Full credit is given to the above companies including the OS that this PDF file was generated!

Red Hat Enterprise Linux Release 9.2 Manual Pages on 'pthread_mutexattr_destroy.3p' command

$ man pthread_mutexattr_destroy.3p

PTHREAD_MUTEXATTR_DESTROY(3POSIX Programmer's ManPTHREAD_MUTEXATTR_DESTROY(3P)

PROLOG

This manual page is part of the POSIX Programmer's Manual. The Linux implementation of this interface may differ (consult the corresponding Linux manual page for details of Linux behavior), or the interface may not be implemented on Linux.

NAME

pthread_mutexattr_destroy, pthread_mutexattr_init ? destroy and ini? tialize the mutex attributes object

SYNOPSIS

#include <pthread.h>

int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);

int pthread_mutexattr_init(pthread_mutexattr_t *attr);

DESCRIPTION

The pthread_mutexattr_destroy() function shall destroy a mutex at? tributes object; the object becomes, in effect, uninitialized. An im? plementation may cause pthread_mutexattr_destroy() to set the object referenced by attr to an invalid value.

A destroyed attr attributes object can be reinitialized using pthread_mutexattr_init(); the results of otherwise referencing the ob? ject after it has been destroyed are undefined.

The pthread_mutexattr_init() function shall initialize a mutex at? tributes object attr with the default value for all of the attributes

defined by the implementation.

Results are undefined if pthread_mutexattr_init() is called specifying an already initialized attr attributes object.

After a mutex attributes object has been used to initialize one or more mutexes, any function affecting the attributes object (including de? struction) shall not affect any previously initialized mutexes.

The behavior is undefined if the value specified by the attr argument to pthread_mutexattr_destroy() does not refer to an initialized mutex attributes object.

RETURN VALUE

Upon successful completion, pthread_mutexattr_destroy() and pthread_mu? texattr_init() shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The pthread_mutexattr_init() function shall fail if:

ENOMEM Insufficient memory exists to initialize the mutex attributes object.

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

The following sections are informative.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

If an implementation detects that the value specified by the attr argu? ment to pthread_mutexattr_destroy() does not refer to an initialized mutex attributes object, it is recommended that the function should fail and report an [EINVAL] error.

See pthread_attr_destroy() for a general explanation of attributes. At? tributes objects allow implementations to experiment with useful exten? sions and permit extension of this volume of POSIX.1?2017 without changing the existing functions. Thus, they provide for future extensi? bility of this volume of POSIX.1?2017 and reduce the temptation to

standardize prematurely on semantics that are not yet widely imple? mented or understood.

Examples of possible additional mutex attributes that have been dis? cussed are spin_only, limited_spin, no_spin, recursive, and metered. (To explain what the latter attributes might mean: recursive mutexes would allow for multiple re-locking by the current owner; metered mu? texes would transparently keep records of queue length, wait time, and so on.) Since there is not yet wide agreement on the usefulness of these resulting from shared implementation and usage experience, they are not yet specified in this volume of POSIX.1?2017. Mutex attributes objects, however, make it possible to test out these concepts for pos? sible standardization at a later time.

Mutex Attributes and Performance

Care has been taken to ensure that the default values of the mutex at? tributes have been defined such that mutexes initialized with the de? faults have simple enough semantics so that the locking and unlocking can be done with the equivalent of a test-and-set instruction (plus possibly a few other basic instructions).

There is at least one implementation method that can be used to reduce the cost of testing at lock-time if a mutex has non-default attributes. One such method that an implementation can employ (and this can be made fully transparent to fully conforming POSIX applications) is to se? cretly pre-lock any mutexes that are initialized to non-default at? tributes. Any later attempt to lock such a mutex causes the implementa? tion to branch to the ``slow path'' as if the mutex were unavailable; then, on the slow path, the implementation can do the ``real work'' to lock a non-default mutex. The underlying unlock operation is more com? plicated since the implementation never really wants to release the pre-lock on this kind of mutex. This illustrates that, depending on the hardware, there may be certain optimizations that can be used so that whatever mutex attributes are considered ``most frequently used'' can be processed most efficiently.

The existence of memory mapping functions in this volume of POSIX.1?2017 leads to the possibility that an application may allocate the synchronization objects from this section in memory that is ac? cessed by multiple processes (and therefore, by threads of multiple processes).

In order to permit such usage, while at the same time keeping the usual case (that is, usage within a single process) efficient, a process-shared option has been defined.

If an implementation supports the _POSIX_THREAD_PROCESS_SHARED option, then the process-shared attribute can be used to indicate that mutexes or condition variables may be accessed by threads of multiple pro? cesses.

The default setting of PTHREAD_PROCESS_PRIVATE has been chosen for the process-shared attribute so that the most efficient forms of these syn? chronization objects are created by default.

Synchronization variables that are initialized with the PTHREAD_PROCESS_PRIVATE process-shared attribute may only be operated on by threads in the process that initialized them. Synchronization variables that are initialized with the PTHREAD_PROCESS_SHARED process-shared attribute may be operated on by any thread in any process that has access to it. In particular, these processes may exist beyond the lifetime of the initializing process. For example, the following code implements a simple counting semaphore in a mapped file that may be used by many processes.

```
/* sem.h */
struct semaphore {
    pthread_mutex_t lock;
    pthread_cond_t nonzero;
    unsigned count;
};
typedef struct semaphore semaphore_t;
semaphore_t *semaphore_create(char *semaphore_name);
semaphore_t *semaphore_open(char *semaphore_name);
```

```
void semaphore_post(semaphore_t *semap);
void semaphore_wait(semaphore_t *semap);
void semaphore_close(semaphore_t *semap);
/* sem.c */
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <pthread.h>
#include "sem.h"
semaphore_t *
semaphore_create(char *semaphore_name)
{
int fd;
    semaphore_t *semap;
    pthread_mutexattr_t psharedm;
    pthread_condattr_t psharedc;
    fd = open(semaphore_name, O_RDWR | O_CREAT | O_EXCL, 0666);
    if (fd < 0)
        return (NULL);
    (void) ftruncate(fd, sizeof(semaphore_t));
    (void) pthread_mutexattr_init(&psharedm);
    (void) pthread_mutexattr_setpshared(&psharedm,
        PTHREAD_PROCESS_SHARED);
    (void) pthread_condattr_init(&psharedc);
    (void) pthread_condattr_setpshared(&psharedc,
        PTHREAD_PROCESS_SHARED);
    semap = (semaphore_t *) mmap(NULL, sizeof(semaphore_t),
            PROT_READ | PROT_WRITE, MAP_SHARED,
            fd, 0);
    close (fd);
    (void) pthread_mutex_init(&semap->lock, &psharedm);
    (void) pthread_cond_init(&semap->nonzero, &psharedc);
```

```
    semap->count = 0;
    return (semap);
}
semaphore_t *
semaphore_open(char *semaphore_name)
{
    int fd;
    semaphore_t *semap;
    fd = open(semaphore_name, O_RDWR, 0666);
    if (fd < 0)
        return (NULL);
    semap = (semaphore_t *) mmap(NULL, sizeof(semaphore_t),
            PROT_READ | PROT_WRITE, MAP_SHARED,
            fd, 0);
    close (fd);
    return (semap);
}
void
semaphore_post(semaphore_t *semap)
{
    pthread_mutex_lock(&semap->lock);
    if (semap->count == 0)
        pthread_cond_signal(&semapx->nonzero);
    semap->count++;
    pthread_mutex_unlock(&semap->lock);
}
void
semaphore_wait(semaphore_t *semap)
{
    pthread_mutex_lock(&semap->lock);
    while (semap->count == 0)
        pthread_cond_wait(&semap->nonzero, &semap->lock);
    semap->count--;
```

```
    pthread_mutex_unlock(&semap->lock);
}
void
semaphore_close(semaphore_t *semap)
{
    munmap((void *) semap, sizeof(semaphore_t));
}
```

The following code is for three separate processes that create, post, and wait on a semaphore in the file /tmp/semaphore. Once the file is created, the post and wait programs increment and decrement the count? ing semaphore (waiting and waking as required) even though they did not initialize the semaphore.

```
/* create.c */
#include "pthread.h"
#include "sem.h"
int
main()
{
    semaphore_t *semap;
    semap = semaphore_create("/tmp/semaphore");
    if (semap == NULL)
        exit(1);
    semaphore_close(semap);
    return (0);
}
/* post */
#include "pthread.h"
#include "sem.h"
int
main()
{
    semaphore_t *semap;
    semap = semaphore_open("/tmp/semaphore");
```

```
    if (semap == NULL)
        exit(1);
    semaphore_post(semap);
    semaphore_close(semap);
    return (0);
}
/* wait */
#include "pthread.h"
#include "sem.h"
int
main()
{
    semaphore_t *semap;
    semap = semaphore_open("/tmp/semaphore");
    if (semap == NULL)
        exit(1);
    semaphore_wait(semap);
    semaphore_close(semap);
    return (0);
}
```

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_cond_destroy(), pthread_create(), pthread_mutex_destroy()

The Base Definitions volume of POSIX.1?2017, <pthread.h>

is the referee document. The original Standard can be obtained online at http://www.opengroup.org/unix/online.html .

Any typographical or formatting errors that appear in this page are most likely to have been introduced during the conversion of the source files to man page format. To report such errors, see https://www.ker?nel.org/doc/man-pages/reporting_bugs.html .