NAME

cancellation, pthread_cancel, pthread_setcancelstate, pthread_setcanceltype, pthread_testcancel, pthread_cleanup_push, pthread_cleanup_pop – canceling execution of a thread

SYNOPSIS

#include <pthread.h>

int pthread_cancel(pthread_t *target_thread***);**

int pthread_setcancelstate(int *state***, int** **oldstate***);**
int pthread_setcanceltype(int *type***, int** **oldtype***);**
void pthread_testcancel( );

void pthread_cleanup_push(void (**handler***)(void *) void** **arg***);**
void pthread_cleanup_pop(int *execute***);**

MT-LEVEL

MT-Safe

DESCRIPTION

Thread cancellation enables a thread to terminate the execution of any other thread in the process. When the notice of cancellation is acted upon, the target thread (the thread being cancelled) is allowed to hold pending cancellation requests in several ways and to perform application-specific cleanup processing.

As a thread acquires resources around areas where it may get cancelled (i.e., before a cancellation point), it needs to push cancellation cleanup handlers along with the acquisition of these resources. The cleanup handlers release these resources and are invoked only if the thread were to be cancelled. As the thread leaves the last cancellation point before releasing a resource, it needs to pop the cleanup handler it had pushed earlier for this resource.

When a thread is cancelled, all the currently stacked cleanup handlers are executed and thread execution is terminated when the last cancellation cleanup handler returns. Its exit status of **PTHREAD_CANCELED** is then available to any threads joining with the cancelled target thread.

The thread's cancellation state and type determine when a thread could get cancelled.

State

PTHREAD_CANCEL_DISABLE
All cancellation requests to the *target_thread* are held pending.

PTHREAD_CANCEL_ENABLE
Cancellation requests are acted upon, depending upon the thread's cancellation type:

PTHREAD_CANCEL_ASYNCHRONOUS
If the cancellation state is enabled, new or pending cancellation requests may be acted upon at any time.

PTHREAD_CANCEL_DEFERRED
Cancellation requests are held pending until a cancellation point (see below) is reached.

Disabling cancellation will cause the setting of the cancellation type to be ineffective because all cancellation requests are held pending; however, when cancellation is enabled again, the new type will be in effect. The cancellation state is set to enabled, by default.

Type

When the cancellation state is disabled, a thread's cancellation *type* is meaningless. The following cancellation types behave as follows when enabled:

PTHREAD_CANCEL_ASYNCHRONOUS
Receipt of a **pthread_cancel( )** call will result in an immediate cancellation.

PTHREAD_CANCEL_DEFERRED
Cancellation will not occur until the target thread reaches a cancellation point (see below). Receipt of a **pthread_cancel( )** call will result in an immediate cancellation at this cancellation point.

The cancellation type is set to **PTHREAD_CANCEL_DEFERRED,** by default.

Cancellation Points

Cancellation begins at a point in a thread's execution when pending cancellation requests are tested and the cancellation state is found to be enabled. This is called the cancellation point.

A cancellation point can be explicitly set by inserting a call to the **pthread_testcancel( )** function.

In addition to explicit **pthread_testcancel( )** cancellation points, implicit cancellation points occur at a defined list of system entry points. Typically, any call that might require a long term wait should be a cancellation point. Operations need only check for pending cancellation requests when the operation is about to block indefinitely. This includes threads waiting in **pthread_cond_wait**(3T) and **pthread_cond_timedwait**(3T), threads waiting for the termination of another thread in **pthread_join**(3T), and threads blocked on **sigwait**(2).

POSIX has also defined several other functions (in **libc** and **libposix4**), as implicit cancellation points. In general, these are functions in which threads may block:

aio_suspend(3R), **close**(2), **creat**(2), **fcntl**(2), **fsync**(3C), **mq_receive**(3R), **mq_send**(3R), **msync**(3C), **nanosleep**(3R), **open**(2), **pause**(2), **pthread_cond_timedwait**(3T), **pthread_cond_wait**(3T), **pthread_join**(3T), **pthread_testcancel**, **read**(2), **sem_wait**(3R), **sigwaitinfo**(3R), **sigsuspend**(2), **sigtimedwait**(3R), **sigwait**(2), **sleep**(3C), **system**(3S), **tcdrain**(3), **wait**(2), **waitpid**(2), and **write**(2).

A cancellation point may also occur when a thread is executing the following functions:

closedir(3C), **ctermid**(3S), **fclose**(3S), **fcntl**(2), **fflush**(3S), **fgetc**(3S), **fgets**(3S), **fopen**(3S), **fprintf**(3S), **fputc**(3S), **fputs**(3S), **fread**(3S), **freopen**(3S), **fscanf**(3S), **fseek**(3S), **ftell**(3S), **fwrite**(3S), **getc**(3S), **getc_unlocked**(3S), **getchar**(3S), **getchar_unlocked**(3S), **getcwd**(3C), **getgrgid**(3C), **getgrgid_r**(3C), **getgrnam**(3C), **getgrnam_r**(3C), **getlogin**(3C), **getlogin_r**(3C), **getpwnam**(3C), **getpwnam_r**(3C), **getpwuid**(3C), **getpwuid_r**(3C), **gets**(3S), **lseek**(2), **rename**(2), **opendir**(3C), **perror**(3C), **printf**(3S), **putc**(3S), **putc_unlocked**(3S), **putchar**(3S), **putchar_unlocked**(3S), **puts**(3S), **readdir**(3C),

remove(3C), **rewind**(3S), **rewinddir**(3C), **scanf**(3S), **tmpfile**(3S), **ttyname**(3C), **ttyname_r**(3C), **ungetc**(3S), and **unlink**(2).

Cleanup Handling

An application should set up a cancellation cleanup handling function to restore any resources before a thread reaches a cancellation point. Specified cancellation points allow programmers to easily keep track of actions needed in a cancellation cleanup handler. A thread should only be made asynchronously cancelable when it is not in the process of acquiring or releasing resources (or locks), or otherwise, not in a difficult or impossible recover state.

When a cancellation request is acted upon, the routines in the list are invoked one-by-one in LIFO (last-in, first-out) order. When a scope's cancellation cleanup handler is invoked, the storage for that scope remains valid.

pthread_cancel

pthread_cancel( ) requests that *target_thread* be canceled. If the *target_thread*'s cancellation state is enabled, the **pthread_cancel( )** call will result in an immediate cancellation, if the target thread has the **PTHREAD_CANCEL_ASYNCHRONOUS** type set. Cancellation cleanup handlers for *target_thread* are called when the cancellation is acted on. Upon return of the last cancellation cleanup handler, the thread-specific data destructor functions are called for *target_thread*. *target_thread* is terminated when the last destructor function returns.

pthread_setcancelstate

pthread_setcancelstate( ) atomically sets the calling thread's cancellation state to the specified *state* and, if *oldstate* is not **NULL,** stores the previous cancellation state in *oldstate*.

A cancellation point occurs in the calling thread once the state is set if **pthread_setcancelstate( )** is called with **PTHREAD_CANCEL_ENABLE**, and *type* is **PTHREAD_CANCEL_ASYNCHRONOUS**.

pthread_setcanceltype

pthread_setcanceltype( ) atomically sets the calling thread's cancellation type to the specified *type* and, if *oldtype* is not **NULL,** stores the previous cancellation type in *oldtype*.

If **pthread_setcanceltype( )** is called with **PTHREAD_CANCEL_ASYNCHRONOUS**, and if *state* is **PTHREAD_CANCEL_ENABLE**, a cancellation point is set in the calling thread after *type* is specified.

Legal values for *state* are **PTHREAD_CANCEL_ENABLE** and **PTHREAD_CANCEL_DISABLE**. Legal values for *type* are **PTHREAD_CANCEL_DEFERRED** and **PTHREAD_CANCEL_ASYNCHRONOUS**. The cancellation state and type for newly created threads, including the thread in which **main( )** was first invoked, are **PTHREAD_CANCEL_ENABLE** and **PTHREAD_CANCEL_DEFERRED**, respectively.

pthread_testcancel

pthread_testcancel( ) creates a cancellation point in the calling thread; it has no effect if cancellation is disabled.

pthread_cleanup_push

pthread_cleanup_push( ) pushes the specified cancellation cleanup handler routine, *handler*, onto the cancellation cleanup stack of the calling thread. When the thread exits, implicitly or explicitly, or is cancelled, its cancellation cleanup handler is popped from the cancellation cleanup stack and invoked with the argument *arg*. The thread acts upon a cancellation request, or the thread calls **pthread_cleanup_pop( )** with a non-zero *execute* argument.

pthread_cleanup_pop

pthread_cleanup_pop( ) removes the cleanup handler routine at the top of the cancellation cleanup stack of the calling thread and executes it if *execute* is non-zero.

If there are any calls to **pthread_cleanup_push( )** or **pthread_cleanup_pop( )** made without the matching call after the jump buffer is full, the effect of calling **longjmp**(3C) or **siglongjmp**(3C) is undefined.

Calls to **longjmp( )** or **siglongjmp( )** from within a cancellation cleanup handler is also undefined unless the jump buffer was also filled in the cancellation cleanup handler.

RETURN VALUES

If successful, **pthread_cancel( ) pthread_setcancelstate( )** and **pthread_setcanceltype( )** returns **0**; otherwise, an error number is returned.

pthread_testcancel( ), **pthread_cleanup_push( )**. and **pthread_cleanup_pop( )** are statements and do not return anything.

ERRORS

For each of the following conditions, **pthread_cancel( )** returns the corresponding error number if the condition is detected:

ESRCH No thread was found corresponding to that specified by the *target_thread* ID.

For each of the following conditions, **pthread_setcancelstate( )** returns the corresponding error if the condition is detected:

EINVAL The specified state is not **PTHREAD_CANCEL_ENABLE** or **PTHREAD_CANCEL_DISABLE**.

For each of the following conditions, **pthread_setcanceltype( )** returns the corresponding error if the condition is detected:

EINVAL The specified type is not **PTHREAD_CANCEL_DEFERRED** or **PTHREAD_CANCEL_ASYNCHRONOUS**.

SEE ALSO

condition(3T), **pthread_exit**(3T), **pthread_join**(3T), **setjmp**(3C)

NOTES

Please see **Intro**(3) for the notion of cancel-safety, Deferred-cancel-safety, and Asynchronous-cancel-safety. All libraries that have cancellation points but do not push/pop cancellation cleanup handlers are cancel-unsafe. If they push/pop cancellation handlers around cancellation points, they would become Deferred-cancel-safe, but could still be Asynchronous-cancel-unsafe.

In general, on Solaris, unless stated otherwise, all libraries are Asynchronous-cancel-unsafe and they may always remain so, because it may be too expensive for the common case (which is deferred cancellation) to make them Asynchronous-cancel-safe.

Libraries that do not have cancellation points are, by definition, Deferred-cancel-safe. Libraries that do have cancellation points but do not acquire any resources, such as locks or memory around these cancellation points, are also Deferred-cancel-safe. Those libraries which acquire locks and/or other resources before cancellation points are

Deferred-cancel-unsafe. Currently, there does not exist any labeling of libraries on Solaris about their cancel-safety status.

Applications can ensure cancel-safety of libraries by disabling cancellation before entering the library and restoring the old cancellation state on exit from the library.

Solaris threads do not offer this functionality.

Use of asynchronous cancellation while holding resources that need to be released may result in resource loss. Similarly, cancellation scopes may be safely manipulated (pushed and popped) only when the thread is in the deferred or disabled cancellation states.

For every **push( )** there must be the same number of **pop( )**s to compile the application.

EXAMPLES

The following is a short C++ example that shows the pushing/popping of cancellation handlers, the disabling/enabling of cancellation, the usage of **pthread_testcancel( )**, etc. The cancellation handler is "**free_res( )**," which is a dummy function that simply prints a message in this example, but in a real application would actually free resources. The function "**f2( )**" is called from the main thread, and it goes deep into its call stack by calling itself recursively.

Before **f2( )** starts running, the newly created thread has most likely posted a cancellation on the main thread since the main thread calls **thr_yield( )** right after creating thread2. Since cancellation has been disabled in the main thread initially, via the call to **pthread_setcancelstate( )**, the call to **f2( )** from **main( )** proceeds fine with "**X**" being constructed at each recursive call, although the main thread has a pending cancellation. Now, when **f2( )** is called for the fifty-first time (i.e., when "i == 50"), **f2( )** enables cancellation by calling **pthread_setcancelstate( )** and then establishes a cancellation point for itself by calling **pthread_testcancel( )**.

Instead of **pthread_testcancel( )**, there could have been a call to a cancellation point such as **read**(2) or **write**(2), which would have a similar effect (i.e., cause the caller to get cancelled at this point since there is a pending cancellation). Hence, the **main( )** thread gets cancelled at the fifty-first iteration and then all the cleanup handlers that were pushed, are called in sequence; this is indicated by the calls to **free_res( )** and the calls to the destructor for "**X**". At each level, the C++ runtime calls the destructor for **X** and then the cancellation handler, **free_res( )**. The print messages from **free_res( )** and **X**'s destructor show the sequence of calls.

At the end, the main thread is joined by thread2, and since the main thread has been cancelled, its return status is **PTHREAD_CANCELED,** which is obtained from the **pthread_join( )**. This status is printed out and then thread2 returns, killing the process, since it is the last thread in the process.

```
#include <pthread.h>
#include <string.h>
extern "C" void thr_yield(void);

extern "C" void printf(...);

struct X {
```

```
          int x;
          X(int i){x = i; printf("X(%d) constructed.0, i);}
          ~X(){ printf("X(%d) destroyed.0, x);}
};

void
free_res(void *i)
{
          printf("Freeing '%d'0,i);
}

char* f2(int i)
{
   try {
          X dummy(i);
          pthread_cleanup_push(free_res, (void *)i);
          if (i == 50){
             pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
             pthread_testcancel();
          }
          f2(i+1);
          pthread_cleanup_pop(0);
   }
   catch (int) {
          printf("Error: In handler.0);
   }
   return "f2";
}

void *
thread2(void *tid)
{
          void *sts;

          printf("I am new thread :%d0, pthread_self());

          pthread_cancel((pthread_t)tid);

          pthread_join((pthread_t)tid, &sts);

          printf("main thread cancelled due to %d0, sts);

          return (sts);

}
```

```
main()
{
   pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
   pthread_create(NULL, NULL, thread2, (void *)pthread_self());
   thr_yield();
   printf("Returned from %s0, f2(0));
}
```