NAME

semaphore, sema_init, sema_destroy, sema_wait, sema_trywait, sema_post – semaphores

SYNOPSIS

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

#include <synch.h>

int sema_init(sema_t **sp*, **unsigned int** *count*, **int** *type*, **void** * *arg***);**

int sema_destroy(sema_t **sp***);**

int sema_wait(sema_t **sp***);**

int sema_trywait(sema_t **sp***);**

int sema_post(sema_t **sp***);**

MT-LEVEL

MT-Safe

sema_post( ) is Async-Signal-Safe

DESCRIPTION

A semaphore is a non-negative integer count and is generally used to coordinate access to resources. The initial semaphore count is set to the number of free resources, then threads slowly increment and decrement the count as resources are added and removed. If the semaphore count drops to zero, which means no available resources, threads attempting to decrement the semaphore will block until the count is greater than zero.

Semaphores can synchronize threads in this process and other processes if they are allocated in writable memory and shared among the cooperating processes (see **mmap**(2)), and have been initialized for this purpose.

Semaphores must be initialized before use; semaphores pointed to by *sp* to *count* are initialized by **sema_init( )**. *type* can assign several different types of behavior to a semaphore. No current type uses *arg* although it may be used in the future.

type may be one of the following:

USYNC_PROCESS The semaphore can synchronize threads in this process and other processes. Initializing the semaphore should be done by only one process. *arg* is ignored.

USYNC_THREAD The semaphore can synchronize threads only in this process. *arg* is ignored.

A semaphore must not be simultaneously initialized by multiple threads, nor re-initialized while in use by other threads.

Default semaphore initialization (intra-process):

sema_t sp;

sema_init(&sp, NULL, NULL);

OR

sema_init(&sp, USYNC_THREAD, NULL);

OR

```
sema_t  sp  =  DEFAULTSEMA;
```

Customized semaphore initialization (inter-process):

```
sema_init(&sp, USYNC_PROCESS, NULL);
```

sema_destroy( ) destroys any state related to the semaphore pointed to by *sp*. The semaphore storage space is not released.

sema_wait( ) blocks the calling thread until the semaphore count pointed to by *sp* is greater than zero, and then it atomically decrements the count.

sema_trywait( ) atomically decrements the semaphore count pointed to by *sp*, if the count is greater than zero; otherwise, it returns an error.

sema_post( ) atomically increments the semaphore count pointed to by *sp*. If there are any threads blocked on the semaphore, one will be unblocked.

The semaphore functionality described on this man page is for the Solaris threads implementation. For the POSIX-compliant semaphore interface documentation, see **sem_open**(3R), **sem_init**(3R), **sem_wait**(3R), **sem_post**(3R), **sem_getvalue**(3R), **sem_unlink**(3R), **sem_close**(3R), **sem_destroy**(3R)).

RETURN VALUES

Upon successful completion, **0** is returned; otherwise, a non-zero value indicates an error.

ERRORS

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

EINVAL Invalid argument.

EFAULT *sp* or *arg* points to an illegal address.

sema_wait( ) fails and returns the corresponding value if any of the following conditions are detected:

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

sema_trywait( ) fails and returns the corresponding value if any of the following conditions are detected:

EBUSY The semaphore pointed to by *sp* has a zero count.

SEE ALSO

mmap(2), **sem_open**(3R), **sem_init**(3R), **sem_wait**(3R), **sem_post**(3R), **sem_getvalue**(3R), **sem_unlink**(3R), **sem_close**(3R), **sem_destroy**(3R)

EXAMPLES

The customer waiting-line in a bank is analogous to the synchronization scheme of a semaphore using **sema_wait( )** and **sema_trywait( ):**

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

#include <errno.h>
#define TELLERS 10
sema_t        tellers;     /* semaphore */
int   banking_hours(), deposit_withdrawal;
void  *customer(), do_business(), skip_banking_today();
```

```
...

sema_init(&tellers, TELLERS, USYNC_THREAD, NULL);
        /* 10 tellers available */
while(banking_hours())
        pthread_create(NULL, NULL, customer, deposit_withdrawal);
...

void *
customer(int deposit_withdrawal)
{
        int this_customer, in_a_hurry = 50;
        this_customer = rand() % 100;

        if (this_customer == in_a_hurry)  {
                if (sema_trywait(&tellers) != 0)
                      if (errno == EAGAIN) { /* no teller available */
                                   skip_banking_today(this_customer);
                                   return;
                      } /* else go immediately to available teller & decrement tellers */
        }
        else
                sema_wait(&tellers); /* wait for next teller, then proceed,
                                        and decrement tellers */

        do_business(deposit_withdrawal);
        sema_post(&tellers); /* increment tellers;
                         this_customer's teller
                         is now available */
}
```

NOTES

These interfaces are also available via:

#include <thread.h>

If multiple threads are waiting for a semaphore, by default, there is no defined order of unblocking.