the robust AS/400 exception model to implement ANSI C signals (`raise()`, `signal()`, `abort()`). The ILE C also provides the generation of an ANSI C signal when it detects a hardware exception. Thus, using the `signal()` API, you can monitor and handle AS/400 hardware exceptions.

A signal is never automatically generated for an exception message. AS/400 hardware/software exceptions cannot be detected using asynchronous signal mechanisms. In other words, if you use `sigaction()` for the `SIGSEGV` signal, you will not detect that signal when a pointer that is not valid is used. If you use `signal()`, you will detect `SIGSEGV` when your code uses an invalid pointer.

If the preferred signal model is the asynchronous signal model, you can use AS/400 exception handlers or ANSI C signal handlers to generate a asynchronous signal when those events occur.

The following example shows how an error caused by dividing by zero and the use of an invalid pointer might be changed into an asynchronous signal. The following example uses ANSI C signal handlers to perform the signal mapping.

Example

```
#define _MULTI_THREADED
#include <stdio.h>
#include <qp0z1170.h>
#include <time.h>
#include <signal.h>
#include <pthread.h>
#include "check.h"

void myAnsiSignalMapperHdlr(int sigNumber);
void *threadfunc1(void *parm);
void *threadfunc2(void *parm);
```



```

void *threadfunc1(void *parm)
{
    char *p=NULL;
    printf("Thread1: Unhandled exception (pointer fault) about to happen\n");
    *p = `!';
    printf("Thread1: After exception\n");
    return NULL;
}

void *threadfunc2(void *parm)
{
    int i1=0, i2=1, i3=0;
    printf("Thread2: Unhandled exception (divide by zero) about to happen\n");
    i1 = i2 / i3;
    printf("Thread2: After exception\n");
    return NULL;
}

void myAnsiSignalMapperHdlr(int sigNumber) {
    /* In a multithreaded environment, this is slightly difficult. We have
to*/
    /* re-enable the ANSI C handler immediately, because that is the way it
*/
    /* is defined. (A better alternative may be direct monitor exception
*/
    /* handlers which are always valid in the function which they are
*/
    /* registered, and with direct monitors, we can catch the hardware
*/
    /* exception before it is converted to an ANSI C signal
*/
    signal(SIGALL, myAnsiSignalMapperHdlr);
    /* Since ANSI C signals and hardware exceptions will only be handled in
*/
    /* the same thread that caused them, we will send the POSIX signal to
*/
    /* the calling thread (The signal will be delivered before returning from
*/
    /* pthread_kill().
*/
    printf("Mapping ANSI signal to POSIX signal %d\n", sigNumber);
    pthread_kill(pthread_self(), sigNumber);
    return;
}

void fpViolationHldr(int sigNumber) {
    printf("Thread 0x%.8x %%.8x "
        "Handled floating point failure SIGFPE (signal %d)\n",
        pthread_getthreadid_np(), sigNumber);
    /* By definition, returning from a POSIX signal handler handles the
signal*/
}

void segFaultHdlr(int sigNumber) {
    printf("Thread 0x%.8x %%.8x "
        "Handled segmentation violation SIGSEGV (signal %d)\n",
        pthread_getthreadid_np(), sigNumber);
    /* By definition, returning from a POSIX signal handler handles the
signal*/
}

```

```

int main(int argc, char **argv)
{
    int                rc=0;
    pthread_t          threadid;
    struct sigaction    actions;
    void                *status;

    printf("----- Setup Signal Mapping/Handling -----\\n");
    printf("- Register ANSI C signal handler to map ALL\\n"
           "  ANSI C signals & hardware exceptions to POSIX signals\\n");
    signal(SIGALL, myAnsiSignalMapperHdlr);

    printf("- Register normal POSIX signal handling mechanisms\\n"
           "  for floating point violations, and segmentation faults\\n"
           "- Other signals take the default action for asynchronous
signals\\n");
    memset(&actions, 0, sizeof(actions));
    sigemptyset(&actions.sa_mask);
    actions.sa_flags = 0;
    actions.sa_handler = fpViolationHldr;

    rc = sigaction(SIGFPE,&actions,NULL);
    checkResults("sigaction for SIGFPE\\n", rc);

    actions.sa_handler = segFaultHldr;
    rc = sigaction(SIGSEGV,&actions,NULL);
    checkResults("sigaction for SIGSEGV\\n", rc);

    printf("----- Start memory fault thread -----\\n");
    printf("Create a thread\\n");
    rc = pthread_create(&threadid, NULL, threadfunc1, NULL);
    checkResults("pthread_create()\\n", rc);

    rc = pthread_join(threadid, &status);
    checkResults("pthread_join()\\n", rc);

    printf("----- Start divide by 0 thread -----\\n");
    printf("Create a thread\\n");
    rc = pthread_create(&threadid, NULL, threadfunc2, NULL);
    checkResults("pthread_create()\\n", rc);

    rc = pthread_join(threadid, &status);
    checkResults("pthread_join()\\n", rc);

    printf("Main completed\\n");
    return 0;
}

```

Example Output

```

----- Setup Signal Mapping/Handling -----
- Register ANSI C signal handler to map ALL
  ANSI C signals & hardware exceptions to POSIX signals
- Register normal POSIX signal handling mechanisms
  for floating point violations, and segmentation faults
- Other signals take the default action for asynchronous signals
----- Start memory fault thread -----
Create a thread
Thread1: Unhandled exception (pointer fault) about to happen
Mapping ANSI signal to POSIX signal 5

```

```

Thread 0x00000000 00000022 Handled segmentation violation SIGSEGV (signal 5)
Thread1: After exception
----- Start divide by 0 thread -----
Create a thread
Thread2: Unhandled exception (divide by zero) about to happen
Mapping ANSI signal to POSIX signal 2
Thread 0x00000000 00000023 Handled floating point failure SIGFPE (signal 2)
Thread2: After exception
Main completed

```

The following example shows how a divide by zero error, and a dereference of a pointer that is not valid might be mapped to generate a POSIX (asynchronous) signal. This example uses AS/400 exception handlers to perform the signal mapping.

Example

```

#define _MULTI_THREADED
#include <stdio.h>
#include <stdlib.h>
#include <qp0z1170.h>
#include <time.h>
#include <signal.h>
#include <except.h>
#include <qusec.h> /* System API error Code structure */
#include <qmh.h> /* Message Hanlder common defs */
#include <qmhchgmem.h> /* Change exception message */
#include <pthread.h>
#include "check.h"

void myHardwareExceptionMapper(_INTRPT_Hndlr_Parms_T *exception);
void *threadfunc1(void *parm);
void *threadfunc2(void *parm);

void *threadfunc1(void *parm)
{
    char *p=NULL;
    /* Watch for all ESCAPE type exceptions. Other types may be used for
    */
    /* job log messages or C++ exceptions or other control flow in the
    process*/
    /* Adjust the message type as required by your application.
    */
    #pragma exception_handler (myHardwareExceptionMapper, 0, _C1_ALL,
    _C2_MH_ESCAPE)
    printf("Thread1: Unhandled exception (pointer fault) about to happen\n");
    *p = '!';
    printf("Thread1: After exception\n");
    #pragma disable_handler
    return NULL;
}

void *threadfunc2(void *parm)
{
    int i1=0, i2=1, i3=0;
    /* Watch for all ESCAPE type exceptions. Others types may be used for
    */
    /* job log messages or C++ exceptions or other control flow in the
    process*/
    /* Adjust the message type as required by your application.

```

```

*/
#pragma exception_handler (myHardwareExceptionMapper, 0, _C1_ALL,
_C2_MH_ESCAPE)
    printf("Thread2: Unhandled exception (divide by zero) about to happen\n");
    i1 = i2 / i3;
    printf("Thread2: After exception\n");
#pragma disable_handler
    return NULL;
}

void myHardwareExceptionMapper(_INTRPT_Hndlr_Parms_T *exInfo) {
    int          sigNumber;
    Qus_EC_t      errorCode = {0};          /* system API error structure
*/

    printf("Handling system exception\n");
    /* The exception information is available inside the exInfo structure
*/
    /* for this example, we are going to handle all exceptions and then map
*/
    /* them to an \Qappropriate' signal number. We are allowed to decide the
*/
    /* signal mapping however is appropriate for our application.
*/
    if (!memcmp(exInfo->Msg_Id, "MCH3601", 7)) {
        sigNumber = SIGSEGV;
    }
    else if (!memcmp(exInfo->Msg_Id, "MCH1211", 7)) {
        sigNumber = SIGFPE;
    }
    else {
        printf("Unexpected exception! Not Handling!\n");
        abort();
    }
    /* Even if the exception is \Qexpected', we are going to handle it and try
*/
    /* to deliver it as a POSIX signal. Note that we SHOULD NOT HANDLE
*/
    /* exceptions that are unexpected to us. Most code cannot tolerate
*/
    /* getting back into it once the exception occurred, and we could get
into*/
    /* a nice exception loop.
*/

    /* See the system API reference for a description of QMHCHGEM
*/
    QMHCHGEM(&exInfo->Target, 0, &exInfo->Msg_Ref_Key, QMH_MOD_HANDLE,
        (char *)NULL, 0, &errorCode);
    if (errorCode.Bytes_Available != 0) {
        printf("Failed to handle exception. Error Code = %7.7s\n",
            errorCode.Exception_Id);
        return;
    }
    printf("Mapping Exception %7.7s to POSIX signal %d\n",
        exInfo->Msg_Id, sigNumber);
    /* At this point the exception is handled. If the POSIX signal handler
*/
    /* returns, then the signal will be handled, and all will be complete
*/
    pthread_kill(pthread_self(), sigNumber);
}

```

```

    return;
}

void fpViolationHldr(int sigNumber) {
    printf("Thread 0x%.8x %.8x "
           "Handled floating point failure SIGFPE (signal %d)\n",
           pthread_getthreadid_np(), sigNumber);
    /* By definition, return from a POSIX signal handler handles the signal
    */
}

void segFaultHldr(int sigNumber) {
    printf("Thread 0x%.8x %.8x "
           "Handled segmentation violation SIGSEGV (signal %d)\n",
           pthread_getthreadid_np(), sigNumber);
    /* By definition, returning from a POSIX signal handler handles the
    signal*/
}

int main(int argc, char **argv)
{
    int                rc=0;
    pthread_t          threadid;
    struct sigaction    actions;
    void                *status;

    printf("----- Setup Signal Mapping/Handling ----- \n");
    printf("- The threads will register AS/400 Exception handler to map\n"
           "  hardware exceptions to POSIX signals\n");

    printf("- Register normal POSIX signal handling mechanisms\n"
           "  for floating point violations, and segmentation faults\n"
           "- Other signals take the default action for asynchronous
signals\n");
    memset(&actions, 0, sizeof(actions));
    sigemptyset(&actions.sa_mask);
    actions.sa_flags = 0;
    actions.sa_handler = fpViolationHldr;

    rc = sigaction(SIGFPE, &actions, NULL);
    checkResults("sigaction for SIGFPE\n", rc);

    actions.sa_handler = segFaultHldr;
    rc = sigaction(SIGSEGV, &actions, NULL);
    checkResults("sigaction for SIGSEGV\n", rc);

    printf("----- Start memory fault thread ----- \n");
    printf("Create a thread\n");
    rc = pthread_create(&threadid, NULL, threadfunc1, NULL);
    checkResults("pthread_create()\n", rc);

    rc = pthread_join(threadid, &status);
    checkResults("pthread_join()\n", rc);

    printf("----- Start divide by 0 thread ----- \n");
    printf("Create a thread\n");
    rc = pthread_create(&threadid, NULL, threadfunc2, NULL);
    checkResults("pthread_create()\n", rc);

    rc = pthread_join(threadid, &status);
    checkResults("pthread_join()\n", rc);
}

```

```

    printf("Main completed\n");
    return 0;
}

```

Output

```

----- Setup Signal Mapping/Handling -----
- The threads will register AS/400 Exception handler to map
  hardware exceptions to POSIX signals
- Register normal POSIX signal handling mechanisms
  for floating point violations, and segmentation faults
- Other signals take the default action for asynchronous signals
----- Start memory fault thread -----
Create a thread
Thread1: Unhandled exception (pointer fault) about to happen

Handling system exception
Mapping Exception MCH3601 to POSIX signal 5
Thread 0x00000000 00000024 Handled segmentation violation SIGSEGV (signal 5)
Thread1: After exception
----- Start divide by 0 thread -----
Create a thread
Thread2: Unhandled exception (divide by zero) about to happen
Handling system exception
Mapping Exception MCH1211 to POSIX signal 2
Thread 0x00000000 00000025 Handled floating point failure SIGFPE (signal 2)
Thread2: After exception
Main completed

```

Mutexes can be named to aid in application debug

The AS/400 threads support of mutexes allows the application to name mutexes. Named mutexes can be used to aid in problem determination. The performance and behavioral characteristics of named mutexes are identical to normal mutexes.

When an application is using mutexes and has deadlocked, you may be able to determine which mutexes are being used by the application more easily if the mutexes being used are named.

You can use the **DSPJOB** CL command to help debug the application. From **DSPJOB**, choose option **19 - Display mutexes, if active** or option **20 - Display threads, if active** to view the mutexes and threads being used by the application.

See [pthread_mutexattr_setname_np\(\)--Set Name in Mutex Attributes Object](#) and [pthread_mutexattr_getname_np\(\)--Get Name from Mutex Attributes Object](#) if you would like to use named mutexes in your application.

Header files for Pthread functions

Programs that use the Pthread functions must include one or more header files that contain information that the functions need. Header files include the following:

- Macro definitions
- Data type definitions
- Structure definitions
- Function prototypes

The header files are provided in the QSYSINC library which can be installed as an option. Make sure QSYSINC is on your system before compiling programs that use these header files.

In release Version 4 Release 2, these header files are available on your system only if you have also loaded and applied the limited availability program temporary fix (PTF) or I-Listed programming request for price quotation (PRPQ) that provides the header file support. The PTF number is 5769SS1-J664741. The I-Listed PRPQ is "5799-XTH, IBM Pthread Kernel API's for AS/400", number P84311.

Where to Find Header Files

Name of Header File	Name of File in QSYSINC	Name of Member
pthread.h	H	PTHREAD
sched.h	H	SCHED

You can display a header file in QSYSINC by using one of the following methods:

- Use your editor. For example, to display the **pthread.h** header file using the Source Entry Utility editor, enter the following command:

```
STRSEU SRCFILE(QSYSINC/H) SRCMBR(PTHREAD) OPTION(5)
```

- Use the Display Physical File Member command. For example, to display the **sched.h** header file, enter the following command:

```
DSPPFM FILE(QSYSINC/H) MBR(SCHED)
```

You can print a header file in QSYSINC by using one of the following methods:

- Use your editor. For example, to print the **pthread.h** header file using the Source Entry Utility editor, enter the following command:

```
STRSEU SRCFILE(QSYSINC/H) SRCMBR(PTHREAD) OPTION(6)
```

- Use your Copy File command. For example, to print the **sched.h** header file, enter the following command:

```
CPYF FROMFILE(QSYSINC/H) TOFILE(*PRINT) FROMMBR(SCHED)
```


Pthread glossary

A

attribute object

Any of the Pthreads data structures that are used to specify the initial states when creating certain resources (threads, mutexes, and condition variables). A thread attribute object can be used to create a thread. A mutex attributes object can be used to create a mutex. A condition attributes object can be used to create a condition. Functions that create attribute objects are `pthread_attr_init()`, `pthread_mutexattr_init()`, and `pthread_condattr_init()`.

C

cancel

A cancel is delivered to a thread when `pthread_cancel()` is issued and stops a thread. A cancel can be held pending if the target thread has cancellation `DISABLED` or `DEFERRED`. The cancel may be acted upon when cancellation is set to `ENABLED` or `ASYNCHRONOUS`.

cancellation cleanup handler

A function registered to perform some cleanup action. Cancellation cleanup handlers are called if a thread calls `pthread_exit()` or is the target of a `pthread_cancel()`. Cancellation cleanup handlers are stacked onto a cancellation cleanup stack and can be pushed and popped using the `pthread_cleanup_push()` and `pthread_cleanup_pop()` functions.

cancellation point

A function that causes a pending cancel to be delivered if the cancellation state is `ENABLED`, and the cancellation type is `DEFERRED`. `pthread_testcancel()` can be used to create a cancellation point. For a list of other functions that are cancellation points, see `pthread_cancel()`.

cancellation state

Either of two values (`ENABLED` or `DISABLED`) that describe whether cancels in the current thread are acted upon or held pending. If `ENABLED`, the cancellation is acted upon immediately based on the current cancellation type. If `DISABLED`, the cancel is held pending until it is `ENABLED`. You can modify the cancellation state using the `pthread_setcancelstate()` function.

cancellation type

Either of two values (`DEFERRED` or `ASYNCHRONOUS`) that describe how cancels are acted upon in the current thread when the cancellation state is `ENABLED`. If `DEFERRED`, the cancel is held pending, if `ASYNCHRONOUS`, the cancel is acted upon immediately, thus ending the thread with a status of `PTHREAD_CANCELED`. You can modify the cancellation type using the `pthread_setcanceltype()` function.

condition variable

An abstraction that allows a thread to wait for an event to occur. The condition variable is used with a Boolean predicate that indicates the presence or absence of the event and a mutex that protects both the predicate and the resources associated with the event. The condition variable has no ownership associated with it. See `pthread_cond_init()`, and other functions whose names begin with `pthread_cond_`.

D

detach a thread

To mark a thread so that the system reclaims the thread resources when the thread ends. If the thread has already ended, the resources are freed immediately. After a thread's resources are freed, the exit status is no longer available, and the thread cannot be detached or joined to. Use the `pthread_attr_setdetachstate()`, or `pthread_detach()` functions to detach a thread, or the `pthread_join()` function to wait for and then detach a

thread.

E

exit status

The return value from a thread. A variable of type **void ***, which typically contains some pointer to a control block pointer or return value, that shows under what conditions the thread ended. The thread can be ended and the exit status can be set by returning from the thread start routine, by calling `pthread_exit()`, or by canceling a thread using `pthread_cancel()`.

G

global mutex

A single mutex that is stored globally to the process that is provided by the pthreads library to allow easy serialization (a mechanism that allows only one thread to act at one time) to application resources. See the functions `pthread_lock_global_np()` or `pthread_unlock_global_np()`.

I

initial thread

The thread that is started automatically by the system when a job or process is started. Every job has at least one thread. That thread is often referred to as the initial thread or the primary thread. Threads other than the initial thread are referred to as secondary threads. If the initial thread ends, it causes all secondary threads and the job to end. See also 'Secondary thread'.

J

join to a thread

To wait for a thread to complete, detach the thread, and optionally return its exit status. Use `pthread_join()` to wait for a thread to complete.

M

main thread

See initial thread.

multithread capable

This term is specific to AS/400. See thread capable.

multithreaded

A process that has multiple active threads. In the AS/400 documentation, the term multithreaded is sometimes used as a synonym for multithread capable.

mutex

An abstraction that allows two or more threads to cooperate in a MUTual EXclusion protocol that allows safe access to shared resources. See `pthread_mutex_init()` or other functions whose names begin with `pthread_mutex_`. Also see recursive mutex, named mutex, global mutex.

N

named mutex

A mutex with an associated text name used for identification and debugging. The name is used in some system dumps and debug or thread-management user interfaces. The name does not affect the behavior of the mutex, only the ability to debug the use of that mutex. The Pthread run-time names all mutexes by default. See the functions `pthread_mutexattr_setname_np()` or `pthread_mutexattr_getname_np()`.

O

orphaned mutex

A mutex that was held by a thread when that thread ended. Any application data or resources associated with the mutex are most likely in an inconsistent state if a mutex is orphaned. An orphaned mutex is not available to be locked by another thread and causes a locking thread to block indefinitely or to get the EBUSY error when attempting to trylock the mutex.

P

POSIX thread handle

The `pthread_t` data type that is returned to a creator of a POSIX thread. The `pthread_t` represents an opaque handle to the POSIX thread. It should not be modified except through the use of the pthread functions. The `pthread_create()` or `pthread_self()` function returns the POSIX thread handle. The `pthread_equal()` function can be used to confirm whether two handles refer to the same thread. The POSIX thread handle is sometimes referred to as the thread ID.

primary thread

See initial thread.

Pthread

Shorthand for POSIX or Single Unix Specification Thread, as in 'the interfaces described in this document are based on the POSIX standard (ANSI/IEEE Standard 1003.1, 1996 Edition OR ISO/IEC 9945-1: 1996) and the Single UNIX Specification, Version 2, 1997'.

R

recursive mutex

A mutex that can be acquired again by the owning thread. A recursive mutex does not become unlocked until the number of unlock requests equals the number of successful lock requests. A non-recursive (i.e., normal) mutex causes an EDEADLK error if an attempt is made by the owning thread to lock it a second time. See the functions `pthread_mutexattr_setkind_np()` or `pthread_mutexattr_getkind_np()`.

S

scheduling parameters

Information describing the scheduling characteristics of a thread. The `sched_param` structure contains scheduling parameters. On AS/400, the scheduling parameters allow you to only specify the priority of the thread. Scheduling Policy is restricted to the proprietary AS/400 scheduling policy. Use the `pthread_attr_setschedparam()`, `pthread_attr_getschedparam()`, `pthread_setschedparam()`, or `pthread_getschedparam()` functions to manipulate scheduling parameters.

scheduling policy

Information describing which algorithm is used to schedule threads within the process or system. Some scheduling policies are Round Robin or FIFO. AS/400 uses the `SCHED_OTHER` constant to indicate the delay cost scheduling that the system uses. The scheduling parameter functions support only the `SCHED_OTHER` policy, and the `pthread_attr_getschedpolicy()` and `pthread_attr_setschedpolicy()` functions are not supported.

scope

Information describing whether the scheduling policy indicates that threads compete directly with other threads within the process or the system. AS/400 schedules threads within the system, and the `pthread_attr_setscope()` and `pthread_attr_getscope()` functions are not supported.

secondary thread

Any thread started by or on behalf of the application that is not the initial thread. Secondary threads are started by invoking `pthread_create()` or another library service that creates threads. Secondary threads have no parent/child relationship.

signal

An asynchronous mechanism for interrupting the processing of a thread. The system delivers a signal to a thread when the application programmer takes explicit or implicit action to cause the signal to be delivered. The signal can be sent to a thread or process, but is always delivered to a specific thread.

signal handler

A function registered by the application programmer that the system executes when a signal is delivered to a thread. The function runs immediately in the thread, interrupting any application processing that is in progress.

signal safe

A function, macro or operating system service that can be called safely from a signal handler. The function always acts in a well-defined manner. It does not rely on any external state or locks that might be in an inconsistent state at the time the signal handler function is invoked by the system.

signal unsafe

A function, macro or operating system service that cannot be called safely from within a signal handler. A signal unsafe function may acquire locks or otherwise change the state of a resource. When the signal is delivered to the thread, the signal handler runs. The state of the resource or the lock managed by the signal unsafe function is unknown because it was interrupted by the signal before it completed. If the signal unsafe function is called again, the results are non-deterministic.

T

thread

An independent sequence of execution of program code and processing context inside a process. A unique unit of work or flow of control within a process. A thread runs a procedure asynchronously with other threads running the same or different procedures within the process. All threads within a process equally share activation group and process resources (Heap storage, static storage, open files, socket descriptors, other communications ports, environment variables, etc.). A thread has few resources (mutexes, locks, automatic storage, thread specific storage) that are not shared. On a multiprocessor system, multiple threads in a process can run concurrently.

thread capable job

On AS/400, the only job that can create threads. Certain system behavior and the architecture of the process changes slightly to support AS/400 threads. If a job is not thread capable, attempts to create a thread result in the `EBUSY` error. You can create a thread capable process by using the `spawn()` interface or by using other AS/400 job-creation commands that allow you to specify that the new job should be thread capable.

thread ID

The unique integral number can be used to identify the thread. This integral number is available for retrieval via the `pthread_getunique_np()` interface. Although no Pthread interfaces use the integral thread ID to identify a thread for manipulation, thread ID is sometimes used to describe the `pthread_t` data type that represents the abstraction to a thread. See POSIX thread handle.

thread local storage (TLS)

See thread specific storage.

threadsafe

A function, macro or operating system service that can be called from multiple threads in a process at the same time. The function always acts in a well-defined manner. The end results are as if the function was called by each thread in turn, even though all of the threads were running the function at the same time. Some APIs have restrictions about how they can be called in order for them to be thread safe. See the API documentation for all APIs or system services that you use in a multithreaded job.

thread specific storage

Storage that is not shared between threads, but that can be accessed by all functions within that thread. Usually, thread specific storage is indexed by a key. The key is a global value visible to all threads, and it is used to retrieve the thread-specific value of the storage associated with that key. Also called thread private storage, thread local storage or TLS. See the `pthread_getspecific()`, `pthread_setspecific()`, `pthread_key_create()`, and `pthread_key_delete()` functions.

thread unsafe

A function, macro, or operating system service that cannot be called from multiple threads is called thread unsafe. If this function is used in multiple threads or in a process that has multiple threads active, the results are undefined. A thread unsafe function can corrupt or negatively interact with data in another function (thread safe or otherwise) that appears to be unrelated to the first function. Do NOT use thread unsafe functions in your multithreaded application. Do NOT call programs or service programs that use thread-unsafe functions. See the API documentation for all APIs or system services that you use in a multithreaded job.

Other Sources of Pthread Information

The following sources also provide information about Pthreads.

- "Programming with POSIX Threads" by David R. Butenhof, ISBN#: 0201633922
- "Threads Primer: A Guide to Solaris Multithreaded Programming " by Bil Lewis and Daniel J. Berg, Prentice Hall, ISBN#: 0134436989
- The following standards are the base reference documents that the APIs in this section have originated from.
 - ANSI/IEEE 1003.1 1996 (A.K.A. ISO/IEC 9945-1 1996)
 - The Single Unix Specification, Version 2, 1997
- The Internet newsgroup comp.programming.threads

Writing and compiling threaded programs

When writing and compiling code that use threads or that run in a threaded job, make sure to do the following:

- Ensure that all of the APIs or system services that you use are threadsafe. See the [Multithreaded applications](#) for an introduction to threads and general information about AS/400 threads.



- Insert the following lines into any module that uses the thread data types or definitions.

```
#define _MULTI_THREADED  
#include <pthread.h>
```

The preprocessor definition **_MULTI_THREADED** must come before the **pthread.h**.

See [Header files for Pthread functions](#) for more information on header files.

See [Using the _MULTI_THREADED preprocessor definition](#) for more information on the **_MULTI_THREADED** preprocessor definition.

- Compile the program normally; use the **CRTCMOD** followed by the **CRTPGM** or **CRTSRVPGM** commands. You can also use the **CRTBNDC** CL command to create your threaded program.
-  Since Pthread APIs can operate on functions and data which could exist in different compilation units (modules), the same storage model and data model must be used throughout all compilation units within a program or service program that uses Pthread APIs. Otherwise, unpredictable behavior and failures will occur. Refer to "iSeries ILE Concepts V5R1" chapter 4 "Teraspace and single-level store" for more information on storage model and data model.

Running threaded programs

When you run a threaded program, the job that runs a threaded program must be specially initialized by the system to support threads. Currently, several mechanisms allow you to start a job that is capable of creating multiple kernel threads:

- Use the OS/400 QShell Interpreter. In the QShell Interpreter, a program gets descriptors 0, 1, and 2 as the standard files; the parent and child I/O is directed to the console. The QShell interpreter allows you to run multithreaded programs as if they were interactive. See [Qshell Interpreter](#) for a description of the QIBM_MULTI_THREADED shell variable, which, when set to 'Y', allows you to run multithreaded programs the same way you run any other program. The QShell Interpreter is option 30 of Base OS/400.
- Use the **spawn()** API. The **spawn()** API has a flag in the spawn inheritance structure that allows you to turn on the multithread capability for the child job. The **QUSRTOOL** library also provides source code and an example CL command to allow you to create and use a **SPAWN** CL command in a way that is similar to the **SBMJOB** CL command. See the [SPAWN CL command, QUSRTOOL example](#) for more information.
- Use the **SBMJOB** CL command. Setting the 'Allow multiple threads' parameter (keyword **ALWMLTTHD**) on the CL command allows you to turn on the multithread capability of the submitted job.
- Use the **CRTJOB** CL command to create a special job description; then create your job using a mechanism that will use the job description. Setting the 'Allow multiple threads' parameter (keyword **ALWMLTTHD**) on the job description allows you to turn on the multithread capability of the jobs that are created using that job description.

Troubleshooting Pthread errors

The following are common errors users encounter when programming with Pthreads. Follow the appropriate link to find instructions for correcting these errors:

- [Cannot find header files pthread.h or qp0ztype.h or qp0zptha.h](#)
- [Thread creation \(pthread_create\(\)\) fails with EBUSY or 3029](#)
- [Mixing thread models or API sets](#)
- [Reserved fields must be binary zero](#)
- [Powerful OS/400 cleanup mechanisms allow application deadlock \(cancel_handler and C++ automatic destructors\)](#)
- [Thread creation using C++ methods as target does not work](#)
- [MCH3402 from pointer returned by pthread_join\(\)](#)

Cannot find header files pthread.h or qp0ztype.h or qp0zptha.h

You may find that your compilation fails because the system header files required to compile a threaded program or to use the threaded interfaces cannot be found. This problem has one of several causes:

- If you get failure messages similar to the following:

```
KULACK/QCSRC/MYPM line 5: Unable to find #include file *LIBL/H(PTHREAD).
```

you might have one of two problems:

- Either your system does not have the C header files for openness (the QSYSINC library) installed, you are on a Version 4 Release 2 system and you do not have the PTF installed (PTF number 5769SS1-J664741) that provides the Pthread header files.
- Your compile command is not searching the correct locations for system header files.

In order to correct these problems, do one of the following:

- Install the Openness includes (System Openness Includes, 5769-SS1 Option 3) and the QSYSINC library, install the PTF support (PTF number 5769SS1-J664741) for kernel threads header files.
 - Correct your search paths or library list.
- If you get failure messages similar to the following:

```
QSYSINC/H/PTHREAD line 48: Unable to find #include file QCPA/H(PTHREAD).  
QSYSINC/H/PTHREAD line 60: #error "#ifndef _MULTI_THREADED"  
QSYSINC/H/PTHREAD line 61: #error "#ifndef QP0Z_CPA_THREADS_PRESENT"
```

you have forgotten to define the **_MULTI_THREADED** preprocessor symbol. Use the C preprocessor statement `#define _MULTI_THREADED` in your application, or define **_MULTI_THREADED** on the **CRTCMOD** or other compile command that you use to compile your modules. Because the CPA toolkit supported threads before kernel threads being introduced on AS/400, if you do not define **_MULTI_THREADED** when compiling your C modules, the system attempts to compile your application using the CPA header files. The recommended threads model is kernel threads. You must define **_MULTI_THREADED** when you compile your application.

- If you get failure messages similar to the following:

```
QCPA/H/PTHREAD line 171: Unable to find #include file *LIBL/H(QP0ZTYPE).  
QCPA/H/PTHREAD line 183: Unable to find #include file *LIBL/H(QP0ZPTH). 
```

you have forgotten to define the **_MULTI_THREADED** preprocessor symbol. Use the C preprocessor statement `#define _MULTI_THREADED` in your application, or define **_MULTI_THREADED** on the **CRTCMOD** or other compile command that you use to compile your modules. Because the CPA toolkit supported threads prior to kernel threads being introduced on AS/400, if you do not define **_MULTI_THREADED** when compiling your C modules, the system attempts to compile your application using the CPA header files. The recommended threads model is kernel threads. You must define **_MULTI_THREADED** when you compile your application.

Thread creation (pthread_create()) fails with EBUSY or 3029

Because many parts of the operating system are not yet thread safe, not every AS/400 job can start threads. The **pthread_create()** API fails with the **EBUSY** error when the process is not allowed to create threads. See [Running threaded programs](#) for information about how to start a job that can create threads.

Mixing thread models or API sets

If you mix Pthread APIs with other threads management APIs that might be provided on the system, your application can enter an unknown state. For example, you should not use Java or the IBM open class libraries threads implementations to manipulate a thread that was created using the Pthread APIs. Similarly, if you use a Pthread API like `pthread_cancel()` on a thread created and managed by the JVM, you can get unexpected results.

The following example demonstrates this problem. A Java application creates several Java threads. One Java thread runs normally and eventually calls a native method. The native method uses the **`pthread_self()`** API to store the POSIX thread handle for the thread. The native method then returns to Java and continues to run normal Java code in the Java virtual machine (JVM). Eventually, another Java thread in the application calls a native method. The new native method uses the stored POSIX thread handle in a call to **`pthread_cancel()`**. This causes the Java thread to be terminated with Pthread semantics. The Java thread cleanup requirements or the tendency of Java to end the thread with a Java exception may not be honored. The application may not get the results that you expect. Do not manipulate threads from one thread model with APIs from another.

The following example also demonstrates this problem. The priorities of a Pthread may sometimes be manipulated using both Pthread and AS/400 proprietary interfaces. If they are manipulated, the priorities are always set correctly; however the priority returned from the Pthread interface **`pthread_getschedparam()`** is only correct if the priority was always set using either the **`pthread_setschedparam()`** interface or another interface, but not both. If multiple interfaces have been used to set the priority of the thread, **`pthread_getschedparam()`** always returns the priority set by the last **`pthread_setschedparam()`**.

Reserved fields must be binary zero

The AS/400 implementation of many APIs requires that reserved fields in certain parameters or data structures be set to binary zero. Before using a structure as input to an API or system service. You should initialize the structure using `memset()` or an initialization API provided by the system, such as `pthread_condattr_init()`. Using structures with reserved fields that are non-zero causes the `EINVAL` error.

Powerful OS/400 cleanup mechanisms allow application deadlock (cancel_handler and C++ automatic destructors)

AS/400 provides a set of powerful cleanup mechanisms. In OS/400, an application has the ability to register a cancel handler. Your application can enable a cancel handler by using the `#pragma cancel_handler` preprocessor statement if it is written in C or C++ or by using the **CEERTX()** API.

A cancel handler is similar to a Pthread cancellation cleanup handler. However, a cancel handler runs whenever the stack frame or function for which it was registered ends in any way other than a normal return. Pthread cancellation cleanup handlers run only when the thread is terminated with **pthread_exit()** or **pthread_cancel()** or when the thread returns from the threads start routine.

The cancel handler is guaranteed to run for all conditions that cause the stack frame to end (other than return), such as thread termination, job termination, calls to **exit()**, **abort()**, exceptions that percolate up the stack, and cancel stack frames. Similarly, C++ destructors for automatic C++ objects are guaranteed to run when the stack frame (function) or scope in which it was registered ends.

These mechanisms ensure that your application can always clean up its resources. With the added power of these mechanisms, an application can easily cause a deadlock.

The following is an example of such a problem. An application has a function **foo()** that registers a cancel handler called **cleanup()**. The function **foo()** is called by multiple threads in the application. The application is ended abnormally with a call to **abort()** or by system operator intervention (with the **ENDJOB *IMMED CL** command). When this job is ended, every thread is immediately terminated. When the system terminates a thread by terminating each call stack entry in the thread, it eventually reaches the function **foo()** in that thread. When function **foo()** is reached, the system recognizes that it must not remove that function from the call stack without running the function **cleanup()**, and so the system runs **cleanup()**. Because your application is multithreaded, all of the job ending and cleanup processing proceeds in parallel in each thread. Also, because **abort()** or **ENDJOB *IMMED** was used, the current state and location of each thread in your application is cannot be determined. When the **cleanup()** function runs, it is very difficult for the application to correctly assume that any specific cleanup can be done. Any resources that the **cleanup()** function attempts to acquire may be held by other threads in the process, other jobs in the system, or possibly by the same thread running the **cleanup()** function. The state of application variables or resources that your application manipulates may be in an inconsistent state because the call to **abort()** or **ENDJOB *IMMED** asynchronously interrupted every thread in the process at the same time. The application can easily reach a deadlock when running the cancel handlers or C++ destructors. Do not attempt to acquire locks or resources in cancel handlers or C++ automatic object destructors without preparing for the possibility that the resources cannot be acquired.

Important

Neither a cancel handler nor a destructor for a C++ object can prevent the call stack entry from being terminated, but the termination of the call stack entry (and therefore the job or thread) is delayed until the cancel handler or destructor completes.

If the cancel handler or destructor does not complete, the system does not continue terminating the call stack entry (and possibly the job or thread). The only alternative at this point is to use the **WRKJOB CL** command (option 20) to end the thread, or the **ENDJOB *IMMED CL** command. If the **ENDJOB *IMMED** command causes a cancel handler to run in the first place, the only option left is the **ENDJOBABN CL** command because any remaining cancel handlers are still guaranteed to run.

The **ENDJOBABN CL** command is not recommended. The **ENDJOBABN** command causes the job to be terminated with no further cleanup allowed (application or operating system). If the application is suspended while trying to access certain operating system resources, those resources may be damaged. If operating system resources are damaged, you may need to take various reclaim, deletion, or recovery steps and, in extreme conditions, restart the system.

Recommendations

If you want to cleanup your job or application, you can use one of the following mechanisms:

- If you want to do process level or activation group cleanup for normal termination, use the C **atexit()** function to register your cleanup function. The **atexit()** function provides a mechanism to run cleanup after the activation group and possibly the threads, are terminated. This action significantly reduces the complexity.
- If you always want a chance to do process level or activation group cleanup in all cases (normal and abnormal), you could use the Register Activation Group Exit (**CEE4RAGE()**) system API. The **CEE4RAGE()** function provides a mechanism to run cleanup after the activation group (and possibly the threads) are terminated. This action significantly reduces the complexity.
- You can safely use cancel handlers. Simplify your cancel handlers so that they only unlock or release resources and do not attempt to acquire any new resources or locks.
- You can remove your cancel handlers and create a CL command, program, or tool that terminates your application in a more controlled fashion:
 - One possibility is a tool that uses a signal to terminate the application. When the signal comes in, your application can get control in a single location (preferably by using the **sigwait()** API to safely and synchronously get the signal), and then perform some level of cleanup. Then it can use **exit()** or **abort()** to end the application from within. Often this action is sufficient to remove the complexity.
 - A second possibility is to use the **ENDJOB *CNTRLD** CL command and have your application dedicate a thread to watching for the controlled end condition. The application thread can use the **QUSRJOBI** (Get Job Information) or the **QWCRTVCA** (Retrieve Current Attributes) APIs to look at the **End Status** information associated with your job. The **End Status** indicates that the job is ending in a controlled fashion, and your application can take safe and synchronous steps to clean up and exit.
 - A third possibility is to use the asynchronous signals support and set up a handler for the SIGTERM asynchronous signal. Support has been added to the system so that, if an **ENDJOB *CNTRLD** is done and the target job has a handler registered for the SIGTERM signal, that signal gets delivered to the target job. You should dedicate a thread for handling signals by using the **sigwait()** API in the dedicated thread. When the signal handling thread detects a SIGTERM signal using the **sigwait()** API, it can safely clean up and terminate the application. The system support for the delivery of the SIGTERM signal when **ENDJOB *CNTRLD** is issued was added in the base OS/400 in Version 4 Release 3 Modification 0 and is also available in Version 4 Release 2 Modification 0 via program temporary fixes (PTFs) 5769SS1-SF47161 and 5769SS1-SF47175. For more information about ending your job, see the [Work Management](#) topic..
 - A fourth possibility includes other interprocess communications (IPC) mechanisms that can also be used to indicate that your application should terminate in a safe and controlled fashion.
- If you want to do thread-level cleanup, use the pthread APIs, such as **pthread_cleanup_push()**, **pthread_cleanup_pop()**, and **pthread_key_create()** to create cancellation cleanup functions that run when the thread terminates under normal conditions. Often your cleanup functions do not need to run when the job ends. The most common use for these functions is to free heap storage or unlock resources. Unlocking resources is safe in a cancel handler, and you do not need to use **free()** on heap storage when the entire job is ending anyway.

Thread creation using C++ methods as target does not work

Often, as a C++ programmer, you may want to abstract the concept of a thread into a C++ class. To do this, you must realize that the Pthread APIs are C language APIs. The Pthread APIs use functions that have C linkage and calling conventions. For your application to successfully use the pthread functions, you must provide helper functions of the appropriate type and linkage for the Pthread APIs that take function pointers as parameters.

When sharing objects between threads, always be aware of which thread is manipulating the object, which thread is responsible for freeing the object, and what thread safety issues are created by sharing objects between threads.

The following example shows how to successfully create a program that abstracts a thread into a C++ class. It can be easily extended to provide a mechanism by which the thread creation and manipulation itself is also encapsulated into the class.

Example

```
/* This C++ example must be compiled with VisualAge C++ for OS/400 */
#define _MULTI_THREADED
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <pthread.h>

class ThreadClass {
public:
    ThreadClass(char *s) {
        data1 = 42; data2 = strlen(s);
        strncpy(str, s, sizeof(str)-1);
        str[49]=0;
    }
    void *run(void);
private:
    int      data1;
    int      data2;
    char      str[50];
};

extern "C" void *ThreadStartup(void *);

int main(int argc, char **argv)
{
    ThreadClass *t=NULL;
    pthread_t    thread;
    int          rc;
    // Use printf instead of cout.
    // At the time this test was written, the C++ standard class library
    // was not thread safe.
    printf("Entered test %s\n", argv[0]);

    printf("Create a ThreadClass object\n");
    t = new ThreadClass("Testing C++ object/thread creation\n");

    printf("Start a real thread to process the ThreadClass object\n");
    // #define COMPILE_ERROR
    #ifndef COMPILE_ERROR
```



```

// This is an ERROR. You cannot create a thread by using a pointer
// to a member function. Thread creation requires a C linkage function.
// If you remove the comments from the line `#define COMPILE_ERROR'
// the compiler will give a message similar to this:
//      "ATESTCPP0.C", line 46.53: 1540-055: (S) "void*(ThreadClass::*)()"
//      cannot be converted to "extern "C" void*(*)(void*)".
    rc = pthread_create(&thread, NULL, ThreadClass::run, NULL);
#else
// Instead, this is the correct way to start a thread on a C++ object
    rc = pthread_create(&thread, NULL, ThreadStartup, t);
#endif
if (rc) {
    printf("Failed to create a thread\n");
    exit(EXIT_FAILURE);
}

printf("Waiting for thread to complete\n");
rc = pthread_join(thread, NULL);
if (rc) {
    printf("Failed to join to the thread, rc=%d\n");
    exit(EXIT_FAILURE);
}
printf("Testcase complete\n");
exit(EXIT_SUCCESS);
}

// This function is a helper function. It has normal C linkage, and is
// as the base for newly created ThreadClass objects. It runs the
// run method on the ThreadClass object passed to it (as a void *).
// After the ThreadClass method completes normally (i.e returns),
// we delete the object.
void *ThreadStartup(void *_tgtObject) {
    ThreadClass *tgtObject = (ThreadClass *)_tgtObject;
    printf("Running thread object in a new thread\n");
    void *threadResult = tgtObject->run();
    printf("Deleting object\n");
    delete tgtObject;
    return threadResult;
}

void *ThreadClass::run(void)
{
    printf("Entered the thread for object %.8x %.8x %.8x %.8x\n", this);

    printf("Object identity:  %d, %d: %s\n", data1, data2, str);
    return NULL;
}

```

Output

```

Entered test QPOWTEST/ACPPPOBJ
Create a ThreadClass object
Start a real thread to process the ThreadClass object
Waiting for thread to complete
Running thread object in a new thread
Entered the thread for object 80000000 00000000 d017dad2 57001f60
Object identity:  42, 35: Testing C++ object/thread creation
Deleting object
Testcase complete

```

MCH3402 from pointer returned by pthread_join()

Be sure that no threads return pointers to items that can be destroyed when a thread terminates. For example, the threads stack is transitory. It needs to exist only for the life of the thread, and it may be destroyed when the thread terminates. If you return the address of an automatic variable or use the address of an automatic variable as an argument to pthread_exit(), you may experience MCH3402 errors when you use the address.

Example

The following example contains code that brings up the MCH3402 error.

```
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include "check.h"

void *threadfunc(void *parm)
{
    int          rc = 2;
    printf("Inside secondary thread, return address of local variable.\n");
    return &rc;  /* THIS IS AN ERROR! */
    /* AT THIS POINT, THE STACK FOR THIS THREAD MAY BE DESTROYED */
}

int main(int argc, char **argv)
{
    pthread_t      thread;
    int            rc=1;
    void           *status;

    printf("Enter Testcase - %s\n", argv[0]);

    printf("Create thread that returns status incorrectly\n");
    rc = pthread_create(&thread, NULL, threadfunc, NULL);
    checkResults("pthread_create()\n", rc);

    printf("Join to thread\n");
    rc = pthread_join(thread, &status);
    checkResults("pthread_join()\n", rc);
    printf("Checking results from thread. Expect MCH3402\n");
    /* Monitor for the MCH3402 exception in this range */
#pragma exception_handler(TestOk, 0, 0, _C2_ALL, _CTLA_HANDLE_NO_MSG,
"MCH3402")
    rc = *(int *)status;
#pragma disable_handler
TestFailed:
    printf("Did not get secondary thread results (exception) as expected!\n");
    goto TestComplete;

TestOk:      /* Control goes here for an MCH3402 exception */
    printf("Got an MCH3402 as expected\n");

TestComplete:
    printf("Main completed\n");
    return rc;
}
```

Output

MCH3402 from pointer returned by pthread_join()

Enter Testcase - QP0WTEST/TPJOIN7

Create thread that returns status incorrectly

Join to thread

Inside secondary thread, return address of local variable.

Checking results from thread. Expect MCH3402

Got an MCH3402 as expected

Main completed

Information on the Pthread API examples

The API documentation includes example programs for each API. The [header file](#) shown below is used for all of the examples. It should be named **check.h** (member **CHECK** in file **H** in a library in the library list).

In most cases, error checking that is contained in the examples causes the program to exit() if any failure is detected. In some cases, error checking is left out of the examples for brevity. In general, the error checking that is provided should not be considered complete enough for all applications. All return codes from any system functions should be validated and appropriate action should be taken when failures occur.

The examples are provided "as-is" for demonstration and education purposes only. They do not necessarily provide or implement an appropriate level of error checking to be used for production code and should not be used directly for that purpose.

Be sure to see [Writing and compiling threaded programs](#) and [Running threaded programs](#) for more information about compiling and running the example programs.

To create the examples, make sure the member **CHECK** is created in a file **H** in your library list. Use **CRTCMOD** on the name that you download the member to, then use **CRTPGM** to link the module into a program object. Alternatively, you can use **CRTBND** to compile and link the program in one step.

When you run the example programs, you must be aware of a requirement:

The job that runs a threaded program must be specially initialized by the system to support threads. Currently, several mechanisms allow you to start a job that is capable of creating multiple kernel threads.

- Use the OS/400 QShell Interpreter
- Use the spawn() API.
- Use the SMJOB CL command.
- Use the CRTJOB CL command to create a special job description, then create your job using a mechanism that will use the job description.

See [Running threaded programs](#) for detailed information on these methods.

File check.h used by API examples programs

This example header file must be in the library list when you compile the example programs.

```
#ifndef _CHECK_H
#define _CHECK_H
/* headers used by a majority of the example program */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>

/* Simple function to check the return code and exit the program
   if the function call failed
   */
static void checkResults(char *string, int rc) {
    if (rc) {
        printf("Error on : %s, rc=%d",
              string, rc);
        exit(EXIT_FAILURE);
    }
    return;
}

#endif
```