# Multithreading Extension Introduction

The Multithreading extension gives programmers a notation for expressing parallelism in their applications. One or more sibling threads can be created within and sharing the process address space; the process model is extended so that inter-process interactions work as specified elsewhere in the SVID.

This section also includes additional of C library interfaces so that sibling threads do not interfere with each others' data, and synchronization interfaces for coordination of actions between threads within a process as well as threads in different processes.

While multithreading adapts naturally to multiprocessing computer systems, multithreading does not require multiprocessing capabilities. Multithreading used as a notation can simplify programs on uniprocessors and provide the potential for program speedup on; either uniprocessors or multiprocessors.

## Error Handling

Most new functions in section MT_LIB return an error indication instead of setting **`errno`** as a side effect of a function failure. This improves efficiency by avoiding the use of non-shared memory and simplifies the programming model, since the failure / success indication is returned as the function value where relevant.

While **`errno`** continues to be used for pre-existing interfaces, new interfaces need not pay the penalty of determining which thread's error indication location should be updated.

Some functions do not return an error indication because they always succeed (for example, **`thr_self) or because they cannot return (for example, thr_exit).`**

## Summary of Library Routines

The following library routines are specified by the MT_OS Multithreading OS Services.

**fork1** **forkall**

The following utilities are provided by the MT_LIB Multithreading Library Routines.

| | | | | |
|---|---|---|---|---|
| **asctime_r** | **getc_unlocked** | **rand_r** | **rwlock_init** | **thr_getspecific** |
| **barrier** | **getchar_unlocked** | **readir_r** | **sema_destroy** | **thr_join** |
| **barrier_destroy** | **getlogin_r** | **rmutex_destroy** | **sema_init** | **thr_keycreate** |
| **barrier_init** | **getpass_r** | **rmutex_init** | **sema_post** | **thr_keydelete** |
| **barrier_wait** | **gmtime_r** | **rmutex_lock** | **sema_trywait** | **thr_kill** |
| **cond_broadcast** | **l64a_r** | **rmutex_trylock** | **sema_wait** | **thr_minstack** |
| **cond_destroy** | **localtime_r** | **rmutex_unlock** | **semaphore** | **thr_self** |
| **cond_init** | **mutex** | **rw_lock** | **strtok_r** | **thr_setconcurrency** |
| **cond_signal** | **mutex_destroy** | **rw_rdlock** | **thr_continue** | **thr_setprio** |
| **cond_timedwait** | **mutex_init** | **rw_trylock** | **thr_create** | **thr_setscheduler** |
| **cond_wait** | **mutex_lock** | **rw_tryrdlock** | **thr_exit** | **thr_setspecific** |
| **condition** | **mutex_trylock** | **rw_trywrlock** | **thr_get_rr_interval** | **thr_sigsetmask** |
| **ctime_r** | **mutex_unlock** | **rw_unlock** | **thr_getconcurrency** | **thr_suspend** |
| **flockfile** | **putc_unlocked** | **rw_wrlock** | **thr_getprio** | **thr_yield** |
| **ftrylockfile** | **putchar_unlocked** | **rwlock_destroy** | **thr_getscheduler** | **ttyname_r** |
| **funlockfile** | | | | |

## Additions to Other Extensions

The following interfaces are in section BA_OS to provide more complete support for multithreaded and multiprocessing applications.

| | |
|---|---|
| pread | BA_OS |
| pwrite | BA_OS |
| sigwait | BA_OS |
| sysconf | BA_OS |

In addition, the environment variable, **PARALLEL** is included in **errno(BA_ENV)**.

## Changes to Existing Interfaces

The specification of the following commands and utilities have been modified to function correctly when the **`MT_`** extension is installed. **`nice(KE_OS)`** has been deprecated in favor of **`priocntl(KE_OS)`**.

| | | |
|---|---|---|
| auditrpt(AT_CMD) | priocntl(RT_CMD->AU_CMD) | semop(KE_OS) |
| exec(BA_OS) | priocntl(RT_OS->KE_OS) | sigaction(BA_OS) |
| fork(BA_OS) | ps(BU_CMD) | sigaltstack(BA_OS) |
| make(SD_CMD) | sem(KE_ENV) | sigsend(BA_OS) |
| nice(KE_OS) | semctl(KE_OS) | sigsendset(sigsend(BA_OS) |
| plock(KE_OS) | semget(KE_OS) | |

# Threads Overview

The threads interfaces allow a programmer to conveniently express parallel algorithms. In addition, user-level synchronization routines are provided that allow coordination of threads either within a process or across processes.

## What is a Thread?

A process as defined in the SVID consists of an address space and a single thread of execution. The MT_LIB interfaces extend that model to permit multiple threads within a single shared address space.

A thread is a sequence of instructions and associated data that is scheduled and executed as an independent entity. Every UNIX process that uses the threads interfaces contains at least one, and possibly many, threads.

The existence of a thread is strictly tied to the process in which they were created. Threads have no identity outside of that process.

Threads may be implemented in several different ways. One is to support threads in a library by multiplexing threads onto operating system-supported objects; another is to directly implement threads routines as operating system calls. The interfaces defined here do not require either approach, and may be implemented in either manner. The interfaces and discussion below presume that a flexible multiplexing approach is used, but do not require such an approach.

In the following discussion, the operating system supported objects will be called *Lightweight Processes*, or LWPs. LWPs are not defined by, or directly manipulated by, any interface in this section.

## Bound and Multiplexed Threads

If multiplexing is supported by the implementation, consider the actions of an LWP supporting a multithreaded process. The LWP can be considered to pick up a thread for execution, run it for a time, and then set down the thread and select another for execution.

An LWP has additional semantics as a result of operating system support. For example, an LWP is scheduled by the operating system, hence it competes against all other LWPs in the system for processor access and other resources.

A thread can be bound to an LWP for the thread's lifetime [see **thr_create**(MT_LIB)]. Bound threads have the properties of the underlying LWP, hence a bound thread is scheduled by the operating system.

Multiplexed threads, on the other hand, temporarily take on the properties of the underlying LWP, but so do any other multiplexed threads run by that LWP. The LWPs are scheduled by the operating system, but act as a homogeneous pool of processing power available in turn to any multiplexed thread. A multiplexed threads implementation schedules multiplexed threads (see the ''Thread Scheduling'' section). The initial thread is always multiplexed if multiplexing is supported at all.

The size of the pool of available LWPs reflects the actual concurrency level for multiplexed threads: if there are twenty threads, but only three are executing at a time, the concurrency level is three. If there are one hundred threads, but only three are executing at a time, the concurrency level is also three. Users can change the requested level of concurrency with the **THR_INCR_CONC** flag to **thr_create**(MT_LIB) or with **thr_setconcurrency**(MT_LIB), and determine the requested level with **thr_getconcurrency**.

When a program is linked with the Threads Library, an initial thread is created to execute the **main** function. This initial thread is a multiplexed thread.

In certain cases, such as when competing for synchronization objects bound threads are given scheduling priority over multiplexed threads to make better use of system resources.

## Thread Creation

**thr_create**(MT_LIB) creates new threads. The created thread may be either multiplexed or bound threads. The caller can supply a stack for the thread to run on, or the implementation will supply one. The implementation may not check for stack overflow for stacks supplied by the user, but a **SIGSEGV** signal may be generated if a thread overflows a library-allocated stack.

Every thread has an ID, which is recognized only within the current process. **thr_self**(MT_LIB) returns the ID of the calling thread.

## Sibling Threads

Threads within a process are siblings. Unlike processes, where a parent process creates a child process for which it must **wait**(BA_OS), threads create siblings for which they do not have to wait. The implementation of a sibling thread may be awaited with **thr_join**(MT_LIB) (see below), but this is optional.

## Detached Threads

A detached thread may be created [see **thr_create**(MT_LIB)], but a detached thread cannot be joined. [See **thr_join**(MT_LIB)]. On completion a detached thread's resources may be immediately recycled by the implementation. The **thr_exit** parameter for a detached thread is ignored.

## Daemon Threads

An application may create daemon threads (also known as detached threads) to provide ongoing services, for example asynchronous I/O, for other threads. Daemon threads do not need to exit explicitly; when the last non-daemon thread terminates, the process will exit, terminating any daemon threads.

## Thread Exit and Process Exit

**thr_exit**(MT_LIB) causes the calling thread to terminate its execution.

A process containing threads will terminate in any of the following circumstances:

- When the last non-daemon thread terminates, the process exits.
- If any thread calls **exit**(BA_OS) directly, the process and all its threads and LWPs will exit immediately.
- If the initial thread terminates without calling **thr_exit**, **exit** will be called implicitly, causing the entire process to exit.
- If the thread receives a signal whose effect is to terminate the process, the process will exit.
- If the initial thread returns from the initial call to **main**, the process exits as if **exit**(BA_OS) were called directly, with the value received by **main** as the **exit** statement.

## Joining or Waiting for a Thread

A thread uses **thr_join**(MT_LIB) to wait for another thread to exit and to retrieve its exit value. The term *join* emphasizes the sibling relationship between threads. When one thread waits for another, in effect they join control paths. Threads are joinable by default, but if they are created with the **THR_DETACHED** flag [see **thr_create**(MT_LIB)], they cannot be joined.

## Thread Scheduling

A thread may have one of three scheduling policies:

- time-sharing (**SCHED_TS** or **SCHED_OTHER**)
- round-robin (**SCHED_RR**)
- first-in-first-out (**SCHED_FIFO**)

Multiplexed threads must be able to run under the time-sharing policy. Bound threads must be able to run under all of these policies. See **thr_setscheduler**(MT_LIB) and **thr_setprio**(MT_LIB). The above three scheduling policies must be supported. Other, implementation-defined, scheduling policies may be available.

A thread can set its scheduling policy and priority with **thr_setscheduler**(MT_LIB) and its priority only with **thr_setprio**(MT_LIB). It can retrieve its scheduling policy and priority with **thr_getscheduler**(MT_LIB) and its priority only with **thr_getprio**(MT_LIB). **thr_yield**(MT_LIB) causes a thread to stop executing to allow another eligible thread to run.

The interfaces defined here do not protect against priority inversion. That is, it is possible for a thread to be blocked waiting for a lower priority thread to release a resource.

## Error Handling

None of the Threads Library routines set **errno**; most return an error number if an error is encountered. This discourages use of **errno**, which is non-reentrant and inefficient in a multithreaded environment. The routines do do not guarantee to preserve **errno** across calls.

## Signal Handling

UNIX System signals were designed for inter-process communication. They have been enhanced to work with multithreaded programs, but their use here should be restricted. We recommend that only a limited number of threads within a process access and receive signals. These threads can convert the notification provided by signals into internal communication between threads.

Each thread in a process has its own signal mask, which is inherited from its creator thread. Threads can use **thr_sigsetmask**(MT_LIB) to modify their signal masks.

When a multithreaded process receives a signal, the signal is delivered to one thread interested in the signal. Threads express interest in a signal by calling **sigwait**(BA_OS) or by using **signal**(BA_OS), **sigset**(BA_OS) or **sigaction**(BA_OS) to establish a handler for a signal.

Threads use **thr_kill**(MT_LIB) to send a signal to a sibling thread. Communication within a process should use MT_LIB operations wherever possible; using signals and threads can make programs more complex.

## Thread-Specific Data

Thread-specific data routines provide a thread-safe alternative to static or external data. That is, they provide a way for threads to create and access private data that persist across function calls. The thread-specific data routines are: **thr_getspecific**(MT_LIB), **thr_keycreate**(MT_LIB), **thr_keydelete**(MT_LIB), and **thr_setspecific**(MT_LIB).

# Synchronization

The synchronization interfaces allow coordination of threads within a process as well as coordination of threads in different processes. The following synchronization mechanisms are specified:

- mutual exclusion locks (mutex locks)
- condition variables
- semaphores
- reader-writer locks
- barriers
- recursive mutual exclusion locks (rmutex locks)

Most of these mechanisms can be initialized to be of one of two types: **USYNC_THREAD** or **USYNC_PROCESS**. **USYNC_THREAD** mechanisms should be used only by threads within the current process, whether or not the synchronization objects are in shared memory. **USYNC_PROCESS** mechanisms can be used by threads in different processes.

Each of these mechanisms is described briefly below, and in more detail on individual manual pages.

In all cases, data is protected by convention; a thread not following the protocol of acquiring a lock/semaphore, modifying or using the resource, then releasing the lock/semaphore is not prevented from modifying the shared data.

## Error Handling

None of the user synchronization routines set **errno**; most return an error number if an error is encountered. This discourages use of **errno**, which is non-reentrant. The routines do not guarantee to preserve **errno** across calls.

## Mutual Exclusion Locks

Mutual exclusion locks, or mutexes, are a synchronization mechanism used to serialize the execution of threads. They are typically used to ensure that only one thread at a time is operating on a shared datum. When mutexes are locked before and unlocked after every access to shared data, the integrity of that data with respect to cooperating threads is assured. Note that mutexes protect data only when the convention of acquiring and releasing the mutex is faithfully followed before and after any access of the data.

See **mutex**(MT_LIB), **mutex_destroy**(MT_LIB), **mutex_init**(MT_LIB), **mutex_lock**(MT_LIB), **mutex_trylock**(MT_LIB), and **mutex_unlock**(MT_LIB).

Recursive mutex locks are variations of the mutex lock.

## Condition Variables

A condition variable is a synchronization mechanism used to communicate information between cooperating threads, making it possible for a thread to suspend its execution while waiting for an event or condition. For example, the consumer in a producer-consumer algorithm might need to wait for the producer by waiting for the condition **buffer_is_not_empty**.

See **condition**(MT_LIB), **cond_broadcast**(MT_LIB), **cond_destroy**(MT_LIB), **cond_init**(MT_LIB), **cond_signal**(MT_LIB), **cond_timedwait**(MT_LIB), and **cond_wait**(MT_LIB).

## Reader-Writer Locks

Reader-writer locks allow many threads to have simultaneous read-only access to data, while allowing only one thread to have write access at any time. They are typically used to protect data that is searched more often than it is changed.

See **rwlock**(MT_LIB), **rw_rdlock**(MT_LIB), **rw_tryrdlock**(MT_LIB), **rw_trywrlock**(MT_LIB), **rw_unlock**(MT_LIB), **rw_wrlock**(MT_LIB), **rwlock_destroy**(MT_LIB), and **rwlock_init**(MT_LIB).

## Semaphores

Conceptually, a semaphore is a non-negative integer count. Semaphores are typically used to coordinate access to resources. The semaphore count is initialized with **sema_init** to the number of free resources. Threads then atomically increment the count with **sema_post** when resources are released and atomically decrement the count with **sema_wait** when resources are acquired. When the semaphore count becomes zero, indicating that no more resources are present, threads trying to decrement the semaphore with **sema_wait** will block until the count becomes greater than zero.

See **semaphore**(MT_LIB), **sema_destroy**(MT_LIB), **sema_init**(MT_LIB), **sema_post**(MT_LIB), **sema_trywait**(MT_LIB), and **sema_wait**(MT_LIB).

## Barriers

Barriers provide a simple coordination mechanism for threads. Threads wait at a barrier until a specified number of threads have reached the barrier, then they all resume execution.

Threads waiting at a barrier are put to sleep, or blocked, until the specified number of threads have reached the barrier.

When a thread calls **barrier_wait** it is said to have reached the barrier.

See **barrier**(MT_LIB), **barrier_destroy**(MT_LIB), **barrier_init**(MT_LIB), and **barrier_wait**(MT_LIB).

## Recursive Mutex Locks

Recursive mutual exclusion locks, or rmutexes, are mutexes that can be locked recursively. That is, a thread that has locked an rmutex may lock it again without releasing it. The thread that has locked an rmutex is referred to as the owner of the rmutex. Only the owner of an rmutex may lock it again while the rmutex is locked; other threads are denied access as with ordinary mutexes. Each **rmutex_lock** or **rmutex_trylock** call must be matched by a corresponding **rmutex_unlock** before the rmutex is made available to threads other than the owner.

Note that rmutexes, like mutexes, protect data only when the convention of acquiring the rmutex is faithfully followed before any access of the data.

See **rmutex**(MT_LIB), **rmutex_destroy**(MT_LIB), **rmutex_init**(MT_LIB), **rmutex_lock**(MT_LIB), **rmutex_trylock**(MT_LIB), and **rmutex_unlock**(MT_LIB).

# Multithreading Extension OS Service Routines

The following section contains the manual pages for the MT_OS routines.

**fork** – create a new process
**#include <sys/types.h>**
**#include <unistd.h>**

**pid_t fork(void);**
**fork** causes creation of a new process. The new process (child process) is an exact copy of the calling process (parent process). This means the child process inherits the following attributes from the parent process: real user ID, real group ID, effective user ID, effective group ID environment close-on-exec flag [see **exec**(BA_OS)] signal handling settings (that is, **SIG_DFL**, **SIG_IGN**, **SIG_HOLD**, function address) supplementary group IDs set-user-ID mode bit set-group-ID mode bit profiling on / off status nice value [see **nice**(AS_CMD)] scheduler class [see **priocntl**(RT_OS)] all attached shared memory segments process group **ID** session **ID** current working directory root directory file mode creation mask [see **umask**(BA_OS)] resource limits controlling terminal working and maximum privilege sets Scheduling priority and any per-process scheduling parameters that are specific to a given scheduling class may or may not be inherited according to the policy of that particular class [see **priocntl**(RT_OS)]. The child process differs from the parent process in the following ways: The child process has a unique process **ID** which does not match any active process group **ID**. The child process has a different parent process **ID** (that is, the process **ID** of the parent process). The child process has its own copy of the parent's file descriptors and directory streams. Each of the child's file descriptors shares a common file pointer with the corresponding file descriptor of the parent. All **semadj** values are cleared Process locks, text locks and data locks are not inherited by the child The child process's **tms** structure is cleared: **tms_utime**, **stime**, **cutime**, and **cstime** are set to 0 The time left until an alarm clock signal is reset to 0. The set of signals pending for the child process is initialized to the empty set. Record locks set by the parent process are not inherited by the child process [see **fcntl**(BA_OS)].

# Multithreading Extension Library Routines

The following section contains the manual pages for the MT_LIB routines.

**a64l**, **l64a** – convert between long integer and base-64 ASCII string

**#include <stdlib.h>**

**long a64l(const char** **s***);**

**char** ***&#8203;**l64a(long** *l***);**

These functions are used to maintain numbers stored in base-64 ASCII characters. These characters define a notation by which long integers can be represented by up to six characters; each character represents a ‘‘digit’’ in a radix-64 notation. The characters used to represent ‘‘digits’’ are **.** for 0, **/** for 1, **0** through **9** for 2–11, **A** through **Z** for 12–37, and **a** through **z** for 38–63. **a64l** takes a pointer to a null-terminated base-64 representation and returns a corresponding **long** value. If the string pointed to by **s** contains more than six characters, **a64l** will use the first six. **a64l** scans the character string from left to right with the least significant digit on the left, decoding each character as a 6-bit radix-64 number. **l64a** takes a **long** argument and returns a pointer to the corresponding base-64 representation. If the argument is 0, **l64a** returns a pointer to a null string. The value returned by **l64a** is a pointer into a static buffer, the contents of which are overwritten by each call. Level 1.

## NAME

**aio_cancel** – cancel asynchronous I/O operations

## SYNOPSIS

**#include <aio.h>**

**int aio_cancel(int** *fildes*, **struct aiocb** ***aiocbp*);**

## DESCRIPTION

**aio_cancel** allows the cancellation of outstanding asynchronous I/O requests. *fildes* and *aiocbp* are used to identify the asynchronous I/O requests that should be canceled.

The **aio_cancel** function attempts to cancel one or all of the asynchronous I/O requests currently outstanding against file descriptor *fildes*. The *aiocbp* argument points to the asynchronous I/O control block for a particular request to be canceled. If *aiocbp* is **NULL**, all outstanding cancelable asynchronous I/O requests against *fildes* are canceled.

Normal notification occurs for asynchronous I/O operations that are successfully canceled. If there are requests which cannot be canceled, then the normal asynchronous completion process shall take place for those requests when they are completed.

For requested operations that are successfully canceled, **aio_return** returns **-1** and its associated error status is set to **ECANCELED**. For requested operations that are not successfully canceled, the *aiocbp* is not modified by **aio_cancel**.

If *aiocbp* is not **NULL**, and *fildes* does not have the same value as the file descriptor with which the asynchronous operation was initiated, unspecified results occur.

### Return Values

The **aio_cancel** function returns the value **AIO_CANCELED** to the calling process if the requested operation(s) were canceled. The value **AIO_NOTCANCELED** is returned if at least one of the requested operation(s) cannot be canceled because it is in progress. In this case, the state of the other operations, if any, referenced in the call to **aio_cancel** is not indicated by the return value of **aio_cancel**. The application may determine the state of affairs for these operations by using **aio_error**. The value **AIO_ALLDONE** is returned if all of the operations have already completed. Otherwise, the function returns –1, and sets **errno** to indicate the error.

### Errors

Under the following conditions, **aio_cancel** fails and sets **errno** to:

| | |
|---|---|
| **EBADF** | *fildes* is not a valid file descriptor. |
| **EINVAL** | no asynchronous I/O has ever been requested for *fildes*. |
| **ENOSYS** | **aio_cancel** is not supported by this implementation. |

## SEE ALSO

**aiocb**(MT_LIB), **aio_read**(MT_LIB), **aio_suspend**(MT_LIB), **aio_write**(MT_LIB).

## LEVEL

Level 1.

## NAME

**aiocb** – Asynchronous I/O Control Block

## SYNOPSIS

**#include <aio.h>**

## DESCRIPTION

**aiocb** specifies the asynchronous I/O control block that is used by the asynchronous I/O interface routines.

The asynchronous I/O routines pass information with the request and receive completion status information after the I/O operation has completed. An asynchronous I/O control block, structure **aiocb**, is used to specify input parameters and receive completion status information for asynchronous I/O requests. This structure is defined in **aio.h** and includes at least the following members:

```
int             aio_fildes;      /* file descriptor          */
volatile void*  aio_buf;         /* buffer location          */
size_t          aio_nbytes;      /* length of transfer       */
off_t           aio_offset;      /* file offset              */
struct sigevent aio_sigevent;    /* signal number and offset */
int             aio_lio_opcode;  /* listio operation         */
```

The structure members **aio_fildes**, **aio_buf**, and **aio_nbytes** are the same as the *fildes*, *buf*, and *nbytes* arguments to **read** and **write**. With **aio_read**, for example, the caller wishes to read **aio_nbytes** from the file associated with **aio_fildes** into the buffer pointed to by **aio_buf**. All appropriate structure members should be set by the caller when **aio_read** or **aio_write** is called.

If **O_APPEND** is not set for the file descriptor **aio_fildes** and this file descriptor points to a device capable of seeking, the requested I/O operation occurs at the position in the file specified by **aio_offset**.

The **aio_sigevent** member defines the notification method to be used on I/O completion. This structure,found in siginfo.h, includes the following members:

```
int             sigev_notify       /* notification mode */
union {
        int     nisigno     /* signal number */
        void      (*nifunc)(union sigval)
} sigev_notifyinfo
union {
        int     sival_int   /* integer value */
        void    *sival_ptr  /* pointer value */
} sigev_value                      /* signal value      */
```

**aio_sigevent.sigev_notify** can be set to **SIGEV_NONE** or **SIGEV_CALLBACK**. If it is set to **SIGEV_NONE**, no notification is posted on I/O completion, but the error status for the operation and the return status for the operation shall be appropriately set. If it is set to **SIGEV_CALLBACK**, **nifunc** shall be set to the address of the function to be called. **sigev_value** shall be set to the value to be passed to the function call. **nisigno** specifies the signal to be generated.

**aio_lio_opcode** is used only by the **lio_listio** function which allows multiple asynchrounous I/O operations to be submitted at once. [See **lio_listio**(MT_LIB)**]**

To insure forward compatibility, all unused fields of the **aiocb** structure must be set to zero. This can be done by using **calloc** [see **malloc**(BA_OS)] to allocate the structure or by using **memset**(BA_LIB) before the structure is used.

## SEE ALSO

**aio_cancel**(MT_LIB), **aio_error**(MT_LIB), **aio_read**(MT_LIB), **aio_suspend**(MT_LIB), **aio_write**(MT_LIB), **lio_listio**(MT_LIB), **malloc**(BA_OS), **memset**(BA_LIB), **read**(BA_OS), **write**(BA_OS)

## LEVEL

Level 1.

## NAME

**aio_error** – retrieve asynchronous I/O error status

## SYNOPSIS

**#include <aio.h>**

**int aio_error(const struct aiocb ****aiocbp***);**

## DESCRIPTION

**aio_error** returns the error status associated with the *aiocb* structure referenced by the *aiocbp* argument. The error status for an asynchronous I/O operation is the **errno** value that would be set by the corresponding **read** or **write** operation. If the operation is not completed, the error status is set to **EINPROGRESS**.

### Return Values

If the asynchronous I/O operation completes successfully, 0 is returned. If it fails, the corresponding **read** or **write** error status is returned. **EINPROGRESS** is returned if the operation is still in progress.

## SEE ALSO

**aio_cancel**(MT_LIB), **aiocb**(MT_LIB), **aio_read**(MT_LIB), **aio_return**(MT_LIB), **aio_write**(MT_LIB), **read**(BA_OS), **write**(BA_OS)

## LEVEL

Level 1.

## NAME

**aio_read** – asynchronous read

## SYNOPSIS

```
#include <aio.h>

int aio_read(struct aiocb *aiocbp);
```

## DESCRIPTION

**aio_read** supports an asynchronous read capability that allows the calling process to read **aiocbp->aio_nbytes** from the file associated with the file descriptor **aiocbp->aio_fildes** into the buffer pointed to by **aiocbp->aio_buf**.

**aiocbp** points to an **aiocb** structure which contains other input parameters as well as completion status members.

If **aiocbp->aio_sigevent.sigev_notify** is set to **SIGEV_NONE** in the asynchronous I/O control block, no notification is posted on I/O completion. If it is set to **SIGEV_CALLBACK**, **aiocbp->aio_sigevent.sigev_notifyinfo.nifunc** is set to the address of the function to be called, **aiocbp->aio_sigevent.sigev_value** is set to the argument to be passed to the function, and callback notification is posted.

If the control block pointed to by *aiocbp* or the buffer pointed to by **aiocpb->aio_buf** becomes an invalid address prior to asynchronous I/O completion, then the behavior is undefined.

### Return Values

The call to **aio_read** returns 0 when the read request has been initiated or queued to the file or device. If an error condition is encountered during queuing, the call returns –1 without having initiated or queued the request.

Upon successful completion, the function **aio_read** returns 0 and a call to **aio_error** returns **EINPROGRESS**. If the read request fails, **aio_read** returns –1 and the **aio_error** function will return the error number of the failure.

After the read operation has successfully completed, the **aio_error** function returns 0. The **aio_return** function can be used to access the return value of the underlying **read** call.

### Errors

There are two types of errors that are associated with an asynchronous I/O request. The first occurs during the validity checking of the I/O request submitted by the **aio_read** routine. This error is returned to the caller of **aio_read**. The other occurs during the processing of the actual read operation. The read operation may fail for any of the reasons that a normal **read** fails. If the call to **aio_read** successfully queues the I/O operation but the operation is subsequently canceled or encounters an error, the **aio_error** function returns one of the values normally returned by the **read** system call or one of the errors listed below. The time at which the error occurs is implementation-dependent; for example, an invalid file descriptor may not be determined at the time of the call to **aio_read** which could successfully complete. However, **aio_error** then returns **EBADF** to indicate the error.

The notification specified for a request will only be performed if that request was successfully queued.

Under the following conditions, **aio_read** fails and the **aio_error** function returns:

| | |
|---|---|
| **EAGAIN** | The requested asynchronous I/O operation was not queued because of system resource limitations. The first request to encounter a resource limitation and all subsequent requests will be marked so that **aio_error** returns **EAGAIN**. |
| **ECANCELED** | The requested I/O was canceled before the I/O completed because of **aio_cancel**. |
| **EBADF** | The **aiocbp->aio_fildes** argument is not a valid file descriptor open for reading. |
| **EINVAL** | The value of **aiocbp->aio_offset** would be invalid. The notification mode specified by **aiocbp->aio_sigevent.sigev_notify** is not supported by the implementation. |

## REFERENCES

**aio_cancel**(MT_LIB), **aiocb**(MT_LIB), **aio_suspend**(MT_LIB), **aio_write**(MT_LIB), **read**(BA_OS)

## LEVEL

Level 1.

## NAME

**aio_return** – retrieve return status of asynchronous I/O operation

## SYNOPSIS

```
#include <aio.h>

int aio_return(struct aiocb *aiocbp);
```

## DESCRIPTION

**aio_return** returns the return status associated with the **aiocb** structure referenced by the **aiocbp** argument. The return status for an asynchronous I/O operation is the value that would be returned by the corresponding **read** or **write** function call. If the error status [see **aio_error**(MT_LIB)] for the operation is equal to **EINPROGRESS**, the return status for the operation is undefined. The **aio_return** function may be called exactly once to retrieve the return status of a given asynchronous operation; thereafter, if the same **aiocb** referenced structure is used in a call to **aio_return** or **aio_error**, the result is undefined. When the **aiocb** structure referred to by *aiocbp* is used to submit another asynchronous operation, **aio_return** may be successfully used to retrieve the return status for that operation.

### Return Values

The corresponding **read** or **write** status is returned. This function should be called only when **aio_error** does not return **EINPROGRESS**.

## SEE ALSO

**aio_cancel**(MT_LIB), **aiocb**(MT_LIB), **aio_error**(MT_LIB), **aio_read**(MT_LIB), **aio_write**(MT_LIB), **read**(BA_OS), **write**(BA_OS)

## LEVEL

Level 1.

## NAME

**aio_suspend** – suspend until asynchronous I/O completes

## SYNOPSIS

```
#include <aio.h>

int aio_suspend(const struct aiocb *list[],
                int nent, const struct timespec *timeout);
```

## DESCRIPTION

**aio_suspend** suspends the calling thread until at least one of the selected I/O requests completes or a timeout occurs. **aio_suspend** suspends the calling thread until at least one of the asynchronous I/O operations referenced by the *list* argument has completed, until a signal interrupts the function, or, if *timeout* is not **NULL**, until the time interval specified by *timeout* has passed. If any of the **aiocb** structures in the list corresponds to completed asynchronous I/O operations (i.e., the error status for the operation is not equal to **EINPROGRESS**), at the time of the call, the function returns without suspending the calling process. The *list* argument is an array of pointers to asynchronous I/O control blocks. The *nent* argument indicates the number of elements in the array. Each **aiocb** structure pointed to is used in initiating an asynchronous I/O request via **aio_read**, **aio_write**, or **lio_listio**. This array may contain **NULL** pointers which are ignored. If this array contains pointers that refer to **aiocb** structures which have not been used in submitting asynchronous I/O, the effect is undefined.

The **timespec** structure includes at least the following members:

```
time_t  tv_sec      /* seconds     */
long         tv_nsec     /* nanoseconds */
```

If the time interval indicated in the **timespec** structure pointed to by *timeout* passes before any of the I/O operations referenced by *list* is completed, **aio_suspend** returns with an error.

### Return Values

If the **aio_suspend** function returns after one or more asynchronous I/O operations have completed, the function returns zero. Otherwise, the function returns a value of -1 and sets **errno** to indicate the error.

The application determines which asynchronous I/O completed by scanning the associated error and return status, using **aio_error** and **aio_return,** respectively.

### Errors

Under the following conditions, the **aio_suspend** function returns –1 and sets **errno** to:

| | |
|---|---|
| **EAGAIN** | No asynchronous I/O indicated in the list referenced by *list* completed in the time interval indicated by *timeout*. |
| **EINTR** | A signal interrupted the **aio_suspend** function. |
| **ENOSYS** | **aio_suspend** is not supported by this implementation. |

## SEE ALSO

**aiocb**(MT_LIB), **aio_cancel**(MT_LIB), **aio_error**(MT_LIB), **aio_read**(MT_LIB), **aio_return**(MT_LIB), **aio_write**(MT_LIB), **lio_listio**(MT_LIB)

**LEVEL**

Level 1.

## NAME

**aio_write** – asynchronous write

## SYNOPSIS

**#include <aio.h>**

**int aio_write(struct aiocb ***_aiocbp_**);**

## DESCRIPTION

**aio_write** supports an asynchronous write capability that allows the calling process to write **aiocbp->aio_nbytes** to the file associated with file descriptor **aiocbp->aio_fildes** from the buffer pointed to by **aiocbp->aio_buf**.

**aiocbp** points to an **aiocb** structure which contains other input parameters as well as completion status members.

If **aiocbp->aio_sigevent.sigev_notify** is set to **SIGEV_NONE** in the asynchronous I/O control block, no notification is posted on I/O completion. If it is set to **SIGEV_CALLBACK**, **aiocbp->aio_sigevent.sigev_notifyinfo.nifunc** is set to the address of the function to be called, **aiocbp->aio_sigevent.sigev_value** is set to the argument to be passed to the function, and callback notification is posted.

If the control block pointed to by **aiocbp** or the buffer pointed to by **aiocbp->aio_buf** becomes an invalid address prior to asynchronous I/O completion, then the behavior is undefined.

### Return Values

The call to **aio_write** returns 0 when the write request has been initiated or queued to the file or device. If an error condition is encountered during queuing, the call returns –1 without having initiated or queued the request.

Upon successful completion, the function **aio_write** returns 0 and a call to **aio_error** returns **EINPROGRESS**. If the write request fails, **aio_write** returns –1 and the **aio_error** function will return the error number of the failure

After the write operation has successfully completed, the **aio_error** function returns 0. The **aio_return** function can be used to access the return value of the underlying **write** call.

### Errors

There are two types of errors that are associated with an asynchronous I/O request. The first occurs during the validity checking of the I/O request submitted by the **aio_write** routine. This error is returned to the caller of **aio_write**. The other occurs during the processing of the actual write operation. The write operation may fail for any of the reasons that a normal **write** fails. If the call to **aio_write** successfully queues the I/O operation but the operation is subsequently canceled or encounters an error, the **aio_error** function will return one of the values normally returned by the **write** system call or one of the errors listed below. The time at which the error occurs is implementation-dependent; for example, an invalid file descriptor may not be determined at the time of the call to **aio_write** which could successfully complete. However, **aio_error** will return **EBADF** to indicate the error.

The notification specified for a request will only be performed if that request was successfully queued.

Under the following conditions, **aio_write** fails and the **aio_error** function returns:

| | |
|---|---|
| **EAGAIN** | The requested asynchronous I/O operation was not queued because of system resource limitations. The first request to encounter a resource limitation and all subsequent requests will be marked so that **aio_error** returns **EAGAIN**. |
| **ECANCELED** | The requested I/O was canceled before the I/O completed due to **aio_cancel**. |
| **EBADF** | The **aiocbp->aio_fildes** argument is not a valid file descriptor open for writing. |
| **EINVAL** | The value of **aiocbp->aio_offset** would be invalid. The notification mode specified by **aiocbp->aio_sigevent.sigev_notify** is not supported by the implementation. |

## SEE ALSO

**aio_cancel**(MT_LIB), **aiocb**(MT_LIB), **aio_read**(MT_LIB), **aio_suspend**(MT_LIB), **write**(BA_OS)

## LEVEL

Level 1.

## NAME

**barrier_destroy** – destroy a blocking barrier

## SYNOPSIS

**#include <synch.h>**

**int barrier_destroy(barrier_t ***barrier**);**

### Parameters

| | |
|---|---|
| *barrier* | pointer to barrier to be destroyed |

## DESCRIPTION

**barrier_destroy** destroys the barrier pointed to by *barrier*. This includes invalidating the barrier and freeing any associated implementation-allocated dynamic resources.

Any user-allocated dynamic storage is unaffected by **barrier_destroy** and must be explicitly released by the program.

### Return Values

**barrier_destroy** returns zero for success and an error number for failure, as described below.

### Errors

If one of the following conditions is detected, **barrier_destroy** returns the corresponding value:

| | |
|---|---|
| **EBUSY** | A thread is still waiting at the barrier. |
| **EINVAL** | Invalid argument specified. |

## SEE ALSO

**barrier**(MT_LIB), **barrier_init**(MT_LIB), **barrier_wait**(MT_LIB)

## LEVEL

Level 1

## NAME

**barrier_init** – initialize a blocking barrier

## SYNOPSIS

```
#include <synch.h>

int barrier_init(barrier_t *barrier, int count, int type, void *arg);
```

### Parameters

| | |
|---|---|
| *barrier* | pointer to barrier to be initialized |
| *count* | number of threads to use the barrier for synchronization |
| *type* | **USYNC_THREAD** or **USYNC_PROCESS** |
| *arg* | **NULL** (reserved for future use) |

## DESCRIPTION

**barrier_init** initializes the barrier pointed to by *barrier* to be of type *type* and to synchronize *count* threads. Once initialized, the barrier can be used any number of times to synchronize execution of *count* threads.

Threads waiting at a barrier will block, or sleep, until all *count* threads arrive at the barrier.

### barrier Parameter

*barrier* points to the barrier to be initialized.

### count Parameter

*count* is the number of threads that will be synchronized by the barrier. That is, **barrier_wait** will block any calling threads until *count* threads have reached the barrier.

### type Parameter

*type* can be set to one of the following values:

| | |
|---|---|
| **USYNC_THREAD** | Initialize the barrier for threads within the current process. |
| **USYNC_PROCESS** | Initialize the barrier for threads across processes. |

### arg Parameter

*arg* should be set to **NULL**. It is not currently used, but is reserved for future use.

### Return Values

**barrier_init** returns zero for success and an error number for failure, as described below.

### Errors

If one of the following conditions is detected, **barrier_init** returns the corresponding value:

| | |
|---|---|
| **EBUSY** | A thread is waiting at the barrier pointed to by *barrier*. |
| **EINVAL** | Invalid argument specified. |

## USAGE

### Warnings

A barrier should not be re-initialized while threads are waiting at the barrier.

**SEE ALSO**

**barrier**(MT_LIB), **barrier_destroy**(MT_LIB), **barrier_wait**(MT_LIB)

**LEVEL**

Level 1

## NAME

**barrier_wait** – wait at a blocking barrier

## SYNOPSIS

**#include <synch.h>**

**int barrier_wait(barrier_t** ***barrier*****);**

### Parameters

*barrier* pointer to barrier at which to wait

## DESCRIPTION

**barrier_wait** blocks the calling thread at a barrier until *count* threads have called it. *count* is defined during initialization with **barrier_init**. A thread is said to have reached the barrier when it calls **barrier_wait**.

When the last thread reaches the barrier, all *count –1* blocked threads are released from the barrier and are allowed to resume execution. The barrier is reset after the waiting threads are released.

*barrier* must previously have been initialized (see **barrier_init**).

From the point of view of the caller, **barrier_wait** is atomic: even if interrupted by a signal or **forkall**(MT_OS), **barrier_wait** will not return until *count* threads have reached the barrier.

The order in which threads are released from the barrier is scheduling policy specific for bound threads, and may depend on scheduling parameters for multiplexed threads.

### Return Values

**barrier_wait** returns zero for success and an error number for failure, as described below.

### Errors

If the following condition is detected, **barrier_wait** returns the value:

**EINVAL** Invalid argument specified.

## SEE ALSO

**barrier**(MT_LIB), **barrier_destroy**(MT_LIB), **barrier_init**(MT_LIB),

**forkall**(MT_OS)

## LEVEL

Level 1

## NAME

**cond_broadcast** – broadcast a wake up to all threads waiting on a condition variable

## SYNOPSIS

**#include <synch.h>**

**int cond_broadcast(cond_t** **cond***);**

### Parameters

| | |
|---|---|
| *cond* | pointer to condition variable to be broadcast |

## DESCRIPTION

**cond_broadcast** wakes up all threads waiting on the condition *cond*. If more than one thread is waiting, the order of release from the blocked group is scheduling policy-specific for / bound threads, and may depend on scheduling parameters for multiplexed threads.

**cond_broadcast** has no effect if there are no threads waiting on *cond*.

A **cond_broadcast** will be more reliable if the associated *mutex* used by waiters is held across the call.

### cond Parameter

The condition variable denoted by *cond* must previously have been initialized (see **cond_init**).

### Return Values

**cond_broadcast** returns zero for success and an error number for failure, as described below.

### Errors

If the following condition is detected, **cond_broadcast** returns the value:

| | |
|---|---|
| **EINVAL** | Invalid argument specified. |

## USAGE

See the description of how to use condition variables under USAGE on **cond_init**(MT_LIB).

## SEE ALSO

**condition**(MT_LIB), **cond_destroy**(MT_LIB), **cond_init**(MT_LIB), **cond_signal**(MT_LIB), **cond_timedwait**(MT_LIB), **cond_wait**(MT_LIB)

## LEVEL

Level 1

## NAME

**cond_destroy** – destroy a condition variable

## SYNOPSIS

**#include <synch.h>**

**int cond_destroy(cond_t** **cond***);**

### Parameters

| | |
|---|---|
| *cond* | pointer to the condition variable to destroy |

## DESCRIPTION

**cond_destroy** destroys the condition variable *cond*. This includes invalidating *cond* and freeing any associated implementation-allocated dynamic resources.

### Return Values

**cond_destroy** returns zero for success and an error number for failure, as described below.

### Errors

If the following condition is detected, **cond_destroy** returns the value:

| | |
|---|---|
| **EBUSY** | *cond* still has other threads waiting on it. |
| **EINVAL** | Invalid argument specified. |

## SEE ALSO

**condition**(MT_LIB), **cond_broadcast**(MT_LIB), **cond_init**(MT_LIB), **cond_signal**(MT_LIB), **cond_timedwait**(MT_LIB), **cond_wait**(MT_LIB)

## LEVEL

Level 1

## NAME

**cond_init** – initialize a condition variable

## SYNOPSIS

**#include <synch.h>**

**int cond_init(cond_t** **cond*, **int** *type*, **void** **arg***);**

### Parameters

| | |
|---|---|
| *cond* | pointer to condition variable to be initialized |
| *type* | **USYNC_THREAD** or **USYNC_PROCESS** |
| *arg* | **NULL** (reserved for future use) |

## DESCRIPTION

**cond_init** initializes the condition variable pointed to by *cond* to be of type *type*. Once created, the condition *cond* can be used any number of times without being re-initialized.

### cond Parameter

*cond* points to the condition variable to be initialized.

### type Parameter

*type* can be set to one of the following values:

| | |
|---|---|
| **USYNC_THREAD** | Initialize the condition variable for threads within the current process. |
| **USYNC_PROCESS** | Initialize the condition variable for threads across processes. |

### arg Parameter

*arg* should be set to **NULL**. It is not currently used, but is reserved for future use.

### Return Values

**cond_init** returns zero for success and an error number for failure, as described below.

### Errors

If any of the following conditions is detected, **cond_init** returns the corresponding value:

| | |
|---|---|
| **EBUSY** | *cond* is referenced by other threads. |
| **EINVAL** | Invalid argument specified. |

## USAGE

### Warnings

**cond_init** does not examine the *cond* argument before initializing it. If **cond_init** is called more than once for the same condition, it will overwrite its state. It is the user’s responsibility to ensure that **cond_init** is only called once for each condition variable. **NULL** should be passed as the value of *arg* to ensure correct operation.

## SEE ALSO

**condition**(MT_LIB), **cond_broadcast**(MT_LIB), **cond_destroy**(MT_LIB), **cond_signal**(MT_LIB), **cond_timedwait**(MT_LIB), **cond_wait**(MT_LIB)

**LEVEL**

Level 1

## NAME

**cond_signal** – wake up a single thread waiting on a condition variable

## SYNOPSIS

**#include <synch.h>**

**int cond_signal(cond_t** **cond***);**

### Parameters

| | |
|---|---|
| *cond* | pointer to condition variable to be signaled |

## DESCRIPTION

**cond_signal** wakes up a single thread, if one exists, waiting on the condition *cond*. If more than one thread is waiting, the choice of which to release from the blocked group is scheduling policy-specific for bound threads, and may be dependent on scheduling parameters for multiplexed threads.

**cond_signal** has no effect if there are no threads waiting on *cond*.

A **cond_signal** will be more reliable if the associated *mutex* used by waiters is held across the call.

### cond Parameter

The condition variable denoted by *cond* must previously have been initialized (see **cond_init**).

### Return Values

**cond_signal** returns zero for success and an error number for failure, as described below.

### Errors

If the following condition is detected, **cond_signal** returns the corresponding value:

| | |
|---|---|
| **EINVAL** | Invalid argument specified. |

## USAGE

See the description of how to use condition variables under USAGE on **cond_init**(MT_LIB).

## SEE ALSO

**condition**(MT_LIB), **cond_broadcast**(MT_LIB), **cond_destroy**(MT_LIB), **cond_init**(MT_LIB), **cond_timedwait**(MT_LIB), **cond_wait**(MT_LIB)

## LEVEL

Level 1

## NAME

**cond_timedwait** – wait on a condition variable for a limited time

## SYNOPSIS

```
#include <synch.h>
#include <sys/time.h>

int cond_timedwait(cond_t *cond, mutex_t *mutex, timestruc_t *abstime);
```

### Parameters

| | |
|---|---|
| *cond* | pointer to the condition variable to wait for |
| *mutex* | pointer to a locked mutex |
| *abstime* | absolute time to at which to time out |

## DESCRIPTION

**cond_timedwait**, similar to **cond_wait**, blocks the calling thread at the condition variable pointed to by *cond*, to wait for the occurrence of a condition. However, if the absolute time denoted by *abstime* has passed and the indicated condition is not signaled, **cond_timedwait** returns **ETIME** to the caller. The calling thread must lock the mutual exclusion lock (mutex) pointed to by *mutex* before calling **cond_timedwait**, otherwise the behavior is unpredictable.

**cond_timedwait** automatically releases the mutex, and waits on the condition variable *cond*. When the condition is signaled or the time expires, **cond_timedwait** reacquires the *mutex* and returns to the caller. The wait can also be interrupted by a UNIX system signal, in which case *mutex* is reacquired, the signal handler is called, and **cond_timedwait** returns **EINTR**.

User-visible timers are not affected by a call to **cond_timedwait**.

The calling thread can resume execution when the condition is signaled or broadcast, a timeout occurs, or when interrupted. The logical condition should be checked on return, as a return may not have been caused by a change in the condition.

### cond Parameter

The condition variable denoted by *cond* must previously have been initialized (see **cond_init**).

### mutex Parameter

*mutex* is a mutual exclusion variable protecting a shared resource associated with the condition represented by the condition variable, *cond*. The calling thread must lock *mutex* before calling **cond_wait**, otherwise the behavior is unpredictable.

### abstime Parameter

*abstime* represents the time at which **cond_timedwait** should time out. The time is expressed in elapsed seconds and nanoseconds since Universal Coordinated Time, January 1, 1970. **gettimeofday**(RT_OS) returns the current time, but in seconds and microseconds. To construct *abstime*, convert the current time to a **timestruc_t**, and add to that the waiting time.

### Return Values

**cond_timedwait** returns zero for success and an error number for failure, as described below.

### Errors

If any of the following conditions is detected, **cond_timedwait** returns the corresponding value:

| | |
|---|---|
| **EINTR** | The wait was interrupted by a UNIX system signal. |
| **EINVAL** | Invalid argument specified. |
| **EINVAL** | *abstime* is NULL. |
| **ETIME** | Time specified by *abstime* has passed. |

## USAGE

See the description of how to use condition variables under USAGE on **cond_init**(MT_LIB).

Because the condition can change between the time the condition is signaled and the **mutex** is re-locked, the calling thread must always re-check the condition upon return from **cond_timedwait**.

### Warnings

Condition variables are not asynchronous-safe, and should not be used to communicate between signal handlers and base level user code. Semaphores provide asynchronous-safe communication for such cases; see **semaphore**(MT_LIB).

The behavior is undefined if *mutex* is destroyed or deallocated while the thread is waiting on *cond*.

## SEE ALSO

**condition**(MT_LIB), **cond_broadcast**(MT_LIB), **cond_destroy**(MT_LIB), **cond_init**(MT_LIB), **cond_signal**(MT_LIB), **cond_wait**(MT_LIB),

**gettimeofday**(RT_OS)

## LEVEL

Level 1

## NAME

**cond_wait** – wait on a condition variable

## SYNOPSIS

**#include <synch.h>**

**int cond_wait(cond_t** **cond*, **mutex_t** **mutex*);

### Parameters

| | |
|---|---|
| *cond* | pointer to the condition variable to wait for |
| *mutex* | pointer to a locked mutex |

## DESCRIPTION

**cond_wait** blocks the calling thread at the condition variable pointed to by *cond* to wait for the occurrence of a condition. The calling thread must lock the mutual exclusion lock (mutex) pointed to by *mutex* before calling **cond_wait**, otherwise the behavior is unpredictable.

**cond_wait** automatically releases the mutex, and waits on the condition variable *cond*. When the condition is signaled **cond_wait** reacquires the *mutex* and returns to the caller. The wait can also be interrupted by a UNIX system signal, in which case *mutex* is reacquired, the signal handler is called, and **cond_wait** returns **EINTR**.

The calling thread can resume execution when the condition is signaled or broadcast, or when interrupted. The logical condition should be checked on return, as a return may not have been caused by a change in the condition.

### cond Parameter

The condition variable denoted by *cond* must previously have been initialized (see **cond_init**).

### mutex Parameter

*mutex* is a mutual exclusion variable protecting a shared resource associated with the condition represented by the condition variable, *cond*. The calling thread must lock *mutex* before calling **cond_wait**, otherwise the behavior is unpredictable.

### Return Values

**cond_wait** returns zero for success and an error number for failure, as described below.

### Errors

If any of the following conditions is detected, **cond_wait** fails and returns the corresponding value:

| | |
|---|---|
| **EINTR** | The wait was interrupted by a UNIX system signal. |
| **EINVAL** | Invalid argument specified. |

## USAGE

See the description of how to use condition variables under USAGE on **cond_init**(MT_LIB).

Because the condition can change between the time the condition is signaled and the mutex is re-locked, the calling thread must always re-check the condition upon return from **cond_wait**.

### Warnings

Condition variables are not asynchronous-safe, and should not be used to communicate between signal handlers and base level user code. Semaphores provide asynchronous-safe communication for such cases; see **semaphore**(MT_LIB).

The behavior is undefined if *mutex* is destroyed or deallocated while the thread is waiting on *cond*.

## SEE ALSO

**condition**(MT_LIB), **cond_broadcast**(MT_LIB), **cond_destroy**(MT_LIB), **cond_init**(MT_LIB), **cond_signal**(MT_LIB), **cond_timedwait**(MT_LIB), **sema_init**(MT_LIB), **sema_post**(MT_LIB), **sema_trywait**(MT_LIB),

**sema_wait**(MT_LIB)

## LEVEL

Level 1

**ctime**, **localtime**, **gmtime**, **asctime**, **tzset** – convert date and time to string

**#include <time.h>**

**char *ctime(const time_t** ***clock*****);**

**struct tm *localtime(const time_t** ***clock*****);**

**struct tm *gmtime(const time_t** ***clock*****);**

**char *asctime(const struct tm** ***tm*****);**

**extern int daylight;**

**extern char *tzname[2];**

**void tzset(void);**

**ctime**, **localtime**, and **gmtime** accept arguments of type **time_t**, pointed to by *clock*, representing the time in seconds since 00:00:00 UTC, January 1, 1970. **ctime** returns a pointer to a 26-character string as shown below. Time zone and daylight savings corrections are made before the string is generated. The fields are constant in width:

```
Fri Aug 13 00:00:00 1993\n\0
```

**localtime** and **gmtime** return pointers to **tm** structures, described below. **localtime** corrects for the main time zone and possible alternate (‘‘daylight savings’’) time zone; **gmtime** converts directly to Coordinated Universal Time (UTC), which is the time the UNIX system uses internally. **asctime** converts a **tm** structure to a 26-character string, as shown in the above example, and returns a pointer to the string. Declarations of all the functions and externals, and the **tm** structure, are in the **time.h** header file. The value of **tm_isdst** is positive if daylight savings time is in effect, zero if daylight savings time is not in effect, and negative if the information is not available. (Previously, the value of **tm_isdst** was defined as non-zero if daylight savings time was in effect.) The external variable **timezone** contains the difference, in seconds, between UTC and local standard time. The external variable **daylight** indicates whether time should reflect daylight savings time. **timezone** defaults to 0 (UTC). The external variable **daylight** is non-zero if an alternate time zone exists. The time zone names are contained in the external variable **tzname**, which by default is set to:

```
char *tzname[2] = { "GMT", "   " };
```

These functions know about the peculiarities of this conversion for various time periods for the U.S.A. (specifically, the years 1974, 1975, and 1987). They will handle the new daylight savings time starting with the first Sunday in April, 1987.

**tzset** uses the contents of the environment variable **TZ** to override the value of the different external variables. It also sets the external variable **daylight** to zero if Daylight Savings Time conversions should never be applied for the time zone in use; otherwise, non-zero. **tzset** is called by **asctime** and may also be called by the user. See **environ**() for a description of the **TZ** environment variable.

**getenv**(BA_LIB), **mktime**(BA_LIB), **printf**(BA_LIB), **putenv**(BA_LIB), **setlocale**(BA_OS), **strftime**(BA_LIB), **time**(BA_OS), Level 1. The functions **ctime**, **localtime**, **fgmtime**, **tzset** and **asctime** are BA_LIB functions, and identical to the **ctime** BA_LIB page. **ctime_r**, **localtime_r** and **gmtime_r** are MT_LIB functions. The return values for **ctime**, **localtime**, and **gmtime** point to static data whose content is overwritten by each call. Setting the time during the interval of change from **timezone** to **altzone** or vice versa can produce unpredictable results. The system administrator must change the Julian start and end days annually. Use the reentrant functions for multithreaded applications.

**directory**: **opendir**, **readdir**, **readdir_r**, **rewinddir**, **closedir** – directory operations

**#include <dirent.h>**

**#include <sys/types.h>**

**DIR** ∗**opendir(const char** ∗*filename***);**

**struct dirent** ∗**readdir(DIR** ∗*dirp***);**

**void rewinddir(DIR** ∗*dirp***);**

**int closedir(DIR** ∗*dirp***);**

**opendir** opens the directory named by *filename* and associates a directory stream with it. **opendir** returns a pointer to be used to identify the directory stream in subsequent operations. The directory stream is positioned at the first entry. A null pointer is returned if *filename* cannot be accessed or is not a directory, or if it cannot **malloc** enough memory to hold a **DIR** structure or a buffer for the directory entries. **readdir** returns a pointer to the next active directory entry and positions the directory stream at the next entry. No inactive entries are returned. It returns **NULL** upon reaching the end of the directory or upon detecting an invalid location in the directory. **readdir** buffers several directory entries per actual read operation; **readdir** marks for update the **st_atime** field of the directory each time the directory is actually read. The structure **dirent** defined by the **<dirent.h>** header file describes a directory entry. It includes the filename (**d_name**), which is a null-terminated string of at most **{NAME_MAX}** characters:

```
char d_name[{NAME_MAX}]; /* name of file */
```

**rewinddir** resets the position of the named directory stream to the beginning of the directory. It also causes the directory stream to refer to the current state of the corresponding directory, as a call to **opendir** would. **closedir** closes the named directory stream and frees the **DIR** structure.

**dirent**(BA_ENV), **mkdir**(BA_OS), **rmdir**(BA_OS) Level 1.

## NAME

**flockfile** – grant thread ownership of a file

## SYNOPSIS

```
#include <stdio.h>

void flockfile(FILE *file);
```

## DESCRIPTION

This function provide for explicit application-level locking of standard I/O objects. It can be used by a thread to begin a sequence of I/O statements that are to be executed as a unit.

**flockfile** gives the calling thread ownership of *file* if *file* is not currently owned by another thread. A thread keeps ownership of the file until it calls **funlockfile**.

A thread that tries to get ownership of a file that is currently owned by another thread is suspended until the current owner relinquishes the file.

A thread can do multiple calls to **flockfile** and **ftrylockfile** and not get suspended if it currently owns the file, but an equal number of **funlockfile** calls are necessary to relinquish the object completely.

## USAGE

The **flockfile** lock may not be enforced by the implementation. All accesses to a shared stream should be protected explicitly.

## SEE ALSO

**ftrylockfile**(MT_LIB), **funlockfile**(MT_LIB), **stdio**(BA_LIB)

## LEVEL

Level 1.

## NAME

**ftrylockfile** – grant thread ownership of a file

## SYNOPSIS

**#include <stdio.h>**

**int ftrylockfile(FILE** ∗*file***);**

## DESCRIPTION

This function provide for explicit application-level locking of standard I/O objects. It can be used by a thread to begin a sequence of I/O statements that are to be executed as a unit.

**ftrylockfile** gives the calling thread ownership of *file* if *file* is not currently owned by another thread. This function is similar to **flockfile**, except that it returns nonzero if the *file* is already locked. It returns zero on success.

A thread can do multiple calls to **ftrylockfile** and **flockfile** successfully if it currently owns the file, but an equal number of **funlockfile** calls are necessary to relinquish the object completely.

### Return Values

On success, **ftrylockfile** returns zero and locks the stream *file*. If the stream is already locked by a different thread, it returns nonzero.

## USAGE

The **ftrylockfile** lock may not be enforced by the implementation. All accesses to a shared stream should be protected explicitly.

## SEE ALSO

**flockfile**(MT_LIB), **funlockfile**(MT_LIB), **stdio**(BA_LIB)

## LEVEL

Level 1.

## NAME

**funlockfile** – relinquish thread ownership of a file

## SYNOPSIS

**#include <stdio.h>**

**void funlockfile(FILE** **file*);

## DESCRIPTION

This function provide for explicit application-level locking of standard I/O objects. It can be used by a thread to end a sequence of I/O statements that are to be executed as a unit.

**funlockfile** relinquishes ownership granted to the thread via a previous successful call to **flockfile** or **ftrylockfile**.

A thread can nest calls to **flockfile** or **ftrylockfile** as long as each successful call has a corresponding **funlockfile** call to relinquish ownership completely.

The behavior is undefined if **funlockfile** is called without a matching successful call to **flockfile** or **ftrylockfile**.

## USAGE

The **funlockfile** lock may not be enforced by the implementation. All accesses to a shared stream should be protected explicitly.

## SEE ALSO

**flockfile**(MT_LIB), **ftrylockfile**(MT_LIB), **stdio**(BA_LIB)

## LEVEL

Level 1.

**getc**, **getchar**, **fgetc**, **getw** – get character or word from a stream

**#include <stdio.h>**

**int getc(FILE ****stream***);**

**int getchar(void);**

**int fgetc(FILE ****stream***);**

**int getw(FILE ****stream***);**

**getc** returns the next character (that is, byte) from the named input *stream* as an **unsigned char** converted to an **int**. It also moves the file pointer, if defined, ahead one character in *stream*. **getchar** is defined as **getc(stdin)**. **getc** and **getchar** are macros. **fgetc** behaves like **getc**, but is a function rather than a macro. **fgetc** runs more slowly than **getc**, but it takes less space per invocation and its name can be passed as an argument to a function. **getw** returns the next word (that is, integer) from the named input *stream*. **getw** increments the associated file pointer, if defined, to point to the next word. The size of a word is the size of an integer and varies from machine to machine. **getw** assumes no special alignment in the file.

**scanf**(BA_LIB), **stdio**(BA_LIB), **ungetc**(BA_LIB) Level 1.

**getlogin** – get login name
**#include <stdlib.h>**

**char *getlogin(void);**
**getlogin** returns a pointer to the login name It may be used in conjunction with **getpwnam** to locate the correct password file entry when the same user id is shared by several login names. If **getlogin** is called within a process that is not attached to a terminal, it returns a null pointer. The correct procedure for determining the login name is to call **cuserid**, or to call **getlogin** and if it fails to call **getpwuid**. **cuserid**(BA_LIB), **getgrent**(BA_LIB), **getpwent**(BA_LIB) Level 1. The return values point to static data whose content is overwritten by each call.

**getpass** – read a password

**#include <unistd.h>**

**char** ∗**getpass(const char** ∗*prompt***);**

**getpass** reads up to a newline or **EOF** from the file **/dev/tty**, after prompting on the standard error output with the null-terminated string *prompt* and disabling echoing. A pointer is returned to a null-terminated string of at most 8 characters. If **/dev/tty** cannot be opened, a null pointer is returned. An interrupt will terminate input and send an interrupt signal to the calling program before returning.

## NAME

**lio_listio** – issue list of I/O requests

## SYNOPSIS

```
#include <aio.h>

int lio_listio(int mode, struct aiocb *list[],
               int nent,struct sigevent *sig);
```

## DESCRIPTION

**lio_listio** supports issuing a list of I/O requests with a single function call.

The *mode* argument takes one of the values **LIO_WAIT** or **LIO_NOWAIT** and determines whether the function returns when the I/O operations have been completed, or as soon as the operations have been queued. If the *mode* argument is **LIO_WAIT**, the function waits until all I/O is complete and the *sig* argument is ignored.

When **LIO_WAIT** is selected, **lio_listio** waits for all requests that were successfully queued to complete before returning. If no requests were successfully queued, it returns immediately.

If the *mode* argument is **LIO_NOWAIT**, the function returns immediately, and completion notification occurs according to the *sig* argument, when all the I/O operations complete. If *sig* is **NULL**, no notification occurs. If *sig* is not **NULL**, notification occurs according to the same rules as the **aio_sigevent** field of the **aiocb** structure.

The *list* argument is an array of pointers to **aiocb** structures. The array contains *nent* elements. The array may contain **NULL** elements which are ignored.

The I/O requests enumerated by *list* are submitted in an unspecified order.

The **aio_lio_opcode** field of each **aiocb** structure specifies the operation to be performed. The supported operations are **LIO_READ**, **LIO_WRITE**, and **LIO_NOP**. The **LIO_NOP** operation causes the list entry to be ignored. If the **aio_lio_opcode** element is equal to **LIO_READ**, an I/O operation is submitted as if by a call to **aio_read** with the **aiocbp** equal to the address of the **aiocb** structure. If the **aio_lio_opcode** element is equal to **LIO_WRITE**, an I/O operation is submitted as if by a call to **aio_write** with the **aiocbp** equal to the address of the **aiocb** structure.

Other members of the **aiocb** structure (e.g. **aio_fildes**, **aio_buf**, **aio_nbytes**, If the *mode* argument has the value **LIO_WAIT**, the **lio_listio** function returns zero when all the indicated I/O has completed successfully. Otherwise, **lio_listio** returns a value of –1 and sets **errno** to indicate the error.

Both of these return values only indicate the success or failure of the **lio_listio** call itself, not the status of the individual I/O requests. In some cases one or more of the I/O requests contained in the list may fail. Failure of an individual request does not prevent completion of any other individual request. To determine the outcome of each I/O request, the application examines the error status associated with each **aiocb** structure. The error statuses returned are identical to those returned as the result of an **aio_read** or **aio_write** function.

### Errors

If any of the following conditions occurs, the **lio_listio** function returns –1 and sets **errno** to:

| | |
|---|---|
| **EAGAIN** | The resources necessary to queue all the I/O requests were not available. The application may check the error status for each **aiocb** to determine the individual request(s) that failed. |
| **EAGAIN** | The number of entries indicated by *nent* would cause the system-wide limit **{AIO_MAX}** to be exceeded. All entries return **EAGAIN** when queried with **aio_error**. |
| **EINVAL** | The *mode* argument is not a proper value. |
| **EINVAL** | The value of *nent* was greater than **{AIO_LISTIO_MAX}**, or **nent** is zero. |
| **EINVAL** | All requests are **NULL** or have their **lio_listio** set to **LIO_NOP**. |
| **EINTR** | A signal was delivered while waiting for all I/O requests to complete during a **LIO_WAIT** operation. |
| **EIO** | One or more of the individual I/O operations failed. The application may check the error status for each **aiocb** structure to determine the individual request(s) that failed. |

## SEE ALSO

**aiocb**(MT_LIB), **aio_read**(MT_LIB), **aio_suspend**(MT_LIB), **aio_write**(MT_LIB)

## LEVEL

Level 1.

## NAME

**mutex_destroy** – destroy a mutex

## SYNOPSIS

**#include <synch.h>**

**int mutex_destroy(mutex_t** **mutex***);**

### Parameters

| | |
|---|---|
| *mutex* | pointer to mutex to be destroyed |

## DESCRIPTION

**mutex_destroy** destroys the mutex pointed to by *mutex*. This includes invalidating the mutex and freeing any associated implementation-allocated dynamic resources.

Any user-allocated dynamic storage is unaffected by **mutex_destroy** and must be explicitly released by the program.

### Return Values

**mutex_destroy** returns zero for success and an error number for failure, as described below.

### Errors

If any of the following conditions is detected, **mutex_destroy** returns the corresponding value:

| | |
|---|---|
| **EBUSY** | *mutex* is locked or another thread is waiting to acquire *mutex*. |
| **EINVAL** | Invalid argument specified. |

## USAGE

### Warnings

**mutex_destroy** should not be called on a *mutex* while a thread is **cond_waiting** with a pointer to that *mutex* as a parameter.

## SEE ALSO

**mutex**(MT_LIB), **mutex_init**(MT_LIB), **mutex_lock**(MT_LIB), **mutex_trylock**(MT_LIB), **mutex_unlock**(MT_LIB)

## LEVEL

Level 1

## NAME

**mutex_init** – initialize a mutex

## SYNOPSIS

```
#include <synch.h>

int mutex_init(mutex_t *mutex, int type, void *arg);
```

### Parameters

| | |
|---|---|
| *mutex* | pointer to mutex to be initialized |
| *type* | **USYNC_THREAD** or **USYNC_PROCESS** |
| *arg* | **NULL** (reserved for future use) |

## DESCRIPTION

**mutex_init** initializes the mutual exclusion lock (mutex) pointed to by *mutex* to be of type *type* and in the unlocked state. Once initialized, the mutex can be used any number of times without being re-initialized.

### mutex Parameter

*mutex* points to the mutex to be initialized.

### type Parameter

*type* can be set to one of the following values:

| | |
|---|---|
| **USYNC_THREAD** | Initialize the mutex for threads within the current process. |
| **USYNC_PROCESS** | Initialize the mutex for threads across processes. |

### arg Parameter

*arg* should be set to **NULL**. It is not currently used, but is reserved for future use.

### Static Mutex Initialization

A mutex can be initialized statically if its storage is zero-filled. In this case, the mutex is of *type* **USYNC_THREAD**, and **mutex_init** need not be called.

### Return Values

**mutex_init** returns zero for success and an error number for failure, as described below.

### Errors

If the following condition is detected, the contents of *mutex* are unchanged and **mutex_init** returns the value:

| | |
|---|---|
| **EINVAL** | Invalid *type* argument specified. |

## USAGE

### Warnings

**mutex_init** does not examine the *mutex* argument before initializing it. If **mutex_init** is called more than once for the same mutex, it will overwrite its state. It is the user's responsibility to ensure that **mutex_init** is only called once for each mutex. **mutex_init** should not be called on a *mutex* while a thread is **cond_waiting** with a pointer to that *mutex* as a parameter.

Operations on locks initialized with **mutex_init** are not recursive—a thread can deadlock if it attempts to relock a mutex that it already has locked.

## SEE ALSO

**mutex**(MT_LIB), **mutex_destroy**(MT_LIB), **mutex_lock**(MT_LIB), **mutex_trylock**(MT_LIB), **mutex_unlock**(MT_LIB)

## LEVEL

Level 1

## NAME

**mutex_lock** – lock a mutex

## SYNOPSIS

**#include <synch.h>**

**int mutex_lock(mutex_t** **mutex***);**

### Parameters

| | |
|---|---|
| *mutex* | pointer to mutex to be locked |

## DESCRIPTION

**mutex_lock** locks the mutual exclusion lock (mutex) pointed to by *mutex*. If *mutex* is locked, the calling thread is blocked until *mutex* becomes available. When **mutex_lock** returns successfully, the caller has locked *mutex*.

*mutex* must previously have been initialized, either by **mutex_init**, or statically (see **mutex_init**).

From the point of view of the caller, **mutex_lock** is atomic: even if interrupted by a signal or **forkall**(MT_OS), **mutex_lock** will not return until it holds the locked *mutex*. As a consequence, if **mutex_lock** is interrupted, an error indication such as **EINTR** is never returned to the caller.

### Return Values

**mutex_lock** returns zero for success and an error number for failure, as described below.

### Errors

If the following condition is detected, **mutex_lock** returns the value:

| | |
|---|---|
| **EINVAL** | Invalid argument specified. |

## USAGE

Mutexes acquired with **mutex_lock** should be released with **mutex_unlock**.

### Warnings

If a thread exits while holding a mutex, the mutex will not be unlocked, and other threads waiting for the mutex will wait forever. Similarly, if a process exits while holding a **USYNC_PROCESS** mutex, the mutex will not be unlocked, and other processes or threads waiting for the mutex may wait forever.

## SEE ALSO

**forkall**(MT_OS), **mutex**(MT_LIB), **mutex_destroy**(MT_LIB), **mutex_init**(MT_LIB), **mutex_trylock**(MT_LIB), **mutex_unlock**(MT_LIB)

## LEVEL

Level 1

## NAME

**mutex_trylock** – conditionally lock a mutex

## SYNOPSIS

**#include <synch.h>**

**int mutex_trylock(mutex_t** **mutex***);**

### Parameters

*mutex* pointer to mutex to be locked

## DESCRIPTION

**mutex_trylock** attempts once to lock the mutual exclusion lock (mutex) pointed to by *mutex*.

If *mutex* is available, **mutex_trylock** will return successfully with *mutex* locked. If *mutex* is already locked by another thread, **mutex_trylock** immediately returns **EBUSY** to the caller without acquiring *mutex* or blocking.

*mutex* must previously have been initialized, either by **mutex_init**, or statically [see **mutex_init**(MT_LIB)].

### Return Values

**mutex_trylock** returns zero for success and an error number for failure, as described below.

### Errors

If the following condition occurs, **mutex_trylock** returns the value:

**EBUSY** *mutex* is locked by another thread.

If any of the following conditions is detected, **mutex_trylock** fails and returns the corresponding value:

**EINVAL** Invalid argument specified.

## USAGE

**mutex_trylock** is used when the caller does not want to block.

Mutexes acquired with **mutex_trylock** should be released with **mutex_unlock**.

## SEE ALSO

**mutex**(MT_LIB), **mutex_destroy**(MT_LIB), **mutex_init**(MT_LIB), **mutex_lock**(MT_LIB), **mutex_unlock**(MT_LIB)

## LEVEL

Level 1

## NAME

**mutex_unlock** – unlock a mutex

## SYNOPSIS

```
#include <synch.h>

int mutex_unlock(mutex_t *mutex);
```

### Parameters

| | |
|---|---|
| *mutex* | pointer to mutex to be unlocked |

## DESCRIPTION

**mutex_unlock** unlocks the mutex pointed to by *mutex*.

If there are one or more threads waiting for the mutex when **mutex_unlock** is called, at least one waiting thread is allowed to try again to acquire the mutex.

### Return Values

**mutex_unlock** returns zero for success and an error number for failure, as described below.

### Errors

If the following condition is detected, **mutex_unlock** returns the value:

| | |
|---|---|
| **EINVAL** | Invalid argument specified. |

## USAGE

Mutexes acquired with **mutex_lock** and **mutex_trylock** should be released with **mutex_unlock**.

## SEE ALSO

**mutex**(MT_LIB), **mutex_destroy**(MT_LIB), **mutex_init**(MT_LIB), **mutex_lock**(MT_LIB), **mutex_trylock**(MT_LIB)

## LEVEL

Level 1

**putc**, **putchar**, **fputc**, **putw** – put character or word on a stream

**#include <stdio.h>**

**int putc(int** *c*, **FILE** **stream*);

**int putchar(int** *c*);

**int fputc(int** *c*, **FILE** **stream*);

**int putw(int** *w*, **FILE** **stream*);

**putc** writes *c* (converted to an **unsigned char**) onto the output *stream* at the position where the file pointer (if defined) is pointing, and advances the file pointer appropriately. If the file cannot support positioning requests, or *stream* was opened with append mode, the character is appended to the output *stream*. **putchar(c)** is defined as **putc(c, stdout)**. **putc** and **putchar** are macros. **fputc** behaves like **putc**, but is a function rather than a macro. **fputc** runs more slowly than **putc**, but it takes less space per invocation and its name can be passed as an argument to a function. **putw** writes the word (that is, integer) *w* to the output *stream* (where the file pointer, if defined, is pointing). The size of a word is the size of an integer and varies from machine to machine. **putw** neither assumes nor causes special alignment in the file.

**rand**, **srand** – simple random-number generator

**#include <stdlib.h>**

**int rand(void);**

**void srand(unsigned int** *seed***);**

**rand** uses a multiplicative congruent random-number generator with period $2^{32}$ that returns successive pseudo-random numbers in the range from 0 to **RAND_MAX** (defined in **stdlib.h**). The function **srand** uses the argument *seed* as a seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to the function **rand**. If the function **srand** is then called with the same *seed* value, the sequence of pseudo-random numbers will be repeated. If the function **rand** is called before any calls to **srand** have been made, the same sequence will be generated as when **srand** is first called with a *seed* value of 1. **drand48**(BA_LIB)

Level 2: September 30, 1989.

*Level 2: June 1993.

The spectral properties of **rand** are limited. **drand48**(BA_LIB) provides a much better, though more elaborate, random-number generator. Each thread that accesses one of the functions **drand48, lrand48, mrand48, srand48, seed48**, or **lcong48** should be coded as per the following example:

```
        mutex_lock(I_am_using_drand48);
        value = FUNCTION();
        mutex_unlock(I_am_using_drand48);
```

where **FUNCTION** is one of those listed. The same mutex must be used for all six functions.

## NAME

**rmutex_destroy** – destroy a recursive mutex

## SYNOPSIS

**#include <synch.h>**

**int rmutex_destroy(rmutex_t** **rmutex***);**

### Parameters

*rmutex* pointer to recursive mutex to be destroyed

## DESCRIPTION

**rmutex_destroy** destroys the recursive mutual exclusion lock (rmutex) pointed to by *rmutex*. This includes invalidating the rmutex and freeing any associated implementation-allocated dynamic resources.

Any user-allocated dynamic storage is unaffected by **rmutex_destroy** and must be explicitly released by the program.

### Return Values

**rmutex_destroy** returns zero for success and an error number for failure, as described below.

### Errors

If any of the following conditions is detected, **rmutex_destroy** returns the corresponding value:

| | |
|---|---|
| **EBUSY** | *rmutex* is locked by another thread or another thread is waiting to acquire *rmutex*. |
| **EINVAL** | Invalid argument specified. |

## SEE ALSO

**mutex**(MT_LIB), **rmutex**(MT_LIB), **rmutex_init**(MT_LIB), **rmutex_lock**(MT_LIB), **rmutex_trylock**(MT_LIB), **rmutex_unlock**(MT_LIB)

## LEVEL

Level 1

## NAME

**rmutex_init** – initialize a recursive mutex

## SYNOPSIS

**#include <synch.h>**

**int rmutex_init(rmutex_t** **rmutex*, **int** *type*, **void** **arg***);**

### Parameters

| | |
|---|---|
| *rmutex* | pointer to recursive mutex to be initialized |
| *type* | **USYNC_THREAD** or **USYNC_PROCESS** |
| *arg* | **NULL** (reserved for future use) |

## DESCRIPTION

**rmutex_init** initializes the recursive mutual exclusion lock (rmutex) pointed to by *rmutex* to be of type *type* and in the unlocked state. Once initialized, the rmutex can be used any number of times without being re-initialized.

All operations on locks initialized with **rmutex_init** are recursive.

### rmutex Parameter

*rmutex* points to the rmutex to be initialized.

### type Parameter

*type* can be set to one of the following values:

| | |
|---|---|
| **USYNC_THREAD** | Initialize the rmutex for threads within the current process. |
| **USYNC_PROCESS** | Initialize the rmutex for threads across processes. |

### arg Parameter

*arg* should be set to **NULL**. It is not currently used, but is reserved for future use.

### Return Values

**rmutex_init** returns zero for success and an error number for failure, as described below.

### Errors

If the following condition is detected, **rmutex_init** returns the value:

| | |
|---|---|
| **EINVAL** | Invalid argument specified. |

## SEE ALSO

**mutex**(MT_LIB), **rmutex**(MT_LIB), **rmutex_destroy**(MT_LIB), **rmutex_lock**(MT_LIB), **rmutex_trylock**(MT_LIB), **rmutex_unlock**(MT_LIB)

## LEVEL

Level 1

## NAME

**rmutex_lock** – lock a recursive mutex

## SYNOPSIS

**#include <synch.h>**

**int rmutex_lock(rmutex_t** **rmutex***);**

### Parameters

| | |
|---|---|
| *rmutex* | pointer to recursive mutex to be locked |

## DESCRIPTION

**rmutex_lock** locks the recursive mutual exclusion lock (rmutex) pointed to by *rmutex*. If *rmutex* is locked by another thread, the calling thread is blocked until *rmutex* becomes available. When **rmutex_lock** returns successfully, the caller has locked *rmutex*.

If *rmutex* is already locked by the calling thread, the recursive depth is incremented and control is returned to the caller, as if the lock had just been acquired.

*rmutex* must previously have been initialized (see **rmutex_init**).

From the point of view of the caller, **rmutex_lock** is atomic: even if interrupted by a signal or **forkall**(MT_OS), **rmutex_lock** will not return until the lock is held.

### Return Values

**rmutex_lock** returns zero for success and an error number for failure, as described below.

### Errors

If the following condition is detected, **rmutex_lock** fails and returns the value:

| | |
|---|---|
| **EINVAL** | Invalid argument specified. |

## USAGE

The locks acquired with **rmutex_lock** should be released with **rmutex_unlock**.

### Warnings

If a thread exits while holding an rmutex, the rmutex will not be unlocked, and other threads waiting for the rmutex will wait forever. Similarly, if a process exits while holding a **USYNC_PROCESS** rmutex, the rmutex will not be unlocked, and other processes or threads waiting for the rmutex will wait forever.

## SEE ALSO

**forkall**(MT_OS), **mutex**(MT_LIB), **rmutex**(MT_LIB), **rmutex_destroy**(MT_LIB), **rmutex_init**(MT_LIB), **rmutex_trylock**(MT_LIB), **rmutex_unlock**(MT_LIB)

## LEVEL

Level 1

## NAME

**rmutex_trylock** – conditionally lock a recursive mutex

## SYNOPSIS

```
#include <synch.h>

int rmutex_trylock(rmutex_t *rmutex);
```

### Parameters

*rmutex* pointer to recursive mutex to be locked

## DESCRIPTION

**rmutex_trylock** attempts once to lock the recursive mutual exclusion lock (rmutex) pointed to by *rmutex*.

If *rmutex* is available, **rmutex_trylock** will return successfully with *rmutex* locked. If *rmutex* is already locked by another thread, **rmutex_trylock** immediately returns **EBUSY** to the caller without locking *rmutex* or blocking. If *rmutex* is already held by the calling thread, the recursive depth is incremented and control is returned to the caller, as if the lock had just been acquired.

*rmutex* must previously have been initialized (see **rmutex_init**).

### Return Values

**rmutex_trylock** returns zero for success and an error number for failure, as described below.

### Errors

If any of the following conditions is detected, **rmutex_trylock** returns the corresponding value:

| | |
|---|---|
| **EBUSY** | *rmutex* is locked by another thread. |
| **EINVAL** | Invalid argument specified. |

## USAGE

**rmutex_trylock** is used when the caller does not want to block.

The locks acquired with **rmutex_trylock** should be released with **rmutex_unlock**.

## SEE ALSO

**mutex**(MT_LIB), **rmutex**(MT_LIB), **rmutex_destroy**(MT_LIB), **rmutex_init**(MT_LIB), **rmutex_lock**(MT_LIB), **rmutex_unlock**(MT_LIB)

## LEVEL

Level 1

## NAME

**rmutex_unlock** – unlock a recursive mutex

## SYNOPSIS

**#include <synch.h>**

**int rmutex_unlock(rmutex_t** **rmutex***);**

### Parameters

*rmutex* pointer to recursive mutex to be unlocked

## DESCRIPTION

**rmutex_unlock** unlocks the recursive mutual exclusion lock (rmutex) pointed to by *rmutex*.

**rmutex_unlock** checks the identity of the caller and if the caller is the current owner of *rmutex* it checks the depth count.

> If the depth count is greater than 0, it decrements the count and returns to the caller without unlocking the rmutex.
>
> If the depth count is 0, the rmutex is unlocked.

If the caller is not the current owner of *rmutex*, that is, the caller does not hold the lock, **rmutex_unlock** will fail and return **EACCES**.

If there are one or more threads waiting for *rmutex* when it is unlocked, at least one waiting thread is allowed to try again to lock *rmutex*.

### Return Values

**rmutex_unlock** returns zero for success and an error number for failure, as described below.

### Errors

If any of the following conditions is detected, **rmutex_unlock** returns the corresponding value:

| | |
|---|---|
| **EACCES** | The caller did not previously lock the *rmutex*. |
| **EINVAL** | Invalid argument specified. |

## SEE ALSO

**mutex**(MT_LIB), **rmutex**(MT_LIB), **rmutex_destroy**(MT_LIB), **rmutex_init**(MT_LIB), **rmutex_lock**(MT_LIB), **rmutex_trylock**(MT_LIB)

## LEVEL

Level 1

## NAME

**`rw_rdlock`** – acquire a reader-writer lock in read mode.

## SYNOPSIS

```
#include <synch.h>

int rw_rdlock(rwlock_t *lock);
```

### Parameters

| | |
|---|---|
| *lock* | pointer to the reader-writer lock to be acquired |

## DESCRIPTION

**`rw_rdlock`** acquires the reader-writer lock pointed to by *lock* in read mode.

A reader-writer lock can be held by any number of readers at one time, but only one writer at a time can hold the lock.

*lock* must previously have been initialized (see **`rwlock_init`**).

From the point of view of the caller, this function is atomic: even if interrupted by a signal or **`forkall`**(BA_OS), **`rw_rdlock`** will not return until it holds the lock. As a consequence, if **`rw_rdlock`** is interrupted, an error indication such as **`EINTR`** is never returned to the user.

### Return Values

**`rw_rdlock`** returns zero for success and an error number for failure, as described below.

### Errors

If any of the following conditions is detected, **`rw_rdlock`** returns the corresponding value:

| | |
|---|---|
| **`EFAULT`** | *lock* points to an illegal address. (This error may not be detected; a **`SIGSEGV`** signal may be posted to the faulting thread if an illegal address is used.) |
| **`EINVAL`** | Invalid argument specified |
| **`ENOMEM`** | Insufficient memory |

## USAGE

For consistency, locks acquired with **`rw_rdlock`** should be released with **`rw_unlock`**.

### Warnings

If a thread exits while holding a reader-writer lock, the lock will not be unlocked, and other threads waiting for the lock will wait forever. Similarly, if a process exits while holding a **`USYNC_PROCESS`** reader-writer lock, the lock will not be unlocked, and other processes waiting for the reader-writer lock will wait forever.

### Comment

In a typical implementation, once a writer has requested the lock with **`rw_wrlock`**, all subsequent requests for the lock in either read or write mode are queued.

If the lock is free, or is currently held by another reader and there are no writers waiting, **`rw_rdlock`** increments the reader count and the caller proceeds. If a writer holds the lock or if any writer is waiting for the lock, the caller blocks to wait for the

## SEE ALSO

**forkall**(BA_OS), **rwlock**(MT_LIB), **rwlock_destroy**(MT_LIB), **rwlock_init**(MT_LIB), **rw_tryrdlock**(MT_LIB), **rw_trywrlock**(MT_LIB), **rw_unlock**(MT_LIB), **rw_wrlock**(MT_LIB)

## LEVEL

Level 1

## NAME

**rw_tryrdlock** – conditionally acquire a reader-writer lock in read mode

## SYNOPSIS

**#include <synch.h>**

**int rw_tryrdlock(rwlock_t** **lock***);**

### Parameters

| | |
|---|---|
| *lock* | pointer to the lock to be acquired |

## DESCRIPTION

**rw_tryrdlock** attempts once to acquire the reader-writer lock pointed to by *lock* in read mode; it does not block the caller if the lock is unavailable.

A reader-writer lock can be held by any number of readers at one time, but only one writer at a time can hold the lock.

If the lock is free, **rw_trywrlock** increments the reader count and the caller proceeds.

If the lock is not free, **rw_tryrdlock** immediately returns **EBUSY** to the caller, without acquiring the lock.

*lock* must previously have been initialized (see **rwlock_init**).

### Return Values

**rw_tryrdlock** returns zero for success and an error number for failure, as described below.

### Errors

If any of the following conditions is detected, **rw_tryrdlock** fails and returns the corresponding value:

| | |
|---|---|
| **EBUSY** | The reader-writer lock pointed to by *lock* cannot be acquired. |
| **EINVAL** | Invalid argument specified. |

## USAGE

**rw_tryrdlock** is used when the caller does not want to block if the lock is unavailable.

For consistency, locks acquired with **rw_tryrdlock** should be released with **rw_unlock**.

## SEE ALSO

**rwlock**(MT_LIB), **rwlock_destroy**(MT_LIB), **rwlock_init**(MT_LIB), **rw_rdlock**(MT_LIB), **rw_trywrlock**(MT_LIB), **rw_unlock**(MT_LIB),

**rw_wrlock**(MT_LIB)

## LEVEL

Level 1

## NAME

**rw_trywrlock** – conditionally acquire a reader-writer lock in write mode

## SYNOPSIS

**#include <synch.h>**

**int rw_trywrlock(rwlock_t** **lock***);**

### Parameters

| | |
|---|---|
| *lock* | pointer to the reader-writer lock to be acquired |

## DESCRIPTION

**rw_trywrlock** makes a single attempt to acquire the reader-writer lock pointed to by *lock* in write mode; it does not block the caller if the lock is unavailable.

A reader-writer lock can be held by any number of readers at one time, but only one writer at a time can hold the lock.

If the lock is free, **rw_trywrlock** acquires the lock in write mode and the caller proceeds.

If the lock is not free, **rw_trywrlock** immediately returns **EBUSY** to the caller, without acquiring the lock.

*lock* must previously have been initialized (see **rwlock_init**).

### Return Values

**rw_trywrlock** returns zero for success and an error number for failure, as described below.

### Errors

If any of the following conditions is detected, **rw_trywrlock** fails and returns the corresponding value:

| | |
|---|---|
| **EBUSY** | The reader-writer lock pointed to by *lock* cannot be acquired. |
| **EINVAL** | Invalid argument specified. |

## USAGE

**rw_trywrlock** is used when the caller does not want to block if the lock is unavailable.

For consistency, locks acquired with **rw_trywrlock** should be released with **rw_unlock**.

## SEE ALSO

**rwlock**(MT_LIB), **rwlock_destroy**(MT_LIB), **rwlock_init**(MT_LIB), **rw_rdlock**(MT_LIB), **rw_tryrdlock**(MT_LIB), **rw_unlock**(MT_LIB),

**rw_wrlock**(MT_LIB)

## LEVEL

Level 1

## NAME

**rw_unlock** – release a reader-writer lock

## SYNOPSIS

**#include <synch.h>**

**int rw_unlock(rwlock_t** **lock***);**

### Parameters

| | |
|---|---|
| *lock* | pointer to the lock to be released |

## DESCRIPTION

**rw_unlock** releases a reader-writer lock previously acquired by **rw_rdlock**, **rw_wrlock**, **rw_tryrdlock**, or **rw_trywrlock**. The behavior differs according to whether the caller is a reader or a writer, and is implementation defined (see **rw_wrlock**).

*lock* must previously have been initialized (see **rwlock_init**).

### Return Values

**rw_unlock** returns zero for success and an error number for failure, as described below.

### Errors

If any of the following conditions is detected, **rw_unlock** fails and returns the corresponding value:

| | |
|---|---|
| **EINVAL** | Invalid argument specified. |
| **ENOLCK** | *lock* is not locked. |

### Comment

In a typical implementation, when a writer calls **rw_unlock**, the lock is unlocked.

When a reader calls **rw_unlock**, the reader count is decremented. If the reader count is zero, **rw_unlock** unlocks the lock, otherwise, the lock is not unlocked.

When **rw_unlock** unlocks the lock, the first waiter (reader or writer) is activated.

If the thread activated is a reader, all subsequent readers are activated (up to the next writer or end of queue) and the count of readers holding the lock is changed to reflect this.

If the thread activated is a writer, no other threads are activated and the lock is marked as being held by a writer.

## SEE ALSO

**rwlock**(MT_LIB), **rwlock_destroy**(MT_LIB), **rwlock_init**(MT_LIB), **rw_rdlock**(MT_LIB), **rw_tryrdlock**(MT_LIB), **rw_trywrlock**(MT_LIB),

**rw_wrlock**(MT_LIB)

## LEVEL

Level 1

## NAME

**rw_wrlock** – acquire a reader-writer lock in write mode

## SYNOPSIS

```
#include <synch.h>

int rw_wrlock(rwlock_t *lock);
```

### Parameters

| | |
|---|---|
| *lock* | pointer to the reader-writer lock to be acquired |

## DESCRIPTION

**rw_wrlock** acquires the reader-writer lock pointed to by *lock* in write mode.

Only one writer at a time can hold a reader-writer lock, although any number of readers can hold the lock at any time.

*lock* must previously have been initialized (see **rwlock_init**).

From the point of view of the application, this function is atomic: even if interrupted by a signal or **forkall**, **rw_wrlock** will not return until it holds the lock. As a consequence, if **rw_wrlock** is interrupted, an error indication such as **EINTR** is never returned to the user.

### Return Values

**rw_wrlock** returns zero for success and an error number for failure, as described below.

### Errors

If any of the following conditions is detected, **rw_wrlock** returns the corresponding value:

| | |
|---|---|
| **EINVAL** | Invalid argument specified. |
| **ENOMEM** | Insufficient memory |

## USAGE

For consistency, locks acquired with **rw_rdlock** should be released with **rw_unlock**.

### Warnings

If a thread exits while holding a reader-writer lock, the lock will not be unlocked, and other threads waiting for the lock will wait forever. Similarly, if a process exits while holding a **USYNC_PROCESS** reader-writer lock, the lock will not be unlocked, and other processes waiting for the reader-writer lock will wait forever.

### Comment

In a typical implementation, once a writer has requested the lock with **rw_wrlock**, all subsequent requests for the lock in either read or write mode are blocked.

The queries and blocking detailed or a reader / writer lock are unspecified, and will likely depend on the scheduling policy or policies.

When no other readers or writers hold the lock, **rw_wrlock** will acquire the lock, and the caller will proceed. Any subsequent write or read requests for the lock will block until the caller unlocks the lock with **rw_unlock**(MT_LIB).

If the lock is held by any readers when **rw_wrlock** is called, and no writer is waiting for the lock, the caller blocks until all the current readers have released the lock. If the lock is held by another writer, or if there are any other writers already waiting for the lock, the caller blocks to wait for the lock.

## SEE ALSO

**rwlock**(MT_LIB), **rwlock_destroy**(MT_LIB), **rwlock_init**(MT_LIB), **rw_rdlock**(MT_LIB), **rw_tryrdlock**(MT_LIB), **rw_trywrlock**(MT_LIB),

**rw_unlock**(MT_LIB)

## LEVEL

Level 1

## NAME

**rwlock**: **rwlock_init**, **rw_rdlock**, **rw_wrlock**, **rw_tryrdlock**, **rw_trywrlock**, **rw_unlock**, **rwlock_destroy**, – overview of reader-writer lock routines

## SYNOPSIS

**#include <synch.h>**

**int rwlock_init(rwlock_t** **lock*, **int** *type*, **void** **arg***);**

**int rw_rdlock(rwlock_t** **lock***);**

**int rw_wrlock(rwlock_t** **lock***);**

**int rw_tryrdlock(rwlock_t** **lock***);**

**int rw_trywrlock(rwlock_t** **lock***);**

**int rw_unlock(rwlock_t** **lock***);**

**int rwlock_destroy(rwlock_t** **lock***);**

### Parameters

| | |
|---|---|
| *lock* | pointer to reader-writer lock to be initialized |
| *type* | **USYNC_THREAD** or **USYNC_PROCESS** |
| *arg* | **NULL** (reserved for future use) |

## DESCRIPTION

Reader-writer locks allow many threads to have simultaneous read-only access to data, while allowing only one thread to have write access at any time. They are typically used to protect data that is searched more often than it is changed.

### rwlock_init

**rwlock_init** initializes the reader-writer lock pointed to by *rwlock* to be of type *type* and in the unlocked state. Once initialized, the lock can be used any number of times without being re-initialized.

### rw_rdlock

**rw_rdlock** acquires the reader-writer lock pointed to by *lock* in read mode.

A reader-writer lock can be held by any number of readers at one time, but only

### rw_tryrdlock

**rw_tryrdlock** attempts once to acquire the reader-writer lock pointed to by *lock* in read mode; it does not block the caller if the lock is unavailable.

A reader-writer lock can be held by any number of readers at one time, but only one writer at a time can hold the lock.

If the lock is free, **rw_trywrlock** increments the reader count and the caller proceeds.

If the lock is not free, **rw_tryrdlock** immediately returns **EBUSY** to the caller, without acquiring the lock.

### rw_trywrlock

**rw_trywrlock** makes a single attempt to acquire the reader-writer lock pointed to by *lock* in write mode; it does not block the caller if the lock is unavailable.

A reader-writer lock can be held by any number of readers at one time, but only one writer at a time can hold the lock.

If the lock is free, **rw_trywrlock** acquires the lock in write mode and the caller proceeds.

If the lock is not free, **rw_trywrlock** immediately returns **EBUSY** to the caller, without acquiring the lock.

### rw_unlock

**rw_unlock** releases a reader-writer lock previously acquired by **rw_rdlock**, **rw_wrlock**, **rw_tryrdlock**, or **rw_trywrlock**. The behavior differs according to whether the caller is a reader or a writer, and is implementation specific.

> When a writer calls **rw_unlock**, the lock is unlocked.
>
> If the thread activated is a writer, no other threads are activated and the lock is marked as being held by a writer.

### rwlock_destroy

**rwlock_destroy** destroys the reader-writer lock pointed to by *lock*. This includes invalidating the lock and freeing any associated dynamically allocated resources.

### USYNC_THREAD and USYNC_PROCESS Reader-Writer Locks

Reader-writer locks are initialized to be one of two types: **USYNC_THREAD** or **USYNC_PROCESS**. **USYNC_THREAD** locks are available only to threads within the current process. **USYNC_PROCESS** locks can be used by threads in different processes.

## USAGE

### Warnings

Operations on locks initialized with **rwlock_init** are not recursive—a thread can deadlock if it attempts to reacquire a reader-writer lock that it already has acquired.

## SEE ALSO

**rwlock_destroy**(MT_LIB), **rwlock_init**(MT_LIB), **rw_rdlock**(MT_LIB), **rw_tryrdlock**(MT_LIB), **rw_trywrlock**(MT_LIB), **rw_unlock**(MT_LIB),

**rw_wrlock**(MT_LIB)

**LEVEL**

Level 1

## NAME

**rwlock_destroy** – destroy a reader-writer lock

## SYNOPSIS

**#include <synch.h>**

**int rwlock_destroy(rwlock_t** **lock***);**

### Parameters

| | |
|---|---|
| *lock* | pointer to the lock to be destroyed |

## DESCRIPTION

**rwlock_destroy** destroys the reader-writer lock pointed to by *lock*. This includes invalidating the lock and freeing any associated dynamically allocated resources.

*lock* must previously have been initialized (see **rwlock_init**).

### Return Values

**rwlock_destroy** returns zero for success and an error number for failure, as described below.

### Errors

If any of the following conditions is detected, **rwlock_destroy** returns the corresponding value:

| | |
|---|---|
| **EBUSY** | *lock* is locked or another thread is waiting to acquire *lock*. |
| **EINVAL** | Invalid argument specified. |

## SEE ALSO

**rwlock**(MT_LIB), **rwlock_init**(MT_LIB), **rw_rdlock**(MT_LIB), **rw_tryrdlock**(MT_LIB), **rw_trywrlock**(MT_LIB), **rw_unlock**(MT_LIB), **rw_wrlock**(MT_LIB)

## LEVEL

Level 1

## NAME

**rwlock_init** – initialize a reader-writer lock

## SYNOPSIS

**#include <synch.h>**

**int rwlock_init(rwlock_t** **rwlock*, **int** *type*, **void** **arg*);

### Parameters

| | |
|---|---|
| *rwlock* | pointer to reader-writer lock to be initialized |
| *type* | **USYNC_THREAD** or **USYNC_PROCESS** |
| *arg* | **NULL** (reserved for future use) |

## DESCRIPTION

**rwlock_init** initializes the reader-writer lock pointed to by *rwlock* to be of type *type* and in the unlocked state. Once initialized, the lock can be used any number of times without being re-initialized.

### rwlock Parameter

*rwlock* points to the reader-writer lock to be initialized.

### type Parameter

*type* can be set to one of the following values:

| | |
|---|---|
| **USYNC_THREAD** | Initialize the reader-writer lock for threads within the current process. |
| **USYNC_PROCESS** | Initialize the reader-writer lock for threads across processes. |

### arg Parameter

*arg* should be set to **NULL**. It is not currently used, but is reserved for future use.

### Return Values

**rwlock_init** returns zero for success and an error number for failure, as described below.

### Errors

If any of the following conditions is detected, the contents of *rwlock* are not changed, and **rwlock_init** returns the corresponding value:

| | |
|---|---|
| **EBUSY** | *rwlock* is locked. |
| **EINVAL** | Invalid *type* argument specified. |

## USAGE

### Warnings

**rwlock_init** does not examine the *rwlock* argument before initializing it. If **rwlock_init** is called more than once for the same reader-writer lock, it will overwrite its state. It is the user's responsibility to ensure that **rwlock_init** is only called once for each reader-writer lock.

Operations on locks initialized with **rwlock_init** are not recursive—a thread can deadlock if it attempts to reacquire a reader-writer lock that it already has acquired.

## SEE ALSO

**rwlock**(MT_LIB), **rwlock_destroy**(MT_LIB), **rw_rdlock**(MT_LIB), **rw_tryrdlock**(MT_LIB), **rw_trywrlock**(MT_LIB), **rw_unlock**(MT_LIB),

**rw_wrlock**(MT_LIB)

**LEVEL**

Level 1

## NAME

**sema_destroy** – destroy a semaphore

## SYNOPSIS

**#include <synch.h>**

**int sema_destroy(sema_t** **sema***);**

### Parameters

| | |
|---|---|
| *sema* | pointer to the semaphore to destroy |

## DESCRIPTION

**sema_destroy** destroys the semaphore pointed to by *sema*. This includes invalidating *sema* and freeing any associated dynamically allocated resources.

### sema Parameter

*sema* must have been previously initialized, either by **sema_init** or statically (see **sema_init**).

### Return Values

**sema_destroy** returns zero for success and an error number for failure, as described below.

### Errors

If any of the following conditions is detected, **sema_destroy** returns the corresponding value:

| | |
|---|---|
| **EBUSY** | *sema* still has threads waiting. |
| **EINVAL** | Invalid argument specified. |

## SEE ALSO

**semaphore**(MT_LIB), **sema_init**(MT_LIB), **sema_post**(MT_LIB), **sema_trywait**(MT_LIB), **sema_wait**(MT_LIB)

## LEVEL

Level 1

## NAME

**sema_init** – initialize a semaphore

## SYNOPSIS

**#include <synch.h>**

**int sema_init(sema_t** **sema*, **int** *sema_count*, **int** *type*, **void** **arg*);

### Parameters

| | |
|---|---|
| *sema* | pointer to semaphore to initialize |
| *sema_count* | number of resources to be protected by the semaphore |
| *type* | **USYNC_THREAD** or **USYNC_PROCESS** |
| *arg* | **NULL** (reserved for future use) |

## DESCRIPTION

**sema_init** initializes the semaphore *sema* of type *type* to protect *sema_count* resources. Once initialized, the semaphore can be used any number of times without being re-initialized.

### sema_count Parameter

*sema_count*, which must be greater than or equal to zero, defines the initial count of resources protected by the semaphore.

### sema Parameter

*sema* points to the semaphore to be initialized.

### type Parameter

*type* can be set to one of the following values:

| | |
|---|---|
| **USYNC_THREAD** | Initialize the semaphore for threads within the current process. |
| **USYNC_PROCESS** | Initialize the semaphore for threads across processes. |

### arg Parameter

*arg* should be set to **NULL**. It is not currently used, but is reserved for future use.

### Static Semaphore Initialization

A semaphore can be initialized statically if its storage is zero-filled. In this case, the semaphore is of *type* **USYNC_THREAD**, its *sema_count* is 0 (that is, it is ‘‘locked’’; no resources are available), and **sema_init** need not be called. **sema_post** must be called to unlock the semaphore.

### Return Values

**sema_init** returns zero for success and an error number for failure, as described below.

### Errors

If the following condition is detected, **sema_init** returns the value:

| | |
|---|---|
| **EINVAL** | Invalid argument specified. |

## USAGE

### Warnings

**sema_init** does not examine the *sema* argument before initializing it. If **sema_init** is called more than once for the same semaphore, it will overwrite its state. It is the user’s responsibility to ensure that **sema_init** is only called once for each semaphore.

Operations on semaphores initialized with **sema_init** are not recursive; a thread can block itself if it attempts to reacquire a semaphore that it has already acquired.

## SEE ALSO

**semaphore**(MT_LIB), **sema_destroy**(MT_LIB), **sema_post**(MT_LIB), **sema_trywait**(MT_LIB), **sema_wait**(MT_LIB)

## LEVEL

Level 1

## NAME

**sema_post** – release a lock by incrementing the count value of the semaphore

## SYNOPSIS

**#include <synch.h>**

**int sema_post(sema_t** **sema***);**

### Parameters

| | |
|---|---|
| *sema* | pointer to the semaphore to increment |

## DESCRIPTION

**sema_post** increments the count of the semaphore pointed to by *sema*, and if the new count value is less than or equal to zero, makes the next thread waiting at the semaphore runnable.

If more than one thread is waiting, release from the blocked group is scheduling policy-specific for bound threads, and may be dependent on scheduling parameters for multiplexed threads.

### sema Parameter

*sema* must previously have been initialized, either by **sema_init** or statically (see **sema_init**).

### Return Values

**sema_post** returns zero for success and an error number for failure, as described below.

### Errors

If the following condition is detected, **sema_post** returns the value:

| | |
|---|---|
| **EINVAL** | Invalid argument specified. |

## SEE ALSO

**semaphore**(MT_LIB), **sema_destroy**(MT_LIB), **sema_init**(MT_LIB), **sema_trywait**(MT_LIB), **sema_wait**(MT_LIB)

## LEVEL

Level 1

## NAME

**sema_trywait** – conditionally claim resources under the semaphore's control

## SYNOPSIS

**#include <synch.h>**

**int sema_trywait(sema_t** **sema***);**

### Parameters

| | |
|---|---|
| *sema* | pointer to the semaphore to acquire |

## DESCRIPTION

**sema_trywait** makes a single attempt to acquire the semaphore pointed to by *sema*. If the semaphore is available, **sema_trywait** decrements the semaphore value and returns to the caller.

If **sema_trywait** cannot immediately acquire the semaphore, it returns **EBUSY** to the caller, it does not block the caller to wait for the semaphore or decrement the semaphore value.

### sema Parameter

*sema* must have been previously initialized, either by **sema_init** or statically (see **sema_init**).

### Return Values

**sema_trywait** returns zero for success and an error number for failure, as described below.

### Errors

If the following condition occurs, **sema_trywait** fails and returns the value:

| | |
|---|---|
| **EBUSY** | The semaphore cannot be acquired immediately |

If the following condition is detected, **sema_trywait** returns the value:

| | |
|---|---|
| **EINVAL** | Invalid argument specified. |

## USAGE

**sema_trywait** is used when the caller does not want to block if the semaphore is unavailable.

## SEE ALSO

**semaphore**(MT_LIB), **sema_destroy**(MT_LIB), **sema_init**(MT_LIB), **sema_post**(MT_LIB), **sema_wait**(MT_LIB)

## LEVEL

Level 1

## NAME

**sema_wait** – acquire a semaphore

## SYNOPSIS

```
#include <synch.h>

int sema_wait(sema_t *sema);
```

### Parameters

| | |
|---|---|
| *sema* | pointer to the semaphore to acquire |

## DESCRIPTION

**sema_wait** acquires the semaphore pointed to by *sema*.

If the semaphore is available (that is, if the semaphore value is greater than zero), **sema_wait** decrements the semaphore value and returns to the caller.

If the semaphore is unavailable (that is, the semaphore value is zero or less), **sema_wait** decrements the semaphore value and suspends execution of the calling thread until the semaphore becomes available to the caller.

If a thread waiting on a semaphore is interrupted by a signal, the signal handler will run, but then the thread will resume waiting for the semaphore. Thus, when **sema_wait** returns without an error, it will always have acquired the semaphore.

### sema Parameter

*sema* must previously have been initialized, either by **sema_init** or statically (see **sema_init**).

### Return Values

**sema_wait** returns zero for success and an error number for failure, as described below.

### Errors

If the following condition is detected, **sema_wait** returns the value:

| | |
|---|---|
| **EINVAL** | Invalid argument specified. |

## USAGE

See the description of semaphores under USAGE on **sema_init**(MT_LIB).

In general, **sema_wait** is used to block wait for an event, or when a critical section is long. Semaphores are asynchronous-safe, and may be used to communicate between signal handlers and base level code.

## SEE ALSO

**semaphore**(MT_LIB), **sema_destroy**(MT_LIB), **sema_init**(MT_LIB), **sema_post**(MT_LIB), **sema_trywait**(MT_LIB)

## LEVEL

Level 1

**string**: **strcat**, **strncat**, **strcmp**, **strncmp**, **strcpy**, **strncpy**, **strdup**, **strlen**, **strchr**, **strrchr**, **strpbrk**, **strspn**, **strcspn**, **strtok**, **strstr** – string operations

**#include <string.h>**

**char** ∗**strcat(char** ∗*s1*, **const char** ∗*s2*);

**char** ∗**strncat(char** ∗*s1*, **const char** ∗*s2*, **size_t** *n*);

**int strcmp(const char** ∗*s1*, **const char** ∗*s2*);

**int strncmp(const char** ∗*s1*, **const char** ∗*s2*, **size_t** *n*);

**char** ∗**strcpy(char** ∗*s1*, **const char** ∗*s2*);

**char** ∗**strncpy(char** ∗*s1*, **const char** ∗*s2*, **size_t** *n*);

**char** ∗**strdup(const char** ∗*s1*);

**size_t strlen(const char** ∗*s*);

**char** ∗**strchr(const char** ∗*s*, **int** *c*);

**char** ∗**strrchr(const char** ∗*s*, **int** *c*);

**char** ∗**strpbrk(const char** ∗*s1*, **const char** ∗*s2*);

**size_t strspn(const char** ∗*s1*, **const char** ∗*s2*);

**size_t strcspn(const char** ∗*s1*, **const char** ∗*s2*);

**char** ∗**strtok(char** ∗*s1*, **const char** ∗*s2*);

**char** ∗**strstr(const char** ∗*s1*, **const char** ∗*s2*);

The arguments *s*, *s1*, and *s2* point to strings (arrays of characters terminated by a null character). The functions **strcat**, **strncat**, **strcpy**, **strncpy**, and **strtok** alter *s1*. These functions do not check for overflow of the array pointed to by *s1*. **strcat** appends a copy of string *s2*, including the terminating null character, to the end of string *s1*. **strncat** appends at most *n* characters. Each returns a pointer to the null-terminated result. The initial character of *s2* overrides the null character at the end of *s1*. **strcmp** compares its arguments and returns an integer less than, equal to, or greater than 0, based upon whether *s1* is lexicographically less than, equal to, or greater than *s2*. **strncmp** makes the same comparison but looks at most *n* characters. Characters following a null character are not compared. **strcpy** copies string *s2* to *s1* including the terminating null character, stopping after the null character has been copied. **strncpy** copies exactly *n* characters, truncating *s2* or adding null characters to *s1* if necessary. The result will not be null-terminated if the length of *s2* is *n* or more. Each function returns *s1*. **strdup** returns a pointer to a new string which is a duplicate of the string pointed to by *s1*. The space for the new string is obtained using **malloc**(BA_OS). If the new string can not be created, a **NULL** pointer is returned. **strlen** returns the number of characters in *s*, not including the terminating null character. **strchr** (or **strrchr**) returns a pointer to the first (last) occurrence of *c* (converted to a **char**) in string *s*, or a **NULL** pointer if *c* does not occur in the string. The null character terminating a string is considered to be part of the string. **strpbrk** returns a pointer to the first occurrence in string *s1* of any character from string *s2*, or a **NULL** pointer if no character from *s2* exists in *s1*. **strspn** (or **strcspn**) returns the length of the initial segment of string *s1* which consists entirely of characters from (not from) string *s2*. **strtok** considers the string *s1* to consist of a sequence of zero or more text tokens separated by spans of one or more characters from the separator string *s2*. The first call (with pointer *s1* specified) returns a pointer to the first character of the first token, and will have written a null character into *s1* immediately following the returned token. The function keeps track of its position in the string between separate calls, so that subsequent calls (which must be made with the first argument a **NULL** pointer) will work through the string *s1* immediately following that token. In this way subsequent calls will work through the string *s1* until no tokens remain. The separator string *s2* may be different from call to call. When no token remains in *s1*, a **NULL** pointer is returned. **strstr** locates the first occurrence in string *s1* of the sequence of characters (excluding the terminating null character) in string *s2*. **strstr** returns a pointer to the located string, or a null pointer if the string is not found. If *s2* points to a string with zero length (that is, the string **""**), the function returns *s1*. **malloc**(BA_OS), **setlocale**(BA_OS), **strxfrm**(BA_LIB), Level 1. All of these functions assume the default locale ‘‘C.’’ For some locales, **strxfrm** should be applied to the strings before they are passed to the functions.

## NAME

**thr_continue** – continue the execution of a suspended thread

## SYNOPSIS

```
#include <thread.h>

int thr_continue(thread_t target_thread);
```

### Parameters

| | |
|---|---|
| *target_thread* | thread ID of the thread to be continued |

## DESCRIPTION

**thr_continue** makes *target_thread* runnable. *target_thread* is the ID of a thread previously suspended with **thr_suspend**(MT_LIB) or created suspended with **thr_create**(MT_LIB). If *target_thread* is not suspended, **thr_continue** will have no effect.

### Return Values

**thr_continue** returns zero for success and an error number for failure, as described below.

### Errors

If any of the following conditions occurs, **thr_continue** returns the corresponding value:

| | |
|---|---|
| **ESRCH** | *target_thread* cannot be found in the current process |

## SEE ALSO

**thr_create**(MT_LIB), **thr_suspend**(MT_LIB)

## LEVEL

Level 1

## NAME

**thr_create** – create a thread

## SYNOPSIS

```
#include <thread.h>

int thr_create(void *stack_address, size_t stack_size,
     void *(*start_routine)(void *arg), void *arg,
     long flags, thread_t *new_thread);
```

### Parameters

| | |
|---|---|
| *stack_address* | pointer to the base address for the new thread's stack |
| *stack_size* | size of the new thread's stack |
| *start_routine* | pointer to the function the new thread will execute |
| *arg* | pointer to the argument to *start_routine* |
| *flags* | attributes for the new thread |
| *new_thread* | pointer to the identifier for the new thread, set by **thr_create** |

## DESCRIPTION

**thr_create** creates a new thread in the current process. The new thread will execute the function specified by *start_routine* with the argument specified by *arg*. The new thread will be immediately runnable, unless it is created with the **THR_SUSPENDED** flag (see **flags Parameter** below).

### stack_address and stack_size Parameters

The new thread will run on the stack described by *stack_address* and *stack_size*. The stack can be explicitly allocated by the user or allocated automatically.

For an explicitly allocated stack:

Set *stack_address* to point to the user-allocated stack. *stack_address* is the base, or lowest, address.

Set *stack_size* to the size in bytes of the user-allocated stack. *stack_size* must be larger than the value returned by **thr_minstack**(MT_LIB).

For an automatically allocated stack:

Set *stack_address* to **NULL**.

To get a default size stack, set *stack_size* to zero.

To get a stack of a particular size, set *stack_size* to the number of bytes you want. *stack_size* must be larger than the value returned by **thr_minstack**(MT_LIB).

## arg Parameter

*arg* is a pointer to the argument to *start_routine*.

## flags Parameter

*flags* specifies attributes for the new thread. It is constructed from the bitwise inclusive OR of any of the following:

| | |
|---|---|
| **THR_SUSPENDED** | create the new thread in the suspended state. This permits modification of scheduling parameters and other attributes before the new thread executes *start_routine*. The creating thread or another thread must call **thr_continue** for the new thread to begin executing. |
| **THR_BOUND** | bind the new thread to a new lightweight process (LWP) created for the purpose, regardless of other flags. The thread will not be scheduled on other LWPs even if the implementation supports multiplexing of threads across LWPs. If the implementation does not support multiplexing, this flag is irrelevant. |
| **THR_DETACHED** | create the new thread in the detached state. The new thread cannot be awaited with **thr_join**(MT_LIB). This gives a hint to the implementation that immediate reuse of the new thread's resources on **thr_exit** is acceptable to the user. The exit status of a detached thread cannot be retrieved. |
| **THR_INCR_CONC** | increase the concurrency level as returned by **thr_getconcurrency** If the implementation does not support multiplexing, this flag is ignored. |
| **THR_DAEMON** | create the new thread as a daemon thread. The new thread will not be counted when determining when the last thread has terminated. The process will terminate when the last non-daemon thread terminates. Such ‘‘helper threads’’ may be created for asynchronous I/O or other activities, but do not prolong the life of the process when there are no real application threads remaining. |

If both **THR_BOUND** and **THR_INCR_CONC** are set, the concurrency level is increased by one and a new bound thread is created.

## new_thread Parameter

If *new_thread* is not **NULL**, **thr_create** sets the location pointed to by *new_thread* to the identifier for the created thread. The thread ID is only valid within the calling process.

## Signal Mask and Scheduling Characteristics

The newly created thread inherits the creating thread's signal mask, as established by **thr_sigsetmask**(MT_LIB), but without any pending signals.

Scheduling attributes are also inherited as appropriate and permitted.

## Security Restrictions

**thr_create** requires no special permissions or privilege.

### Return Values

**thr_create** returns zero for success and an error number for failure, as described below.

### Errors

If any of the following conditions occurs, **thr_create** returns the corresponding value:

| | |
|---|---|
| **ENOMEM** | Insufficient memory to complete **thr_create**. |
| **EINVAL** | *stack_size* is zero and *stack_address* is not **NULL**. |
| **EINVAL** | *stack_size*, after shrinking to aligned offsets, is smaller than an implementation-defined lower bound as returned by **thr_minstack**. |
| **EINVAL** | *start_routine* is **NULL**. |

If any of the following conditions is detected, **thr_create** returns the corresponding value:

| | |
|---|---|
| **EAGAIN** | A resource limit would be exceeded if the call succeeded. |
| **EFAULT** | One or more parameters point to an illegal address. (The likely result is a subsequent **SIGSEGV** rather than detecting an **EFAULT** condition.) |

## USAGE

### Examples

The following example creates a multiplexed thread that will use a library-allocated stack. Note that using 0 as the *flags* argument will create a thread that is multiplexed (not bound) and can be awaited with **thr_join**. The library will allocate a default size stack because *stack_address* is **NULL** and *stack_size* is 0.

```
void *
t_main(void *arg)
{
     ...
}
int error;
void *t_arg = NULL;
thread_t t1_id;

error = thr_create((void*)NULL, 0, t_main, t_arg, 0, &t1_id);
```

The following example creates a multiplexed thread that will use a user-allocated stack.

```
void *
t_main(void *arg)
{
     ...
}
int error;
void *t1_stack;
void *t_arg = NULL;
size_t t_size = 8192;
thread_t t1_id;
```

```
t1_stack = malloc(t_size);
error = thr_create(t1_stack, t_size, t_main, t_arg, 0, &t1_id);
```

The following example creates a daemon thread that will use a minimally-sized stack allocated by the library:

```
void *
t_main(void *arg)
{
     ...
}
int error;
void *t1_stack;
void *t_arg = NULL;
thread_t t1_id;

error = thr_create((void*)NULL, (size_t)thr_minstack(), t_main,
           t_arg, THR_DAEMON, &t1_id);
```

The following example creates a suspended, bound thread, then modifies its scheduling parameters before continuing. See **thr_setscheduler**(MT_LIB) for more details about modifying thread scheduling parameters. The library will allocate a default size stack because *stack_address* is **NULL** and *stack_size* is 0.

```
void *
t_main(void *arg)
{
     ...
}
int error;
void *t1_stack;
void *t_arg = NULL;
thread_t t1_id;
sched_param_t sched_param;

error = thr_create((void*)NULL, 0, t_main, t_arg,
           THR_SUSPENDED|THR_BOUND, &t1_id);

/* initialize sched_param to the SCHED_FIFO policy with priority 62 */
sched_param.policy = SCHED_FIFO;
((struct fifo_param *) sched_param.policy_params)->prio = 62;

error = thr_setscheduler(t1_id, &sched_param);
error = thr_continue(t1_id);
```

## SEE ALSO

**fork1**(MT_OS), **forkall**(MT_OS), **thr_continue**(MT_LIB), **thr_exit**(MT_LIB), **thr_getconcurrency**(MT_LIB), **thr_join**(MT_LIB), **thr_minstack**(MT_LIB), **thr_self**(MT_LIB), **thr_setconcurrency**(MT_LIB), **thr_setscheduler**(MT_LIB), **thr_suspend**(MT_LIB)

**LEVEL**

Level 1

## NAME

**thr_exit** – terminate execution of the calling thread

## SYNOPSIS

**#include <thread.h>**

**void thr_exit(void ***status***);**

### Parameters

| | |
|---|---|
| *status* | the exit value of the thread |

## DESCRIPTION

**thr_exit** terminates execution of the calling thread. *status* is the exit value of the terminating thread. The *status* will be returned to one of any sibling threads that call **thr_join**(MT_LIB).

If the start function of the thread [see **thr_create**(MT_LIB)] returns without calling **thr_exit**, **thr_exit** is called implicitly with *status* set to the return value of the function.

No error checking of *status* is done, as other values may be cast to **(void *)**. See Warnings in the USAGE section below.

After **thr_exit** has been called, all thread-specific data bindings are discarded [see **thr_keycreate**(MT_LIB)], and the thread data structures may be recycled.

**thr_exit** will terminate the process when it is called by the last thread not created with the **THR_DAEMON** flag. If a specific exit status is required for the process, **exit**(BA_OS) should be called explicitly. An implicit **exit** leaves the exit status of the process undefined.

A call to **thr_exit** by the initial thread does not terminate the process, unless it is the last non-daemon thread.

### Return Values

**thr_exit** does not return a value.

### Errors

None

## USAGE

### Warnings

For portability, use *status* only as a pointer; do not cast an **int** to **void *** to be used as the *status* argument, and then cast it back to **int** when it is retrieved by **thr_join**. The ANSI C standard does not require that implementations cast an **int** value to **(void *)** and then back to the initial type without losing information.

## SEE ALSO

**exit**(BA_OS), **thr_create**(MT_LIB), **thr_getspecific**(MT_LIB), **thr_join**(MT_LIB), **thr_keycreate**(MT_LIB), **thr_setspecific**(MT_LIB)

## LEVEL

Level 1

## NAME

**thr_get_rr_interval** – get the round-robin scheduling interval

## SYNOPSIS

```
#include <thread.h>

void thr_get_rr_interval(timestruc_t *rr_time);
```

### Parameters

*rr_time* pointer to the **timestruc_t** containing the value of the round-robin scheduling interval (set by **thr_get_rr_interval**)

## DESCRIPTION

**thr_get_rr_interval** stores the round-robin scheduling time quantum used by the Threads Library implementation for threads using the **SCHED_RR** scheduling policy in the **timestruc_t** pointed to by *rr_time*.

### rr_time Parameter

*rr_time* is set by **thr_get_rr_interval**. The user supplies a pointer to a **timestruc_t** object. **timestruc_t** is defined in **sys/time.h** to include the following members:

```
time_t  tv_sec;
long    tv_nsec;
```

### Security Restrictions

**thr_get_rr_interval** requires no special permissions or privilege.

### Return Values

**thr_get_rr_interval** does not return a value.

### Errors

None

## SEE ALSO

**thr_getscheduler**(MT_LIB), **thr_setscheduler**(MT_LIB)

## LEVEL

Level 1

## NAME

**thr_getconcurrency** – retrieve the level of concurrency

## SYNOPSIS

```
#include <thread.h>

int thr_getconcurrency(void);
```

## DESCRIPTION

**thr_getconcurrency** returns the level, or degree, of concurrency, which is the number of lightweight processes (LWPs)(see **intro**) the user has requested to be available for running multiplexed threads. This is the number from the most recent **thr_setconcurrency** call (or zero if there has been no call) plus the number of threads created with the **THR_INCR_CONC** flag set since the last call to **thr_setconcurrency**. The return value does not reflect the number of LWPs actually available.

### Return Values

**thr_getconcurrency** returns the degree of concurrency as described above.

### Errors

**thr_getconcurrency** cannot fail.

## SEE ALSO

**thr_create**(MT_LIB), **thr_setconcurrency**(MT_LIB)

## LEVEL

Level 1

## NAME

**thr_getprio** – retrieve a thread's scheduling priority

## SYNOPSIS

```
#include <thread.h>

int thr_getprio(thread_t tid, int *prio);
```

### Parameters

| | |
|---|---|
| *tid* | target thread ID |
| *prio* | pointer to priority value (set by **thr_getprio**) |

## DESCRIPTION

**thr_getprio** stores *tid*'s scheduling priority in the location pointed to by *prio*. **thr_getscheduler** can also be used to retrieve the priority of a thread, but **thr_getprio** is a shorthand for use when only the priority, not the scheduling class, is needed.

### Security Restrictions

No privileges or special permissions are required to use **thr_getprio**.

### Return Values

**thr_getprio** returns zero for success and an error number for failure, as described below.

### Errors

If any of the following error conditions is detected, **thr_getprio** returns the corresponding value:

| | |
|---|---|
| **ESRCH** | No thread with identifier *tid* can be found in the process. |

## SEE ALSO

**priocntl**(KE_OS), **thr_getscheduler**(MT_LIB), **thr_setprio**(MT_LIB), **thr_setscheduler**(MT_LIB), **thr_yield**(MT_LIB)

## LEVEL

Level 1

## NAME

**thr_getscheduler** – get the scheduling policy information for a thread

## SYNOPSIS

**#include <thread.h>**

**int thr_getscheduler(thread_t** *tid***, sched_param_t ****param***);**

### Parameters

| | |
|---|---|
| *tid* | a thread ID |
| *param* | a pointer to a **sched_param_t** structure containing the policy-specific parameters (set by **thr_getscheduler**) |

## DESCRIPTION

**thr_getscheduler** sets the **sched_param_t** pointed to by *param* to the policy-specific parameters for *tid*.

### tid Parameter

*tid* is the ID of the thread whose scheduling policy information **thr_getscheduler** will retrieve.

### param Parameter

*param* points to a **sched_param_t** structure in which **thr_getscheduler** will store *tid*'s scheduling policy and parameters.

**sched_param_t** is defined to contain the following members:

```
id_t  policy;
long  policy_params[POLICY_PARAM_SZ];
```

See **thr_setscheduler**(MT_LIB) for the available scheduling policies and parameters.

### Security Restrictions

No privileges or special permissions are required to use **thr_getscheduler**.

### Return Values

**thr_getscheduler** returns zero for success and an error number for failure, as described below.

### Errors

If the following condition is detected, **thr_getscheduler** fails and returns the value:

| | |
|---|---|
| **ESRCH** | No thread can be found in the current process with identity *tid*. |

## USAGE

**thr_getscheduler** is used by multithreaded applications which

## NAME

**thr_getspecific** – get thread-specific data

## SYNOPSIS

**#include <thread.h>**

**int thr_getspecific(thread_key_t** *key*, **void *****valuep***);**

### Parameters

| | |
|---|---|
| *key* | key whose value is to be returned |

## DESCRIPTION

**thr_getspecific** returns the value currently bound to the specified *key* on behalf of the calling thread.

### key Parameter

*key* is a key obtained with a previous call to **thr_keycreate**(MT_LIB).

The effect of calling **thr_getspecific** with a *key* value not obtained with **thr_keycreate** or after *key* has been deleted with **thr_keydelete** is undefined.

### valuep Parameters

**thr_getspecific** sets the location pointed to by *valuep* to the value set by a previous call of **thr_specific** in the calling thread. If no value is bound to the specified key for the calling thread, the location pointed to by value is set to **NULL**.

### Return Values

**thr_getspecific** returns zero for success and an error number for failure as described below.

### Errors

If the following condition is detected, **thr_getspecific** returns the value:

| | |
|---|---|
| **EINVAL** | The *key* is invalid. |

## SEE ALSO

**thr_keycreate**(MT_LIB), **thr_keydelete**(MT_LIB), **thr_setspecific**(MT_LIB)

## LEVEL

Level 1

## NAME

**thr_join** – join control paths with another thread

## SYNOPSIS

**#include <thread.h>**

**int thr_join(thread_t** *wait_for*, **thread_t ****departed_thread*, **void *****status***);**

### Parameters

| | |
|---|---|
| *wait_for* | the ID of the thread to wait for or 0 |
| *departed_thread* | pointer to the ID of the thread joined, set by **thr_join** |
| *status* | pointer to the joined thread's exit status |

## DESCRIPTION

**thr_join** waits for termination of the undetached sibling thread designated by *wait_for* and retrieves the exit status of the terminated thread [see **thr_exit**(MT_LIB)].

When the thread being waited for (*wait_for*) terminates, and **thr_join** returns, we say that the terminated thread has joined control paths with the thread that called **thr_join**. When a thread calls **thr_join**, we say that it is waiting for a thread.

### wait_for Parameter

If **wait_for** is set to the thread ID of an undetached sibling thread, **thr_join** waits for that thread to terminate. If *wait_for* is equal to **(thread_t)0**, **thr_join** waits for termination of any undetached sibling thread, and returns when the first one completes **thr_exit**.

Threads created with the **THR_DETACHED** flag cannot be joined.

### departed_thread Parameter

If the value of the pointer *departed_thread* is not **NULL**, **thr_join** sets the location pointed to by *departed_thread* to the identifier of the terminated sibling thread. This tells the calling thread which thread it joined.

### status Parameter

If *status* is not **NULL**, the location pointed to by *status* is set to the exit status from the terminated sibling thread.

Only one thread can successfully return from a **thr_join** for a given *departed_thread*. If more than one thread is waiting for a specific thread to terminate, one is successful and the others fail with error indication **ESRCH**

### Errors

If any of the following conditions occurs, **thr_join** returns the corresponding value:

**ESRCH** there is no joinable (undetached) thread in the current process with thread ID *wait_for*.

**EDEADLK** *wait_for* is the calling thread's thread ID.

If the following condition is detected, **thr_join** returns the value:

**EINVAL** There are no remaining threads that can be *thr_join*ed with, for example, all other threads are detached and *waitfor* is **(thread_t) 0**.

## SEE ALSO

**condition**(MT_LIB), **thr_create**(MT_LIB), **thr_exit**(MT_LIB), **wait**(BA_OS)

## LEVEL

Level 1

## NAME

**thr_keycreate** – create thread-specific data key

## SYNOPSIS

**#include <thread.h>**

**int thr_keycreate(thread_key_t** ***key*****, void (*****destructor*****)(void** ***value*****));**

### Parameters

| | |
|---|---|
| *key* | pointer to the new thread-specific data key (set by **thr_setspecific**) |
| *destructor* | pointer to function to be called at thread exit, or **NULL** |

## DESCRIPTION

**thr_keycreate** creates a key visible to all threads in the process. The key plays the role of identifier for per-thread data. A thread can then bind a value to *key* with **thr_setspecific**(3thread). Although the same key identifier may be used by different threads, the values bound to the key are maintained on a per-thread basis and persist for the life of the calling thread, or until explicitly replaced.

**thr_keycreate** sets the initial value of the key in all active and subsequently created threads to **NULL**. When **thr_keycreate** returns successfully the new key is stored in the location pointed to by *key*. The caller must ensure that creation and use of this key are synchronized [see **synch**(MT_LIB)].

Normally, the value bound to a key by a thread will be a pointer to dynamically-allocated storage. When a thread terminates, per-thread context is automatically destroyed and, if a binding exists, the reference to the key is released. If the key has a destructor (see below), the destructor is called with the bound value.

There is no fixed limit on the number of keys per process.

### key Parameter

*key* points to the **thread_key_t** in which **thr_keycreate** will store the newly created key.

### destructor Parameter

*destructor* points to an optional destructor function to be associated with *key*. *destructor* can also be **NULL**. When a thread terminates, if it has a non-**NULL** destructor function and a non-**NULL** value associated with *key*, the destructor function will be called with the bound value as an argument. If the value associated with *key* is **NULL**, the destructor is not called. Destructors are intended to free any dynamically-allocated storage associated with the bound value.

If destructor functions call **thr_setspecific** or **thr_getspecific**, it may not be possible to destroy all bindings for a terminating thread. The order in which the destructor functions are called is unspecified.

### Return Values

**thr_keycreate** returns zero for success and an error number for failure, as described below.

### Errors

If any of the following conditions is detected, **thr_keycreate** returns the corresponding value:

| | |
|---|---|
| **EAGAIN** | The key name space is exhausted. |
| **ENOMEM** | Insufficient memory exists to create the key. |

## USAGE

### Examples

This example shows the use of thread-specific data in a function that can be called from more than one thread without special initialization. For the sake of simplicity, no error checking is done.

```
static mutex_t keylock;
static thread_key_t key;
static int once = 0;

void
func()
{
     void *ptr;

     (void) mutex_lock(&keylock);
     if (!once) {
          (void) thr_keycreate(&key, free);
          once++;
     }
     (void) mutex_unlock(&keylock);
     (void) thr_getspecific(key, (void *) &ptr);
     if (ptr == NULL) {
          ptr = malloc(SIZE);
          (void) thr_setspecific(key, ptr);
     }
}
```

## SEE ALSO

**mutex**(MT_LIB), **thr_exit**(MT_LIB), **thr_getspecific**(MT_LIB), **thr_keydelete**(MT_LIB), **thr_setspecific**(MT_LIB)

## LEVEL

Level 1

## NAME

**thr_keydelete** – thread-specific data key

## SYNOPSIS

```
#include <thread.h>

int thr_keydelete(thread_key_t key)
```

### Parameters

| | |
|---|---|
| *key* | the key to be deleted |

## DESCRIPTION

**thr_keydelete** deletes the specified *key*, which was obtained from a previous call to **thr_keycreate**.

### key Parameter

*key* is the key to be deleted. *key* must no longer be in use, that is, no thread may have a non-**NULL** value bound to *key*, otherwise **thr_keydelete** will return **EBUSY**.

### Return Values

**thr_keydelete** returns zero for success and an error number for failure, as described below.

### Errors

If any of the following conditions occur, **thr_keydelete** returns the corresponding value:

| | |
|---|---|
| **EBUSY** | *key* has thread-specific data associated with it. |
| **EINVAL** | *key* is invalid. |

## USAGE

A typical use would be for a dynamically linked library to create its private key with **thr_keycreate** as part of its initialization, use **thr_getspecific** and **thr_setspecific** while in use, and then call **thr_keydelete** before unlinking.

The application should ensure that other thread-specific data functions for *key* are not called concurrently with **thr_keydelete**.

## SEE ALSO

**thr_getspecific**(MT_LIB), **thr_keycreate**(MT_LIB),

**thr_setspecific**(MT_LIB)

## LEVEL

Level 1

## NAME

**thr_kill** – send a signal to a sibling thread

## SYNOPSIS

**#include <thread.h>**

**int thr_kill(thread_t** *tid***, int** *sig***)**

### Parameters

| | |
|---|---|
| *tid* | thread ID of the thread to receive the signal |
| *sig* | signal number of the signal to be sent |

## DESCRIPTION

**thr_kill** sends the signal *sig* to the sibling thread *tid*.

If *tid* is blocking signal *sig*, the signal will become pending for *tid*.

**thr_kill** is the thread analog of **kill**(BA_OS).

### tid Parameter

*tid* is the thread ID of the sibling thread which is to receive the signal. A thread cannot send a signal to a thread in another process (a non-sibling thread).

### sig Parameter

*sig* is the signal number of the signal to be sent and is either 0 or a value from the list given in **signal**(BA_ENV).

If *sig* is 0 (the null signal), error checking is performed but no signal is actually sent; this can be used to check the validity of *tid*.

### Return Values

**thr_kill** returns zero for success and an error number for failure, as described below.

### Errors

If any of the following conditions occurs, **thr_kill** returns the corresponding value:

| | |
|---|---|
| **EINVAL** | *sig* is not a valid signal number |
| **ESRCH** | *tid* cannot be found in the current process |

## SEE ALSO

**kill**(BA_OS), **signal**(BA_ENV), **sigwait**(BA_OS), **thr_sigsetmask**(MT_LIB)

## LEVEL

Level 1

## NAME

**thr_minstack** – return the minimum stack size for a thread

## SYNOPSIS

```
#include <thread.h>

size_t thr_minstack(void);
```

## DESCRIPTION

**thr_minstack** returns the implementation-defined value for the minimum stack size for a thread required by **thr_create**(MT_LIB).

### Return Values

**thr_minstack** returns the minimum stack size for a thread.

### Errors

None

## USAGE

The value returned by **thr_minstack** can be used as the *stack_size* argument to **thr_create** when the new thread needs only a minimal stack. Threads that will call any functions, or that need much space for local variables, must use a larger stack.

## SEE ALSO

**thr_create**(MT_LIB)

## LEVEL

Level 1

## NAME

**thr_self** – get thread identifier of the calling thread

## SYNOPSIS

```
#include <thread.h>

thread_t thr_self(void);
```

## DESCRIPTION

**thr_self** returns the identifier of the calling thread.

### Return Values

**thr_self** returns the identifier of the calling thread.

### Errors

None. This function always succeeds.

## SEE ALSO

**getuid**(BA_OS), **thr_create**(MT_LIB)

## LEVEL

Level 1

## NAME

**thr_setconcurrency** – request a level of concurrency

## SYNOPSIS

```
#include <thread.h>

int thr_setconcurrency(int new_level);
```

### Parameters

| | |
|---|---|
| *new_level* | the requested level of concurrency |

## DESCRIPTION

**thr_setconcurrency** tells the implementation the number of implementation supported (see **intro**) lightweight processes (LWPs) that the user would like available for running multiplexed threads.

**thr_setconcurrency** sets to *new_level* the requested level, or degree, of concurrency, which is the number of LWPs that the user would like available to execute multiplexed threads in the process. The requested concurrency is a hint to the implementation as to the level of concurrency expected by the user; the implementation may use this value to affect the number of LWPs available for running multiplexed threads.

### new_level Parameter

*new_level* must be a non-negative integer. **thr_setconcurrency** interprets it as follows:

> If *new_level* is zero, **thr_setconcurrency** sets the level of concurrency to the default level.
>
> If *new_level* is greater than the current number of LWPs, **thr_setconcurrency** may create LWPs until the number of LWPs in the pool equals *new_level*.
>
> If *new_level* is less than the current number of LWPs, **thr_setconcurrency** may release LWPs until the number of LWPs in the pool equals *new_level*.

When the number of LWPs becomes greater than the number of threads in the process, the concurrency level may automatically decay over time to be equal to or less than the number of threads in the process.

### Return Values

**thr_setconcurrency** returns zero for success and an error number for failure, as described below.

### Errors

If any of the following conditions is detected, **thr_setconcurrency** returns the corresponding value:

| | |
|---|---|
| **EINVAL** | *new_level* is negative |
| **EAGAIN** | A system resource limit would have been exceeded by using the requested value in the implementation's concurrency algorithm. LWPs created up to the failed **_lwp_create** will not be killed but will continue to exist. Because the change of concurrency level is not necessarily synchronous with the call to **thr_setconcurrency**, this condition is not always detected. |

## NOTICES

The creation or termination of LWPs is not necessarily synchronous with the call to **thr_setconcurrency**, therefore an error may not be returned. For example, if *new_level* exceeds a system limit, **EAGAIN** may not be returned, and any LWPs created asynchronously as a result of the call will not be terminated.

The Threads Library will always ensure that an LWP is available to run multiplexed threads.

## USAGE

The Threads Library ensures that a sufficient number of threads are active so that the process can continue to make progress. While this conserves system resources, it may not produce the most effective level of concurrency. **thr_setconcurrency** permits the application to give the Threads Library a hint about the desired level of concurrency.

## SEE ALSO

**thr_create**(MT_LIB), **thr_getconcurrency**(MT_LIB)

## LEVEL

Level 1

## NAME

**`thr_setprio`** – set a thread's scheduling priority

## SYNOPSIS

**`#include <thread.h>`**

**`int thr_setprio(thread_t`** *tid*, **`int`** *prio*);

### Parameters

| | |
|---|---|
| *tid* | target thread ID |
| *prio* | priority value for *tid* |

## DESCRIPTION

**`thr_setprio`** sets *tid*'s scheduling priority to be *prio*. **`thr_setscheduler`** can also be used to set the priority of a thread, but **`thr_setprio`** is a shorthand for use when only the priority, not the scheduling class, needs to be changed.

### Priority Range for Multiplexed Threads

The priority range for the **SCHED_TS** policy for multiplexed threads is implementation-specific.

### Priority Range for Bound Threads

Bound threads running under any scheduling policy are subject to the priority ranges set by the system. Use **`priocntl`**() or **`priocntl`**(KE_OS) to find what scheduling priorities are available on your system.

### Security Restrictions

No privileges or special permissions are required to use **`thr_setprio`** to set the priority of a multiplexed thread. The following rules apply to changing the priority of bound threads:

You can always lower the priority of any bound thread.

You can always raise the priority of bound threads in the **SCHED_FIFO** and **SCHED_RR** classes.

You must have privilege to raise the priority of a bound thread in the **SCHED_TS** class. The required privileges may vary across installations.

### Return Values

**`thr_setprio`** returns zero for success and an error number for failure, as described below.

### Errors

If any of the following error conditions is detected, **`thr_setprio`** fails and returns the corresponding value:

| | |
|---|---|
| **EINVAL** | The value of **`prio`** is invalid for *tid*'s current scheduling policy. |
| **EPERM** | The caller does not have appropriate privilege to set the priority of *tid*. |
| **ESRCH** | No thread with identifier *tid* can be found in the process. |

## SEE ALSO

**`priocntl`**(KE_OS), **`thr_getprio`**(MT_LIB), **`thr_getscheduler`**(MT_LIB), **`thr_setscheduler`**(MT_LIB), **`thr_yield`**(MT_LIB)

**LEVEL**

Level 1

## NAME

**thr_setscheduler** – set the scheduling policy for a thread

## SYNOPSIS

**#include <thread.h>**

**int thr_setscheduler(thread_t** *tid***, const sched_param_t ****param***);**

### Parameters

| | |
|---|---|
| *tid* | target thread ID |
| *param* | pointer to a structure containing scheduling policy parameters to be used |

## DESCRIPTION

**thr_setscheduler** sets the scheduling policy and corresponding policy-specific parameters of *tid* to those specified by the **sched_param_t** structure pointed to by *param*.

### tid Parameter

*tid* is the ID of the thread whose scheduling policy **thr_setscheduler** will set.

### param Parameter

*param* is a pointer to a **sched_param_t** structure that has been initialized to contain the scheduling policy and corresponding parameters to which *tid* will be set. The priority of the thread is the only corresponding parameter that can be set.

**sched_param_t** includes:

```
id_t  policy;
long  policy_params[POLICY_PARAM_SZ];
```

The **policy** member of **sched_param_t** can be set to any of the following scheduling policies:

| | |
|---|---|
| **SCHED_TS** | time-sharing scheduling policy. Both bound and multiplexed threads can use **SCHED_TS**. |
| **SCHED_FIFO** | a fixed-priority scheduling policy. Only bound threads can use **SCHED_FIFO**. Threads scheduled under this policy will run on an LWP in the kernel fixed-priority scheduling class with an infinite time quantum. |
| **SCHED_RR** | a fixed-priority scheduling policy. Only bound threads can use **SCHED_RR**. Threads scheduled under this policy will run on an LWP in the kernel fixed-priority scheduling class with the time quantum returned by **thr_get_rr_interval**(MT_LIB). |
| **SCHED_OTHER** | an alias for **SCHED_TS**. |

**policy_params** contains the priority to which the thread should be assigned. **policy_params** can be cast to a pointer to the parameter structure corresponding to the scheduling policy. Each of the parameter structures contains the integer **prio**. The parameter structures are:

| | |
|---|---|
| **ts_param** | for time-sharing scheduling parameters |

| | |
|---|---|
| **fifo_param** | for FIFO scheduling parameters |
| **rr_param** | for round-robin scheduling parameters |

### Priority Range for Multiplexed Threads

The priority range for the **SCHED_TS** policy for multiplexed threads is implementation-specific.

### Priority Range for Bound Threads

Bound threads running under any scheduling policy are subject to the priority ranges set by the system. Use **priocntl**(AU_CMD) or **priocntl**(KE_OS) to find what scheduling priorities are available on your system.

### Security Restrictions

No privileges or special permissions are required to use **thr_setscheduler** to set the policy or priority of a multiplexed thread. Appropriate privilege is required to set the policy of any thread or process to **SCHED_FIFO** or **SCHED_RR**. The following rules apply to changing the priority of bound threads:

You can always lower the priority of any bound thread.

You can always raise the priority of bound threads in the **SCHED_FIFO** and **SCHED_RR** classes.

You must have privilege to raise the priority of a bound thread in the **SCHED_TS** class. The required privileges may vary across installations.

### Notes to the User

Note that each multiplexed thread run by a lightweight process (LWP) will affect the priority of that LWP in the system scheduler. Over time, there will be approximate balance across the multiplexed threads in a process.

### Return Values

**thr_setscheduler** returns zero for success and an error number for failure, as described below.

### Errors

If any of the following conditions is detected, **thr_setscheduler** returns the corresponding value:

| | |
|---|---|
| **EINVAL** | *param* points to a structure containing parameters that are invalid for the requested policy. |
| **ENOSYS** | *tid* is multiplexed (not bound to an LWP), and the scheduling policy being set is not supported for multiplexed threads. (In general, only **SCHED_TS** can be counted on to work with multiplexed threads.) |
| **EPERM** | The caller does not have appropriate privilege for the operation. |
| **ESRCH** | No thread can be found in the current process with ID *tid*. |

## USAGE

**thr_setscheduler** is used by multithreaded applications that need to control their scheduling.

### Example of Setting sched_param_t

```
sched_param_t s;
s.policy = SCHED_TS;
(struct ts_param *) (s.policy_params)->prio = 63;
```

## SEE ALSO

**priocntl**(AU_CMD), **priocntl**(KE_OS), **thr_create**(MT_LIB), **thr_getprio**(MT_LIB), **thr_getscheduler**(MT_LIB), **thr_setprio**(MT_LIB), **thr_yield**(MT_LIB)

## LEVEL

Level 1

## NAME

**thr_setspecific** – set thread-specific data

## SYNOPSIS

**#include <thread.h>**

**int thr_setspecific(thread_key_t** *key***, void ****value***);**

### Parameters

| | |
|---|---|
| *key* | key to which *value* is to be bound |
| *value* | pointer to thread-specific data, or **NULL** |

## DESCRIPTION

**thr_setspecific** associates a thread-specific *value* with *key*. Different threads may bind different values to the same key.

If the value bound to *key* must be updated during the lifetime of the thread, the caller must release the storage associated with the old value before a new value is bound, or the storage is lost.

### key Parameter

*key* is a key obtained with a previous call to **thr_keycreate**(MT_LIB).

The effect of calling **thr_setspecific** with a *key* value not obtained with **thr_keycreate** or after *key* has been deleted with **thr_keydelete** is undefined.

### value Parameter

*value* is typically a pointer to blocks of dynamically-allocated memory that have been reserved for use by the calling thread. If *value* is **NULL**, the calling thread will give up a non-**NULL** reference to *key*.

### Return Values

**thr_setspecific** returns zero for success and an error number for failure, as described below.

### Errors

If any of the following conditions is detected, **thr_setspecific** returns the corresponding value:

| | |
|---|---|
| **EINVAL** | The key value is invalid. |
| **ENOMEM** | There is not sufficient memory available to establish the binding. |

## SEE ALSO

**thr_getspecific**(MT_LIB), **thr_keycreate**(MT_LIB), **thr_keydelete**(MT_LIB)

## LEVEL

Level 1

## NAME

**thr_sigsetmask** – change or examine the signal mask of a thread

## SYNOPSIS

**#include <thread.h>**

**int thr_sigsetmask(int** *how*, **const sigset_t ****set*, **sigset_t ****oset***);**

### Parameters

| | |
|---|---|
| *how* | **SIG_BLOCK**, **SIG_UNBLOCK**, or **SIG_SETMASK** |
| *set* | pointer to a set of signals to be blocked or unblocked |
| *oset* | pointer to the value of the previous signal mask (set by **thr_sigsetmask**) |

## DESCRIPTION

**thr_sigsetmask** changes or examines the calling thread's signal mask according to the way *how* and *set* are set.

If there are any pending unblocked signals after the call to **thr_sigsetmask**, at least one of those signals will be delivered before the call to **thr_sigsetmask** returns.

The Threads Library implementation may affect the signal mask of the lightweight process (LWP) running the calling thread.

### how Parameter

*how* determines how *set* is interpreted. Set *how* to one of the following values:

| | |
|---|---|
| **SIG_BLOCK** | Add the set of signals denoted by *set* to the current signal mask. |
| **SIG_UNBLOCK** | Remove the set of signals denoted by *set* from the current signal mask. |
| **SIG_SETMASK** | Replace the current signal mask with the set of signals denoted by *set*. |

Note that 0 is not a valid value for *how*.

### set Parameter

*set* points to a set of signals to be blocked or unblocked (according to the value of *how*) in the current thread. If *set* is **NULL**, the value of *how* is not significant, and the thread's signal mask will not be changed.

It is not possible to block those signals that cannot be ignored [see **sigaction**(BA_OS)]; this restriction is silently imposed by the system.

### oset Parameter

If *oset* is not **NULL**, **thr_sigsetmask** stores the value of the previous mask in that location. If *set* is **NULL** and *oset* is not **NULL**, *oset* will point to the value of the thread's current signal mask.

### Return Values

**thr_sigsetmask** returns zero for success and an error number for failure, as described below.

If **thr_sigsetmask** fails, the thread's signal mask is not changed.

### Errors

If the following condition occurs, **thr_sigsetmask** returns the value:

**EINVAL** The value of *how* is not equal to one of the defined values.

## USAGE

### Portability Considerations

Threads should use **thr_sigsetmask** rather than **sigprocmask**(BA_OS). In some implementations, **sigprocmask** may be trapped and behaves identically to **thr_sigsetmask**, but for portability, **thr_sigsetmask** should be used.

### Style Considerations

The preferred coding style is to mask all signals and use **sigwait** synchronously.

## SEE ALSO

**kill**(BA_OS), **sigaction**(BA_OS), **signal**(BA_ENV), **sigprocmask**(BA_OS), **sigsend**(BA_OS), **sigwait**(BA_OS), **thr_kill**(MT_LIB)

## LEVEL

Level 1

## NAME

**thr_suspend** – suspend the execution of a thread

## SYNOPSIS

**#include <thread.h>**

**int thr_suspend(thread_t** *target_thread***);**

### Parameters

| | |
|---|---|
| *target_thread* | thread ID of the thread to be suspended |

## DESCRIPTION

**thr_suspend** suspends execution of *target_thread*. **thr_suspend** will return no sooner than the time at which the implementation has started suspending *target_thread*. The critical point at which **thr_suspend** can return is determined purely by the implementation. A concurrent **thr_continue** of the same thread may be lost or may take effect depending on the timing.

If *target_thread* is already suspended, **thr_suspend** has no effect.

A thread may suspend itself.

**thr_continue** will resume the execution of *target_thread*.

### Return Values

**thr_suspend** returns zero for success and an error number for failure, as described below.

### Errors

If any of the following conditions occurs, **thr_suspend** returns the corresponding value:

| | |
|---|---|
| **ESRCH** | *target_thread* cannot be found in the current process |

## USAGE

We don't recommend using **thr_suspend** and **thr_continue** to synchronize threads. Use synchronization routines instead.

## SEE ALSO

**thr_continue**(MT_LIB), **thr_create**(MT_LIB)

## LEVEL

Level 1

## NAME

**thr_yield** – yield the processor

## SYNOPSIS

```
#include <thread.h>

void thr_yield(void);
```

## DESCRIPTION

**thr_yield()** causes the calling thread to stop executing to allow another eligible thread (if any) to run. The calling thread will remain in a runnable state.

The criteria for choosing a thread to run after the calling thread has yielded are not specified. It is possible for the calling thread to be rescheduled immediately, even if other runnable threads exist.

This function should be viewed as a hint from the caller to the system, indicating that the caller has reached a point at which it is convenient to yield the processor to other threads.

### Security Restrictions

**thr_yield** requires no special permissions or privilege.

### Return Values

None

### Errors

None.

## USAGE

**thr_yield()** is used by multithreaded applications which need to control their scheduling.

## SEE ALSO

**priocntl**(KE_OS), **thr_getscheduler**(MT_LIB), **thr_getprio**(MT_LIB), **thr_setprio**(MT_LIB), **thr_setscheduler**(MT_LIB)

## LEVEL

Level 1

**ttyname**, **isatty** – find name of a terminal

**#include <stdlib.h>**

**char *ttyname(int** *fildes***);**

**int isatty(int** *fildes***);**

**ttyname** returns a pointer to a string containing the null-terminated path name of the terminal device associated with file descriptor *fildes*. **isatty** returns 1 if *fildes* is associated with a terminal device, 0 otherwise.