Basic Threads Programming 2

The Threads Library

This chapter introduces the basic threads programming routines from the POSIX threads library, `libpthread(3T)`. This chapter covers *default threads*, or threads with default attribute values, which are the kinds of threads that are most often used in multithreaded programming.

The next chapter, “Thread Create Attributes,” explains how to create and use threads with nondefault attributes.

The POSIX (`libpthread`) routines introduced here have programming interfaces that are similar to the original (`libthread`) Solaris multithreading library.

Create a Default Thread

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

- Unbound
- Nondetached
- With a default stack and stack size
- With the parent's priority

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 "Initialize Attributes" on page 45 for details.

pthread_create(3T)

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

```
Prototype:
int pthread_create(pthread_t *tid, const pthread_attr_t *tattr,
    void*(*start_routine)(void *), void *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 the *attr* having 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_exit(3T)`" on page 33).

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

Creating a thread using a NULL attribute argument has the same effect as using a default attribute. Both create a default thread. When *tattr* is initialized, it acquires the default behavior.

Return Values

Returns a zero and exits when it completes successfully. Any other returned value indicates that an error occurred. When any of the following conditions are detected, `pthread_create()` fails and returns the corresponding value.

`EAGAIN` – A system limit is exceeded, such as when too many LWPs have been created.

`EINVAL` – The value of *tattr* is invalid.

Wait for Thread Termination

pthread_join(3T)

Use the `pthread_join()` function to wait for a thread to terminate.

```
Prototype:
int pthread_join(thread_t tid, void **status);
```

```
#include <phread.h>

pthread_t tid;
int ret;
int 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 `pthread_join()` function blocks the calling thread until the specified thread terminates.

The specified thread must be in the current process and must not be detached. For information on thread detachment, see "Set Detach State" on page 47.

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

Multiple threads cannot wait for the same thread to terminate. If they try to, one thread returns successfully and the others fail with an error of ESRCH.

After `pthread_join` returns, any stack storage associated with the thread can be reclaimed by the application.

Return Values

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

`ESRCH` – *tid* is not a valid, undetached thread in the current process.

`EDEADLK` – *tid* specifies the calling thread.

`EINVAL` – The value of *tid* is invalid.

The `pthread_join()` routine takes two arguments, giving you some flexibility in its use. When you want the caller to wait until a specific thread terminates, supply that thread's ID as the first argument.

If you are interested in the exit code of the defunct thread, supply the address of an area to receive it.

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

Think of a detached thread as being the usual sort of thread and reserve nondetached threads for only those situations that require them.

A Simple Threads Example

In Code Example 2-1, one thread executes the procedure at the top, creating a helper thread that executes the procedure `fetch`, which involves a complicated database lookup and might take some time.

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

The result is passed as a stack parameter, which can be done here because the main thread waits for the spun-off thread to terminate. In general, though, it is better to `malloc(3C)` storage from the heap instead of passing an address to thread stack storage, which can disappear or be reassigned if the thread terminated.

Code Example 2-1 A Simple Threads Program

```
void mainline (...)
{
        char int result;
        pthread_attr_t tattr;
        pthread_t helper;
        int status;

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

            /* do something else for a while */

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

void fetch(int *result)
{
        /* fetch value from a database */

        *result = value;
        pthread_exit(0);
}
```

Detaching a Thread

pthread_detach(3T)

`pthread_detach(3T)` is an alternate way (as opposed to `pthread_join(3T)`) to reclaim storage for a thread that is created with a *detachstate* attribute set to `PTHREAD_CREATE_JOINABLE`.

Prototype:

```
int pthread_detach(thread_t tid);
```

```
#include <phread.h>

pthread_t tid;
int ret;

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

The `pthread_detach(3T)` function is used to indicate to the implementation that storage for the thread *tid* can be reclaimed when the thread terminates. If *tid* has not terminated, `pthread_detach(3T)` does not cause it to terminate. The effect of multiple `pthread_detach(3T)` calls on the same target thread is unspecified.

Return Values

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

`EINVAL` – *tid* is not a valid thread.

`ESRCH` – *tid* is not a valid, undetached thread in the current process.

Create 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 is added—thread-specific data (TSD). This is very much like global data, except that it is private to a thread.

Thread-specific data 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. Using the key, a thread can access a pointer (`void *`) that is maintained per-thread.

`pthread_keycreate(3T)`

Use `pthread_keycreate()` 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, and all threads initially have the value NULL associated with the key when it is created.

`pthread_keycreate()` is called once for each key before the key is used. There is no implicit synchronization.

Once a key has been created, each thread can bind a value to the key. The values are specific to the thread 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.

```
Prototype:
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);
```

When pthread_keycreate() 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. When 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.

Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred. When any of the following conditions occur, pthread_keycreate() fails and returns the corresponding value.

EAGAIN – The *key* name space is exhausted.

ENOMEM – Not enough virtual memory is available in this process to create a new key.

Delete the Thread-Specific Data Key

pthread_keydelete(3T)

Use pthread_keydelete() to destroy an existing thread-specific data key. This can be used to cause an error return when trying to access some thread-specific data set that is no longer valid. (Read the POSIX standard document for the rationale.) There is no comparable function in Solaris threads.

Prototype:
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);
```

Once a key has been deleted, any reference to it with the `pthread_setspecific()` or `pthread_getspecific()` calls results in the `EINVAL` error.

It is the responsibility of the programmer to free any thread-specific resources before calling the delete function. This function does not invoke any of the destructors.

Return Values

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

`EINVAL` – The *key* value is invalid.

Set the Thread-Specific Data Key

pthread_setspecific(3T)

Use `pthread_setspecific()` for a thread that has not yet established a binding to the specified thread-specific data key.

Prototype:

```
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);
```

Return Values

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

`ENOMEM` – Not enough virtual memory is available.

`EINVAL` – *key* is invalid.

Note – A memory leak can occur if you set a new binding for a thread to a key that the thread has already used.

Get the Thread-Specific Data Key

pthread_getspecific(3T)

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

```
Prototype:
int pthread_getspecific(pthread_key_t key);
```

```
#include <pthread.h>

pthread_key_t key;
void *value;

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

Return Values

No errors are returned.

Global and Private Thread-Specific Data Example

Code Example 2-2 shows an excerpt from a multithreaded program. This code is executed by any number of threads, but it has references to two global variables, `errno` and `mywindow`, that really should be references to items private to each thread.

Code 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. So, references to `errno` by one thread refer to a different storage location than references to `errno` by other threads.

The `mywindow` variable is intended to refer to a `stdio` stream 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 (and, ultimately, a different window) than references to `mywindow` by other threads. The only difference here is that the threads library takes care of `errno`, but the programmer must somehow make this work 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(3T)`, passing it the `mywindow_key` global variable (it really is a global variable) and an output parameter, `win`, that receives the identity of this thread's window.

Code Example 2-3 Turning Global References Into Private References

```
#define mywindow _mywindow()

thread_key_t mywindow_key;

FILE *_mywindow(void) {
    FILE *win;

    pthread_getspecific(mywindow_key, &win);
    return(win);
}


void thread_start(...) {
    ...
    make_mywindow();
    ...
}
```

The `mywindow_key` variable identifies a class of variables for which each thread has its own private copy; that is, these variables are thread-specific data. Each thread calls `make_mywindow()` to initialize its window and to arrange for its instance of `mywindow` to refer to it.

Once this routine is called, the thread can safely refer to `mywindow` and, after `_mywindow`, the thread gets the reference to its private window. So, references to `mywindow` behave as if they were direct references to data private to the thread.

Code Example 2-4 shows how to set this up.

Code 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_keycreate(&mywindow_key, free_key);
}

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

First, get a unique value for the key, `mywindow_key`. This key is used to identify the thread-specific class of data. So, the first thread to call `make_mywindow` eventually calls `pthread_keycreate(3T)`, 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, a call is made to the `create_window` routine, which sets up a window for the thread and sets the storage pointed to by `win` to refer to it. Finally, a call is made to `pthread_setspecific`, which associates the value contained in `win` (that is, the location of the storage containing the reference to the window) with the key.

After this, whenever this thread calls `pthread_getspecific()`, passing the global key, it gets the value that was associated with this key by this thread when it called `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()`.

Get the Thread Identifier

`pthread_self(3T)`

Use `pthread_self()` to get the ID of the calling thread.

```
Prototype:
pthread_t pthread_self(void);
```

```
#include &lt;pthread.h&gt;

pthread_t tid;

tid = pthread_self();
```

Return Values

Returns the ID of the calling thread.

Compare Thread IDs

pthread_equal(3T)

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

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

```
#include <pthread.h>

pthread_t tid1, tid2;
int ret;

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

Return Values

Returns a non-zero value when *tid1* and *tid2* are equal; otherwise, zero is returned. When either *tid1* or *tid2* is an invalid thread identification number, the result is unpredictable.

Initializing Threads

pthread_once(3T)

Use `pthread_once(3T)` to call an initialization routine the first time `pthread_once(3T)` is called. Subsequent calls to `pthread_once(3T)` have no effect..

```
Prototype:
int  pthread_once(pthread_once_ *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.

Return Values

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

`EINVAL` – *once_control* or *init_routine* is NULL.

Yield Thread Execution

sched_yield(3R)

Use `sched_yield()` to cause the current thread to yield its execution in favor of another thread with the same or greater priority.

```
Prototype:
int  sched_yield(void);
```

```
#include <sched.h>

int ret;

ret = sched_yield();
```

Return Values

Returns zero after completing successfully. Otherwise -1 is returned and `errno` is set to indicate the error condition.

`ENOSYS` – `sched_yield(3R)` is not supported in this implementation.

Set the Thread Priority

pthread_setschedparam(3T)

Use `pthread_setschedparam()` to modify the priority of an existing thread.This function has no effect on scheduling policy.

```
Prototype:
int  pthread_setschedparam(pthread_t tid, int policy,
     const struct schedparam *param);
```

```
#include <pthread.h>

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

/* sched_priority will be the priority of the thread */
schedparam.sched_priority = priority;

/* only supported policy, others will result in ENOTSUP */
policy = SCHED_OTHER;

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

Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred. When either of the following conditions occurs, the function fails and returns the corresponding value.

`EINVAL` – The value of the attribute being set is not valid.

`ENOTSUP` – An attempt was made to set the attribute to an unsupported value.

Get the Thread Priority

pthread_getschedparam(3T)

Gets the priority of the existing thread.

```
Prototype:
int  pthread_getschedparam(pthread_t tid, int policy,
    struct schedparam *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 = schedparam.sched_priority;
```

Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred. When the following condition occurs, the function fails and returns the corresponding value.

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

Send a Signal to a Thread

pthread_kill(3T)

Use `pthread_kill()` to send a signal to a thread.

```
Prototype:
int  pthread_kill(thread_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 given in `signal(5)`.

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

Return Values

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

`EINVAL` – *sig* is not a valid signal number.

`ESRCH` – *tid* cannot be found in the current process.

Access the Signal Mask of the Calling Thread

pthread_sigmask(3T)

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

```
Prototype:
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. It 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 and the signal mask of the thread is unchanged. So, 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 it is NULL.

Return Values

Returns a zero when it completes successfully. Any other returned value indicates that an error occurred. When the following condition occurs, `pthread_sigmask()` fails and returns the corresponding value.

`EINVAL` – The value of *how* is not defined.

Re-create and Reinitialize Critical Threads

`pthread_atfork(3T)`

See the discussion about `pthread_atfork()` on page 123.

```
Prototype:

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

Terminate a Thread

`pthread_exit(3T)`

Use `pthread_exit()` to terminate a thread.

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

```
#include <pthread.h>

int 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 the thread is waited for. Otherwise, *status* is ignored and the thread's ID can be reclaimed immediately. For information on thread detachment, see "Set Detach State" on page 47.

Return Values

The calling thread terminates with its exit status set to the contents of *status* if *status* is not NULL.

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(3T)`
- By calling `pthread_exit(3T)`, supplying an exit status
- By termination with POSIX cancel functions; see `pthread_cancel(3T)`

The default behavior of a thread is to remain until some other thread has acknowledged its demise by "joining" with it. This is the same as the default pthread create attribute being non-detached; see `pthread_detach(3T)`. The result of the join is that the joining thread picks up the exit status of the dying thread and the dying thread vanishes.

An important special case arises when the main thread, the one that existed initially, returns from the main procedure or calls `exit(3C)`. This action causes the entire process to be terminated, along with all its threads. So take care to ensure that the main thread does not return from `main` prematurely.

Note that when the main thread merely calls `pthread_exit(3T)`, it terminates only itself—the other threads in the process, as well as the process, continue to exist. (The process terminates when all threads terminate.)

Cancellation

POSIX threads introduces the notion of cancellability to threads programming. Cancellation allows a thread to terminate the execution of any other thread, or all threads, in the process. Cancellation is an option when all further operations of a related set of threads are undesirable or unnecessary. A good method is to cancel all threads, restore the process to a consistent state, and then return to the point of origin.

One example is an asynchronously generated cancel condition such as a user requesting to close or exit some 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 they are serving no purpose at that point, they all should be cancelled.

There are dangers 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 it might leave a region of memory allocated with no way to identify it and therefore no way to free it.

pthreads specifies a cancellation interface that permits or forbids cancellation programmatically. pthreads defines the set of points at which cancellation can occur *(cancellation points)*. It also allows the scope of cancellation handlers, which provide clean up services, to be defined so that they are sure to operate when and where intended.

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 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(3T)` preserves the current cancel state in a referenced variable; `pthread_setcanceltype(3T)` 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 discrete points specified by the application

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.

Cancellation Points

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

- The programmatically-determined `pthread_testcancel(3T)` call
- Threads waiting in `pthread_cond_wait(3T)` or `pthread_cond_timedwait(3T)`.
- Threads waiting for termination of another thread in `pthread_join(3T)`.
- Threads blocked on `sigwait(2)`.
- Some standard library calls. In general, these are functions in which threads can block; see the man page `cancellation(3T)` for a list.

By default cancellation is enabled. At times you might want an application to disable cancellation. This has the result of deferring all cancellation requests until they are enabled again. Note that enabling cancellation constitutes a cancellation point.

See `pthread_setcancelstate()` for information about disabling cancellation.

Cancel a Thread

pthread_cancel(3T)

Use `pthread_cancel()` to cancel a thread.

```
Prototype:
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(3T)` and `pthread_setcanceltype(3T)`, determine that state.

Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred. When the following condition occurs, the function fails and returns the corresponding value.

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

Enable or Disable Cancellation

pthread_setcancelstate(3T)

Use `pthread_setcancelstate()` to enable or disable cancellability of a thread. When a thread is created, cancellability is enabled by default.

```
Prototype:
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);
```

Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred. When the following condition occurs, the function fails and returns the corresponding value.

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

Set Cancellation Type

pthread_setcanceltype(3T)

Use `pthread_setcanceltype()` to set the cancellation type to either deferred or asynchronous mode. 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 any point during its execution. Using asynchronous mode is discouraged.

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

```
#include <pthread.h>

int oldtype;
int ret;

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

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

Return Values

Returns zero after completing successfully. Any other returned value indicates that an error occurred. When the following condition occurs, the function fails and returns the corresponding value.

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

Create a Cancellation Point

`pthread_testcancel(3T)`

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

```
Prototype:
void pthread_testcancel(void);
```

```
#include <pthread.h>

pthread_testcancel();
```

The `pthread_testcancel()` function is effective when cancellability is enabled and in deferred mode. Calling this function while cancellability is disabled has no effect.

Be careful to insert `pthread_testcancel()` only in sequences where it is safe to cancel a thread. In addition to the programmatically determined `pthread_testcancel()` call, the pthreads standard specifies several cancellation points. See "Cancellation Points" on page 36 for more details.

There is no return value.

Push a Handler Onto the Stack

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

Cleanup handlers are pushed and popped in the same lexical scope of a program. They should always match; otherwise compiler errors will be generated.

pthread_cleanup_push(3T)

Use the `pthread_cleanup_push()` function to push a cleanup handler onto a cleanup stack (FIFO).

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

```
#include <pthread.h>

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

Pull a Handler Off the Stack

pthread_cleanup_pop(3T)

Use the `pthread_cleanup_pop()` function to pull the cleanup handler off the cleanup stack.

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

`pthread_cleanup_pop()` is effectively called with a nonzero argument if a thread either explicitly or implicitly calls `pthread_exit(3T)` or if the thread accepts a cancel request.

Prototype:
`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);
```

There are no return values.