NAME

pthread_cleanup_push, pthread_cleanup_pop, pthread_cleanup_push_defer_np, pthread_cleanup_pop_restore_np − install and remove cleanup handlers

SYNOPSIS

#include <pthread.h>

void pthread_cleanup_push(void (*_routine_**) (void *), void ***_arg_**);**

void pthread_cleanup_pop(int _execute_**);**

void pthread_cleanup_push_defer_np(void (*_routine_**) (void *), void ***_arg_**);**

void pthread_cleanup_pop_restore_np(int _execute_**);**

DESCRIPTION

Cleanup handlers are functions that get called when a thread terminates, either by calling **pthread_exit**(3) or because of cancellation. Cleanup handlers are installed and removed following a stack-like discipline.

The purpose of cleanup handlers is to free the resources that a thread may hold at the time it terminates. In particular, if a thread exits or is cancelled while it owns a locked mutex, the mutex will remain locked forever and prevent other threads from executing normally. The best way to avoid this is, just before locking the mutex, to install a cleanup handler whose effect is to unlock the mutex. Cleanup handlers can be used similarly to free blocks allocated with **malloc**(3) or close file descriptors on thread termination.

pthread_cleanup_push installs the *routine* function with argument *arg* as a cleanup handler. From this point on to the matching **pthread_cleanup_pop**, the function *routine* will be called with arguments *arg* when the thread terminates, either through **pthread_exit**(3) or by cancellation. If several cleanup handlers are active at that point, they are called in LIFO order: the most recently installed handler is called first.

pthread_cleanup_pop removes the most recently installed cleanup handler. If the *execute* argument is not 0, it also executes the handler, by calling the *routine* function with arguments *arg*. If the *execute* argument is 0, the handler is only removed but not executed.

Matching pairs of **pthread_cleanup_push** and **pthread_cleanup_pop** must occur in the same function, at the same level of block nesting. Actually, **pthread_cleanup_push** and **pthread_cleanup_pop** are macros, and the expansion of **pthread_cleanup_push** introduces an open brace **{** with the matching closing brace **}** being introduced by the expansion of the matching **pthread_cleanup_pop**.

pthread_cleanup_push_defer_np is a non-portable extension that combines **pthread_cleanup_push** and **pthread_setcanceltype**(3). It pushes a cleanup handler just as **pthread_cleanup_push** does, but also saves the current cancellation type and sets it to deferred cancellation. This ensures that the cleanup mechanism is effective even if the thread was initially in asynchronous cancellation mode.

pthread_cleanup_pop_restore_np pops a cleanup handler introduced by **pthread_cleanup_push_defer_np**, and restores the cancellation type to its value at the time **pthread_cleanup_push_defer_np** was called.

pthread_cleanup_push_defer_np and **pthread_cleanup_pop_restore_np** must occur in matching pairs, at the same level of block nesting.

The following sequence

```
pthread_cleanup_push_defer_np(routine, arg);
pthread_cleanup_pop_defer_np(execute);
```

is functionally equivalent to (but more compact and more efficient than)

```
{ int oldtype;
  pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, &oldtype);
  pthread_cleanup_push(routine, arg);
  ...
  pthread_cleanup_pop(execute);
  pthread_setcanceltype(oldtype, NULL);
}
```

RETURN VALUE

None.

ERRORS

None.

AUTHOR

Xavier Leroy <Xavier.Leroy@inria.fr>

SEE ALSO

pthread_exit(3), **pthread_cancel**(3), **pthread_setcanceltype**(3).

EXAMPLE

Here is how to lock a mutex *mut* in such a way that it will be unlocked if the thread is canceled while *mut* is locked:

```
pthread_cleanup_push(pthread_mutex_unlock, (void *) &mut);
pthread_mutex_lock(&mut);
/* do some work */
pthread_mutex_unlock(&mut);
pthread_cleanup_pop(0);
```

Equivalently, the last two lines can be replaced by

```
pthread_cleanup_pop(1);
```

Notice that the code above is safe only in deferred cancellation mode (see **pthread_setcanceltype**(3)). In asynchronous cancellation mode, a cancellation can occur between **pthread_cleanup_push** and **pthread_mutex_lock**, or between **pthread_mutex_unlock** and **pthread_cleanup_pop**, resulting in both cases in the thread trying to unlock a mutex not locked by the current thread. This is the main reason why asynchronous cancellation is difficult to use.

If the code above must also work in asynchronous cancellation mode, then it must switch to deferred mode for locking and unlocking the mutex:

```
pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, &oldtype);
pthread_cleanup_push(pthread_mutex_unlock, (void *) &mut);
pthread_mutex_lock(&mut);
/* do some work */
pthread_cleanup_pop(1);
pthread_setcanceltype(oldtype, NULL);
```

The code above can be rewritten in a more compact and more efficient way, using the non-portable functions **pthread_cleanup_push_defer_np** and **pthread_cleanup_pop_restore_np**:

```
pthread_cleanup_push_restore_np(pthread_mutex_unlock, (void *) &mut);
pthread_mutex_lock(&mut);
/* do some work */
pthread_cleanup_pop_restore_np(1);
```