Extrait du man POSIX.4

14 Février 2002

NAME

pthread_create - create a new thread

SYNOPSIS

```
#include <pthread.h>

int  pthread_create(pthread_t  *  thread, pthread_attr_t *
attr, void * (*start_routine)(void *), void * arg);
```

DESCRIPTION

pthread_create creates a new thread of control that exe cutes concurrently with the calling thread. The new thread applies the function start_routine passing it arg as first argument. The new thread terminates either explicitly, by calling pthread_exit(3), or implicitly, by returning from the start_routine function. The latter case is equivalent to calling pthread_exit(3) with the result returned by start_routine as exit code.

The attr argument specifies thread attributes to be applied to the new thread. See pthread_attr_init(3) for a complete list of thread attributes. The attr argument can also be NULL, in which case default attributes are used: the created thread is joinable (not detached) and has default (non real-time) scheduling policy.

RETURN VALUE

On success, the identifier of the newly created thread is stored in the location pointed by the thread argument, and a 0 is returned. On error, a non-zero error code is returned.

ERRORS

EAGAIN not enough system resources to create a process for the new thread.

EAGAIN more than PTHREAD_THREADS_MAX threads are already active.

```
NAME
       pthread_join - wait for termination of another thread

SYNOPSIS
       #include <pthread.h>

       int pthread_join(pthread_t th, void **thread_return);

DESCRIPTION
       pthread_join  suspends the execution of the calling thread
       until the thread identified by th  terminates,  either  by
       calling pthread_exit(3) or by being cancelled.

       If  thread_return  is  not NULL, the return value of th is
       stored in the location pointed to by  thread_return.   The
       return  value  of  th  is  either  the argument it gave to
       pthread_exit(3), or PTHREAD_CANCELED if th was  cancelled.

       The  joined  thread  th  must be in the joinable state: it
       must not have been detached using pthread_detach(3) or the
       PTHREAD_CREATE_DETACHED attribute to pthread_create(3).

       When  a  joinable  thread terminates, its memory resources
       (thread descriptor and stack) are  not  deallocated  until
       another  thread  performs  pthread_join  on it. Therefore,
       pthread_join must be called once for each joinable  thread
       created to avoid memory leaks.

       At most one thread can wait for the termination of a given
       thread. Calling pthread_join  on  a  thread  th  on  which
       another  thread is already waiting for termination returns
       an error.

RETURN VALUE
       On  success, the return value of th is stored in the loca
       tion pointed to by thread_return, and 0  is  returned.  On
       error, a non-zero error code is returned.

ERRORS
       ESRCH  No  thread  could  be  found  corresponding to that
              specified by th.

       EINVAL The th thread has been detached.

       EINVAL Another thread is already waiting on termination of
              th.

       EDEADLK
              The th argument refers to the calling thread.
```

```
NAME
       pthread_exit - terminate the calling thread

SYNOPSIS
       #include <pthread.h>

       void pthread_exit(void *retval);

DESCRIPTION
       pthread_exit  terminates  the  execution  of  the  calling
       thread.  All cleanup handlers that have been set  for  the
       calling  thread  with pthread_cleanup_push(3) are executed
       in reverse order (the most recently pushed handler is exe
       cuted  first).  Finalization functions for thread-specific
       data are then called for all keys that have non- NULL val
       ues  associated  with  them  in  the  calling  thread (see
       pthread_key_create(3)).  Finally, execution of the calling
       thread is stopped.

       The  retval argument is the return value of the thread. It
       can   be   consulted   from   another   thread    using
       pthread_join(3).

RETURN VALUE
       The pthread_exit function never returns.
```

NAME

pthread_mutex_init, pthread_mutex_lock,
pthread_mutex_unlock, pthread_mutex_destroy - opera
tions on mutexes

SYNOPSIS

```
#include <pthread.h>

pthread_mutex_t fastmutex = PTHREAD_MUTEX_INITIALIZER;

pthread_mutex_t  recmutex  =  PTHREAD_RECURSIVE_MUTEX_INI
TIALIZER_NP;

pthread_mutex_t               errchkmutex               =
PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP;

int   pthread_mutex_init(pthread_mutex_t   *mutex,   const
pthread_mutexattr_t *mutexattr);

int pthread_mutex_lock(pthread_mutex_t *mutex));

int pthread_mutex_unlock(pthread_mutex_t *mutex);

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

DESCRIPTION

A mutex is a MUTual EXclusion device, and is useful for protecting shared data structures from concurrent modifi cations, and implementing critical sections and monitors.

A mutex has two possible states: unlocked (not owned by any thread), and locked (owned by one thread). A mutex can never be owned by two different threads simultaneously. A thread attempting to lock a mutex that is already locked by another thread is suspended until the owning thread unlocks the mutex first.

pthread_mutex_init initializes the mutex object pointed to by mutex according to the mutex attributes specified in mutexattr. If mutexattr is NULL, default attributes are used instead.

Variables of type pthread_mutex_t can also be initialized statically, using the constants PTHREAD_MUTEX_INITIALIZER (for fast mutexes), PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP (for recursive mutexes), and PTHREAD_ERRORCHECK_MUTEX_INI TIALIZER_NP (for error checking mutexes).

pthread_mutex_lock locks the given mutex. If the mutex is currently unlocked, it becomes locked and owned by the calling thread, and pthread_mutex_lock returns immedi ately. If the mutex is already locked by another thread, pthread_mutex_lock suspends the calling thread until the mutex is unlocked.

If the mutex is already locked by the calling thread, the behavior of pthread_mutex_lock depends on the kind of the mutex. If the mutex is of the ``fast'' kind, the calling thread is suspended until the mutex is unlocked, thus effectively causing the calling thread to deadlock. If the mutex is of the ``error checking'' kind, pthread_mutex_lock returns immediately with the error code EDEADLK. If the mutex is of the ``recursive'' kind, pthread_mutex_lock succeeds and returns immediately, recording the number of times the calling thread has locked the mutex. An equal number of pthread_mutex_unlock operations must be performed before the mutex returns to the unlocked state.

pthread_mutex_unlock unlocks the given mutex. The mutex is assumed to be locked and owned by the calling thread on entrance to pthread_mutex_unlock. If the mutex is of the ``fast'' kind, pthread_mutex_unlock always returns it to the unlocked state. If it is of the ``recursive'' kind, it decrements the locking count of the mutex (number of pthread_mutex_lock operations performed on it by the call ing thread), and only when this count reaches zero is the mutex actually unlocked.

On ``error checking'' mutexes, pthread_mutex_unlock actu ally checks at run-time that the mutex is locked on entrance, and that it was locked by the same thread that is now calling pthread_mutex_unlock. If these conditions are not met, an error code is returned and the mutex remains unchanged. ``Fast'' and ``recursive'' mutexes perform no such checks, thus allowing a locked mutex to be unlocked by a thread other than its owner. This is non-portable behavior and must not be relied upon.

pthread_mutex_destroy destroys a mutex object, freeing the resources it might hold. The mutex must be unlocked on entrance.

RETURN VALUE

pthread_mutex_init always returns 0. The other mutex func tions return 0 on success and a non-zero error code on error.

ERRORS

The pthread_mutex_lock function returns the following error code on error:

EINVAL the mutex has not been properly initialized.

EDEADLK
: the mutex is already locked by the calling thread (``error checking'' mutexes only).

EINVAL the mutex has not been properly initialized.

The pthread_mutex_unlock function returns the following error code on error:

EINVAL the mutex has not been properly initialized.

EPERM the calling thread does not own the mutex (‘‘error checking’’ mutexes only).

The pthread_mutex_destroy function returns the following error code on error:

EBUSY the mutex is currently locked.

NAME

sem_init, sem_wait, sem_post,
sem_destroy - operations on semaphores

SYNOPSIS

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);

int sem_wait(sem_t * sem);

int sem_post(sem_t * sem);

int sem_destroy(sem_t * sem);
```

DESCRIPTION

Semaphores are counters for resources shared between threads. The basic operations on semaphores are: increment the counter atomically, and wait until the counter is non-null and decrement it atomically.

sem_init initializes the semaphore object pointed to by sem. The count associated with the semaphore is set ini tially to value. The pshared argument indicates whether the semaphore is local to the current process (pshared is zero) or is to be shared between several processes (pshared is not zero).

sem_wait suspends the calling thread until the semaphore pointed to by sem has non-zero count. It then atomically decreases the semaphore count.

sem_post atomically increases the count of the semaphore pointed to by sem. This function never blocks and can safely be used in asynchronous signal handlers.

sem_destroy destroys a semaphore object, freeing the resources it might hold. No threads should be waiting on the semaphore at the time sem_destroy is called.

RETURN VALUE

The sem_wait and sem_getvalue functions always return 0. All other semaphore functions return 0 on success and -1 on error, in addition to writing an error code in errno.

ERRORS

The sem_init function sets errno to the following codes on error:

EINVAL value exceeds the maximal counter value SEM_VALUE_MAX

ENOSYS pshared is not zero

```
The sem_post function sets errno to  the  following  error
code on error:

       ERANGE after  incrementation,  the  semaphore value
              would exceed  SEM_VALUE_MAX  (the  semaphore
              count is left unchanged in this case)

The sem_destroy function sets errno to the following error
code on error:

       EBUSY  some threads are currently  blocked  waiting
              on the semaphore.
```