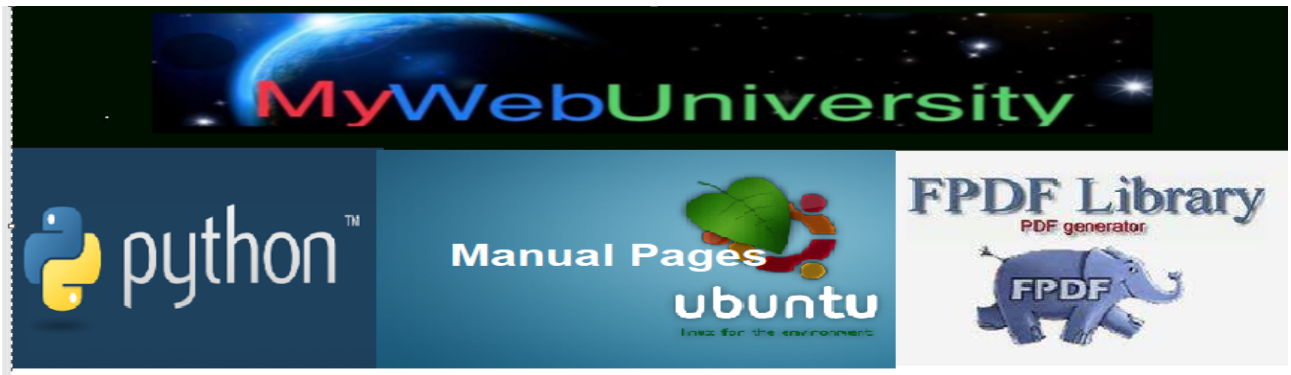

Full credit is given to the above companies including the OS that this PDF file was generated!

Linux Ubuntu 26.04 Manual Pages on command 'pthread_cond_wait.3'

$ man pthread_cond_wait.3

pthread_cond_init(3) Library Functions Manual pthread_cond_init(3)

NAME

pthread_cond_init, pthread_cond_signal, pthread_cond_broadcast, pthread_cond_wait, pthread_cond_timedwait, pthread_cond_destroy - operations on conditions

SYNOPSIS

```
#include <pthread.h>

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

int pthread_cond_init(pthread_cond_t *cond,
                      pthread_condattr_t *cond_attr);

int pthread_cond_signal(pthread_cond_t *cond);

int pthread_cond_broadcast(pthread_cond_t *cond);

int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);

int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex,
                      const struct timespec *abstime);

int pthread_cond_destroy(pthread_cond_t *cond);
```

DESCRIPTION

A condition (short for "condition variable") is a synchronization device that allows threads to suspend execution and relinquish the processors until some predicate on shared data is satisfied. The basic operations on conditions are: signal the condition (when the predicate becomes true), and wait for the condition, suspending the thread execution until another thread signals the condition.

A condition variable must always be associated with a mutex, to avoid the race condition where a thread prepares to wait on a condition variable and another thread signals the con?

dition just before the first thread actually waits on it.

pthread_cond_init() initializes the condition variable cond, using the condition attributes specified in cond_attr, or default attributes if cond_attr is NULL. The LinuxThreads imple? mentation supports no attributes for conditions, hence the cond_attr parameter is actually ignored.

Variables of type pthread_cond_t can also be initialized statically, using the constant PTHREAD_COND_INITIALIZER.

pthread_cond_signal() restarts one of the threads that are waiting on the condition variable cond. If no threads are waiting on cond, nothing happens. If several threads are waiting on cond, exactly one is restarted, but it is not specified which.

pthread_cond_broadcast() restarts all the threads that are waiting on the condition variable cond. Nothing happens if no threads are waiting on cond.

pthread_cond_wait() atomically unlocks the mutex (as per pthread_unlock_mutex()) and waits for the condition variable cond to be signaled. The thread execution is suspended and does not consume any CPU time until the condition variable is signaled. The mutex must be locked by the calling thread on entrance to pthread_cond_wait(). Before returning to the calling thread, pthread_cond_wait() re-acquires mutex (as per pthread_mutex_lock()).

Unlocking the mutex and suspending on the condition variable is done atomically. Thus, if all threads always acquire the mutex before signaling the condition, this guarantees that the condition cannot be signaled (and thus ignored) between the time a thread locks the mu? tex and the time it waits on the condition variable. See CAVEATS below.

pthread_cond_timedwait() atomically unlocks mutex and waits on cond, as pthread_cond_wait() does, but it also bounds the duration of the wait. If cond has not been signaled within the amount of time specified by abstime, the mutex mutex is re-acquired and pthread_cond_timed? wait() returns the error ETIMEDOUT. The abstime parameter specifies an absolute time, with the same origin as time(2) and gettimeofday(2): an abstime of 0 corresponds to 00:00:00 GMT, January 1, 1970.

pthread_cond_destroy() destroys a condition variable, freeing the resources it might hold. No threads must be waiting on the condition variable on entrance to pthread_cond_destroy(). In the LinuxThreads implementation, no resources are associated with condition variables, thus pthread_cond_destroy() actually does nothing except checking that the condition has no waiting threads.

CANCELLATION

pthread_cond_wait() and pthread_cond_timedwait() are cancelation points. If a thread is cancelled while suspended in one of these functions, the thread immediately resumes execu? tion, then locks again the mutex argument to pthread_cond_wait() and pthread_cond_timed? wait(), and finally executes the cancelation. Consequently, cleanup handlers are assured that mutex is locked when they are called.

ASYNC-SIGNAL SAFETY

The condition functions are not async-signal safe, and should not be called from a signal handler. In particular, calling pthread_cond_signal() or pthread_cond_broadcast() from a signal handler may deadlock the calling thread.

RETURN VALUE

All condition variable functions return 0 on success and a non-zero error code on error.

ERRORS

pthread_cond_init(), pthread_cond_signal(), pthread_cond_broadcast(), and pthread_cond_wait() never return an error code.

The pthread_cond_timedwait() function returns the following error codes on error:

ETIMEDOUT

The condition variable was not signaled until the timeout specified by abstime .

The pthread_cond_destroy() function returns the following error code on error:

EBUSY Some threads are currently waiting on cond .

SEE ALSO

pthread_condattr_init(3), pthread_mutex_lock(3), pthread_mutex_unlock(3), gettimeofday(2), nanosleep(2).

CAVEATS

The implementation of the provided functions until glibc 2.25 used an internal data lock. This lock did not support priority-inheritance and was subject to unbounded priority inver? sion, visible on a real-time system.

After the rewrite of the implementation in glibc 2.25 the usage of internal lock changed. The internal lock is always acquired by the signaling functions pthread_cond_signal() and pthread_cond_broadcast(). The waiting function acquires the lock if the waiting process was interrupted. The interruption can be caused for instance by a specified timeout, and de? noted by the error value ETIMEDOUTA, or by a received signal, which is denoted by the error value EINTR.

EXAMPLE

Consider two shared variables x and y , protected by the mutex mut, and a condition variable cond that is to be signaled whenever x becomes greater than y.

```
int x,y;
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

Waiting until x is greater than y is performed as follows:

```
pthread_mutex_lock(&mut);
while (x <= y) {
        pthread_cond_wait(&cond, &mut);
}
/* operate on x and y */
pthread_mutex_unlock(&mut);
```

Modifications on x and y that may cause x to become greater than y should signal the condi? tion if needed:

```
pthread_mutex_lock(&mut);
/* modify x and y */
if (x > y) pthread_cond_broadcast(&cond);
pthread_mutex_unlock(&mut);
```

If it can be proved that at most one waiting thread needs to be waken up (for instance, if there are only two threads communicating through x and y), pthread_cond_signal() can be used as a slightly more efficient alternative to pthread_cond_broadcast(). In doubt, use pthread_cond_broadcast().

To wait for x to become greater than y with a timeout of 5 seconds, do:

```
struct timeval now;
struct timespec timeout;
int retcode;
pthread_mutex_lock(&mut);
gettimeofday(&now);
timeout.tv_sec = now.tv_sec + 5;
timeout.tv_nsec = now.tv_usec * 1000;
retcode = 0;
while (x <= y && retcode != ETIMEDOUT) {
```

```
        retcode = pthread_cond_timedwait(&cond, &mut, &timeout);
    }
    if (retcode == ETIMEDOUT) {
        /* timeout occurred */
    } else {
        /* operate on x and y */
    }
    pthread_mutex_unlock(&mut);
```