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_cond_broadcast.3p' command

$ man pthread_cond_broadcast.3p

PTHREAD_COND_BROADCAST(3P) POSIX Programmer's ManualPTHREAD_COND_BROADCAST(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_cond_broadcast, pthread_cond_signal ? broadcast or signal a condition

SYNOPSIS

```
#include <pthread.h>

int pthread_cond_broadcast(pthread_cond_t *cond);

int pthread_cond_signal(pthread_cond_t *cond);
```

DESCRIPTION

These functions shall unblock threads blocked on a condition variable.

The pthread_cond_broadcast() function shall unblock all threads cur? rently blocked on the specified condition variable cond.

The pthread_cond_signal() function shall unblock at least one of the threads that are blocked on the specified condition variable cond (if any threads are blocked on cond).

If more than one thread is blocked on a condition variable, the sched? uling policy shall determine the order in which threads are unblocked.

When each thread unblocked as a result of a pthread_cond_broadcast() or

pthread_cond_signal() returns from its call to pthread_cond_wait() or pthread_cond_timedwait(), the thread shall own the mutex with which it called pthread_cond_wait() or pthread_cond_timedwait(). The thread(s) that are unblocked shall contend for the mutex according to the sched? uling policy (if applicable), and as if each had called pthread_mu? tex_lock().

The pthread_cond_broadcast() or pthread_cond_signal() functions may be called by a thread whether or not it currently owns the mutex that threads calling pthread_cond_wait() or pthread_cond_timedwait() have associated with the condition variable during their waits; however, if predictable scheduling behavior is required, then that mutex shall be locked by the thread calling pthread_cond_broadcast() or pthread_cond_signal().

The pthread_cond_broadcast() and pthread_cond_signal() functions shall have no effect if there are no threads currently blocked on cond.

The behavior is undefined if the value specified by the cond argument to pthread_cond_broadcast() or pthread_cond_signal() does not refer to an initialized condition variable.

RETURN VALUE

If successful, the pthread_cond_broadcast() and pthread_cond_signal() functions shall return zero; otherwise, an error number shall be re? turned to indicate the error.

ERRORS

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

The following sections are informative.

EXAMPLES

None.

APPLICATION USAGE

The pthread_cond_broadcast() function is used whenever the shared-vari? able state has been changed in a way that more than one thread can pro? ceed with its task. Consider a single producer/multiple consumer prob? lem, where the producer can insert multiple items on a list that is ac? cessed one item at a time by the consumers. By calling the

pthread_cond_broadcast() function, the producer would notify all con?
sumers that might be waiting, and thereby the application would receive
more throughput on a multi-processor. In addition, pthread_cond_broad?
cast() makes it easier to implement a read-write lock. The
pthread_cond_broadcast() function is needed in order to wake up all
waiting readers when a writer releases its lock. Finally, the two-phase
commit algorithm can use this broadcast function to notify all clients
of an impending transaction commit.

It is not safe to use the pthread_cond_signal() function in a signal
handler that is invoked asynchronously. Even if it were safe, there
would still be a race between the test of the Boolean
pthread_cond_wait() that could not be efficiently eliminated.

Mutexes and condition variables are thus not suitable for releasing a
waiting thread by signaling from code running in a signal handler.

RATIONALE

If an implementation detects that the value specified by the cond argu?
ment to pthread_cond_broadcast() or pthread_cond_signal() does not re?
fer to an initialized condition variable, it is recommended that the
function should fail and report an [EINVAL] error.

Multiple Awakenings by Condition Signal

On a multi-processor, it may be impossible for an implementation of
pthread_cond_signal() to avoid the unblocking of more than one thread
blocked on a condition variable. For example, consider the following
partial implementation of pthread_cond_wait() and pthread_cond_sig?
nal(), executed by two threads in the order given. One thread is trying
to wait on the condition variable, another is concurrently executing
pthread_cond_signal(), while a third thread is already waiting.

```
pthread_cond_wait(mutex, cond):
    value = cond->value; /* 1 */
    pthread_mutex_unlock(mutex); /* 2 */
    pthread_mutex_lock(cond->mutex); /* 10 */
    if (value == cond->value) { /* 11 */
        me->next_cond = cond->waiter;
```

```
            cond->waiter = me;
            pthread_mutex_unlock(cond->mutex);
            unable_to_run(me);
        } else
            pthread_mutex_unlock(cond->mutex); /* 12 */
        pthread_mutex_lock(mutex); /* 13 */
    pthread_cond_signal(cond):
        pthread_mutex_lock(cond->mutex); /* 3 */
        cond->value++; /* 4 */
        if (cond->waiter) { /* 5 */
            sleeper = cond->waiter; /* 6 */
            cond->waiter = sleeper->next_cond; /* 7 */
            able_to_run(sleeper); /* 8 */
        }
        pthread_mutex_unlock(cond->mutex); /* 9 */
```

The effect is that more than one thread can return from its call to pthread_cond_wait() or pthread_cond_timedwait() as a result of one call to pthread_cond_signal(). This effect is called ``spurious wakeup''. Note that the situation is self-correcting in that the number of threads that are so awakened is finite; for example, the next thread to call pthread_cond_wait() after the sequence of events above blocks. While this problem could be resolved, the loss of efficiency for a fringe condition that occurs only rarely is unacceptable, especially given that one has to check the predicate associated with a condition variable anyway. Correcting this problem would unnecessarily reduce the degree of concurrency in this basic building block for all higher-level synchronization operations.

An added benefit of allowing spurious wakeups is that applications are forced to code a predicate-testing-loop around the condition wait. This also makes the application tolerate superfluous condition broad? casts or signals on the same condition variable that may be coded in some other part of the application. The resulting applications are thus more robust. Therefore, POSIX.1?2008 explicitly documents that spurious

wakeups may occur.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_cond_destroy(), pthread_cond_timedwait()

The Base Definitions volume of POSIX.1?2017, Section 4.12, Memory Syn?

chronization, <pthread.h>

COPYRIGHT