

NAME

pthread_create - create a thread

SYNOPSIS

```
cc -mt [ _f_l_a_g... ] _f_ i_l_e...- lpthread [ _l_ i_b_r_a_r_y... ]
```

```
#include <pthread.h>
```

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

DESCRIPTION

The pthread_create() function is used to create a new thread, with attributes specified by attr, within a process.

If attr is NULL, the default attributes are used. (See pthread_attr_init(3THR)). If the attributes specified by attr are modified later, the thread's attributes are not affected. Upon successful completion, pthread_create() stores the ID of the created thread in the location referenced by thread.

The thread is created executing start_routine with arg as its sole argument. If start_routine returns, the effect is as if there was an implicit call to pthread_exit() using the return value of start_routine as the exit status. Note that the thread in which main() was originally invoked differs from this. When it returns from main(), the effect is as if there was an implicit call to exit() using the return value of main() as the exit status.

The signal state of the new thread is initialised as follows:

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

Default thread creation:

```
pthread_t tid;
void *start_func(void *), *arg;

pthread_create(&tid, NULL, start_func, arg);
```

This would have the same effect as:

```
pthread_attr_t attr;

pthread_attr_init(&attr); /* initialize attr with default attributes */
pthread_create(&tid, &attr, start_func, arg);
```

User-defined thread creation: To create a thread that is scheduled on a system-wide basis, use:

```
pthread_attr_init(&attr); /* initialize attr with default attributes */
pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM); /* system-wide contention */
pthread_create(&tid, &attr, start_func, arg);
```

To customize the attributes for POSIX threads, see `pthread_attr_init(3THR)`.

A new thread created with `pthread_create()` uses the stack specified by the `_s_t_a_c_k_a_d_d_r` attribute, and the stack continues for the number of bytes specified by the `_s_t_a_c_k_s_i_z_e` attribute. By default, the stack size is 1 megabyte for 32-bit processes and 2 megabyte for 64-bit processes (see `pthread_attr_setstacksize(3THR)`). If the default is used for both the `_s_t_a_c_k_a_d_d_r` and `_s_t_a_c_k_s_i_z_e` attributes,

`pthread_create()` creates a stack for the new thread with at least 1 megabyte for 32-bit processes and 2 megabyte for 64-bit processes. (For customizing stack sizes, see NOTES).

If `pthread_create()` fails, no new thread is created and the contents of the location referenced by `_t_h_r_e_a_d` are undefined.

RETURN VALUES

If successful, the `pthread_create()` function returns 0. Otherwise, an error number is returned to indicate the error.

ERRORS

The `pthread_create()` function will fail if:

ENOMEM

The system lacked the necessary resources to create another thread.

EINVAL

The value specified by `_a_t_t_r` is invalid.

EPERM The caller does not have appropriate permission to set the required scheduling parameters or scheduling policy.

EXAMPLES

Example 1: This is an example of concurrency with multithreading. Since POSIX threads and Solaris threads are fully compatible even within the same process, this example uses `pthread_create()` if you execute `a.out 0`, or `thr_create()` if you execute `a.out 1`.

Five threads are created that simultaneously perform a time-consuming function, `sleep(10)`. If the execution of this

process is timed, the results will show that all five individual calls to `sleep` for ten-seconds completed in about ten

seconds, even on a uniprocessor. If a single-threaded process calls sleep(10) five times, the execution time will be about 50-seconds.

The command-line to time this process is:

```
/usr/bin/time a.out 0 (for POSIX threading)
```

or

```
/usr/bin/time a.out 1 (for Solaris threading)
```

```
/* cc thisfile.c -lthread -lpthread */
#define _REENTRANT /* basic 3-lines for threads */
#include <pthread.h>
#include <thread.h>

#define NUM_THREADS 5
#define SLEEP_TIME 10

void *sleeping(void *); /* thread routine */
int i;
thread_t tid[NUM_THREADS]; /* array of thread IDs */

int
main(int argc, char *argv[])
{
    if (argc == 1) {
        printf("use 0 as arg 1 to use pthread_create()\n");
        printf("or use 1 as arg 1 to use thr_create()\n");
        return (1);
    }

    switch (*argv[1]) {
    case '0': /* POSIX */
        for (i = 0; i < NUM_THREADS; i++)
            pthread_create(&tid[i], NULL, sleeping,
                (void *)SLEEP_TIME);
        for (i = 0; i < NUM_THREADS; i++)
            pthread_join(tid[i], NULL);
        break;

    case '1': /* Solaris */
        for (i = 0; i < NUM_THREADS; i++)
            thr_create(NULL, 0, sleeping, (void *)SLEEP_TIME, 0,
                &tid[i]);
        while (thr_join(NULL, NULL, NULL) == 0)
            ;
        break;
    } /* switch */

    printf("main() reporting that all %d threads have terminated\n", i);
    return (0);
} /* main */

void *
```

```

sleeping(void *arg)
{
    int sleep_time = (int)arg;
    printf("thread %d sleeping %d seconds ...\n", thr_self(), sleep_time);
    sleep(sleep_time);
    printf("\nthread %d awakening\n", thr_self());
    return (NULL);
}

```

Example 2: If main() had not waited for the completion of the other threads (using pthread_join(3THR) or thr_join(3THR)), it would have continued to process concurrently until it reached the end of its routine and the entire process would have exited prematurely (see exit(2)).

NOTES

MT application threads execute independently of each other, thus their relative behavior is unpredictable. Therefore, it is possible for the thread executing main() to finish before all other user application threads.

pthread_join(3THR), on the other hand, must specify the terminating thread (IDs) for which it will wait.

A user-specified stack size must be greater than the value PTHREAD_STACK_MIN. A minimum stack size may not accommodate the stack frame for the user thread function `_s_t_a_r_t__f_u_n_c`. If a stack size is specified, it must accommodate `_s_t_a_r_t__f_u_n_c` requirements and the functions that it may call in turn, in addition to the minimum requirement.

It is usually very difficult to determine the runtime stack requirements for a thread. PTHREAD_STACK_MIN specifies how much stack storage is required to execute a `_N_U_L_ L_s_t_a_r_t__f_u_n_c`. The total runtime requirements for stack storage are dependent on the storage required to do runtime linking, the amount of storage required by library runtimes (as printf()) that your thread calls. Since these storage parameters are not known before the program runs, it is best to use default stacks. If you know your runtime requirements or decide to use stacks that are larger than the default, then it makes sense to specify your own stacks.

Threads Library Functions pthread_exit(3THR)

NAME

pthread_exit - terminate calling thread

SYNOPSIS

```
cc -mt [ _f_l_a_g... ] _f_ i_l_e...- lpthread [ _l_ i_b_r_a_r_y... ]
```

```
#include <pthread.h>
```

```
void pthread_exit(void *value_ptr);
```

DESCRIPTION

The `pthread_exit()` function terminates the calling thread, in a similar way that `exit(3C)` terminates the calling process. If the thread is not detached, the exit status specified by `_v_a_l_u_e__p_t_r` is made available to any successful join with the terminating thread. See `pthread_join(3THR)`. Any cancellation cleanup handlers that have been pushed and not yet popped are popped in the reverse order that they were pushed and then executed. After all cancellation cleanup handlers have been executed, if the thread has any thread-specific data, appropriate destructor functions will be called in an unspecified order. Thread termination does not release any application visible process resources, including, but not limited to, mutexes and file descriptors, nor does it perform any process level cleanup actions, including, but not limited to, calling any `atexit()` routines that may exist.

An implicit call to `pthread_exit()` is made when a thread other than the thread in which `main()` was first invoked returns from the start routine that was used to create it. The function's return value serves as the thread's exit status.

The behavior of `pthread_exit()` is undefined if called from a cancellation cleanup handler or destructor function that was invoked as a result of either an implicit or explicit call to `pthread_exit()`.

After a thread has terminated, the result of access to local (auto) variables of the thread is undefined. Thus, references to local variables of the exiting thread should not be used for the `pthread_exit()` `_v_a_l_u_e__p_t_r` parameter value.

The process exits with an exit status of 0 after the last thread has been terminated. The behavior is as if the implementation called `exit()` with a 0 argument at thread termination time.

RETURN VALUES

The `pthread_exit()` function cannot return to its caller.

ERRORS

No errors are defined.

NAME

`pthread_join` - wait for thread termination

SYNOPSIS

```
cc -mt [ _f_l_a_g... ] _f_      i_l_e...- lpthread [ _l_  i_b_r_a_r_y... ]
```

```
#include <pthread.h>
```

```
int pthread_join(pthread_t thread, void **value_ptr);
```

DESCRIPTION

The `pthread_join()` function suspends processing of the calling thread until the target `_t_h_r_e_a_d` completes. `_t_h_r_e_a_d` must be a member of the current process and it cannot be a detached or daemon thread. See `pthread_create(3THR)`.

Several threads cannot wait for the same thread to complete; one thread will complete successfully and the others will terminate with an error of `ESRCH`. `pthread_join()` will not block processing of the calling thread if the target `_t_h_r_e_a_d` has already terminated.

`pthread_join()` returns successfully when the target `_t_h_r_e_a_d` terminates. If a `pthread_join()` call returns successfully with a non-null `_s_t_a_t_u_s` argument, the value passed to `pthread_exit(3THR)` by the terminating thread will be placed in the location referenced by `_s_t_a_t_u_s`.

If the `pthread_join()` calling thread is cancelled, then the target `_t_h_r_e_a_d` will remain joinable by `pthread_join()`. However, the calling thread may set up a cancellation cleanup handler on `_t_h_r_e_a_d` prior to the join call, which may detach the target thread by calling `pthread_detach(3THR)`. (See `pthread_detach(3THR)` and `pthread_cancel(3THR)`.)

RETURN VALUES

If successful, the `pthread_join()` function returns 0. Otherwise, an error number is returned to indicate the error.

ERRORS

The `pthread_join()` function will fail if:

EINVAL

The implementation has detected that the value specified by `_t_h_r_e_a_d` does not refer to a joinable thread.

ESRCH No thread could be found corresponding to that specified by the given thread ID.

The `pthread_join()` function may fail if:

EDEADLK

NOTES

`pthread_join(3THR)`, must specify the `_t_h_r_e_a_d` ID for whose termination it will wait.

Calling `pthread_join()` also "detaches" the thread, that is, `pthread_join()` includes the effect of `pthread_detach()`. Hence, if a thread were to be cancelled when blocked in

pthread_join(), an explicit detach would have to be done in the cancellation cleanup handler. In fact, the routine pthread_detach() exists mainly for this reason.

NAME

pthread_mutexattr_init, pthread_mutexattr_destroy - initialize and destroy mutex attributes object

SYNOPSIS

```
cc -mt [ _ f_ l_ a_ g... ] _ f_      i_ l_ e...-  lpthread [ _ l_   i_ b_ r_ a_ r_ y... ]  
#include <pthread.h>
```

```
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
```

```
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
```

DESCRIPTION

The pthread_mutexattr_init() function initializes a mutex attributes object `_ a_ t_ t_ r` with the default value for all of the attributes defined by the implementation.

The effect of initializing an already initialized mutex attributes object is undefined.

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

The pthread_mutexattr_destroy() function destroys a mutex attributes object; the object becomes, in effect, uninitialized. An implementation may cause pthread_mutexattr_destroy() to set the object referenced by `_ a_ t_ t_ r` to an invalid value. A destroyed mutex attributes object can be re-initialized using pthread_mutexattr_init(); the results of otherwise referencing the object after it has been destroyed are undefined.

RETURN VALUES

Upon successful completion, pthread_mutexattr_init() and pthread_mutexattr_destroy() return 0. Otherwise, an error number is returned to indicate the error.

ERRORS

The pthread_mutexattr_init() function may fail if:

ENOMEM

Insufficient memory exists to initialize the mutex attributes object.

The pthread_mutexattr_destroy() function may fail if:

EINVAL

The value specified by `_ a_ t_ t_ r` is invalid.

NAME

pthread_mutexattr_getpshared, pthread_mutexattr_setpshared -
get and set process-shared attribute

SYNOPSIS

```
cc -mt [ _f_l_a_g... ] _f_      i_l_e... - lpthread [ _l_ i_b_r_a_r_y... ]  
#include <pthread.h>
```

```
int pthread_mutexattr_getpshared(const pthread_mutexattr_t  
*attr, int *pshared);
```

```
int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr,  
int pshared);
```

DESCRIPTION

The pthread_mutexattr_getpshared() function obtains the value of the pthread_attr_t pshared attribute from the attributes object referenced by attr. The pthread_mutexattr_setpshared() function is used to set the pthread_attr_t pshared attribute in an initialized attributes object referenced by attr.

The pthread_attr_t pshared attribute is set to PTHREAD_PROCESS_SHARED to permit a mutex to be operated upon by any thread that has access to the memory where the mutex is allocated, even if the mutex is allocated in memory that is shared by multiple processes. If the pthread_attr_t pshared attribute is PTHREAD_PROCESS_PRIVATE, the mutex will only be operated upon by threads created within the same process as the thread that initialized the mutex; if threads of differing processes attempt to operate on such a mutex, the behavior is undefined. The default value of the attribute is PTHREAD_PROCESS_PRIVATE.

RETURN VALUES

Upon successful completion, pthread_mutexattr_getpshared() returns 0 and stores the value of the pthread_attr_t pshared attribute of attr into the object referenced by the pthread_attr_t pshared parameter. Otherwise, an error number is returned to indicate the error.

Upon successful completion, pthread_mutexattr_setpshared() returns 0. Otherwise, an error number is returned to indicate the error.

ERRORS

The pthread_mutexattr_getpshared() and pthread_mutexattr_setpshared() functions may fail if:

EINVAL

The value specified by attr is invalid.

EINVAL

The new value specified for the attribute is outside the range of legal values for that attribute.

NAME

pthread_mutex_init, pthread_mutex_destroy - initialize or destroy a mutex

SYNOPSIS

```
cc -mt [ _f_l_a_g... ] _f_ i_l_e... - lpthread [ _l_ i_b_r_a_r_y... ]  
#include <pthread.h>
```

```
int pthread_mutex_init(pthread_mutex_t *mutex, const  
pthread_mutexattr_t *attr);
```

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);  
pthread_mutex_t _m_u_t_e_x=
```

PTHREAD_MUTEX_INITIALIZER

DESCRIPTION

The pthread_mutex_init() function initializes the mutex referenced by _m_u_t_e_x with attributes specified by _a_t_t_r. If _a_t_t_r is _N_U_L, the default mutex attributes are used; the effect is the same as passing the address of a default mutex attributes object. Upon successful initialization, the state of the mutex becomes initialized and unlocked.

Attempting to initialize an already initialized mutex results in undefined behavior.

The pthread_mutex_destroy() function destroys the mutex object referenced by _m_u_t_e_x; the mutex object becomes, in effect, uninitialized. A destroyed mutex object can be re-initialized using pthread_mutex_init(); the results of otherwise referencing the object after it has been destroyed are undefined.

It is safe to destroy an initialized mutex that is unlocked. Attempting to destroy a locked mutex results in undefined behavior.

In cases where default mutex attributes are appropriate, the macro PTHREAD_MUTEX_INITIALIZER can be used to initialize mutexes that are statically allocated. The effect is equivalent to dynamic initialization by a call to pthread_mutex_init() with parameter _a_t_t_r specified as _N_U_L, except that no error checks are performed.

RETURN VALUES

If successful, the pthread_mutex_init() and pthread_mutex_destroy() functions return 0. Otherwise, an error number is returned to indicate the error.

ERRORS

The `pthread_mutex_init()` function will fail if:

EAGAIN

The system lacked the necessary resources (other than memory) to initialize another mutex.

ENOMEM

Insufficient memory exists to initialize the mutex.

EPERM The caller does not have the privilege to perform the operation.

The `pthread_mutex_init()` function may fail if:

EBUSY An attempt was detected to re-initialize the object referenced by `_m_u_t_e_x`, a mutex previously initialized but not yet destroyed.

EINVAL

The value specified by `_a_t_t_r` or `_m_u_t_e_x` is invalid.

The `pthread_mutex_destroy()` function may fail if:

EBUSY An attempt was detected to destroy the object referenced by `_m_u_t_e_x` while it is locked or referenced (for example, while being used in a `pthread_cond_wait(3THR)` or `pthread_cond_timedwait(3THR)`) by another thread.

EINVAL

The value specified by `_m_u_t_e_x` is invalid.

NAME

`pthread_mutex_lock`, `pthread_mutex_trylock`,
`pthread_mutex_unlock` - lock or unlock a mutex

SYNOPSIS

```
cc -mt [ _f_l_a_g... ] _f_ i_l_e... - lpthread [ _l_ i_b_r_a_r_y... ]  
#include <pthread.h>
```

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

DESCRIPTION

The mutex object referenced by `mutex` is locked by calling `pthread_mutex_lock()`. If the mutex is already locked, the calling thread blocks until the mutex becomes available. This operation returns with the mutex object referenced by

mutex in the locked state with the calling thread as its owner.

If the mutex type is `PTHREAD_MUTEX_NORMAL`, deadlock detection is not provided. Attempting to relock the mutex causes deadlock. If a thread attempts to unlock a mutex that it has not locked or a mutex that is unlocked, undefined behavior results.

If the mutex type is `PTHREAD_MUTEX_ERRORCHECK`, then error checking is provided. If a thread attempts to relock a mutex that it has already locked, an error will be returned. If a thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, an error will be returned.

If the mutex type is `PTHREAD_MUTEX_RECURSIVE`, then the mutex maintains the concept of a lock count. When a thread successfully acquires a mutex for the first time, the lock count is set to 1. Every time a thread relocks this mutex, the lock count is incremented by one. Each time the thread unlocks the mutex, the lock count is decremented by one. When the lock count reaches 0, the mutex becomes available for other threads to acquire. If a thread attempts to unlock a mutex that it has not locked or a mutex that is unlocked, an error will be returned.

If the mutex type is `PTHREAD_MUTEX_DEFAULT`, attempting to recursively lock the mutex results in undefined behavior. Attempting to unlock the mutex if it was not locked by the calling thread results in undefined behavior. Attempting to unlock the mutex if it is not locked results in undefined behavior.

The `pthread_mutex_trylock()` function is identical to `pthread_mutex_lock()` except that if the mutex object referenced by `_m_u_t_e_x` is currently locked (by any thread, including the current thread), the call returns immediately.

The `pthread_mutex_unlock()` function releases the mutex object referenced by `_m_u_t_e_x`. The manner in which a mutex is released is dependent upon the mutex's type attribute. If there are threads blocked on the mutex object referenced by `_m_u_t_e_x` when `pthread_mutex_unlock()` is called, resulting in the mutex becoming available, the scheduling policy is used to determine which thread will acquire the mutex. (In the case of `PTHREAD_MUTEX_RECURSIVE` mutexes, the mutex becomes available when the count reaches 0 and the calling thread no longer has any locks on this mutex.)

If a signal is delivered to a thread waiting for a mutex, upon return from the signal handler the thread resumes waiting for the mutex as if it was not interrupted.

RETURN VALUES

If successful, the `pthread_mutex_lock()` and `pthread_mutex_unlock()` functions return 0. Otherwise, an

error number is returned to indicate the error.

The `pthread_mutex_trylock()` function returns 0 if a lock on the mutex object referenced by `_m_u_t_e_x` is acquired. Otherwise, an error number is returned to indicate the error.

ERRORS

The `pthread_mutex_lock()` and `pthread_mutex_trylock()` functions will fail if:

EINVAL

The `_m_u_t_e_x` was created with the protocol attribute having the value `PTHREAD_PRIO_PROTECT` and the calling thread's priority is higher than the mutex's current priority ceiling.

The `pthread_mutex_trylock()` function will fail if:

EBUSY The `_m_u_t_e_x` could not be acquired because it was already locked.

The `pthread_mutex_lock()`, `pthread_mutex_trylock()` and `pthread_mutex_unlock()` functions may fail if:

EINVAL

The value specified by `_m_u_t_e_x` does not refer to an initialized mutex object.

EAGAIN

The mutex could not be acquired because the maximum number of recursive locks for `_m_u_t_e_x` has been exceeded.

The `pthread_mutex_lock()` function may fail if:

EDEADLK

The current thread already owns the mutex.

The `pthread_mutex_unlock()` function may fail if:

EPERM The current thread does not own the mutex.

When a thread makes a call to `pthread_mutex_lock()` or `pthread_mutex_trylock()`, if the symbol `_POSIX_THREAD_PRIO_INHERIT` is defined and the mutex is initialized with the protocol attribute having the value `PTHREAD_PRIO_INHERIT` and the robustness attribute having the value `PTHREAD_MUTEX_ROBUST_NP` (see `pthread_mutexattr_getrobust_np(3THR)`), the `pthread_mutex_lock()` and `pthread_mutex_trylock()` functions will fail if:

EOWNERDEAD

The last owner of this mutex died while holding the mutex. This mutex is now owned by the caller. The caller must now attempt to make the state protected by

the mutex consistent. If it is able to clean up the state, then it should call `pthread_mutex_consistent_np()` for the mutex and unlock the mutex. Subsequent calls to `pthread_mutex_lock()` and `pthread_mutex_trylock()` will behave normally, as before. If the caller is not able to clean up the state, `pthread_mutex_consistent_np()` should not be called for the mutex, but it should be unlocked. Subsequent calls to `pthread_mutex_lock()` and `pthread_mutex_trylock()` will fail to acquire the mutex with the error value `ENOTRECOVERABLE`. If the owner who acquired the lock with `EOWNERDEAD` dies, the next owner will acquire the lock with `EOWNERDEAD`.

ENOTRECOVERABLE

The mutex trying to be acquired is protecting the state that has been left irrecoverable by the mutex's last owner, who died while holding the lock. The mutex has not been acquired. This condition can occur when the lock was previously acquired with `EOWNERDEAD`, and the owner was not able to clean up the state and unlocked the mutex without making the mutex consistent.

ENOMEM

NOTES

In the current implementation of threads, `pthread_mutex_lock()`, `pthread_mutex_unlock()`, `mutex_lock()`, `mutex_unlock()`, `pthread_mutex_trylock()`, and `mutex_trylock()` do not validate the mutex type. Therefore, an uninitialized mutex or a mutex with an invalid type does not return `EINVAL`. Interfaces for mutexes with an invalid type have unspecified behavior.

Uninitialized mutexes that are allocated locally may contain junk data. Such mutexes need to be initialized using `pthread_mutex_init()` or `mutex_init()`.

NAME

`pthread_mutex_lock`, `pthread_mutex_trylock`,
`pthread_mutex_unlock` - lock or unlock a mutex

SYNOPSIS

```
cc -mt [ _f_l_a_g... ] _f_ i_l_e... - lpthread [ _l_ i_b_r_a_r_y... ]  
#include <pthread.h>
```

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

DESCRIPTION

The mutex object referenced by `mutex` is locked by calling

`pthread_mutex_lock()`. If the mutex is already locked, the calling thread blocks until the mutex becomes available. This operation returns with the mutex object referenced by `mutex` in the locked state with the calling thread as its owner.

If the mutex type is `PTHREAD_MUTEX_NORMAL`, deadlock detection is not provided. Attempting to relock the mutex causes deadlock. If a thread attempts to unlock a mutex that it has not locked or a mutex that is unlocked, undefined behavior results.

If the mutex type is `PTHREAD_MUTEX_ERRORCHECK`, then error checking is provided. If a thread attempts to relock a mutex that it has already locked, an error will be returned. If a thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, an error will be returned.

If the mutex type is `PTHREAD_MUTEX_RECURSIVE`, then the mutex maintains the concept of a lock count. When a thread successfully acquires a mutex for the first time, the lock count is set to 1. Every time a thread relocks this mutex, the lock count is incremented by one. Each time the thread unlocks the mutex, the lock count is decremented by one. When the lock count reaches 0, the mutex becomes available for other threads to acquire. If a thread attempts to unlock a mutex that it has not locked or a mutex that is unlocked, an error will be returned.

If the mutex type is `PTHREAD_MUTEX_DEFAULT`, attempting to recursively lock the mutex results in undefined behavior. Attempting to unlock the mutex if it was not locked by the calling thread results in undefined behavior. Attempting to unlock the mutex if it is not locked results in undefined behavior.

The `pthread_mutex_trylock()` function is identical to `pthread_mutex_lock()` except that if the mutex object referenced by `_m_u_t_e_x` is currently locked (by any thread, including the current thread), the call returns immediately.

The `pthread_mutex_unlock()` function releases the mutex object referenced by `_m_u_t_e_x`. The manner in which a mutex is released is dependent upon the mutex's type attribute. If there are threads blocked on the mutex object referenced by `_m_u_t_e_x` when `pthread_mutex_unlock()` is called, resulting in the mutex becoming available, the scheduling policy is used to determine which thread will acquire the mutex. (In the case of `PTHREAD_MUTEX_RECURSIVE` mutexes, the mutex becomes available when the count reaches 0 and the calling thread no longer has any locks on this mutex.)

If a signal is delivered to a thread waiting for a mutex, upon return from the signal handler the thread resumes waiting for the mutex as if it was not interrupted.

RETURN VALUES

If successful, the `pthread_mutex_lock()` and `pthread_mutex_unlock()` functions return 0. Otherwise, an error number is returned to indicate the error.

The `pthread_mutex_trylock()` function returns 0 if a lock on the mutex object referenced by `_m_u_t_e_x` is acquired. Otherwise, an error number is returned to indicate the error.

ERRORS

The `pthread_mutex_lock()` and `pthread_mutex_trylock()` functions will fail if:

EINVAL

The `_m_u_t_e_x` was created with the protocol attribute having the value `PTHREAD_PRIO_PROTECT` and the calling thread's priority is higher than the mutex's current priority ceiling.

The `pthread_mutex_trylock()` function will fail if:

EBUSY The `_m_u_t_e_x` could not be acquired because it was already locked.

The `pthread_mutex_lock()`, `pthread_mutex_trylock()` and `pthread_mutex_unlock()` functions may fail if:

EINVAL

The value specified by `_m_u_t_e_x` does not refer to an initialized mutex object.

EAGAIN

The mutex could not be acquired because the maximum number of recursive locks for `_m_u_t_e_x` has been exceeded.

The `pthread_mutex_lock()` function may fail if:

EDEADLK

The current thread already owns the mutex.

The `pthread_mutex_unlock()` function may fail if:

EPERM The current thread does not own the mutex.

When a thread makes a call to `pthread_mutex_lock()` or `pthread_mutex_trylock()`, if the symbol `_POSIX_THREAD_PRIO_INHERIT` is defined and the mutex is initialized with the protocol attribute having the value `PTHREAD_PRIO_INHERIT` and the robustness attribute having the value `PTHREAD_MUTEX_ROBUST_NP` (see `pthread_mutexattr_getrobust_np(3THR)`), the `pthread_mutex_lock()` and `pthread_mutex_trylock()` functions will fail if:

EOWNERDEAD

The last owner of this mutex died while holding the mutex. This mutex is now owned by the caller. The caller must now attempt to make the state protected by the mutex consistent. If it is able to clean up the state, then it should call `pthread_mutex_consistent_np()` for the mutex and unlock the mutex. Subsequent calls to `pthread_mutex_lock()` and `pthread_mutex_trylock()` will behave normally, as before. If the caller is not able to clean up the state, `pthread_mutex_consistent_np()` should not be called for the mutex, but it should be unlocked. Subsequent calls to `pthread_mutex_lock()` and `pthread_mutex_trylock()` will fail to acquire the mutex with the error value `ENOTRECOVERABLE`. If the owner who acquired the lock with `EOWNERDEAD` dies, the next owner will acquire the lock with `EOWNERDEAD`.

ENOTRECOVERABLE

The mutex trying to be acquired is protecting the state that has been left irrecoverable by the mutex's last owner, who died while holding the lock. The mutex has not been acquired. This condition can occur when the lock was previously acquired with `EOWNERDEAD`, and the owner was not able to clean up the state and unlocked the mutex without making the mutex consistent.

ENOMEM

NOTES

In the current implementation of threads, `pthread_mutex_lock()`, `pthread_mutex_unlock()`, `mutex_lock()`, `mutex_unlock()`, `pthread_mutex_trylock()`, and `mutex_trylock()` do not validate the mutex type. Therefore, an uninitialized mutex or a mutex with an invalid type does not return `EINVAL`. Interfaces for mutexes with an invalid type have unspecified behavior.

Uninitialized mutexes that are allocated locally may contain junk data. Such mutexes need to be initialized using `pthread_mutex_init()` or `mutex_init()`.

NAME

`pthread_mutex_lock`, `pthread_mutex_trylock`,
`pthread_mutex_unlock` - lock or unlock a mutex

SYNOPSIS

```
cc -mt [ _f_l_a_g... ] _f_ i_l_e... - lpthread [ _l_ i_b_r_a_r_y... ]  
#include <pthread.h>
```

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

DESCRIPTION

The mutex object referenced by `mutex` is locked by calling `pthread_mutex_lock()`. If the mutex is already locked, the calling thread blocks until the mutex becomes available. This operation returns with the mutex object referenced by `mutex` in the locked state with the calling thread as its owner.

If the mutex type is `PTHREAD_MUTEX_NORMAL`, deadlock detection is not provided. Attempting to relock the mutex causes deadlock. If a thread attempts to unlock a mutex that it has not locked or a mutex that is unlocked, undefined behavior results.

If the mutex type is `PTHREAD_MUTEX_ERRORCHECK`, then error checking is provided. If a thread attempts to relock a mutex that it has already locked, an error will be returned. If a thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, an error will be returned.

If the mutex type is `PTHREAD_MUTEX_RECURSIVE`, then the mutex maintains the concept of a lock count. When a thread successfully acquires a mutex for the first time, the lock count is set to 1. Every time a thread relocks this mutex, the lock count is incremented by one. Each time the thread unlocks the mutex, the lock count is decremented by one. When the lock count reaches 0, the mutex becomes available for other threads to acquire. If a thread attempts to unlock a mutex that it has not locked or a mutex that is unlocked, an error will be returned.

If the mutex type is `PTHREAD_MUTEX_DEFAULT`, attempting to recursively lock the mutex results in undefined behavior. Attempting to unlock the mutex if it was not locked by the calling thread results in undefined behavior. Attempting to unlock the mutex if it is not locked results in undefined behavior.

The `pthread_mutex_trylock()` function is identical to `pthread_mutex_lock()` except that if the mutex object referenced by `_m_u_t_e_x` is currently locked (by any thread, including the current thread), the call returns immediately.

The `pthread_mutex_unlock()` function releases the mutex object referenced by `_m_u_t_e_x`. The manner in which a mutex is released is dependent upon the mutex's type attribute. If there are threads blocked on the mutex object referenced by `_m_u_t_e_x` when `pthread_mutex_unlock()` is called, resulting in the mutex becoming available, the scheduling policy is used to determine which thread will acquire the mutex. (In the case of `PTHREAD_MUTEX_RECURSIVE` mutexes, the mutex becomes available when the count reaches 0 and the calling thread no longer has any locks on this mutex.)

If a signal is delivered to a thread waiting for a mutex,

upon return from the signal handler the thread resumes waiting for the mutex as if it was not interrupted.

RETURN VALUES

If successful, the `pthread_mutex_lock()` and `pthread_mutex_unlock()` functions return 0. Otherwise, an error number is returned to indicate the error.

The `pthread_mutex_trylock()` function returns 0 if a lock on the mutex object referenced by `_m_u_t_e_x` is acquired. Otherwise, an error number is returned to indicate the error.

ERRORS

The `pthread_mutex_lock()` and `pthread_mutex_trylock()` functions will fail if:

EINVAL

The `_m_u_t_e_x` was created with the protocol attribute having the value `PTHREAD_PRIO_PROTECT` and the calling thread's priority is higher than the mutex's current priority ceiling.

The `pthread_mutex_trylock()` function will fail if:

EBUSY The `_m_u_t_e_x` could not be acquired because it was already locked.

The `pthread_mutex_lock()`, `pthread_mutex_trylock()` and `pthread_mutex_unlock()` functions may fail if:

EINVAL

The value specified by `_m_u_t_e_x` does not refer to an initialized mutex object.

EAGAIN

The mutex could not be acquired because the maximum number of recursive locks for `_m_u_t_e_x` has been exceeded.

The `pthread_mutex_lock()` function may fail if:

EDEADLK

The current thread already owns the mutex.

The `pthread_mutex_unlock()` function may fail if:

EPERM The current thread does not own the mutex.

When a thread makes a call to `pthread_mutex_lock()` or `pthread_mutex_trylock()`, if the symbol `_POSIX_THREAD_PRIO_INHERIT` is defined and the mutex is initialized with the protocol attribute having the value `PTHREAD_PRIO_INHERIT` and the robustness attribute having the value `PTHREAD_MUTEX_ROBUST_NP` (see `pthread_mutexattr_getrobust_np(3THR)`), the `pthread_mutex_lock()` and `pthread_mutex_trylock()` functions

will fail if:

EOWNERDEAD

The last owner of this mutex died while holding the mutex. This mutex is now owned by the caller. The caller must now attempt to make the state protected by the mutex consistent. If it is able to clean up the state, then it should call `pthread_mutex_consistent_np()` for the mutex and unlock the mutex. Subsequent calls to `pthread_mutex_lock()` and `pthread_mutex_trylock()` will behave normally, as before. If the caller is not able to clean up the state, `pthread_mutex_consistent_np()` should not be called for the mutex, but it should be unlocked. Subsequent calls to `pthread_mutex_lock()` and `pthread_mutex_trylock()` will fail to acquire the mutex with the error value `ENOTRECOVERABLE`. If the owner who acquired the lock with `EOWNERDEAD` dies, the next owner will acquire the lock with `EOWNERDEAD`.

ENOTRECOVERABLE

The mutex trying to be acquired is protecting the state that has been left irrecoverable by the mutex's last owner, who died while holding the lock. The mutex has not been acquired. This condition can occur when the lock was previously acquired with `EOWNERDEAD`, and the owner was not able to clean up the state and unlocked the mutex without making the mutex consistent.

ENOMEM

NOTES

In the current implementation of threads, `pthread_mutex_lock()`, `pthread_mutex_unlock()`, `mutex_lock()`, `mutex_unlock()`, `pthread_mutex_trylock()`, and `mutex_trylock()` do not validate the mutex type. Therefore, an uninitialized mutex or a mutex with an invalid type does not return `EINVAL`. Interfaces for mutexes with an invalid type have unspecified behavior.

Uninitialized mutexes that are allocated locally may contain junk data. Such mutexes need to be initialized using `pthread_mutex_init()` or `mutex_init()`.

Threads Library Functions `pthread_mutex_init(3THR)`

NAME

`pthread_mutex_init`, `pthread_mutex_destroy` - initialize or destroy a mutex

SYNOPSIS

```
cc -mt [ _f_l_a_g... ] _f_      i_l_e... - lpthread [ _l_ i_b_r_a_r_y... ]
```

```
#include <pthread.h>
```

```
int pthread_mutex_init(pthread_mutex_t *mutex, const  
pthread_mutexattr_t *attr);
```

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);  
pthread_mutex_t _m_u_t_e_x=  
PTHREAD_MUTEX_INITIALIZER
```

DESCRIPTION

The `pthread_mutex_init()` function initializes the mutex referenced by `_m_u_t_e_x` with attributes specified by `_a_t_r`. If `_a_t_r` is `_N_U_L`, the default mutex attributes are used; the effect is the same as passing the address of a default mutex attributes object. Upon successful initialization, the state of the mutex becomes initialized and unlocked.

Attempting to initialize an already initialized mutex results in undefined behavior.

The `pthread_mutex_destroy()` function destroys the mutex object referenced by `_m_u_t_e_x`; the mutex object becomes, in effect, uninitialized. A destroyed mutex object can be re-initialized using `pthread_mutex_init()`; the results of otherwise referencing the object after it has been destroyed are undefined.

It is safe to destroy an initialized mutex that is unlocked. Attempting to destroy a locked mutex results in undefined behavior.

In cases where default mutex attributes are appropriate, the macro `PTHREAD_MUTEX_INITIALIZER` can be used to initialize mutexes that are statically allocated. The effect is equivalent to dynamic initialization by a call to `pthread_mutex_init()` with parameter `_a_t_r` specified as `_N_U_L`, except that no error checks are performed.

RETURN VALUES

If successful, the `pthread_mutex_init()` and `pthread_mutex_destroy()` functions return 0. Otherwise, an error number is returned to indicate the error.

ERRORS

The `pthread_mutex_init()` function will fail if:

EAGAIN

The system lacked the necessary resources (other than memory) to initialize another mutex.

ENOMEM

Insufficient memory exists to initialize the mutex.

EPERM The caller does not have the privilege to perform the operation.

The `pthread_mutex_init()` function may fail if:

EBUSY An attempt was detected to re-initialize the object referenced by `_m_u_t_e_x`, a mutex previously initialized but not yet destroyed.

EINVAL

The value specified by `_a_t_t_r` or `_m_u_t_e_x` is invalid.

The `pthread_mutex_destroy()` function may fail if:

EBUSY An attempt was detected to destroy the object referenced by `_m_u_t_e_x` while it is locked or referenced (for example, while being used in a `pthread_cond_wait(3THR)` or `pthread_cond_timedwait(3THR)`) by another thread.

EINVAL

The value specified by `_m_u_t_e_x` is invalid.

NAME

`pthread_cond_signal`, `pthread_cond_broadcast` - signal or broadcast a condition

SYNOPSIS

```
cc -mt [ _f_l_a_g... ] _f_i_l_e...- lpthread [ _l_i_b_r_a_r_y... ]
```

```
#include <pthread.h>
```

```
int pthread_cond_signal(pthread_cond_t *cond);
```

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

DESCRIPTION

These two functions are used to unblock threads blocked on a condition variable.

The `pthread_cond_signal()` call unblocks at least one of the threads that are blocked on the specified condition variable `_c_o_n_d` (if any threads are blocked on `_c_o_n_d`).

The `pthread_cond_broadcast()` call unblocks all threads currently blocked on the specified condition variable `_c_o_n_d`.

If more than one thread is blocked on a condition variable, the scheduling policy determines the order in which threads are unblocked. When each thread unblocked as a result of a `pthread_cond_signal()` or `pthread_cond_broadcast()` returns from its call to `pthread_cond_wait()` or `pthread_cond_timedwait()`, the thread owns the mutex with which it called `pthread_cond_wait()` or `pthread_cond_timedwait()`. The thread(s) that are unblocked

contend for the mutex according to the scheduling policy (if applicable), and as if each had called `pthread_mutex_lock()`.

The `pthread_cond_signal()` or `pthread_cond_broadcast()` functions may be called by a thread whether or not it currently owns the mutex that threads calling `pthread_cond_wait()` or `pthread_cond_timedwait()` have associated with the condition variable during their waits; however, if predictable scheduling behavior is required, then that mutex is locked by the thread calling `pthread_cond_signal()` or `pthread_cond_broadcast()`.

The `pthread_cond_signal()` and `pthread_cond_broadcast()` functions have no effect if there are no threads currently blocked on `_c_o_n_d`.

RETURN VALUES

If successful, the `pthread_cond_signal()` and `pthread_cond_broadcast()` functions return 0. Otherwise, an error number is returned to indicate the error.

EINVAL

The value `_c_o_n_d` does not refer to an initialized condition variable.

NAME

`pthread_cond_signal`, `pthread_cond_broadcast` - signal or broadcast a condition

SYNOPSIS

```
cc -mt [ _f_l_a_g... ] _f_ i_l_e...- lpthread [ _l_ i_b_r_a_r_y... ]
```

```
#include <pthread.h>
```

```
int pthread_cond_signal(pthread_cond_t *cond);
```

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

DESCRIPTION

These two functions are used to unblock threads blocked on a condition variable.

The `pthread_cond_signal()` call unblocks at least one of the threads that are blocked on the specified condition variable `_c_o_n_d` (if any threads are blocked on `_c_o_n_d`).

The `pthread_cond_broadcast()` call unblocks all threads currently blocked on the specified condition variable `_c_o_n_d`.

If more than one thread is blocked on a condition variable, the scheduling policy determines the order in which threads are unblocked. When each thread unblocked as a result of a `pthread_cond_signal()` or `pthread_cond_broadcast()` returns from its call to `pthread_cond_wait()` or

pthread_cond_timedwait(), the thread owns the mutex with which it called pthread_cond_wait() or pthread_cond_timedwait(). The thread(s) that are unblocked contend for the mutex according to the scheduling policy (if applicable), and as if each had called pthread_mutex_lock().

The pthread_cond_signal() or pthread_cond_broadcast() functions may be called by a thread whether or not it currently owns the mutex that threads calling pthread_cond_wait() or pthread_cond_timedwait() have associated with the condition variable during their waits; however, if predictable scheduling behavior is required, then that mutex is locked by the thread calling pthread_cond_signal() or pthread_cond_broadcast().

The pthread_cond_signal() and pthread_cond_broadcast() functions have no effect if there are no threads currently blocked on _c_o_n_d.

RETURN VALUES

If successful, the pthread_cond_signal() and pthread_cond_broadcast() functions return 0. Otherwise, an error number is returned to indicate the error.

ERRORS

The pthread_cond_signal() and pthread_cond_broadcast() function may fail if:

EINVAL

The value _c_o_n_d does not refer to an initialized condition variable.

NAME

pthread_cond_wait, pthread_cond_timedwait - wait on a condition

SYNOPSIS

```
cc -mt [ _f_l_a_g... ] _f_ i_l_e...- lpthread [ _l_ i_b_r_a_r_y... ]
```

```
#include <pthread.h>
```

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

```
int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime);
```

DESCRIPTION

The pthread_cond_wait() and pthread_cond_timedwait() functions are used to block on a condition variable. They are called with _m_u_t_e_x locked by the calling thread or undefined

behaviour will result.

These functions atomically release `_m_u_t_e_x` and cause the calling thread to block on the condition variable `_c_o_n_d`; atomically here means "atomically with respect to access by another thread to the mutex and then the condition variable". That is, if another thread is able to acquire the mutex after the about-to-block thread has released it, then a subsequent call to `pthread_cond_signal()` or `pthread_cond_broadcast()` in that thread behaves as if it were issued after the about-to-block thread has blocked.

Upon successful return, the mutex has been locked and is owned by the calling thread.

When using condition variables there is always a boolean predicate, an invariant, associated with each condition wait that must be true before the thread should proceed. Spurious wakeups from the `pthread_cond_wait()` or `pthread_cond_timedwait()` functions may occur. Since the return from `pthread_cond_wait()` or `pthread_cond_timedwait()` does not imply anything about the value of this predicate, the predicate should always be re-evaluated.

The order in which blocked threads are awakened by `pthread_cond_signal()` or `pthread_cond_broadcast()` is determined by the scheduling policy. See `pthread(3THR)`.

The effect of using more than one mutex for concurrent `pthread_cond_wait()` or `pthread_cond_timedwait()` operations on the same condition variable will result in undefined behavior.

A condition wait (whether timed or not) is a cancellation point. When the cancelability enable state of a thread is set to `PTHREAD_CANCEL_DEFERRED`, a side effect of acting upon a cancellation request while in a condition wait is that the mutex is re-acquired before calling the first cancellation cleanup handler.

A thread that has been unblocked because it has been canceled while blocked in a call to `pthread_cond_wait()` or `pthread_cond_timedwait()` does not consume any condition signal that may be directed concurrently at the condition variable if there are other threads blocked on the condition variable.

The `pthread_cond_timedwait()` function is the same as `pthread_cond_wait()` except that an error is returned if the absolute time specified by `_a_b_s_t_i_m_e` passes (that is, system time equals or exceeds `_a_b_s_t_i_m_e`) before the condition `_c_o_n_d` is signaled or broadcasted, or if the absolute time specified by `_a_b_s_t_i_m_e` has already been passed at the time of the call. When such time-outs occur, `pthread_cond_timedwait()` will nonetheless release and reacquire the mutex referenced by `_m_u_t_e_x`. The function `pthread_cond_timedwait()` is also a can-

cellation point.

If a signal is delivered to a thread waiting for a condition variable, upon return from the signal handler the thread resumes waiting for the condition variable as if it was not interrupted, or it returns 0 due to spurious wakeup.

RETURN VALUES

Except in the case of ETIMEDOUT, all these error checks act as if they were performed immediately at the beginning of processing for the function and cause an error return, in effect, prior to modifying the state of the mutex specified by `_m_u_t_e_x` or the condition variable specified by `_c_o_n_d`.

Upon successful completion, a value of 0 is returned. Otherwise, an error number is returned to indicate the error.

ERRORS

The `pthread_cond_timedwait()` function will fail if:

ETIMEDOUT

The time specified by `_a_b_s_t_i_m_e` to `pthread_cond_timedwait()` has passed.

The `pthread_cond_wait()` and `pthread_cond_timedwait()` functions may fail if:

EINVAL

The value specified by `_c_o_n_d`, `_m_u_t_e_x`, or `_a_b_s_t_i_m_e` is invalid.

EINVAL

Different mutexes were supplied for concurrent `pthread_cond_wait()` or `pthread_cond_timedwait()` operations on the same condition variable.

EINVAL

The mutex was not owned by the current thread at the time of the call.