Full credit is given to the above companies including the OS that this PDF file was generated!

Red Hat Enterprise Linux Release 9.2 Manual Pages on 'pthread_cleanup_pop.3p' command

$ man pthread_cleanup_pop.3p

PTHREAD_CLEANUP_POP(3P) POSIX Programmer's Manual PTHREAD_CLEANUP_POP(3P)

PROLOG

This manual page is part of the POSIX Programmer's Manual. The Linux implementation of this interface may differ (consult the corresponding Linux manual page for details of Linux behavior), or the interface may not be implemented on Linux.

NAME

pthread_cleanup_pop, pthread_cleanup_push ? establish cancellation han? dlers

SYNOPSIS

```
#include <pthread.h>
void pthread_cleanup_pop(int execute);
void pthread_cleanup_push(void (*routine)(void*), void *arg);
```

DESCRIPTION

The pthread_cleanup_pop() function shall remove the routine at the top of the calling thread's cancellation cleanup stack and optionally in? voke it (if execute is non-zero).

The pthread_cleanup_push() function shall push the specified cancella? tion cleanup handler routine onto the calling thread's cancellation cleanup stack. The cancellation cleanup handler shall be popped from the cancellation cleanup stack and invoked with the argument arg when:

* The thread exits (that is, calls pthread_exit()).
* The thread acts upon a cancellation request.

* The thread calls pthread_cleanup_pop() with a non-zero execute ar?
gument.

It is unspecified whether pthread_cleanup_push() and pthread_cleanup_pop() are macros or functions. If a macro definition is suppressed in order to access an actual function, or a program defines an external identifier with any of these names, the behavior is unde? fined. The application shall ensure that they appear as statements, and in pairs within the same lexical scope (that is, the pthread_cleanup_push() macro may be thought to expand to a token list whose first token is '{' with pthread_cleanup_pop() expanding to a to? ken list whose last token is the corresponding '}').

The effect of calling longjmp() or siglongjmp() is undefined if there have been any calls to pthread_cleanup_push() or pthread_cleanup_pop() made without the matching call since the jump buffer was filled. The effect of calling longjmp() or siglongjmp() from inside a cancellation cleanup handler is also undefined unless the jump buffer was also filled in the cancellation cleanup handler.

The effect of the use of return, break, continue, and goto to prema? turely leave a code block described by a pair of pthread_cleanup_push() and pthread_cleanup_pop() functions calls is undefined.

RETURN VALUE

The pthread_cleanup_push() and pthread_cleanup_pop() functions shall not return a value.

ERRORS

No errors are defined.

These functions shall not return an error code of [EINTR].

The following sections are informative.

EXAMPLES

The following is an example using thread primitives to implement a can? celable, writers-priority read-write lock:

```
typedef struct {
    pthread_mutex_t lock;
    pthread_cond_t rcond,
```

```
        wcond;
    int lock_count; /* < 0 .. Held by writer. */
                    /* > 0 .. Held by lock_count readers. */
                    /* = 0 .. Held by nobody. */
    int waiting_writers; /* Count of waiting writers. */
} rwlock;
void
waiting_reader_cleanup(void *arg)
{
    rwlock *l;
    l = (rwlock *) arg;
    pthread_mutex_unlock(&l->lock);
}
void
lock_for_read(rwlock *l)
{
    pthread_mutex_lock(&l->lock);
    pthread_cleanup_push(waiting_reader_cleanup, l);
    while ((l->lock_count < 0) || (l->waiting_writers != 0))
        pthread_cond_wait(&l->rcond, &l->lock);
    l->lock_count++;
   /*
    * Note the pthread_cleanup_pop executes
    * waiting_reader_cleanup.
    */
    pthread_cleanup_pop(1);
}
void
release_read_lock(rwlock *l)
{
    pthread_mutex_lock(&l->lock);
    if (--l->lock_count == 0)
        pthread_cond_signal(&l->wcond);
```

```
    pthread_mutex_unlock(&l->lock);
}
void
waiting_writer_cleanup(void *arg)
{
    rwlock *l;
    l = (rwlock *) arg;
    if ((--l->waiting_writers == 0) && (l->lock_count >= 0)) {
       /*
        * This only happens if we have been canceled. If the
        * lock is not held by a writer, there may be readers who
        * were blocked because waiting_writers was positive; they
        * can now be unblocked.
        */
        pthread_cond_broadcast(&l->rcond);
    }
    pthread_mutex_unlock(&l->lock);
}
void
lock_for_write(rwlock *l)
{
    pthread_mutex_lock(&l->lock);
    l->waiting_writers++;
    pthread_cleanup_push(waiting_writer_cleanup, l);
    while (l->lock_count != 0)
        pthread_cond_wait(&l->wcond, &l->lock);
    l->lock_count = -1;
   /*
    * Note the pthread_cleanup_pop executes
    * waiting_writer_cleanup.
    */
    pthread_cleanup_pop(1);
}
```

```
void
release_write_lock(rwlock *l)
{
    pthread_mutex_lock(&l->lock);
    l->lock_count = 0;
    if (l->waiting_writers == 0)
        pthread_cond_broadcast(&l->rcond);
    else
        pthread_cond_signal(&l->wcond);
    pthread_mutex_unlock(&l->lock);
}
/*
 * This function is called to initialize the read/write lock.
 */
void
initialize_rwlock(rwlock *l)
{
    pthread_mutex_init(&l->lock, pthread_mutexattr_default);
    pthread_cond_init(&l->wcond, pthread_condattr_default);
    pthread_cond_init(&l->rcond, pthread_condattr_default);
    l->lock_count = 0;
    l->waiting_writers = 0;
}
reader_thread()
{
    lock_for_read(&lock);
    pthread_cleanup_push(release_read_lock, &lock);
   /*
    * Thread has read lock.
    */
    pthread_cleanup_pop(1);
}
writer_thread()
```

```
{
    lock_for_write(&lock);
    pthread_cleanup_push(release_write_lock, &lock);
   /*
    * Thread has write lock.
    */
pthread_cleanup_pop(1);
}
```

APPLICATION USAGE

The two routines that push and pop cancellation cleanup handlers, pthread_cleanup_push() and pthread_cleanup_pop(), can be thought of as left and right-parentheses. They always need to be matched.

RATIONALE

The restriction that the two routines that push and pop cancellation cleanup handlers, pthread_cleanup_push() and pthread_cleanup_pop(), have to appear in the same lexical scope allows for efficient macro or compiler implementations and efficient storage management. A sample im? plementation of these routines as macros might look like this:

```
#define pthread_cleanup_push(rtn,arg) { \
    struct _pthread_handler_rec __cleanup_handler, **__head; \
    __cleanup_handler.rtn = rtn; \
    __cleanup_handler.arg = arg; \
    (void) pthread_getspecific(_pthread_handler_key, &__head); \
    __cleanup_handler.next = *__head; \
    *__head = &__cleanup_handler;
#define pthread_cleanup_pop(ex) \
    *__head = __cleanup_handler.next; \
    if (ex) (*__cleanup_handler.rtn)(__cleanup_handler.arg); \
}
```

A more ambitious implementation of these routines might do even better by allowing the compiler to note that the cancellation cleanup handler is a constant and can be expanded inline.

This volume of POSIX.1?2017 currently leaves unspecified the effect of

calling longjmp() from a signal handler executing in a POSIX System In? terfaces function. If an implementation wants to allow this and give the programmer reasonable behavior, the longjmp() function has to call all cancellation cleanup handlers that have been pushed but not popped since the time setjmp() was called.

Consider a multi-threaded function called by a thread that uses sig? nals. If a signal were delivered to a signal handler during the opera? tion of qsort() and that handler were to call longjmp() (which, in turn, did not call the cancellation cleanup handlers) the helper threads created by the qsort() function would not be canceled. Instead, they would continue to execute and write into the argument array even though the array might have been popped off the stack.

Note that the specified cleanup handling mechanism is especially tied to the C language and, while the requirement for a uniform mechanism for expressing cleanup is language-independent, the mechanism used in other languages may be quite different. In addition, this mechanism is really only necessary due to the lack of a real exception mechanism in the C language, which would be the ideal solution.

There is no notion of a cancellation cleanup-safe function. If an ap? plication has no cancellation points in its signal handlers, blocks any signal whose handler may have cancellation points while calling async-unsafe functions, or disables cancellation while calling async-unsafe functions, all functions may be safely called from cancellation cleanup routines.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_cancel(), pthread_setcancelstate()

The Base Definitions volume of POSIX.1?2017, <pthread.h>

COPYRIGHT