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_timedwait.3p' command

$ man pthread_cond_timedwait.3p

PTHREAD_COND_TIMEDWAIT(3P) POSIX Programmer's ManualPTHREAD_COND_TIMEDWAIT(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_timedwait, pthread_cond_wait ? wait on a condition

SYNOPSIS

```
#include <pthread.h>
int pthread_cond_timedwait(pthread_cond_t *restrict cond,
    pthread_mutex_t *restrict mutex,
    const struct timespec *restrict abstime);
int pthread_cond_wait(pthread_cond_t *restrict cond,
    pthread_mutex_t *restrict mutex);
```

DESCRIPTION

The pthread_cond_timedwait() and pthread_cond_wait() functions shall block on a condition variable. The application shall ensure that these functions are called with mutex locked by the calling thread; other? wise, an error (for PTHREAD_MUTEX_ERRORCHECK and robust mutexes) or un? defined behavior (for other mutexes) results.

These functions atomically release mutex and cause the calling thread to block on the condition variable cond; atomically here means ``atomi?

cally with respect to access by another thread to the mutex and then the condition variable''. That is, if another thread is able to acquire the mutex after the about-to-block thread has released it, then a sub? sequent call to pthread_cond_broadcast() or pthread_cond_signal() in that thread shall behave as if it were issued after the about-to-block thread has blocked.

Upon successful return, the mutex shall have been locked and shall be owned by the calling thread. If mutex is a robust mutex where an owner terminated while holding the lock and the state is recoverable, the mu? tex shall be acquired even though the function returns an error code.

When using condition variables there is always a Boolean predicate in? volving shared variables associated with each condition wait that is true if the thread should proceed. Spurious wakeups from the pthread_cond_timedwait() or pthread_cond_wait() functions may occur. Since the return from pthread_cond_timedwait() or pthread_cond_wait() does not imply anything about the value of this predicate, the predi? cate should be re-evaluated upon such return.

When a thread waits on a condition variable, having specified a partic? ular mutex to either the pthread_cond_timedwait() or the pthread_cond_wait() operation, a dynamic binding is formed between that mutex and condition variable that remains in effect as long as at least one thread is blocked on the condition variable. During this time, the effect of an attempt by any thread to wait on that condition variable using a different mutex is undefined. Once all waiting threads have been unblocked (as by the pthread_cond_broadcast() operation), the next wait operation on that condition variable shall form a new dynamic binding with the mutex specified by that wait operation. Even though the dynamic binding between condition variable and mutex may be removed or replaced between the time a thread is unblocked from a wait on the condition variable and the time that it returns to the caller or begins cancellation cleanup, the unblocked thread shall always re-acquire the mutex specified in the condition wait operation call from which it is returning.

A condition wait (whether timed or not) is a cancellation point. When the cancelability type of a thread is set to PTHREAD_CANCEL_DEFERRED, a side-effect of acting upon a cancellation request while in a condition wait is that the mutex is (in effect) re-acquired before calling the first cancellation cleanup handler. The effect is as if the thread were unblocked, allowed to execute up to the point of returning from the call to pthread_cond_timedwait() or pthread_cond_wait(), but at that point notices the cancellation request and instead of returning to the caller of pthread_cond_timedwait() or pthread_cond_wait(), starts the thread cancellation activities, which includes calling cancellation cleanup handlers.

A thread that has been unblocked because it has been canceled while blocked in a call to pthread_cond_timedwait() or pthread_cond_wait() shall not consume any condition signal that may be directed concur? rently at the condition variable if there are other threads blocked on the condition variable.

The pthread_cond_timedwait() function shall be equivalent to pthread_cond_wait(), except that an error is returned if the absolute time specified by abstime passes (that is, system time equals or ex? ceeds abstime) before the condition cond is signaled or broadcasted, or if the absolute time specified by abstime has already been passed at the time of the call. When such timeouts occur, pthread_cond_timed? wait() shall nonetheless release and re-acquire the mutex referenced by mutex, and may consume a condition signal directed concurrently at the condition variable.

The condition variable shall have a clock attribute which specifies the clock that shall be used to measure the time specified by the abstime argument. The pthread_cond_timedwait() function is also a cancellation point.

If a signal is delivered to a thread waiting for a condition variable, upon return from the signal handler the thread resumes waiting for the condition variable as if it was not interrupted, or it shall return zero due to spurious wakeup.

The behavior is undefined if the value specified by the cond or mutex argument to these functions does not refer to an initialized condition variable or an initialized mutex object, respectively.

RETURN VALUE

Except for [ETIMEDOUT], [ENOTRECOVERABLE], and [EOWNERDEAD], all these error checks shall act as if they were performed immediately at the be? ginning of processing for the function and shall cause an error return, in effect, prior to modifying the state of the mutex specified by mutex or the condition variable specified by cond.

Upon successful completion, a value of zero shall be returned; other? wise, an error number shall be returned to indicate the error.

ERRORS

These functions shall fail if:

ENOTRECOVERABLE

The state protected by the mutex is not recoverable.

EOWNERDEAD

The mutex is a robust mutex and the process containing the pre? vious owning thread terminated while holding the mutex lock. The mutex lock shall be acquired by the calling thread and it is up to the new owner to make the state consistent.

EPERM The mutex type is PTHREAD_MUTEX_ERRORCHECK or the mutex is a ro? bust mutex, and the current thread does not own the mutex.

The pthread_cond_timedwait() function shall fail if:

ETIMEDOUT

The time specified by abstime to pthread_cond_timedwait() has passed.

EINVAL The abstime argument specified a nanosecond value less than zero or greater than or equal to 1000 million.

These functions may fail if:

EOWNERDEAD

The mutex is a robust mutex and the previous owning thread ter? minated while holding the mutex lock. The mutex lock shall be acquired by the calling thread and it is up to the new owner to

make the state consistent.

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

The following sections are informative.

EXAMPLES

None.

APPLICATION USAGE

Applications that have assumed that non-zero return values are errors will need updating for use with robust mutexes, since a valid return for a thread acquiring a mutex which is protecting a currently inconsistent state is [EOWNERDEAD]. Applications that do not check the error returns, due to ruling out the possibility of such errors arising, should not use robust mutexes. If an application is supposed to work with normal and robust mutexes, it should check all return values for error conditions and if necessary take appropriate action.

RATIONALE

If an implementation detects that the value specified by the cond argument to pthread_cond_timedwait() or pthread_cond_wait() does not refer to an initialized condition variable, or detects that the value specified by the mutex argument to pthread_cond_timedwait() or pthread_cond_wait() does not refer to an initialized mutex object, it is recommended that the function should fail and report an [EINVAL] error.

Condition Wait Semantics

It is important to note that when pthread_cond_wait() and pthread_cond_timedwait() return without error, the associated predicate may still be false. Similarly, when pthread_cond_timedwait() returns with the timeout error, the associated predicate may be true due to an unavoidable race between the expiration of the timeout and the predicate state change.

The application needs to recheck the predicate on any return because it cannot be sure there is another thread waiting on the thread to handle the signal, and if there is not then the signal is lost. The burden is on the application to check the predicate.

Some implementations, particularly on a multi-processor, may sometimes cause multiple threads to wake up when the condition variable is sig? naled simultaneously on different processors.

In general, whenever a condition wait returns, the thread has to re- evaluate the predicate associated with the condition wait to determine whether it can safely proceed, should wait again, or should declare a timeout. A return from the wait does not imply that the associated predicate is either true or false.

It is thus recommended that a condition wait be enclosed in the equiva? lent of a ``while loop'' that checks the predicate.

Timed Wait Semantics

An absolute time measure was chosen for specifying the timeout parame? ter for two reasons. First, a relative time measure can be easily im? plemented on top of a function that specifies absolute time, but there is a race condition associated with specifying an absolute timeout on top of a function that specifies relative timeouts. For example, assume that clock_gettime() returns the current time and cond_rela? tive_timed_wait() uses relative timeouts:

```
clock_gettime(CLOCK_REALTIME, &now)
reltime = sleep_til_this_absolute_time -now;
cond_relative_timed_wait(c, m, &reltime);
```

If the thread is preempted between the first statement and the last statement, the thread blocks for too long. Blocking, however, is irrel? evant if an absolute timeout is used. An absolute timeout also need not be recomputed if it is used multiple times in a loop, such as that en? closing a condition wait.

For cases when the system clock is advanced discontinuously by an oper? ator, it is expected that implementations process any timed wait expir? ing at an intervening time as if that time had actually occurred.

Cancellation and Condition Wait

A condition wait, whether timed or not, is a cancellation point. That is, the functions pthread_cond_wait() or pthread_cond_timedwait() are points where a pending (or concurrent) cancellation request is noticed.

The reason for this is that an indefinite wait is possible at these points?whatever event is being waited for, even if the program is to? tally correct, might never occur; for example, some input data being awaited might never be sent. By making condition wait a cancellation point, the thread can be canceled and perform its cancellation cleanup handler even though it may be stuck in some indefinite wait.

A side-effect of acting on a cancellation request while a thread is blocked on a condition variable is to re-acquire the mutex before call? ing any of the cancellation cleanup handlers. This is done in order to ensure that the cancellation cleanup handler is executed in the same state as the critical code that lies both before and after the call to the condition wait function. This rule is also required when interfac? ing to POSIX threads from languages, such as Ada or C++, which may choose to map cancellation onto a language exception; this rule ensures that each exception handler guarding a critical section can always safely depend upon the fact that the associated mutex has already been locked regardless of exactly where within the critical section the ex? ception was raised. Without this rule, there would not be a uniform rule that exception handlers could follow regarding the lock, and so coding would become very cumbersome.

Therefore, since some statement has to be made regarding the state of the lock when a cancellation is delivered during a wait, a definition has been chosen that makes application coding most convenient and error free.

When acting on a cancellation request while a thread is blocked on a condition variable, the implementation is required to ensure that the thread does not consume any condition signals directed at that condi? tion variable if there are any other threads waiting on that condition variable. This rule is specified in order to avoid deadlock conditions that could occur if these two independent requests (one acting on a thread and the other acting on the condition variable) were not pro? cessed independently.

Mutexes are expected to be locked only for a few instructions. This practice is almost automatically enforced by the desire of programmers to avoid long serial regions of execution (which would reduce total ef? fective parallelism).

When using mutexes and condition variables, one tries to ensure that the usual case is to lock the mutex, access shared data, and unlock the mutex. Waiting on a condition variable should be a relatively rare sit? uation. For example, when implementing a read-write lock, code that ac? quires a read-lock typically needs only to increment the count of read? ers (under mutual-exclusion) and return. The calling thread would actu? ally wait on the condition variable only when there is already an ac? tive writer. So the efficiency of a synchronization operation is bounded by the cost of mutex lock/unlock and not by condition wait. Note that in the usual case there is no context switch.

This is not to say that the efficiency of condition waiting is unimpor? tant. Since there needs to be at least one context switch per Ada ren? dezvous, the efficiency of waiting on a condition variable is impor? tant. The cost of waiting on a condition variable should be little more than the minimal cost for a context switch plus the time to unlock and lock the mutex.

Features of Mutexes and Condition Variables

It had been suggested that the mutex acquisition and release be decou? pled from condition wait. This was rejected because it is the combined nature of the operation that, in fact, facilitates realtime implementa? tions. Those implementations can atomically move a high-priority thread between the condition variable and the mutex in a manner that is trans? parent to the caller. This can prevent extra context switches and pro? vide more deterministic acquisition of a mutex when the waiting thread is signaled. Thus, fairness and priority issues can be dealt with di? rectly by the scheduling discipline. Furthermore, the current condi? tion wait operation matches existing practice.

Scheduling Behavior of Mutexes and Condition Variables

Synchronization primitives that attempt to interfere with scheduling

policy by specifying an ordering rule are considered undesirable.

Threads waiting on mutexes and condition variables are selected to pro?
ceed in an order dependent upon the scheduling policy rather than in
some fixed order (for example, FIFO or priority). Thus, the scheduling
policy determines which thread(s) are awakened and allowed to proceed.

Timed Condition Wait

The pthread_cond_timedwait() function allows an application to give up
waiting for a particular condition after a given amount of time. An ex?
ample of its use follows:

```
(void) pthread_mutex_lock(&t.mn);
    t.waiters++;
    clock_gettime(CLOCK_REALTIME, &ts);
    ts.tv_sec += 5;
    rc = 0;
    while (! mypredicate(&t) && rc == 0)
        rc = pthread_cond_timedwait(&t.cond, &t.mn, &ts);
    t.waiters--;
    if (rc == 0 || mypredicate(&t))
        setmystate(&t);
(void) pthread_mutex_unlock(&t.mn);
```

By making the timeout parameter absolute, it does not need to be recom?
puted each time the program checks its blocking predicate. If the time?
out was relative, it would have to be recomputed before each call.
This would be especially difficult since such code would need to take
into account the possibility of extra wakeups that result from extra
broadcasts or signals on the condition variable that occur before ei?
ther the predicate is true or the timeout is due.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_cond_broadcast()

The Base Definitions volume of POSIX.1?2017, Section 4.12, Memory Syn?
chronization, <pthread.h>