Rocky Enterprise Linux 9.2 Manual Pages on command 'pthread_cleanup_push.3'

$ man pthread_cleanup_push.3

PTHREAD_CLEANUP_PUSH(3)   Linux Programmer's Manual   PTHREAD_CLEANUP_PUSH(3)

NAME

pthread_cleanup_push, pthread_cleanup_pop - push and pop thread cancellation clean-up han?

dlers

SYNOPSIS

```
#include <pthread.h>

void pthread_cleanup_push(void (*routine)(void *),
                          void *arg);

void pthread_cleanup_pop(int execute);
```

Compile and link with -pthread.

DESCRIPTION

These functions manipulate the calling thread's stack of thread-cancellation clean-up han?

dlers.   A clean-up handler is a function that is automatically executed when a thread is

canceled (or in various other circumstances described below); it might, for example, un?

lock a mutex so that it becomes available to other threads in the process.

The pthread_cleanup_push() function pushes routine onto the top of the stack of clean-up

handlers. When routine is later invoked, it will be given arg as its argument.

The pthread_cleanup_pop() function removes the routine at the top of the stack of clean-up

handlers, and optionally executes it if execute is nonzero.

A cancellation clean-up handler is popped from the stack and executed in the following

circumstances:

1. When a thread is canceled, all of the stacked clean-up handlers are popped and executed

   in the reverse of the order in which they were pushed onto the stack.

2. When a thread terminates by calling pthread_exit(3), all clean-up handlers are executed as described in the preceding point. (Clean-up handlers are not called if the thread terminates by performing a return from the thread start function.)

3. When a thread calls pthread_cleanup_pop() with a nonzero execute argument, the top-most clean-up handler is popped and executed.

POSIX.1 permits pthread_cleanup_push() and pthread_cleanup_pop() to be implemented as macros that expand to text containing '{' and '}', respectively. For this reason, the caller must ensure that calls to these functions are paired within the same function, and at the same lexical nesting level. (In other words, a clean-up handler is established only during the execution of a specified section of code.)

Calling longjmp(3) (siglongjmp(3)) produces undefined results if any call has been made to pthread_cleanup_push() or pthread_cleanup_pop() without the matching call of the pair since the jump buffer was filled by setjmp(3) (sigsetjmp(3)). Likewise, calling longjmp(3) (siglongjmp(3)) from inside a clean-up handler produces undefined results un?
less the jump buffer was also filled by setjmp(3) (sigsetjmp(3)) inside the handler.

RETURN VALUE

These functions do not return a value.

ERRORS

There are no errors.

ATTRIBUTES

For an explanation of the terms used in this section, see attributes(7).

Interface	Attribute	Value
pthread_cleanup_push(),	Thread safety	MT-Safe
pthread_cleanup_pop()		

CONFORMING TO

POSIX.1-2001, POSIX.1-2008.

NOTES

On Linux, the pthread_cleanup_push() and pthread_cleanup_pop() functions are implemented as macros that expand to text containing '{' and '}', respectively. This means that vari?
ables declared within the scope of paired calls to these functions will be visible within

only that scope.

POSIX.1 says that the effect of using return, break, continue, or goto to prematurely leave a block bracketed pthread_cleanup_push() and pthread_cleanup_pop() is undefined. Portable applications should avoid doing this.

EXAMPLES

The program below provides a simple example of the use of the functions described in this page. The program creates a thread that executes a loop bracketed by pthread_cleanup_push() and pthread_cleanup_pop(). This loop increments a global variable, cnt, once each second. Depending on what command-line arguments are supplied, the main thread sends the other thread a cancellation request, or sets a global variable that causes the other thread to exit its loop and terminate normally (by doing a return).

In the following shell session, the main thread sends a cancellation request to the other thread:

```
$ ./a.out
New thread started
cnt = 0
cnt = 1
Canceling thread
Called clean-up handler
Thread was canceled; cnt = 0
```

From the above, we see that the thread was canceled, and that the cancellation clean-up handler was called and it reset the value of the global variable cnt to 0.

In the next run, the main program sets a global variable that causes other thread to ter?
minate normally:

```
$ ./a.out x
New thread started
cnt = 0
cnt = 1
Thread terminated normally; cnt = 2
```

From the above, we see that the clean-up handler was not executed (because cleanup_pop_arg was 0), and therefore the value of cnt was not reset.

In the next run, the main program sets a global variable that causes the other thread to terminate normally, and supplies a nonzero value for cleanup_pop_arg:

```
    $ ./a.out x 1

    New thread started

    cnt = 0

    cnt = 1

    Called clean-up handler

    Thread terminated normally; cnt = 0
```

In the above, we see that although the thread was not canceled, the clean-up handler was executed, because the argument given to pthread_cleanup_pop() was nonzero.

Program source

```
#include <pthread.h>

#include <sys/types.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <errno.h>

#define handle_error_en(en, msg) \

        do { errno = en; perror(msg); exit(EXIT_FAILURE); } while (0)

static int done = 0;

static int cleanup_pop_arg = 0;

static int cnt = 0;

static void

cleanup_handler(void *arg)

{

    printf("Called clean-up handler\n");

    cnt = 0;

}

static void *

thread_start(void *arg)

{

    time_t start, curr;

    printf("New thread started\n");

    pthread_cleanup_push(cleanup_handler, NULL);

    curr = start = time(NULL);
    while (!done) {
        pthread_testcancel();           /* A cancellation point */
        if (curr < time(NULL)) {
            curr = time(NULL);
            printf("cnt = %d\n", cnt);  /* A cancellation point */
            cnt++;
        }
    }
    pthread_cleanup_pop(cleanup_pop_arg);
    return NULL;
}
int
main(int argc, char *argv[])
{
    pthread_t thr;
    int s;
    void *res;
    s = pthread_create(&thr, NULL, thread_start, NULL);
    if (s != 0)
        handle_error_en(s, "pthread_create");
    sleep(2);           /* Allow new thread to run a while */
    if (argc > 1) {
        if (argc > 2)
            cleanup_pop_arg = atoi(argv[2]);
        done = 1;
    } else {
        printf("Canceling thread\n");
        s = pthread_cancel(thr);
        if (s != 0)
            handle_error_en(s, "pthread_cancel");
    }
    s = pthread_join(thr, &res);
    if (s != 0)
```

```
           handle_error_en(s, "pthread_join");

       if (res == PTHREAD_CANCELED)
           printf("Thread was canceled; cnt = %d\n", cnt);
       else
           printf("Thread terminated normally; cnt = %d\n", cnt);
       exit(EXIT_SUCCESS);
   }
```

SEE ALSO

pthread_cancel(3), pthread_cleanup_push_defer_np(3), pthread_setcancelstate(3), pthread_testcancel(3), pthreads(7)

COLOPHON

This page is part of release 5.10 of the Linux man-pages project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at https://www.kernel.org/doc/man-pages/.