The

The Pthr list of
the funct

This sect ed
accordin ns. The
followin

- Creating a Default Thread
- Waiting for Thread Termination
- Simple Threads Example
- Detaching a Thread
- Creating a Key for Thread-Specific Data
- Deleting the Thread-Specific Data Key
- Setting Thread-Specific Data
- Getting Thread-Specific Data
- Global and Private Thread-Specific Data Example
- Getting the Thread Identifier
- Comparing Thread IDs
- Calling an Initialization Routine for a Thread
- Yielding Thread Execution
- Setting the Thread Policy and Scheduling Parameters
- Getting the Thread Policy and Scheduling Parameters
- Setting the Thread Priority
- Sending a Signal to a Thread
- Accessing the Signal Mask of the Calling Thread

Creating a Default Thread

When an attribute object is not specified, the object is `NULL`, and the default thread is created with the following attributes:

- Process scope
- Nondetached
- A default stack and stack size
- A priority of zero

You can also create a default attribute object with **`pthread_attr_init()`** , and then use this attribute object to create a default thread. See the section Initializing Attributes for details.

`pthread_create` Syntax

Use pthread_create(3C) to add a new thread of control to the current process.

```
int pthread_create(pthread_t *restrict tid, const pthread_attr_t
    *restrict tattr, void*(*start_routine)(void *), void *restrict arg);
```

```
#include <pthread.h>

pthread_attr_t() tattr;
pthread_t tid;
extern void *start_routine(void *arg);
void *arg;
int ret;

/* default behavior*/
ret = pthread_create(&tid, NULL, start_routine, arg);

/* initialized with default attributes */
ret = pthread_attr_init(&tattr);
/* default behavior specified*/
ret = pthread_create(&tid, &tattr, start_routine, arg);
```

The **pthread_create()** function is called with *attr* that has the necessary state behavior. *start_routine* is the function with which the new thread begins execution. When *start_routine* returns, the thread exits with the exit status set to the value returned by *start_routine*. See pthread_create Syntax.

When **pthread_create()** is successful, the ID of the created thread is stored in the location referred to as *tid*.

When you call **pthread_create()** with either a NULL attribute argument or a default attribute, **pthread_create()** creates a default thread. When *tattr* is initialized, the thread acquires the default behavior.

pthread_create Return Values

pthread_create() returns zero when the call completes successfully. Any other return value indicates that an error occurred. When any of the following conditions are detected, **pthread_create()** fails and returns the corresponding value.

EAGAIN

Description:

A system limit is exceeded, such as when too many threads have been created.

EINVAL

Description:

The value of *tattr* is invalid.

EPERM

Description:

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

Waiting for Thread Termination

The **pthread_join()** function blocks the calling thread until the specified thread terminates.

pthread_join Syntax

Use pthread_join(3C) to wait for a thread to terminate.

```
int pthread_join(pthread_t tid, void **status);

#include <pthread.h>

pthread_t tid;
int ret;
void *status;
```

```
/* waiting to join thread "tid" with status */
ret = pthread_join(tid, &status);

/* waiting to join thread "tid" without status */
ret = pthread_join(tid, NULL);
```

The specified thread must be in the current process and must not be detached. For information on thread detachment, see Setting Detach State.

When **status** is not `NULL`, **status** points to a location that is set to the exit status of the terminated thread when **`pthread_join()`** returns successfully.

If multiple threads wait for the same thread to terminate, all the threads wait until the target thread terminates. Then one waiting thread returns successfully. The other waiting threads fail with an error of `ESRCH`.

After **`pthread_join()`** returns, any data storage associated with the terminated thread can be reclaimed by the application.

`pthread_join` Return Values

`pthread_join()` returns zero when the call completes successfully. Any other return value indicates that an error occurred. When any of the following conditions are detected, **`pthread_join()`** fails and returns the corresponding value.

`ESRCH`

Description:

No thread could be found corresponding to the given thread ID.

`EDEADLK`

Description:

A deadlock would exist, such as a thread waits for itself or thread A waits for thread B and thread B waits for thread A.

`EINVAL`

Description:

The thread corresponding to the given thread ID is a detached thread.

`pthread_join()` works only for target threads that are nondetached. When no reason exists to synchronize with the termination of a particular thread, then that thread should be detached.

Simple Threads Example

In Example 2–1, one thread executes the procedure at the top, creating a helper thread that executes the procedure **`fetch()`**. The **`fetch()`** procedure executes a complicated database lookup and might take some

time.

The main thread awaits the results of the lookup but has other work to do in the meantime. So, the main thread perform those other activities and then waits for its helper to complete its job by executing `pthread_join()`.

An argument, *pbe*, to the new thread is passed as a stack parameter. The thread argument can be passed as a stack parameter because the main thread waits for the spun-off thread to terminate. However, the preferred method is to use `malloc` to allocate storage from the heap instead of passing an address to thread stack storage. If the argument is passed as an address to thread stack storage, this address might be invalid or be reassigned if the thread terminates.

Example 2–1 Simple Threads Program

```
void mainline (...)
{
        struct phonebookentry *pbe;
        pthread_attr_t tattr;
        pthread_t helper;
        void *status;

        pthread_create(&helper, NULL, fetch, &pbe);

            /* do something else for a while */

        pthread_join(helper, &status);
        /* it's now safe to use result */
}

void *fetch(struct phonebookentry *arg)
{
        struct phonebookentry *npbe;
        /* fetch value from a database */

        npbe = search (prog_name)
            if (npbe != NULL)
                *arg = *npbe;
        pthread_exit(0);
}

struct phonebookentry {
        char name[64];
        char phonenumber[32];
        char flags[16];
}
```

Detaching a Thread

pthread_detach(3C) is an alternative to pthread_join(3C) to reclaim storage for a thread that is created with a *detachstate* attribute set to `PTHREAD_CREATE_JOINABLE`.

`pthread_detach` Syntax

```
int pthread_detach(pthread_t tid);
```

```
#include <pthread.h>

pthread_t tid;
int ret;

/* detach thread tid */
ret = pthread_detach(tid);
```

The **pthread_detach()** function is used to indicate to your application that storage for the thread **tid** can be reclaimed when the thread terminates. Threads should be detached when they are no longer needed. If **tid** has not terminated, **pthread_detach()** does not cause the thread to terminate.

pthread_detach Return Values

pthread_detach() returns zero when the call completes successfully. Any other return value indicates that an error occurred. When any of the following conditions is detected, **pthread_detach()** fails and returns the corresponding value.

EINVAL

Description:

tid is a detached thread.

ESRCH

Description:

tid is not a valid, undetached thread in the current process.

Creating a Key for Thread-Specific Data

Single-threaded C programs have two basic classes of data: local data and global data. For multithreaded C programs a third class, **thread-specific data**, is added. Thread-specific data is very much like global data, except that the data is private to a thread.

Note –

The Solaris OS supports an alternative facility that allows a thread to have a private copy of a global variable. This mechanism is referred to as **thread local storage** (TLS). The keyword `__thread` is used to declare variables to be thread-local, and the compiler automatically arranges for these variables to be allocated on a per-thread basis. See Chapter 8, *Thread-Local Storage,* in *Linker and Libraries Guide* for more information.

Thread-specific data (TSD) is maintained on a per-thread basis. TSD is the only way to define and refer to data that is private to a thread. Each thread-specific data item is associated with a *key* that is global to all threads in the process. By using the *key*, a thread can access a pointer (*void* *) maintained per-thread.

pthread_key_create Syntax

```
int pthread_key_create(pthread_key_t *key,
    void (*destructor) (void *));

#include <pthread.h>

pthread_key_t key;
int ret;

/* key create without destructor */
ret = pthread_key_create(&key, NULL);

/* key create with destructor */
ret = pthread_key_create(&key, destructor);
```

Use pthread_key_create(3C) to allocate a *key* that is used to identify thread-specific data in a process. The key is global to all threads in the process. When the thread-specific data is created, all threads initially have the value `NULL` associated with the key.

Call **`pthread_key_create()`** once for each key before using the key. No implicit synchronization exists for the keys shared by all threads in a process.

Once a key has been created, each thread can bind a value to the key. The values are specific to the threads and are maintained for each thread independently. The per-thread binding is deallocated when a thread terminates if the key was created with a `destructor` function.

When **`pthread_key_create()`** returns successfully, the allocated key is stored in the location pointed to by *key*. The caller must ensure that the storage and access to this key are properly synchronized.

An optional destructor function, `destructor`, can be used to free stale storage. If a key has a non-`NULL` `destructor` function and the thread has a non-`NULL` value associated with that key, the `destructor` function is called with the current associated value when the thread exits. The order in which the `destructor` functions are called is unspecified.

`pthread_key_create` Return Values

`pthread_key_create()` returns zero after completing successfully. Any other return value indicates that an error occurred. When any of the following conditions occur, **`pthread_key_create()`** fails and returns the corresponding value.

`EAGAIN`

Description:

The *key* name space is exhausted.

`ENOMEM`

Description:

Insufficient virtual memory is available in this process to create a new key.

Deleting the Thread-Specific Data Key

Use pthread_key_delete(3C) to destroy an existing thread-specific data key. Any memory associated with the key can be freed because the key has been invalidated. Reference to an invalid key returns an error.

`pthread_key_delete` Syntax

```
int pthread_key_delete(pthread_key_t key);

#include <pthread.h>

pthread_key_t key;
int ret;

/* key previously created */
ret = pthread_key_delete(key);
```

If a *key* has been deleted, any reference to the key with the **`pthread_setspecific()`** or **`pthread_getspecific()`** call yields undefined results.

The programmer must free any thread-specific resources before calling the **`pthread_key_delete()`** function. This function does not invoke any of the destructors. Repeated calls to **`pthread_key_create()`** and **`pthread_key_delete()`** can cause a problem.

The problem occurs because, in the Solaris implementation, a *key* value is never reused after **`pthread_key_delete()`** marks it as invalid. Every **`pthread_key_create()`** allocates a new key value and allocates more internal memory to hold the key information. An infinite loop of **`pthread_key_create()`** ... **`pthread_key_delete()`** will eventually exhaust all memory. If possible, call **`pthread_key_create()`** only once for each desired key and never call **`pthread_key_delete()`**.

`pthread_key_delete` Return Values

`pthread_key_delete()` returns zero after completing successfully. Any other return value indicates that an error occurred. When the following condition occurs, **`pthread_key_delete()`** fails and returns the corresponding value.

`EINVAL`

Description:

The *key* value is invalid.

Setting Thread-Specific Data

Use pthread_setspecific(3C) to set the thread-specific binding to the specified thread-specific data key.

`pthread_setspecific` Syntax

```
int pthread_setspecific(pthread_key_t key, const void *value);

#include <pthread.h>

pthread_key_t key;
void *value;
int ret;
```

```
/* key previously created */
ret = pthread_setspecific(key, value);
```

`pthread_setspecific` Return Values

`pthread_setspecific()` returns zero after completing successfully. Any other return value indicates that an error occurred. When any of the following conditions occur, **`pthread_setspecific()`** fails and returns the corresponding value.

`ENOMEM`

Description:

Insufficient virtual memory is available.

`EINVAL`

Description:

key is invalid.

Note –

`pthread_setspecific()` does not free its storage when a new binding is set. The existing binding must be freed, otherwise a memory leak can occur.

Getting Thread-Specific Data

Use pthread_getspecific(3C) to get the calling thread's binding for **key**, and store the binding in the location pointed to by **value**.

`pthread_getspecific` Syntax

```
void *pthread_getspecific(pthread_key_t key);

#include <pthread.h>

pthread_key_t key;
void *value;

/* key previously created */
value = pthread_getspecific(key);
```

`pthread_getspecific` Return Values

`pthread_getspecific` returns no errors.

Global and Private Thread-Specific Data Example

Example 2–2 shows an excerpt from a multithreaded program. This code is executed by any number of threads, but the code has references to two global variables, *errno* and *mywindow*. These global values really should be references to items private to each thread.

Example 2–2 Thread-Specific Data—Global but Private

```
body() {
    ...

    while (write(fd, buffer, size) == -1) {
        if (errno != EINTR) {
            fprintf(mywindow, "%s\n", strerror(errno));
            exit(1);
        }
    }

    ...

}
```

References to `errno` should get the system error code from the routine called by this thread, not by some other thread. Including the header file `errno.h` causes a reference to `errno` to be a reference to a thread-private instance of `errno`, so that references to `errno` by one thread refer to a different storage location than references to `errno` by other threads.

The *mywindow* variable refers to a `stdio` stream that is connected to a window that is private to the referring thread. So, as with `errno`, references to *mywindow* by one thread should refer to a different storage location than references to *mywindow* by other threads. Ultimately, the reference is to a different window. The only difference here is that the system takes care of `errno` , but the programmer must handle references for *mywindow* .

The next example shows how the references to *mywindow* work. The preprocessor converts references to *mywindow* into invocations of the **`_mywindow()`** procedure.

This routine in turn invokes **`pthread_getspecific()`**. **`pthread_getspecific()`** receives the *mywindow_key* global variable and *win* an output parameter that receives the identity of this thread's window.

Example 2–3 Turning Global References Into Private References

```
thread_key_t mywin_key;

FILE *_mywindow(void) {
 FILE *win;
 win = pthread_getspecific(mywin_key);
 return(win);
 }
#define mywindow _mywindow()

void routine_uses_win( FILE *win) {
 ...
}
void thread_start(...) {
```

```
 ...
 make_mywin();
 ...
 routine_uses_win( mywindow )
 ...
}
```

The *mywin_key* variable identifies a class of variables for which each thread has its own private copy. These variables are thread-specific data. Each thread calls **make_mywin()** to initialize its window and to arrange for its instance of *mywindow* to refer to the thread-specific data.

Once this routine is called, the thread can safely refer to *mywindow* and, after **_mywindow()**, the thread gets the reference to its private window. References to *mywindow* behave as if direct references were made to data private to the thread.

Example 2–4 shows how to set up the reference.

Example 2–4 Initializing the Thread-Specific Data

```
void make_mywindow(void) {
    FILE **win;
    static pthread_once_t mykeycreated = PTHREAD_ONCE_INIT;

    pthread_once(&mykeycreated, mykeycreate);

    win = malloc(sizeof(*win));
    create_window(win, ...);

    pthread_setspecific(mywindow_key, win);
}

void mykeycreate(void) {
    pthread_key_create(&mywindow_key, free_key);
}

void free_key(void *win) {
    free(win);
}
```

First, get a unique value for the key, *mywin_key*. This key is used to identify the thread-specific class of data. The first thread to call **make_mywin()** eventually calls **pthread_key_create()** , which assigns to its first argument a unique *key*. The second argument is a `destructor` function that is used to deallocate a thread's instance of this thread-specific data item once the thread terminates.

The next step is to allocate the storage for the caller's instance of this thread-specific data item. Having allocated the storage, calling **create_window()** sets up a window for the thread. *win* points to the storage allocated for the window. Finally, a call is made to **pthread_setspecific()**, which associates *win* with the key.

Subsequently, whenever the thread calls **pthread_getspecific()** to pass the global *key*, the thread gets the value that is associated with this key by this thread in an earlier call to **pthread_setspecific()** .

When a thread terminates, calls are made to the `destructor` functions that were set up in **pthread_key_create()**. Each `destructor` function is called only if the terminating thread established a value

for the *key* by calling **`pthread_setspecific()`**.

Getting the Thread Identifier

Use pthread_self(3C) to get the thread identifier of the calling thread.

`pthread_self` Syntax

```
pthread_t  pthread_self(void);
```

```
#include <pthread.h>

pthread_t tid;

tid = pthread_self();
```

`pthread_self` Return Values

`pthread_self()` returns the thread identifier of the calling thread.

Comparing Thread IDs

Use pthread_equal(3C) to compare the thread identification numbers of two threads.

`pthread_equal` Syntax

```
int  pthread_equal(pthread_t tid1, pthread_t tid2);
```

```
#include <pthread.h>

pthread_t tid1, tid2;
int ret;

ret = pthread_equal(tid1, tid2);
```

`pthread_equal` Return Values

`pthread_equal()` returns a nonzero value when *tid1* and *tid2* are equal, otherwise, `0` is returned. When either *tid1* or *tid2* is an invalid thread identification number, the result is unpredictable.

Calling an Initialization Routine for a Thread

Use pthread_once(3C) in a threaded process to call an initialization routine the first time `pthread_once` is called. Subsequent calls to **`pthread_once()`** from any thread in the process have no effect.

`pthread_once` Syntax

```
int  pthread_once(pthread_once_t *once_control, void (*init_routine)(void));
```

```
#include <pthread.h>

pthread_once_t once_control = PTHREAD_ONCE_INIT;
int ret;
```

```
ret = pthread_once(&once_control,
init_routine);
```

The *once_control* parameter determines whether the associated initialization routine has been called.

`pthread_once` Return Values

`pthread_once()` returns zero after completing successfully. Any other return value indicates that an error occurred. When the following condition occurs, **`pthread_once()`** fails and returns the corresponding value.

`EINVAL`

Description:

once_control or *init_routine* is `NULL`.

Yielding Thread Execution

Use sched_yield(3RT) to cause the current thread to yield its execution in favor of another thread with the same or greater priority. If no such threads are ready to run, the calling thread continues to run. The **`sched_yield()`** function is not part of the Pthread API, but is a function in the Realtime Library Functions. You must include `<sched.h>` to use **`sched_yield()`**.

`sched_yield` Syntax

```
int  sched_yield(void);
```

```
#include <sched.h>
int ret;
ret = sched_yield();
```

`sched_yield` Return Values

`sched_yield()` returns zero after completing successfully. Otherwise, -1 is returned and *errno* is set to indicate the error condition.

Setting the Thread Policy and Scheduling Parameters

Use pthread_setschedparam(3C) to modify the scheduling policy and scheduling parameters of an individual thread.

`pthread_setschedparam` Syntax

```
int pthread_setschedparam(pthread_t tid, int  policy,
    const struct sched_param *param);
```

```
#include <pthread.h>

pthread_t tid;
int ret;
struct sched_param param;
int priority;
```

```
/* sched_priority will be the priority of the thread */
sched_param.sched_priority = priority;
policy = SCHED_OTHER;

/* scheduling parameters of target thread */
ret = pthread_setschedparam(tid,
policy, &param);
```

Supported policies are `SCHED_FIFO`, `SCHED_RR`, and `SCHED_OTHER`.

`pthread_setschedparam` Return Values

`pthread_setschedparam()` returns zero after completing successfully. Any other return value indicates that an error occurred. When either of the following conditions occurs, the **`pthread_setschedparam()`** function fails and returns the corresponding value.

`EINVAL`

Description:

The value of the attribute being set is not valid.

`EPERM`

Description:

The caller does not have the appropriate permission to set either the scheduling parameters or the scheduling policy of the specified thread.

`ESRCH`

Description:

The value specified by *tid* does not refer to an existing thread.

Getting the Thread Policy and Scheduling Parameters

pthread_getschedparam(3C) gets the scheduling policy and scheduling parameters of an individual thread.

`pthread_getschedparam` Syntax

```
int  pthread_getschedparam(pthread_t tid, int *restrict policy,
    struct sched_param *restrict param);

#include <pthread.h>

pthread_t tid;
sched_param param;
int priority;
int policy;
int ret;
```

```
/* scheduling parameters of target thread */
ret = pthread_getschedparam (tid, &policy, &param);

/* sched_priority contains the priority of the thread */
priority = param.sched_priority;
```

`pthread_getschedparam` Return Values

`pthread_getschedparam()` returns zero after completing successfully. Any other return value indicates that an error occurred. When the following condition occurs, the function fails and returns the corresponding value.

`ESRCH`

Description:

The value specified by *tid* does not refer to an existing thread.

Setting the Thread Priority

pthread_setschedprio(3C) sets the scheduling priority for the specified thread.

`pthread_setschedprio` Syntax

```
int pthread_setschedprio(pthread_t tid, int prio);

#include <pthread.h>

pthread_t tid;
int prio;
int ret;
```

`pthread_setschedprio` Return Values

`pthread_setschedprio()` returns zero after completing successfully. Any other return value indicates that an error occurred. When the following condition occurs, the function fails and returns the corresponding value.

`EINVAL`

Description:

The value of *prio* is invalid for the scheduling policy of the specified thread.

`ENOTSUP`

Description:

An attempt was made to set the priority to an unsupported value.

`EPERM`

Description:

The caller does not have the appropriate permission to set the scheduling priority of the specified thread.

`ESRCH`

Description:

The value specified by *tid* does not refer to an existing thread.

Sending a Signal to a Thread

Use pthread_kill(3C) to send a signal to a thread.

`pthread_kill` Syntax

```
int  pthread_kill(pthread_t tid, int sig);
```

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

int sig;
pthread_t tid;
int ret;

ret = pthread_kill(tid,
sig);
```

`pthread_kill()` sends the signal **sig** to the thread specified by **tid**. **tid** must be a thread within the same process as the calling thread. The**sig** argument must be from the list that is given in signal.h(3HEAD).

When **sig** is zero, error checking is performed but no signal is actually sent. This error checking can be used to check the validity of **tid**.

`pthread_kill` Return Values

`pthread_kill()` returns zero after completing successfully. Any other return value indicates that an error occurred. When either of the following conditions occurs, **`pthread_kill()`** fails and returns the corresponding value.

`EINVAL`

Description:

sig is not a valid signal number.

`ESRCH`

Description:

tid cannot be found in the current process.

Accessing the Signal Mask of the Calling Thread

Use pthread_sigmask(3C) to change or examine the signal mask of the calling thread.

`pthread_sigmask` Syntax

```
int pthread_sigmask(int how, const sigset_t *new, sigset_t *old);

#include <pthread.h>
#include <signal.h>

int ret;
sigset_t old, new;

ret = pthread_sigmask(SIG_SETMASK, &new, &old); /* set new mask */
ret = pthread_sigmask(SIG_BLOCK, &new, &old); /* blocking mask */
ret = pthread_sigmask(SIG_UNBLOCK, &new, &old); /* unblocking */
```

how determines how the signal set is changed. *how* can have one of the following values:

- `SIG_BLOCK`. Add *new* to the current signal mask, where *new* indicates the set of signals to block.
- `SIG_UNBLOCK`. Delete *new* from the current signal mask, where *new* indicates the set of signals to unblock.
- `SIG_SETMASK` . Replace the current signal mask with *new*, where *new* indicates the new signal mask.

When the value of *new* is `NULL`, the value of *how* is not significant. The signal mask of the thread is unchanged. To inquire about currently blocked signals, assign a `NULL` value to the *new* argument.

The *old* variable points to the space where the previous signal mask is stored, unless *old* is `NULL`.

`pthread_sigmask` Return Values

`pthread_sigmask()` returns zero when the call completes successfully. Any other return value indicates that an error occurred. When the following condition occurs, **`pthread_sigmask()`** fails and returns the corresponding value.

`EINVAL`

Description:

The value of *how* is not defined and *old* is `NULL`.

Forking Safely

See the discussion about pthread_atfork(3C) in Solution: pthread_atfork.

`pthread_atfork` Syntax

```
int pthread_atfork(void (*prepare) (void), void (*parent) (void),
    void (*child) (void) );
```

`pthread_atfork` Return Values

`pthread_atfork()` returns zero when the call completes successfully. Any other return value indicates that an error occurred. When the following condition occurs, **`pthread_atfork()`** fails and returns the corresponding value.

`ENOMEM`

Description:

Insufficient table space exists to record the fork handler addresses.

Terminating a Thread

Use pthread_exit(3C) to terminate a thread.

`pthread_exit` Syntax

```
void     pthread_exit(void *status);
```

```
#include <pthread.h>
void *status;
pthread_exit(status); /* exit with status */
```

The **`pthread_exit()`** function terminates the calling thread. All thread-specific data bindings are released. If the calling thread is not detached, then the thread's ID and the exit status specified by **status** are retained until your application calls **`pthread_join()`** to wait for the thread. Otherwise, **status** is ignored. The thread's ID can be reclaimed immediately. For information on thread detachment, see Setting Detach State.

`pthread_exit` Return Values

The calling thread terminates with its exit status set to the contents of *status*.

Finishing Up

A thread can terminate its execution in the following ways:

- By returning from its first (outermost) procedure, the threads start routine. See `pthread_create`.
- By calling **`pthread_exit()`**, supplying an exit status.
- By termination with POSIX cancel functions. See **`pthread_cancel()`** .

The default behavior of a thread is to linger until some other thread has acknowledged its demise by “joining” with the lingering thread. This behavior is the same as the default **`pthread_create()`** attribute that is nondetached, see `pthread_detach`. The result of the join is that the joining thread picks up the exit status of the terminated thread and the terminated thread vanishes.

An important special case arises when the initial thread, calling **`main()`**, returns from calling **`main()`** or calls **`exit()`**. This action causes the entire process to be terminated, along with all its threads. So, take care to ensure that the initial thread does not return from **`main()`** prematurely.

Note that when the main thread merely calls `pthread_exit`, the main thread terminates itself only. The other threads in the process, as well as the process, continue to exist. The process terminates when all threads terminate.

Cancel a Thread

Cancellation allows a thread to request the termination of any other thread in the process. Cancellation is an option when all further operations of a related set of threads are undesirable or unnecessary.

One example of thread cancellation is an asynchronously generated cancel condition, such as, when a user requesting to close or exit a running application. Another example is the completion of a task undertaken by a number of threads. One of the threads might ultimately complete the task while the others continue to operate. Since the running threads serve no purpose at that point, these threads should be cancelled.

Cancellation Points

Be careful to cancel a thread only when cancellation is safe. The pthreads standard specifies several cancellation points, including:

- Programmatically, establish a thread cancellation point through a `pthread_testcancel` call.
- Threads waiting for the occurrence of a particular condition in `pthread_cond_wait` or `pthread_cond_timedwait(3C)`.
- Threads blocked on `sigwait(2)`.
- Some standard library calls. In general, these calls include functions in which threads can block. See the cancellation(5) man page for a list.

Cancellation is enabled by default. At times, you might want an application to disable cancellation. Disabled cancellation has the result of deferring all cancellation requests until cancellation requests are enabled again.

See `pthread_setcancelstate` Syntax for information about disabling cancellation.

Placing Cancellation Points

Dangers exist in performing cancellations. Most deal with properly restoring invariants and freeing shared resources. A thread that is cancelled without care might leave a mutex in a locked state, leading to a deadlock. Or a cancelled thread might leave a region of allocated memory with no way to identify the memory and therefore unable to free the memory.

The standard C library specifies a cancellation interface that permits or forbids cancellation programmatically. The library defines **cancellation points** that are the set of points at which cancellation can occur. The library also allows the scope of **cancellation handlers** to be defined so that the handlers are sure to operate when and where intended. The cancellation handlers provide clean up services to restore resources and state to a condition that is consistent with the point of origin.

Placement of cancellation points and the effects of cancellation handlers must be based on an understanding of the application. A mutex is explicitly not a cancellation point and should be held only for the minimal essential time.

Limit regions of asynchronous cancellation to sequences with no external dependencies that could result in dangling resources or unresolved state conditions. Take care to restore cancellation state when returning from some alternate, nested cancellation state. The interface provides features to facilitate restoration:

`pthread_setcancelstate(3C)` preserves the current cancel state in a referenced variable, `pthread_setcanceltype(3C)` preserves the current cancel type in the same way.

Cancellations can occur under three different circumstances:

- Asynchronously
- At various points in the execution sequence as defined by the standard
- At a call to **`pthread_testcancel()`**

By default, cancellation can occur only at well-defined points as defined by the POSIX standard.

In all cases, take care that resources and state are restored to a condition consistent with the point of origin.

Cancelling a Thread

Use pthread_cancel(3C) to cancel a thread.

`pthread_cancel` Syntax

```
int pthread_cancel(pthread_t thread);
```

```
#include <pthread.h>

pthread_t thread;
int ret;

ret = pthread_cancel(thread);
```

How the cancellation request is treated depends on the state of the target thread. Two functions, `pthread_setcancelstate(3C)` and `pthread_setcanceltype(3C)` , determine that state.

`pthread_cancel` Return Values

`pthread_cancel()` returns zero after completing successfully. Any other return value indicates that an error occurred. When the following condition occurs, the function fails and returns the corresponding value.

`ESRCH`

Description:

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

Enabling or Disabling Cancellation

Use pthread_setcancelstate(3C) to enable or disable thread cancellation. When a thread is created, thread cancellation is enabled by default.

`pthread_setcancelstate` Syntax

```
int pthread_setcancelstate(int state, int *oldstate);
```

```
#include <pthread.h>

int oldstate;
int ret;

/* enabled */
ret = pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, &oldstate);

/* disabled */
ret = pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &oldstate);
```

`pthread_setcancelstate` Return Values

`pthread_setcancelstate()` returns zero after completing successfully. Any other return value indicates that an error occurred. When the following condition occurs, the **`pthread_setcancelstate()`** function fails and returns the corresponding value.

`EINVAL`

Description:

The state is not `PTHREAD_CANCEL_ENABLE` or `PTHREAD_CANCEL_DISABLE`.

Setting Cancellation Type

Use pthread_setcanceltype(3C) to set the cancellation type to either deferred or asynchronous mode.

`pthread_setcanceltype` Syntax

```
int pthread_setcanceltype(int type, int *oldtype);

#include <pthread.h>

int oldtype;
int ret;

/* deferred mode */
ret = pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, &oldtype);

/* async mode*/
ret = pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, &oldtype);
```

When a thread is created, the cancellation type is set to deferred mode by default. In deferred mode, the thread can be cancelled only at cancellation points. In asynchronous mode, a thread can be cancelled at any point during its execution. The use of asynchronous mode is discouraged.

`pthread_setcanceltype` Return Values

`pthread_setcanceltype()` returns zero after completing successfully. Any other return value indicates that an error occurred. When the following condition occurs, the function fails and returns the corresponding value.

`EINVAL`

Description:

The type is not `PTHREAD_CANCEL_DEFERRED` or `PTHREAD_CANCEL_ASYNCHRONOUS`.

Creating a Cancellation Point

Use pthread_testcancel(3C) to establish a cancellation point for a thread.

`pthread_testcancel` Syntax

```
void pthread_testcancel(void);
```

```
#include <pthread.h>

pthread_testcancel();
```

The **`pthread_testcancel()`** function is effective when thread cancellation is enabled and in deferred mode. **`pthread_testcancel()`** has no effect if called while cancellation is disabled.

Be careful to insert **`pthread_testcancel()`** only in sequences where thread cancellation is safe. In addition to programmatically establishing cancellation points through the **`pthread_testcancel()`** call, the pthreads standard specifies several cancellation points. See Cancellation Points for more details.

`pthread_testcancel` Return Values

`pthread_testcancel()` has no return value.

Pushing a Handler Onto the Stack

Use cleanup handlers to restore conditions to a state that is consistent with that state at the point of origin. This consistent state includes cleaning up allocated resources and restoring invariants. Use the `pthread_cleanup_push(3C)` and `pthread_cleanup_pop(3C)` functions to manage the handlers.

Cleanup handlers are pushed and popped in the same lexical scope of a program. The push and pop should always match. Otherwise, compiler errors are generated.

`pthread_cleanup_push` Syntax

Use pthread_cleanup_push(3C) to push a cleanup handler onto a cleanup stack (LIFO).

```
void pthread_cleanup_push(void(*routine)(void *), void *args);
```

```
#include <pthread.h>

/* push the handler "routine" on cleanup stack */
pthread_cleanup_push (routine, arg);
```

`pthread_cleanup_push` Return Values

`pthread_cleanup_push()` has no return value.

Pulling a Handler Off the Stack

Use pthread_cleanup_pop(3C) to pull the cleanup handler off the cleanup stack.

`pthread_cleanup_pop` Syntax

```
void pthread_cleanup_pop(int execute);
```

```
#include <pthread.h>

/* pop the "func" out of cleanup stack and execute "func" */
pthread_cleanup_pop (1);

/* pop the "func" and DONT execute "func" */
pthread_cleanup_pop (0);
```

A nonzero argument in the pop function removes the handler from the stack and executes the handler. An argument of zero pops the handler without executing the handler.

`pthread_cleanup_pop()` is effectively called with a nonzero argument when a thread either explicitly or implicitly calls **`pthread_exit()`** or when the thread accepts a cancel request.

`pthread_cleanup_pop` Return Values

`pthread_cleanup_pop()` has no return values.

- *Previous*: Lifecycle of a Thread
- *Next*: Chapter 3 Thread Attributes