NAME

pthread_mutexattr_getrobust, pthread_mutexattr_setrobust − get and set the robustness attribute of a mutex attributes object

SYNOPSIS

```
#include <pthread.h>

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

Compile and link with *−pthread*.

Feature Test Macro Requirements for glibc (see **feature_test_macros**(7)):

pthread_mutexattr_getrobust(), **pthread_mutexattr_setrobust**():
_POSIX_C_SOURCE >= 200809L

DESCRIPTION

The **pthread_mutexattr_getrobust**() function places the value of the robustness attribute of the mutex attributes object referred to by *attr* in **robustness*. The **pthread_mutexattr_setrobust**() function sets the value of the robustness attribute of the mutex attributes object referred to by *attr* to the value specified in **robustness*.

The robustness attribute specifies the behavior of the mutex when the owning thread dies without unlocking the mutex. The following values are valid for *robustness*:

PTHREAD_MUTEX_STALLED
: This is the default value for a mutex attributes object. If a mutex is initialized with the **PTHREAD_MUTEX_STALLED** attribute and its owner dies without unlocking it, the mutex remains locked afterwards and any future attempts to call **pthread_mutex_lock**(3) on the mutex will block indefinitely.

PTHREAD_MUTEX_ROBUST
: If a mutex is initialized with the **PTHREAD_MUTEX_ROBUST** attribute and its owner dies without unlocking it, any future attempts to call **pthread_mutex_lock**(3) on this mutex will succeed and return **EOWNERDEAD** to indicate that the original owner no longer exists and the mutex is in an inconsistent state. Usually after **EOWNERDEAD** is returned, the next owner should call **pthread_mutex_consistent**(3) on the acquired mutex to make it consistent again before using it any further.

  If the next owner unlocks the mutex using **pthread_mutex_unlock**(3) before making it consistent, the mutex will be permanently unusable and any subsequent attempts to lock it using **pthread_mutex_lock**(3) will fail with the error **ENOTRECOVERABLE**. The only permitted operation on such a mutex is **pthread_mutex_destroy**(3).

  If the next owner terminates before calling **pthread_mutex_consistent**(3), further **pthread_mutex_lock**(3) operations on this mutex will still return **EOWNERDEAD**.

Note that the *attr* argument of **pthread_mutexattr_getrobust**() and **pthread_mutexattr_setrobust**() should refer to a mutex attributes object that was initialized by **pthread_mutexattr_init**(3), otherwise the behavior is undefined.

RETURN VALUE

On success, these functions return 0. On error, they return a positive error number.

In the glibc implementation, **pthread_mutexattr_getrobust**() always return zero.

ERRORS

EINVAL
: A value other than **PTHREAD_MUTEX_STALLED** or **PTHREAD_MUTEX_ROBUST** was passed to **pthread_mutexattr_setrobust**().

VERSIONS

pthread_mutexattr_getrobust() and **pthread_mutexattr_setrobust**() were added to glibc in version 2.12.

CONFORMING TO

POSIX.1-2008.

NOTES

In the Linux implementation, when using process-shared robust mutexes, a waiting thread also receives the **EOWNERDEAD** notification if the owner of a robust mutex performs an **execve**(2) without first unlocking the mutex. POSIX.1 does not specify this detail, but the same behavior also occurs in at least some other implementations.

Before the addition of **pthread_mutexattr_getrobust**() and **pthread_mutexattr_setrobust**() to POSIX, glibc defined the following equivalent nonstandard functions if **_GNU_SOURCE** was defined:

int pthread_mutexattr_getrobust_np(const pthread_mutexattr_t ****attr*,**
int ****robustness*);**
int pthread_mutexattr_setrobust_np(const pthread_mutexattr_t ****attr*,**
int *robustness***);**

Correspondingly, the constants **PTHREAD_MUTEX_STALLED_NP** and **PTHREAD_MUTEX_ROBUST_NP** were also defined.

These GNU-specific APIs, which first appeared in glibc 2.4, are nowadays obsolete and should not be used in new programs.

EXAMPLE

The program below demonstrates the use of the robustness attribute of a mutex attributes object. In this program, a thread holding the mutex dies prematurely without unlocking the mutex. The main thread subsequently acquires the mutex successfully and gets the error **EOWNERDEAD**, after which it makes the mutex consistent.

The following shell session shows what we see when running this program:

```
$ ./a.out
[original owner] Setting lock...
[original owner] Locked. Now exiting without unlocking.
[main thread] Attempting to lock the robust mutex.
[main thread] pthread_mutex_lock() returned EOWNERDEAD
[main thread] Now make the mutex consistent
[main thread] Mutex is now consistent; unlocking
```

Program source

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#include <errno.h>

#define handle_error_en(en, msg) \
               do { errno = en; perror(msg); exit(EXIT_FAILURE); } while (0)

static pthread_mutex_t mtx;

static void *
original_owner_thread(void *ptr)
{
    printf("[original owner] Setting lock...\n");
    pthread_mutex_lock(&mtx);
    printf("[original owner] Locked. Now exiting without unlocking.\n");
```

```
        pthread_exit(NULL);
    }

    int
    main(int argc, char *argv[])
    {
        pthread_t thr;
        pthread_mutexattr_t attr;
        int s;

        pthread_mutexattr_init(&attr);
                                    /* initialize the attributes object */
        pthread_mutexattr_setrobust(&attr, PTHREAD_MUTEX_ROBUST);
                                   /* set robustness */

        pthread_mutex_init(&mtx, &attr);   /* initialize the mutex */

        pthread_create(&thr, NULL, original_owner_thread, NULL);

        sleep(2);

        /* "original_owner_thread" should have exited by now */

        printf("[main thread] Attempting to lock the robust mutex.\n");
        s = pthread_mutex_lock(&mtx);
        if (s == EOWNERDEAD) {
            printf("[main thread] pthread_mutex_lock() returned EOWNERDEAD\n");
            printf("[main thread] Now make the mutex consistent\n");
            s = pthread_mutex_consistent(&mtx);
            if (s != 0)
                handle_error_en(s, "pthread_mutex_consistent");
            printf("[main thread] Mutex is now consistent; unlocking\n");
            s = pthread_mutex_unlock(&mtx);
            if (s != 0)
                handle_error_en(s, "pthread_mutex_unlock");

            exit(EXIT_SUCCESS);
        } else if (s == 0) {
            printf("[main thread] pthread_mutex_lock() unexpectedly succeeded\n");
            exit(EXIT_FAILURE);
        } else {
            printf("[main thread] pthread_mutex_lock() unexpectedly failed\n");
            handle_error_en(s, "pthread_mutex_lock");
        }
    }
```

SEE ALSO

get_robust_list(2), **set_robust_list**(2), **pthread_mutex_init**(3), **pthread_mutex_consistent**(3), **pthread_mutex_lock**(3), **pthreads**(7)

COLOPHON

This page is part of release 5.02 of the Linux *man-pages* project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at https://www.kernel.org/doc/man−pages/.