NAME

condvar, cv_init, cv_destroy, cv_wait, cv_signal, cv_broadcast, cv_wait_sig, cv_timedwait, cv_timedwait_sig – condition variable routines

SYNOPSIS

#include <sys/ksynch.h>

void cv_init(kcondvar_t **cvp*, **char** **name*, **kcv_type_t** *type*, **void** **arg***);**

void cv_destroy(kcondvar_t **cvp***);**

void cv_wait(kcondvar_t **cvp*, **kmutex_t** **mp***);**

void cv_signal(kcondvar_t **cvp***);**

void cv_broadcast(kcondvar_t **cvp***);**

int cv_wait_sig(kcondvar_t **cvp*, **kmutex_t** **mp***);**

int cv_timedwait(kcondvar_t **cvp*, **kmutex_t** **mp*, **long** *timeout***);**

int cv_timedwait_sig(kcondvar_t **cvp*, **kmutex_t** **mp*, **long** *timeout***);**

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

ARGUMENTS

cvp A pointer to an abstract data type **kcondvar_t**.

mp A pointer to a mutual exclusion lock (**kmutex_t**), initialized by **mutex_init**(9F) and held by the caller.

name A name for the condition variable, used in statistics and debugging.

type The constant **CV_DRIVER**.

arg A type-specific argument, drivers should pass arg as **NULL**.

timeout A time, in absolute ticks since boot, when **cv_timedwait( )** or **cv_timedwait_sig( )** should return.

DESCRIPTION

Condition variables are a standard form of thread synchronization. They are designed to be used with mutual exclusion locks (mutexes). The associated mutex is used to ensure that a condition can be checked atomically and that the thread can block on the associated condition variable without missing either a change to the condition or a signal that the condition has changed. Condition variables must be initialized by calling **cv_init( ),** and must be deallocated by calling **cv_destroy( )**.

The usual use of condition variables is to check a condition (for example, device state, data structure reference count, etc.) while holding a mutex which keeps other threads from changing the condition. If the condition is such that the thread should block, **cv_wait( )** is called with a related condition variable and the mutex. At some later point in time, another thread would acquire the mutex, set the condition such that the previous thread can be unblocked, unblock the previous thread with **cv_signal( )** or **cv_broadcast( )**, and then release the mutex.

cv_wait( ) suspends the calling thread and exits the mutex atomically so that another thread which holds the mutex cannot signal on the condition variable until the blocking thread is blocked. Before returning, the mutex is reacquired.

cv_signal( ) signals the condition and wakes one blocked thread. All blocked threads can be unblocked by calling **cv_broadcast( )**. You must aquire the mutex passed into **cv_wait( )** before calling **cv_signal( )** or **cv_broadcast( )**.

The function **cv_wait_sig( )** is similar to **cv_wait( )** but returns **0** if a signal (for example, by **kill**(2)) is sent to the thread. In any case, the mutex is reacquired before returning.

The function **cv_timedwait( )** is similar to **cv_wait( )**, except that it returns −**1** without the condition being signaled after the timeout time has been reached.

The function **cv_timedwait_sig( )** is similar to **cv_timedwait( )**, and **cv_wait_sig( )**, except that it returns −**1** without the condition being signaled after the timeout time has been reached, or **0** if a signal (for example, by **kill**(2)) is sent to the thread.

For both **cv_timedwait( )** and **cv_timedwait_sig( )**, time is in absolute clock ticks since the last system reboot. The current time may be found by calling **drv_getparm**(9F) with the argument **LBOLT**.

RETURN VALUES

0 For **cv_wait_sig( )** and **cv_timedwait_sig( )** indicates that the condition was not necessarily signaled and the function returned because a signal (as in **kill**(2)) was pending.

-1 For **cv_timedwait( )** and **cv_timedwait_sig( )** indicates that the condition was not necessarily signaled and the function returned because the timeout time was reached.

> 0 For **cv_wait_sig( )**, **cv_timedwait( )** or **cv_timedwait_sig( )** indicates that the condition was met and the function returned due to a call to **cv_signal( )** or **cv_broadcast( )**.

CONTEXT

These functions can be called from user, kernel or interrupt context. In most cases, however, **cv_wait( )**, **cv_timedwait( )**, **cv_wait_sig( )**, and **cv_timedwait_sig( )** should not be called from interrupt context, and cannot be called from a high-level interrupt context.

If **cv_wait( )**, **cv_timedwait( )**, **cv_wait_sig( )**, or **cv_timedwait_sig( )** are used from interrupt context, lower-priority interrupts will not be serviced during the wait. This means that if the thread that will eventually perform the wakeup becomes blocked on anything that requires the lower-priority interrupt, the system will hang.

For example, the thread that will perform the wakeup may need to first allocate memory. This memory allocation may require waiting for paging I/O to complete, which may require a lower-priority disk or network interrupt to be serviced. In general, situations like this are hard to predict, so it is advisable to avoid waiting on condition variables or semaphores in an interrupt context.

EXAMPLES

Here the condition being waited for is a flag value in a driver's unit structure. The condition variable is also in the unit structure, and the flag word is protected by a mutex in the unit structure.

```
mutex_enter(&un->un_lock);
while (un->un_flag & UNIT_BUSY)
        cv_wait(&un->un_cv, &un->un_lock);
```

```
un->un_flag |= UNIT_BUSY;
mutex_exit(&un->un_lock);
```

At some later point in time, another thread would execute the following to unblock any threads blocked by the above code.

```
mutex_enter(&un->un_lock);
un->un_flag &= ~UNIT_BUSY;
cv_broadcast(&un->un_cv);
mutex_exit(&un->un_lock);
```

SEE ALSO

kill(2), **drv_getparm**(9F), **mutex**(9F), **mutex_init**(9F)

Writing Device Drivers