NAME

condition, pthread_cond_init, pthread_cond_wait, pthread_cond_timedwait, pthread_cond_signal, pthread_cond_broadcast, pthread_cond_destroy, cond_init, cond_wait, cond_timedwait, cond_signal, cond_broadcast, cond_destroy – condition variables

SYNOPSIS

POSIX

cc [*flag* ...] *file* ... **–lpthread** [*library* ...]

#include <pthread.h>
int pthread_cond_init(pthread_cond_t **cond,* **const pthread_condattr_t** **attr***);**
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_signal(pthread_cond_t **cond***);**
int pthread_cond_broadcast(pthread_cond_t **cond***);**
int pthread_cond_destroy(pthread_cond_t **cond***);**

Solaris

cc [*flag* ...] *file* ... **–lthread** [*library* ...]

#include <thread.h>
#include <synch.h>
int cond_init(cond_t **cvp*, **int** *type*, **int** *arg***);**
int cond_wait(cond_t **cvp*, **mutex_t** **mp***);**
int cond_timedwait(cond_t **cvp*, **mutex_t** **mp*, **timestruc_t** **abstime***);**
int cond_signal(cond_t **cvp***);**
int cond_broadcast(cond_t **cvp***);**
int cond_destroy(cond_t **cvp***);**

MT-LEVEL

MT-Safe

DESCRIPTION

Occasionally, a thread running within a mutex needs to wait for an event, in which case, it blocks or sleeps. When a thread is waiting for another thread to communicate its disposition, it uses a condition variable in conjunction with a mutex. Although a mutex is exclusive and the code it protects is sharable (at certain moments), condition variables enable the synchronization of differing events that share a mutex, but not necessarily data. Several condition variables may be used by threads to signal each other when a task is complete, which then allows the next waiting thread to take ownership of the mutex.

A condition variable enables threads to atomically block and test the condition under the protection of a mutual exclusion lock (mutex) until the condition is satisfied. If the condition is false, a thread blocks on a condition variable and atomically releases the mutex that is waiting for the condition to change. If another thread changes the condition, it may wake up waiting threads by signaling the associated condition variable. The waiting threads, upon awakening, reacquire the mutex and re-evaluate the condition.

Initialize

Condition variables and mutexes should be global. Condition variables that are allocated in writable memory can synchronize threads among processes if they are shared by the cooperating processes (see **mmap**(2)) and are initialized for this purpose.

The scope of a condition variable is either intra-process or inter-process. This is dependent upon whether the argument is passed implicitly or explicitly to the initialization of that condition variable. A condition variable does not need to be explicitly initialized. A condition variable is initialized with all zeros, by default, and its scope is set to within the calling process. For inter-process synchronization, a condition variable must be initialized once, and only once, before use.

A condition variable must not be simultaneously initialized by multiple threads or re-initialized while in use by other threads.

Condition variables attributes may be set to the default or customized at initialization. POSIX threads even allow the default values to be customized. Establishing these attributes varies depending upon whether POSIX or Solaris threads are used. Similar to the distinctions between POSIX and Solaris thread creation, POSIX condition variables implement the default, intra-process, unless an attribute object is modified for inter-process prior to the initialization of the condition variable. Solaris condition variables also implement as the default, intra-process; however, they set this attribute according to the argument, *type*, passed to their initialization function.

POSIX Initialize

POSIX condition variables mutexes, and threads use attributes objects in the same manner; they are initialized with the configuration of an attributes object (see **pthread_condattr_init**(3T)). The **pthread_cond_init( )** function initializes the condition variable referenced by *cond* with attributes referenced by *attr*. If *attr* is **NULL**, the default condition variable attributes are used, which is the same as passing the address of a default condition variable attributes object. When the initialization is complete, the state of the condition variable is then initialized. If a default condition variable is used, then only threads created within the same process can operate on the initialized condition variable.

A condition variable can possess two different types of shared-scope behavior, which is determined by the second argument to **pthread_condattr_setpshared**(3T). This argument can be set to either of the following:

PTHREAD_PROCESS_PRIVATE The condition variable can synchronize threads only in this process. The **PTHREAD_PROCESS_PRIVATE** POSIX setting for process scope is equivalent to the **USYNC_THREAD** flag to **cond_init( )** in the Solaris API. This is the default.

PTHREAD_PROCESS_SHARED The condition variable can synchronize threads in this process and other processes. Only one process should initialize the condition variable. The **PTHREAD_PROCESS_SHARED** POSIX setting for system-wide scope is equivalent to the **USYNC_PROCESS** flag to **cond_init( )** in the Solaris API.

Initializing condition variables can also be accomplished by allocating-in zeroed memory (default), in which case, **PTHREAD_PROCESS_PRIVATE** is assumed. The same condition variable must not be simultaneously initialized by multiple threads nor re-initialized while in use by other threads.

If default condition variable attributes are used, statically allocated condition variables can be initialized by the macro **PTHREAD_COND_INITIALIZER.** The effect is the same as a dynamic initialization by a call to **pthread_cond_init( )** with parameter *attr* specified as **NULL**, except error checks are not performed.

Default condition variable initialization (intra-process):

```
pthread_cond_t          cvp;
pthread_condattr_t      cv_attr;

pthread_cond_init(&cvp, NULL);/* initialize cv with defaults */
   OR
pthread_condattr_init(&cv_attr);/* initialize cv_attr with defaults */
pthread_cond_init(&cvp, &cv_attr);      /* initialize cv with default cv_attr */
   OR
pthread_condattr_setpshared(&cv_attr, PTHREAD_PROCESS_PRIVATE);
pthread_cond_init(&cvp, &cv_attr);      /* initialize cv with defaults */
   OR
pthread_cond_t  cond  = PTHREAD_COND_INITIALIZER;
   OR
pthread_cond_t cond;
cond = calloc(1, sizeof (pthread_cond_t));
```

Customized condition variable initialization (inter-process):

```
pthread_condattr_init(&cv_attr);/* initialize cv_attr with defaults */
pthread_condattr_setpshared(&cv_attr, PTHREAD_PROCESS_SHARED);
pthread_cond_init(&cvp, &cv_attr);      /* initialize cv with inter-process scope */
```

Solaris Initialize

cond_init( ) initializes the condition variable pointed to by *cvp*. A condition variable can have several different types of behavior, specified by *type*. No current type uses *arg* although a future type may specify additional behavior parameters via *arg*. *type* may be one of the following:

USYNC_THREAD The condition variable can synchronize threads only in this process. The **USYNC_THREAD** Solaris condition variable type for process scope is equivalent to the POSIX condition variable attribute setting **PTHREAD_PROCESS_PRIVATE.** *arg* is ignored.

USYNC_PROCESS The condition variable can synchronize threads in this process and other processes. Only one process should initialize the condition variable. The **USYNC_PROCESS** Solaris condition variable type for system-wide scope is equivalent to the POSIX condition variable attribute setting **PTHREAD_PROCESS_SHARED.** *arg* is ignored.

Initializing condition variables can also be accomplished by allocating in zeroed memory, in which case, a *type* of **USYNC_THREAD** is assumed.

If default condition variable attributes are used, statically allocated condition variables can be initialized by the macro **DEFAULTCV.**

Default condition variable initialization (intra-process):

```
cond_t    cvp;

cond_init(&cvp, NULL, NULL); /* initialize condition variable with default */
 OR
cond_init(&cvp, USYNC_THREAD, NULL);
 OR
cond_t  cond  = DEFAULTCV;
```

Customized condition variable initialization (inter-process):

```
cond_init(&cvp, USYNC_PROCESS, NULL); /* initialize cv with inter-process scope */
```

Condition Wait

The condition wait interface allows a thread to wait for a condition and atomically release the associated mutex that it needs to hold to check the condition. The thread waits for another thread to make the condition true and that thread's resulting call to signal and wakeup the waiting thread.

POSIX Wait

pthread_cond_wait( ) and **pthread_cond_timedwait( )** block on a condition variable, which atomically release the mutex pointed to by *mp* and cause the calling thread to block on the condition variable pointed to by *cond*. The blocked thread may be awakened by **pthread_cond_signal( )**, **pthread_cond_broadcast( )**, or interrupted by a UNIX signal.

These functions atomically release the *mutex*, causing the calling thread to block on the condition variable *cond*.

Upon successful completion, the mutex is locked and owned by the calling thread.

pthread_cond_timedwait( ) is the same as **pthread_cond_wait( )**, except an error is returned if the system time equals or exceeds the time specified by *abstime* before the condition *cond* is signaled or broadcasted, or if the absolute time specified by *abstime* has already passed at the time of the call. When timeouts occur, **pthread_cond_timedwait( )** releases and reacquires the mutex referenced by *mutex*.

When using condition variables, there is always a boolean predicate involving shared variables related to each condition wait that is true, if the thread should proceed. Since the return from **pthread_cond_wait( )** or **pthread_cond_timedwait( )** does not indicate anything about the value of this predicate, the predicate should be reevaluated on return. Unwanted wakeups from **pthread_cond_wait( )** or **pthread_cond_timedwait( )** may occur.

The functions **pthread_cond_wait( )** and **pthread_cond_timedwait( )** are cancellation points. If a cancellation request is acted upon while in a condition wait when the cancellation enable state of a thread is set to **PTHREAD_CANCEL_DEFERRED**, the mutex will be reacquired before calling the first cancellation cleanup handler. In other words, the thread is unblocked, allowed to execute up to the point of returning from the call to **pthread_cond_wait( )** or **pthread_cond_timedwait( )**, but then notices the cancellation request and, instead of returning to the caller of **pthread_cond_wait( )** or **pthread_cond_timedwait( )**, it starts the thread cancellation activities including cancellation cleanup handlers.

A thread that is unblocked because it was canceled while blocked in a call to **pthread_cond_wait( )** or **pthread_cond_timedwait( )** does not awaken anyone else asleep on the condition.

Solaris Wait

cond_wait( ) atomically releases the mutex pointed to by *mp* and causes the calling thread to block on the condition variable pointed to by *cvp*. The blocked thread may be awakened by **cond_signal( )**, **cond_broadcast( )**, or when interrupted by delivery of a UNIX signal or a **fork( )**.

cond_wait( ) and **cond_timedwait( )** always return with the mutex locked and owned by the calling thread even when returning an error.

Condition Signaling

A condition signal allows a thread to unblock the next thread waiting on the condition variable, whereas, a condition broadcast allows a thread to unblock all threads waiting on the condition variable.

POSIX Signal and Broadcast

pthread_cond_signal( ) and **pthread_cond_broadcast( )** unblock threads blocked on a condition variable.

pthread_cond_signal( ) unblocks at least one thread blocked on the specified condition variable *cond*, if any threads are blocked on *cond*.

pthread_cond_broadcast( ) unblocks all threads blocked on the condition variable *cond*.

pthread_cond_signal( ) and **pthread_cond_broadcast( )** have no effect if there are no threads blocked on *cond*.

pthread_cond_signal( ) or **pthread_cond_broadcast( )** may be called by a thread regardless of whether it owns the mutex which 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 should be locked by the thread calling **pthread_cond_signal( )** or **pthread_cond_broadcast( )**.

Solaris Signal and Broadcast

cond_signal( ) unblocks one thread that is blocked on the condition variable pointed to by *cvp*.

cond_broadcast( ) unblocks all threads that are blocked on the condition variable pointed to by *cvp*.

If no threads are blocked on the condition variable, then **cond_signal( )** and **cond_broadcast( )** have no effect.

Both functions should be called under the protection of the same mutex that is used with the condition variable being signaled. Otherwise, the condition variable may be signaled between the test of the associated condition and blocking in **cond_wait( )**. This can cause an infinite wait.

Destroy

The condition destroy functions destroy any state, but not the space, associated with the condition variable.

POSIX Destroy

pthread_cond_destroy( ) destroys the condition variable specified by *cond*. The space for destroying the condition variable is not freed.

Solaris Destroy

cond_destroy( ) destroys any state associated with the condition variable pointed to by *cvp*. The space for storing the condition variable is not freed.

RETURN VALUES

0 is returned when any of these functions are successful. A non-zero value indicates an error, except **pthread_timedwait( )**, which returns **ETIME**.

ERRORS

These functions fail and return the corresponding value if any of the following conditions are detected:

EFAULT *cond*, *attr*, *cvp*, *arg*, *abstime*, or *mutex* point to an illegal address.

EINVAL Invalid argument.
For **pthread_cond_init( )**, the value specified for *attr* is invalid.

For **cond_init( )**, *type* is not a recognized type.

For **pthread_cond_timedwait( )** or **cond_timedwait( )**, the specified number of seconds, *abstime*, is greater than pgm_start_time + 50,000,000, where pgm_start_time is the start time of the application, or the number of nanoseconds is greater than or equal to 1,000,000,000.

cond_wait( ) or **cond_timedwait( )** fails and returns the corresponding value if any of the following conditions are detected:

EINTR The wait was interrupted by a signal or **fork( )**.

pthread_cond_timedwait( ) or **cond_timedwait( )** fails and returns the corresponding value if any of the following conditions are detected:

ETIME The time specified by *abstime* has passed.

SEE ALSO

mmap(2), **fork**(2), **signal**(3C), **mutex**(3T), **pthread_condattr_init**(3T)

NOTES

The only policy currently supported is **SCHED_OTHER**. In Solaris, under the **SCHED_OTHER** policy, there is no established order in which threads are unblocked.

If more than one thread is blocked on a condition variable, the order in which threads are unblocked is determined by the scheduling policy. When each thread, unblocked as a result of a **pthread_cond_signal( )** or **pthread_cond_broadcast( )**, returns from its call to

pthread_cond_wait( ) or **pthread_cond_timedwait( )**, the thread owns the mutex with which it called **pthread_cond_wait( )** or **pthread_cond_timedwait( )**. The thread(s) that are unblocked compete for the mutex according to the scheduling policy, and as if each had called **pthread_mutex_lock**(3T).

When **cond_wait( )** returns the value of the condition is indeterminate and must be reevaluated.

cond_timedwait( ) is similar to **cond_wait( )**, except that the calling thread will not wait for the condition to become true past the absolute time specified by *abstime*. Note that **cond_timedwait( )** may continue to block as it trys to reacquire the mutex pointed to by *mp*, which may be locked by another thread. If *abstime* then **cond_timedwait( )** returns because of a timeout, it returns the error code **ETIME .**